

Managing Personal Information - Using the PIM API for Java ME

By [C. Enrique Ortiz](#), April 2007

Mobile handsets such as cellphones typically have an *address book* to keep track of people we like to stay in touch with, a *calendar* to keep track of important events, and a *to-do list* to keep track of items we don't want to forget. This type of *personal information* is one of the most important functions found in a handset, just second to voice. You can use the PIM API for Java ME to enable your mobile Java applications to read and write to/from the locally stored personal information databases. You can even write synchronizers to keep your handset PIM data in-sync with remote PIM data stores. This article, part 5 of this series on using the PIM API for Java ME, provides a comprehensive introduction with code examples on how to use the PIM API.

Contents

- [Using the PIM API](#)
- [The Sun Java Wireless Toolkit](#)
- [Resources](#)
- [Acknowledgements](#)
- [About the Author](#)

Using the PIM API

Let's now look at the PIM API in more detail and how to use it. We'll commence by reviewing the typical actions or usage, and considerations when using the API:

- [Portability considerations](#)
- [Security considerations](#)
- Get the PIM instance
- Get names of available PIM databases
- Open the desired PIM lists
- Get or search for PIM items
- Add, edit, delete PIM contact, events, and to-do items
- Commit changes
- Import and export to/from [vCard](#) and [vCalendar](#) personal data interchange formats
- Close the PIM lists

With the PIM API for Java ME, managing contacts, events, and to-do items is accomplished using similar concepts across the different types of PIM data, where `PIMList`, `PIMItem`, and fields allow for a generic approach to the handling of PIM data, and where the specifics are encapsulated by the sub-interfaces for `PIMList` and `PIMItem` as previously discussed. In many respects, you can think of the PIM API as a tuple-based API, where a specific PIM data object is described as `(list, item, fieldID)`.

Since the API is generic across the different types of PIM data, for the sake of keeping the article short enough, the following sections will use calendar `Events` for all code examples. It is left up to the reader to adapt the event examples and extrapolate those for contact and to-do items.

Retrieving the PIM instance

The first step when using the PIM API is to retrieve an *instance of the PIM class*, by calling the static method `PIM.getInstance()`, as shown in the following code snippet:

```
import javax.microedition.pim.*;
:

private PIM pim;
:

pim = PIM.getInstance(); // retrieve instance of PIM
```

The PIM instance gives us access to PIM lists or databases, then to PIM items and fields.

Getting the Names of PIM Databases

To retrieve a list of names for all the available PIM databases of a given type, use the method `PIM.listPIMLists(PIMListType)` passing as argument the PIM list type of interest. The following code snippet shows how to get the names of all calendar event PIM lists:

```
import javax.microedition.pim.*;
:

private PIM pim;
:

pim = PIM.getInstance(); // retrieve instance of PIM
:

// Get names of PIM list by PIM LIST type:
// PIM.CONTACT_LIST, PIM.EVENT_LIST & PIM.TODO_LIST
String[] eventLists = pim.listPIMLists(PIM.EVENT_LIST);

// If the size of the returned String array is greater than
// zero means there are PIM databases of the specified
// type.
if (eventLists.length > 0) {
    // Open the event PIM list in read/write mode
    :
}
```

Supported PIM list types are: `PIM.CONTACT_LIST`, `PIM.EVENT_LIST`, or `PIM.TODO_LIST`. The `listPIMLists()` method returns a `String` array containing the *unique names* for each list that currently exist, with the first entry in the array being the *default list* for the specified PIM list type. A zero-length `String` array is returned if no lists of the specified type exist.

As previously covered, recall that all PIM databases might not be supported on a given PIM implementation.

Opening a PIM List Database

When opening a PIM list database, you can open the default database of a given type, or open a specific database by name. Note the PIM API

supports more than one database of the same type.

On its simplest form, you can **open the default database** by calling `openPIMList`, passing as arguments the PIM list type, and mode:

```
PIMList openPIMList(PIMListType, mode);
```

...where `PIMListType` is the type of database to open: for Contacts use `PIM.CONTACT_LIST`, for calendar Events use `PIM.EVENT_LIST`, and for To-Do database use `PIM.TODO_LIST`. The valid modes are `PIM.READ_ONLY`, `PIM.WRITE_ONLY`, or `PIM.READ_WRITE`. The following code snippet shows how to open the default event PIM list in the specified mode:

```
import javax.microedition.pim.*;
:

private PIM pim;
:

pim = PIM.getInstance(); // retrieve instance of PIM
:

/**
 * Opens the default event PIM list
 * @param mode is PIM.READ_WRITE, PIM.READ_ONLY, PIM.WRITE_ONLY
 * @return the EventList instance if successful, null otherwise
 */
static public EventList openDefaultEventList(int mode) {
    EventList eventList = null;
    if (mode != PIM.READ_ONLY &&
        mode != PIM.WRITE_ONLY &&
        mode != PIM.READ_WRITE) {
        return null;
    }
    try {
        // Open the default event PIM list in the specified mode
        eventList = (EventList) pim.openPIMList(PIM.EVENT_LIST, mode);
    } catch (PIMException pe) {
        // process PIM exception
        eventList = null;
    }
    return eventList;
}
```

To **open a specific database by name** use the method `openPIMList(PIMListType, mode, listName)` passing as arguments the PIM list type and mode, similarly to how it was shown above, with the addition of a third argument for the list name. The following code snippet shows how to open a PIM list by name:

```
import javax.microedition.pim.*;
:

private PIM pim;
:

pim = PIM.getInstance(); // retrieve instance of PIM
:

/**
 * Opens the event list by name
 * @param name the name of the event list to open
 * @param mode is PIM.READ_WRITE, PIM.READ_ONLY, PIM.WRITE_ONLY
 * @return the EventList instance if named list is found, null otherwise
 */
static public EventList openDefaultEventListByName(String name, int mode) {
    EventList eventList = null;
    try {
        eventList = (EventList) pim.openPIMList(PIM.EVENT_LIST, mode, name);
    } catch (PIMException pe) {
        // process PIM exception
        eventList = null;
    }
    return eventList;
}
```

Alternatively, a different way to open the *default* database for a give type is to open it by name, by first calling the method `listPIMLists()` to retrieve the list of databases, then calling the method `openPIMList(PIMListType, mode, listName)` passing as argument the *first* element on the array that was returned by `listPIMLists()` - if the length of the array is greater than zero, this is would be the element at offset 0, which as previously explained, is always the default list:

```
/**
 * Opens the default event list by name
 * @param name the name of the event list to open
 * @return the EventList instance if successful, null otherwise
 */
static public EventList openDefaultEventList2() {
    EventList eventList = null;
    try {
        // Get names of PIM list by PIM LIST type:
        // PIM.CONTACT_LIST, PIM.EVENT_LIST & PIM.TODO_LIST
        String[] eventLists = pim.listPIMLists(PIM.EVENT_LIST);

        // If the size of the returned String array is greater than
        // zero means there are PIM databases of the specified
        // type. For illustration purposes, open the PIM list with
        // index of zero (which is the default list)
        if (eventLists.length > 0) {
            // Open the event PIM list in read/write mode
            eventList = (EventList) pim.openPIMList(PIM.EVENT_LIST, PIM.READ_WRITE, eventLists[0]);
        }
    } catch (PIMException pe) {
        // process PIM exception
        eventList = null;
    }
}
```

```

        return eventList;
    }

```

Data Exchange Formats

The PIM API provides a number of methods to import and export PIM data using standard data exchange formats. To discover the supported data exchange formats, call the method `supportedSerialFormats(PIMListType)`, passing as argument the type of PIM list to import or export. The following code snippet shows how to discover the data exchange formats for PIM events lists:

```

import javax.microedition.pim.*;
:

private PIM pim;
:

pim = PIM.getInstance(); // retrieve instance of PIM
:

/**
 * Discover the supported data exchange serial formats
 * for PIM List items
 */
static public String[] discoverEventListSupportedDataExchangeFormats() {
    String[] serialFormats = null;
    serialFormats = pim.supportedSerialFormats(PIM.EVENT_LIST);
    return serialFormats;
}

```

To discover the other types of PIM list data exchange formats, just replace the PIM list type `PIM.EVENT_LIST` above, for `PIM.CONTACT_LIST` or `PIM.TODO_LIST` as appropriate.

The returned serial format names follow the proper common naming convention that is suitable for input to the `toSerialFormat` and `fromSerialFormat` import/export methods. Supported data exchange formats include the `vCard` and `vCalendar` formats-for more information on `vCard` and `vCalendar`, see the [Personal Data Interchange web site](#).

Exporting a PIM Item

To export PIM data use the method `PIM.toSerialFormat(PIMItem, OutputStream, encoding, dataFormat)`, passing as arguments an `OutputStream` to serialize, the character encoding to use, and the supported (previously discovered) data exchange format to convert to. As previously covered, to discover the supported serial data formats for a given type of PIM data, as such as `vCard` or `vCalendar`, use the method `PIM.supportedSerialFormats(PIMListType)`. The following code snippet shows how to export an Event item:

```

import javax.microedition.pim.*;
import java.io.OutputStream;
:

private static final String exportEncoding = "UTF-8";
private PIM pim;
:

pim = PIM.getInstance(); // retrieve instance of PIM
:

/**
 * Export the specified event onto the speficied OutputStream
 * using the specified format
 * @param event the event PIM item to export
 * @param os the OutputStream to export to
 * @return true if successful, false otherwise
 */
private static final String exportEncoding = "UTF-8";
static public boolean exportEvent(Event event, OutputStream os) {
    boolean status = false;
    String[] supportedFormats = discoverEventListSupportedDataExchangeFormats();
    if (supportedFormats.length > 0) {
        try {
            pim.toSerialFormat(
                event,
                os,
                exportEncoding,
                supportedFormats[0]);
            status = true;
        } catch (PIMException pe){
            // handle PIM exception
            status = false;
        } catch (UnsupportedEncodingException ue){
            // handle unsupported exception
            status = false;
        } catch (Exception e){
            // handle other exception
            status = false;
        }
    }
    return status;
}

```

Exporting the other types of PIM data, such as calendar and to-do PIM items, is done exactly the same way as shown above, except you would use the PIM list type of `PIM.CONTACT_LIST` and `PIM.TODO_LIST` as appropriate.

Note that PIM data fields that are not supported by the specified export data exchange format are written as extended fields.

Importing a PIM Item

To import PIM data, use the method `fromSerialFormat(InputStream, encoding)`, passing as arguments the `InputStream` to serialize from, and the character encoding to use. Note that the PIM API doesn't provide helper methods for *transfer character encoding* and implementing such is the responsibility of the developer. The transfer character encoding is specified via the headers `Content-Transfer-Encoding` and `Transfer-Encoding` for MIME and HTTP respectively. A common transfer encoding scheme is "quoted-printable", on which characters are represented in hexadecimal and where line lengths can't exceed 76 characters. Another popular transfer encoding type is `base64`. The following code snippet shows how to import event items from an input stream:

```

import javax.microedition.pim.*;
import java.io.InputStream;

```

```

:

private static final String importEncoding = "UTF-8";
private static final String qp = "quoted-printable";
private static final String b64 = "base64";
private PIM pim;
:

pim = PIM.getInstance(); // retrieve instance of PIM
:

/**
 * Imports an event. This method assumes the list is properly opened
 * and in the proper write mode.
 * @param transferEncoding the transfer encoding to use
 * @param is the InputStream to import from
 * @param el the EventList to import to
 * @return true if import was successful, false otherwise
 */
private static final String importEncoding = "UTF-8";
private static final String qp = "quoted-printable";
private static final String b64 = "base64";
static public boolean importEvent(
    String transferEncoding,
    InputStream is,
    EventList el) {
    PIMItem[] items = null;
    boolean status = true;
    try {
        if (transferEncoding.trim().toLowerCase().equals(b64)) {
            // Decode using application defined Base64 input stream decoder
            Base64InputStream b64is = new Base64InputStream(is);
            items = pim.fromSerialFormat(b64is, importEncoding);
        } else if (transferEncoding.trim().toLowerCase().equals(qp)) {
            // Decode using application defined Quoted-Printable input stream decoder
            QuotedPrintableInputStream qpis = new QuotedPrintableInputStream(is);
            items = pim.fromSerialFormat(qpis, importEncoding);
        } else {
            // items is null
            status = false;
        }
        if (items != null) {
            // For each imported PIM event item, import it into the Event list.
            for (int i = 0; i < items.length; i++) {
                Event event = (Event) (items[i]);
                el.importEvent(event);
                event.commit();
            }
        }
    } catch (PIMException pe) {
        // handle PIM exception
        status = false;
    } catch (UnsupportedEncodingException ue) {
        // handle unsupported exception
        status = false;
    } catch (Exception e) {
        // handle other exception
        status = false;
    }
    return status;
}
}

```

The above code snippets use two application-defined classes `QuotedPrintableInputStream` and `Base64InputStream` to properly decode the input stream prior to import; these are not included in this article.

The method `fromSerialFormat` creates an array of PIM items from the `InputStream`, but these items are *not* imported into the corresponding PIM list. To import into the corresponding PIM list you must call the appropriate PIM item import method: `ContactList.importContact` to import Contacts, `EventList.importEvent` to import Events, and `ToDoList.importToDo` to import To-Do items. Note that imported events are not automatically added to the PIM list, and that it's necessary to call the item's `commit()` method to ensure the item is saved into the list.

Importing calendar and to-do PIM items is done similarly as shown above, except you would handle instead a `ContactList` or `ToDoList` as appropriate.

Note that PIM data fields that are not supported by the import data exchange format are either imported as extended fields, or are silently discarded.

Adding a PIM Item

Adding a PIM item, regardless if it is an event, contact or to-do item, is done using the same approach. To add a PIM item you must know the item's field names, data-types, and values to set. The following code snippet shows how to add fields to a PIM `Event` item; in this example, instead of using the introspection design approach (see [Design Considerations](#)), the helper method explicitly sets the standard event fields:

```

import javax.microedition.pim.*;
:

pim = PIM.getInstance(); // retrieve instance of PIM
:

/**
 * Add fields to the event.
 * @param event the event's to add
 * @param startDate the event's start date/time
 * @param endDate the event's end date/time
 * @param summary the event's summary description
 * @param location the event's location
 * @param note the event's note for further description
 * @param privacyClass the event's privacy class: private, public
 * @param alarm the event's delta time for the event's alarm (i.e. 5 minutes prior)
 * @param uid the event's unique ID
 * @return true if info successfully added, false otherwise
 */

```

```

static public boolean addEventInfo(
    Event event,          // The event
    long startDate,       // START
    long endDate,         // END
    String summary,       // SUMMARY
    String location,       // LOCATION
    String note,          // NOTE
    int privacyClass,     // CLASS
    int alarm) {          // ALARM
    boolean status = false;
    if (event == null) {
        return false;
    }

    // Get list for the event
    PIMList eventList = event.getPIMList();
    if(eventList == null) {
        return false; // event doesn't belong to a list
    }

    // Add each field. For portability, before using a field, test if it is supported.
    if (eventList.isSupportedField(Event.START) == true) {
        event.addDate(Event.START, PIMItem.ATTR_NONE, startDate);
    }
    if (eventList.isSupportedField(Event.END) == true) {
        event.addDate(Event.END, PIMItem.ATTR_NONE, endDate);
    }
    if (eventList.isSupportedField(Event.SUMMARY) == true) {
        event.addString(Event.SUMMARY, PIMItem.ATTR_NONE, summary);
    }
    if (eventList.isSupportedField(Event.LOCATION) == true) {
        event.addString(Event.LOCATION, PIMItem.ATTR_NONE, location);
    }
    if (eventList.isSupportedField(Event.NOTE) == true) {
        event.addString(Event.NOTE, PIMItem.ATTR_NONE, note);
    }
    if (eventList.isSupportedField(Event.CLASS) == true) {
        event.addInt(Event.CLASS, PIMItem.ATTR_NONE, privacyClass);
    }
    if (eventList.isSupportedField(Event.ALARM) == true) {
        event.addInt(Event.ALARM, PIMItem.ATTR_NONE, alarm);
    }
    return true;
}

```

Note how prior to using a field, each individual field is tested by calling the method `isSupportedField()` to see if it is supported on the running platform. If an unsupported field is used, an `UnsupportedFieldException` is thrown. Note that the UID field is not added, since it is automatically assigned by the PIM implementation at the time the item is committed.

Adding PIM items of type `Contact` and `ToDo` is done as shown above, but instead, use the appropriate fields for the given PIM item.

The above code example assumes that the caller of method `addEventInfo()` will `commit` the PIM item so that changes are saved. Once a PIM list is no longer needed, it should be `closed` to release resources. The following code snippet illustrates this:

```

:

EventList el = PimEventUtils.openDefaultEventList(PIM.READ_WRITE);
Event event = el.createEvent();
PimEventUtils.addEventInfo(
    event,
    System.currentTimeMillis(), // START
    System.currentTimeMillis(), // END
    "This is the summary",       // SUMMARY
    "This is the location",      // LOCATION
    "This is the note",          // NOTE
    Event.CLASS_PUBLIC,          // CLASS
    0); // ALARM

try {
    event.commit();
} catch (PIMException pe) {
    // handle PIM exception
}
try {
    el.close();
} catch (PIMException pe) {
    // handle PIM exception
}

PIMItem.EXTENDED_FIELD_MIN_VALUEgetFieldLabel()getFieldDataType()getAttributes()

```

Setting (Changing) a PIM Item

Setting (updating) a PIM item is very similar to adding a PIM item, except that instead of calling the `PIMItem.addXXX()` methods, you call the corresponding `setXXX()` methods based on the field's data-type; see Table 1 below for more information on `add`, `get` and `set` methods.

Table 1: PIMItem Field Add, Set, and Get Methods

| Data Type | Add Method | Get Method | Set Method |
|--------------|----------------------------------|----------------------------------|----------------------------------|
| BINARY | <code>addBinary(...)</code> | <code>getBinary(...)</code> | <code>setBinary(...)</code> |
| BOOLEAN | <code>addBoolean(...)</code> | <code>getBoolean(...)</code> | <code>setBoolean(...)</code> |
| DATE | <code>addDate(...)</code> | <code>getDate(...)</code> | <code>setDate(...)</code> |
| INT | <code>addInt(...)</code> | <code>getInt(...)</code> | <code>setInt(...)</code> |
| STRING | <code>addString(...)</code> | <code>getString(...)</code> | <code>setString(...)</code> |
| STRING_ARRAY | <code>addStringArray(...)</code> | <code>getStringArray(...)</code> | <code>setStringArray(...)</code> |

...where arguments are `field ID`, `value-index` (0 for single-value fields), `attributes`, and `data` value as appropriate; note that the signature of `PIMItem.addXXX` methods is similar to the signature of the `PIMItem.setXXX` methods except the latter defines an index argument for multi-value fields. Please refer to the Javadoc for more information.

The following code snippet shows how to `set/update` an `Event` PIM item; in this example instead of using the introspection design approach (see [Design Considerations](#)), the helper method explicitly sets the standard event fields:

```

import javax.microedition.pim.*;
:

pim = PIM.getInstance(); // retrieve instance of PIM
:

/**
 * Set (updates) the event fields
 * @param event the event to set
 * @param startDate the event's start date/time
 * @param endDate the event's end date/time
 * @param summary the event's summary description
 * @param location the event's location
 * @param note the event's note for further description
 * @param privacyClass the event's privacy class: private, public
 * @param alarm the event's delta time for the event's alarm (i.e. 5 minutes prior)
 * @param uid the event's unique ID
 * @return true if info successfully added, false otherwise
 */
static public boolean setEventInfo(
    Event event,
    long startDate, // START
    long endDate, // END
    String summary, // SUMMARY
    String location, // LOCATION
    String note, // NOTE
    int privacyClass, // CLASS
    int alarm) { // ALARM
    boolean status = false;
    if (event == null) {
        return false;
    }
    // Get the list for the event
    PIMList eventList = event.getPIMList();
    if(eventList == null) {
        return false; // event doesn't belong to a list
    }
    if ((eventList.isSupportedField(Event.START) == true) &&
        (startDate != -1)) {
        event.setDate(Event.START, 0, PIMItem.ATTR_NONE, startDate);
    }
    if ((eventList.isSupportedField(Event.END) == true) &&
        (endDate != -1)) {
        event.setDate(Event.END, 0, PIMItem.ATTR_NONE, endDate);
    }
    if ((eventList.isSupportedField(Event.SUMMARY) == true) &&
        (summary != null)) {
        event.setString(Event.SUMMARY, 0, PIMItem.ATTR_NONE, summary);
    }
    if ((eventList.isSupportedField(Event.LOCATION) == true) &&
        (location != null)) {
        event.setString(Event.LOCATION, 0, PIMItem.ATTR_NONE, location);
    }
    if ((eventList.isSupportedField(Event.NOTE) == true) &&
        (note != null)) {
        event.setString(Event.NOTE, 0, PIMItem.ATTR_NONE, note);
    }
    if ((eventList.isSupportedField(Event.CLASS) == true) &&
        (privacyClass != -1)) {
        event.setInt(Event.CLASS, 0, PIMItem.ATTR_NONE, privacyClass);
    }
    if ((eventList.isSupportedField(Event.ALARM) == true) &&
        (alarm != -1)) {
        event.setInt(Event.ALARM, 0, PIMItem.ATTR_NONE, alarm);
    }
    return true;
}
}

```

Similar to the `addEventInfo` method covered above, prior to using a field, each individual field is tested by calling the method `isSupportedField()` to ensure the field is supported. If an unsupported field is used, an `UnsupportedFieldException` is thrown. Also note the UID value is not set (changed), as its value remains for the life of the PIM item.

The caller must commit the changes, and close the list when it is no longer needed to release resources, as previously covered.

Changing PIM items of type `Contact` and `ToDo` is done as shown above, but instead, use the appropriate fields for the given PIM item.

Getting a PIM Item's Fields and Values

To retrieve a field and its value, you must first know the PIM item to which the field belongs, the ID of the field of interest, and its data-types. Some fields have a single value while others are multi-value. In addition, it is a good practice to always test if a particular field is supported before using it. Let's define a set of generic `getXXXField()` methods to simplify field value retrieval based on data-type:

```

/**
 * Generic PIMItem get string field accessor method.
 * @pimItem the PIM item to reference
 * @param id the field id to retrieve
 * @param index the value id to retrieve
 */
static public String getSimpleStringField(
    PIMItem pimItem,
    int id,
    int index) {
    if (pimItem == null) {
        return null;
    }
    // Get the list for the item
    PIMList pimList = pimItem.getPIMList();
    if(pimList == null) {
        return null; // PIM item doesn't belong to a list
    }
    if (pimList.isSupportedField(id) == false) {
        return null;
    }
}

```

```

    }
    return pimItem.getString(id, index);
}

/**
 * Generic PIMItem get compound string field accessor method.
 * @pimItem the PIM item to reference
 * @param id the field id to retrieve
 * @param index the value id to retrieve
 */
static public String getCompoundStringField(
    PIMItem pimItem,
    int id,
    int index) {
    if (pimItem == null) {
        return null;
    }
    // Get the list for the PIM item
    PIMList pimList = pimItem.getPIMList();
    if (pimList == null) {
        return null; // PIM item doesn't belong to a list
    }
    if (pimList.isSupportedField(id) == false) {
        return null; // field not supported
    }
    String[] values = pimItem.getStringArray(id, index);
    return joinStringArray(values);
}

/**
 * Joins a StringArray returning a single String
 * (from the Sun Java Wireless Toolkit PIMDemo example)
 *
 * @param stringArray the String array to join
 * @return joined String array as a single String
 */
static private String joinStringArray(String[] stringArray) {
    if (stringArray == null) {
        return null;
    }
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < stringArray.length; i++) {
        if (stringArray[i] != null && stringArray[i].length() > 0) {
            if (sb.length() > 0) {
                sb.append(", ");
            }
            sb.append(stringArray[i]);
        }
    }
    return sb.toString();
}

/**
 * Generic PIMItem get date (long) field accessor method
 * @event the PIM item to reference
 * @param id the field id to retrieve
 * @param index the value id to retrieve
 */
static public long getDateField(
    PIMItem pimItem,
    int id,
    int index) {
    if (pimItem == null) {
        return -1;
    }
    // Get the list for the PIM item
    PIMList pimList = pimItem.getPIMList();
    if (pimList == null) {
        return -1; // PIM item doesn't belong to a list
    }
    if (pimList.isSupportedField(id) == false) {
        return -1; // field is not supported
    }
    return pimItem.getDate(id, index);
}

/**
 * Generic PIMItem get int field accessor method
 * @event the PIM event item to reference
 * @param id the field id to retrieve
 * @param index the value id to retrieve
 */
static public int getIntField(
    PIMItem pimItem,
    int id,
    int index) {
    if (pimItem == null) {
        return -1;
    }
    // Get the list for the PIM item
    PIMList pimList = pimItem.getPIMList();
    if (pimList == null) {
        return -1; // PIM item doesn't belong to a list
    }
    if (pimList.isSupportedField(id) == false) {
        return -1; // field not supported
    }
    int value = pimItem.getInt(id, index);
    return value;
}

```

The following code snippet shows how to retrieve different field values for an `Event` PIM item, using the above generic get methods:

```
Event event = ...;
:

// Get the event's UID
String eventUid = getSimpleStringField(event, Event.UID, 0);
if (eventUid == null) {
    // invalid UID...
}

// Get the event's start date/time
long eventStartDate = getDateField(event, Event.START, 0);
if (eventStartDate == -1) {
    // invalid Start Date...
}

// Get the event's summary
String eventSumm = getSimpleStringField(event, Event.SUMMARY, 0);
if (eventSumm == null) {
    // invalid Summary...
}

// Get the event's location
String eventLoc = getSimpleStringField(event, Event.LOCATION, 0);
if (eventLoc == null) {
    // invalid Location...
}

// Get the event's note (description)
String eventNote = getSimpleStringField(event, Event.NOTE, 0);
if (eventNote == null) {
    // invalid Note...
}

// Get the event's privacy class: public vs. private
int eventClass = getIntField(event, Event.CLASS, 0);
if (eventClass == -1) {
    // invalid Class...
}

// Get the event's alarm reminder delta time
int eventAlarm = getIntField(event, Event.ALARM, 0);
if (eventAlarm == -1) {
    // invalid Alarm...
}
}
```

The generic get by data-type helper methods hide repetitive details such as retrieving the list associated with a particular item, and testing if the field is supported.

Removing a PIM Item

To remove a particular PIM item, call the remove item method on the appropriate PIM list. The following code snippet shows how to remove an event item:

```
import javax.microedition.pim.*;
:

pim = PIM.getInstance(); // retrieve instance of PIM
:

/**
 * Remove the specified Event
 * @param event the event to remove
 */
static public void removeEvent(Event event) {
    if (event != null) {
        EventList el = (EventList) event.getPIMList();
        try {
            el.removeEvent(event);
        } catch (PIMException pe) {
            // handle PIM exception
        }
    }
}
```

Removing a `Contact` or `ToDo` item is done as shown above, but instead, use the appropriate PIM item and list; to remove a contact, use `Contact`, `ContactList`, and `removeContact`, and for to-do, use `ToDo`, `ToDoList`, and `removeToDo`.

Commit the Item and Close the List

Changes to a PIM item are not automatically saved into the associated PIM list. You must always commit your changes by calling the method `PIMItem.commit()`. Also PIM lists should be closed when not longer needed to help preserve local resources.

Using Categories

Most PIM implementations support the concept of categories. Categories are `String` tags that you can associate with PIM lists and PIM items to help organize and retrieve PIM data. The PIM API for Java ME supports a complete set of methods to help manage categories:

Methods to manage categories on PIM lists:

- `addCategory` - adds the specified category to the PIM list
- `getCategories` - gets all the categories `String[]` associated with the PIM list
- `deleteCategory` - removes the specified category from the PIM list
- `isCategory` - returns true if the specified category is associated to the PIM list, and false otherwise
- `itemsByCategory` - returns an `Enumeration` of all the items in the PIM list that are part of the specified category
- `maxCategories` - returns the maximum number of categories that can be added to the PIM list
- `renameCategory` - renames the specified PIM list category. References to the old category name are fixed to use the new name

Methods to manage categories on PIM items:

- `addToCategory` - adds the specified category to the PIM item
- `getCategories` - gets all the categories `String[]` associated with the PIM item
- `maxCategories` - returns the maximum number of categories that can be added to the PIM item

`removeFromCategory` - removes the specified category from the PIM item

The following code snippets show how to use the different category methods; for completion we'll start by declaring some categories, and opening the default event list to use:

```
import javax.microedition.pim.*;
:

pim = PIM.getInstance(); // retrieve instance of PIM
:

// Define the categories
private static final String MOMO_CATEGORY = "MobileMonday";
private static final String MOMO_AUSTIN_CATEGORY = "MobileMonday Austin";
:

// Open the default event list
EventList el = openDefaultEventList(PIM.READ_WRITE);
:
```

Add a category to the PIM list:

```
// Add the specified category to the PIM list
try {
    el.addCategory(MOMO_AUSTIN_CATEGORY);
} catch (PIMException pe) {
    // Handle PIM exception
}
```

Get all the categories associated with the PIM list:

```
// Get all the categories associated with the PIM list
String[] categories = null;
try {
    categories = el.getCategories();
} catch (PIMException pe) {
    // Handle PIM exception
}
```

Remove a category from the PIM list:

```
// Remove the specified category from the PIM list
try {
    // Specify false for the argument deleteUnassignedItems,
    // as we don't want to delete items that no longer have any
    // categories assigned to them as a result of this method
    el.deleteCategory(MOMO_AUSTIN_CATEGORY, false);
} catch (PIMException pe) {
    // Handle PIM exception
}
```

Test if the specified category is associated with the PIM list:

```
// Test if the specified category is associated to the PIM list
try {
    if (el.isCategory(MOMO_AUSTIN_CATEGORY)) {
        // The list has MOMO_AUSTIN_CATEGORY category already defined
    }
} catch (PIMException pe) {
    // Handle PIM exception
}
```

Get all the items in the PIM list with the specified category:

```
// Get all the items in the PIM list that are part of the
// specified category
Enumeration items;
try {
    items = el.itemsByCategory(MOMO_AUSTIN_CATEGORY);
} catch (PIMException pe) {
    // Handle PIM exception
}
```

Get the maximum number of categories that can be added to the PIM list:

```
// Get the maximum number of categories that can
// be added to the PIM list
int maxCategories = el.maxCategories();
```

Rename the specified PIM list category:

```
// Rename the specified PIM list category
try {
    el.renameCategory(
        MOMO_AUSTIN_CATEGORY, // current value
        MOMO_CATEGORY);      // new value
} catch (PIMException pe) {
    // Handle PIM exception
}
```

Some tips to be aware of when working with categories:

Catch the `PIMException` that is thrown if categories are not supported by the list. The same exception is thrown if the PIM list is no longer available (perhaps because it has been closed), the maximum number of categories for this list have been reached, or other kind of error has occurred.

While the PIM API treats category names as case-sensitive, the underlying PIM implementation may or may not be case-sensitive.

Strings of zero length (`" "`) may or not be valid. An invalid category results in a `PIMException` when attempting to add it.

Some PIM implementations may only allow categories that already are defined at the PIM list level to be added to PIM items. For example, adding to a PIM item a category not defined in the PIM list may result in the category assignment to be silently ignored or a `PIMException` to be thrown.

Typically you would first discover the existing categories in the PIM list by calling `PIMList.getCategories()`, present them to the user, and allow the user to choose category to use or assign.

Enumerating and Searching for Items

The PIM API provides methods to retrieve all or some of the PIM items on a PIM list. These methods are summarized next:

`items()` - returns an Enumeration of *all* the items on the PIM list

`items(PIMItem matchingItem)` - returns an Enumeration of all the items on the list that exactly *matches all the field values* on the specified `matchingItem`

`items(java.lang.String matchingValue)` - returns an Enumeration of all the items on the list that *matches matchingValue on any one of its fields values*

`itemsByCategory(java.lang.String category)` - returns an Enumeration of all the items on the list *with the specified category*

The following code snippet illustrates how to get all the items on a generic PIM list:

```
/**
 * Get all the PIM items on the specified list.
 * @param pimList the PIM List to retrieve items from
 * @return Enumeration containing the PIM items
 */
static public Enumeration getAllItems(PIMList pimList) {
    Enumeration allItems = null;
    try {
        allItems = pimList.items();
    } catch (PIMException pe) {
        allItems = null;
    }
    return allItems;
}
```

The following code snippet shows how to get all the items on a list that matches a specified matching item:

```
/**
 * Get all the PIM items on the specified list that matches
 * all the fields on the specified matching item.
 * @param pimList the PIM List to retrieve items from
 * @return Enumeration containing the matching PIM items
 */
static public Enumeration getMatchingItems(
    PIMList pimList, PIMItem matchingItem) {
    Enumeration matchingItems = null;
    try {
        matchingItems = pimList.items(matchingItem);
    } catch (PIMException pe) {
        matchingItems = null;
    }
    return matchingItems;
}
```

Before calling the above helper method, the matching `PIMItem` must be constructed. The following code snippet illustrates how to construct a matching Event PIM item, to find items that match the specified event location *and* class:

```
:
// Open the Event list
EventList el = ...;

// Create an instance of an event item
Event matchingItem = el.createEvent();

// Create the matching event, to find items that matches a
// specific location, and item class. For this
only assign values
// to the location and class fields on the matchingItem.
if (el.isSupportedField(Event.LOCATION) == true) {
    matchingItem.addString(Event.LOCATION,
        PIMItem.ATTR_NONE,
        "6th Street, Austin");
}
if (el.isSupportedField(Event.CLASS) == true) {
    matchingItem.addInt(Event.CLASS,
        PIMItem.ATTR_NONE,
        Event.CLASS_PUBLIC);
}

// Now get matching items
Enumeration matchingItems = getMatchingItems(el, matchingItem);

// For illustration purposes, walk through the returned (matched) items
int counter = 0;
while (matchingItems.hasMoreElements()) {
    Event event = (Event) matchingItems.nextElement();
    dumpPimEventItem(event); // dump the event item
    counter++;
}
```

In the above code snippet, the method `dumpPimEventItem()`, not illustrated here, dumps the content of the event item to the screen.

Similarly you can find all the items in a PIM list that match a specific `String` value on *any* of its fields:

```
/**
 * Get all the PIM items on the specified list that match
 * on any of its fields the specified String value.
 * @param pimList the PIM List to retrieve items from
 * @return Enumeration containing the matching PIM items
 */
static public Enumeration getMatchingItemsByStringValue(
```

```

        PIMList pimList, String stringValue){
    Enumeration matchingItems = null;
    try {
        matchingItems = pimList.items(stringValue);
    } catch (PIMException pe) {
        matchingItems = null;
    }
    return matchingItems;
}

```

To use the above method, all you have to do is pass a matching `String` value:

```

:

// Open the Event list
EventList el = ...;

// Create an instance of an event item
String matchingValue = "MobileMonday Austin";

// Get matching items
Enumeration matchingItems = getMatchingItems(el, matchingValue);

// Walk through the returned/matching items
int counter = 0;
while (matchingItems.hasMoreElements()) {
    Event event = (Event) matchingItems.nextElement();
    dumpPimEventItem(event); // dump the event item
    counter++;
}

```

To retrieve all the items on the list with a specific category, use the method `itemsByCategory(...)` as explained in section [Using Categories](#).

More on Contacts, Events and To-Do PIM Items

The PIM API is very extensive, and we have covered quite a bit of information about how to manage PIM lists and items. We have covered how to add, change, remove, and import items, with a focus on `Event`. But there are more methods worth mentioning.

More on Event and EventList

The `EventList` PIM list defines the method `items` to enumerate events that fall (inclusive) within a specified date range. `EventList` also defines the method `getSupportedRepeatRuleFields` to get the supported *repeat rules fields* for a given frequency (daily, weekly and so on). The following code snippet shows how to use the `EventList.items` to retrieve all items within the specified date range:

```

:

// Define the start of vacation date/time range limit
Calendar calStart = Calendar.getInstance();
calStart.set(Calendar.MONTH, Calendar.DECEMBER);
calStart.set(Calendar.DAY_OF_MONTH, 18);
calStart.set(Calendar.YEAR, 2006);
calStart.set(Calendar.HOUR_OF_DAY, 8);
calStart.set(Calendar.MINUTE, 00);
calStart.set(Calendar.SECOND, 00);
calStart.set(Calendar.MILLISECOND, 0);
Date startDate = calStart.getTime();

// Define the end of vacation date/time range limit
Calendar calEnd = Calendar.getInstance();
calEnd.set(Calendar.MONTH, Calendar.DECEMBER);
calEnd.set(Calendar.DAY_OF_MONTH, 24);
calEnd.set(Calendar.YEAR, 2006);
calEnd.set(Calendar.HOUR_OF_DAY, 23);
calEnd.set(Calendar.MINUTE, 00);
calEnd.set(Calendar.SECOND, 00);
calEnd.set(Calendar.MILLISECOND, 0);
Date endDate = calEnd.getTime();

// Now that we have defined the vacation date range above,
// identify/search for items that conflict with the vacation (i.e. that
// fall within the specified date range)
Enumeration e = null;
try {
    e = eventList.items(
        EventList.STARTING, // STARTING, ENDING, or OCCURRING.
        startDate.getTime(),
        endDate.getTime(),
        true); // only return the initial event if recurring
} catch (PIMException pe) {
    // handle PIM exception
} finally {
    return e;
}

```

The `Event` PIM item also defines the method `getRepeat` to retrieve the `RepeatRule` value associated with the `Event`, as well as the method `setRepeat` to set the `RepeatRule` value for the `Event`. Using `RepeatRule` is covered next in section [Using the RepeatRule Class](#).

More on ToDoList

The `ToDoList` PIM list also defines the method `items` to enumerate all the to-do items that fall (inclusive) within a specified data range. Using this method is similar as explained above for `EventList`. The following code snippet shows how to use method `ToDoList.items()` to retrieve all items within the specified date range:

```

:

// Define the start of vacation date/time range limit
Calendar calStart = Calendar.getInstance();
Date startDate = calStart.getTime();

// Define the end of vacation date/time range limit
Calendar calEnd = Calendar.getInstance();
Date endDate = calEnd.getTime();

```

```

:

// Identify/search for items to be completed that conflict with the
// vacation (i.e. that fall within the specified date range)
Enumeration e = null;
try {
    e = todoList.items(
        ToDo.COMPLETION_DATE, // ToDo.DUE or ToDo.COMPLETION_DATE
        startDate.getTime(),
        endDate.getTime());
} catch (PIMException pe) {
    // handle PIM exception
} finally {
    return e;
}

```

More on Contact

The `Contact` PIM item defines method `getPreferredIndex` to retrieve the index of the *preferred value*, as defined by attribute `ATTR_PREFERRED`, for a given field.

Using the RepeatRule Class

A repeat rule is used to define recurring events. A repeat rule has a frequency: daily, weekly, monthly, or yearly. A repeat rule may also have additional fields such as occurrence count, interval, exception dates, and an end-date. Repeat rule attributes are set by calling the appropriate *set methods* based on the field's data-type. The following table, from Part 1 of this series on Using the PIM API), shows the `RepeatRule` fields, methods, and valid values.

Table 2: RepeatRule Fields and Data-types

| Repeat Rule Field Name | Method | Valid Values |
|------------------------|---------|--|
| FREQUENCY | setInt | DAILY, WEEKLY, MONTHLY, YEARLY |
| COUNT | setInt | Any positive int |
| INTERVAL | setInt | Any positive int |
| END | setDate | Any valid Date |
| MONTH_IN_YEAR | setInt | JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER (Note: these are different from ones in Calendar class) |
| DAY_IN_WEEK | setInt | SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY (Note: these are different from ones in CLDC Calendar class) |
| WEEK_IN_MONTH | setInt | FIRST, SECOND, THIRD, FOURTH, FIFTH, LAST, SECONDLAST, THIRDLAST, FOURTHLAST, FIFTHLAST |
| DAY_IN_MONTH | setInt | 1-31 |
| DAY_IN_YEAR | setInt | 1-366 |

As in the case of PIM items, a particular PIM implementation may or not support all the repeat rule fields as defined in the JSR 75 specification. Before using a particular `RepeatRule` field, you *should* test if it is supported by the underlying implementation by calling the `EventList` method `getSupportedRepeatRuleFields(int frequency)`, passing for argument the repeat rule frequency. Attempting to use an unsupported `RepeatRule` field will result in a `java.lang.IllegalArgumentException` being thrown that you must catch.

The repeat rules are very flexible yet powerful. The following code snippet sets a recurring event to occur weekly until November 15 of 2007:

```

:

// Set Start Date/Time
Calendar calStart = Calendar.getInstance();
Date startDate = calStart.getTime();
:

// Set End Date/Time
Calendar calEnd = Calendar.getInstance();
Date endDate = calEnd.getTime();
:

EventList el = ...;
Event event = el.createEvent();
PimEventUtils.addEventInfo(
    event,
    startDate.getTime(), // START
    endDate.getTime(), // END
    "This is the summary", // SUMMARY
    "This is the location", // LOCATION
    "This is the note", // NOTE
    Event.CLASS_PUBLIC, // CLASS
    0); // ALARM

// Define the until date repeat rule
Calendar utilDate = Calendar.getInstance();
utilDate.set(Calendar.MONTH, Calendar.NOVEMBER);
utilDate.set(Calendar.DAY_OF_MONTH, 15);
utilDate.set(Calendar.YEAR, 2007);
utilDate.set(Calendar.AM_PM, Calendar.PM);
utilDate.set(Calendar.HOUR_OF_DAY, 23);
utilDate.set(Calendar.MINUTE, 59);

// Set the recurring event to occur weekly
RepeatRule rr = new RepeatRule();
rr.setInt(RepeatRule.FREQUENCY, RepeatRule.WEEKLY);

// Set the repeat until rule, to end on the specified date/time
rr.setDate(RepeatRule.END, utilDate.getTime().getTime());
event.setRepeat(rr);

try {
    event.commit();
} catch (PIMException pe) {
    // handle PIM exception
}
try {
    el.close();
} catch (PIMException pe) {

```

```

    // handle PIM exception
}

```

Remember that it is a good practice to test if fields are supported before using them:

```

/**
 * Tests if the specified RepeatRule field is supported
 * @param el the EventList to test
 * @param frequency the RepeatRule frequency: DAILY, WEEKLY, MONTHLY, YEARLY
 * @param field the RepeatRule field to test
 * @return true if the field is valid, false if otherwise
 */
public boolean isRepeatRuleFieldSupported(EventList el, int frequency, int field) {
    boolean isSupported = false;
    int[] supportedFields = el.getSupportedRepeatRuleFields(frequency);
    for (int i = 0; i < supportedFields.length; i++) {
        if (supportedFields[i] == field) {
            isSupported = true;
            break;
        }
    }
    return isSupported;
}

```

The following code snippet shows how to use the above helper method to test if a specific repeat rule is supported:

```

:

EventList el = ...;
:

// Test if repeat rule END is supported
boolean isSupported = isRepeatRuleFieldSupported(
    el,                // list to test
    RepeatRule.DAILY, // frequency for repeat rule to test
    RepeatRule.END);   // repeat rule to test

if (isSupported) == true) {
    // RepeatRule.END is supported!
}

:

```

Exception Handling

The PIM API defines and uses Java exceptions throughout the API. These exceptions should be trapped by your applications. Below is a summary of the most important exceptions to consider:

`javax.microedition.pim.FieldEmptyException` - thrown when attempting to access a field that has no associated values

`javax.microedition.pim.FieldFullException` - thrown when attempting to add values to a field which already has reached its maximum number of values. You can use `PIMItem.countValues(fieldID)` and `PIMList.maxValues(fieldID)` to discover the current and maximum values for a given field

`javax.microedition.pim.PIMException` - thrown when a generic PIM API exception is encountered such as if the list is not available, the operation is unsupported or an error occurs

`javax.microedition.pim.UnsupportedFieldException` - thrown when a referenced field is not supported

`java.lang.SecurityException` - thrown when the application is not given permission to access PIM resources

`java.lang.IllegalArgumentException` - thrown when if an invalid argument is used

`java.lang.NullPointerException` - thrown when a null pointer is passed to a method

`java.io.UnsupportedEncodingException` - thrown when/if the specified encoding is supported
Please consult the PIM API specification and Javadoc for more information.

The Sun Java Wireless Toolkit

The [Sun Java Wireless Toolkit](#) is the tool of choice for writing mobile applications that are based on Java ME Connected Limited Device Configuration (CLDC) and Mobile Information Device Profile (MIDP). The toolkit includes the emulation environments, performance optimization and tuning features, documentation, and examples. The toolkit comes with support for the PDA Optional Packages for the J2ME Platform (JSR 75), including support for the PIM API.

Resources

[PDA Optional Packages for the J2ME Platform](#)

[Sun Java Wireless Toolkit](#)

[Internet Mail Consortium Personal Data Interchange vCard and vCalendar web site](#)

Acknowledgements

Many thanks to Maxim Sokolnikov for all the feedback and helping improve this article.

About the Author

C. Enrique Ortiz is a mobile technologist, software architect and developer, and a writer and [blogger](#). He has been author or co-author of many publications, and is an active participant in the Java mobility community and in various Java ME Expert Groups. Enrique holds a B.S. in Computer Science from the University of Puerto Rico and has more than 17 years of software engineering, product development, and management experience.

 E-mail this page  Printer View

ORACLE CLOUD

[Learn About Oracle Cloud](#)

[Get a Free Trial](#)

[Learn About PaaS](#)

[Learn About SaaS](#)

JAVA

[Learn About Java](#)

[Download Java for Consumers](#)

[Download Java for Developers](#)

[Java Resources for Developers](#)

CUSTOMERS AND EVENTS

[Explore and Read Customer Stories](#)

[All Oracle Events](#)

[Oracle OpenWorld](#)

COMMUNITIES

[Blogs](#)

[Discussion Forums](#)

[Wikis](#)

[Oracle ACEs](#)

SERVICES AND STORE

[Log In to My Oracle Support](#)

[Training and Certification](#)

[Become a Partner](#)

[Find a Partner Solution](#)

[Learn About IaaS](#)

[Java Cloud Service](#)


[JavaOne](#)

[User Groups](#)

[Purchase from the Oracle Store](#)

[Java Magazine](#)

[Social Media Channels](#)

 **CONTACT AND CHAT**
Phone: +1.800.633.0738
[Global Contacts](#)
[Oracle Support](#)
[Partner Support](#)

Hardware and Software, Engineered to Work Together



[Subscribe](#) | [Careers](#) | [Contact Us](#) | [Site Maps](#) | [Legal Notices](#) | [Terms of Use](#) | [Privacy](#) | [Cookie Preferences](#) | 