## *Homework Assignment #11: Java Server, Due Feb. 8, 2001 9:30* *30 points*

The last assignment is to implement your own server for the library system. The assignment is split into two parts. The first part is rather simple and should get you accustomed to implementing CORBA objects in Java. In the second part you will have to use some advanced features of the Portable Object Adaptor, which you will learn in next week's tutorial.

General rules for the server: Use the package structure and class names that are given in this document. All generated classes have to be in packages under the package `generated`, i.e. use `jidl` with the option `--package generated`. Never ever change any source code inside the generated packages. The `handin` script will not copy them and I will generate them again when I test your servers.

### PART 1

a) Change your client or the client from `/proj/i4oods/pub/assignment10` so that it gets the reference to the Librarian object from the Naming Service instead of the file `Librarian.ior`. Your should lookup the name "`<login>/Librarian`", where <login> is your Login name. There is a Name Server running at `corbaloc::faui40u.informatik.uni-erlangen.de:4711/NameService` which already has Naming Contexts for each user that registered for the tutorial. *Never ever resolve or bind/rebind in someone else's Naming Context! There is nothing to prevent you from overwriting someone else's bindings and it results in total confusion if you do that.*

b) Write a class `oods1.library.server.librarian.Main` that contains only a static `main` method to start your server up. The `main` method should create an object of the class `oods1.library.server.librarian.LibrarianServant` (which does not yet exist) and bind/rebind it in the Naming Service under the name "`<login>/Librarian`".

c) Write a class `oods1.library.server.librarian.LibrarianServant` which inherits from `generated.oods1.library.LibrarianPOA` and implements the three methods from `generated.oods1.library.LibrarianOperations`. Since you don't have Books or Cards at the moment, the methods should have the following behavior:

d) `issueCard` should throw a `DeniedCard` exception.

e) `findCardByOwner` should throw a `NoSuchCard` exception.

f) `findBookByTitle` should return an *empty* sequence of Books.

Now you can test your server with the client from item (a).




### PART 2

Your application will consist of three server processes. One that manages Books, one that manages Cards and the one that you have already started to develop in Part 1. The file `LibraryManagement.idl` contains the internal interfaces that the three servers use to communicate with each other. The whole system is started by first of all starting the Librarian server. This creates a Servant for the `Librarian` interface and registers it with the Naming Service – you have already done this in Part1.

g) Extend the Librarian server so that also creates a Servant for the `LibraryManager` interface (Java class `oods1.library.server.librarian.LibraryManagerServant`) using the RootPOA and writes its reference to the file `LibraryManager.ior` in the current directory. For now you should only provide usable implementation for the IDL operations `registerBookManager` and `registerCardManager`.

h) Now write a class `oods1.library.server.book.Main` that starts the Book server. It should first of all read the reference to the Library Manager from the file `LibraryManager.ior`. Then it creates a Servant for the `BookManager` interface (Java class `oods1.library.server.librarian.BookManagerServant`) using the RootPOA. The Book Manager can now be registered with the LibraryManager using the IDL operation `registerBookManager`.

i) A library usually contains huge numbers of books, so it is impossible to instantiate a Servant for each Book object. Therefore you will implement a cache that only contains the recently accessed Book Servants. Because you have to store the data for the books somewhere, I have provided a very simplistic database for books. The class `oods1.library.server.book.BookDatabase` contains the methods to read and search the database. The data for one book is stored in an object of class `oods1.library.server.book.BookRecord`. (There is also a method to write the database which you don't need in the assignment but you can use it to extend the database file I provided for you).

j) To build a cache you have to manage the Active Object Map yourself, so you have to create you own POA. In the constructor of `BookManagerServant` create a POA named "BookPOA" underneath the RootPOA and provide the following policies: `USER_ID`, `USE_SERVANT_MANAGER`, and `NON_RETAIN`. You can use the same POA Manager as the RootPOA, i.e. get the reference from the RootPOA and provide it to the factory method that creates your new POA.

k) Now you also have to set a Servant Manager that the POA uses to find Servants (`NON_RETAIN` means that the POA has no Active Object Map). In this case the Servant Manager must implement the `PortableServer::ServantLocator` interface. The Java class that you write for that should be named `oods1.library.server.book.BookServantLocator`. The Servant Locator manages the Servant cache. This cache is just a simple array with a constant but configurable length (make it 2 for testing). Every time the Servant Locator is asked to find a Servant it also receives its Object ID. This Object ID (an array of `byte`s) has to be converted into a Java `long` – this is also the record number of this book in the database, so there is a

one-to-one relation between Object IDs and record numbers. The only difference is that our Object ID is a `byte` array of length 8 whereas the record number is a `long` (that also uses 8 bytes) – so you have to write two utility methods that convert between Object IDs and record numbers.
The index into the cache is then computed using the formula `index = record_number % cache_size`. Each object in the cache is a `oods1.library.server.book.BookServant` and has a reference to a `BookRecord` that contains the data for this book and reads this data from the record to provide the methods for accessing the IDL attributes of Book. If the record of the BookServant at the computed index in the cache has not the same record number as the object requested by the POA, you can simply throw away the existing BookRecord (it only stores immutable data) and read the record for the requested book from the database.

l) Implement the `findBookByTitle` method in `BookManagerServant` using the `search` method of the database. You receive an array of record numbers as the return data but the method should return CORBA references. To get a reference first of all convert the record number into an ObjectID (see (k)) and then you simply ask the BookPOA to create a reference with this ID (`create_reference_with_id`). There is no need to read records or create servants – this should all be done by the `BookServantLocator` when it receives calls.

m) With the above it is easy to implement the remaining operations `bookForId` and `idForBook` in `BookManagerServant`. When you get a reference, you have to extract the ObjectID from it. This can only be done by the POA that created the reference. It provides a method `reference_to_id` which does exactly that. The opposite mapping can only be achieved using `create_reference_with_id` because `id_to_reference` only works with the `RETAIN` policy.

Now you have most of the Books server and can test it with your client.

Next you should repeat steps (h) to (m) with the Cards server in package `oods1.library.server.card`. Simply substitute each `book` with `card` and each `Book` with `Card`. There is a special implementation of CardDatabase and CardRecord in already in this package. Beware that this time you have to write the records back to the database because the charges can change.

n) In additon to the `findCardByOwner` operation (which is similar to the book operation `findBookByTitle`) you also have to implement `issueCard` in the `CardManagerServant`. To create a new record in the database you simply create a `CardRecord` object, set its record number to the current size of the database (`size()`), and write it into the database.

Now we have a look at the `LibrarianServant` again:

o) All calls to `findBookByTitle`, `findCardByOwner`, and `issueCard` should be forwarded to the BookManager and CardManager objects that have been registered with the `LibraryManager`.

The only remaining methods are now for borrowing and returning books. The `LibraryManagerServant` is the central object to store all this information.

p) There is a simple database in the package `oods1.library.server.librarian` that can store pairs of record numbers for book and cards with an associated date when the book is due. `LibraryManagerServant` should manage this database directly (no Servant for each record). It uses the `searchBook` and `searchCard` methods of `LibraryDatabase` to implement `cardForBookId` and `booksForCardId`.

q) Every time it receives a request to borrow a book (`setBorrowed()`), it should first of all find out if the book was already borrowed and then append a new `LibraryRecord` that stores the id of the Book and the Card as well as the date when the book ist due to be returned. Since this call is only issued by a `CardServant` it can provide its own ID directly but cannot decode the Book reference. This has to be done in the `LibraryManagerServant` by calling `idForBook()` at the registered BookManager.

r) Implement `unsetBorrowed()` in a similar way as in step (q).

s) Now change `BookServant` to call `cardForBookId()` on the LibraryManager to find out if it was borrowed, by whom it is borrowed, and when it is due back. Do not store any of this information locally in your `BookServant`.

t) Finally write an implementation for the `CardServant` methods `borrowBook` and `returnBook` to call methods of the `LibraryManager`. Remember that you should do some checks on the local card data (still valid, open charges) before you call `setBorrowed()` on the `LibraryManager`.

In order to check and manipulate dates the class `java.util.Calendar` is very useful. You can simply set day, month, and year, do date-related operations correctly, and read the new day, month, and year afterwards.

Generally you should keep in mind that anything in a CORBA server process can be executed concurrently, so think about a strategy to synchronize access to data.

That's it. Good luck for your server implementations!