Chapter 4

NOI Part 3: Item, RichTextItem, RichTextStyle, EmbeddedObject, DateTime, DateRange

This chapter goes into more detail about the classes relating to data values and data types, with a fair amount of attention paid to rich text and embedded objects.

The lotus.notes.Item Class

The Item class is where we really start to get nitty gritty about data values, since it is the set of Items in a Document that really hold all the data. NOI considers an Item's value to be an attribute of the Item, so most of the calls in the Item class are properties. I've arranged the set of properties into two groups: value properties and attributes. You'll see what I mean.

Item Value Properties

These calls are equivalent to the Document.getItemValueXXX() calls.

java.util.Vector getValues()

void setValues(java.util.Vector)

Unlike column values or view lookup keys, Items that contain multiple values are always homogeneous with respect to data type. Thus, an Item that contains a text list cannot also contain a number, and vice versa. The Values property is how you access all the values of an Item at one time. The getValues() call always returns a Vector, even if there is only a single value. The setValues call always takes a Vector as input, even if you have put only one value in it. Use the standard Vector methods to discover how many values there are (Vector.size()), and to iterate through them.

String getValueString()

void setValueString(String text)

These calls allow you to retrieve and set text values for an Item. If you call getValueString() on an Item that contains another data type, NOI will attempt to coerce the value to a String for you. If you call setValueString() on an Item that contains a value of a different data type, then that value is replaced and the type of the Item is changed to text. If there is no value, or if the value cannot be coerced, a *null* is returned.

double getValueDouble()

void setValueDouble(double value)

Like the String version of this property, it attempts to coerce the value type on the read side, and overwrites any existing value type on the write side. If the value type cannot be coerced, a 0 is returned.

int getValueInteger()

void setValueInteger(int value)

As with Strings and doubles, the Integer version of the Value Property coerces on retrieval (if possible). If the value cannot be coerced, a 0 is returned.

lotus.notes.DateTime getDateTimeValue()

void setDateTimeValue(lotus.notes.DateTime)

Notes stores DateTime values internally as numbers, so we added a special property to retrieve a date value as a DateTime object. This is really only useful when you already know that the Item contains a DateTime value, or when you want to set an Item to a DateTime value regardless of what data might have been stored there before. If you ask for a DateTime object from an Item that does not contain a date/time value, then a *null* is returned.

DateTime values are pretty much useless represented as numbers, because unlike in LotusScript, where date values are double precision numbers, Notes uses its own internal format, composed of two 4-byte integer values, one for date and one for time (some of the bits are actually allocated to a Daylight Savings Time flag and to a time zone id).

Item Attributes

lotus.notes.DateTime getLastModified()

Returns the date and time that the Item was last modified within the Document. The Item level information on modification is updated when the Document is saved, not when the in-memory value is modified. Thus the Item's LastModified property can never be later than the Document's, although different Items in the same Document can have different LastModified values. If the Document is new (has never been saved), then this call returns *null*.

This is the property Notes uses to implement field level replication.

String getName()

Returns the name of the Item. All Items have a name.

lotus.notes.Document getParent()

Returns the parent Document of the Item. All Items have a parent Document instance.

int getType()

Returns a constant representing the type of the Item's value. These constants are (as usual) declared *public static final int* in the Item class. Most of the item types listed below are either obsolete (from old versions of the product) or belong to various design elements, and as such are not interesting to most developers. They are all listed here anyway, for the sake of completeness. The possible values are:

- ITEM.ACTIONCD. Simple action information for Agents.
- ITEM.ASSISTANTINFO. Agent design data.
- ITEM.ATTACHMENT. A file attachment. Always named $FILE.
- ITEM.AUTHORS. Item is of type Text, but the Authors flag has been set. This means that the Item contains the names of users/groups allowed to read and write the Document.

- ITEM.COLLATION. Special character set collation data.
- ITEM.DATETIMES. Item contains one or more date/time values.
- ITEM.EMBEDDEDOBJECT. Item is part of an embedded OLE or other object in the Document. Usually named $FILE.
- ITEM.ERRORITEM. Error item (obsolete).
- ITEM.FORMULA. Formula item (obsolete).
- ITEM.HTML. Item contains raw HTML text, which has also (often, but not necessarily) been rendered into Notes rich text format elsewhere in this Document.
- ITEM.ICON. Icon item (obsolete).
- ITEM.LSOBJECT. Item contains LotusScript program data.
- ITEM.NAMES. The Item is a names Item, containing user and/or group names. If an Item is of type AUTHORS or READERS, it is also implicitly of type NAMES.
- ITEM.NOTELINKS. Item contains links to other Documents.
- ITEM.NOTEREFS. Item contains the UNID of Document's parent.
- ITEM.NUMBERS. Item contains one or more numeric values.
- ITEM.OTHEROBJECT. Item references an object other than a file attachment or embedded object. Not often seen.
- ITEM.QUERYCD. Item contains a saved query for an Agent.
- ITEM.READERS. Item is a names list containing the names of user/groups allowed read access to the Document.
- ITEM.RICHTEXT. Item contains rich text CD (Composite Document) records.
- ITEM.SIGNATURE. Item contains a signature. Always named $Signature.
- ITEM.TEXT. Item contains text (or a text list).
- ITEM.UNAVAILABLE. Obsolete.
- ITEM.UNKNOWN. Item's type is unknown.
- ITEM.USERDATA. Item is used by an API program (not Notes itself) to store data in a format that Notes does not know about. A "bit bucket" for some application.
- ITEM.USERID. Item contains a user name from Notes Release 2.
- ITEM.VIEWMAPDATA. Viewmap design information.
- ITEM.VIEWMAPLAYOUT. Viewmap design information.

int getValueLength()

Returns the size in bytes of the Item's value. Does not include any overhead for Item level data other than the value.

boolean isAuthors()

void setAuthors(boolean flag)

The Authors property is used to attach a list of users/groups to a Document to provide Document level access control. An Authors Item is usually named $Authors, and contains the list of people allowed to modify the Document.

Setting the Authors property to *true* will set the Names flag as well as the Authors flag on the item. Setting the Authors property to *false* will not clear the Names flag, however.

boolean isEncrypted()

void setEncrypted(boolean flag)

When you encrypt a Document, only the Items marked for encryption actually get encrypted. Use the Encrypted property on the Item class to mark or unmark Items individually.

boolean isNames()

void setNames(boolean flag)

The Names property indicates which Item(s) in a Document contain user/group names. Setting the Authors or Readers property will also set the Names flag on the Item, but you might want to set the Names flag alone. A few places in the Notes UI check for a Names flag on an Item, and will let you do special kinds of address book lookups automatically on the corresponding fields in a form. You can set the Names flag on a form's field in the field property box when you're in form design mode.

boolean isProtected()

void setProtected(boolean flag)

Set this property to *true* if you want only users with Editor (or better) access to be able to modify the Item.

boolean isReaders()

void setReaders(boolean flag)

Similar to the Authors property: defines the list of users/groups who have read access to the Document.

boolean isSaveToDisk()

void setSaveToDisk(boolean flag)

This property is exceptionally useful in those applications where you want to temporarily store data in a Document, but don't want that data written to disk when the Document is saved. The standard mail template uses it extensively for Calendaring and Scheduling.

If the SaveToDisk flag on an Item is *true* (the default), then that Item gets written to disk when the Document is saved. If not, then the Item is simply skipped at Document save() time.

boolean isSigned()

void setSigned(boolean flag)

As with the Encrypted property, when you sign a Document only the Items marked for signing are included in the "digest" of the Document (see the discussion of Document.encrypt() in Chapter 2). Use this property to set or clear the flag for an Item.

boolean isSummary()

void setSummary(boolean flag)

Only Items whose Summary property is set can appear as values in a View, because only Items with their Summary flag set are included in the View's summary data. Author and Reader Items need to have their Summary flags set as well; otherwise, the access control features for which they were designed won't work. Setting either the Authors or Readers property will automatically also set the Names and Summary flags.

Not all Items with the Summary flag set can appear in a View, however. Rich text Items, for example, are explicitly excluded, as are text Items where the length of the text exceeds 15KB. You will also be prevented from setting the Summary flag on any Item whose size exceeds 32KB (the flag is automatically cleared, though no error is raised).

Apart from these restrictions, when you create an Item using NOI, the Summary flag is set for you automatically.

Item Methods

String abstractText(int maxlen, boolean dropvowels, boolean usedict)

This little known method is an interesting way to shrink the contents of a text Item. Users sometimes use it to get around the fact that NOI has a maximum string length of 32,000 characters (64,000 bytes, but when you use Unicode internally it cuts the number of characters you can hold in a 64KB buffer down to 32,000), and many rich text Items contain much more text than that.

The abstractText() call compresses the original text (including rich text) in an Item by optionally dropping all vowels (if you specify *true* for the "dropvowels" parameter), and by attempting to replace words with common abbreviations. It optionally does the abbreviating using a dictionary file if you specify *true* for the "usedict" parameter). The dictionary file is a simple text file, formatted with each word and its abbreviation on a single line, separated by at least one space. The entries should be in alphabetical order. The file is named “noteabbr.txt” (in lowercase for those operating systems that care), and must be somewhere on your path (your execution path, not the Notes data directory). A sample dictionary is included on the CD, you can add your own abbreviations at will.

AbstractText() also automatically trims whitespace (compressing multiple spaces into a single space, and so on), and trims punctuation where possible.

The method will perform its magic on the first 64KB of text in the Item.

void appendToTextList(String value)

void appendToTextList(java.util.Vector textlist)

This method takes a String, or Vector of Strings, and appends it to the current Item's text or text list. The Item must by of type Text or TextList for this call to work. If the original Item value was only a single String, then this call will convert the Item into a text list. If the original Item contained a String or a text list, then the new String or text list is appended to it.

boolean containsValue(Object value)

Returns *true* if the Item contains the value that you pass in as a parameter. The data type of the object that you pass in doesn't need to be exactly the same as the data type of the Item, but they must be compatible. For example,

- **Text** is compatible with rich text, text, and text list.
- **Number** is compatible with number and number list.
- **DateTime** is compatible with DateTime or DateRange.

The rules for a match vary somewhat with data type:

- **Numbers.** If the Item contains a number, the two must match exactly. If the Item contains a number list, the input argument must be one of the numbers in the list.
- **DateTime.** If the Item contains a DateTime value, the two must match exactly. If the Item contains a DateTime list, the input argument must be one of the values in the list.
- **Text.** If the Item contains rich text, the input String must be a substring in the rich text stream. If the Item contains a single String, then the two must match exactly. If the Item contains a text list, then the input argument must exactly match one of the elements in the list (comparisons are case- and accent-sensitive). One special case: if the Item's Name flag is set (the

Item contains one or more user/group names), then the match will succeed if the input argument is a common name that either matches exactly or matches the common part of a distinguished name in the Item. For example, if the Item is a Name Item and contains "CN=Bob Balaban/O=Looseleaf" and I call Item.contains() with an input argument of "Bob Balaban", then the match will succeed.

lotus.notes.Item copyItemToDocument(lotus.notes.Document destination)

lotus.notes.Item copyItemToDocument(lotus.notes.Document destination, String newname)

This method makes a copy of the current Item in the specified Document. The Item's flags (Summary, Name, etc.) as well as its value are copied. By default the new Item will keep the same name as the original, but you can also specify a new name for it. No checking is done to see whether the destination Document already contains an Item of that name. If it does, you will end up with a Document containing multiple Items of the same name which, while okay for things like file attachments, is a definite no-no for data Items.

You may have problems with this call if you use it on a RichTextItem. The problem will arise if the source and destination Documents do not have the same font layouts: The rich text may have different fonts in the destination than it did in the source. The reason for this has to do with the way font information is stored. Each Document has a special Item (named $Fonts) containing a font table. Each font name in the table is associated with a font index, and the table's scope is the entire Document. Any font settings contained in a RichTextItem will contain only the font's index into the table, not the full font information. When you copy a RichTextItem from one Document to another, the $Fonts tables are not merged. Thus, if the destination Document has a different font table layout than the source Document (a likely occurrence, unless you're lucky), the font indices in the copied Item will be wrong. This is a bug that is scheduled to be fixed in Domino Release 5.0. One possible workaround is to explicitly copy the

$Fonts Item at the same time. That won't work, though, in cases where the Document has more than one RichTextItem in it.

When you copy a RichTextItem, all of its embedded objects (file attachments, OLE objects) are copied as well.

String getText()

String getText(int maxlen)

The getText() method returns a text version of the contents of the Item. It operates a bit differently from the getValueString() call, however. Both operate the same when the Item contains rich text, but when the Item contains a text list, getValueString() will only return the first element of the list, while getText will return the entire list, with each element separated by the default text list separator (usually a semicolon).

The getText() call also has an option that limits the length of the returned String to a specified maximum. This is particularly useful when you're accessing rich text and only want a small part of what might be a large String. The maximum length is specified in characters (not bytes).

void remove()

Removes an Item from the current Document. If the Item is a multi-part RichTextItem (see the discussion on Document.getFirstItem() in Chapter 3 for details on multi-part Items), all parts are removed.

The lotus.notes.RichTextItem Class

The RichTextItem class is the only instance in NOI of class inheritance. RichTextItem extends (the Java term for inherits from) the Item class, meaning that all of the methods and properties that you find in an Item are also available to RichTextItem. RichTextItem also adds some additional methods and properties that are specific to manipulating rich text, and those are what we document in this section.

The RichTextItem class is not as fully functional with Notes rich text as anyone (including the developers of the interface) would like. It will continue to evolve, though, and it does allow you to examine certain aspects of a rich text item, and to add new text, styles, doclinks, and file attachments to them.

RichTextItem Properties

java.util.Vector getEmbeddedObjects()

This is the only property in the RichTextItem class (other than the ones that RichTextItem inherits from Item). The getEmbeddedObjects() call returns a Vector containing all of the embedded objects (including file attachments, OLE/1 objects and OLE/2 objects) in the RichTextItem. Your program does not have to be running on a Windows machine in order to successfully instantiate EmbeddedObjects for OLE objects—it will work on any platform. In LotusScript NOI you do have to be running on a Windows platform in order to activate an OLE object, because in order to run an OLE object the Microsoft OLE libraries have to be available.

Unfortunately, in Java NOI (for Domino 4.6) OLE isn't supported. See the discussion of the EmbeddedObject class later in this chapter for all the gory details.

RichTextItem Methods

void addNewLine()
void addNewLine(int n)
void addNewLine(int n, boolean newparagraph)

This method adds a newline to the rich text stream. You can specify how many newlines to add, and you can also specify (using the *newparagraph* parameter) whether the newline(s) acts as a paragraph break or not (the default value for newparagraph is *true*).

Why should you care about the distinction between newlines and new paragraphs? Usually, you won't need to care, but you should also know that Notes does make the distinction, and that a single paragraph in a RichTextItem cannot hold more than 64KB worth of data. If you were to simply append text for a long time either without adding newlines, or adding newlines that were not paragraph breaks, Notes would at some point most likely insert a paragraph break for you, and the resultant rendering for that RichTextItem in the UI might look odd. Or, you might get an exception for the paragraph exceeding 64KB.

Using the addNewLine() call is the recommended way to insert newlines. Explicitly including constructs such as "\n" or "\n\r" in your text stream is not recommended, as these tend to be platform-specific values (a newline in Win95 is not the same as a newline on the Mac, for example). The addNewLine() method is guaranteed to work correctly for all platforms.

void addTab()

void addTab(int n)

This call adds one or more tab characters to the rich text stream. You should use this method instead of explicitly adding constructs like "\t" to your text.

void appendDocLink(lotus.notes.Document doc)

void appendDocLink(lotus.notes.Document doc, String comment)

This call adds a Notes doclink to the specified Document instance to the current RichTextItem. Due to a last minute problem during the development of Domino 4.6 (the developer, yours truly, screwed up a bit), the interface to this method was not correctly specified, and there was no time to correct the error before the product shipped. There should have been three sets of methods here: one each for adding a link to a Document, View and Database. In LotusScript the argument is a Variant, so passing in any of these three object types works. But because Java is strongly typed, there's no way to fool the call into accepting a View or a Database. Sorry.

The doclink is rendered as the standard link icon in the RichTextItem, and contains a server "hint" as well to assist in link resolution later. The server hint is taken from the input Document's parent Database. You can optionally add a comment to the link as well, which displays in the status bar of the UI when you highlight the link.

void appendRTItem(lotus.notes.RichTextItem item)

Use this method to merge a RichTextItem (could be from the same Document or from another one) into the current RichTextItem. Be aware, however, that you may have font problems, as described above in this chapter for the Item.copyToDocument() method.

void appendStyle(lotus.notes.RichTextStyle style)

Adds a new RichTextStyle to the current rich text stream. It allows you to modify the font, color, size, and so on of text as you append it to the RichTextItem. See below for more details on the RichTextStyle class (new in Release 4.6).

void appendText(String text)

Adds text to the current RichTextItem. Text is always appended to the end of the stream—there is no way (currently) to insert or modify text in the middle of the Item. You should not use explicit tabs ("\t") or newlines ("\n" or "\n\r") in the text stream, as they are not always platform portable. Instead use the addTab() and addNewLine() methods.

lotus.notes.EmbeddedObject embedObject(int type, String classname, String source, String name)

Adds an EmbeddedObject to the RichTextItem. EmbeddedObject instances encompass file attachments as well as OLE objects. Unfortunately, OLE doesn't really work with the Java NOI in Release 4.6 (should be fixed in 5.0), so this method in 4.6 is limited to embedding file attachments. See the following section on the EmbeddedObject class for more details on the problems with OLE.

The first argument to the embedObject() call is a constant which specifies the type of embedding you want:

- RichTextItem.EMBED_ATTACHMENT. Embed a file attachment.
- RichTextItem.EMBED_OBJECT. Embed an OLE object.
- RichTextItem.EMBED_OBJECTLINK. Embed a link to an OLE object.

Embedding a file attachment simply means that the file is attached to the document, and an icon for it is rendered into the RichTextItem at the current location. The icon doesn't look exactly the way it does when you attach a file using the Notes UI: You get the same icon, but the name of the file doesn't appear underneath it; you get the name following the icon. The reason for this is that the UI generates the icon/name rendering using a graphical metafile, and the back-end NOI classes don't have the ability to do that. Instead we just stick a generic icon in there and add the name of the file following it.

The difference between embedding an OLE object and embedding an OLE object link is that in the first case the entire object (and all its instance data) is attached to the Document, whereas in the link case only a pointer to a file on disk is stored in the Document. The advantage of a link is that it uses much less space in the Database, however you can't really make use of a link across replicas. Since the link is to a file on disk, everyone who accessed the link would have to have the original file in the same location on his or her own disk, not a very easy setup to maintain.

The "classname" argument is used for OLE objects only. It allows you to specify an application (say, 1-2-3 or Excel) and have NOI create an "empty" embedded object of that class in the RichTextItem. If you don't want an empty embedding, then you specify *null* for the classname, and provide an explicit file path for the "source" argument. The file type is checked against the OLE registry, and if it is a file belonging to a valid OLE application installed on your machine, then OLE will (in a future release of Java NOI, remember, though it does work fine in LotusScript) launch the application, load the file, and re-save it in your Document. You cannot specify both a class name and a source path (when you specify a file, the class is implied).

The third argument, "name," is the name by which you want the embedded object to be known in Domino. This is the name that the UI shows you (right-click on the embedded object in the UI and bring up the Object Properties box), and the name you can use in the getEmbeddedObject() call to retrieve it. It can be any string you like, though the name should be unique within the Document.

lotus.notes.EmbeddedObject getEmbeddedObject(String name)

Locates the EmbeddedObject instance in the current RichTextItem of the name you specify, and returns it. The name should be the user-defined name of the object, which you can specify in the embedObject() call, or edit in the Object Properties box in the UI.

String getFormattedText(boolean striptabs, int linelength, int maxlength)

This method converts a RichTextItem to a text only representation, and allows you to specify whether tabs should be removed (each tab is converted to one space), how long each line should be (newlines are inserted at the end of each line), and the maximum amount of text you want returned. If you specify 0 for the line length, a length of 80 is used.

The maximum length that NOI will return for this call is 32,000 Unicode characters (64KB).

The lotus.notes.RichTextStyle Class

This class is new in Release 4.6, and (finally!) adds functionality to NOI that allows you to manipulate rich text styles when appending text to a RichTextItem. Following the *append* mode of interaction with RichTextItem, you use the RichTextItem.appendStyle() call to add a style to the rich text stream. All text appended following the style will be rendered with that style.

RichTextStyle has no methods, only properties. All properties are read-write. There are a number of constants that I've documented separately. As usual, all constants are defined as *public static final int* in the RichTextStyle class, so you would reference them using the RichTextStyle. prefix. The reason for using constants for colors and so on (instead of, say, allowing you to specify RGB combinations) is that this was the only way to guarantee platform portability: We wanted you to be confident that you would get the right color on all operating systems.

Can you use any number, or are you just limited to the predefined constants? The actual answer is that, for colors and fonts at least, you are free to use any integer you like. The problem is, though, that only the pre-defined ones are platform portable. Be aware that if you find some color or font that you like to use on, say, Windows, that isn't part of the predefined set, then that's okay, but it might very well display differently on OS/2 or the Mac.

Note that if you set a RichTextStyle into a RichTextItem and then invoke RichTextItem.appendRichTextItem(), the incoming Item (which has its own styles in it) will not be affected by the RichTextStyle.

You instantiate a RichTextStyle object using the Session.createRichTextStyle() call. Style objects are scoped to the Session so that you can reuse them in multiple Databases.

Note that the RichTextStyle class was omitted from the Java Programmer's Guide database distributed with Domino 4.6, but the LotusScript version of the interface is documented in the online help.

RichTextStyle Constants

Color constants are listed as follows:

- RichTextStyle.COLOR_BLACK
- RichTextStyle.COLOR_BLUE
- RichTextStyle.COLOR_CYAN
- RichTextStyle.COLOR_DARK_BLUE

- RichTextStyle.COLOR_DARK_CYAN
- RichTextStyle.COLOR_DARK_GREEN
- RichTextStyle.COLOR_DARK_MAGENTA
- RichTextStyle.COLOR_DARK_RED
- RichTextStyle.COLOR_DARK_YELLOW
- RichTextStyle.COLOR_GRAY
- RichTextStyle.COLOR_GREEN
- RichTextStyle.COLOR_LIGHT_GRAY
- RichTextStyle.COLOR_MAGENTA
- RichTextStyle.COLOR_RED
- RichTextStyle.COLOR_WHITE
- RichTextStyle.COLOR_YELLOW

Font effects are listed as follows:

- RichTextStyle.EFFECTS_EMBOSS
- RichTextStyle.EFFECTS_EXTRUDE
- RichTextStyle.EFFECTS_NONE
- RichTextStyle.EFFECTS_SHADOW
- RichTextStyle.EFFECTS_SUBSCRIPT
- RichTextStyle.EFFECTS_SUPERSCRIPT

Font names are listed as follows:

- RichTextStyle.FONT_COURIER
- RichTextStyle.FONT_HELV
- RichTextStyle.FONT_ROMAN

Other constants are:

- RichTextStyle.STYLE_NO_CHANGE
- RichTextStyle.YES
- RichTextStyle.NO
- RichTextStyle.MAYBE (equivalent to STYLE_NO_CHANGE)

RichTextStyle Properties

int getBold()

void setBold(int setting)

Retrieves or turns the bold text attribute on or off (use YES or NO), or explicitly carries over the bold setting of the most recent style object added to the rich text stream (use STYLE_NO_CHANGE). Default is STYLE_NO_CHANGE.

int getColor()

void setColor(int color)

Retrieves or sets the current color (use one of the color constants), or explicitly tells NOI to carry over the most recent color setting (STYLE_NO_CHANGE). Default is STYLE_NO_CHANGE.

int getEffects()

void setEffects(int value)

Retrieves or sets one of the special effects settings (use one of the EFFECTS constants), or carries over the most recent setting (STYLE_NO_CHANGE). Default is STYLE_NO_CHANGE.

int getFont()

void setFont(int font)

Retrieves or sets the font that is used for text (use one of the font name constants), or carries over the most recent font (STYLE_NO_CHANGE). Default is STYLE_NO_CHANGE.

int getFontSize()

void setFontSize(int size)

Retrieves or sets the font size (in points) or maintains the most recent setting (STYLE_NO_CHANGE). Default is STYLE_NO_CHANGE. The valid values are between 1 and 250, inclusive.

int getItalic()

void setItalic(int value)

Retrieves or sets the italic property of the text stream (use YES or NO) or maintains the most recent setting (STYLE_NO_CHANGE). Default is STYLE_NO_CHANGE.

int getStrikeThrough()

void setStrikeThrough(int value)

Retrieves or sets the strike-through attribute (use YES or NO) or maintains the most recent setting (STYLE_NO_CHANGE). Default is STYLE_NO_CHANGE.

int getUnderline()

void setUnderline(int value)

Retrieves or sets the underline attribute (YES or NO) or maintains the most recent setting (STYLE_NO_CHANGE). Default is STYLE_NO_CHANGE.

Since this is a new class, let's do a simple example (see Listing 4.1).

Listing 4.1 Rich Text Style Example (Ex41RTStyle.java)

```
import java.lang.*;
import java.util.*;
import lotus.notes.*;

public class Ex41RTStyle
{
    public static void main(String argv[])
        {
        try {
            NotesThread.sinitThread();
            Session s = Session.newInstance();
            Database db = s.getDatabase("",
"book\\Ex41.nsf");
            Document doc = db.createDocument();
            RichTextItem rti =
doc.createRichTextItem("body");

// first style
            RichTextStyle style1 = s.createRichTextStyle();
            style1.setBold(RichTextStyle.YES);
            style1.setColor(RichTextStyle.COLOR_DARK_CYAN);
            style1.setEffects(RichTextStyle.EFFECTS_EMBOSS);
            style1.setFont(RichTextStyle.FONT_ROMAN);
            style1.setFontSize(24);

            // second style
            RichTextStyle style2 = s.createRichTextStyle();
            style2.setBold(RichTextStyle.NO);
            style2.setColor(RichTextStyle.COLOR_DARK_RED);

style2.setEffects(RichTextStyle.EFFECTS_EXTRUDE);
            style2.setFont(RichTextStyle.FONT_HELV);
            style2.setFontSize(18);

            rti.appendText("First line is default
everything");
            rti.addNewLine();
            rti.appendStyle(style1);
            rti.appendText("This text is in style 1.");
            rti.addNewLine();
            rti.appendStyle(style2);
            rti.appendText("This text is in style 2.");

// save it
            doc.save();
            } // end try
        catch (Exception e) { e.printStackTrace(); }
        finally { NotesThread.stermThread(); }
    }  // end main
}  // end class
```

If you examine the document created by this program in the Ex41.nsf database on your CD, you'll see that we got what might be some unexpected behavior with respect to special effects. In style1, I turned on the EMBOSS effect and used that for the second line of text. In style2, I set the effect to EXTRUDE, which you might expect to replace the EMBOSS setting, as they are both set with the same call. However, the third line of text comes out with both EXTRUDE and EMBOSS set (you can verify this by editing the document, bringing up the Text Properties box and moving the cursor between the two lines).

Luckily we can get a behavior by adding a call to RichTextItem.appendStyle() using a third RichTextStyle instance whose Effects property has been set to EFFECTS_NONE before appending style2.

The lotus.notes.EmbeddedObject Class

As mentioned above, the EmbeddedObject class encapsulates both file attachments and OLE objects. While the Java NOI fully supports manipulation of file attachments through this class, unfortunately (at least in Domino Release 4.6) OLE object activation (required for embedding and activating embedded objects and links) is not supported in Java (it is fully supported via LotusScript). There were two reasons for this limitation in 4.6:

- OLE requires that each thread on which OLE calls will be made must not only initialize OLE (no big deal there), but each thread must also implement a Windows message pump (usually implemented as a GetMessage()/DispatchMessage() loop). If the message pump is not run on each thread, that thread's message queue can get backed up, and the thread will eventually hang. This is because OLE uses cross-process messaging to communicate between the container program (usually Notes) and the embedded object, especially when the embedded object or control is an EXE ("out of process") application, as opposed to a DLL ("in process") control or Active/X. This required some nontrivial architectural changes in Notes, and there just wasn't time to complete them before Release 4.6 shipped.
- The second reason had to do with the scriptability of embedded OLE objects in Java. With LotusScript, the language has built-in OLE Automation capabilities, just as Visual Basic does. You can get an

Automation "handle" (an IDispatch interface, for those of you who know about COM interfaces and OLE) to any embedded object just by "activating" it, which causes OLE to load and run it. Using the Automation handle, LotusScript can transmit commands that are specific to the embedded application (or control) through the IDispatch interface to the object. This allows you a very nice scripting capability directly from LotusScript. Java, however, doesn't have anything like that built in. Even if we had been able to solve the OLE-per-thread problem in time, we would have had to invent an entirely new IDispatch like interface in Java in order for you to be able to manipulate embedded objects. That was just too much work given the time we had, especially since the forthcoming Java Beans/COM Bridge architectures that are coming out soon will solve the Automation problem for us.

Hopefully Domino 5.0 will do a much better job with OLE. In the meantime, some of the methods and properties continue to work in the Java NOI. You create a new EmbeddedObject instance with the RichTextItem.embedObject() call (all EmbeddedObjects live in a rich text item). You can also access existing EmbeddedObjects via the RichTextItem.getEmbeddedObjects(), RichTextItem.getEmbeddedObject() and Document.getEmbeddedObjects() calls.

EmbeddedObject Properties

String getClassName()

Returns the name of the OLE class of the embedded object, if it is an OLE object; it returns *null* for other types of objects. This call works in Java NOI, on any platform, so long as the class is known to Domino.

int getFileSize()

Returns the size of an attached file, in bytes. If you make this call on an EmbeddedObject instance that represents an OLE object, you will probably get misleading results. That's because OLE objects store their data in $FILE Items in a Notes

Document, just as file attachments do. The problem is that OLE objects most always use more than one $FILE item, and the getFileSize() call isn't aware of that.

String getName()

Returns the user-defined name of the file attachment or embedded object. Works for OLE objects on all platforms, if a user-defined name was supplied when the object was embedded.

int getObject()

Returns the OLE Automation handle for an embedded OLE object. Does not work in the Java NOI.

lotus.notes.RichTextItem getParent()

Returns the RichTextItem in which the current object is embedded/attached.

String getSource()

Returns the file name of the original file attachment or OLE object. Works on all platforms for OLE objects.

int getType()

Returns a constant representing the kind of object that is embedded/attached, one of: EmbeddedObject.EMBED_ATTACHMENT, EmbeddedObject.EMBED_OBJECT, EmbeddedObject.EMBED_OBJECTLINK. These constants are identical to the ones described above in the RichTextItem class. They were declared in both classes only for convenience.

java.util.Vector getVerbs()

Returns a Vector containing the list of OLE "verbs" supported by the embedded object. This call has no meaning for file attachments. Because it requires activation of the embedded object, this call does not currently work in the Java NOI.

EmbeddedObject Methods

int activate(boolean show)

Activates an embedded OLE object and returns an Automation handle for it. The argument specifies whether the OLE object should create a separate window to display its UI ("show" set to *true*), or to activate in-place ("show" = *false*).

This method does not currently work in the Java NOI.

void doVerb(String verb)

Execute one of the supported "verbs" in the embedded object (typical supported verbs are Open and Edit). Does not currently work in the Java NOI.

void extractFile(String filepath)

For file attachments only, make a copy of the attachment on disk, at the specified location.

void remove()

Removes the embedded object/attachment from the RichTextItem, including its rendering. Works for OLE objects on all platforms. You must invoke save() on the Document in order to commit the changes to disk.

String toString()

Returns the user-defined name (if any) of the object.

The lotus.notes.DateTime Class

The DateTime class represents the internal format of a Notes date value, which includes a date, a time, a time zone and a flag indicating whether daylight savings time is in effect. The methods and properties of this class allow you to do several kinds of date arithmetic (add/subtract months, days, hours, and so on), and to convert between the internal format and Strings, or between the internal format and the LotusScript format.

It seems logical to wonder why there's no conversion between the Notes format and the Java Date class, which actually has a lot of the same functionality (though the Java implementation was kind of buggy, at least in Release 1.1.1). The reason is that the Java date format is an 8-byte integer (a Java *long*), representing the number of milliseconds since 1/1/1970 00:00:00 GMT. There just wasn't time to write a platform portable conversion routine that would handle this data type, which is not native to all platforms supported by Notes. In the meantime, you can accurately convert between Notes and Java native date formats by using Strings.

The LotusScript date format (in case you were curious) follows the BASIC convention. It is a double precision number, where the integer part is the number of days since the reference date (12/30/1899), and the fraction is the ratio of the number of seconds since midnight to the number of seconds in a day. Clearly, this format is not as granular, or as accurate, as either the Java or Notes formats.

Note that as a general rule with this class, all String formatted DateTime values follow the default formatting as set for the current location, and usually include a time zone designation (EDT, EST, whatever). Input Strings should also follow the local format, and can either include or omit the time zone designation.

Notes always converts a DateTime value to GMT for storage internally, though it remembers the time zone in which the original was specified. The value is accessible either in GMT or in the local time zone.

You create a DateTime object using the Session.createDateTime() call.

DateTime Properties

String getDateOnly()

String getTimeOnly()

Returns either the date portion or the time portion of the DateTime in String format.

String getGMTTime()

String getLocalTime()

void setLocalTime(String time)

Returns the current DateTime value in String format for either the GMT or local version of the DateTime value. You can also modify the stored date value using the setLocalTime() call.

Note that if a date value was originally specified in a time zone different from the local time zone, the getLocalTime() and getGMTTime() calls convert that value to their respective zones. To get the DateTime as originally specified, you can use the getZoneTime() call.

int getTimeZone()

Returns the time zone in which the date value was originally specified. The time zone value is usually (but not always) a number of hours plus or minus that you add to the current time to get Greenwich Mean Time (GMT). Some zones have special values, however.

String getZoneTime()

Returns the String version of the DateTime value for the time zone in which it was originally specified. If the original value was not specified with a time zone different from the current zone, then this call is equivalent to getLocalTime().

boolean isDST()

Returns *true* if daylight savings time is in effect for the current DateTime value.

DateTime Methods

void adjustSecond(int value)

void adjustSecond(int value, boolean localzone)

void adjustMinute(int value)

void adjustMinute(int value, boolean localzone)

void adjustHour(int value)

void adjustHour(int value, boolean localzone)

void adjustDay(int value)
void adjustDay(int value, boolean localzone)
void adjustMonth(int value)
void adjustMonth(int value, boolean localzone)
void adjustYear(int value)
void adjustYear(int value, boolean localzone)

The adjustXXX() methods all work essentially the same way: given the current DateTime value, adjust it (plus or minus) by the value provided. Optionally, you can specify that the local time zone should be taken into account in the adjustment. This becomes important when an adjustment (either forward or backward) results in the start and end times being on opposite sides of a Daylight Savings Time boundary. In such a case, if you don't explicitly use *true* for the localzone parameter, you might end up with unexpected results.

For example, if you started with a DateTime of March 15, 1997 2:00:00 PM and adjusted by one month (without taking time zones into account), you'd end with a simple adjustment to the month part of the value: April 15, 1997 2:00:00 PM, even though Daylight Savings Time went into effect during that month.

If, however, you used the version of the adjustMonth() call that lets you specify *true* for the localzone parameter, 3/15/97 2:00:00 PM (implicitly Eastern Standard Time) becomes 4/15/97 1:00:00 PM Eastern Daylight Time. The second case is more accurate, in the sense that the second date is exactly one month ahead of the first, while in the first case the second date is one month and one hour ahead. There are cases, especially in Calendaring applications, where the distinction is important.

void convertToZone(int newzone, boolean isdst)

Converts the current DateTime value to a new time zone. You can optionally specify whether Daylight Savings Time should be considered.

void setAnyDate()

void setAnyTime()

Notes implements the concept of date and time value *wildcards*, which not all systems do. LotusScript (and VB), for example, make no distinction between a date value with "no time" attached to it, and that same date at midnight (0 time value). Notes does make that distinction, and you can have a date value with no time attached, or a time value with no date attached.

For example, you can create a date-only DateTime object by calling Session.createDateTime("today") (or your local language equivalent keyword). This date value is most definitely not the same as "today 00:00:00" (or any other specific time). Likewise for time values without dates.

You can convert a DateTime value that contains both a date and a time to one that contains only one or the other with these two methods. The setAnyDate() call converts whatever date value is in the current object to "no date," and the setAnyTime() call converts the current time value to "no time." Strings formatted from a DateTime object containing one of these wildcards will simply omit the wildcard portion.

void setLocalDate(int year, int month, int day, boolean isdst)

void setLocalTime(int hour, int minute, int second, int hundredth)

These two calls are not in the LotusScript interface, as LotusScript has built-in functions that do the same thing. They allow you to specify a date and/or time value by providing the components of the value in integer format. The year should be a four-digit year, to avoid unexpected results with Year 2000 default conversions. January is always 1.

void setNow()

Store the current date/time as the value of the object.

int timeDifference(lotus.notes.DateTime t2)

Returns the number of seconds between the current DateTime value and the one provided as an argument (subtracts the value of t2 from the current object's value). The two values are first normalized to a common time zone.

String toString()

Returns the result of the getLocalTime() method.

The lotus.notes.DateRange Class

A DateRange simply represents a pair of DateTime objects, referred to as the start and end times. It provides a convenient way to format ranges as well. You create a DateRange object using the Session.createDateRange() call. If you use the flavor of createDateRange that takes no arguments, you get an "empty" DateRange. If you use the flavor that takes start and end DateTime objects, those objects are linked to the DateRange. If you modify either DateTime object after instantiating the DateRange, the value of the DateRange implicitly changes too. Be careful!

DateRange has no methods.

DateRange Properties

lotus.notes.DateTime getStartDateTime()

void setStartDateTime(lotus.notes.DateTime)

lotus.notes.DateTime getEndDateTime()

void setEndDateTime(lotus.notes.DateTime)

Accepts or returns the starting or ending DateTime object. If you use this technique to set the start and end values, then the DateTime objects are linked to the DateRange. When you (or any NOI method that takes a DateRange as input) accesses the value of the range, the current values for the starting and ending DateTime instances are used. Thus you can change the values of either DateTime object after linking it to the range, and the range's value implicitly changes.

Be sure not to modify the value of the range unintentionally, and also be sure not to make the starting time later than the ending time.

String getText()

void setText()

Sets or retrieves a value for the DateRange in text format. The text format for a date range is two DateTime strings (in whatever local date/time formats are supported), separated by a hyphen.

If you set the value of the range using a String, then any previously linked DateTime objects are unlinked, and new ones are generated. Thus, the following code fragment will correctly modify the ending value of the DateRange:

```
DateRange dr = s.createDateRange();
dr.setText("1/1/97 12:01 AM - 10/13/97 3:29 PM");
DateTime enddt = dr.getEndDateTime();
enddt.adjustDay(1);
```

String toString()

Returns the value of DateRange.getText().

Summary

Next, Chapter 5 continues with a discussion of still more NOI classes: Agent, AgentContext, International, Form, and Name.