



CHAPTER 11

Working with collections

- 11.1 What is a collection? 130
- 11.2 DocumentCollection class 130
- 11.3 Creating a DocumentCollection object 131
- 11.4 Properties 131
- 11.5 Accessing documents in a collection 132
- 11.6 Refining a collection 135
- 11.7 Adding/removing a document to/from a collection 136
- 11.8 Working with folders 138
- 11.9 Updating documents in a collection 139
- 11.10 stampAll 139
- 11.11 Working with profiled documents 140
- 11.12 Sorting a collection 140
- 11.13 Newsletter class 143
- 11.14 Chapter review 143

11.1 WHAT IS A COLLECTION?

A document collection resembles a `Vector` or `Enumeration` object. It is an object that contains a group of documents assembled using various methods in the Domino Java classes. A collection of documents can be formed with all documents from a database, a subset of documents from a view, a group of profile documents, all of a document's responses, or as the result of a search.

11.2 DOCUMENTCOLLECTION CLASS

The `DocumentCollection` class works with collections of documents in the Domino Java classes. It contains a few properties and a number of methods for working with documents in the collection. You can instantiate a `DocumentCollection` object in any one of a variety of ways.

11.3 CREATING A DOCUMENTCOLLECTION OBJECT

The AgentContext, Database, Document, and View classes all provide the means to create a DocumentCollection object. They have the following formats:

- DocumentCollection = AgentContext.getUnprocessedDocuments();
- DocumentCollection = AgentContext.UnprocessedFTSearch (“Search Query”, maximum returned);
- DocumentCollection = AgentContext.UnprocessedSearch (“Search formula”, DateTime cutoff, maximum returned);
- DocumentCollection = Document.getResponses();
- DocumentCollection = Database.FTSearch (“Search Query”, maximum returned);
- DocumentCollection = Database.getAllDocuments();
- DocumentCollection = Database.getProfileDocCollection (“Profile name”);
- DocumentCollection = Database.search (“Search formula”, DateTime cutoff, maximum returned);
- DocumentCollection = View.getAllDocumentsByKey (Vector/Object key value, boolean exact match);

We will explore a number of the formats in the various examples in this chapter and throughout the rest of the book.

11.4 PROPERTIES

A number of properties in the DocumentCollection class prove invaluable when you work with a collection of documents.

11.4.1 Count

The getCount method returns the number of documents in a DocumentCollection object:

```
DocumentCollection dc = db.getAllDocuments();
for (int x = 1; x < dc.getCount(); x++) {
    Document doc = dc.getNthDocument(x); }
```

11.4.2 isSorted

The isSorted property signals true/false depending on whether or not a DocumentCollection object is sorted. It will be sorted only when it is returned as the result of a full text search. The next snippet of code will return “true” for this property:

```
DocumentCollection dc = db.FTSearch("FIELD FORM=\"Test\"", 0);
if (dc.isSorted()) {
    System.out.println("The collection is sorted."); }
```

11.4.3 Parent

The `parent` property of the `DocumentCollection` object returns a `Database` object containing the `DocumentCollection` object:

```
Session s = getSession();
AgentContext ac = s.getAgentContext();
DocumentCollection dc = ac.getUnprocessedDocuments()
Database dcParent = dc.getParent();
```

11.4.4 Query

The `query` property of a `DocumentCollection` object represents the query used to assemble a collection. It returns a value for only those `DocumentCollection` objects formed from a full text search or regular search.

```
DocumentCollection dc = db.FTSearch("FIELD FORM=\"Test\"", 0);
String query = dc.getQuery();
```

11.5 ACCESSING DOCUMENTS IN A COLLECTION

You can use a number of methods available in the `DocumentCollection` class to access individual documents in a collection. They allow you to move forward, backward, to the first document, to the last document, and to a specified document in the collection. Let's take a closer look at these methods.

11.5.1 `getDocument`

The `getDocument` method searches a collection for the specified `Document` object. It accepts only one parameter: the `Document` to find. A null value is returned if the `Document` is not found. This method is useful for comparing collections and determining if documents in one collection are in the other.

```
Document = DocumentCollection.getDocument(Document);
```

11.5.2 `getFirstDocument`

The `getFirstDocument` method retrieves the first `Document` object from a collection. It accepts no parameters and returns a null value if the collection is empty.

```
Document = DocumentCollection.getFirstDocument();
```

11.5.3 `getLastDocument`

The `getLastDocument` method retrieves the last `Document` object from a collection. It accepts no parameters and returns a null value if the collection is empty.

```
Document = DocumentCollection.getLastDocument();
```

11.5.4 `getNextDocument`

The `getNextDocument` method retrieves the next `Document` object from a collection using the `Document` object passed into it as the reference point. A null value is returned if there are no more documents in the collection.

```
Document = DocumentCollection.getNextDocument(Document);
```

Example 11.1 uses the `getDocument`, `getFirstDocument`, and `getNextDocument` methods to determine how many documents are in two different collections.

Example 11.1 (Domino Agent)

```
import lotus.domino.*;
import java.util.*;
import java.io.PrintWriter;
public class Example_11_1 extends AgentBase {
    public void NotesMain() {
        try {
            Session session = getSession();
            PrintWriter pw = getAgentOutput();
            AgentContext ac = session.getAgentContext();
            Database db = ac.getCurrentDatabase();
            DocumentCollection dc1 = db.FTSearch("In Review",0); ❶
            DocumentCollection dc2 = db.FTSearch("Bobby Abreu",0);
            Document doc1 = dc1.getFirstDocument(); ❷
            int matches = 0; ❸
            while (doc1 != null) { ❹
                Document doc2 = dc2.getDocument(doc1); ❺
                if (doc2 != null) { ❻
                    matches += 1; ❼
                    doc2.recycle();
                    doc1 = dc1.getNextDocument(doc1); ❽
                }
                pw.print("There are " + matches + " documents ");
                pw.println("that are in both collections.");
                dc1.recycle();
                dc2.recycle();
                db.recycle();
                ac.recycle();
                session.recycle();
            } catch(Exception e) {
                e.printStackTrace(); } } }
```

Explanation

- ❶ Create a `DocumentCollection` object via the `FTSearch` method of the `Database` class.
- ❷ Retrieve the first `Document` object from the first `DocumentCollection` object.
- ❸ Initialize the counter variable to zero.
- ❹ Loop through all documents in the collection via a `while` loop.
- ❺ Use the `getDocument` method of the `DocumentCollection` class to determine if the `Document` from the first collection is in the second collection.
- ❻ A non-null value is returned if a match is found.
- ❼ Increment our counter of the number of matches found.

- 8 Retrieve the next Document object from the DocumentCollection object.

Output

There are 0 documents that are in both collections.

11.5.5 getNthDocument

The `getNthDocument` method retrieves a Document object from a collection using its location within the collection. A null value is returned if there is no document at the specified position within the collection. This method is often used with the `getCount` method and a `for` loop to traverse all elements of the collection.

```
Document = DocumentCollection.getNthDocument(int);
```

Example 11.2 (Domino Agent)

```
import lotus.domino.*;
import java.util.*;
import java.io.PrintWriter;
public class Example_11_2 extends AgentBase {
    public void NotesMain() {
        try {
            Session session = getSession();
            PrintWriter pw = getAgentOutput();
            AgentContext ac = session.getAgentContext();
            Database db = ac.getCurrentDatabase();
            DocumentCollection dc = db.getAllDocuments(); ❶
            for (int i = 0; i < dc.getCount(); i++) { ❷
                Document doc = dc.getNthDocument(i); ❸
                pw.println("Document NoteID: " + doc.getNoteID()); ❹
                doc.recycle();
            }
            dc.recycle();
            db.recycle();
            ac.recycle();
            session.recycle();
        } catch(Exception e) {
            e.printStackTrace(); } } }
```

Explanation

- ❶ Create a DocumentCollection object containing all documents from the database.
- ❷ Loop through the DocumentCollection object using a `for` loop and the `getCount` method.
- ❸ Retrieve a Document object from the collection via the `getNthDocument` method and the loop index.
- ❹ Display the NoteID for the retrieved Document object.

11.5.6 getPrevDocument

The `getPrevDocument` method retrieves the previous `Document` object from a collection using the `Document` object passed into it as the reference point. A null value is returned if there are no more documents in the collection.

```
Document = DocumentCollection.getPrevDocument(Document);
```

11.6 REFINING A COLLECTION

Implementing its `FTSearch` method can further refine a `DocumentCollection` object. This method searches a collection for the specified query. After calling the method, the `DocumentCollection` object will contain only those documents returned as a result of the full-text search.

```
DocumentCollection.FTSearch("Query", optional maximum number of matches to return);
```

Example 11.3 demonstrates the syntax and use of the `FTSearch` method.

Example 11.3 (Domino Agent)

```
import lotus.domino.*;
import java.util.*;
import java.io.PrintWriter;
public class Example_11_3 extends AgentBase {
    public void NotesMain() {
        try {
            Session session = getSession();
            PrintWriter pw = getAgentOutput();
            AgentContext ac = session.getAgentContext();
            Database db = ac.getCurrentDatabase();
            DocumentCollection dc = db.getAllDocuments(); ❶
            pw.print("There are " + dc.getCount()); ❷
            pw.println(" documents in the database.");
            dc.FTSearch("Julius Erving",0); ❸
            pw.print("There were " + dc.getCount()); ❹
            pw.println(" documents returned from the search.");
            dc.recycle();
            db.recycle();
            ac.recycle();
            session.recycle();
        } catch(Exception e) {
            e.printStackTrace();} } }
```

Explanation

- ❶ Assemble a `DocumentCollection` object containing all documents in the database.
- ❷ Display the number of documents in the database via the `getCount` method.

- ③ Refine the `DocumentCollection` object via the `FTSearch` method. This will remove all documents that do not match the search query from the `DocumentCollection` object.
- ④ Display the number of documents left in the `DocumentCollection` object after the search.

Output

There are 20 documents in the database.
There were 0 documents returned from the search.

11.7 ADDING/REMOVING A DOCUMENT TO/FROM A COLLECTION

The `DocumentCollection` object provides two methods for adding and/or removing `Document` objects from it. The `FTSearch` method allows a collection to be pared down by a search, but you may want to remove only one `Document` or add one as well. The `addDocument` method allows a `Document` object to be added to a `DocumentCollection` object, and the `deleteDocument` method deletes a `Document` object from a `DocumentCollection` object.

11.7.1 addDocument

The `addDocument` method accepts one required parameter and one optional parameter. The first specifies the `Document` object to be added. The second parameter applies to IIOP applications; it is a boolean value that signals whether or not the system should perform an immediate check for duplicates before performing the add procedure.

NOTE An exception is thrown if a duplicate is found in the `DocumentCollection` object.

```
DocumentCollection.addDocument(Document);
```

11.7.2 deleteDocument

The `deleteDocument` method accepts one and only one parameter, the `Document` object to be removed. An exception is thrown if the `Document` cannot be found in the collection or if it has already been deleted. Use the `getDocument` method to avoid an exception.

```
DocumentCollection.deleteDocument(Document);
```

Example 11.4 illustrates the use of both the `deleteDocument` and `addDocument` methods. The code is not of practical use, but it gives a good demonstration of the use and syntax of the methods.

Example 11.4 (Domino Agent)

```
import lotus.domino.*;
import java.util.*;
public class Example_11_4 extends AgentBase {
    public void NotesMain() {
        try {
            Session session = getSession();
            AgentContext ac = session.getAgentContext();
            Database db = ac.getCurrentDatabase();
            View vw = db.getView("Examples \\ by Chapter");
            Document doc = vw.getDocumentByKey("Test", false); ❶
            DocumentCollection dc = db.getAllDocuments(); ❷
            if (doc != null) { ❸
                Document docFound = dc.getDocument(doc); ❹
                if (docFound != null) { ❺
                    dc.deleteDocument(doc); }
                else {
                    dc.addDocument(doc); } } ❻
            dc.recycle();
            doc.recycle();
            vw.recycle();
            db.recycle();
            ac.recycle();
            session.recycle();
        } catch(Exception e) {
            e.printStackTrace(); } } }
```

Explanation

- ❶ Retrieve a Document object from a View.
- ❷ Create a DocumentCollection object containing all documents in the Database object.
- ❸ We will proceed only if the Document from the View exists; a null Document object passed to the getDocument method will throw an exception. We want to avoid this.
- ❹ Search for the Document object from the View in the DocumentCollection object.
- ❺ Delete the Document object from the DocumentCollection object, if it is found.
- ❻ Add the Document object to the DocumentCollection object, if it is not found.

11.8 WORKING WITH FOLDERS

I stated earlier that the Domino Java classes treat a View and Folder the same when you work with them in code. This is true, but a number of unique methods exist throughout the classes for use only with a folder. Documents are displayed in a view via a selection formula, whereas a folder has `Document` objects placed into it either through code or via user interaction. There is no selection formula for a folder.

The `DocumentCollection` class provides two methods for working with folders. The `putAllInFolder` method places all `Document` objects in a `DocumentCollection` object into a specific folder. The `removeAllFromFolder` removes all `Document` objects in a `DocumentCollection` object from a specific folder. These methods do not delete or add documents to a database; they just place or remove them from a folder. Folders contain pointers to `Document` objects, not the objects themselves.

The methods' syntax is as follows:

```
DocumentCollection.putAllInFolder("folder name", boolean);
DocumentCollection.removeAllFromFolder("folder name");
```

NOTE The `putAllInFolder` method accepts an optional second parameter that specifies whether the specified folder should be created if it does not exist. The `removeAllFromFolder` method does nothing if the folder does not exist or does not contain the documents.

Example 11.5 (Domino Agent)

```
import lotus.domino.*;
import java.util.*;
public class Example_11_5 extends AgentBase {
    public void NotesMain() {
        try {
            Session session = getSession();
            AgentContext ac = session.getAgentContext();
            Database db = ac.getCurrentDatabase();
            DocumentCollection dc = db.getAllDocuments(); ❶
            dc.FTSearch("Chapter 11"); ❷
            dc.putAllInFolder("Chapter 11"); ❸
            dc.removeAllFromFolder("Temp"); ❹
            dc.recycle();
            db.recycle();
            ac.recycle();
            session.recycle();
        } catch (Exception e) {
            e.printStackTrace(); } } }
```

Explanation

- ❶ Create a `DocumentCollection` object containing all documents in the database.
- ❷ Refine the `DocumentCollection` object.
- ❸ Put all of the `Document` objects in the chapter 11 folder.
- ❹ Remove all of the `Document` objects from the Temp folder.

11.9 UPDATING DOCUMENTS IN A COLLECTION

Many agents are triggered by specific events, such as new documents being created, edited, or mailed into the database. For this reason, a `Document` object must be marked as processed when an agent “touches” it. This ensures that the agent will not process the `Document` again and again in an endless loop. The `DocumentCollection` class provides the `updateAll` method to handle this task:

```
DocumentCollection.updateAll();
```

The `updateAll` method marks all `Document` objects in a collection as processed. The `updateProcessedDoc` method in the `AgentContext` class marks a specific `Document` as processed. The next two snippets of code perform the same tasks:

```
Session session = getSession();
AgentContext ac = session.getAgentContext();
DocumentCollection dc = ac.getUnprocessedDocuments();
dc.updateAll();
```

or

```
Session session = getSession();
AgentContext ac = session.getAgentContext();
DocumentCollection dc = ac.getUnprocessedDocuments();
for (int x = 0; x < dc.getCount(); x++) {
    ac.updateProcessedDoc(dc.getNthDocument(x));
}
```

11.10 STAMPALL

The `stampAll` method of the `DocumentCollection` class is a very nice feature for streamlining the updating of a particular field on a set of `Document` objects. It places a value in a specific field on all documents in a collection. The great part is that the documents do not have to be saved for the changes to take effect; changes are handled by the system. Also, the specified field is created if it does not already exist.

```
DocumentCollection.stampAll("field/item name", value);
```

The next two snippets of code perform the same task. Notice how much shorter/cleaner the `stampAll` method is.

```
Session s = getSession();
AgentContext ac = s.getAgentContext();
Database db = ac.getCurrentDatabase();
DocumentCollection dc = db.getAllDocuments();
dc.StampAll("Status", "Done");
```

or

```
Session s = getSession();
AgentContext ac = s.getAgentContext();
Database db = ac.getCurrentDatabase();
DocumentCollection dc = db.getAllDocuments();
Document doc = dc.getFirstDocument();
while (doc != null) {
doc.replaceItemValue("Status", "Done");
doc.save();
doc = dc.getNextDocument(doc); }
```

11.11 WORKING WITH PROFILED DOCUMENTS

The Database class provides a method for working with a collection of profile documents. The method, called `getProfileDocCollection`, returns a `DocumentCollection` object. It accepts one parameter, the name of the profile. An empty `DocumentCollection` object is returned if there are no matching profiles.

Example 11.6 (Domino Agent)

```
import lotus.domino.*;
import java.util.*;
public class Example_11_6 extends AgentBase {
    public void NotesMain() {
        try {
            Session session = getSession();
            AgentContext ac = session.getAgentContext();
            Database db = ac.getCurrentDatabase();
            DocumentCollection dc = db.getProfileDocCollection("User Profile"); ❶
            if (dc.getCount() > 0) { ❷
                dc.stampAll("ExpiresDate", "1/1/2001"); } ❸
            dc.recycle();
            db.recycle();
            ac.recycle();
            session.recycle();
        } catch(Exception e) {
            e.printStackTrace(); } } }
```

Explanation

- ❶ Create a `DocumentCollection` object containing all “User Profile” profile documents from the Database.
- ❷ Proceed only if the `DocumentCollection` object is not empty.
- ❸ Populate/create the field called `ExpiresDate` on each `Document` object in the collection.

11.12 SORTING A COLLECTION

A `DocumentCollection` object is sorted only when it is returned from a full-text search. There is no built-in method for sorting it, so we must create our own routine.

Example 11.7 will sort Document objects in a collection using a field called LastName. You can change the sort field and utilize it for your own needs. The example takes advantage of the Vector, DocumentCollection, and Document objects. Let's take a look at the code.

Example 11.7 (Domino Agent)

```
import lotus.domino.*;
import java.util.*;
import java.io.PrintWriter;
public class Example_11_7 extends AgentBase {
    public void NotesMain() {
        try {
            Session session = getSession();
            PrintWriter pw = getAgentOutput();
            AgentContext ac = session.getAgentContext();
            Database db = ac.getCurrentDatabase();
            DocumentCollection dc = db.getAllDocuments();
            if (dc.getCount() > 0) {
                Document[] adocs = new Document[dc.getCount()]; ❶
                int count = 0; ❷
                Document doc = dc.getFirstDocument(); ❸
                pw.println("-----");
                pw.println("                Before Sorting                ");
                pw.println("-----");
                while (doc != null) {
                    adocs[count] = doc; ❹
                    count++; ❺
                    pw.print("Last Name: ");
                    pw.println(doc.getItemValueString("LastName"));
                    doc = dc.getNextDocument(doc); ❻
                }
                // Start Bubble Sort Routine ❼
                for (int y = 0; y < adocs.length; y++) {
                    for (int x = 0; x < (adocs.length - y - 1); x++) {
                        Document temp1 = (Document)adocs[x];
                        Document temp2 = (Document)adocs[x+1];
                        String temp3 = temp1.getItemValueString("LastName");
                        String temp4 = temp2.getItemValueString("LastName");
                        if (temp4.toString().compareTo(temp3.toString()) < 0) {
                            adocs[x] = temp2;
                            adocs[x + 1] = temp1; } } }
                // End Bubble Sort Routine
                pw.println("-----");
                pw.println("                After Sorting                ");
                pw.println("-----");
                for (int i = 0; i < adocs.length; i++) {
                    pw.print("Last Name: "); ❽
                    pw.println(adocs[i].getItemValueString("LastName")); } }
                dc.recycle();
                db.recycle();
                ac.recycle();
                session.recycle();
            } catch (Exception e) {
                e.printStackTrace(); } } }
```

Explanation

- ❶ Create a new array of Document objects. The size of the array is set to the size of the DocumentCollection object.
- ❷ Initialize our count variable to zero.
- ❸ Retrieve the first Document object from the DocumentCollection object.
- ❹ Add the Document object to the array using the count variable as the index.
- ❺ Increment the count variable.
- ❻ Retrieve the next Document object.
- ❼ The bubble routine code is placed between the two comment lines. Java comment lines can start with double forward slashes. The bubble sort is not the most efficient sort routine, so it could easily be replaced by other routines, such as the quicksort. The bubble sort compares adjacent values in the array with the smaller values “bubbling” to the top of the array.
- ❽ Use a for loop to print the sorted contents of the array.

Output

Before Sorting

```
Last Name: Jones
Last Name: Patton
Last Name: Hamilton
Last Name: Zeus
Last Name: Ackerman
Last Name: Williams
Last Name: Smith
Last Name: Hall
Last Name: Spotts
Last Name: Erving
```

After Sorting

```
Last Name: Ackerman
Last Name: Erving
Last Name: Hall
Last Name: Hamilton
Last Name: Jones
Last Name: Patton
Last Name: Smith
Last Name: Spotts
Last Name: Williams
Last Name: Zeus
```

11.13 NEWSLETTER CLASS

Another topic that deserves a bit of attention in a discussion of the `DocumentCollection` class is the `Newsletter` class. The `Newsletter` class allows a group of related documents to be sent via email to a user's mailbox. The `Newsletter` class takes a collection and mails a document with details about all documents in the collection. The group of documents is in the form of a `DocumentCollection` object. It is created using the `createNewsletter` method of the `Session` class and returns the newly created `Newsletter` object.

```
Newsletter = Session.createNewsletter(DocumentCollection);
```

Chapter 19 covers the `Newsletter` class in greater detail.

11.14 CHAPTER REVIEW

The `DocumentCollection` class offers a quick way to work with a bunch of documents from a database. It can be formed through a search from a view, unprocessed documents for an agent, or just plain all documents from a database. Once you have a `DocumentCollection` object created, you can get its size, remove documents from it, add documents to it, and traverse it. You can go to a specific document within it, or move to the previous, next, last, or first document. You can use the position number to navigate through documents as well. Finally, you can move all the contents to a folder or remove them from a folder. Sorting the contents of a `DocumentCollection` is not easy; there is no built-in function for it. The results of a full-text search are returned in sorted order (by score). You must write your own routine if you want results returned unsorted.

- You can create `DocumentCollection` objects from the `Database`, `View`, and `AgentContext` objects.
- The `getNextDocument`, `getLastDocument`, `getFirstDocument`, `getPrevDocument`, `getDocument`, and `getNthDocument` methods can access and manipulate separate entries.
- The `deleteDocument` method removes a `Document` from the collection, while the `addDocument` method adds one.
- All contents of a `DocumentCollection` object can be removed from a folder (`removeAllFromFolder`) or added to a folder (`PutAllInFolder`).
- The `stampAll` method creates and/or populates a field on all elements in the collection.
- The `updateAll` method stamps the contents of a collection as processed, so an agent won't hit them again mistakenly.
- You can perform a full-text search on a collection to refine the contents.
- Sorting a collection can be cumbersome, but algorithms like the bubble sort and quicksort offer some help.