Writing Programs for the Harland Property Store

The Harland Group
harland-support@parc.xerox.com

October 5, 2000

Release: har23

Computer Science Laboratory
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto
CA 94304

Contents

How to Use this Document

This document accompanies an alpha release of the research system Harland.

The body of the document is a guide to writing programs that use Harland. It documents features that are supported and (at least intended to be) functional.

We are very interested in receiving feedback from Harland users. We are happy to receive bug reports and feature suggestions as well as reports of how you are using Harland and which features are particularly valuable. Please send comments by email to harland-support@parc.xerox.com.

Introduction: Bantam/Harland Concepts

The purpose of this document is to help you get started writing programs that use the Harland library. It is intended to be tutorial rather than complete. The on-line JavaDoc should be used as a reference for the full API. Note that the API itself goes by the name “Bantam” and so uses the Java package com.xerox.bantam; Harland is the name of both the primary implementation and the research project.

What is Harland?

Harland is a persistent document storage library for Java programs. It provides facilities to store, search, retrieve and manage a variety of documents. Harland documents can be used to store and manage data for all sorts of application objects. Harland is *middleware*, that is, it fits between existing database/filing components and applications, adding functionality.

The organizing principle around which Harland is arranged is the use of *document properties* with *schemas*. Harland allows programmers to organize persistent document objects not just in terms of containment hierarchies as in most document management systems (file systems, email browsers, etc.), but also according to the *properties* of the documents. A document can have any number of properties, and Harland provides both fast searching over the property space and collections for organization based on containment. Schemas allow programmers to impose some constraints on properties that Harland will enforce (for program stability) and exploit (to enhance efficiency).

Properties are simple name/value pairs. Names are merely text strings and no interpretation is applied to them, although programmers are encouraged to use conventions to reduce the possibility of accidental name collisions between different applications or parts of applications. Every document has a single namespace of properties. In general, there may be more than one value associated with each property name. The collection of values of a single property is a *homogeneous bag*. It is a bag because it preserves multiple values of the same Java type keeping duplicates distinct, but not preserving any ordering among those values. It is homogeneous because all elements must be of the same type. Harland is capable of storing values of any Java serializable object type, but programmers are ***strongly*** encouraged to restrict themselves to standard Java data types such as Integer and Date, for efficiency and smooth evolution free from SerialVersionUID problems[1].

Schemas are named groups of property constraints. Each constraint specifies the Java type that values of the property may have, along with the number of values that the property may have. A constraint also specifies whether the property is optional for this schema. Schemas may be selectively *enforced* on a document by document basis. When a schema is enforced on a particular document, Harland will not permit any change to the properties of the document that would cause it to violate the schema constraints, such as clearing

1.See the release notes for a parameter you can set during development to have Harland catch any unexpected use of application serializable values.

(removing) a required property, setting a property value of the wrong type, or adding a second value to a property that is restricted to a single value. Harland keeps a persistent record of all schemas in use.

A key feature of Harland is its flexibility. Property values can be added to and removed from documents at any time as long as schema constraints are not violated. Schemas may be enforced or unenforced on documents at any time, and new schemas may be introduced to the system and used at any time even if they share properties with existing schemas as long as constraint definitions do not conflict. We plan to permit schema evolution at any time as long as no conflicts are introduced, though this functionality may not be complete in the release you are using. All of these changes may be made by applications while running, without the need to shut down and restart or to perform external manipulations on the database.

For the rest of this document, we will be concerned with how to develop programs that use Harland as their persistent associative storage library.

To begin our exploration of Bantam/Harland programming, we'll explore some of the core object classes in the Bantam API and several important issues for application structure. We've tried to keep to fundamental classes and concepts here, so discussion of some very important technical details is deferred to a later chapter.

The Architecture of Harland Programs

Documents, Collections and Repositories

The primary entities manipulated in Bantam/Harland are *Documents*. Documents, in our terminology, are objects with properties, which may or may not have content, and may or may not have members. Documents with members are called *collections* (or *CollectionDocuments*); they are similar to traditional folders or file directories. Documents with content are called *ContentDocuments*, and are similar to traditional files. Harland can store content in the same database it uses for property values (the default) or can support content stored elsewhere. External systems storing content are called *repositories*; the local file system is the only example presently implemented. The default repository for content is the *internal repository* sharing the database that Harland uses for property values.

A document with no content and no members is just a *Document*. Although it has few of the facets we would normally associate with a document, it retains a stable and unique identity and can accumulate properties.

Single and Multiple Values

In general, properties can have multiple values and the meaning of operations is defined with respect to the general case of multiple values. The general case interfaces emphasize this by the string `Multi` in their method names, e.g. `getMultiValues()`. Methods that get or set multiple values at once either return or take objects of class `java.util.Collection`. These objects are treated as temporary containers for passing the values back and forth. Harland will not preserve either the class or semantics of a particular collection handed to it; it will merely extract and preserve the individual values according to its own homogeneous bag semantics. Similarly, when Harland returns a collection, no manipulation of that collection by the application has any effect upon the values Harland is persistently storing. Only direct API calls into Harland can change the set of values of a property.

Naturally, the case of properties that have just one value is an important and common special case, so it receives some special support. A schema constraint may limit the number of values of a particular property to one, which allows an application to count on finding only one value on documents that have the schema enforced. For such situations, convenience interfaces are provided that deal with just a single value: `getPropertyValue()` and `setPropertyValue()`. These interfaces will throw an exception if they are

not used in a *single-valued context*, which is the context of a property and document where either the property is constrained to a single value by an enforced schema, or the property has no schema constraint at all and has at most one value at the time of the call. Any application that needs to operate on properties that are unconstrained should use the general, multivalued interfaces.

A property *exists* on a document if it has at least one value. Regardless of the enforced constraints or the method used, removing a property and causing it to have zero values are exactly equivalent.

Collections and Queries

A query is an object that represents a search over the property space. Query objects must be constructed by calling methods of the `QueryFactory` obtained from the Storage object. Harland does not have any external query language to allow you to specify queries as strings, though this is a possible future enhancement. Queries are typically tests for the presence of a named property, or for documents whose properties have particular values or for documents that have a particular schema enforced. A query can be evaluated to return a list of documents that match the query.

A collection is a type of document which, in addition to having properties, can also contain other documents (including other collections). Collections have the semantics of a set: documents occur at most once and no ordering is preserved. Retrieval of members is through a list which imposes an arbitrary ordering. Although you may see versions of the Bantam API that allow for a query to define the membership of a collection, as of this writing Harland does not support queries in collections.

Schemas

A schema is a named definition of a group of properties that will be used together. Typically, a schema describes the properties that hold the data for a particular *role* that a document may play within an application. For example, we might decide that a “To-do list item” is characterized by a name, a priority, and a description, and we can write a schema that defines a property for each piece of data. When we wish to have a document object play the role of such an item within our application, we can ask Harland to associate the schema. Harland allows any number of schemas to be associated with a document and allows changes in the set of associated schemas at any time, so a document can play multiple roles and change its roles on the fly.

Schemas define a set of constraints, and the association of a schema with a document is called *enforcement*, because Harland will enforce the constraints. Each individual property definition, called a *FieldDescriptor*, indicates whether the property is required or optional on documents with the schema enforced. In addition, a FieldDescriptor specifies the Java object type that values of the property must have and specifies the number of values that is permissible. Harland will not permit a schema to be enforced on a document if any required property is missing or if the value(s) of any schema property violate type or number constraints. Once a schema is enforced, Harland will not permit any change to the document’s properties that would cause it to violate the schema. The requirement to have values of certain properties applies only to the specific documents that have the schema enforced. Harland enforces type constraints more broadly however. If you have enforced on any document a schema that specifies that “`name.size`” must have an `Integer` value then Harland will not allow a `String` value for “`name.size`” on any document, regardless of whether or not it has the schema enforced.

The fact that a schema is enforced on a document is itself a kind of special property of the document. You can find out if a particular schema is enforced by calling `isEnforced()` on the document object, and you can query for documents that have a particular schema enforced.

If a document satisfies the constraints of a schema, then it is said to *conform* to the schema whether or not the schema is enforced. You can ask a document about conformance and query for conforming documents

just as you can ask and query about enforcement. The concepts of enforcement and conformance are distinguished in the API, and you should be careful to choose the right one.

Schemas are typically defined in a stylized Java class definition that inherits from `com.xerox.bantam.Schema`. The unique schema name in this case is exactly the name of the Java class that defines it. Information from the schema definition that is required by Harland is obtained by introspecting the class. This arrangement for defining schemas has the advantage of automatically creating Java objects that function as constants for referencing properties within an application program. Schemas may also be defined dynamically. There is no external language that will allow you to define a schema with a String.

Databases, User Spaces, and Instances

Harland stores all property values, schema records, and internal bookkeeping information in a relational database. By default, it also stores document content in the database. To avoid confusion, we use the term *database* exclusively to refer to the relational database that provides the persistent store for Harland.

In almost every deployment of Harland, however, there is a need to share one database among multiple users or applications that must not have their documents intermingled. For example, Bob and Sue may need to be able to work independently yet share a single installation of Oracle. Harland uses the concept of a *user* to partition data stored within the same database. A *user space* is one complete set of documents and related metadata that is associated with one user name and independent of the spaces for every other user name.

An *instance* is one Harland Storage object providing access to a particular user space. If we start two or more application processes using Harland to access the same user space, we say that there are *multiple identical instances* of Harland running. This is a situation that must be carefully avoided, because Harland is not yet capable of coordinating the actions of multiple identical instances on the database so that there are no conflicts and no cases of data corruption. Our plan is to improve Harland so that multiple instances can function properly together, but that is future work.

The limitation on multiple instances places restrictions on the form of application parallelism that can be made to work with Harland. If you must have separate Java Virtual Machines (JVMs) that access the same user space of documents, you must create a server component that runs the sole Harland instance for that space and is accessed by other application processes through some mechanism of your own choosing such as HTTP, Java RMI, or CORBA. On the other hand, Harland supports access to the same instance by multiple threads within a single JVM, as described later.

Harland does not support access to more than one user space from a single JVM. This limitation is likely to be removed in the future.

A Harland user space is defined by three things: the database in which the data is stored (identified by server name and instance name/port number), the database login used to access the data (username and password), and the Harland user name. It is the Harland user name that allows Harland to distinguish different user spaces stored in the same database with the same login.

So when a Harland program runs, there are three different user names involved:

- The database account that is used by Harland to establish a connection to the database (login). This account must be created and allocated space by the database administrator. The database login information is supplied to Harland through Java system properties as documented in the release notes.
- The host operating system account under which the Harland process executes. The host account has no significance for Harland if document content is stored in the database. If content is stored in the filesystem, the operating system account becomes relevant as the owner of the content files.
- The Harland user name that identifies a user space of documents. The Harland user name may be explicitly specified through a Java system property. Otherwise, the name of the host operating system account is

used as the Harland user name, avoiding accidental collisions. Harland does not accept “root” as its user name, so if you are running a Harland application under a root account on Unix or Linux you must explicitly specify the Harland user name.

Note that Harland’s isolation of user spaces prevents unintended interference but does not provide any access control. Security is explicitly outside the scope of Harland itself. Access control for user space data can be ensured by using different database user accounts for different Harland user spaces and setting database access controls appropriately. Access control within a user space must be provided by the application according to its own concepts of security principals and its own policies. Harland’s position on security issues is substantially similar to that of a JDBC driver that provides a Java program with direct SQL-level access to a relational database. In many development environments, access control between developers and different test installations is not an issue and it is convenient to share both the database and a single database account. Both Sue and Bob may test their applications using the same database account and password but rely on different Harland user names to guarantee that there are no interactions between them.

Writing A Harland Program

Having seen the basic outline of the Bantam programming interface, let's turn to specifics and see how to construct a simple Harland program.

Most of the time, your Harland program is going to be operating on documents. In Harland, a document is an object that potentially corresponds to some content, and which can hold properties. The main things you can do with documents are to read and write their contents, set and get property values, and move them in and out of collections. Of course, before you can do any of these things, you have to get hold of the documents themselves. You'll normally do this either by creating them (making new document objects), or searching for them (finding documents according to their properties).

However, in order even to do this, you have to initialize the Harland library context by creating yourself a *Storage object*. The Storage object is an object that corresponds to a single instance of Harland. The Storage object is an object that implements the interface `com.xerox.bantam.Storage` and you obtain one by calling a static method on `com.xerox.bantam.util.StorageFactory`.

If you want to create new document objects, you can do this by operating on the Storage object, using the `create*Document()` methods. There are two of these methods allowing for creation of documents with or without a schema enforced initially. Similarly, a set of `import*Document()` methods allow you to import documents whose content lives in an external repository such as the filesystem.

The `create*Document()` and `import*Document()` methods will create and return new document objects, instances of a Document class. The other way to get a handle to Document objects is via a query. In Harland, a query is a temporary representation of a search expression in the form of an object which is an instance of class `com.xerox.bantam.Query`. Query objects are obtained from methods of a Query Factory object, which implements the interface `com.xerox.bantam.QueryFactory` and comes from the Storage object. Complex queries can be constructed piece by piece from simpler queries, all using the methods of the Query Factory.

The result of a query search is a `DocumentList`. This is a `List` interface that is specialized for the type of its members, to save you from having to cast return types yourself.

So, you now have your hands on a Document object. What can you do with it? The most obvious set of document operations are those related to properties and those related to content.

You can set and get the values of properties using methods on the Document object. In general cases, you would use the various methods that operate with multiple property values, such as `setMultiValues()` and `getMultiValues()`. If you are using a property that is constrained to have only one value, you may safely use the special convenience methods `setPropertyValue()` and `getPropertyValue()`.

Document content can be accessed using the `getInputStream()` and `getOutputStream()` calls on a ContentDocument object. These return Java I/O streams allowing, respectively, reading and writing of the

document content. You may also get readers and writers for document content, or use the ContentDocument object directly as a `javax.activation.DataSource` within an activation framework.

A Harland Program

Having seen the basic elements involved in writing a Harland program, we will now look at a minimal sample program in more detail. The program is shown below.

```
1.  package com.xerox.bantam.demo;
2.
3.  import com.xerox.bantam.*;
4.  import com.xerox.bantam.util.StorageFactory;
5.
6.  public class SimpleExample {
7.      public static void main(String args[]) {
8.          Storage stobj = null;
9.          Document d1 = null, d2 = null;
10.         try {
11.             stobj = StorageFactory.open();
12.         } catch (StorageException ex) {
13.             System.out.println(“Unable to initialize storage: “ +
ex.getMessage());
14.             System.exit(1);
15.         }
16.
17.         try {
18.             d1 = stobj.createDocument(DocumentType.OBJECT);
19.             d1.setPropertyValue(“test.testproperty”, “xxx”);
20.             d2 = stobj.createDocument(DocumentType.OBJECT);
21.             d2.setPropertyValue(“test.testproperty”, “yyy”);
22.             d2.setPropertyValue(“testproperty2”, “zzz”);
23.         } catch (StorageException ex) {
24.             System.out.println(“Exception reported: “ + ex.getMessage());
25.             System.exit(1);
26.         }
27.
28.         try {
29.             Query q =
stobj.getQueryFactory().propertyEquals(“test.testproperty”, “xxx”);
30.             DocumentList list = stobj.find(q);
31.             System.out.println(“Query returns “ + list.size() + “
document(s).”);
32.            list.clear();
33.         } catch (StorageException ex) {
34.             System.out.println(“Exception during query: “ + ex.getMessage());
35.             System.exit(1);
36.         }
37.         stobj.shutdown();
38.     }
39. }
```

Let’s step through this line by line.

The very first line just declares that this example is in the package `com.xerox.bantam.demo`. These demo programs are part of the Harland distribution so that you can run them yourself. We also supply the source code so that you can experiment with modifications.

The first significant line (3) is an import of everything from the package `com.xerox.bantam`. This provides access to all the interfaces and classes of the Bantam API, which is the interface to Harland. The Bantam package is the only one you will typically need to include. Since this example class also initializes Harland and obtains the Storage object, it will also use the class `com.xerox.bantam.util.StorageFactory` which is imported here for convenience.

As stated earlier, the first thing that a Harland client needs to do is obtain a Storage object. This is done by calling a static method of a factory class called `com.xerox.bantam.util.StorageFactory`. That class is imported for convenience at line 4. The call to `open()` at line 11 initializes Harland. Note that no arguments are required because all configuration parameters are obtained from Java system properties. The open call can fail, most likely because of some database connectivity problem, so the call is wrapped in a `try/catch`.

Once the initialization is complete, the program creates and manipulates a couple of test documents. These are simple documents created with the call `createDocument()` at lines 18 and 20. Calls to create documents are performed on the Storage object `stobj`; they return document handles on which document operations, such as property manipulations, can be performed.

Lines 19, 21 and 22 show some simple document operations. We set one property on the first document, and two on the second. In this case, we use the special convenience methods for setting single-valued properties. In this simple example, we know that the properties we are setting will be single-valued because the documents are newly created, but in general we recommend that the single-value interfaces be used only when there is a schema enforced that restricts a property to one value. Note that we prefixed our property names with "`test`" followed by a period. This is a convention that is often useful for structuring names to avoid accidental collisions, but it means nothing to Harland; the period character is not significant in property names[1].

Document operations can report a variety of error states as exceptions using the `StorageException` which is trapped at line 23. This exception is the only declared exception in the Bantam API, and so is the only exception that most programs will ever need to catch.

Finally, the program runs a query to determine how many documents have the test property set to a specific value. Harland document spaces are persistent, so new documents will be accumulated each time this program is run, causing the count to increase. The query itself is set up at line 29, using a method of the `QueryFactory` obtained from the Storage object. At line 30, this query is passed to the `find()` method on the Storage object, causing a query to be initiated. The result of this method is a `DocumentList`. We can ask for the size of the `DocumentList` (as at line 31) if that is all we need, or we can use the standard `iterator()` method of the Java `List` interface to get an iterator to iterate through the result list. Although you can access a result list as a list, you should be aware that it is implemented as a stream so best performance for processing the elements will be achieved by retrieving them sequentially using an iterator, rather than using a `for` loop with a call to `size(),` which forces early processing of the entire stream . Once the desired document has been retrieved, we call `clear()` on the list. While not essential, this is a good practice, again because Harland `DocumentList` objects are not implemented as simple in-memory lists and various resources can be released early when the list is cleared.

1.In fact, no characters are significant in property names, although some are illegal. See the release notes for details.

Having completed the manipulations, we call the `shutdown()` method of the Storage object at line 37 in preparation for process termination. A shutdown closes Harland cleanly, which involves writing out any modified data that may not have reached the database. It is essential that clients call `shutdown` prior to process termination to avoid data loss or corruption.

At line 39, we reach the end of the `main()` method in our example, and the program will terminate. Harland clients are always multi-threaded programs but Harland threads will terminate as a consequence of a shutdown, so no explicit exit is required to terminate the VM for this program.

A Harland Schema

Before we look at more typical usage involving Schemas, we need to examine a complete Schema in detail.

```
1. /*
2.  * SampleSchema.java
3.  */
4.
5. package com.xerox.bantam.demo;
6.
7. import com.xerox.bantam.*;
8. import java.util.Vector;
9.
10.public class SampleSchema extends Schema {
11.
12.    /**
13.     * Constructor is for internal use only.
14.     */
15.    protected SampleSchema() {
16.        // no-arg superclass constructor does work
17.    }
18.
19.    /**
20.     * Name
21.     */
22.    static public final FieldDescriptor name =
23.    new FieldDescriptor(“sample.Name”, String.class);
24.
25.    /**
26.     * Number
27.     */
28.    static public final FieldDescriptor number =
29.    new FieldDescriptor(“sample.number”, Integer.class);
30.
31.    /**
32.     * Age
33.     */
34.    static public final FieldDescriptor age =
35.      new FieldDescriptor(“sample.age”, Integer.class);
36.
37.    /**
38.     * State
39.     */
40.    static public final String STATE_NEW = “new”;
41.    static public final String STATE_OLD = “old”;
42.    static public final FieldDescriptor state =
```

```
43.         new FieldDescriptor(“sample.state”, new String[]{STATE_NEW,
STATE_OLD}, false);
44.
45.     /**
46.      * Users
47.      */
48.     static public final FieldDescriptor users =
49.         new FieldDescriptor(“sample.users”, String.class, -1, true);
50.     /**
51.      * The singleton schema.
52.      */
53.     static private final Schema schema = new SampleSchema();
54.
55.     /**
56.      * Return the schema for this descriptor. This method must
57.      * have this name, but we can’t enforce that in the interface
58.      * because this is a static.
59.      * @return Schema return the singleton schema for this descriptor
60.      */
61.     static public Schema getSchema() {
62.       return schema;
63.     }
64.
65.     public static void main(String[] argv) {
66.         Schema sch = Schema.getSchema(SampleSchema.class);
67.         System.out.println(“sch is “ + sch);
68.     }
69. }
```

Although this example is longer than the minimal program example, it is more regular so we’ll look at the overall structure and then focus on selected pieces.

The first thing to observe is that this is a complete Java class definition. The standard way to write a definition of a Bantam schema is to create a Java class following the pattern illustrated here. The name of the schema is exactly the fully qualified name of the class: in this example it is `com.xerox.bantam.demo.SampleSchema` which we obtain by combining the package name from line 5 with the class name from line 10. Any class defining a schema must inherit from `com.xerox.bantam.Schema` as we see here at line 10. The superclass provides all of the functionality required by Harland.

The individual properties in the schema (also called fields of the schema) are defined by objects assigned to static variables. In our example, there are five fields defined: `name`, `number`, `age`, `state`, and `users`. Notice that these comprise the bulk of the code here. For one of the definitions, there is also a pair of value constants defined at lines 40 and 41.

In addition to the field definitions, we find a few bits of boilerplate code. There is an empty constructor at the top to emphasize that all construction should be left to the superclass. A single instance of the schema is created and assigned to another static variable at line 53. A static method at line 61 is provided so that other code can obtain an instance of the schema. Finally, this example includes a `main()` so that you can execute the class, a feature that is handy for a demonstration but not required for real schemas.

Now let’s look at the details.

Each property of the schema is specified by a field definition. The definition takes the form of an instance of `com.xerox.bantam.FieldDescriptor` created in an initializer. There are several different con-

structors for `FieldDescriptor` that provide various ways to specify the requirements of a property. The simplest form is illustrated at line 23 and requires only the property name, and the class type of its values. The meaning of this definition is that the schema requires the property "`sample.Name`" to exist and have a single value which is a String (an instance of `java.lang.String`). A more elaborate example is provided at lines 43 and 44 where we have the requirement that the property "`sample.state`" have one of two String values: "`new`" or "`old`". The argument `false` in this case specifies that the property is not optional. Finally consider the example of a definition that is much less constraining, at line 49. This definition says that the property "`sample.users`" may have an unlimited number of values (`-1`) of class String and that it is optional (indicated by `true`). See the Javadoc for `FieldDescriptor` for all the options.

Each `FieldDescriptor` is assigned to a static variable, which effectively defines a Java constant that you may use to identify the property in the rest of your program. The Java type of these variables must always be exactly `static public final FieldDescriptor` as they are automatically collected from the class using reflection. You can pick any names for these variables that you like.

Don't get confused by the fact that there are *two* names for each property in the schema definition. The actual property name is the string supplied as the first argument to the `FieldDescriptor` constructor. It is this property name that Harland will look at. The property name does not need to be the name of the static variable or even contain any reference to the class because it is perfectly legal for two different schemas, each defined by its own class, to share a property in common. The class variable name simply functions as a constant in code that uses the schema: it is ignored by Harland. Though not strictly required, it is a good idea to construct schema property names using some reference to the schema because that reduces the chance of accidental conflicts in property names and makes things easier to follow when properties of a document are printed out. In this example, we used the prefix "`sample.`" on our property names to associate them with the sample schema.

While you need a schema object in order to perform operations involving the schema, there is no good reason why there should ever be more than one instance of a single schema class within a JVM and multiple instances will hurt performance. For these reasons, the schema constructor should have restricted access and there *must* be a static method `getSchema()` defined precisely as shown at line 61. Java does not allow this requirement to be enforced on subclasses of `Schema` using the type system, so you must remember to include the method. If it is omitted, performance will suffer in certain cases. The `getSchema()` method is the way that your application code will get its hands on an instance of the schema. You should arrange to ensure that only one instance is ever created. The example illustrates one way to do this.

For applications written to use specific properties, we recommend that you define schemas as illustrated by this example. You can even use the sample code as a template for defining your own schemas. It is possible to dynamically define schemas as well but it is not particularly convenient to do so and the process is beyond the scope of this introduction.

An Extended Example

Now that we have a schema in hand, we can see how it is used in an application program. At the same time, we'll look at a number of other features.

```
1.  package com.xerox.bantam.demo;
2.
3.  import java.util.Collection;
4.  import java.util.HashMap;
5.  import java.util.Iterator;
6.  import com.xerox.bantam.*;
7.  import com.xerox.bantam.util.StorageFactory;
```

```
8.
9. public class FullExample {
10.    public static void main(String args[]) {
11.        Storage stobj = null;
12.        Document sample = null, found = null;
13.        HashMap props = new HashMap();
14.        try {
15.            stobj = StorageFactory.open();
16.        } catch (StorageException ex) {
17.            System.out.println(“Unable to initialize storage: “ +
ex.getMessage());
18.            System.exit(1);
19.        }
20.
21.        try {
22.            props.put(SampleSchema.name, “First Sample”);
23.            props.put(SampleSchema.number, new Integer(1));
24.            props.put(SampleSchema.state, SampleSchema.STATE_NEW);
25.            props.put(SampleSchema.age, new Integer(5));
26.            sample = stobj.createDocument(DocumentType.OBJECT,
SampleSchema.getSchema(),
27.                props);
28.        } catch (StorageException ex) {
29.            System.out.println(“Exception reported: “ + ex.getMessage());
30.            System.exit(1);
31.        }
32.
33.        try {
34.            Query q =
stobj.getQueryFactory().propertyEquals(SampleSchema.number, new Integer(1));
35.            DocumentList list = stobj.find(q);
36.            System.out.println(“Found “ + list.size() + “ sample.”);
37.            found = list.getDocument(0);
38.           list.clear();
39.        } catch (StorageException ex) {
40.           System.out.println(“Exception during query: “ + ex.getMessage());
41.            System.exit(1);
42.        }
43.
44.        try {
45.            Collection users = found.getMultiValues(SampleSchema.users);
46.            int num =
((Integer)found.getPropertyValue(SampleSchema.number)).intValue();
47.            System.out.println(“Sample “ + num + “ has “ + users.size() + “
users”);
48.
49.            found.addMultiValue(SampleSchema.users, “George”);
50.            found.addMultiValue(SampleSchema.users, “Paul”);
51.            Iterator iter =
found.getMultiValues(SampleSchema.users).iterator();
52.            while (iter.hasNext()) {
53.                System.out.println(“User “ + (String)iter.next());
54.            }
55.            found.setPropertyValue(SampleSchema.state,
SampleSchema.STATE_OLD);
```

```
56.
57.             found.clearProperty(SampleSchema.users);
58.             found.unenforceSchema(SampleSchema.getSchema());
59.             found.clearProperty(SampleSchema.age);
60.             found.delete();
61.         } catch (StorageException ex) {
62.             System.out.println("Exception during manipulation: " +
ex.getMessage());
63.             System.exit(1);
64.         }
65.         stobj.shutdown();
66.     }
67. }
```

This program begins very much like our first example. There are a few additional imports of standard java.util classes (lines 3-5) and some additional variable declarations, but then we're off with a regular StorageFactory.`open()` as before.

Our first operation with the Storage object is creation of a document, but this time we do it a bit differently. We create the document and enforce the schema all at once. Since our schema requires several properties to be set, we must also supply a value for each of them. The way this is done is to create a `java.util.Map` and add name/value pairs to it for each required property. The map is then passed to the `createDocument()` method. At lines 22-25 we provide values for the required properties: name, number, state, and age. Notice that the code identifies the properties by using the `final static FieldDescriptor` defined in the schema class for each one. By using SampleSchema.name we avoid having to duplicate "`sample.Name`" throughout the code and the *compiler* will catch typos. This way of collecting required properties is sufficient to satisfy the requirements of a schema but has one limitation: you cannot supply more than one value for a property in the map. If you do use some form of collection as a value it will be treated as a single value that happens to be a collection. Remember that Harland accepts any serializable object as a value. Through the map you can supply a first value at least for every required property, ensuring that they all exist. Additional values need to be added later. To create the document, at line 26, we use a form of `createDocument()` that takes three arguments: the document type, the schema to be enforced, and the map of properties to be set. As an experiment, you might want to try commenting out one of the property sets, say line 25, and then running the program. You will get an exception from `createDocument()`; no value has been supplied for one of the required properties so the schema cannot be enforced.

Next up, at lines 34-35, we form and execute a simple query. The query is not fundamentally different from the query we used in our first example. In this case, we test for an integer value rather than a string, and we identify the property by `FieldDescriptor` from the schema class. Notice that an `Integer` object must be used in the query rather than a primitive `int`, just as objects must be used when setting property values.

In this case, we retrieve the first document from the result set to use for further manipulations. At line 97 we call `getDocument()` on the list to retrieve element 0, the first element. This method returns a `Document` so there is no need for a cast. Alternatively, we could have used the generic `get()` method of a `List` with a cast. As this program is structured, there should always be exactly one element in the result set for this query, and that element should be the document created at the beginning. You can add an `equals()` test to verify this.

The next part of the program demonstrates some operations involving multiple values of a property. At line 46, we retrieve all of the values of the property we're going to use. Recall from the schema definition that `SampleSchema.users` allows an unlimited number of values. Since the found document should always be the document we just created, the output at line 47 should show 0 values for the property. This is equivalent to saying that the property doesn't exist, which is no surprise because we haven't set it yet! At lines 49

and 50 we add two values. Adding the first at line 49 causes the property to come into existence with a single value “George”. The second add at line 50 simply augments that with an additional value. At lines 51-54 we retrieve all of the values and print them out one at a time. Note that Harland does not guarantee to preserve any ordering among the individual values, so we could see “Paul” followed by “George” although that is unlikely for these test conditions.

Finally a few miscellaneous operations are illustrated. At line 57 the values just displayed for the users property are removed. The `clearProperty()` method removes all existing values of a property, which is equivalent to unsetting the property. The users property is declared to be optional under the schema, so unsetting it is a perfectly legal thing to do.

At line 58 we unenforce the schema. Remember that the schema was enforced as part of the action of creating the document back at line 26. This document has had the schema enforced throughout its lifetime so far. Once the schema is no longer enforced, it is legal to remove properties that the schema required. We demonstrate this with removal of the age property at line 59. As an experiment, try switching the order of lines 58 and 59. The `clearProperty()` method will then throw an exception, rejecting the attempt to remove a property that is required by an enforced schema. Note that unenforcing a schema may not eliminate the value type restriction, because Harland enforces consistency of value types for properties in schemas enforced on any document.

To conclude this example, the found document is deleted at line 60. This cleanup step is the reason why the query at line 35 should always return exactly one document. Unlike the first example, documents do not accumulate as you run this program because it deletes each document that it creates. Of course, if you modify the program so that it terminates early with an exception, the document will not be deleted.

A Content Example

Most Harland programs will need to access document content. Let’s take a look at that now.

```
1.  package com.xerox.bantam.demo;
2.
3.  import java.io.*;
4.  import java.util.Iterator;
5.  import com.xerox.bantam.*;
6.  import com.xerox.bantam.util.StorageFactory;
7.
8.  public class ContentExample {
9.      public static void main(String args[]) {
10.         Storage stobj = null;
11.         ContentDocument doc = null;
12.         try {
13.             stobj = StorageFactory.open();
14.         } catch (StorageException ex) {
15.             System.out.println(“Unable to initialize storage: “ +
ex.getMessage());
16.             System.exit(1);
17.         }
18.
19.         try {
20.            doc=(ContentDocument)stobj.createDocument(DocumentType.CONTENT);
21.             doc.setPropertyValue(SampleSchema.name, “Content Sample”);
22.             doc.setPropertyValue(SampleSchema.number, new Integer(2));
23.             doc.setPropertyValue(SampleSchema.age, new Integer(7));
24.             doc.setPropertyValue(SampleSchema.state, SampleSchema.STATE_NEW);
```

```
25.             doc.enforceSchema(SampleSchema.getSchema());
26.             OutputStream os = doc.getOutputStream();
27.             PrintStream ps = new PrintStream(os);
28.             ps.println(“My kingdom for some content”);
29.             ps.close();
30.         } catch (StorageException ex) {
31.             System.out.println(“Exception reported: “ + ex.getMessage());
32.             System.exit(1);
33.         } catch (IOException io) {
34.             System.out.println(“IO error writing content: “ +
io.getMessage());
35.         }
36.
37.         try {
38.             Query q =
stobj.getQueryFactory().propertyEquals(SampleSchema.number, new Integer(2));
39.             DocumentList list = stobj.find(q);
40.             Iterator iter = list.iterator();
41.             while (iter.hasNext()) {
42.                 doc = (ContentDocument)iter.next();
43.                 System.out.println(“Sample “ +
doc.getPropertyValue(SampleSchema.number) + “ content:”);
44.                 InputStream is = doc.getInputStream();
45.                 InputStreamReader isr = new InputStreamReader(is);
46.                 BufferedReader reader = new BufferedReader(isr);
47.                 String line;
48.                 while ((line = reader.readLine()) != null) {
49.                     System.out.println(line);
50.                 }
51.                reader.close();
52.                 doc.delete();
53.             }
54.            list.clear();
55.         } catch (StorageException ex) {
56.             System.out.println(“Exception during query: “ + ex.getMessage());
57.             System.exit(1);
58.         } catch (IOException io) {
59.             System.out.println(“IO error reading content: “ +
io.getMessage());
60.         }
61.         stobj.shutdown();
62.         System.exit(0);
63.     }
64. }
```

The program starts off in the now familiar manner to initialize Harland and obtain the Storage object. The only early difference to point out is that we’ve imported at line 3 everything from `java.io`. Now that we’re dealing with reading and writing content, we’ll need a few of these classes.

As in the previous examples, this program begins by creating a document to manipulate. The creation itself is at line 20. Notice that this time we request the creation of a content document where previously we have just asked for an Object document. Several property values are set at lines 21-24. At line 25 we enforce a schema. This sequence provides an example of an alternate way of initializing documents and enforcing schemas that does not use the map of required property values.

We begin dealing with the content by writing some into our newly minted content document. Harland supports simple input and output streams to access document content. In many cases, these will need to wrapped by other Java I/O streams as in other applications involving I/O. Here we obtain the low-level output stream from the document at line 26. At line 27 we wrap it in a print stream so that it is easy to emit a line of text as the content. Line 28 is where the content is generated. Finally, at line 29, we close the stream.

The reading side begins with execution of a query at line 39 which should retrieve the document just created. In this program we illustrate a more substantial loop through the result set, printing out the content for each document. The loop uses an iterator for the result list, obtained at line 40. The iterator interface does not have a document-specific method, so we need to cast the return from `next()` at line 42. Just as we did on the output side, we obtain a low-level stream from the document at line 44. For convenient, line-at-a-time reading, we wrap another stream and a reader around the input stream at lines 45 and 46. With all that preparation out of the way, it is a simple matter to iterate through the content one line at a time calling `readLine()` on the reader. As with all I/O streams, it is important to remember the `close()`.

As we finish reading each document, we delete it at line 51, so documents will not accumulate and there will only be one found for each execution of the program.

That's the last of our examples! We have not illustrated everything here, but we hope you now have good overview of programming with Harland. For complete information about all the API calls please consult the online JavaDoc. Please also look over the technical details covered in the next chapter to avoid surprises.

Technical Details

This section covers some important details that are too technical to be covered in the high level introduction earlier. Many of these details will explain otherwise mysterious things and help you avoid a number of potential problems.

Schema Persistence and Evolution

Harland persistently stores schema definitions in use. When an application starts using a schema, there is a possibility of conflict with a definition of a schema previously stored or with property values already enforced. Harland will reject conflicting schemas by throwing an exception. The `Storage.validateSchema()` method is provided to allow an application to test a schema for conflicts before attempting to use it in a real operation. At this point, Harland may reject new versions of schemas even if they create no conflict, but it is our intention to automatically support schema evolution that is conflict-free.

Designing for smooth, simple schema evolution without inconsistency is an important but complex problem. We have done very little work on this problem to date, so Harland is at a very immature stage of development in this area.

Concurrency and Transactions

As explained in the preceding chapter, Harland does not support concurrent access to a space from multiple JVM processes but does support multithreading. Harland has been designed and built so that it is safe for multiple threads in the same JVM to simultaneously call methods on objects from a single Harland instance. This means that concurrent calls will not cause a fault in Harland, will not deadlock threads, and will not result in any state of Harland that could not be obtained by serializing the calls according to some ordering.

Given an arbitrary interleaving of the execution of multiple threads by the scheduler, however, it is always possible to get surprising results when two threads conflict in operations involving the same data. For example, thread A may issue a query whose result includes a document that no longer matches the query by the time it is examined because thread B has modified the properties of the document. Harland effectively serializes operations only at the granularity of single API methods, and even there the effective order may be surprising given arbitrary instruction interleaving by the thread scheduler.

The Bantam API does not yet provide methods for achieving any transactional behavior over multiple operations. We are working to identify the features that are essential for the kinds of applications that fit the Bantam data model. To preserve simplicity for programmers and maximize performance we do not wish to impose a general ACID model that may not be necessary. We are particularly interested in receiving feedback about the requirements of applications for transactional semantics. In the meantime, it is possible for applications to implement some of the ACID properties on top of Harland.

Exceptions and Failure Recovery

Since Harland is a moderately complex library that depends on a stable connection to a database, almost all calls can generate exceptions. For programmer convenience, the only declared exception in the Bantam API is the StorageException. Each StorageException has a tag which indicates a specific problem that is being reported. The tags are designed to identify errors in use of the API for applications that are sophisticated enough to examine them. Internal problems, such as loss of connectivity to the database, will all be reported with the tag value `UNDERLYING_EXCEPTION` because there is generally nothing that can be done by an application to recover from such errors or even to translate them for users.

Harland is not yet robust against temporary failures such as loss of database connectivity, and so an application that experiences such a condition will have to be restarted in order to continue functioning. We hope to improve robustness in these areas substantially in future releases.

Harland ships with a large number of internal assertion checks enabled. An assertion failure indicates a bug in Harland and is reported by throwing an unchecked (runtime) exception, specifically `java.lang.IllegalStateException` or `java.lang.IllegalArgumentException`. Applications should not attempt to catch these exceptions in general, since they report problems that will never occur in a correct implementation. In a future release it may be possible to disable assertion checking at runtime to improve performance.

Caching and Object Management

As an essential tool for achieving reasonable application performance, Harland maintains a transparent cache of property values. During normal operation, there will typically be dirty data in memory and with an appropriate level of logging enabled you will see messages from the writeback thread that flushes dirty data to the database. It is essential that application programs terminate gracefully by invoking the `Storage.shutdown()` method prior to exiting. Harland is not robust against sudden hardware failure. Various `flush` methods exist or are contemplated to allow some level of application assurance that dirty data has been committed. These issues are still under consideration as a part of the question of transactional semantics.

Since Harland maintains a cache, and Java does not offer a generic deep-copy mechanism for arbitrary instance objects, we cannot prevent an application from holding a reference to exactly the object for some property value sitting in the cache waiting to be written to the database. When mutable objects are used as property values, the application author must ensure that the state is not changed after the object is handed to Harland, otherwise it is impossible to reliably predict what value Harland will preserve. Similarly, mutable objects returned by Harland as property values must not be altered. Collections that Harland returns, whether lists of documents that are the result of a query or the members of a collection, or groups of values, may be altered by the application without having any effect on the persistent document state. Harland is carefully designed to support modifications of these collections without side-effects because it makes common algorithms so much easier to write and avoids unnecessary copying of data.

The document objects that Harland provides to an application are effectively handles to the documents and are only meaningful to Harland itself. For this reason, serializing these objects to some stream and similar operations are of very little value. Also, applications should not expect that there will never be more than one Java object for a particular document. Comparisons between document objects should always be performed by calling the `equals()` method. Harland's caching of property values is completely decoupled from the holding of document objects by an application, so it is never necessary for the garbage collector to process those objects in order to recover cache space.

Logging

Harland employs a logging package internally to produce standardized messages reporting various events. A large volume of messages may be emitted during normal operation, particularly if debugging messages are enabled. It is possible to control both the level and destination of logging messages at runtime - see the release notes for details. The internal logging package is not available for use by applications at this time.

Importing Documents

We have not covered a less common operation that you may wish to perform: *importing* documents. Harland, remember, is capable of accessing content stored in an external repository such as the filesystem. Importing is the process of setting up an association between a Harland document and some content in an external repository.

Please note that importing has not been tested and may not function correctly. Also, the filesystem repository is the only one that supports importing and it cannot be used in conjunction with the internal (database) repository at this time.

The way to do this is with `importDocument()`, which is a method on the Storage object. There are two versions of this method, one of which allows a schema to be enforced upon import. We will consider the simpler form here, which takes two String arguments: the first is `repo`, which specifies the repository on which the document content resides, and the second is `ref`, a reference to the content on that repository:

Harland maintains a set of Repository objects corresponding to the different repositories it knows about. The repo argument names one of these repositories. You may find constants for the names of standard repositories defined in the `com.xerox.bantam.Storage` class, although there may not be repositories available for all of these constants and some may not support import.

The second argument is a reference to the location of the content for that repository. So, in the case of the filesystem repository, the second argument should be a filename; in the case of a Web repository[1], it would be a URL.

The result of the `importDocument()` method is a `ContentDocument` object. `ContentDocument` is a sub-interface of `Document`. Accessing the document's content will automatically fetch it from the external repository.

Note that importing is a one-time operation. It creates a new document that contains a reference to the external content. Importing is not like copying; importing deals with the reference to the file rather than the content. This means that the content returned by `getInputStream()` is always up-to-date even when the external content changes; but that if the external file is deleted or moved, Harland will *not* track these changes in the external repository. Using a filesystem repository introduces dependencies on the local platform that may adversely affect portability of the application. We urge you to use this facility with caution.

1. The `WebRepo` constant is defined for possible future expansion. There is not presently any Web repository.

Appendix I: Release Notes for har23

Most of this document is written so as to be neutral with respect to the installation details and other rude mechanics of any given Harland release or installation. This appendix provides some details that pertain to the specifics of the current release.

Documentation

For full documentation on the Bantam API, you should refer to the JavaDoc documentation included in the distribution you received. If you are inside Xerox, you may access the JavaDoc online at

http://parcweb.parc.xerox.com/project/placeless/harland/builds/har23/doc

Prerequisites

Harland requires an installation of the Java 2 Platform. In particular, the har23 release is built on the 1.3.0 release from Sun. We recommend 1.3.0 because of severe bugs in the socket code of earlier releases. In addition, you will need a library that we do not distribute:

- activation.jar, the Java activation framework (javax.activation). This is needed because the Bantam API uses javax.activation.DataSource. It is available from Sun.

Harland also requires an installation of a supported relational database system and libraries for the appropriate JDBC driver. Harland works with both Oracle and PostgreSQL.

Oracle

The Oracle database version must be 8.1.6 (also called 8i release 2) or higher. Enterprise vs. Standard edition shouldn't really matter and Linux vs. Solaris shouldn't really matter either. We test against Oracle 8.1.6 Enterprise on Solaris. In addition to the database itself, you will need the following Java libraries that we do not distribute:

- an appropriate Oracle8 JDBC driver. We recommend the Thin driver and currently test with version 8.1.6.0.1. It is available from Oracle.
- jndi.jar, the JNDI (javax.naming) library. This file is needed because the Oracle8 Thin JDBC driver requires it. It is available from Sun.

PostgreSQL

We test against the PostgreSQL database version 7.0.2. We currently supply a version of the PostgreSQL JDBC driver that has been modified to fix a couple of bugs. This driver is delivered as pgsql.jar. You will need one additional library required by the JDBC driver and not distributed by us:

- jdbc2_0-stdext.jar, the JDBC 2.0 standard extension, available from Sun.

Installing Harland

Harland ships as a single distribution JAR file. Begin by unpacking this in a suitable disk location. That will provide you with the documentation, sample programs, and the JAR files.

Harland code is currently structured as a two JAR files. The first, harland.jar, contains the Bantam and Harland classes themselves. The second, logger.jar contains additional code required for logging. The pgsql.jar file is a version of the PostgreSQL JDBC driver that you should use (see the prerequisites section).

In order to run programs that use Harland, you must include the supplied JAR files and the prerequisite JAR files in the classpath of the program and run with a Java virtual machine.

Configuration Parameters

As this document has described, Harland obtains its configuration parameter values from Java system properties. In order to run an application with Harland, you will need to arrange to supply the Java Virtual Machine (JVM) with appropriate values for the parameters for your installation. This may be done on the command line directly using Java -D arguments or may be done by specifying the values in a Java properties file and referencing that file with a `-Dharland.propfile` argument.

The following table describes all of the configuration parameters.

Table 1: Harland Configuration Parameters

Property Name	Default	Description
Basics		
`harland.propfile`		Properties file to be used as a source of values for other properties. See Java documentation of `java.util.Properties` for information about this file format
`harland.user`	*process user name*	Harland user name, the name of the user space of documents
Database		
`harland.db.name`	`oracle`	Name of database to use. Options: • `oracle` • `postgresql`
`Oracle`		
`oracle.databaseName`	`TEST8K`	Oracle database SID
`oracle.dataSourceName`	`oracle.jdbc.pool.OracleDataSource`	Specific data source subclass. Options: `oracle.jdbc.pool.` • `OracleConnectionCacheImpl` • `OracleConnectionPoolDataSource` • `OracleXADataSource` *For future use – only tested with default*

Table 1: Harland Configuration Parameters

Property Name	Default	Description
`oracle.user`	`presto`	Oracle user name. Oracle names are case-insensitive
`oracle.password`	`prestopass`	Oracle password. Oracle passwords are case-insensitive.
`oracle.portNumber`	`1521`	TCP port number as decimal-encoded string
`oracle.serverName`	`pythia`	Name of the Oracle server machine
`oracle.driverType`	`thin`	Type of JDBC driver. Options: • `thin` – Type 4 (all Java) • `oci8` – Type 2 (native) *Only tested with default*
`PostgreSQL`		
`pgsql.databaseName`	`TEST8K`	PostgreSQL database name
`pgsql.user`	`presto`	PostgreSQL user name
`pgsql.password`	`prestopass`	PostgreSQL password
`pgsql.portNumber`	`5432`	TCP port number as decimal-encoded string
`pgsql.serverName`	`pythia`	Name of the PostgreSQL server machine
Repository		
`harland.usefs`	`false`	Determines whether the filesystem repository is enabled.
`fs.repo`		Directory under which content files in the filesystem repository will be stored.
`fs.repoRoot`	*process home dir*	Directory under which filesystem repository directories will be created by Harland user name if `fs.repo` is not set.
Cache		
`harland.cache.max`	`1000`	Maximum number of live documents in the cache
`harland.cache.delay`	`10000`	Sleep delay for writing thread in msecs. 0 = disable writing (*debugging only*)
`harland.cache.writebatch`	`20`	Maximum number of dirty documents to be written out in a batch
Logging		
`harland.logfile`		File to which log output should be written instead of standard output/error

Table 1: Harland Configuration Parameters

Property Name	Default	Description
`thinkdoc.config.logDebug`	`true`	Determines whether debug messages appear in the log output
`thinkdoc.config.logStorage`	`true`	Determines whether Harland log messages appear at all
`Warnings`		
`harland.serializable`	`allow`	Controls handling of uses of serializable values of application classes. Options: • `allow` - No special handling • `warn` - Log uses • `trace` - Report uses by stack trace • `reject` - Reject uses by exception Note that reference to a schema requiring an application class for property value(s) is considered a 'use'

Running a Harland Program

You may run one of the sample applications to verify that your installation is working properly. We recommend that you create a properties file with appropriate values for all of the configuration parameters. For example, you might create a properties file that looks like this:

```
harland.user = myuser
oracle.databaseName = mydbsid
oracle.user = mydbuser
oracle.password = mydbpass
oracle.serverName = mydbserver
thinkdoc.config.logDebug = false
```

Supposing that you create such a file with the appropriate values for your installation and store it as harland.properties in your home directory, you can run SimpleExample with the following Unix command:

```
% java -Dharland.propfile=$HOME/harland.properties  \
      com.xerox.bantam.demo.SimpleExample
```

On an empty Harland space, the above command should produce output similar to this:

```
Loading parameter values from /tilde/jthornto/harland.properties
Query returns 1 document(s).
```

Resetting Your Harland Space

You can reset your space with the command:

```
% java -Dharland.propfile=$HOME/harland.properties \
      com.xerox.bantam.harland.BootManager reset
```

A reset will remove all data for documents in the target user space, including content created through Harland. In some cases, a simple reset may fail due to some corruption or inconsistency in the database. In that case you should run the same command but substitute "brute" instead of "reset".

Restrictions and Limitations

Property Names

The name of any property is a string containing at least one character and at most 255 characters. All property names prefixed with “harland” (regardless of case) are reserved for internal use. Property names have no significant internal structure, though we suggest hierarchical allocation of names using dots as separators as a convention, e.g. “a.b.c.d”. The following characters are illegal in property names and may result in strange errors and undefined results if used:

```
' " %
```

(single quote, double quote, percent)

Large String Values

Harland will persistently store String property values of arbitrary length, limited only by the database large object capacity. Query predicates, however, will not work as expected for very long String values (longer than about 4K).

Repositories

Harland only has implementations of the internal and filesystem repositories and they may not be used simultaneously. We recommend that you use the internal repository in the database, but if you wish to use the filesystem instead you must switch to it by setting the `harland.usefs` property to `true`. The internal repository does not support any form of import.

Database Connections

Harland does not yet deal gracefully with loss of connectivity to the database. If one of the database connections is disrupted, the Harland application will need to be restarted in order to recover.

Security

As described earlier, Harland itself does not provide any security features. Access control can be achieved by appropriate configuration of different Oracle database user accounts and tablespaces, since Harland will connect to the database using whatever name and password it is given. Security configuration of databases is beyond the scope of this document and we are unable to help with it. If you use the filesystem repository, document content will be stored in a filesystem accessible to the application process. In this case, normal filesystem access controls may be used to restrict access.

Appendix II: Glossary

Collection:
: a form of Document that supports the Collection protocol. It collects together a set of documents. Documents can be added and removed, and the contents retrieved. A collection may some day embody a *query* whose result set is, dynamically, the membership of the collection.

Content:
: the actual bits that a *document* contains (or may contain). Content resides on an external *repository.*

Content Provider:
: an internal Harland component that delivers content between the *repository* and the application. Different content providers enable different repository access protocols to be incorporated into Harland.

Database:
: a database system (such as Oracle) that provides the backing store for *properties* and possibly *content*

Document:
: the basic object of the Harland system. A Document may or may not have *content*, but it always has *properties*.

Field Descriptor:
: definition of the value constraints on an individual *property* within a *schema*

Instance:
: a *Storage object* accessing a *user space*

Property:
: a name/value pair associated with a document.

Query:
: an object representing a set of search terms over the property space.

Repository:
: an external store of *content*. Examples include the database (*internal repository*), the filesystem, the World Wide Web, etc.

Schema:
: a definition of a group of *properties* and constraints on the values of those properties

Storage object:
: the primary entry point into the Harland library for clients. The Storage object allows users to connect to Harland *user spaces*, search for *documents*, etc.

User Space:
: the collection of *documents*, *schemas*, etc. associated with a single Harland user

Appendix III: Release History

The first Harland release (build har1) was in March 2000. All early releases were very limited and internal to Xerox. The first general release was har18, released under the trial license found in Appendix IV.

Changes for har23

- Introduced support for PostgreSQL
- Added query operators for inequality tests on Comparable values (<,>,<=,>=)
- Fixed thread race condition manifested by intermittent exception during shutdown

Appendix IV: License

Xerox AlphaX 90-Day Trial

Software License Agreement

BY DOWNLOADING OR OTHERWISE INSTALLING THIS SOFTWARE YOU AGREE TO BE BOUND BY THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU DO NOT AGREE WITH THESE TERMS AND CONDITIONS, DO NOT DOWNLOAD OR INSTALL THIS SOFTWARE.

This Agreement is between Xerox Corporation, with offices at 3333 Coyote Hill Drive ("Xerox"), and you, the person or entity downloading and installing the Software referenced in paragraph one below. This Agreement sets forth terms and conditions applicable to your use of this Software. The term "Software" shall include the software, its documentation and other supporting materials packaged and downloaded with the software. Xerox is providing the Software for your trial and evaluation free of charge, and encourages your feedback on the Software.

1. License Grant. Xerox grants you a non-exclusive and non-transferable license to install and use the Harland Software solely for the purpose of evaluating the Software and providing feedback to Xerox. This Agreement does not constitute a license to use any other version or copy of the Software other then the copy or version obtained by accepting this Agreement.

2. Acceptance of Agreement. By downloading the Software, you agree to be bound by the terms and conditions of this Agreement.

3. Term. You may use the Software for ninety (90) days after the date on which you download the Software. Upon termination or expiration of this Agreement, you shall destroy all full or partial copies of the Software and all other materials made available hereunder by Xerox.

4. Ownership and Copyright. You agree with Xerox that the Software and related information is owned by Xerox and that, unless otherwise specifically agreed by Xerox in writing, you:

(a) shall not distribute, transfer, loan or otherwise provide the Software to any third party, and shall not copy, reverse compile, reverse engineer or disassemble the Software, the sole exception being that a copy of the Software may be made for back-up purposes ;

(b) will grant to Xerox an irrevocable license under all intellectual property rights (including copyright) to use, copy distribute, sublicense, display, perform and prepare derivative works based upon any feedback that you provide to Xerox, including materials, fixes, error corrections, enhancements, applications and uses, suggestions and the like; and

(c) shall display, and shall not alter or remove, Xerox' and/or any of its' licensors copyright notices and other proprietary notices.

5. Warranty and Liability. Due to the Software being in a developmental stage, Xerox makes no warranties whatsoever as to the operational performance of the Software. THE SOFTWARE IS BEING PROVIDED "AS-IS". XEROX DISCLAIMS ALL WARRANTIES WITH REGARD TO THE SOFTWARE, INCLUDING WITHOUT LIMITATION ALL WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. YOU ARE SOLELY RESPONSIBLE FOR DETERMINING THE APPROPRIATENESS OF USING THIS SOFTWARE AND ASSUME ALL RISKS ASSOCIATED WITH ITS USE, INCLUDING BUT NOT LIMITED TO THE RISKS OF PROGRAM ERRORS, DAMAGE TO OR LOSS OF DATA, PROGRAMS OR EQUIPMENT, AND UNAVAILABILITY OR INTERRUPTION OF OPERATIONS. XEROX SHALL HAVE NO LIABILITY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL OR TORT DAMAGES ARISING OUT OF USE OR PERFORMANCE OF THE SOFTWARE, EVEN IF XEROX HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Some states or provinces do not allow the exclusion or limitation of implied warranties or limitation of liability for incidental or consequential damages, so the above exclusion or limitation may not apply to you. Because software is inherently complex and may not be completely free of errors, you are advised to verify and back up your work. Additionally, Xerox does not guarantee compatibility between the Software and any future versions of the Software.

6. No Obligation. Xerox is under no obligation to develop the Software or market the Software as a final product. Your use of the Software is at no charge, and your use shall not obligate you to purchase additional software from Xerox.

Updated: 08/18/2000