Openwave WAP Push Library, Java Edition

Developer's Guide

Release 1.0

Openwave Systems Inc.
1400 Seaport Boulevard
Redwood City, CA 94063 U.S.A.
http://www.openwave.com

Part Number LPDJ-10-008

Legal Notice

For technical information on Openwave products, go to

`http://developer.openwave.com`

Please send comments about this book or corrections to

`doc-comments@openwave.com`

Contents

About This Book

This book provides instructions for using the Openwave WAP Push Library, Java Edition 1.0 to build Push Submissions for Openwave Push Proxy Gateway (PPG) servers.

IMPORTANT Check the Openwave Developer web site (`http://developer.openwave.com`) for a list of publicly available wireless devices and their browser versions. For unannounced products and services, contact your carrier to determine the Openwave Mobile Browser releases that you need to support.

Openwave SDK

The Openwave software development kit (SDK) is a tool for developing, debugging, and maintaining your wireless programs. Using the SDK, you can test your code from your local disk, through your own server, or through a Mobile Access Gateway. The tools in the SDK include an editor, an output window that lists transaction information, an HTTP window that displays source code, and the ability to view history, cookies, and variables.

In order to use the SDK, you must download and install the correct release. SDK releases support the following browsers:

Openwave SDK Version 5	Mobile Browser, WAP Edition 5.0 or later
UP.SDK 4.1	Mobile Browser 4.1 or later
UP.SDK 4.0	Mobile Browser 4.0 or later
UP.SDK 3.2 for WML1.1	Mobile Browser 3.1 or later

Audience and Prerequisites

This book is intended for developers who are creating wireless push services for mobile browser devices that are accessing an Openwave Mobile Access Gateway.

To use this book profitably, you must have the following background:

- A general understanding of the Internet and the World Wide Web
- Good working knowledge of HTML, WML, and XML
- Good working knowledge of the Java programming language and object-oriented programming
- Good working knowledge of the Push Access Protocol (PAP)
- A thorough understanding of the WAP Push Access Protocol and related specifications:
    - *WAP Push Message*
    - *WAP Push Proxy Gateway Service*
    - *WAP Push Access Protocol*
    - *WAP Push Architectural Overview*
    - *WAP Service Indication*
    - *WAP Service Loading*
    - *WAG UAPROF*
    - *WAP Cache Operation*
    - *WAP Push OTA Protocol*
    - *Wireless Datagram Protocol Specification*

All of these specification documents and all updates to them are available for free download from the Wireless Application Protocol Forum at `http://www.wapforum.org`. Visit this site regularly to make sure that you have all of the latest specification documents and updates.

Style and Typographical Conventions

The term *mobile browser device* refers to all Openwave Mobile Access Gateway-enabled mobile devices, including wireless phones, personal digital assistants, and two-way pagers. The term refers to both the hardware and the Openwave Mobile Browser software installed on it.

In this documentation, all illustrations and examples refer to a generic mobile browser device with the following characteristics:

- A 4 x 20 character display. Note that the Openwave Mobile Browser always reserves one line for function key labels and status.
- `accept`, `prev`, and `options` function keys. The actual labels and locations of these keys vary from one device to another.

  IMPORTANT Keep in mind that the display area and key arrangements on real mobile browser devices vary considerably. Some mobile browser devices reverse the location of the `accept` and `options` keys relative to the illustrations in this documentation. Others have fewer or no function keys and use different mechanisms for implementing the `accept`, `prev`, and `options` actions, such as a jog shuttle or other user-interface gestures.

This manual uses different fonts to represent information:

- `Text that appears like this` identifies code elements such as method and field names.
- *`Text that appears like this`* identifies parameter names in method declarations.

Code Examples

Omitted code is indicated with ellipses. For instance, the ellipses in the following example indicate that additional code exists in this WAP Push Library code block:

```
try {
    ppgURL = new URL(ppgAddress);
    coURI  = new URL(cacheOpURI);
    Pusher ppg = new Pusher(ppgURL);
    CacheOperation co = new CacheOperation(coURI,
        CacheOperationType.cachedService);
...
}
```

Other Documentation

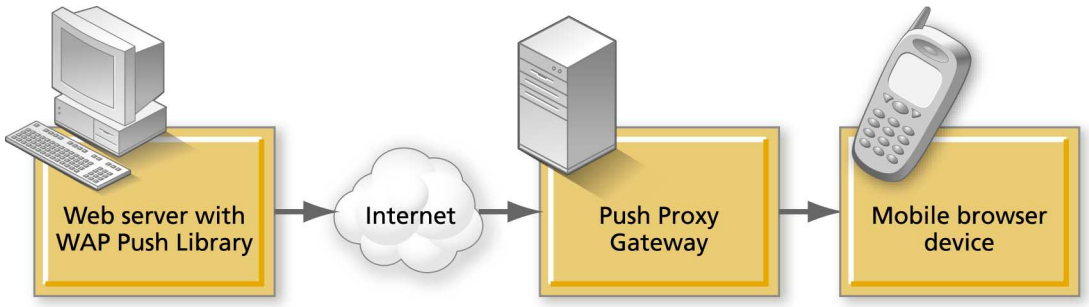
For a complete list of available documentation, see the Openwave Developer site at:

`http://developer.openwave.com`

1 Getting Started

The Openwave WAP Push Library facilitates implementation of the Push Access Protocol (PAP). With the libraries and tools provided, you can develop push applications and services.

Figure 1-1. Push configuration



As shown in Figure 1-1, a web server with the WAP Push Library initiates push operations. The Push Proxy Gateway (PPG) handles those operations and sends the appropriate information to the mobile browser device. In this case, the user has subscribed to a push service and the web server initiates the push operation; this is known as a *server-initiated push*.

It's also possible for the user to initiate a push operation. The web site can include a link that initiates a push operation when the user selects it. This is known as a *client-initiated push*.

Requirements

Refer to the Release Notes for a complete list of requirements for installing and using the WAP Push Library.

WAP Push Library Package Overview

When you install the WAP Push Library, the following components are available.

Libraries

wappush.jar

The `wappush.jar` file contains the WAP Push Library Java package and accompanying JavaDoc API documentation. This package provides all of the functionality you need to create push applications and services.

The WAP Push Library package includes the `SimplePush` class, which you can use to create simple PAP applications quickly. The `SimplePush` class consists of a series of methods that encapsulate essential WAP Push Library functionality, greatly reducing the amount and complexity of code required to develop PAP applications.

servlet.jar

The `servlet.jar` file is part of Tomcat. It is included with the WAP Push Library so you can compile the Travel demo source code using the `build.xml` file, without having to insert `servlet.jar` into your `classpath` environment variable.

NOTE The Tomcat software was developed by the Apache Software Foundation (`http://www.apache.org`).

xerces.jar

Xerces is a general purpose XML parser used by the WAP Push Library.

> **NOTE** The Xerces library software was developed by the Apache Software Foundation (`http://www.apache.org`).

Tools and Utilities

PushIT

PushIT is a Java tool that you can use to submit push operations and review the results. Using the PushIT GUI, you can quickly set the various parameters and content required by the Push Proxy Gateway and submit the operation. Complete source code can be found in the `<installroot>\examples\PushIT\src\java` directory.

For more information, see "Using the Push Initiator Tool" on page 120.

Examples

The WAP Push Library includes an example application and source code that show you how to use the WAP Push Library APIs.

Travel

The Travel example depicts a web site at which users can make travel arrangements. This application uses the WAP Push Library APIs to alert users to changes in flight plans.

Source Code

The WAP Push Library includes the source code for the Travel example and PushIT. You can use the source code as a reference showing how to use the WAP Push Library APIs. The source code is located in the following directories:

- PushIT: `<installroot>\examples\PushIT\src\java`
- TravelDemo: `<installroot>\examples\TravelDemo\src\java`

WAP Push Developer Resources

Openwave provides a variety of developer support resources online including a quick start tutorial, a WAP Push developer's page, and an application style guide. Visit the Openwave Developer site at:

`http://developer.openwave.com`

Click the WAP Push link under the Products and Technology heading to access all of these resources.

WAP Gateway for Openwave Developers

Use Openwave's Developer WAP Gateway with the Openwave SDK to test your applications in a fully compliant and secure WAP environment. Our Openwave Mobile Access Gateway Server is open to all registered Openwave developers. Visit the Openwave Developer site at:

```
http://developer.openwave.com
```

Click the Mobile Access Gateway Provisioning link for more information on creating and managing your subscriber accounts.

Installation and Configuration 2

Installing and Configuring

This chapter explains how to install and configure the WAP Push Library and associated utilities. See the *Release Notes* information on system requirements for the WAP Push Library.

To Install the WAP Push Library

- Download the `OPWVWAPPushLib.jar` file and open it. The file starts the installation process and by default places the WAP Push Library files in the `C:\Openwave\wappushjava` directory. Included in this installation are library `jar` files and examples.

To Configure Tomcat

- If you plan to use WML or WMLScript files, you need to add the following to the mime-mapping section of the `web.xml` file in the `WEB-INF` folder of the Tomcat context that hosts the WML or WMLScript files:

```
<mime-mapping>
    <extension>
        wmls
    </extension>
    <mime-type>
        text/vnd.wap.wmlscript
    </mime-type>
</mime-mapping>
<mime-mapping>
    <extension>
        wml
    </extension>
    <mime-type>
        text/vnd.wap.wml
    </mime-type>
</mime-mapping>
```

To Configure the Travel Example

The Travel example is designed to work with Tomcat. You must copy the example files into the Tomcat directory to make it work.

1. Copy `travel.war` to the `[tomcat_home]/webapps` directory.
2. If Tomcat is currently running, shut it down and then restart it.
3. Access the Travel example with the following URL: `http://[ipaddress]:[port]/travel/travel2.wml`.

   For example, `http://devgate2.openwave.com:8080/travel/travel2.wml`. It is important not to access the Travel example using a URL that begins with `http://localhost` because this disables some Travel example features.

Using the Example

Travel

Make sure that you have copied the Travel example files to the Tomcat directory, as described in "To Configure the Travel Example."

To run the Travel example, point your mobile browser to the `travel2.wml` file. When the WML deck loads, select option 2, Travel Status. Continue with the example until you reach the Notify Me of Changes option. Select this option to trigger a push operation that informs the user of any flight changes.

For more information on using the Travel example, see the Openwave Mobile Browser WAP Edition 5.0 *Graphical Browser Application Style Guide*.

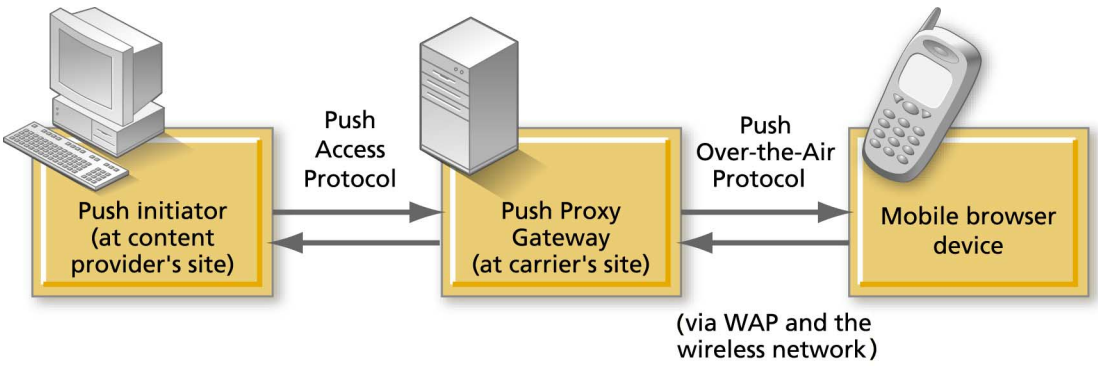
For an explanation of how the Travel example uses the WAP Push Library, see Chapter 6, "WAP Push Library Essentials."

3 Push Access Protocol Overview

About the Push Access Protocol

The Push Access Protocol (PAP) provides a means of sending content from the Internet to a mobile device, such as a WAP-enabled mobile phone. PAP achieves this by managing communications between the Internet server that sends the content, known in this context as a *push initiator* and the Push Proxy Gateway (PPG), the server that sends the content on to the target mobile device. The PPG acts as an intermediary, connecting the wired and wireless networks, which would otherwise have no way of communicating with each other because they use different communication protocols.

Figure 3-1. WAP push architecture



Because PAP is a *push* technology, it operates differently from other types of network protocols. Unlike HTTP on the Internet, in which the *client* initiates the transfer of content by requesting information, PAP allows a *server* to initiate the transfer of content to one or more client mobile devices.

PAP is built on Extensible Markup Language (XML) and transported using HTTP (Hypertext Transfer Protocol), although other transport protocols, such as SMTP (Simple Mail Transfer Protocol), may be available in the future.

A push request is a multipart document that can contain three entities:

- The control entity is an XML document containing delivery instructions destined for the Mobile Access Gateway PPG. The control entity is mandatory; it identifies the target mobile device and contains delivery instructions such as time delivery restrictions.
- The content entity contains content destined for the mobile device. A content entity is required only for a Push Submission. The content entity must be the second entity in the multipart document.
- The optional capabilities entity contains the mobile device capabilities for which the message was formatted. The push initiator may create this entity to indicate what it assumes the capabilities to be. The PPG also sends a capabilities entity in response to a Client Capabilities Query message.

These entities are bundled together as a Multi-Purpose Internet Mail Extensions (MIME) document, which is sent from the push initiator to the Mobile Access Gateway PPG using HTTP.

PAP Operations

PAP makes possible the following operations:

- **Push Submission** Delivers a push message from a push initiator to a mobile device. Because the Push Submission contains address, content, and optional capabilities entities, it is delivered as a multipart/related document. A Push Submission can deliver any of the content types described in the next section.
- **Status Query** The push initiator can request the current status of a Push Submission. All Status Query requests are delivered as XML documents.
- **Push Cancellation** Allows the push initiator to attempt to cancel a Push Submission. All Push Cancellation requests are delivered as XML documents.
- **Client Capabilities Query** The push initiator can query the PPG to retrieve the capabilities for a specific mobile device. All CCQ requests are delivered as XML documents. The PPG returns the CCQ information in a multipart/related document that the push initiator must parse to retrieve the relevant information.
- **Result Notification** The PPG informs the push initiator of the final outcome of the Push Submission; for example, confirmation of content delivery to the target mobile device. Result Notification is optional and occurs only if the push initiator requests it. All Result Notifications are delivered to the push initiator as XML documents when the final outcome of the Push Submission is known. Result Notifications are generally not available immediately after sending a Push Submission. See *WAP Push Access Protocol* for more information about Result Notification.

Push Submission Content Types

PAP can deliver the following types of content:

- **Service Indication (SI)** This content type consists of asynchronous notifications about new email, changes in selected stock prices, news headlines, advertisements, reminders, and so on. At its most basic, an SI contains a brief message and a URI specifying a service. The wireless client can either start the service immediately or store it for later action.
- **Service Loading (SL)** This content type allows a user agent on a client device to load and execute a service, specified by a URI, without user intervention. For example, an SL can notify a mobile device of a low prepaid service balance and require user action by loading a WML deck that presents a variety of options.
- **Cache Operation** This content type makes it possible to invalidate content objects in the wireless client's cache. All invalidated content objects must be reloaded from the server on which they originated the next time they are accessed. Use Cache Operation if an application cannot predict when content that it creates will expire. For example, use a Cache Operation to ensure that a mobile device always loads the most current contents from a mailbox application. A Cache Operation is an XML document that consists of a URI and one of the following operation types:
  - **Invalidate object** Invalidates the specific object that the URI identifies
  - **Invalidate service** Invalidates all objects that share the same URI prefix
- **Custom Content** This content type can consist of virtually any text, image, or other media.

Device Types

PAP is designed to meet the constraints of a wide range of small, narrowband devices. These devices are primarily characterized in four ways:

- **Display size** Smaller screen size and resolution. A small mobile device such as a phone may have only a few lines of textual display, each line containing 8 to 12 characters.
- **Input devices** A limited or special-purpose input device. A phone typically has a numeric keypad and a few additional function-specific keys. A more sophisticated device may have software-programmable buttons, but does not have a mouse or other pointing device.
- **Computational resources** Low-power CPU and small memory size, often limited by power constraints.
- **Narrowband network connectivity** Low bandwidth and high latency. Devices with 300 bps to 10 kbps network connections and 5 to 10 second round-trip latency are not uncommon.

This document uses the following terms to define broad classes of device functionality:

- **Phone** The typical display size ranges from 2 to 10 lines. Input is usually accomplished with a combination of a numeric keypad and a few additional function keys. Computational resources and network throughput are typically limited, especially when compared with more general-purpose computer equipment.
- **Personal digital assistant (PDA)** A device with a broader range of capabilities than a phone. In this document, PDA refers specifically to devices with additional display and input characteristics. A PDA display often supports resolution in the range of 160x100 pixels. A PDA may support a pointing device, handwriting recognition, and a variety of other advanced features.

How the WAP Push Library Implements PAP

The WAP Push Library is a Java package that encapsulates all of the XML and multipart/related message-building functionality into a series of classes. When you use the WAP Push Library, the implementation details of PAP are hidden. You don't need to know how to build a Push Submission or any other PAP operation to use the WAP Push Library. You do need to know how PAP works so that you can construct push messages and handle responses from the PPG, but you don't need to know the details of building or parsing the necessary XML or multipart/related documents.

System Configuration Requirements

Before you attempt to send content using the WAP Push Library, make sure that your corporate firewall allows you to send packets to the desired ports on the PPGs you are using. The mechanisms for client addressing and the ports that the PPG listens on are implementation specific. For specific information, check with the PPG administrator at the communication service provider you intend to use.

4 WAP Push Library Overview

WAP Push Library Basics

The WAP Push Library is a Java package composed of classes that encapsulate the most useful aspects of Push Access Protocol (PAP) communication, including Push Submission and response, content building, and exception handling. The WAP Push Library hides the details of PAP communication, building all required XML or multipart/related documents internally, forming the completed HTTP transaction, and embodying the transport layer security, either Secure Sockets Layer (SSL) or Transport Layer Security (TLS), used to protect a transmission.

The WAP Push Library dramatically simplifies the process of building PAP applications. As a developer, you do not need to know any of the details of the XML or multipart/related documents that make up a Push Submission. All of the elements needed to build a well-formed Push Submission are required parameters of the various WAP Push Library class constructors, making it impossible to forget any of the necessary elements. The WAP Push Library includes classes that parse all of the XML documents the PPG sends in response to a Push Submission, making it easy for your applications to provide detailed information to users.

This chapter provides an overview of the major WAP Push Library classes and how they correspond to specific PAP entities.

PAP operations

Table 4-1 lists the WAP Push Library classes that encapsulate PAP operations.

Table 4-1. PAP operations and WAP Push Library classes

PAP operation	WAP Push Library class
Push Submission	`PushMessage`
Status Query	`StatusQueryMessage`
Push Cancellation	`CancelMessage`
Client Capabilities Query	`CcqMessage`

When building a WAP Push Library application, use the class that corresponds to the desired PAP operation. For descriptions of the PAP operations, see "PAP Operations" on page 18.

Push Submission Content Types

As with PAP operations, a WAP Push Library class encapsulates each Push Submission content type. Table 4-2 lists the content types and the corresponding WAP Push Library classes.

Table 4-2. Push Submission content types and WAP Push Library classes

Content type	WAP Push Library class
Service Indication	`ServiceIndication`
Service Loading	`ServiceLoading`
Cache Operation	`CacheOperation`
Custom Content	`CustomContent`

For descriptions of the content types, see "Push Submission Content Types" on page 19.

NOTE Openwave is developing new Push Submission content types. Check the Openwave Developer web site (`http://developer.openwave.com`) for the latest information.

PPG and Client Addresses

All PAP operations require PPG and client addresses. The PPG address is a `java.net.URL` object, which the WAP Push Library encapsulates in the `Pusher` class. The PPG address specifies the application on the PPG that delivers a PAP operation to the wireless client, as shown in the following example:

```
http://devgate2.openwave.com:9002/pap
```

The `Pusher` class constructor renders a properly formed XML PPG address value.

The client address is simply a `java.lang.String` object that you pass to the constructor of the desired PAP operation class. Client addresses must be properly formatted using either the subscriber ID (SUB_ID) or phone number of the client, as shown in the following examples.

For the user `jdoe` provisioned on `devgate2.openwave.com`, the client address is rendered as:

```
jdoe_devgate2.openwave.com/TYPE=USER@ppg.openwave.com
```

For a user whose phone number is 123-456-7890, the client address is rendered as:

```
1234567890/TYPE=PLMN@ppg.openwave.com
```

The class constructor into which you pass the client address renders a properly formed XML address value for the desired PAP operation.

Extracting PPG and Client Addresses from PPG Headers

You can extract the PPG and client addresses from the HTTP headers returned with any request from the PPG. You can use this information dynamically in WAP Push Library applications or in other applications that are devoted to building lists of users who visit your WAP sites. The following HTTP headers returned from the PPG contain this information.

- `X-Up-Subno` The address of the client
- `X-Up-Uplink` The address of the Openwave MAG, if connected
- `X-Up-Wap-Push-Secure` Secure (HTTPS) PPG address
- `X-Up-Wap-Push-Unsecure` Nonsecure (standard HTTP) PPG address

Query these headers whenever you need to capture a PPG address. Doing so allows you, for example, to provide a sign-up service for push messages that can get the PPG address directly from the HTTP request headers rather than requiring the user to type in the desired PPG address.

You should use the `X-Up-Uplink` header only to determine if a connection is through an Openwave Mobile Access Gateway (MAG).

See “Travel Example” on page 65 for an example of how to use these headers to extract the desired information.

Secure Versus Nonsecure PPG Addresses

The PPG infrastructure supports both secure (HTTPS) and nonsecure (standard HTTP) connections. The WAP Push Library supports both connection types transparently. The only difference is the PPG address itself.

A nonsecure connection is ideally suited for initial application testing because it does not introduce any additional complexity. A secure connection is often desirable, but does impose some constraints in the form of server and client certificates. There are two issues to consider:

- The server certificate is not issued by a well-known certificate authority (CA). Server certificates from well-known CA do not generally pose any difficulties. Certificates from other CAs must be manually added to the list of trusted certificates on the client. This is generally true of PPG test installations. To add the server certificate to the list of trusted certificates on the client, follow these steps:

1. Copy the server certificate to the client machine
2. Save the server certificate to a file
3. Install the server certificate on the Java client

- The PPG is configured to require client (application) authentication, in which case only certain push initiators have access to secure connections on the PPG. In this case, the PPG is configured to request the client certificate during the SSL handshake process, something not normally required during HTTPS communication using a web browser. To satisfy the PPG request, the client must have an appropriate client certificate installed to access the PPG using a secure connection.

Multicasting

The WAP Push Library supports delivery of a single Push Submission to multiple client addresses. This capability, called *multicasting*, saves enormous time and effort whenever you want to deliver a general message to all or part of a subscriber base. Use the `PushMessage.addAddress` method to build the desired list of recipients for a Push Submission. The Push Submission can deliver any of the available content types as its payload.

Push Submission Identification

Every Push Submission, regardless of the payload type, must include a unique identifier. The identifier distinguishes one Push Submission from all others, a critical factor if you want to cancel or retrieve status information for a Push Submission. The `PushMessage`, `CancelMessage`, and `StatusQueryMessage` class constructors define this parameter as a `java.lang.String` object. Assign each Push Submission a unique identifier that includes the domain name of the push initiator, as in the following examples:

```
2903011435@www.openwave.com

www.openwave.com/2903011435
```

See *WAP Push Access Protocol* for more information and examples.

Specifying Push Submission Delivery Timing

You can use the WAP Push Library to specify the delivery timing for any Push Submission by setting a deliver-before or deliver-after time. You can set one or both of these values to ensure that a Push Submission is delivered at the correct time. The `PushMessage` class provides two methods that set the delivery timing: `setDeliverBeforeTimestamp` and `setDeliverAfterTimestamp`. See the WAP Push Library JavaDoc API documentation for more information.

PPG Response Message

The WAP Push Library `Pusher` class, which encapsulates the address of a PPG, includes a `send` method. As its name implies, the `send` method dispatches a Push Submission to the PPG. The `send` method returns the response message from the PPG encapsulated in an object of the class corresponding to the PAP operation response type.

Table 4-3 lists the response types and the corresponding WAP Push Library classes.

Table 4-3. PAP response types and WAP Push Library classes

PAP operation response	WAP Push Library class
Push Submission Response	`PushResponse`
Status Query Response	`StatusQueryResponse`
Push Cancellation Response	`CancelResponse`
Client Capabilities Query Response	`CcqResponse`

The PPG response message indicates the immediate outcome of a Push Submission. The response indicates only that the PPG accepted the message for processing or rejected it for any of several reasons. The PPG response does not indicate that the client received the submission. The final outcome of the message is indicated by the PAP Result Notification operation, which falls outside the scope of the WAP Push Library. For a description of the Result Notification operation, see “PAP Operations” on page 18.

Each WAP Push Library response class parses the XML response document from the PPG and provides methods that extract the desired response information, which can include both numeric codes and text messages. The numeric codes fall into the following ranges:

- **1xxx: Success** The action was successfully received, understood, and accepted
- **2xxx: Client Error** The request contains bad syntax or cannot be fulfilled
- **3xxx: Server Error** The server failed to fulfil an apparently valid request
- **4xxx: Service Failure** The service could not be performed. The operation may be retried
- **5xxx: Mobile Device Abort** The mobile device aborted the operation

Table 4-4 lists the response message codes and descriptions returned from the PPG.

Table 4-4. PAP response codes and descriptions

Code	Description
1000	OK
1001	Accepted for Processing
2000	Bad Request
2001	Forbidden
2002	Address Error
2003	Address Not Found
2004	Push ID Not Found
2005	Capabilities Mismatch
2006	Required Capabilities Not Supported
2007	Duplicate Push ID
3000	Internal Server Error
3001	Not Implemented
3002	Version Not Supported
3003	Not Possible
3004	Capability Matching Not Supported
3005	Multiple Addresses Not Supported
3006	Transformation Failure
3007	Specified Delivery Method Not Possible
3008	Capabilities Not Available
3009	Required Network Not Available
3010	Required Bearer Not Available
4000	Service Failure
4001	Service Unavailable
5xxx	Mobile Client Aborted

In addition to the standard numeric code and text message, the Push Status Query Response also contains an attribute that indicates the status of the specified Push Submission. Nine message states are currently specified:

- `aborted` The addressee aborted the message.
- `cancelled` A Push Cancellation successfully canceled the message.
- `delivered` The PPG successfully delivered the message to the addressee.
- `expired` The message reached the maximum age allowed by PPG policy or could not be delivered by the time specified in the Push Submission.
- `pending` The PPG accepted the message and is in the process of delivering it.
- `rejected` The addressee rejected the message.
- `timeout` The delivery process timed out on the PPG.
- `undeliverable` A problem prevented the message from being delivered. Call the `StatusQueryResult.getCode` and `StatusQueryResult.getDesc` methods to retrieve the code and text message returned from the PPG.
- `unknown` The PPG has no information about the status of the message.

Exception Handling

The WAP Push Library uses the `WapPushException` class to throw an exception if a parameter is missing, out of range, or specified improperly, or if a response from the PPG is missing one or more required elements. All `public` constructors and methods that require one or more parameters throw this exception, as do all of the message response classes. Your WAP Push Library applications should always catch and handle these exceptions, which can be very helpful to you during debugging.

The only specific exception that the WAP Push Library generates internally occurs if a PAP message is garbled when the PPG receives it. In that case, the PPG returns the `<badmessage-response>` XML element with a message describing the cause of the exception and a fragment of the garbled submission. All of the WAP Push Library response classes are designed to instantiate a `BadMessageException` object and return the exception message in the response object. Your WAP Push Library applications should always catch and handle this exception.

All other exceptions are handled by the various Java classes that your application imports. At a minimum, all WAP Push Library applications should import the following Java exception classes and handle any exceptions they throw:

```
java.io.IOException
java.io.FileNotFoundException
java.net.MalformedURLException
```

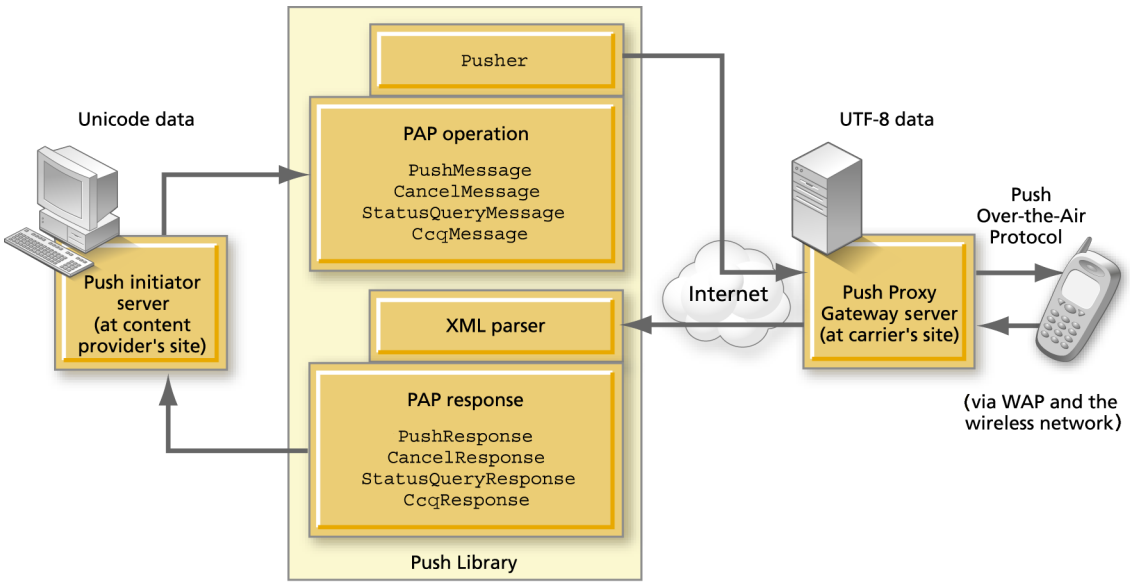
For complete exception-handling information, refer to the manuals for your Java development environment.

WAP Push Library Architecture Overview

Applications built on the WAP Push Library construct PAP operations using the appropriate class. Each PAP operation class provides one or more mutator (set) methods that allow the caller to specify optional PAP values for each operation type. The `Pusher` class provides a `send` method, which delivers the PAP message to the specified PPG. The PPG then returns a corresponding response object, which the WAP Push Library application can query to determine specific information about the immediate status of the PAP operation.

A WAP Push Library application running on the push initiator instantiates the appropriate PAP operation subclass and sends it to the PPG using an instance of the `Pusher` object. The PPG receives the PAP message, processes it, and returns an XML document. The WAP Push Library XML Parser converts the XML response to the proper PAP response object and returns it to the push initiator. Figure 4-1 illustrates the WAP Push Library architecture.

Figure 4-1. WAP Push Library architecture



The `Pusher` object encapsulates the transport layer protocol, which is limited to HTTP. Instances of the `Pusher` object format the various message types into properly formed HTTP transactions and handle any specified transport layer security (SSL or TLS). All WAP Push Library communications with the PPG are synchronous.

The WAP Push Library application running on the push initiator can instantiate any number of `Pusher` objects, each of which represents a different PPG. The push initiator can therefore send a single PAP operation to any or all of the PPGs for which a `Pusher` object exists. The push initiator can also send the same PAP operation to the same PPG as many times as necessary.

If the PAP operation is an instance of the `PushMessage` object, the WAP Push Library application must first build any one of the possible payload types (Cache Operation, Custom Content, Service Indication, or Service Loading), using the corresponding payload class. A `MimeEntity` object instance then encapsulates the payload object into the PAP operation transmission.

Finally, the WAP Push Library handles all conversion between the Java standard UNICODE encoding and the PPG standard UTF-8 encoding of HTTP transactions.

Sending a Push Submission

To send a Push Submission, follow the general steps outlined in this section. For example code with full explanations, see Chapter 6, "WAP Push Library Essentials."

1. Get the subscriber IDs or phone numbers of the subscribers to whom you want to send a Push Submission. You can send a single Push Submission to as many subscribers as desired using the `PushMessage.addAddress` method. There are two general ways to get the desired subscriber IDs or phone numbers:

- **Subscription list** Users have subscribed to your service and have provided their subscriber IDs or phone numbers.
- **HTTP headers** Check the HTTP headers for requests from wireless devices. When the Mobile Access Gateway Server makes an HTTP request to a WML service, it adds headers that provide information about the subscriber, the mobile browser, and the Mobile Access Gateway Server used to deliver the requests. The web server converts these headers into environment variables, which you can retrieve using facilities such as the C function, `getenv`, or the special Perl array, `@ENV`. The `whoami.cgi` example CGI script included with the Openwave SDK provides a simple example of how to retrieve these environment variables. The header that contains subscriber IDs is `HTTP_X_UP_SUBNO`. An example of a Java method that retrieves information from the HTTP headers is:

```
HttpServletRequest request;
String subscriber_address = request.getHeader("X-Up-Subno") +
    "/TYPE=USER@ppg.openwave.com";
```

2. Get the PPG address of the carrier that you want to use to send a Push Submission.

- **Carrier Contract** Contact the carrier whose PPG you want to use.
- **HTTP headers** Check the HTTP headers for requests from wireless devices as described above. The following headers are available:
  `HTTP_X_UP_UPLINK` Contains the MAG address
  `HTTP_X_UP_WAP_PUSH_UNSECURE` Contains the nonsecure (standard HTTP) PPG address
  `HTTP_X_UP_WAP_PUSH_SECURE` Contains the secure (HTTPS) PPG address
  An example of a Java method that retrieves PPG address information from the HTTP headers is:

```
HttpServletRequest request;
String PPGaddress = request.getHeader("X-Up-Wap-Push-Unsecure");
```

3. Make sure that your corporate firewall allows you to send Push Submissions.

Because the PPG ports are implementation specific, check with the PPG administrator at the carrier you use to send Push Submissions for specific information.

4. Pass a unique push identification string and at least one subscriber address to an instance of the `PushMessage` class.

5. Assemble the desired content using one or more content-type classes: `CacheOperation`, `CustomContent`, `ServiceIndication`, or `ServiceLoading`.

6. Set any desired Push Submission options using `PushMessage` mutator (set) methods.

7. Pass the `PushMessage` and assembled content objects to the `MimeEntity` class.

8. Define the URL of the desired PPG, using the `Pusher` class.

9. Send the Push Submission by calling the `Pusher.send` method.

10. Examine the response object to determine specific information about the immediate status of the Push Submission.

11. Handle any exceptions.

12. Use the appropriate WAP Push Library class to check the status of a Push Submission, cancel a Push Submission, or initiate a Client Capabilities Query, respectively `StatusQueryMessage`, `CancelMessage`, and `CcqMessage`.

Sending a Service Indication using the WAP Push Library

This section describes how a WAP application pushes a Service Indication (SI), commonly called an “alert,” to a mobile device using Openwave WAP Push Library and the Openwave PPG.

An SI is a short text message that links to a URL on the application server. Applications push SIs whenever events occur that require user attention, for example, new email arrival, traffic or weather alerts, stock trade results, or field service dispatches. When the user selects the SI, the WAP browser fetches the associated URL, which allows applications to deliver more detailed or time-sensitive content with the short alert message. In most cases, the device pops up the alert message or beeps the user when the SI is received. The SI is persistently stored in the user's inbox so that the user can select it at any time before deleting it from the inbox.

Extracting the Addresses and Sending the Push

Here is a code fragment from the Travel service demo application supplied with the WAP Push Library. The application uses the Openwave WAP Push Library `SimplePush` class to send the SI to the user's mobile device. When the user first visits the travel service using the WAP browser, the application extracts the client (device) address and PPG address from the HTTP request header. The application should save this data for later use in its user profile database.

```
// Get the client and push proxy gateway address from the
// HTTP request header
subID = request.getHeader("X-Up-Subno");
ppgString = "http://" + request.getHeader("X-Up-Wap-Push-Unsecure");
clientAddress = subID + "/TYPE=USER@ppg.phone.com";
...
// Initialize the alert data and submit the push request
private String  alertTitle = "Travel Service";
private String  alertURL =
    "http://flights.openwave.com/status.wml?ID=4393312";
try
{
    pushResponse = sp.pushServiceIndication(new String[] {clientAddress},
        "Travel Service: Flight Status Update",
        alertURL,
        ServiceIndicationAction.signalHigh);
}
```

The Push Submission

The WAP Push Library generates a PAP-compliant XML multi-part document and posts the document to the PPG on behalf of the application. The document complexities, including encoding, headers, spacing, boundary IDs, and so forth, are abstracted by the WAP Push Library API, so application developers do not have to worry about the details:

```
--plibbgtTtrewZjFtpqoK

Content-type: application/xml; charset=UTF-8

<?xml version="1.0"?>
<!DOCTYPE pap PUBLIC "-//WAPFORUM//DTD PAP 1.0//EN"
    "http://www.wapforum.org/DTD/pap_1.0.dtd">
<pap>
    <push-message push-id="60767/949/ServiceInd Example">
        <address
            address-value=
            "WAPPUSH=jdoe_devgate.openwave.com/
            TYPE=USER@ppg.openwave.com"/>
        <quality-of-service priority="high"
            delivery-method="unconfirmed"/>
    </push-message>
</pap>

--plibbgtTtrewZjFtpqoK

Content-type: text/vnd.wap.si; charset=UTF-8

<?xml version="1.0"?>
<!DOCTYPE si PUBLIC "-//WAPFORUM//DTD SI 1.0//EN"
    "http://www.wapforum.org/DTD/si.dtd">
<si>
    <indication href="http://flights.openwave.com/status.wml?ID=4393312"
        action="signal-high">
        Travel Service: Flight Status Update
    </indication>
</si>

--plibbgtTtrewZjFtpqoK
```

The SI User Interface

The SI message is presented to the user as soon as the PPG delivers the SI to the mobile device as shown in Figure 4-2 on page 34. When the user selects View the alert URL is fetched from the application server and the resulting WML content is presented. The message is saved in the alert inbox, so the user can select Skip and view it later. Each time the user selects the SI, the application can deliver an updated version of the content, so the user always sees the most current information available.

Figure 4-2. Travel demo SI user display

SimplePush Class Essentials 5

This chapter introduces the use of the `SimplePush` class, which you can use to create simple Push Application Protocol (PAP) applications quickly. The `SimplePush` class consists of a series of methods that encapsulate essential WAP Push Library functionality, greatly reducing the amount and complexity of code required to develop PAP applications.

Chapter 6, "WAP Push Library Essentials,"provides details and examples that help you to create more complex push applications and to have more control over Push Operations.

Required Import Statements

To develop any application using the WAP Push Library, your code must include the following statement:

```
import com.openwave.wappush.*;
```

You can import specific WAP Push Library classes instead of the entire package, if desired.

In addition to the WAP Push Library, your applications may require functionality of the following Java classes:

```
import java.io.IOException;
import java.io.FileNotFoundException;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Enumeration;
```

You may also need to import other Java classes, particularly the Abstract Windowing Toolkit (AWT), Swing, and event-handling classes, to support the user interface elements that your application implements.

PPG and Client Addresses

The examples in this chapter depict all PPG and client addresses as hard-coded constants, in part to show the proper format and in part to make the examples easy to understand. A much better approach would be to extract the PPG and client addresses from the HTTP request headers to the PPG or from a user interface. For more information on the former approach, see "Extracting PPG and Client Addresses from PPG Headers" on page 23, "Sending a Push Submission" on page 30, and "Travel Example" on page 65.

SimplePush Class Basics

All applications that use the `SimplePush` class should perform these operations:

- Instantiate a `SimplePush` object that contains the PPG address, the name of your program, and the push identification suffix to use. The latter item helps ensure that each push identifier is unique
- Send the desired PAP operation using the appropriate `SimplePush` method
- Read the response returned from the PPG
- Query the response for any desired information
- Handle all exceptions

The following sections provide examples of basic `SimplePush` applications.

Service Indication Payload Example

This example demonstrates how to send a Push Submission with a Service Indication (SI) payload. This content type sends notifications to addressees in an asynchronous manner. These notifications may, for example, be about new email, changes in stock price, news headlines, advertising, or reminders of various types.

In its most basic form, an SI contains a short message and a URI specifying a service. The message is presented to the addressee upon reception. The addressee has the option of starting the service indicated by the URI immediately or postponing the SI for later handling. If the addressee postpones the SI, the client device stores it.

What It Does

The `spServiceInd.java` file declares two constant fields that define the recipient address and the URL of the PPG. The public `spServiceInd` class encapsulates all of the program's functionality in three methods:

- The `printResults` method prints the results of the Push Submission.
- The `SubmitMsg` method instantiates the `SimplePush` object, sends the Push Submission, and handles any exceptions returned from the PPG.
- The `main` method calls the `SubmitMsg` method.

The exception handling demonstrated in this example is limited to displaying error messages on the console. It would be better, however, to build a more automated exception-handling and response mechanism into any applications that you write.

A complete code listing follows.

spServiceInd.java

```
/*
 * Title: SimplePush Service Indication Payload Example
 * Description: A basic Push Submission example using a Service
 * Indication payload
 */

import java.net.MalformedURLException;
import java.io.IOException;
import com.openwave.wappush.*;

public class spServiceInd {
    private final static String ppgAddress =
        "http://devgate2.openwave.com:9002/pap";
    private final static String[] clientAddress =
       {"jdoe_devgate2.openwave.com/TYPE=USER@ppg.openwave.com"};
    private final static String SvcIndURI =
        "http://devgate2.openwave.com/cgi-bin/mailbox.cgi";

    private static void printResults(PushResponse pushResponse)
        throws WapPushException, MalformedURLException, IOException {
        //Read the response to find out if the Push Submission succeded.
        //1001 = "Accepted for processing"
        if (pushResponse.getResultCode() == 1001) {
            try {
                String pushID = pushResponse.getPushID();
                SimplePush sp = new SimplePush(new
                    java.net.URL(ppgAddress),
                    "SampleApp", "/sampleapp");
                StatusQueryResponse queryResponse =
                    sp.queryStatus(pushID, null);
                StatusQueryResult queryResult =
                    queryResponse.getResult(0);
                System.out.println("Message status: " +
                    queryResult.getMessageState());
            }
                catch (WapPushException exception) {
                System.out.println("*** ERROR - WapPushException (" +
                    exception.getMessage() + ")");
            }
            catch  (MalformedURLException exception) {
                System.out.println("*** ERROR - MalformedURLException ("
                + exception.getMessage() + ")");
            }
            catch (IOException exception) {
                System.out.println("*** ERROR - IOException ("
                    + exception.getMessage() + ")");
            }
        }
        else
            System.out.println("Message failed");
    }//printResults
```

```
    public void SubmitMsg() throws WapPushException, IOException {
        try {
            //Instantiate a SimplePush object passing in the PPG URL,
            //product name, and PushID suffix, which ensures that the
            //PushID is unique.
            SimplePush sp = new SimplePush(new
                java.net.URL(ppgAddress),
                "SampleApp", "/sampleapp");

            //Send the Service Indication.
            PushResponse response =
                sp.pushServiceIndication(clientAddress,
                    "Mobile Mail: New message!", SvcIndURI,
                    ServiceIndicationAction.signalHigh);

            //Print the response from the PPG.
            printResults(response);
        }//try
        catch (WapPushException exception) {
            System.out.println("*** ERROR - WapPushException (" +
                exception.getMessage() + ")");
        }
        catch (IOException exception) {
            System.out.println("*** ERROR - IOException ("
                + exception.getMessage() + ")");
        }
    }//SubmitMsg()

    public static void main(String[] args) throws WapPushException,
        IOException {
        spServiceInd spsi = new spServiceInd();
        spsi.SubmitMsg();
    }//main
}//class spServiceInd
```

How It Works

This simple example declares a class, `spServiceInd`, in which the recipient address, the PPG URL, and the SI URI are declared as constants:

```
private final static String ppgAddress =
    "http://devgate2.openwave.com:9002/pap";
private final static String[] clientAddress =
    {"jdoe_devgate2.openwave.com/TYPE=USER@ppg.openwave.com"};
private final static String SvcIndURI =
    "http://devgate2.openwave.com/cgi-bin/mailbox.cgi";
```

Notice that the `clientAddress` field is a String array. This is necessary because the `pushCustomContent` method requires a String array for its *`addresses`* parameter.

The next block defines the private `printResults` method, which tests the response object returned from the PPG for information regarding the Push Submission. If the Push Submission succeeded, this method prints some of the pertinent response information.

The `SubmitMsg` method contains the code that builds and sends the Push Submission and prints the response from the PPG. The first line instantiates a `SimplePush` object, passing in the PPG URL, the product name, and the PushID suffix, which ensures that the push identifier is unique.

```
SimplePush sp = new SimplePush(new java.net.URL(ppgAddress), "SampleApp",
    "/sampleapp");
```

The second line sends the Push Submission using the `SimplePush` method that encapsulates an SI payload. The last line prints the response from the PPG.

```
PushResponse response =
    sp.pushServiceIndication(clientAddress,
        "Mobile Mail: New message!", SvcIndURI,
        ServiceIndicationAction.signalHigh);
printResults(response);
```

The preceding lines are enclosed in a `try...catch` block, allowing the `SubmitMsg` method to report and act upon any exceptions that might occur. In this example, the action taken in response to any exception is to print the contents of the exceptions to the console.

Finally, the `main` method instantiates an `spServiceInd` object and calls the `SubmitMsg` method:

```
spServiceInd spsi = new spServiceInd();
spsi.SubmitMsg();
```

Service Loading Payload Example

This example demonstrates how to send a Push Submission with a Service Loading (SL) payload. This content type causes a user agent on a mobile client to load and execute a service that, for example, can be in the form of a WML deck. The SL contains a URI specifying the service that the user agent is to load, with or without user intervention as appropriate.

This example uses some of the same constants and other code as the previous examples, but demonstrates another useful payload type.

What It Does

The `spServiceLoad.java` file declares three constant fields that define the recipient address, the URL of the PPG, and the SL URI. The public `spServiceInd` class encapsulates all of the program's functionality in three methods:

- The `printResults` method prints the results of the Push Submission.
- The `SubmitMsg` method instantiates the `SimplePush` object, sets Quality of Service attributes, sends the Push Submission, prints the pertinent information from the PPG response, and handles any exceptions returned from the PPG.
- The `main` method calls the `SubmitMsg` method.

The exception handling demonstrated in this example is limited to displaying error messages on the console. It would be better, however, to build a more automated exception-handling and response mechanism into any applications that you write.

A complete code listing follows.

spServiceLoad.java

```
/*
 * Title: SimplePush Service Loading Payload Example
 * Description: A basic Push Submission example using a Service Loading
 * payload
 */

import java.net.MalformedURLException;
import java.io.IOException;
import com.openwave.wappush.*;

public class spServiceLoad {
    private final static String ppgAddress =
        "http://devgate2.openwave.com:9002/pap";
    private final static String[] clientAddress =
       {"jdoe_devgate2.openwave.com/TYPE=USER@ppg.openwave.com"};
    static final String SvcLoadURI =
        "http://devgate2.openwave.com/cgi-bin/mailbox.wml";

    private static void printResults(PushResponse pushResponse)
        throws WapPushException, MalformedURLException, IOException {
        //Read the response to find out if the Push Submission succeded.
        //1001 = "Accepted for processing"
        if (pushResponse.getResultCode() == 1001) {
            try {
                String pushID = pushResponse.getPushID();
                SimplePush sp = new SimplePush(new
                    java.net.URL(ppgAddress),
                    "SampleApp", "/sampleapp");
                StatusQueryResponse queryResponse =
                    sp.queryStatus(pushID, null);
                StatusQueryResult queryResult =
                    queryResponse.getResult(0);
                System.out.println("Message status: " +
                    queryResult.getMessageState());
            }
                catch (WapPushException exception) {
                System.out.println("*** ERROR - WapPushException (" +
                    exception.getMessage() + ")");
            }
            catch  (MalformedURLException exception) {
                System.out.println("*** ERROR - MalformedURLException ("
                + exception.getMessage() + ")");
            }
            catch (IOException exception) {
                System.out.println("*** ERROR - IOException ("
                    + exception.getMessage() + ")");
            }
        }
        else
            System.out.println("Message failed");
    }//printResults
```

```
    public void SubmitMsg() throws WapPushException, IOException {
        try {
            //Instantiate a SimplePush object passing in the PPG URL,
            //product name, and PushID suffix, which ensures that the
            //PushID is unique.
            SimplePush sp = new SimplePush(new java.net.URL(ppgAddress),
                "SampleApp", "/sampleapp");

            //Instantiate a Quality of Service (QOS) object for the Push
            //message.
            QualityOfService sp_qos = new QualityOfService();

            //Set the desired QOS attributes.
            sp_qos.setDeliveryMethod(DeliveryMethod.confirmed);
            sp_qos.setPriority(DeliveryPriority.high);
            sp_qos.setNetwork("GSM");
            sp_qos.setNetworkRequired(true);
            sp_qos.setBearer("USSD");
            sp_qos.setBearerRequired(false);

            //Set the QOS for the push message.
            sp.setQualityOfService(sp_qos);

            //Send the Service Loading.
            PushResponse response =
                sp.pushServiceLoading(clientAddress, SvcLoadURI,
                    ServiceLoadingAction.executeHigh);

            //Print the response from the PPG.
            printResults(response);
        }//try
        catch (WapPushException exception) {
            System.out.println("*** ERROR - WapPushException (" +
                exception.getMessage() + ")");
        }
        catch (IOException exception) {
            System.out.println("*** ERROR - IOException ("
                + exception.getMessage() + ")");
        }
    }//SubmitMsg()

    public static void main(String[] args) throws WapPushException,
        IOException {
        spServiceLoad spsl = new spServiceLoad();
        spsl.SubmitMsg();
    }//main
}//class spServiceLoad
```

How It Works

This simple example declares a class, `spServiceLoad`, in which the recipient address, the PPG URL, and the Service Loading URI are declared as constants:

```
private final static String ppgAddress =
    "http://devgate2.openwave.com:9002/pap";
private final static String[] clientAddress =
    {"jdoe_devgate2.openwave.com/TYPE=USER@ppg.openwave.com"};
static final String SvcLoadURI =
    "http://devgate2.openwave.com/cgi-bin/mailbox.wml";
```

Notice that the `clientAddress` field is a String array. This is necessary because the `pushServiceLoading` method requires a String array for its *`addresses`* parameter.

The next block defines the private `printResults` method, which tests the response object returned from the PPG for information regarding the Push Submission. If the Push Submission succeeded, this method prints some of the pertinent response information.

The `SubmitMsg` method contains the code that builds and sends the Push Submission and prints the response from the PPG. The first line instantiates a `SimplePush` object, passing in the PPG URL, the product name, and the PushID suffix, which ensures that the push identifier is unique.

```
SimplePush sp = new SimplePush(new java.net.URL(ppgAddress), "SampleApp",
    "/sampleapp");
```

The next block of lines instantiates a `QualityOfService` object, sets a variety of attributes, and adds them to the push message:

```
QualityOfService sp_qos = new QualityOfService();

//Set the desired QOS attributes.
sp_qos.setDeliveryMethod(DeliveryMethod.confirmed);
sp_qos.setPriority(DeliveryPriority.high);
sp_qos.setNetwork("GSM");
sp_qos.setNetworkRequired(true);
sp_qos.setBearer("USSD");
sp_qos.setBearerRequired(false);

//Set the QOS for the push message.
sp.setQualityOfService(sp_qos);
```

See `QualityOfService` in the accompanying JavaDoc API documentation for more information.

The next two lines send the Push Submission, using the `SimplePush` method that encapsulates an SL payload, and print the response from the PPG.

```
PushResponse response =
    sp.pushServiceLoading(clientAddress, SvcLoadURI,
    ServiceLoadingAction.executeHigh);
printResults(response);
```

The preceding lines are enclosed in a `try...catch` block, allowing the `SubmitMsg` method to report and act upon any exceptions that might occur. In this example, the action taken in response to any exception is to print the contents of the exceptions to the console.

Finally, the `main` method instantiates an `spServiceLoad` object and calls the `SubmitMsg` method:

```
spServiceLoad spsl = new spServiceLoad();
spsl.SubmitMsg();
```

Cache Operation Payload Example

This example demonstrates how to send a Push Submission with a Cache Operation payload. This content type invalidates content objects in the user agent cache. All invalidated content objects must be reloaded from the server on which they originated the next time they are accessed. This example demonstrates the essentials of building and sending a Push Submission using the `SimplePush` class.

This example uses some of the same constants and other code as the previous examples.

What It Does

The `spCacheOp.java` file declares two constant fields that define the recipient address and the URL of the PPG. The public `spCacheOp` class encapsulates all of the program's functionality in three methods:

- The `printResults` method prints the results of the Push Submission.
- The `SubmitMsg` method instantiates the `SimplePush` object, sends the Push Submission, and handles any exceptions returned from the PPG.
- The `main` method calls the `SubmitMsg` method.

The exception handling demonstrated in this example is limited to displaying error messages on the console. It would be better, however, to build a more automated exception-handling and response mechanism into any applications that you write.

A complete code listing follows.

spCacheOp.java

```
/*
 * Title: SimplePush Cache Operation Payload Example
 * Description: A basic Push Submission example using a Cache Operation
 * payload
 */

import java.net.MalformedURLException;
import java.io.IOException;
import com.openwave.wappush.*;

public class spCacheOp {
    private final static String ppgAddress =
        "http://devgate2.openwave.com:9002/pap";
    private final static String[] clientAddress =
       {"jdoe_devgate2.openwave.com/TYPE=USER@ppg.openwave.com"};

    private static void printResults(PushResponse pushResponse)
        throws WapPushException, MalformedURLException, IOException {
        //Read the response to find out if the Push Submission succeded.
        //1001 = "Accepted for processing"
        if (pushResponse.getResultCode() == 1001) {
            try {
                String pushID = pushResponse.getPushID();
                SimplePush sp = new SimplePush(new
                    java.net.URL(ppgAddress),
                    "SampleApp", "/sampleapp");
                StatusQueryResponse queryResponse =
                    sp.queryStatus(pushID, null);
                StatusQueryResult queryResult =
                    queryResponse.getResult(0);
                System.out.println("Message status: " +
                    queryResult.getMessageState());
            }
            catch (WapPushException exception) {
                System.out.println("*** ERROR - WapPushException (" +
                    exception.getMessage() + ")");
            }
            catch  (MalformedURLException exception) {
                System.out.println("*** ERROR - MalformedURLException ("
                + exception.getMessage() + ")");
            }
            catch (IOException exception) {
                System.out.println("*** ERROR - IOException ("
                    + exception.getMessage() + ")");
            }
        }
        else
            System.out.println("Message failed");
    }//printResults

    public void SubmitMsg() throws WapPushException, IOException {
        try {
```

```
                //Instantiate a SimplePush object passing in the PPG URL,
                //product name, and PushID suffix, which ensures that the
                //PushID is unique.
                SimplePush sp = new SimplePush(new java.net.URL(ppgAddress),
                    "SampleApp", "/sampleapp");
                //Send the Cache Operation message.
                PushResponse response = sp.pushCacheOperation(clientAddress,
                    CacheOperationType.cachedService,
                    "http://www.transitel.net/wap");

                //Print the response from the PPG.
                printResults(response);
            }//try
            catch (WapPushException exception) {
                System.out.println("*** ERROR - WapPushException (" +
                    exception.getMessage() + ")");
            }
            catch (IOException exception) {
                System.out.println("*** ERROR - IOException ("
                    + exception.getMessage() + ")");
            }
        }//SubmitMsg()

        public static void main(String[] args) throws WapPushException,
            IOException {
            spCacheOp spco = new spCacheOp();
            spco.SubmitMsg();
        }//main
    }//class spCacheOp
```

How It Works

This simple example declares a class, `spCacheOp`, in which the recipient address and the PPG URL are declared as constants:

```
private final static String ppgAddress =
    "http://devgate2.openwave.com:9002/pap";
private final static String[] clientAddress =
    {"jdoe_devgate2.openwave.com/TYPE=USER@ppg.openwave.com"};
```

Notice that the `clientAddress` field is a String array. This is necessary because the `pushCacheOperation` method requires a String array for its *`addresses`* parameter.

The next block defines the private `printResults` method, which tests the response object returned from the PPG for information regarding the Push Submission. If the Push Submission succeeded, this method prints some of the pertinent response information.

The `SubmitMsg` method contains the code that builds and sends the Push Submission and prints the response from the PPG. The first line instantiates a `SimplePush` object, passing in the PPG URL, the product name, and the PushID suffix, which ensures that the push identifier is unique.

```
SimplePush sp = new SimplePush(new java.net.URL(ppgAddress), "SampleApp",
    "/sampleapp");
```

The second line sends the Push Submission using the `SimplePush` method that encapsulates a Cache Operation payload. The last line prints the response from the PPG.

```
PushResponse response = sp.pushCacheOperation(clientAddress,
    CacheOperationType.cachedService, "http://www.transitel.net/wap");
printResults(response);
```

The preceding lines are enclosed in a `try...catch` block, allowing the `SubmitMsg` method to report and act upon any exceptions that might occur. In this example, the action taken in response to any exception is to print the contents of the exceptions to the console.

Finally, the `main` method instantiates an `spCacheOp` object and calls the `SubmitMsg` method:

```
spCacheOp spco = new spCacheOp();
spco.SubmitMsg();
```

Custom Content Payload Example

This example demonstrates how to send a Push Submission with a Custom Content payload. This content type can contain virtually any type of content. This example demonstrates the essentials of building and sending a Push Submission.

This example uses some of the same constants and other code as the previous examples.

What It Does

The `spCustomContent.java` file declares two constant fields that define the recipient address and the URL of the PPG. The public `spCustomContent` class encapsulates all of the program's functionality in three methods:

- The `printResults` method prints the results of the Push Submission.
- The `SubmitMsg` method instantiates the `SimplePush` object, sends the Push Submission, and handles any exceptions returned from the PPG.
- The `main` method calls the `SubmitMsg` method.

The exception handling demonstrated in this example is limited to displaying error messages on the console. It would be better, however, to build a more automated exception-handling and response mechanism into any applications that you write.

A complete code listing follows.

spCustomContent.java

```
/*
 * Title: SimplePush Custom Content Payload Example
 * Description: A basic Push Submission example using a Custom Content
 * payload
 */

import java.net.MalformedURLException;
import java.io.IOException;
import com.openwave.wappush.*;

public class spCustomContent {
    private final static String ppgAddress =
        "http://devgate2.openwave.com:9002/pap";
    private final static String[] clientAddress =
       {"jdoe_devgate2.openwave.com/TYPE=USER@ppg.openwave.com"};

    private static void printResults(PushResponse pushResponse)
        throws WapPushException, MalformedURLException, IOException {
        //Read the response to find out if the Push Submission succeded.
        //1001 = "Accepted for processing"
        if (pushResponse.getResultCode() == 1001) {
            try {
                String pushID = pushResponse.getPushID();
                SimplePush sp = new SimplePush(new
                    java.net.URL(ppgAddress),
                    "SampleApp", "/sampleapp");
                StatusQueryResponse queryResponse =
                    sp.queryStatus(pushID, null);
                StatusQueryResult queryResult =
                    queryResponse.getResult(0);
                System.out.println("Message status: " +
                    queryResult.getMessageState());
            }
                catch (WapPushException exception) {
                System.out.println("*** ERROR - WapPushException (" +
                    exception.getMessage() + ")");
            }
            catch  (MalformedURLException exception) {
                System.out.println("*** ERROR - MalformedURLException ("
                + exception.getMessage() + ")");
            }
            catch (IOException exception) {
                System.out.println("*** ERROR - IOException ("
                    + exception.getMessage() + ")");
            }
        }
        else
            System.out.println("Message failed");
    }//printResults

    public void SubmitMsg() throws WapPushException,
        UnknownMediaTypeException {
```

```
            try {
                //Instantiate a SimplePush object passing in the PPG URL,
                //product name, and PushID suffix, which ensures that the
                //PushID is unique.
                SimplePush sp = new SimplePush(new java.net.URL(ppgAddress),
                    "SampleApp", "/sampleapp");
                //Send the Custom Content payload.
                PushResponse response = sp.pushCustomContent(clientAddress,
                    "Happy Birthday!!", "text/plain");

                //Print the response from the PPG.
                printResults(response);
            }//try
            catch (IOException exception) {
                System.out.println("*** ERROR - IOException ("
                    + exception.getMessage() + ")");
            }
            catch (WapPushException exception) {
                System.out.println("*** ERROR - WapPushException (" +
                    exception.getMessage() + ")");
            }
        }//SubmitMsg()

        public static void main(String[] args) throws WapPushException,
            UnknownMediaTypeException {
            spCustomContent spcc = new spCustomContent();
            spcc.SubmitMsg();
        }//main
    }//class spCustomContent
```

How It Works

This simple example declares a class, `spCustomContent`, in which the recipient address and the PPG URL are declared as constants:

```
private final static String ppgAddress =
    "http://devgate2.openwave.com:9002/pap";
private final static String[] clientAddress =
    {"jdoe_devgate2.openwave.com/TYPE=USER@ppg.openwave.com"};
```

Notice that the `clientAddress` field is a String array. This is necessary because the `pushCustomContent` method requires a String array for its *`addresses`* parameter.

The next block defines the private `printResults` method, which tests the response object returned from the PPG for information regarding the Push Submission. If the Push Submission succeeded, this method prints some of the pertinent response information.

The `SubmitMsg` method contains the code that builds and sends the Push Submission and prints the response from the PPG. The first line instantiates a `SimplePush` object, passing in the PPG URL, the product name, and the PushID suffix, which ensures that the push identifier is unique.

```
SimplePush sp = new SimplePush(new java.net.URL(ppgAddress), "SampleApp",
    "/sampleapp");
```

The second line sends the Push Submission using the `SimplePush` method that encapsulates a Custom Content payload. The last line prints the response from the PPG.

```
PushResponse response = sp.pushCustomContent(clientAddress,
    "Happy Birthday!!", "text/plain");
printResults(response);
```

The preceding lines are enclosed in a `try...catch` block, allowing the `SubmitMsg` method to report and act upon any exceptions that might occur. In this example, the action taken in response to any exception is to print the contents of the exceptions to the console.

Finally, the `main` method instantiates an `spCustomContent` object and calls the `SubmitMsg` method:

```
spCustomContent spcc = new spCustomContent();
spcc.SubmitMsg();
```

Status Query Message and Response Example

This example demonstrates how to send a Status Query message, which requests the current status of a Push Submission from the PPG. In addition to the standard numeric code and text message, the Push Status Query response also contains an attribute that indicates the status of the specified Push Submission. Nine message states are currently specified:

- `aborted` The addressee aborted the message.
- `cancelled` A Push Cancellation successfully canceled the message.
- `delivered` The PPG successfully delivered the message to the addressee.
- `expired` The message reached the maximum age allowed by PPG policy or could not be delivered by the time specified in the Push Submission.
- `pending` The PPG accepted the message and is in the process of delivering it.
- `rejected` The addressee rejected the message.
- `timeout` The delivery process timed out on the PPG.
- `undeliverable` A problem prevented delivery of the message. Call the `StatusQueryResult.getCode` and `StatusQueryResult.getDesc` methods to retrieve the code and text message returned from the PPG.
- `unknown` The PPG has no information about the status of the message.

See *WAP Push Access Protocol* for definitions of the numeric codes and accompanying text messages.

NOTE PPG implementation of Status Query reporting is optional. If the PPG does not support Status Query reporting, the response contains status code 3001 with the text message Not Implemented.

What It Does

The `spStatusQM.java` file declares three constant fields that define the specific addressee for which to retrieve status information, the URL of the PPG, and the unique identifier of the Push Submission for which to retrieve status information. The public `spStatusQM` class encapsulates all of the program's functionality in three methods:

- The `printStatusQueryResponse` method prints the results of the Status Query operation.
- The `SubmitMsg` method instantiates the `SimplePush` object, sends the Status Query message, prints the results, and handles any exceptions returned from the PPG.
- The `main` method calls the `SubmitMsg` method.

The exception handling demonstrated in this example is limited to displaying error messages on the console. It would be better, however, to build a more automated exception-handling and response mechanism into any applications that you write.

A complete code listing follows.

spStatusQM.java

```
/*
 * Title: SimplePush Status Query Message Example
 * Description: A basic Push Submission example that sends a
 * Status Query message and reports the results
 */

import java.net.MalformedURLException;
import java.io.IOException;
import com.openwave.wappush.*;

public class spStatusQM {
    private final static String ppgAddress =
        "http://devgate2.openwave.com:9002/pap";
    private final static String[] clientAddress =
       {"jdoe_devgate2.openwave.com/TYPE=USER@ppg.openwave.com"};
    private final static String pushID = "9f1000a021@openwave.com";

    //Private method that prints the response information.
    private static void printStatusQueryResponse (StatusQueryResponse
        response) {
        System.out.println("StatusQueryResponse");
        System.out.println("     push-id = " + response.getPushID());
        int resultCount = response.getResultCount();
        System.out.println("     result-count = " + resultCount);
        for (int i = 0; i < resultCount; ++i) {
            System.out.println("     result #" + i);
            StatusQueryResult result = response.getResult(i);
            MessageState mState = result.getMessageState();
            System.out.println("          message-state = " +
                mState.toString());
            System.out.println("          code = " +
                result.getCode());
            System.out.println("          description = " +
                result.getDesc());
            DateTime dt = result.getEventTime();
            System.out.println("          event-time = " +
                dt.toString());
            int addressCount = result.getAddressCount();
            System.out.println("          address-count = " +
                addressCount);

            for (int j = 0; j < addressCount; ++j) {
                System.out.println("          address #" + j +
                    " = " + result.getAddress(j));
            }
        }
    }//printStatusQueryResponse

    public void SubmitMsg() throws WapPushException, IOException {
        try {
            //Instantiate a SimplePush object passing in the PPG URL,
            //product name, and PushID suffix, which ensures that the
```

```
                //PushID is unique.
                SimplePush sp = new SimplePush(new java.net.URL(ppgAddress),
                    "SampleApp", "/sampleapp");

                //Send the Status Query message.
                StatusQueryResponse response =
                    sp.queryStatus(pushID, clientAddress);

                //Print the response from the PPG.
                printStatusQueryResponse(response);
            }//try
            catch (WapPushException exception) {
                System.out.println("*** ERROR - WapPushException (" +
                    exception.getMessage() + ")");
            }
            catch (IOException exception) {
                System.out.println("*** ERROR - IOException ("
                    + exception.getMessage() + ")");
            }
        }//SubmitMsg()

        public static void main(String[] args) throws WapPushException,
            IOException {
            spStatusQM spsqm = new spStatusQM();
            spsqm.SubmitMsg();
        }//main
    }//class spStatusQM
```

How It Works

This simple example declares a class, `spStatusQM`, in which the recipient address, the PPG URL, and the unique identifier of the Push Submission for which to retrieve status information are declared as constants:

```
private final static String ppgAddress =
    "http://devgate2.openwave.com:9002/pap";
private final static String[] clientAddress =
    {"jdoe_devgate2.openwave.com/TYPE=USER@ppg.openwave.com"};
private final static String pushID = "9f1000a021@openwave.com";
```

Notice that the `clientAddress` field is a String array. This is necessary because the `queryStatus` method requires a String array for its *`addresses`* parameter.

The next block defines the private `printStatusQueryResponse` method, which tests the response object returned from the PPG for information regarding the Status Query operation.

The `SubmitMsg` method contains the code that builds and sends the Status Query message and prints the response from the PPG. The first line instantiates a `SimplePush` object, passing in the PPG URL, the product name, and the PushID suffix, which ensures that the push identifier is unique.

```
SimplePush sp = new SimplePush(new java.net.URL(ppgAddress), "SampleApp",
    "/sampleapp");
```

The next two lines send the Status Query message, using the `SimplePush` method that encapsulates a Status Query operation, and print the response from the PPG.

```
StatusQueryResponse response = sp.queryStatus(pushID, clientAddress);
printStatusQueryResponse(response);
```

The preceding lines are enclosed in a `try...catch` block, allowing the `SubmitMsg` method to report and act upon any exceptions that might occur. In this example, the action taken in response to any exception is to print the contents of the exceptions to the console.

Finally, the `main` method instantiates an `spStatusQM` object and calls the `SubmitMsg` method:

```
spStatusQM spsqm = new spStatusQM();
spsqm.SubmitMsg();
```

Push Cancel Message and Response Example

This example demonstrates how to send a Push Cancellation, which allows the push initiator to attempt to cancel a Push Submission, and how to read the response from the PPG, which indicates the status of the cancellation request. A Push Submission can be canceled only before it has been delivered.

PPG support for the Push Cancellation operation is optional. If the PPG does not support Push Cancellation, the response contains status code 3001 with the text message Not Implemented.

What It Does

The `spCancelPush.java` file declares three constant fields that define the recipient address, the URL of the PPG, and the unique identifier of the Push Submission to cancel. In this example, the Push Cancellation operation attempts to cancel the specified Push Submission for one addressee only. The public `spCancelPush` class encapsulates all of the program's functionality in three methods:

- The `printCancelResponse` method prints the results of the Push Cancellation operation.
- The `SubmitMsg` method instantiates the `SimplePush` object, sends the Cancel message, prints the pertinent information from the PPG response, and handles any exceptions returned from the PPG.
- The `main` method calls the `SubmitMsg` method.

The exception handling demonstrated in this example is limited to displaying error messages on the console. It would be better, however, to build a more automated exception-handling and response mechanism into any applications that you write.

A complete code listing follows.

spCancelPush.java

```
/*
 * Title: SimplePush Push Cancellation Example
 * Description: A basic Push Submission example that cancels a
 * push message.
 */

import java.net.MalformedURLException;
import java.io.IOException;
import com.openwave.wappush.*;

public class spCancelPush {
    private final static String ppgAddress =
        "http://devgate2.openwave.com:9002/pap";
    private final static String[] clientAddress =
       {"jdoe_devgate2.openwave.com/TYPE=USER@ppg.openwave.com"};
    private final static String pushID = "9f1000a021@openwave.com";

    private static void printCancelResponse(CancelResponse
        cancelResponse) {
        System.out.println("CancelResponse");
        System.out.println(" push-id = " + cancelResponse.getPushID());
        int resultCount = cancelResponse.getResultCount();
        System.out.println(" result-count = " + resultCount);
        for (int i = 0; i < resultCount; ++i) {
            System.out.println(" result #" + i);
            CancelResult result = cancelResponse.getResult(i);
            System.out.println(" code = " + result.getCode());
            System.out.println(" description = " + result.getDesc());
            int addressCount = result.getAddressCount();
            System.out.println(" address-count = " + addressCount);
            for (int j = 0; j < addressCount; ++j) {
                System.out.println(" address #" + j + " = " +
                    result.getAddress(j));
            }
        }
    }//printCancelResponse

    public void SubmitMsg() throws WapPushException, IOException {
        try {
            //Instantiate a SimplePush object passing in the PPG URL,
            //product name, and PushID suffix, which ensures that the
            //PushID is unique.
            SimplePush sp = new SimplePush(new java.net.URL(ppgAddress),
                "SampleApp", "/sampleapp");

            //Send the Cancel message.
            CancelResponse response =
                sp.cancel(pushID, clientAddress);

            //Print the response from the PPG.
            printCancelResponse(response);
        }//try
```

```
            catch (WapPushException exception) {
                System.out.println("*** ERROR - WapPushException (" +
                    exception.getMessage() + ")");
            }
            catch (IOException exception) {
                System.out.println("*** ERROR - IOException ("
                    + exception.getMessage() + ")");
            }
        }//SubmitMsg()

        public static void main(String[] args) throws WapPushException,
            IOException {
            spCancelPush spcp = new spCancelPush();
            spcp.SubmitMsg();
        }//main
    }//class spCancelPush
```

How It Works

This simple example declares a class, `spCancelPush`, in which the recipient address, the PPG URL, and the unique identifier of the Push Submission to cancel are declared as constants:

```
private final static String ppgAddress =
    "http://devgate2.openwave.com:9002/pap";
private final static String[] clientAddress =
    {"jdoe_devgate2.openwave.com/TYPE=USER@ppg.openwave.com"};
private final static String pushID = "9f1000a021@openwave.com";
```

Notice that the `clientAddress` field is a String array. This is necessary because the `cancel` method requires a String array for its *`addresses`* parameter.

The next block defines the private `printCancelResponse` method, which tests the response object returned from the PPG for information regarding the Cancel operation.

The `SubmitMsg` method contains the code that builds and sends the Cancel message and prints the response from the PPG. The first line instantiates a `SimplePush` object, passing in the PPG URL, the product name, and the PushID suffix, which ensures that the push identifier is unique.

```
SimplePush sp = new SimplePush(new java.net.URL(ppgAddress), "SampleApp",
    "/sampleapp");
```

The next two lines send the Cancel message, using the `SimplePush` method that encapsulates a Push Cancellation operation, and print the response from the PPG.

```
sp.cancel(pushID, clientAddress);
printCancelResponse(response);
```

The preceding lines are enclosed in a `try...catch` block, allowing the `SubmitMsg` method to report and act upon any exceptions that might occur. In this example, the action taken in response to any exception is to print the contents of the exceptions to the console.

Finally, the `main` method instantiates an `spCancelPush` object and calls the `SubmitMsg` method:

```
spCancelPush spcp = new spCancelPush();
spcp.SubmitMsg();
```

Client Capabilities Query Message and Response Example

This example demonstrates how to send a Client Capabilities Query (CCQ) message, which requests the capabilities for a specific client device from the PPG. A CCQ response also includes a complete User Agent Profile (UAProf), which provides detailed information about the specified client device.

> **NOTE** PPG implementation of Client Capabilities Query reporting is optional. If the PPG does not support CCQ reporting, the response contains status code 3001 with the text message Not Implemented.

What It Does

The `spClientCaps.java` file declares two constant fields that define the specific addressee for which to retrieve CCQ information and the URL of the PPG. The public `spClientCaps` class encapsulates all of the program's functionality in four methods:

- The `printCCQResponse` and `printUAProfile` methods print client capabilities and UAProf information resulting from the CCQ operation.
- The `SubmitMsg` method instantiates the `SimplePush` object, sends the CCQ message, prints the results, and handles any exceptions returned from the PPG.
- The `main` method calls the `SubmitMsg` method.

The exception handling demonstrated in this example is limited to displaying error messages on the console. It would be better, however, to build a more automated exception-handling and response mechanism into any applications that you write. A complete code listing follows.

spClientCaps.java

```
/*
 * Title: SimplePush Client Capabilities Query Message Example
 * Description: A basic Push Submission example that sends a
 * Client Capabilities Query message
 */

import java.net.MalformedURLException;
import java.io.IOException;
import java.util.Enumeration;
import java.net.URL;
import com.openwave.wappush.*;

public class spClientCaps {
    private final static String ppgAddress =
        "http://devgate2.openwave.com:9002/pap";
    private final static String clientAddress =
       "jdoe_devgate2.openwave.com/TYPE=USER@ppg.openwave.com";

    //Private methods that print the response information.
    private static void printCCQResponse(CcqResponse response) {
        System.out.println("ClientCapsQueryResponse");
        System.out.println("     result-code = " + response.getCode());
        System.out.println("     result-description = " +
            response.getDesc());
        System.out.println("     query-id = " + response.getQueryID());
        System.out.println("     address = " + response.getAddress());
        UAProfile profile = response.getUserAgentProfile();
        if (profile == null)
            System.out.println("UAProfile = null");
        else
            printUAProfile(profile);
    }//printCCQResponse

    private static void printUAProfile (UAProfile profile) {
        System.out.println("UAProfile");
        int compCount = profile.componentCount();
        System.out.println("     profile-id = " + profile.getID());
        System.out.println("     component-count = " + compCount);
        for (int i = 0; i < compCount; ++i) {
            UAComponent comp = profile.getComponent(i);
            System.out.println("          component #" + i);
            System.out.println("          component-id = " +
                comp.getID());
            System.out.print("          defaults-resource = ");
            try {
                URL defaultsURL = comp.getDefaultsResource();
                System.out.println(defaultsURL);
            }
            catch (Exception e) {
                System.out.println("**malformed URL**");
            }
            System.out.print(" schema-resource = ");
```

```
            try {
                URL schemaURL = comp.getSchemaResource();
                System.out.println(schemaURL);
            }
            catch (Exception e) {
                System.out.println("**malformed URL**");
            }

            Enumeration attNames = comp.getAttributeNames();
            while (attNames.hasMoreElements()) {
                String attName = (String) attNames.nextElement();
                System.out.println("               " + attName + " = ");
                Enumeration attValues = comp.getAllValuesOf(attName);
                while (attValues.hasMoreElements()) {
                    Object attValue = attValues.nextElement();
                    if (attValue instanceof Bag) {
                       Bag bag = (Bag) attValue;
                        int itemCount = bag.count();
                        System.out.print("                 (");
                        for (int j = 0; j < itemCount; ++j) {
                            if (j != 0)
                                System.out.print(", ");
                            System.out.print(bag.get(j));
                        }
                        System.out.println(")");
                    }
                    else
                    System.out.println("                " + attValue);
                }
            }
        }
    }//printUAProf

    public void SubmitMsg() throws WapPushException, IOException {
        try {
            //Instantiate a SimplePush object passing in the PPG URL,
            //product name, and PushID suffix, which ensures that the
            //PushID is unique.
            SimplePush sp = new SimplePush(new java.net.URL(ppgAddress),
                "SampleApp", "/sampleapp");

            //Send the CCQ message.
            CcqResponse response = sp.queryCapabilities(clientAddress,
                null);

            //Print the response from the PPG.
            printCCQResponse(response);
        }//try
        catch (WapPushException exception) {
            System.out.println("*** ERROR - WapPushException (" +
                exception.getMessage() + ")");
        }
        catch (IOException exception) {
```

```
            System.out.println("*** ERROR - IOException ("
                + exception.getMessage() + ")");
        }
    }//SubmitMsg()

    public static void main(String[] args) throws WapPushException,
        IOException {
        spClientCaps spccq = new spClientCaps();
        spccq.SubmitMsg();
    }//main
}//class spClientCaps
```

How It Works

This simple example declares a class, `spClientCaps`, in which the PPG URL and recipient address for which to retrieve capabilities information are declared as constants:

```
private final static String ppgAddress =
    "http://devgate2.openwave.com:9002/pap";
private final static String clientAddress =
    "jdoe_devgate2.openwave.com/TYPE=USER@ppg.openwave.com";
```

The next block defines the private `printCCQResponse` and `printUAProfile` methods, which print client capabilities and UAProf information resulting from the CCQ operation

The `SubmitMsg` method contains the code that builds and sends the CCQ message and prints the response from the PPG. The first line instantiates a `SimplePush` object, passing in the PPG URL, the product name, and the PushID suffix, which ensures that the push identifier is unique.

```
SimplePush sp = new SimplePush(new java.net.URL(ppgAddress), "SampleApp",
    "/sampleapp");
```

The next two lines send the CCQ message, using the `SimplePush` method that encapsulates a CCQ operation, and print the response from the PPG.

```
CcqResponse response = sp.queryCapabilities(clientAddress, null);
printCCQResponse(response);
```

The preceding lines are enclosed in a `try...catch` block, allowing the `SubmitMsg` method to report and act upon any exceptions that might occur. In this example, the action taken in response to any exception is to print the contents of the exceptions to the console.

Finally, the `main` method instantiates an `spClientCaps` object and calls the `SubmitMsg` method:

```
spClientCaps spccq = new spClientCaps();
spccq.SubmitMsg();
```

Travel Example

The Travel example is based on and integrated with a collection of more than 20 WML files and one WMLScript file. This example which presents an Openwave Mobile Browser user with a wireless air application, in which the user can reserve and check the status of flights. The `SimplePush` portion of the Travel example is integrated into the section of the application that allows users to check flight status. In this example, a flight has a 50-50 chance of being canceled or delayed.

What It Does

The Travel example illustrates all PAP operations except Push Cancellation, Client Capabilities Query, and Result Notification. PAP operations are mapped to user scenarios as follows:

- When a user requests notification of flight status changes and the flight is canceled, a Service Loading Push is sent to the user.
- When a user requests notification of flight status changes and the flight is delayed, the following sequence of events occurs:
  - A Service Indication is sent to the user notifying the user of the delay, with a flight status URL as the Service Indication URI.
  - A period of time elapses and then the example code generates a second flight delay.
  - The Travel example sends a Status Query message to determine the state of the first Service Indication.
  - If the first Service Indication has been delivered, the Travel example sends a Cache Operation message to invalidate the flight status URL in the user's cache.
  - The Travel example then sends a second Service Indication to the user with the same flight status URL as the first Service Indication URI. The flight status URL now informs the user of a different departure time because the previous step invalidated the user's cache.

To run the Travel example, use the Openwave Simulator to open the `travel2.wml` file in the `<installroot>/examples/TravelDemo/source/wml` directory. For more information on installing and configuring the WAP Push Library and associated utilities and examples, see "Installing and Configuring" on page 15.

The Travel example consists of two Java servlets, which are integrated into the Travel example by means of a slight modification of a single WML file (`statfltnum.wml`). The FlightStatus servlet is not called directly from the Travel example, but rather is only used as the target URI for Service Loading and Service Indication push messages. The FlightStatus servlet generates a WML deck that informs the user of a flight cancellation or of the new departure time in the event of a delay. The FlightStatus servlet does not use the Push Library, so it is not covered in detail here. For more information on the FlightStatus servlet, open the `FlightStatus.java` file in the `examples/TravelDemo/src/java` directory. The source file contains extensive comments.

The TravelServer servlet, which is covered in this chapter, is called from `statfltnum.wml` when the user presses the Notify Me of Changes button. The servlet then performs the sequence of events just outlined.

A complete code listing of the `TravelServer.java` servlet follows.

TravelServer.java

```
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import com.openwave.wappush.CacheOperationType;
import com.openwave.wappush.DeliveryMethod;
import com.openwave.wappush.PushResponse;
import com.openwave.wappush.ServiceLoadingAction;
import com.openwave.wappush.ServiceIndicationAction;
import com.openwave.wappush.SimplePush;
import com.openwave.wappush.StatusQueryResponse;
import com.openwave.wappush.StatusQueryResult;
import com.openwave.wappush.WapPushException;

public class TravelServer extends HttpServlet
{
    /**
     * Parameters supplied by the "get" request from the WML deck.
     */
    private String subID      = null;
    private String Airline    = null;
    private String Flight     = null;

    /**
     * Internal variables.
     */
    private String ppgString       = null;
    private String clientAddress   = null;
    private String alertURL        = null;

    /**
     * One hour in milliseconds.
     */
    static final int millisPerHour  = 60 * 60 * 1000;

    /*
     * Set this to true to get debugging messages in the Tomcat console
     * window and log.
     */
    static final boolean debug = true;

    /**
```

```
 * Initializes the servlet.
 */
public void init(ServletConfig config) throws ServletException
{
    super.init(config);
}

/**
 * Destroys the servlet.
 */
public void destroy()
{
    ;//Do nothing.
}

/**
 * Handles the HTTP <code>GET</code> method.
 * @param request servlet request
 * @param response servlet response
 *
 * The response is handled by the Service Indication to the PPG.
 */
protected synchronized void
doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, java.io.IOException
{
    int tokType;

if (request.getParameter("notify").equalsIgnoreCase("no"))
{
    // User didn't request flight status notification, so just
    // redirect user to home page.
    response.sendRedirect("/travel/travel2.wml");
    response.flushBuffer();
    return;
}

if (request.getHeader("X-Up-Wap-Push-Unsecure" == null)
{
    // The phone is in direct mode, so this example will not work.
    // Warn the user and exit.
    response.sendRedirect("/travel/nohttpdirect.wml");
    response.flushBuffer();
    return;
}

    /**
     * Output a simple WML deck and flush the buffer. This closes the
     * connection to the Mobile Browser, allowing subsequent pushes
     * to be displayed immediately. If this is not done, the
     * connection between the servlet and the Mobile Browser remains
     * open until the servlet terminates, and no pushes
     * can be delivered to the Mobile Browser until that occurs.
     */
```

```
response.sendRedirect("/travel/notifychange.wml");
response.flushBuffer();

/**
 * Get parameter from the get request.
 *
 * @param subID - the subscriber's ID
 * @param Airline - the name of the Airline
 * @param Flight - the flight number
 * @param ppgString - the address of the PPG
 */
subID = request.getHeader("X-Up-Subno");
Airline = request.getParameter("airline");
Flight = request.getParameter("flight");
ppgString = "http://" +
    request.getHeader("X-Up-Wap-Push-Unsecure");
/*
 * Create the full client address.
 */
clientAddress = subID + "/TYPE=USER@ppg.phone.com";

/**
 * If any of the params are null, just return.
 */
if(Airline == null || Flight == null)
{
    return;
}

//This yields a value like
//http://loki.phone.com:8080/travel/FlightStatus
alertURL = request.getScheme() + "://" + request.getServerName()
    + ":" + request.getServerPort()
    + "/travel/servlet/FlightStatus?airline="+Airline+"&amp;
    flight="+Flight;

if(debug)
{
    System.out.println("subID = "+subID+"\nPPG =
        "+ppgString+"\nairline = "+Airline+ "\nflight =
        "+Flight);
    System.out.flush();
}

URL ppgURL = null;

try
{
    ppgURL = new URL(ppgString);
}
catch (MalformedURLException e)
{
    System.err.println(e.toString());
    System.exit(-1);
```

```
}

SimplePush sp = new SimplePush(ppgURL, "TravelDemo",
    "/TravelDemo");

sp.setQualityOfService(null, DeliveryMethod.confirmed, null,
    false, null, false);
PushResponse pushResponse = null;

// 50-50 chance of a delay or a cancellation.
if(Math.random() > 0.5)
{
    //Cancellation
    alertURL += "&amp;cancelled=cancelled";
    try
    {
        pushResponse = sp.pushServiceLoading(new String[]
            {clientAddress}, alertURL,
            ServiceLoadingAction.executeHigh);
    }
    catch (WapPushException e)
    {
        System.err.println(e.toString());
        System.exit(-1);
    }
    catch (IOException e)
    }
        System.err.println(e.toString());
        System.exit(-1);
    }
}
else
{
    //Delays (plural)
    try
    {
        pushResponse = sp.pushServiceIndication(new String[]
            {clientAddress},
            "Travel Service: Flight Status Update",
            alertURL,
            ServiceIndicationAction.signalHigh);
    }
    catch (WapPushException e)
    {
        System.err.println(e.toString());
        System.exit(-1);
    }
    catch (IOException e)
    }
        System.err.println(e.toString());
        System.exit(-1);
    }
    //Now sit idle for 10 seconds before issuing another delay.
    //For purposes on this demo, wait only 10 seconds just to
```

```
//keep things moving.
int sleepTime = 10;    // Wait for 10 seconds.
try
{
    wait(sleepTime * 1000);
}
catch (Exception exception)
{
    System.out.println("Exception during wait: " +
        exception.getMessage());
}

//Has the previous Alert been delivered?
StatusQueryResponse statusResponse = null;

try
{
    statusResponse = sp.queryStatus(pushResponse.getPushID(),
        new String[] {clientAddress} );
}
catch (WapPushException e)
{
    System.err.println(e.toString());
    System.exit(-1);
}
catch (IOException e)
}
    System.err.println(e.toString());
    System.exit(-1);
}

//Now check to see if the message has already been delivered.
//Do a very simple check here. You could be more elegant and
//actually parse the response string.
//If the push message has been delivered, send a CacheOp
//to invalidate the previous push message.
StatusQueryResult sqResult = statusResponse.getResult(0);

if (sqResult != null &&
    sqResult.getMessageState().equals("delivered"))
{
    try
    {
        pushResponse = sp.pushCacheOperation(new String[]
            {clientAddress},
            CacheOperationType.cachedService, alertURL);
    }
    catch (WapPushException e)
    {
        System.err.println(e.toString());
        System.exit(-1);
    }
    catch (IOException e)
    }
```

```
                        System.err.println(e.toString());
                        System.exit(-1);
                    }
                }

                //If it has been delivered, this Alert replaces it.
                try
                {
                    pushResponse = sp.pushServiceIndication(new String[]
                        {clientAddress},
                        "Travel Service: Flight Status Update",
                        alertURL,
                        ServiceIndicationAction.signalHigh);
                    }
                    catch (WapPushException e)
                    {
                        System.err.println(e.toString());
                        System.exit(-1);
                    }
                    catch (IOException e)
                    }
                        System.err.println(e.toString());
                        System.exit(-1);
                    }
                }
            }
        }

        /**
         * Handles the HTTP <code>POST</code> method.
         * @param request servlet request
         * @param response servlet response
         */
        protected void doPost(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, java.io.IOException
        {
            ;//Do nothing.
        }

        /**
         * Returns a short description of the servlet.
         */
        public String getServletInfo()
        {
            return "Short description";
        }
    }//TravelServer.java
```

How It Works

NOTE Only those portions of the TravelServer servlet that are based on the `SimplePush` class are described in detail.

This example begins with several `import` statements, declares the `TravelServer` class, declares and initializes a series of parameters and variables, and declares the Servlet initialization and destruction methods.

The first section of the `doGet` method checks to see if the user requested flight status notification and if the mobile browser or SDK is in HTTP direct mode. If the user did not request flight status notification, the Travel Demo home page appears on the mobile browser:

```
if (request.getParameter("notify").equalsIgnoreCase("no"))
{
    // User didn't request flight status notification, so just
    // redirect user to home page.
    response.sendRedirect("/travel/travel2.wml");
    response.flushBuffer();
    return;
}
```

If the mobile browser or SDK is in HTTP direct mode, it cannot receive push messages. In that case, the user receives a notification in the form of a WML deck and the Travel Demo closes:

```
if (request.getHeader("X-Up-Wap-Push-Unsecure" == null)
{
    // The phone is in direct mode, so this example will not work.
    // Warn the user and exit.
    response.sendRedirect("/travel/nohttpdirect.wml");
    response.flushBuffer();
    return;
}
```

The next section of the `doGet` method extracts a series of parameters that are needed for the various Push Submissions, including the subscriber ID and PPG address from the HTTP headers.

The method then instantiates the `SimplePush` object it will use throughout, sets the confirmed delivery Quality of Service attribute for all of the Push Submissions, and initializes a `PushResponse` object to hold the response messages from the PPG:

```
SimplePush sp = new SimplePush(ppgURL, "TravelDemo",
    "/TravelDemo");
sp.setQualityOfService(null, DeliveryMethod.confirmed, null,
    false, null, false);
    PushResponse pushResponse = null;
```

The *`ppgURL`* parameter is derived from the HTTP headers in an earlier section of the method.

The method then uses a random number generator to determine whether the flight has been canceled or delayed. The results of this step control the Push Submissions issued later.

The next section of the doGet method is broken into a series of try...catch blocks that perform the Push Submissions using the SimplePush class. These blocks are described next.

The first try...catch block uses the SimplePush method that sends a Service Loading message to inform the user of a canceled flight. The *clientAddress* parameter is derived from the HTTP headers in an earlier section of the method. The remainder of this block catches any exceptions, prints the corresponding message, and exits the program:

```
if(Math.random() > 0.5)
{
    //Cancellation
    alertURL += "&amp;cancelled=cancelled";
    try
    {
        pushResponse = sp.pushServiceLoading(new String[]
            {clientAddress}, alertURL,
            ServiceLoadingAction.executeHigh);
    }
    catch (WapPushException e)
    {
        System.err.println(e.toString());
        System.exit(-1);
    }
    catch (IOException e)
    }
        System.err.println(e.toString());
        System.exit(-1);
    }
}
```

The next two try...catch blocks are activated when the random number generator produces a delay instead of a cancellation. The first block sends a Service Indication to inform the user of the delayed flight. The *clientAddress* parameter is derived from the HTTP headers in an earlier section of the method. The remainder of this block catches any exceptions, prints the corresponding message, and exits the program:

```
//Delays (plural)
try
{
    pushResponse = sp.pushServiceIndication(new String[]
        {clientAddress},
        "Travel Service: Flight Status Update",
        alertURL,
        ServiceIndicationAction.signalHigh);
}
catch
{
...
```

The next few lines insert a 10-second delay and catch any exceptions that might occur:

```
int sleepTime = 10; // Wait for 10 seconds.
try
{
    wait(sleepTime * 1000);
}
catch (Exception exception)
{
    System.out.println("Exception during wait: " +
        exception.getMessage());
}
```

The next section sends a Status Query message to determine whether the client received the Service Indication. The first line initializes a `StatusQueryResponse` object to hold the response to the Status Query message. The next line sends the Status Query message and catches any exceptions.

The next two lines extract the Status Query response and determine whether the message was delivered. If it was, the subsequent `try...catch` block sends a Cache Operation to invalidate the client's cache in preparation for the next Service Indication:

```
StatusQueryResponse statusResponse = null;

try
{
    statusResponse = sp.queryStatus(pushResponse.getPushID(),
        new String[] {clientAddress} );
}
catch
...

StatusQueryResult sqResult = statusResponse.getResult(0);

if (sqResult != null &&
    sqResult.getMessageState().equals("delivered"))
{
    try
    {
        pushResponse = sp.pushCacheOperation(new String[]{clientAddress},
            CacheOperationType.cachedService, alertURL);
    }
catch
...
```

The final `try...catch` block sends another Service Indication informing the user that the flight has been delayed yet again:

```
try
{
    pushResponse = sp.pushServiceIndication(new String[] {clientAddress},
        "Travel Service: Flight Status Update",
        alertURL,
        ServiceIndicationAction.signalHigh);
}
catch
...
```

This Service Indication displays updated delay information on the mobile browser because of the preceding Cache Operation, which invalidated the client's cache, requiring the mobile browser to load the second Service Indication URL from the PPG, and with it the updated delay information. If the Cache Operation were omitted, the user would not see the new information supplied with the second Service Indication because the URL is the same for both Service Indications. As a result, the previous delay information would be retrieved from the client's cache.

6 WAP Push Library Essentials

This chapter introduces the essentials of application development using the WAP Push Library.

Required Import Statements

To develop any application using the WAP Push Library, your code must include the following statement:

```
import com.openwave.wappush.*;
```

You can import specific WAP Push Library classes instead of the entire package, if desired.

In addition to the WAP Push Library, your applications may require functionality of the following Java classes:

```
import java.io.IOException;
import java.io.FileNotFoundException;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Enumeration;
```

You may also need to import other Java classes, particularly the Abstract Windowing Toolkit (AWT), Swing, and event-handling classes, to support the user interface elements that your application implements.

WAP Push Library Application Basics

All WAP Push Library applications should perform these operations:

- Build Push Access Protocol (PAP) content appropriate for the type of PAP operation
- Set any desired options for the PAP operation
- Instantiate a `Pusher` object that contains the Push Proxy Gateway (PPG) address for the PAP operation
- Send the PAP operation with the desired content payload
- Read the response returned from the PPG
- Query the response for any desired information
- Handle all exceptions

The following sections provide examples of basic WAP Push Library applications.

PPG and Client Addresses

The examples in this chapter depict all PPG and client addresses as hard-coded constants, in part to show the proper format and in part to make the examples easy to understand. A much better approach would be to extract the PPG and client addresses from the HTTP request headers to the PPG or from a user interface. For more information on the former approach, see "Extracting PPG and Client Addresses from PPG Headers" on page 23, "Sending a Push Submission" on page 30, and "Travel Example" on page 65.

Service Indication Payload Example

This example demonstrates how to send a Push Submission with a Service Indication (SI) payload. This content type sends notifications to addressees in an asynchronous manner. These notifications may, for example, be about new email, changes in stock price, news headlines, advertising, or reminders of various types.

In its most basic form, an SI contains a short message and a URI specifying a service. The message is presented to the addressee upon reception. The addressee has the option of starting the service indicated by the URI immediately or postponing the SI for later handling. If the addressee postpones the SI, the client device stores it.

This example uses some of the same constants and other code as the previous examples, but demonstrates additional push message options and an even more complex payload type.

What It Does

The `ServiceInd.java` file imports all the necessary files, including the WAP Push Library. Several constant fields define the recipient address, the ID string of the Push Submission, the URL of the PPG, and the SI URI. The public `ServiceInd` class encapsulates all of the program's functionality in two methods:

- The `SubmitMsg` method instantiates the necessary WAP Push Library objects, sends the Push Submission, and prints some of the pertinent information from the response object returned from the PPG.
- The `main` method calls the `SubmitMsg` method and handles any exceptions returned from the PPG.

The exception handling demonstrated in this example is limited to displaying error messages on the console. It would be better, however, to build a more automated exception-handling and response mechanism into any applications you write.

A complete code listing follows.

ServiceInd.java

```
/*
 * Title: WAP Push Library Service Indication Payload Example
 * Description: A basic Push Submission example using a
 * Service Indication payload
 */

import java.io.IOException;
import java.io.FileNotFoundException;
import java.net.MalformedURLException;
import java.net.URL;
import com.openwave.wappush.*;

public class ServiceInd {
    //Constants used in this example.
    static final String address = "jdoe_devgate2.openwave.com" +
        "/TYPE=USER@ppg.openwave.com";
    static final String pushID = "9f1000a023@openwave.com";
    static final String ppgAddress =
        "http://devgate2.openwave.com:9002/pap";
    static final String SvcIndURI =
        "http://devgate2.openwave.com/cgi-bin/mailbox.cgi";

    static URL ppgURL = null;
    static URL siURI  = null;

    public void SubmitMsg() throws WapPushException, IOException,
        MalformedURLException {
        //Instantiate and initialize the Pusher object.
        Pusher ppg = new Pusher(ppgURL);
        ppg.initialize();

        //Instantiate the Service Indication object.
        //This is the text string to send to the client device.
        String alertText = "Mobile Mail: New message!";
        ServiceIndication serviceIndication =
            new ServiceIndication(alertText);

        //Set the URI for this SI.
        serviceIndication.setHref(siURI);

        //Add some optional information to the SI.
        ServiceIndicationInfo info =
            new ServiceIndicationInfo("Mailbox", "Full");
        info.AddInfoBlock("ReadMessages", "All");
        serviceIndication.setInfo(info);

        //Set the Service Indication action to signal-high.
        serviceIndication.setAction(ServiceIndicationAction.signalHigh);
        //Add some optional time information.
        serviceIndication.setExpires(
            new DateTime(2004, 6, 15, 12, 0, 1));
        serviceIndication.setCreated(
            new DateTime("2002-06-15T12:00:00Z"));

        //Instantiate the push message object and set some
        //optional information.
        PushMessage pushMessage = new PushMessage(pushID, address);
```

```
            pushMessage.setDeliverBeforeTimestamp(new
                DateTime("2004-06-15T12:00:01Z"));

            //Instantiate a MimeEntity and add the PushMessage
            //and ServiceIndication objects.
            MimeEntity me = new MimeEntity();
            me.addEntity(pushMessage);
            me.addEntity(serviceIndication);

            //Send the push message
            PushResponse pushResponse = (PushResponse) ppg.send(me);

            //Read some information from the response.
            System.out.println("reply-Time = " +
                pushResponse.getReplyTime());
            System.out.println("response-result-code = " +
                pushResponse.getResultCode());
            System.out.println("response-result-desc = " +
                pushResponse.getResultDesc());
        }//SubmitMsg()

        public static void main(String[] args) throws WapPushException,
            IOException {
            try {
                ppgURL = new URL(ppgAddress);
                siURI  = new URL(SvcIndURI);
                ServiceInd si = new ServiceInd();
                si.SubmitMsg();
            }
            //Handle possible exceptions.
            catch (BadMessageException exception) {
                System.out.println("*** ERROR - bad message exception");
                BadMessageResponse response = exception.getResponse();
                System.out.println("*** ERROR = " +
                response.getBadMessageFragment());
            }
            catch (WapPushException exception) {
                System.out.println("*** ERROR - WapPushException (" +
                    exception.getMessage() + ")");
            }
            catch (FileNotFoundException exception) {
                System.out.println("*** ERROR - input file not found");
            }
            catch (MalformedURLException exception) {
                System.out.println("*** ERROR - malformed PPG URL");
            }
            catch (IOException exception) {
                System.out.println( "*** ERROR - I/O exception");
            }
            catch (Exception exception) {
                System.out.println("*** ERROR - exception(" +
                    exception.getMessage() + ")");
            }
        }//main()
    }//class ServiceInd
```

How It Works

This simple example declares a class, `ServiceInd`, in which the recipient address, push message ID string, PPG URL, and SI URI are declared as constants:

```
static final String address = "jdoe_devgate2.openwave.com" +
    "/TYPE=USER@ppg.openwave.com";
static final String pushID = "9f1000a023@openwave.com";
static final String ppgAddress =
    "http://devgate2.openwave.com:9002/pap";
static final String SvcIndURI =
    "http://devgate2.openwave.com/cgi-bin/mailbox.cgi";
```

A much better approach would be to extract the PPG and client addresses from the HTTP request headers to the PPG or from a user interface. For more information on the former approach, see "Extracting PPG and Client Addresses from PPG Headers" on page 23, "Sending a Push Submission" on page 30, and "Travel Example" on page 65.

Every Push Submission, regardless of the payload type, must include a unique identifier. The identifier distinguishes one Push Submission from all others, a critical factor if you want to cancel or retrieve status information for a Push Submission. The `PushMessage`, `CancelMessage`, and `StatusQueryMessage` class constructors define this parameter as a `java.lang.String` object. Assign each Push Submission a unique identifier that includes the domain name of the push initiator, as in the following examples:

```
2903011435@www.openwave.com

www.openwave.com/2903011435
```

See *WAP Push Access Protocol* for more information and examples.

The first two lines in the `SubmitMsg` method instantiate and initialize the PPG object. The next two lines set the text to be displayed on the client device when the SI is received and instantiate the SI object:

```
Pusher ppg = new Pusher(ppgURL);
ppg.initialize();

String alertText = "Mobile Mail: New message!";
ServiceIndication serviceIndication = new ServiceIndication(alertText);
```

The next line sets the value of the optional `href` attribute, which specifies the URI used to access the indicated service. Not calling this method indicates that the Service Indication is a simple notification for display on the client device:

```
serviceIndication.setHref(siURI);
```

Notice that the values of the *`ppgURL`* and *`siURI`* parameters are set in the `main` method.

Of the next three lines, the first two define two optional `info` elements for the SI, and the third adds them to the SI. The `info` element includes additional information not provided by the attributes of the `indication` element. The `info` element contains one or more `item` elements that specify the additional information. Each `item` element includes a `class` attribute that describes the content of the `item` element. How a client uses this information depends on the implementation. See `ServiceIndicationInfo` in the accompanying JavaDoc API documentation for more information.

```
ServiceIndicationInfo info = new ServiceIndicationInfo("Mailbox",
    "Full");
info.AddInfoBlock("ReadMessages", "All");
serviceIndication.setInfo(info);
```

The next line sets the SI `action` attribute, which contains text that specifies the action to be taken when the addressee receives an SI. In this case, the action is `signalHigh`, which instructs the PPG to present the SI as soon as implementation and bandwidth allow:

```
serviceIndication.setAction(ServiceIndicationAction.signalHigh);
```

The next two lines set optional time information attributes for the SI. The first line specifies the time when an SI expires and should be automatically deleted. Not calling this method indicates that the SI has no expiration time and is therefore not subject to automatic deletion. The second line specifies the time of creation or last modification of the content indicated by the service URI. This time may differ from the time of creation of the SI.

```
serviceIndication.setExpires(
    new DateTime(2004, 6, 15, 12, 0, 1));
serviceIndication.setCreated(
    new DateTime("2002-06-15T12:00:00Z"));
```

The next line instantiates a `PushMessage` object. This object builds the Push Submission from the push message ID string and the recipient address, both of which are passed to the class constructor as parameters:

```
PushMessage pushMessage = new PushMessage(pushID, address);
```

The next line sets some of the available optional information for the push message. In this case, the `setDeliverBeforeTimestamp` method sets the date and time before which the push message is to be delivered:

```
pushMessage.setDeliverBeforeTimestamp(
    new DateTime("2004-06-15T12:00:01Z"));
```

NOTE The time you set here must be at least 30 minutes following submission of the push message.

The next block of lines builds a miltipart MIME entity containing the SI content, encapsulated in a `MimeEntity` object, and sends the Push Submission to the PPG, while at the same time instantiating an object to hold the response from the PPG:

```
MimeEntity me = new MimeEntity();
me.addEntity(pushMessage);
me.addEntity(serviceIndication);

PushResponse pushResponse = (PushResponse) ppg.send(me);
```

Notice that the response object must be typecast to the `PushResponse` class. This is because the `Pusher.send` method returns a `PAPResponse` type which must then be cast to the desired response type for the current push operation.

The remaining lines in the `SubmitMsg` method print some of the pertinent information returned from the PPG in the form of a `PushResponse` object instance, as in the previous examples.

The `main` method instantiates the PPG URL and SI URI objects, instantiates a `ServiceInd` object, and calls the `SubmitMsg` method:

```
try {
    ppgURL = new URL(ppgAddress);
    siURI  = new URL(SvcIndURI);
    ServiceInd si = new ServiceInd();
    si.SubmitMsg();
}
```

Notice that these four lines are enclosed in a `try` block. The `catch` blocks that make up the remainder of the `main` method handle any exceptions that might occur, although only by printing error text to the console. A more sophisticated application could implement features to deal with some classes of exceptions automatically. Regardless of the level of exception handling that you build into your applications, you should always use `try...catch` blocks to isolate and report all exceptions.

Service Loading Payload Example

This example demonstrates how to send a Push Submission with a Service Loading (SL) payload. This content type causes a user agent on a mobile client to load and execute a service that, for example, can be in the form of a WML deck. The SL contains a URI specifying the service the user agent is to load, with or without user intervention as appropriate.

This example uses some of the same constants and other code as the previous examples, but demonstrates another complex payload type.

What It Does

The `ServiceLoad.java` file imports all the necessary files, including the WAP Push Library. Several constant fields define the recipient address, the ID string of the Push Submission, the URL of the PPG, and the SL URI. The public `ServiceLoad` class encapsulates all of the program's functionality in two methods:

- The `SubmitMsg` method instantiates the necessary WAP Push Library objects, sends the Push Submission, and prints some of the pertinent information from the response object returned from the PPG.
- The `main` method calls the `SubmitMsg` method and handles any exceptions returned from the PPG.

The exception handling demonstrated in this example is limited to displaying error messages on the console. It would be better, however, to build a more automated-exception handling and response mechanism into any applications you write.

A complete code listing follows.

ServiceLoad.java

```
/*
 * Title: WAP Push Library Service Loading Payload Example
 * Description: A basic Push Submission example using a
 * Service Loading payload
 */

import java.io.IOException;
import java.io.FileNotFoundException;
import java.net.MalformedURLException;
import java.net.URL;
import com.openwave.wappush.*;

public class ServiceLoad {
    //Constants used in this example.
    static final String address = "jdoe_devgate2.openwave.com" +
        "/TYPE=USER@ppg.openwave.com";
    static final String pushID = "9f1000a024@openwave.com";
    static final String ppgAddress =
        "http://devgate2.openwave.com:9002/pap";
    static final String SvcLoadURI =
        "http://devgate2.openwave.com/cgi-bin/mailbox.wml";

    static URL ppgURL = null;
    static URL slURI  = null;

    public void SubmitMsg() throws WapPushException, IOException,
        MalformedURLException {
        //Instantiate the Pusher and ServiceLoading objects;
        //initialize the Pusher object.
        Pusher ppg = new Pusher(ppgURL);
        ServiceLoading serviceLoading = new ServiceLoading(slURI);
        ppg.initialize();

        //Set the execute-low action.
        serviceLoading.setAction(ServiceLoadingAction.executeLow);

        //Instantiate the push message object and set some
        //optional information.
        PushMessage pushMessage = new PushMessage(pushID, address);
        pushMessage.addAddress("1234567890/" +
            "TYPE=PLMN@ppg.openwave.com");
        pushMessage.setDeliverBeforeTimestamp(new
            DateTime("2004-06-15T12:00:00Z"));

        //Instantiate a Quality of Service (QOS) object for the Push
        //message.
        QualityOfService pm_qos = new QualityOfService();

        //Set the desired QOS attributes.
        pm_qos.setDeliveryMethod(DeliveryMethod.confirmed);
        pm_qos.setPriority(DeliveryPriority.high);
        pm_qos.setNetwork("GSM");
        pm_qos.setNetworkRequired(true);
        pm_qos.setBearer("USSD");
        pm_qos.setBearerRequired(false);

        //Set the QOS for the push message.
```

```
            pushMessage.setQualityOfService(pm_qos);

            //Instantiate a MimeEntity and add the PushMessage and
            //ServiceLoading objects.
            MimeEntity me = new MimeEntity();
            me.addEntity(pushMessage);
            me.addEntity(serviceLoading);

            //Send the push message contained in the MimeEntity.
            PushResponse pushResponse = (PushResponse) ppg.send(me);

            //Read some information from the response.
            System.out.println("reply-Time = " +
                pushResponse.getReplyTime());
            System.out.println("response-result-code = " +
                pushResponse.getResultCode());
            System.out.println("response-result-desc = " +
                pushResponse.getResultDesc());
        }//SubmitMsg()

        public static void main(String[] args) throws WapPushException,
            IOException {
            try {
                ppgURL = new URL(ppgAddress);
                slURI  = new URL(SvcLoadURI);
                ServiceLoad sl = new ServiceLoad();
                sl.SubmitMsg();
            }
            //Handle possible exceptions.
            catch (BadMessageException exception) {
                System.out.println("*** ERROR - bad message exception");
                BadMessageResponse response = exception.getResponse();
                System.out.println("*** ERROR = " +
                    response.getBadMessageFragment());
            }
            catch (WapPushException exception) {
                System.out.println("*** ERROR - WapPushException (" +
                    exception.getMessage() + ")");
            }
            catch (FileNotFoundException exception) {
                System.out.println("*** ERROR - input file not found");
            }
            catch (MalformedURLException exception) {
                System.out.println("*** ERROR - malformed PPG URL");
            }
            catch (IOException exception){
                System.out.println( "*** ERROR - I/O exception");
            }
            catch (Exception exception) {
                System.out.println("*** ERROR - exception(" +
                    exception.getMessage() + ")");
            }
        }//main()
    }//class ServiceLoad
```

How It Works

This simple example declares a class, `ServiceLoad`, in which the recipient address, push message ID string, PPG URL, and SL URI are declared as constants:

```
static final String address = "jdoe_devgate2.openwave.com" +
    "/TYPE=USER@ppg.openwave.com";
static final String pushID = "9f1000a024@openwave.com";
static final String ppgAddress =
    "http://devgate2.openwave.com:9002/pap";
static final String SvcLoadURI =
    "http://devgate2.openwave.com/cgi-bin/mailbox.wml";
```

A much better approach would be to extract the PPG and client addresses from the HTTP request headers to the PPG or from a user interface. For more information on the former approach, see "Extracting PPG and Client Addresses from PPG Headers" on page 23, "Sending a Push Submission" on page 30, and "Travel Example" on page 65.

Every Push Submission, regardless of the payload type, must include a unique identifier. The identifier distinguishes one Push Submission from all others, a critical factor if you want to cancel or retrieve status information for a Push Submission. The `PushMessage`, `CancelMessage`, and `StatusQueryMessage` class constructors define this parameter as a `java.lang.String` object. Assign each Push Submission a unique identifier that includes the domain name of the push initiator, as in the following examples:

```
2903011435@www.openwave.com

www.openwave.com/2903011435
```

See *WAP Push Access Protocol* for more information and examples.

The first three lines in the `SubmitMsg` method instantiate and initialize the Pusher and SL objects:

```
Pusher ppg = new Pusher(ppgURL);
ServiceLoading serviceLoading = new ServiceLoading(slURI);
ppg.initialize();
```

Notice that the values of the *`ppgURL`* and *`slURI`* parameters are set in the `main` method.

The next line sets the SL `action` attribute to `executeLow`, which indicates that the service identified by the URI is loaded in the same way the user agent generally performs end-user-initiated method requests. This generally means that the service content is retrieved either from an origin server or from the client device cache, if available. After successfully completing the method request, the user agent loads the indicated service into a clean user agent context and executes it in a nonintrusive manner:

```
serviceLoading.setAction(ServiceLoadingAction.executeLow);
```

The next line instantiates a `PushMessage` object. This object builds the Push Submission from the push message ID string and the recipient address, both of which are passed to the class constructor as parameters:

```
PushMessage pushMessage = new PushMessage(pushID, address);
```

The next two lines set some of the available optional information for the push message; in this case, an additional recipient address and the date and time before which the push message is to be delivered:

```
pushMessage.addAddress(
    "1234567890/TYPE=PLMN@ppg.openwave.com");
pushMessage.setDeliverBeforeTimestamp(
    new DateTime("2002-06-15T12:00:01Z"));
```

The next line instantiates a Quality of Service (QOS) object for this Push Submission. The `quality-of-service` element specifies the delivery qualities that the push initiator expects for a push message:

```
QualityOfService pm_qos = new QualityOfService();
```

The next several lines specify the desired QOS attributes for this Push Submission. The first line sets the value of the optional `delivery-method` attribute to `confirmed`, which specifies that the PPG is required to use confirmed delivery of the push message to the recipient. The push initiator can request confirmed delivery without setting a URL for the `ppg-notify-requested-to` attribute of the `push-message` element, as in this example. The result is that the push message is confirmed over the air, but the PPG does not send confirmation to the push initiator. Call the `PushMessage.setPpgNotifyRequestedTo` method to set a URL for this attribute.

```
pm_qos.setDeliveryMethod(DeliveryMethod.confirmed);
```

The next line sets the value of the optional `priority` attribute, which specifies that the delivery priority for this push message is `high`. The way in which the PPG handles delivery priority is implementation specific:

```
pm_qos.setDeliveryPriority(high);
```

The next four lines set the network and bearer for the Push Submission. The network is required, as indicated by the `setNetworkRequired` method. The bearer is optional, as indicated by the `setBearerRequired` method.

```
pm_qos.setNetwork("GSM");
pm_qos.setNetworkRequired(true);
pm_qos.setBearer("USSD");
pm_qos.setBearerRequired(false);
```

The next line adds the defined QOS elements to the push message object instance:

```
pushMessage.setQualityOfService(pm_qos);
```

You can query the QOS response from the PPG, which indicates the delivery qualities that the PPG used during delivery of a push message. You can also send a Status Query message, referencing the desired push message, to obtain the same information.

The next block of lines builds a miltipart MIME entity containing the SL content, encapsulated in a `MimeEntity` object, and sends the Push Submission to the PPG, while at the same time instantiating an object to hold the response from the PPG:

```
MimeEntity me = new MimeEntity();
me.addEntity(pushMessage);
me.addEntity(serviceLoading);

PushResponse pushResponse = (PushResponse) ppg.send(me);
```

Notice that the response object must be typecast to the `PushResponse` class. This is because the `Pusher.send` method returns a `PAPResponse` type which must then be cast to the desired response type for the current push operation.

The remaining lines in the `SubmitMsg` method print some of the pertinent information returned from the PPG in the form of a `PushResponse` object instance, as in the previous examples.

The `main` method instantiates the PPG URL and SL URI objects, instantiates a `ServiceLoad` object, and calls the `SubmitMsg` method:

```
try {
    ppgURL = new URL(ppgAddress);
    slURI  = new URL(SvcLoadURI);
    ServiceLoad sl = new ServiceLoad();
    sl.SubmitMsg();
}
```

Notice that these four lines are enclosed in a `try` block. The `catch` blocks that make up the remainder of the `main` method handle any exceptions that might occur, although only by printing error text to the console. A more sophisticated application could implement features to deal with some classes of exceptions automatically. Regardless of the level of exception handling that you build into your applications, you should always use `try...catch` blocks to isolate and report all exceptions.

Cache Operation Payload Example

This example demonstrates how to send a Push Submission with a Cache Operation (CO) payload. This content type invalidates content objects in the user agent cache. All invalidated content objects must be reloaded from the server on which they originated the next time they are accessed.

This example uses some of the same constants and other code as the previous example, but demonstrates additional push message options and a more complex payload type.

What It Does

The `CacheOp.java` file imports all the necessary files, including the WAP Push Library. Several constant fields define the recipient address, the ID string of the Push Submission, the URL of the PPG, and the CO URI. The public `CacheOp` class encapsulates all of the program's functionality in two methods:

- The `SubmitMsg` method instantiates the necessary WAP Push Library objects, sends the Push Submission, and prints some of the pertinent information from the response object returned from the PPG.
- The `main` method calls the `SubmitMsg` method and handles any exceptions returned from the PPG.

The exception handling demonstrated in this example is limited to displaying error messages on the console. It would be better, however, to build a more automated exception-handling and response mechanism into any applications you write.

A complete code listing follows.

CacheOp.java

```
/*
 * Title: WAP Push Library Cache Operation Payload Example
 * Description: A basic Push Submission example using a
 * Cache Operation payload
 */

import java.io.IOException;
import java.io.FileNotFoundException;
import java.net.MalformedURLException;
import java.net.URL;
import com.openwave.wappush.*;

public class CacheOp {
    //Constants used in this example.
    static final String address = "jdoe_devgate2.openwave.com" +
        "/TYPE=USER@ppg.openwave.com";
    static final String pushID = "9f1000a022@openwave.com";
    static final String ppgAddress =
        "http://devgate2.openwave.com:9002/pap";
    static final String cacheOpURI =
        "http://devgate2.openwave.com/cgi-bin/mailbox.cgi";

    static URL ppgURL = null;
    static URL coURI = null;

    public void SubmitMsg() throws WapPushException, IOException,
        MalformedURLException {
        //Instantiate and initialize the Pusher and Cache Operation
        //objects.
        Pusher ppg = new Pusher(ppgURL);
        ppg.initialize();
        CacheOperation cacheOperation = new
            CacheOperation(coURI, CacheOperationType.cachedService);

        //Instantiate the push message object and set some
        //optional information.
        PushMessage pushMessage = new PushMessage(pushID, address);
        pushMessage.addAddress("jsmith_devgate2.openwave.com" +
            "/TYPE=USER@ppg.openwave.com");
        pushMessage.setDeliverAfterTimestamp(new
            DateTime("2002-06-15T12:00:00Z"));

        //Instantiate a MimeEntity object and add the PushMessage and
        //CacheOperation objects.
        MimeEntity me = new MimeEntity();
        me.addEntity(pushMessage);
        me.addEntity(cacheOperation);

        //Send the push message contained in the MimeEntity.
        PushResponse pushResponse = (PushResponse) ppg.send(me);

        //Read some information from the response.
```

```
            System.out.println("reply-Time = " +
                pushResponse.getReplyTime());
            System.out.println("response-result-code = " +
                pushResponse.getResultCode());
            System.out.println("response-result-desc = " +
                pushResponse.getResultDesc());
        }//SubmitMsg()

        public static void main(String[] args) throws WapPushException,
            IOException {
            try {
                ppgURL = new URL(ppgAddress);
                coURI  = new URL(cacheOpURI);
                CacheOp co = new CacheOp();
                co.SubmitMsg();
            }
            //Handle possible exceptions.
            catch (BadMessageException exception) {
                System.out.println("*** ERROR - bad message exception");
                BadMessageResponse response = exception.getResponse();
                System.out.println("*** ERROR = " +
                    response.getBadMessageFragment());
            }
            catch (WapPushException exception) {
                System.out.println("*** ERROR - WapPushException (" +
                    exception.getMessage() + ")");
            }
            catch (FileNotFoundException exception) {
                System.out.println("*** ERROR - input file not found");
            }
            catch (MalformedURLException exception) {
                System.out.println("*** ERROR - malformed PPG URL");
            }
            catch (IOException exception) {
                System.out.println( "*** ERROR - I/O exception");
            }
            catch (Exception exception) {
                System.out.println("*** ERROR - exception("
                    + exception.getMessage() + ")");
            }
        }//main()
    }//class CacheOp
```

How It Works

This simple example declares a class, `CacheOp`, in which the recipient address, push message ID string, PPG URL, and CO URI are declared as constants:

```
static final String address = "jdoe_devgate2.openwave.com" +
    "/TYPE=USER@ppg.openwave.com";
static final String pushID = "9f1000a022@openwave.com";
static final String ppgAddress =
    "http://devgate2.openwave.com:9002/pap";
static final String cacheOpURI =
    "http://devgate2.openwave.com/cgi-bin/mailbox.cgi";
```

A much better approach would be to extract the PPG and client addresses from the HTTP request headers to the PPG or from a user interface. For more information on the former approach, see "Extracting PPG and Client Addresses from PPG Headers" on page 23, "Sending a Push Submission" on page 30, and "Travel Example" on page 65.

Every Push Submission, regardless of the payload type, must include a unique identifier. The identifier distinguishes one Push Submission from all others, a critical factor if you want to cancel or retrieve status information for a Push Submission. The `PushMessage`, `CancelMessage`, and `StatusQueryMessage` class constructors define this parameter as a `java.lang.String` object. Assign each Push Submission a unique identifier that includes the domain name of the push initiator, as in the following examples:

```
2903011435@www.openwave.com

www.openwave.com/2903011435
```

See *WAP Push Access Protocol* for more information and examples.

The first three lines in the `SubmitMsg` method instantiate and initialize the `Pusher` and CO objects:

```
Pusher ppg = new Pusher(ppgURL);
ppg.initialize();
CacheOperation cacheOperation = new CacheOperation(coURI,
    CacheOperationType.cachedService);
```

Notice that the values of the *`ppgURL`* and *`coURI`* parameters are set in the `main` method. The *`cachedService`* parameter is an enumerated type, defined in the `CacheOperationType` class, which specifies the type of Cache Operation to be performed. See `CacheOperationType` in the accompanying JavaDoc API documentation for more information.

The next line instantiates a `PushMessage` object. This object builds the Push Submission from the push message ID string and the recipient address, both of which are passed to the class constructor as parameters:

```
PushMessage pushMessage = new PushMessage(pushID, address);
```

The next two lines set some of the available optional information for the push message; in this case, an additional recipient address and the date and time after which the push message is to be delivered:

```
pushMessage.addAddress("jsmith_devgate2.openwave.com" +
    "/TYPE=USER@ppg.openwave.com";");
pushMessage.setDeliverAfterTimestamp(new
    DateTime("2002-06-15T12:00:00Z"));
```

NOTE The time you set here must be no more than seven days following submission of the push message.

The next block of lines builds a miltipart MIME entity containing the Cache Operation content, encapsulated in a `MimeEntity` object, and sends the Push Submission to the PPG, while at the same time instantiating an object to hold the response from the PPG:

```
MimeEntity me = new MimeEntity();
me.addEntity(pushMessage);
me.addEntity(cacheOperation);

PushResponse pushResponse = (PushResponse) ppg.send(me);
```

Notice that the response object must be typecast to the `PushResponse` class. This is because the `Pusher.send` method returns a `PAPResponse` type which must then be cast to the desired response type for the current push operation.

A great deal of additional information can be extracted from the response, if desired. The final three lines in the `SubmitMsg` method print some of the pertinent information returned from the PPG in the form of a `PushResponse` object instance.

```
System.out.println("reply-Time = " + pushResponse.getReplyTime());
System.out.println("response-result-code = " +
    pushResponse.getResultCode());
System.out.println("response-result-desc = " +
    pushResponse.getResultDesc());
```

The `main` method instantiates the PPG URL and CO URI objects, instantiates a `CacheOp` object, and calls the `SubmitMsg` method:

```
try {
    ppgURL = new URL(ppgAddress);
    coURI  = new URL(cacheOpURI);
    CacheOp co = new CacheOp();
    co.SubmitMsg();
}
```

Notice that these four lines are enclosed in a `try` block. The `catch` blocks that make up the remainder of the `main` method handle any exceptions that might occur, although only by printing error text to the console. A more sophisticated application could implement features to deal with some classes of exceptions automatically. Regardless of the level of exception handling that you build into your applications, you should always use `try...catch` blocks to isolate and report all exceptions.

Custom Content Payload Example

This example demonstrates how to send a Push Submission with a Custom Content payload. This content type can contain virtually any type of content. This example demonstrates the essentials of building and sending a Push Submission.

What It Does

The `Custom.java` file imports all the necessary files, including the WAP Push Library. Several constant fields define the recipient address, the ID string of the Push Submission, and the URL of the PPG. The public `Custom` class encapsulates all of the program's functionality in two methods:

- The `SubmitMsg` method instantiates the necessary WAP Push Library objects, sends the Push Submission, and prints some of the pertinent information from the response object returned from the PPG.
- The `main` method calls the `SubmitMsg` method and handles any exceptions returned from the PPG.

The exception handling demonstrated in this example is limited to displaying error messages on the console. It would be better, however, to build a more automated exception-handling and response mechanism into any applications you write.

A complete code listing follows.

Custom.java

```
/*
 * Title: WAP Push Library Custom Content Payload Example
 * Description: A basic Push Submission example using a
 * Custom Content payload
 */

import java.io.IOException;
import java.io.FileNotFoundException;
import java.net.MalformedURLException;
import java.net.URL;
import com.openwave.wappush.*;

public class Custom {
    //Constants used in this example.
    static final String address = "jdoe_devgate2.openwave.com" +
        "/TYPE=USER@ppg.openwave.com";
    static final String pushID = "9f1000a022@openwave.com";
    static final String ppgAddress =
        "http://devgate2.openwave.com:9002/pap";
    static final String message = "Happy Birthday!!";

    static URL ppgURL = null;

    public void SubmitMsg() throws WapPushException, IOException,
        MalformedURLException {
        //Instantiate and initialize the Pusher and CustomContent
        //objects.
        Pusher ppg = new Pusher(ppgURL);
        ppg.initialize();
        CustomContent cc = new CustomContent();
        cc.setContentData(message, "text/plain");

        //Instantiate the push message object.
        PushMessage pushMessage = new PushMessage(pushID, address);

        //Instantiate a MimeEntity object and add the PushMessage and
        //CustomContent objects.
        MimeEntity me = new MimeEntity();
        me.addEntity(pushMessage);
        me.addEntity(cc);

        //Send the push message contained in the MimeEntity.
        PushResponse pushResponse = (PushResponse) ppg.send(me);

        //Read some information from the response.
        System.out.println("reply-Time = " +
            pushResponse.getReplyTime());
        System.out.println("response-result-code = " +
            pushResponse.getResultCode());
        System.out.println("response-result-desc = " +
            pushResponse.getResultDesc());
    }//SubmitMsg()
```

```
    public static void main(String[] args) throws WapPushException,
        IOException {
        try {
            ppgURL = new URL(ppgAddress);
            Custom custContent = new Custom();
            custContent.SubmitMsg();
        }
        //Handle possible exceptions.
        catch (BadMessageException exception) {
            System.out.println("*** ERROR - bad message exception");
            BadMessageResponse response = exception.getResponse();
            System.out.println("*** ERROR = " +
                response.getBadMessageFragment());
        }
        catch (WapPushException exception) {
            System.out.println("*** ERROR - WapPushException (" +
                exception.getMessage() + ")");
        }
        catch (FileNotFoundException exception) {
            System.out.println("*** ERROR - input file not found");
        }
        catch (MalformedURLException exception) {
            System.out.println("*** ERROR - malformed PPG URL");
        }
        catch (IOException exception) {
            System.out.println( "*** ERROR - I/O exception");
        }
        catch (Exception exception) {
            System.out.println("*** ERROR - exception("
                + exception.getMessage() + ")");
        }
    }//main()
}//class Custom
```

How It Works

This simple example declares a class, `Custom`, in which the recipient address, push message ID string, and PPG URL are declared as constants:

```
static final String address = "jdoe_devgate2.openwave.com" +
    "/TYPE=USER@ppg.openwave.com";
static final String pushID = "9f1000a022@openwave.com";
static final String ppgAddress =
    "http://devgate2.openwave.com:9002/pap";
static final String message = "Happy Birthday!!";
```

A much better approach would be to extract the PPG and client addresses from the HTTP request headers to the PPG or from a user interface. For more information on the former approach, see "Extracting PPG and Client Addresses from PPG Headers" on page 23, "Sending a Push Submission" on page 30, and "Travel Example" on page 65.

Every Push Submission, regardless of the payload type, must include a unique identifier. The identifier distinguishes one Push Submission from all others, a critical factor if you want to cancel or retrieve status information for a Push Submission. The `PushMessage`, `CancelMessage`, and `StatusQueryMessage` class constructors define this parameter as a `java.lang.String` object. Assign each Push Submission a unique identifier that includes the domain name of the push initiator, as in the following examples:

```
2903011435@www.openwave.com

www.openwave.com/2903011435
```

See *WAP Push Access Protocol* for more information and examples.

The first four lines in the `SubmitMsg` method instantiate and initialize the `Pusher` object, instantiate the Custom Content object, and set the content for the Custom Content object:

```
Pusher ppg = new Pusher(ppgURL);
ppg.initialize();
CustomContent cc = new CustomContent();
cc.setContentData(message, "text/plain");
```

Notice that the value of the *`ppgURL`* parameter is set in the `main` method. The `Mediatype` object defines the content type, while the `CustomContent.setContentData` method sets the content. See `CustomContent` in the accompanying JavaDoc API documentation for more information.

The next line instantiates a `PushMessage` object. This object builds the Push Submission from the push message ID string and the recipient address, both of which are passed to the class constructor as parameters:

```
PushMessage pushMessage = new PushMessage(pushID, address);
```

The next block of lines builds a miltipart MIME entity containing the Custom Content, encapsulated in a `MimeEntity` object, and sends the Push Submission to the PPG, while at the same time instantiating an object to hold the response from the PPG:

```
MimeEntity me = new MimeEntity();
me.addEntity(pushMessage);
me.addEntity(cc);

PushResponse pushResponse = (PushResponse) ppg.send(me);
```

Notice that the response object must be typecast to the `PushResponse` class. This is because the `Pusher.send` method returns a `PAPResponse` type which must then be cast to the desired response type for the current push operation.

A great deal of additional information can be extracted from the response, if desired. The final three lines in the `SubmitMsg` method print some of the pertinent information returned from the PPG in the form of a `PushResponse` object instance.

```
System.out.println("reply-Time = " + pushresponse.getReplyTime());
System.out.println("response-result-code = " +
    pushresponse.getResultCode());
System.out.println("response-result-desc = " +
    pushresponse.getResultDesc());
```

The `main` method instantiates the PPG URL object, instantiates a `CustomContent` object, and calls the `SubmitMsg` method:

```
try {
    ppgURL = new URL(ppgAddress);
    Custom custContent = new Custom();
    custContent.SubmitMsg();
}
```

Notice that these lines are enclosed in a `try` block. The `catch` blocks that make up the remainder of the `main` method handle any exceptions that might occur, although only by printing error text to the console. A more sophisticated application could implement features to deal with some classes of exceptions automatically. Regardless of the level of exception handling that you build into your applications, you should always use `try...catch` blocks to isolate and report all exceptions.

Status Query Message and Response Example

This example demonstrates how to send a Status Query message, which requests the current status of a Push Submission from the PPG. In addition to the standard numeric code and text message, the Push Status Query response also contains an attribute that indicates the status of the specified Push Submission. Nine message states are currently specified:

- `aborted` The addressee aborted the message.
- `cancelled` A Push Cancellation successfully canceled the message.
- `delivered` The PPG successfully delivered the message to the addressee.
- `expired` The message reached the maximum age allowed by PPG policy or could not be delivered by the time specified in the Push Submission.
- `pending` The PPG accepted the message and is in the process of delivering it.
- `rejected` The addressee rejected the message.
- `timeout` The delivery process timed out on the PPG.
- `undeliverable` A problem prevented delivery of the message. Call the `StatusQueryResult.getCode` and `StatusQueryResult.getDesc` methods to retrieve the code and text message returned from the PPG.
- `unknown` The PPG has no information about the status of the message.

See *WAP Push Access Protocol* for definitions of the numeric codes and accompanying text messages.

NOTE PPG implementation of Status Query reporting is optional. If the PPG does not support Status Query reporting, the response contains status code 3001 with the text message Not Implemented.

What It Does

The `StatusQM.java` file imports all the necessary files, including the WAP Push Library. Three constant fields define the ID string of the Push Submission, the specific addressee for which to retrieve status information, and the URL of the PPG. Unless you specify otherwise, status for all addressees of the Push Submission identified in the `StatusQueryMessage` class constructor is returned. The public `StatusQM` class encapsulates all of the program's functionality in three methods:

- The `printStatusQueryResponse` method prints the pertinent information from the response object returned from the PPG.
- The `SubmitMsg` method instantiates the necessary WAP Push Library objects and sends the Status Query message.
- The `main` method calls the `SubmitMsg` method and handles any exceptions returned from the PPG.

The exception handling demonstrated in this example is limited to displaying error messages on the console. It would be better, however, to build a more automated exception-handling and response mechanism into any applications that you write. A complete code listing follows.

StatusQM.java

```
/*
 * Title: WAP Push Library Status Query Message Example
 * Description: A basic Status Query message/response example
 */

import java.io.IOException;
import java.io.FileNotFoundException;
import java.net.MalformedURLException;
import java.net.URL;
import com.openwave.wappush.*;

public class StatusQM {
    //Constants used in this example.
    static final String ppgAddress =
        "http://devgate2.openwave.com:9002/pap";
    static final String address = "jdoe_devgate2.openwave.com" +
        "/TYPE=USER@ppg.openwave.com";

    static URL ppgURL = null;

    //Unique identifier of the Push Submission about which to
    //retrieve status information.
    static final String pushID = "9f1000a024@openwave.com";

    //Private method that prints the response information.
    private static void printStatusQueryResponse (StatusQueryResponse
        response) {
        System.out.println("StatusQueryResponse");
        System.out.println("     push-id = " + response.getPushID());
        int resultCount = response.getResultCount();
        System.out.println("     result-count = " + resultCount);
        for (int i = 0; i < resultCount; ++i) {
            System.out.println("     result #" + i);
            StatusQueryResult result = response.getResult(i);
            MessageState mState = result.getMessageState();
            System.out.println("          message-state = " +
                mState.toString());
            System.out.println("          code = " +
                result.getCode());
            System.out.println("          description = " +
                result.getDesc());
            DateTime dt = result.getEventTime();
            System.out.println("          event-time = " +
                dt.toString());
            int addressCount = result.getAddressCount();
            System.out.println("          address-count = " +
                addressCount);

            for (int j = 0; j < addressCount; ++j) {
                System.out.println("          address #" + j +
                    " = " + result.getAddress(j));
            }
        }
    }//printStatusQueryResponse
```

```
    public void SubmitMsg() throws WapPushException, IOException,
        MalformedURLException {
        //Instantiate and intitialize the Pusher object.
        Pusher ppg = new Pusher(ppgURL);
        ppg.initialize();

        //Instantiate the StatusQueryMessage object.
        StatusQueryMessage queryMessage = new StatusQueryMessage(pushID);

        //Specific addressee for which to retrieve status information.
        queryMessage.addAddress(address);

        //Send the Status Query message.
        StatusQueryResponse queryResponse =
            (StatusQueryResponse) ppg.send(queryMessage);

        //Print the response.
        printStatusQueryResponse(queryResponse);
    }//SubmitMsg()

    public static void main(String[] args) throws WapPushException,
IOException {
        try {
            ppgURL = new URL(ppgAddress);
            StatusQM sqm = new StatusQM();
            sqm.SubmitMsg();
        }
        //Handle possible exceptions.
        catch (BadMessageException exception) {
            System.out.println("*** ERROR - bad message exception");
            BadMessageResponse response = exception.getResponse();
            System.out.println("*** ERROR = " +
                response.getBadMessageFragment());
        }
        catch (WapPushException exception) {
            System.out.println("*** ERROR - WapPushException (" +
                exception.getMessage() + ")");
        }
        catch (FileNotFoundException exception) {
            System.out.println("*** ERROR - input file not found");
        }
        catch (MalformedURLException exception) {
            System.out.println("*** ERROR - malformed PPG URL");
        }
        catch (IOException exception) {
            System.out.println( "*** ERROR - I/O exception");
        }
        catch (Exception exception) {
            System.out.println("*** ERROR - exception(" +
                exception.getMessage() + ")");
        }
    }//main()
}//class StatusQM
```

How It Works

This simple example declares a class, `StatusQM`, in which the push message ID string, specific addressee for which to retrieve status information, and PPG URL are declared as constants:

```
static final String pushID = "9f1000a024@openwave.com";
static final String address = "jdoe_devgate2.openwave.com" +
    "/TYPE=USER@ppg.openwave.com";
static final String ppgAddress =
    "http://devgate2.openwave.com:9002/pap";
```

A much better approach would be to extract the PPG and client addresses from the HTTP request headers to the PPG or from a user interface. For more information on the former approach, see "Extracting PPG and Client Addresses from PPG Headers" on page 23, "Sending a Push Submission" on page 30, and "Travel Example" on page 65.

The private `printStatusQueryResponse` method takes the response object returned from the PPG as a parameter. This method loops through all of the separate results returned from the PPG, typically one for each addressee for which message status is desired, and prints the pertinent information. In this example, only one set of results would be returned because the Status Query message defines a single addressee.

The first two lines in the `SubmitMsg` method instantiate and initialize the `Pusher` object:

```
Pusher ppg = new Pusher(ppgURL);
ppg.initialize();
```

Notice that the value of the *`ppgURL`* parameter is set in the `main` method.

The next two lines instantiate the Status Query message object and add the addressee for which to retrieve status information:

```
StatusQueryMessage queryMessage = new StatusQueryMessage(pushID);
queryMessage.addAddress(address);
```

The next line actually sends the Status Query message to the PPG, while at the same time instantiating an object to hold the response from the PPG:

```
StatusQueryResponse queryResponse =
    (StatusQueryResponse) ppg.send(queryMessage);
```

Notice that the response object must be typecast to the `StatusQueryResponse` class. This is because the `Pusher.send` method returns a `PAPResponse` type which must then be cast to the desired response type for the current push operation.

The last line in the `SubmitMsg` method calls the private `printStatusQueryResponse` method, which prints the pertinent information returned from the PPG in the form of a `StatusQueryResponse` object instance:

```
printStatusQueryResponse(queryResponse);
```

The `main` method instantiates the PPG URL object, instantiates a `StatusQM` object, and calls the `SubmitMsg` method:

```
try {
    ppgURL = new URL(ppgAddress);
    StatusQM sqm = new StatusQM();
    sqm.SubmitMsg();
}
```

Notice that these lines are enclosed in a `try` block. The `catch` blocks that make up the remainder of the `main` method handle any exceptions that might occur, although only by printing error text to the console. A more sophisticated application could implement features to deal with some classes of exceptions automatically. Regardless of the level of exception handling that you build into your applications, you should always use `try...catch` blocks to isolate and report all exceptions.

Push Cancel Message and Response Example

This example demonstrates how to send a Push Cancellation, which allows the push initiator to attempt to cancel a Push Submission, and how to read the response from the PPG, which indicates the status of the cancellation request. A Push Submission can be canceled only before it has been delivered.

> **NOTE** PPG support for the Push Cancellation operation is optional. If the PPG does not support Push Cancellation, the response contains status code 3001 with the text message Not Implemented.

What It Does

The `CancelPush.java` file imports all the necessary files, including the WAP Push Library. Two constant fields define the ID string of the Push Submission to be canceled and the URL of the PPG. Unless you specify otherwise, the Push Submission identified in the `CancelMessage` class constructor is canceled for all addressees. The public `CancelPush` class encapsulates all of the program's functionality in three methods:

- The `printCancelResponse` method prints the pertinent information from the response object returned from the PPG.
- The `SubmitMsg` method instantiates the necessary WAP Push Library objects and sends the Push Cancellation.
- The `main` method calls the `SubmitMsg` method and handles any exceptions returned from the PPG.

The exception handling demonstrated in the following example is limited to displaying error messages on the console. It would be better, however, to build a more automated exception-handling and response mechanism into any applications you write.

A complete code listing follows.

CancelPush.java

```
/*
 * Title: WAP Push Library Push Cancellation Example
 * Description: A basic Push Cancellation/response example
 */

import java.io.IOException;
import java.io.FileNotFoundException;
import java.net.MalformedURLException;
import java.net.URL;
import com.openwave.wappush.*;

public class CancelPush {
    //Constants used in this example.
    static final String ppgAddress =
        "http://devgate2.openwave.com:9002/pap";
    static URL ppgURL = null;

    //Unique identifier of the Push Submission to cancel.
    static final String pushID = "9f1000a024@openwave.com";

    //Private method that prints the response information.
    private static void printCancelResponse (CancelResponse response) {
        System.out.println("CancelResponse");
        System.out.println("     push-id = " + response.getPushID());
        int resultCount = response.getResultCount();
        System.out.println("     result-count = " + resultCount);

        for (int i = 0; i < resultCount; ++i) {
            System.out.println("     result #" + i);
            CancelResult result = response.getResult(i);
            System.out.println("          code = " + result.getCode());
            System.out.println("          description = " +
                result.getDesc());
            int addressCount = result.getAddressCount();
            System.out.println("          address-count = " +
                addressCount);

            for (int j = 0; j < addressCount; ++j) {
                System.out.println("          address #" + j + " = "
                    + result.getAddress(j));
            }
        }
    }//printCancelResponse

    public void SubmitMsg() throws WapPushException, IOException,
        MalformedURLException {
        //Instantiate the CancelMessage and Pusher objects; initialize
        //the Pusher object.
        CancelMessage cancelMsg = new CancelMessage(pushID);
        Pusher ppg = new Pusher(ppgURL);
        ppg.initialize();
```

```
            //Send the cancel request and read the response.
            CancelResponse cancelResponse =
                (CancelResponse) ppg.send(cancelMsg);
            printCancelResponse(cancelResponse);
        }

        public static void main(String[] args) throws WapPushException,
            IOException {
            try {
                ppgURL = new URL(ppgAddress);
                CancelPush cp = new CancelPush();
                cp.SubmitMsg();
            }
            //Handle possible exceptions.
            catch (BadMessageException exception) {
                System.out.println("*** ERROR - bad message exception");
                BadMessageResponse response = exception.getResponse();
                System.out.println("*** ERROR = " +
                    response.getBadMessageFragment());
            }
            catch (WapPushException exception) {
                System.out.println("*** ERROR - WapPushException (" +
                    exception.getMessage() + ")");
            }
            catch (FileNotFoundException exception) {
                System.out.println("*** ERROR - input file not found");
            }
            catch (MalformedURLException exception) {
                System.out.println("*** ERROR - malformed PPG URL");
            }
            catch (IOException exception) {
                System.out.println( "*** ERROR - I/O exception");
            }
            catch (Exception exception) {
                System.out.println("*** ERROR - exception(" +
                    exception.getMessage() + ")");
            }
        }//main()
    }//class CancelPush
```

How It Works

This simple example declares a class, `CancelPush`, in which the push message ID string (specifying the message to cancel) and PPG URL are declared as constants:

```
static final String pushID = "9f1000a024@openwave.com";
static final String ppgAddress =
    "http://devgate2.openwave.com:9002/pap"
```

A much better approach would be to extract the PPG and client addresses from the HTTP request headers to the PPG or from a user interface. For more information on the former approach, see "Extracting PPG and Client Addresses from PPG Headers" on page 23, "Sending a Push Submission" on page 30, and "Travel Example" on page 65.

The private `printCancelResponse` method takes the response object returned from the PPG as a parameter. This method loops through all of the separate results returned from the PPG, typically one for each addressee whose message is to be canceled, and prints the pertinent information.

The first three lines in the `SubmitMsg` method instantiate and initialize the `Pusher` and Push Cancellation objects:

```
CancelMessage cancelMsg = new CancelMessage(pushID);
Pusher ppg = new Pusher(ppgURL);
ppg.initialize();
```

Notice that the value of the *`ppgURL`* parameter is set in the `main` method.

The next line sends the Push Cancellation request to the PPG, while at the same time instantiating an object to hold the response from the PPG:

```
CancelResponse cancelResponse = (CancelResponse) ppg.send(cancelMsg);
```

Notice that the response object must be typecast to the `CancelResponse` class. This is because the `Pusher.send` method returns a `PAPResponse` type which must then be cast to the desired response type for the current push operation.

The last line in the `SubmitMsg` method calls the private `printCancelResponse` method, which prints the pertinent information returned from the PPG in the form of a `CancelResponse` object instance:

```
printCancelResponse(cancelResponse);
```

The `main` method instantiates the PPG URL object, instantiates a`CancelPush` object, and calls the`SubmitMsg` method:

```
try {
    ppgURL = new URL(ppgAddress);
    CancelPush cp = new CancelPush();
    cp.SubmitMsg();
}
```

Notice that these lines are enclosed in a `try` block. The `catch` blocks that make up the remainder of the `main` method handle any exceptions that might occur, although only by printing error text to the console. A more sophisticated application could implement features to deal with some classes of exceptions automatically. Regardless of the level of exception handling that you build into your applications, you should always use `try...catch` blocks to isolate and report all exceptions.

Client Capabilities Query Message and Response Example

This example demonstrates how to send a Client Capabilities Query (CCQ) message, which requests the capabilities for a specific client device from the PPG. A CCQ response also includes a complete User Agent Profile (UAProf), which provides detailed information about the specified client device.

> **NOTE** PPG implementation of CCQ reporting is optional. If the PPG does not support CCQ reporting, the response contains status code 3001 with the text message Not Implemented.

What it Does

The `ClientCaps.java` file imports all the necessary files, including the WAP Push Library. Two constant fields define the specific addressee for which to retrieve CCQ information and the URL of the PPG. The public `ClientCaps` class encapsulates all of the program's functionality in four methods:

- The `printCCQResponse` method prints the pertinent information from the CCQ response object returned from the PPG.
- The `printUAProfile` method prints UAProf information from the CCQ response object returned from the PPG.
- The `SubmitMsg` method instantiates the necessary WAP Push Library objects and sends the CCQ message.
- The `main` method calls the `SubmitMsg` method and handles any exceptions returned from the PPG.

The exception handling demonstrated in this example is limited to displaying error messages on the console. It would be better, however, to build a more automated exception-handling and response mechanism into any applications that you write.

A complete code listing follows.

ClientCaps.java

```
/*
 * Title: WAP Push Library CCQ Message Example
 * Description: A basic CCQ message/response example
 */

import java.io.IOException;
import java.io.FileNotFoundException;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Enumeration;
import com.openwave.wappush.*;

public class ClientCaps {
    //Constants used in this example.
    static final String ppgAddress =
        "http://devgate2.openwave.com:9002/pap";
    static final String address = "jdoe_devgate2.openwave.com" +
        "/TYPE=USER@ppg.openwave.com";

    static URL ppgURL = null;

    //Private methods that print the response information.
    private static void printCCQResponse(CcqResponse response) {
        System.out.println("ClientCapsQueryResponse");
        System.out.println("     result-code = " + response.getCode());
        System.out.println("     result-description = " +
            response.getDesc());
        System.out.println("     query-id = " + response.getQueryID());
        System.out.println("     address = " + response.getAddress());
        UAProfile profile = response.getUserAgentProfile();
        if (profile == null)
            System.out.println("UAProfile = null");
        else
            printUAProfile(profile);
    }//printCCQResponse

    private static void printUAProfile (UAProfile profile) {
        System.out.println("UAProfile");
        int compCount = profile.componentCount();
        System.out.println("     profile-id = " + profile.getID());
        System.out.println("     component-count = " + compCount);
        for (int i = 0; i < compCount; ++i) {
            UAComponent comp = profile.getComponent(i);
            System.out.println("          component #" + i);
            System.out.println("          component-id = " +
                comp.getID());
            System.out.print("          defaults-resource = ");
            try {
                URL defaultsURL = comp.getDefaultsResource();
                System.out.println(defaultsURL);
            }
            catch (Exception e) {
```

```
                System.out.println("**malformed URL**");
            }
            System.out.print(" schema-resource = ");

            try {
                URL schemaURL = comp.getSchemaResource();
                System.out.println(schemaURL);
            }
            catch (Exception e) {
                System.out.println("**malformed URL**");
            }

            Enumeration attNames = comp.getAttributeNames();
            while (attNames.hasMoreElements()) {
                String attName = (String) attNames.nextElement();
                System.out.println("              " + attName + " = ");
                Enumeration attValues = comp.getAllValuesOf(attName);
                while (attValues.hasMoreElements()) {
                    Object attValue = attValues.nextElement();
                    if (attValue instanceof Bag) {
                       Bag bag = (Bag) attValue;
                        int itemCount = bag.count();
                        System.out.print("                 (");
                        for (int j = 0; j < itemCount; ++j) {
                            if (j != 0)
                                System.out.print(", ");
                            System.out.print(bag.get(j));
                        }
                        System.out.println(")");
                    }
                    else {
                    System.out.println("                " + attValue);
                    }
                }
            }
        }
    }//printUAProf

    public void SubmitMsg() throws WapPushException, IOException,
        MalformedURLException {
        //Instantiate the CCQ message and Pusher objects, initiailze the
        //Pusher object, send the CCQ message, and print the response.
        CcqMessage ccqMessage = new CcqMessage(address);
        Pusher ppg = new Pusher(ppgURL);
        ppg.initialize();
        CcqResponse ccqResponse = (CcqResponse) ppg.send(ccqMessage);
        printCCQResponse(ccqResponse);
    }

    public static void main(String[] args) throws WapPushException,
        IOException {
        try {
            ppgURL = new URL(ppgAddress);
            ClientCaps ccq = new ClientCaps();
```

```
                ccq.SubmitMsg();
            }
            //Handle possible exceptions.
            catch (BadMessageException exception) {
                System.out.println("*** ERROR - bad message exception");
                BadMessageResponse response = exception.getResponse();
                System.out.println("*** ERROR = " +
                    response.getBadMessageFragment());
            }
            catch (WapPushException exception) {
                System.out.println("*** ERROR - WapPushException (" +
                    exception.getMessage() + ")");
            }
            catch (FileNotFoundException exception) {
                System.out.println("*** ERROR - input file not found");
            }
            catch (MalformedURLException exception) {
                System.out.println("*** ERROR - malformed PPG URL");
            }
            catch (IOException exception) {
                System.out.println( "*** ERROR - I/O exception");
            }
            catch (Exception exception) {
                System.out.println("*** ERROR - exception(" +
                    exception.getMessage() + ")");
            }
        }//main()
    }//class ClientCaps
```

How It Works

This simple example declares a class, `ClientCaps`, in which the client for which to retrieve CCQ information and the PPG URL are declared as constants:

```
static final String ppgAddress =
    "http://devgate2.openwave.com:9002/pap"
static final String address = "jdoe_devgate2.openwave.com" +
    "/TYPE=USER@ppg.openwave.com";
```

A much better approach would be to extract the PPG and client addresses from the HTTP request headers to the PPG or from a user interface. For more information on the former approach, see "Extracting PPG and Client Addresses from PPG Headers" on page 23, "Sending a Push Submission" on page 30, and "Travel Example" on page 65.

The private `printCCQResponse` method takes the response object returned from the PPG as a parameter. This method loops through all of the separate results returned from the PPG and prints the pertinent information. If the PPG returns UAProf information for the client device, the private `printUAProfile` method loops through and prints all of the UAProf information returned. See `UAProfile` in the accompanying JavaDoc API reference for more information on the methods called in this section.

The first three lines in the `SubmitMsg` method instantiate and initialize the `Pusher` and CCQ message objects:

```
CcqMessage ccqMessage = new CcqMessage(address);
Pusher ppg = new Pusher(ppgURL);
ppg.initialize();
```

Notice that the value of the *`ppgURL`* parameter is set in the `main` method.

The next line actually sends the CCQ message to the PPG, while at the same time instantiating an object to hold the response from the PPG:

```
CcqResponse ccqResponse = (CcqResponse) ppg.send(ccqMessage);
```

Notice that the response object must be typecast to the `CcqResponse` class. This is because the `Pusher.send` method returns a `PAPResponse` type which must then be cast to the desired response type for the current push operation.

The last line in the `SubmitMsg` method calls the private `printCCQResponse` method, which prints the pertinent information returned from the PPG in the form of a `CcqResponse` object instance:

```
printCCQResponse(ccqResponse);
```

If the PPG returns UAProf information, the private `printUAProfile` method loops through and prints all of the UAProf information returned.

The `main` method instantiates the PPG URL object, instantiates a `ClientCaps` object, and calls the `SubmitMsg` method:

```
try {
    ppgURL = new URL(ppgAddress);
    ClientCaps ccq = new ClientCaps();
    ccq.SubmitMsg();
}
```

Notice that these lines are enclosed in a `try` block. The `catch` blocks that make up the remainder of the `main` method handle any exceptions that might occur, although only by printing error text to the console. A more sophisticated application could implement features to deal with some classes of exceptions automatically. Regardless of the level of exception handling that you build into your applications, you should always use `try...catch` blocks to isolate and report all exceptions.

7 Debugging WAP Push Library Applications

Debugging WAP Push Library applications is a relatively straightforward process. There are four major areas to examine when debugging:

- Use your Java development tools to ensure that your Java code executes properly and performs the desired tasks without error.
- Always catch and examine the `WapPushException` class, which indicates that a parameter is missing, out of range, or specified improperly, or if a response from the PPG is missing one or more required elements.
- Always catch and examine the `UnknownMediaTypeException` class, which indicates that a method attempted to use an unknown media type. An unknown media type is a one that is not registered with the `MediaTypeRegistry` class.
- Catch and examine all other exceptions that your WAP Push Library application returns. The text message returned in the exception classes should provide you with enough information to solve any problems in your code. The WAP Push Library does not contain an internal list of error codes and messages.
- Examine the PPG response messages.

Debugging Java Code

All professional Java development environments supply sophisticated debugging tools. At a minimum, you can use these tools to:

- Set breakpoints and watches
- Step through your code line by line and examine the results as each line executes
- Trace into methods and examine them line by line as they execute
- Examine all threads
- View the call stack

For complete instructions on using your debugging tools, refer to the manuals accompanying your Java development environment.

Exception Handling

The WAP Push Library uses the `WapPushException` class to throw an exception if a parameter is missing, out of range, or specified improperly, or if a response from the PPG is missing one or more required elements. All `public` constructors and methods that require one or more parameters throw this exception, as do all of the message response classes. Your WAP Push Library applications should always catch and handle these exceptions, which can be very helpful to you during debugging.

The WAP Push Library generates a `BadMessageException` if a PAP message is garbled when the PPG receives it. In that case, the PPG returns the `<badmessage-response>` XML element with a message describing the cause of the exception and a fragment of the garbled submission. All of the WAP Push Library response classes are designed to instantiate a `BadMessageException` object and return the exception message in the response object.

The WAP Push Library throws an `UnknownMediaTypeException` if an attempt is made to use an unknown media type. An unknown media type is one that is not registered with the `MediaTypeRegistry` class.

Your WAP Push Library applications should always catch and handle these exceptions.

All other exceptions are handled by the various Java classes that your application imports. At a minimum, all WAP Push Library applications should import the following Java exception classes and handle any exceptions they throw:

```
java.io.IOException
java.io.FileNotFoundException
java.net.MalformedURLException
```

For complete exception-handling information, refer to the user manuals for your Java development environment.

Catching and Examining Exceptions

All WAP Push Library applications should catch and examine all exceptions. This section describes exception handling using both libraries.

WAP Push Library Exception Handling

Catching and examining all exceptions that your WAP Push Library application returns is a simple process. First, as mandated by good Java programming practice, make sure that all methods you write throw any possible exceptions, as in this example declaration statement:

```
public SubmitMsg() throws WapPushException, IOException,
    MalformedURLException, UnknownMediaTypeException
```

Then be sure to enclose in a `try` block all statements that might cause an exception to be thrown. Finally, use `catch` blocks to capture and examine any exceptions that might be thrown in the `try` block. Good Java programming practice also mandates the use of `try...catch` blocks. The example `main` method demonstrates a basic exception handling strategy:

```
public static void main(String[] args) throws WapPushException,
    IOException, MalformedURLException, UnknownMediaTypeException
{
    try {
        ppgURL = new URL(ppgAddress);
        CustomContent cc = new CustomContent();
        cc.SubmitMsg();
    }
    //Handle possible exceptions.
    catch (BadMessageException exception) {
        System.out.println("*** ERROR - bad message exception");
        BadMessageResponse response = exception.getResponse();
        System.out.println("*** ERROR = " +
            response.getBadMessageFragment());
    }
    catch (WapPushException exception) {
        System.out.println("*** ERROR - WapPushException (" +
            exception.getMessage() + ")");
    }
    catch (FileNotFoundException exception) {
        System.out.println("*** ERROR - input file not found");
    }
    catch (MalformedURLException exception) {
        System.out.println("*** ERROR - malformed PPG URL");
    }
    catch (IOException exception) {
        System.out.println( "*** ERROR - I/O exception");
    }
    catch (UnknownMediaTypeException exception) {
        System.out.println( "*** ERROR - Media Type not defined");
    }
    catch (Exception exception) {
        System.out.println("*** ERROR - exception(" +
            exception.getMessage() + ")");
    }
}//main()
```

Examining the PPG Response Message

In addition to handling exceptions, your WAP Push Library applications should always examine the response the PPG returns upon receiving a Push Submission. The WAP Push Library parses the XML document that forms the PPG response and encapsulates it in a class corresponding to the PAP operation type. For a complete description of the response message format and content, see "PPG Response Message" on page 25.

All of the examples in Chapter 5, "SimplePush Class Essentials," and in Chapter 6, "WAP Push Library Essentials," show how to use WAP Push Library methods to examine the PPG response object. Example code in the accompanying JavaDoc API reference also shows how to use these methods.

8 Tools and Utilities

This chapter describes a tool that helps you send and analyze Push Submissions: PushIT. The source code for this tool is included. Source code is located in the following directory.

- **PushIT** `<installroot>\examples\PushIT\src\java`

Using the Push Initiator Tool

The Push Initiator Tool (PushIT) provides a an easy way to generate push operations and view the results.

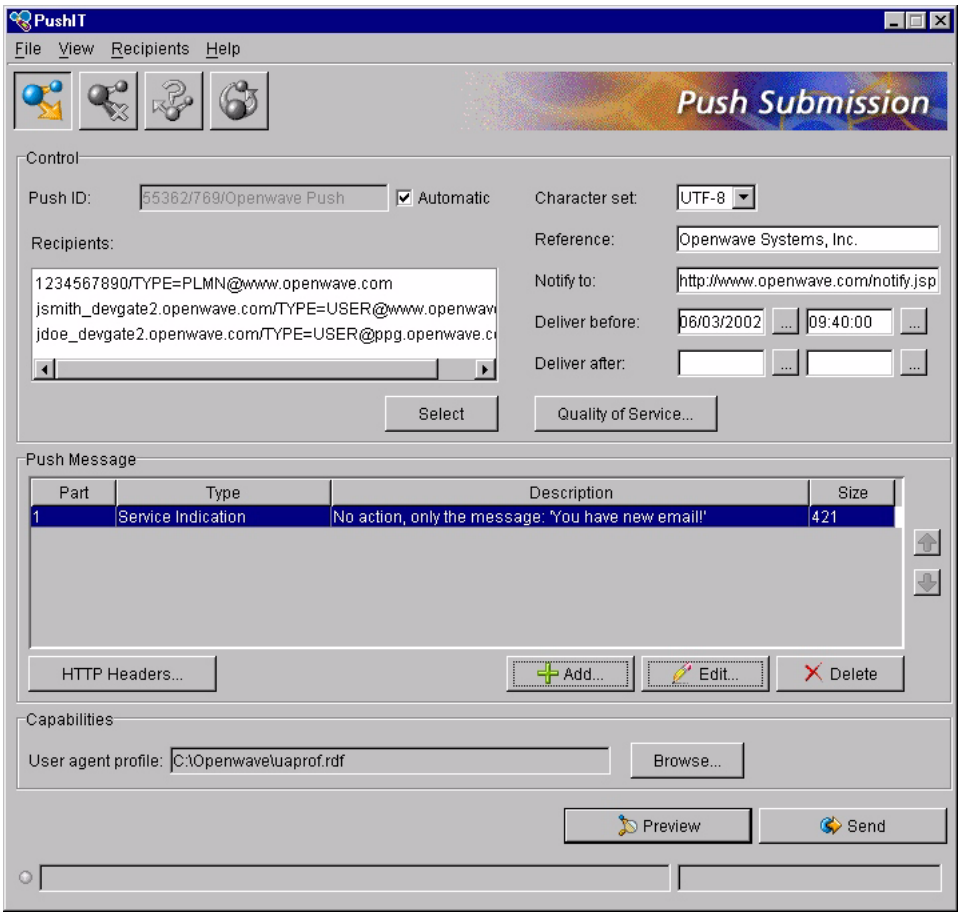
Starting PushIT

To start the Push Initiator Tool, click the Start button on the Windows taskbar, select Programs>Openwave WAP Push Library Java 1.0, and then click Push Initiator Tool.

Push Submission Screen

The Push Submission screen provides everything you need to send a Push Submission with any payload type. Using this screen, you specify the Push ID, character set, reference attribute, result notification URL, recipients, delivery deadlines, Quality of Service, message type and content, HTTP headers, and User Agent Profile.

Figure 8-1. PushIT Push Submission screen

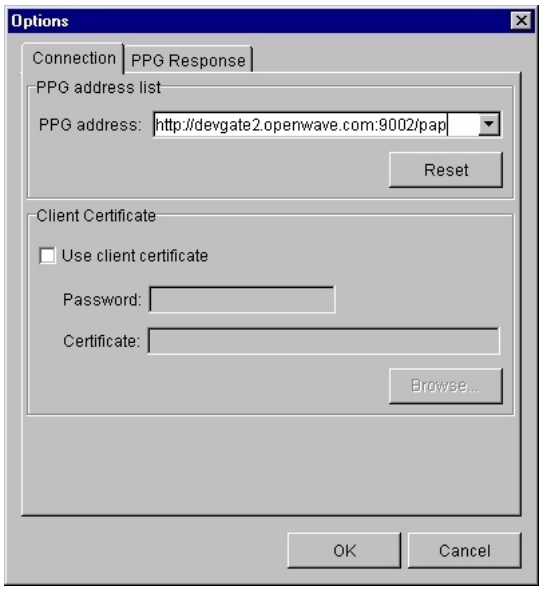


PPG Address

The Openwave Push Proxy Gateway (PPG) server provides push functionality to subscribers and providers. For more information, see “PPG and Client Addresses” on page 23.

To Set the PPG Address

1 Select View>Options to open the Options dialog box.



2 Click the Connection tab and enter the desired PPG address.

You must enter a fully qualified URL. In most cases, the URL contains a port and a subdirectory, depending on how the PPG has been configured. Your provider can give you the exact value.

3 You can also set the client security certificate on this tab, if desired.

4 Click OK to set the PPG address.

Push ID

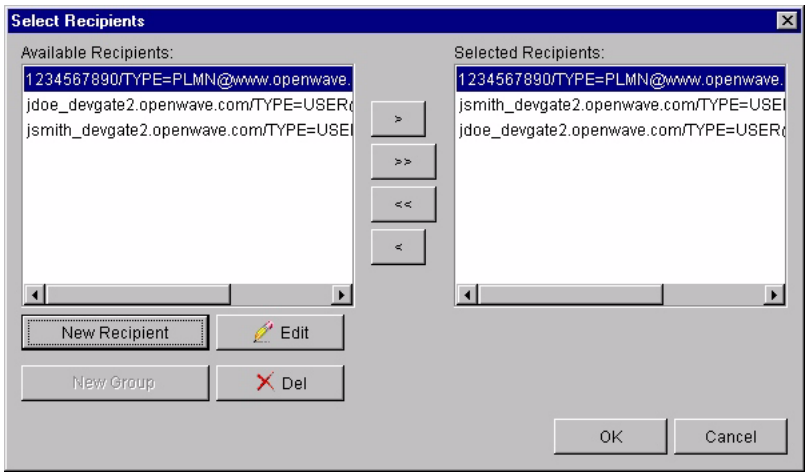
To have PushIT generate a unique push ID for each Push Submission, select the Automatic check box. To generate your own push ID, enter the desired push ID.

Recipients

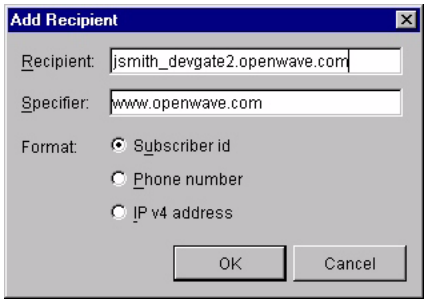
Any Push Submission that you send using PushIT is delivered to all of the recipients shown in the Recipients list. A recipient address is required for all push operations. You can select more than one recipient for each Push Submission. Delivery to more than one recipient is known as *multicasting*.

To Add a New Recipient

1 In the Push Submission screen, click Select to open the Select Recipients dialog box.



2 Click New Recipient to open the Add Recipient dialog box.



3 In the Recipient field, enter the phone number or ID.

Use numbers only to enter the phone number. For example, enter 8005558000 rather than (800) 555-8000.

The subscriber ID is the subscriber's delivery ID as created when the subscriber was provisioned. If you do not know the subscriber ID, contact your provider.

In most cases, the subscriber ID is preferred over a phone number.

4 In the Specifier field, enter the recipient address specifier and then select the radio button corresponding to the address type.

5 Click OK to close the Add Recipient dialog box.

6 Select the new recipient in the Available Recipients list and then click the > button to copy the new recipient to the Selected Recipients list.

7 Click OK.

To Select Recipients

1. In the Push Submission screen, click Select to open the Select Recipients dialog box.
2. Select the desired recipient in the Available Recipients list.
3. Click the > button to copy the highlighted recipient to the Selected Recipients list. You can also click the >> button to copy all entries in the Available Recipients list to the Selected Recipients list.
4. To remove a recipient from the mailing list, select the desired recipient in the Selected Recipients list.
5. Click the < button to remove the highlighted recipient from the Selected Recipients list. You can also click the << button to remove all entries from the Selected Recipients list.
6. Click OK.

To Edit a Recipient

1. Click Select to open the Select Recipients dialog box.
2. Select the desired recipient in the Available Recipients list.
3. Click Edit.
4. Edit the entry.
5. Click OK to submit the changes.

To Delete a Recipient

1. Click Select to open the Select Recipients dialog box.
2. Select the desired recipient in the Available Recipients list.
3. Click Delete.
4. Click OK.

Character Set

PushIT currently supports only UTF-8, an English character set encoding that is ASCII compatible.

Reference

This item sets the value of the optional `source-reference` attribute of the `push-message` element, which specifies the name of the content provider. This information allows the PPG operator to identify the message originator. Enter the desired information.

Notify To

Sets the value of the optional `ppg-notify-requested-to` attribute of the `push-message` element, which specifies the URL that the PPG should use for notification of results of a push message. Enter the desired information.

Deliver Before/After

You can specify that a push be delivered before or after a specified date and time.

- **Deliver before** To have the push delivery made before a specific time and date, use the ... buttons to select the date and time, or enter the date and time directly.
- **Deliver after** To have the push delivery made after a specific time and date, use the ... buttons to select the date and time, or enter the date and time directly.

For more information, see "Specifying Push Submission Delivery Timing" on page 25.

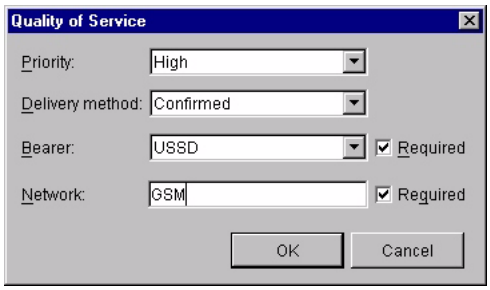
The PPG administrator can change the time and date limitations for delivery timing. For example, the PPG may not be able to deliver before a time specified half an hour from the current time and may report an error. Check with the administrator.

Quality of Service

This item sets the optional delivery qualities required by the push initiator. If the Quality of Service (QOS) requested cannot be honored, the PPG rejects the entire push message.

To Set the Quality of Service

1 Click the Quality of Service button to open the Quality of Service dialog box.



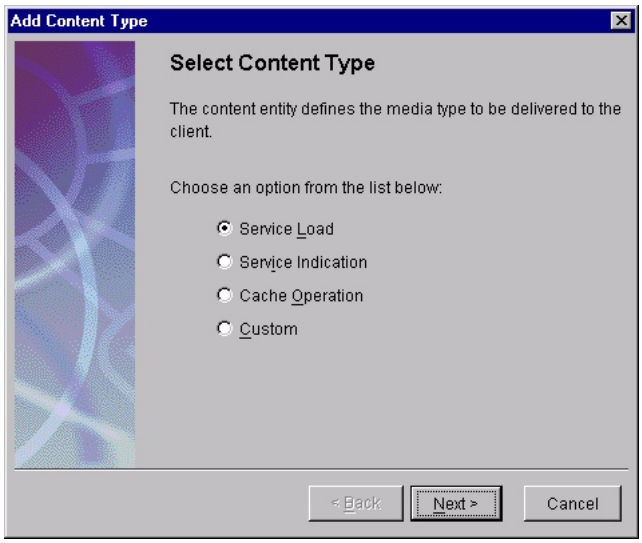
2 Select the desired priority, delivery method, and bearer from the drop-down lists, enter the desired network, then select the Required check boxes as necessary.

3 Click OK to set the QOS for the Push Submission.

Push Message

To set the push message and content, follow these steps.

1 Click the Add button to open the Add Content Type dialog box.



2 Select the radio button for the desired content type.

3 Click Next and follow the wizard steps to build the complete push message.

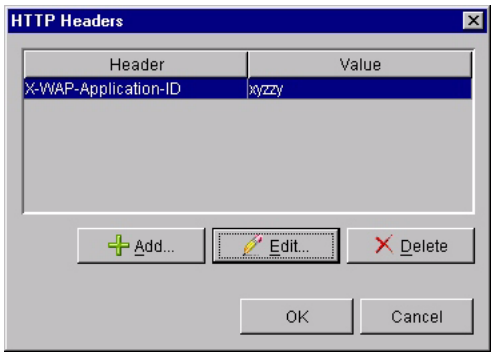
You can build more than one push message to send at a time. All push messages in the Push Message list are sent when you click the Send button.

4 To edit or delete an existing push message, select the desired message in the Push Message list and then click the Edit or Delete button.

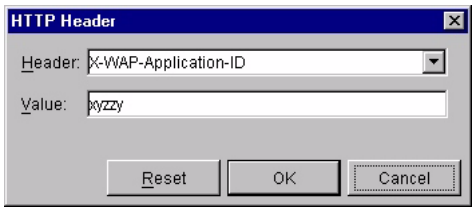
HTTP Headers

To add an additional HTTP header to the Push Submission, follow these steps.

1 Click the HTTP Headers button to open the HTTP Headers dialog box.



2 Click the Add button to open the HTTP Header dialog box.



3 Select the desired header, enter a value, and then click OK.

4 To edit or delete an existing HTTP header, select the desired header from the list in the HTTP Headers dialog box and then click the Edit or Delete button.

User Agent Profile

Click the Browse button and select the `.rdf` file that contains the desired User Agent Profile information.

Preview

Click the Preview button to see the complete XML that contains the Push Submission.

Send

Click the Send button to send the completed Push Submission.

PPG Response Log

To have the PPG response information logged to a file, follow these steps.

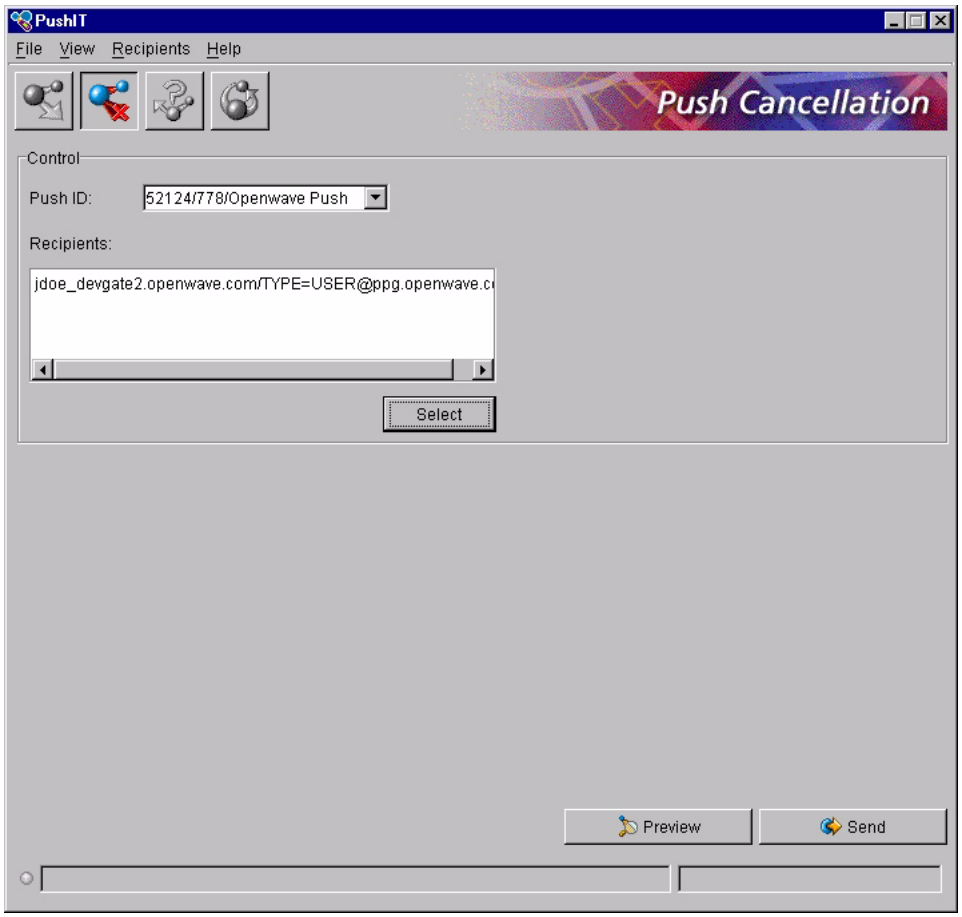
1. Select View>Options to open the Options dialog box.
2. Click the PPG Response tab.
3. Select the Log Responses check box.
4. Click the ... button to locate a directory and create or select a file, or enter the file and path directly.

Push Cancellation Screen

From the Push Submission screen, you can cancel a Push Submission that you previously sent using PushIT. To cancel a Push Submission, select the desired push ID from the drop-down list. If you want to cancel the Push Submission for selected recipients only, click the Select button and select the desired recipients as described in “To Select Recipients” on page 123. If you do not select specific recipients, the Push Submission is canceled for all recipients.

Click the Preview button to see the complete XML containing the Push Cancellation message. Click the Send button to send the Push Cancellation message.

Figure 8-2. PushIT Push Cancellation screen

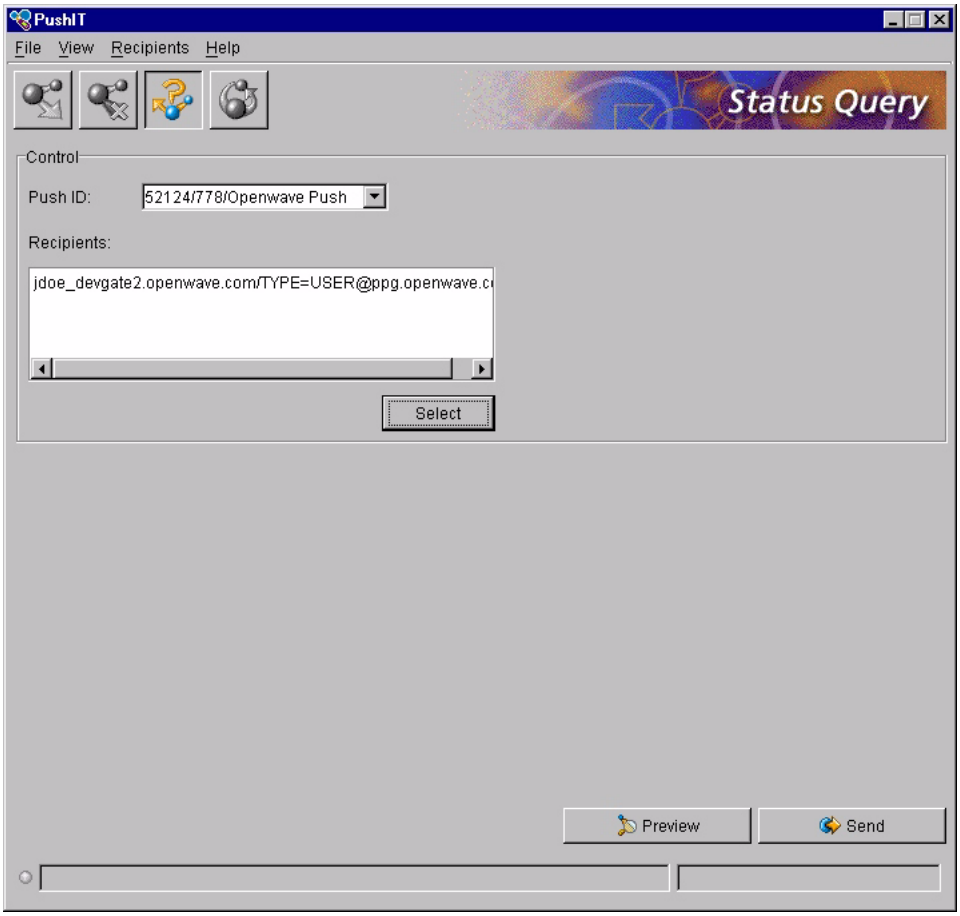


Status Query Screen

From the Status Query screen you can check the status of a Push Submission that you previously sent using PushIT. To send a Status Query message, select the desired push ID from the drop-down list. If you want to retrieve status information for selected recipients only, click the Select button and select the desired recipients as described in “To Select Recipients” on page 123. If you do not select specific recipients, status information is retrieved for all recipients.

Click the Preview button to see the complete XML containing the Status Query message. Click the Send button to send the Status Query message.

Figure 8-3. PushIT Status Query screen



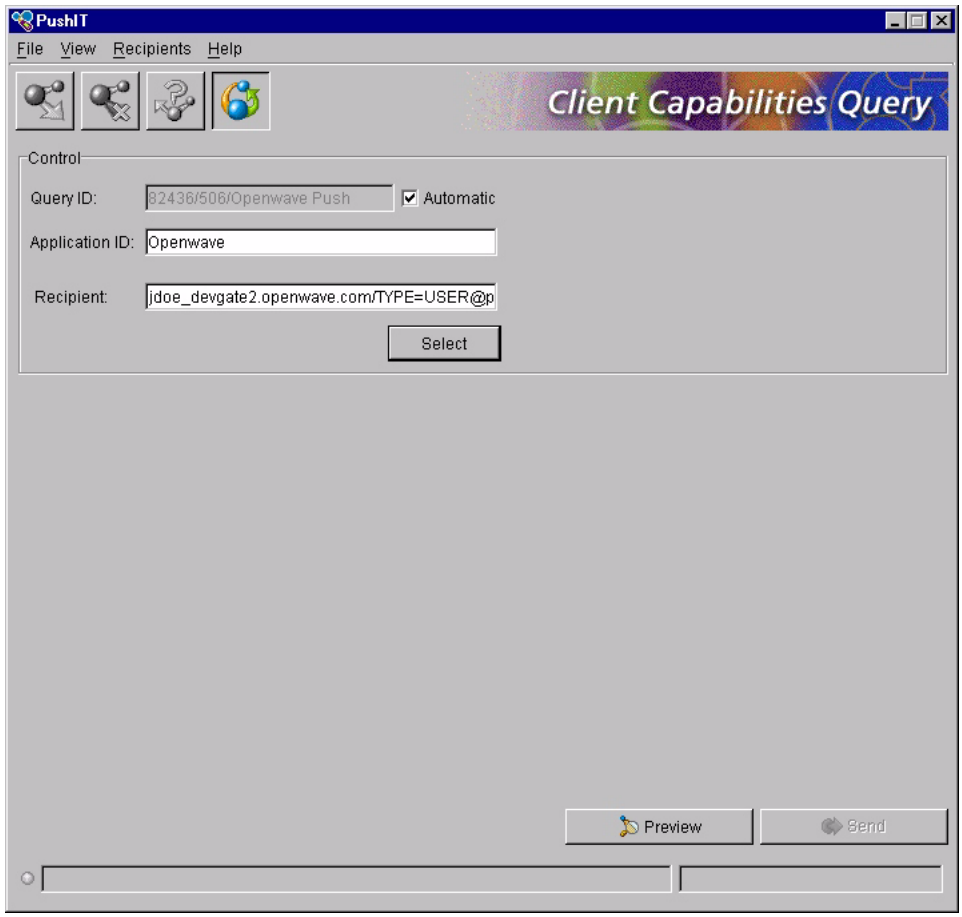
Client Capabilities Query Page

From the Client Capabilities Query (CCQ) screen you can retrieve CCQ information for a specific recipient. A CCQ message is a discrete Push Submission with a unique push ID.

To send a CCQ message, enter the desired push ID or select the Automatic check box to have PushIT generate a unique push ID for this CCQ message. Enter an optional application ID if desired. The application ID sets the value of the optional `app-id` attribute, which uniquely identifies the application that the push initiator targets with a subsequent push message.

To select the recipient for which to retrieve CCQ information, click the Select button and select the desired recipient as described in "To Select Recipients" on page 123. A CCQ message can be sent to one recipient only.

Figure 8-4. PushIT Client Capabilities Query screen



Understanding PushIT

PushIT is implemented as a Java Package and is built on two levels. The graphical user interface level is based on the Java Swing classes. Underneath the GUI is the implementation level, which is based on the WAP Push Library. The different push operations are implemented through classes that use the WAP Push Library, as outlined in the following table.

Table 8-1. PushIT operations and classes

PAP operation	PushIT class
Cancel Message	`MsgCancel`
Client Capabilities Query	`MsgClientCapabilitiesQuery`
Push Submission: Cache Operation	`MsgMultiPart`
Push Submission: Service Indication	`MsgMultiPart`
Push Submission: Service Loading	`MsgMultiPart`
Status Query	`MsgQueryStatus`

The other classes that make up PushIT are devoted to the GUI. These classes gather user input and call the appropriate operational classes to build and send the various push operations. The operational classes implement push operations in a manner very similar to that described in Chapter 6, “WAP Push Library Essentials.” Refer to this chapter and to the PushIT source code for detailed information.

License Agreements A

Xerces

The Apache Software License, Version 1.1 Copyright (c) 1999 The Apache Software Foundation. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The end-user documentation included with the redistribution, if any, must include the following acknowledgment:
- "This product includes software developed by the Apache Software Foundation (`http://www.apache.org`)."
- Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear.
- The names "Xerces" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact `apache@apache.org`.
- Products derived from this software may not be called "Apache", nor may "Apache" appear in their name, without prior written permission of the Apache Software Foundation.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR

SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software consists of voluntary contributions made by many individuals on behalf of the Apache Software Foundation and was originally based on software copyright (c) 1999, International Business Machines, Inc., `http://www.ibm.com`. For more information on the Apache Software Foundation, please see `http://www.apache.org`.

Tomcat

The Apache Software License, Version 1.1 Copyright (c) 1999 The Apache Software Foundation. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The end-user documentation included with the redistribution, if any, must include the following acknowledgement:
- "This product includes software developed by the Apache Software Foundation (`http://www.apache.org`)."
- Alternately, this acknowledgement may appear in the software itself, if and wherever such third-party acknowledgements normally appear.
- The names "The Jakarta Project", "Tomcat", and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact `apache@apache.org`.
- Products derived from this software may not be called "Apache" nor may "Apache" appear in their names without prior written permission of the Apache Group.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software consists of voluntary contributions made by many individuals on behalf of the Apache Software Foundation. For more information on the Apache Software Foundation, please see `http://www.apache.org`.

Glossary

bearer network A network used to carry the messages of a transport-layer protocol between physical devices. Multiple bearer networks may be used over the life of a single push session.

bytecode Content encoding in which the content is typically a set of low-level instructions and operands for a targeted hardware (or virtual) machine.

client A device or application that expects to receive push content from a server.

content Data stored or generated at an origin server. Content is typically displayed or interpreted by a user agent in response to a request from a push initiator.

content encoding When used as a verb, content encoding indicates the act of converting a data object from one format to another. Typically the resulting format requires less physical space than the original, is easier to process or store, or is encrypted. When used as a noun, content encoding specifies a particular format or encoding standard or process.

content format Actual representation of content.

device A network entity that is capable of sending and receiving packets of information and that has a unique device address. A device can act as both a client and a server within a given context or across multiple contexts. For example, a device can service a number of clients (as a server) while being a client to another server.

HTTP Hypertext Transfer Protocol. The standard communication protocol used on the World Wide Web.

HTTPS Secure Hypertext Transfer Protocol. HTTP over a Secure Sockets Layer (SSL) or Transport Layer Security (TLS) connection.

JavaScript A de facto standard language that can be used to add dynamic behavior to HTML documents. JavaScript is one of the originating technologies of ECMAScript.

mobile browser The browser software installed on a mobile device.

mobile device A mobile phone, PDA, or other device. Refers to the device hardware.

origin server The server on which a given resource resides or is to be created. Often referred to as a web server or an HTTP server.

Push Access Protocol (PAP) A protocol used to send client content and push-related control information between a push initiator and a Push Proxy Gateway.

push framework The entire WAP push system. The push framework encompasses the protocols, service interfaces, and software entities that provide the means to push data to user agents in the WAP client.

push initiator The entity that originates push content and submits it to the push framework for delivery to a user agent on a client.

Push OTA Protocol A protocol used to convey content between a Push Proxy Gateway and a specified user agent on a client.

Push Proxy Gateway (PPG) The server that sends content to the target wireless device. The PPG acts as an intermediary, connecting the wired and wireless networks, which would otherwise have no way of communicating with each other because they use different communication protocols.

resource A network data object or service that can be identified by a URL. Resources may be available in multiple representations (for example, multiple languages, data formats, size, resolutions, and so on) or vary in other ways.

server A device (or application) that passively waits for connection requests from one or more clients. A server may accept or reject a connection request from a client.

user A person who interacts with a user agent to view, hear, or otherwise use rendered content.

user agent Any software or device that interprets WML, WMLScript, or resources. User agents include textual browsers, voice browsers, search engines, and so on.

web server A network host that acts as an HTTP server.

WML Wireless Markup Language. A hypertext markup language used to represent information for delivery to a mobile device.

WMLScript A scripting language used to program a mobile device. WMLScript is an extended subset of the JavaScript scripting language.

Index