JBase

Final Project Proposal

DECEMBER 3, 2018

CSCI-24000 Fall 2018

Authored by: Bryan McClain

Project Summary

JBase Database Engine

Project Goal

The purpose of this final project is to create a database engine using the Java programming languages (named “JBase”). The database will use a variety of abstract data types—classes and interfaces—to represent different types of fields in the database. Data in the database will be stored using data structures discussed in class (such as doubly linked lists). The database will also support multiple user login, with a permissions system to limit what users can do to the data. For data persistency, the database will have a system to save and load from a binary file. The final project will also include a simple terminal interface for interacting with these database classes for testing purposes.

Database Features

- Variety of fields to abstractly store and represent data (key, item, foreign key)
- Ability to store multiple datatypes (integers, floats, timestamps, etc.)
- Multiple user login with access control
- Save and load database from a binary file
- Terminal interface interacting with the database

Intended Users

The JBase database engine is intended to be used by software developers and database administrators to have a secure and reliable place to store raw data. As such, the database itself will consist of packages of Java classes that can be used to create a larger project. Classes will need to be well-documented using JavaDocs to allow programmers to understand how to use the features of JBase.

Java Technologies Needed

- Java abstract data types (Classes and Interfaces)
- Various data structures (ArrayList, HashMap, TreeMap, HashSet, Stack)
- File I/O

Use Case Analysis

Software Developers

Software developers need a standard set of methods to interact with the database in their code. This has been done using abstract data types for interacting with the database. (See the section on Data Design for how this works).

Database Administrators

For designing and using databases, database administrators can interact with the database using a terminal interface. The interface allows administrators to create a new database or log in to existing database. The interface also allows database administrators to change the database structure by creating, deleting, or modifying existing fields in the database, as well as inserting values into the database.

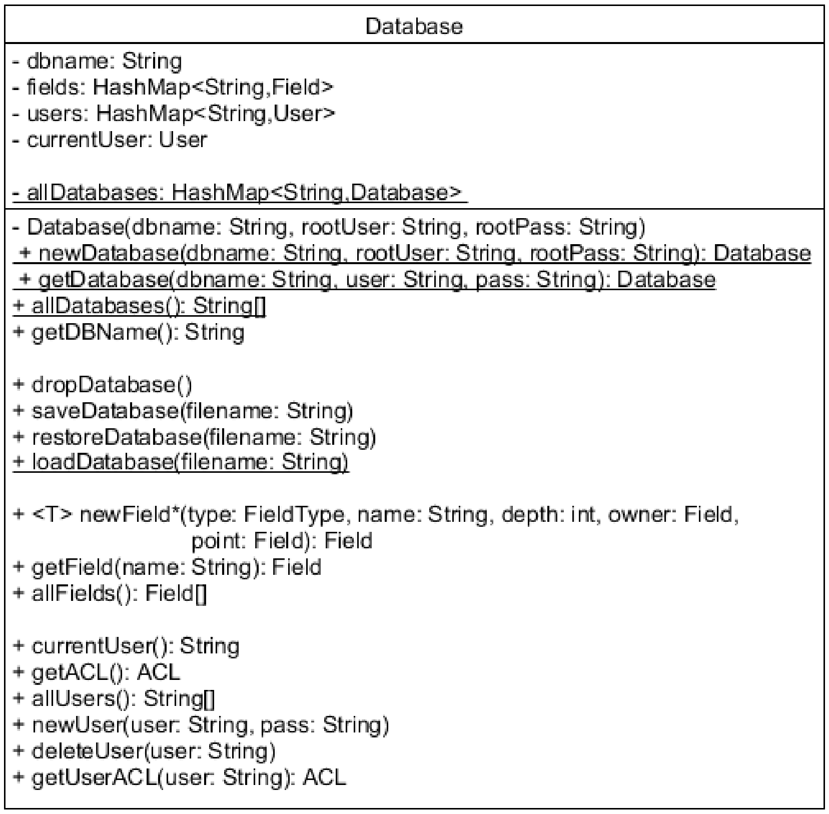
A typical session might look like this

1. Administrator creates a new database or logs in to an existing database. New databases can also be loaded from a binary data file.
2. New fields are created in the database, starting with a key field and then several item or foreign key fields (owned by the key field). This is done using the New Field dialog. Existing fields might be resized to allow more data to be stored (using the Add Rows dialog).
3. Some data is inserted into the database using the built-in terminal command.
4. The administrator might need to create some additional users to access the database, or update the permissions of existing users.
5. The database is saved to a binary file for backup.
6. The administrator logs out with the knowledge of a job well-done.

Data Design

Database Object

The database object represents a "connection" to a JBase database. Each database object stores a list of fields, along with some additional properties about the database (such as the name or a list of Users).



```
Database
- dbname: String
- fields: HashMap<String,Field>
- users: HashMap<String,User>
- currentUser: User

- allDatabases: HashMap<String,Database>
- Database(dbname: String, rootUser: String, rootPass: String)
+ newDatabase(dbname: String, rootUser: String, rootPass: String): Database
+ getDatabase(dbname: String, user: String, pass: String): Database
+ allDatabases(): String[]
+ getDBName(): String

+ dropDatabase()
+ saveDatabase(filename: String)
+ restoreDatabase(filename: String)
+ loadDatabase(filename: String)

+ <T> newField*(type: FieldType, name: String, depth: int, owner: Field,
                        point: Field): Field
+ getField(name: String): Field
+ allFields(): Field[]

+ currentUser(): String
+ getACL(): ACL
+ allUsers(): String[]
+ newUser(user: String, pass: String)
+ deleteUser(user: String)
+ getUserACL(user: String): ACL
```

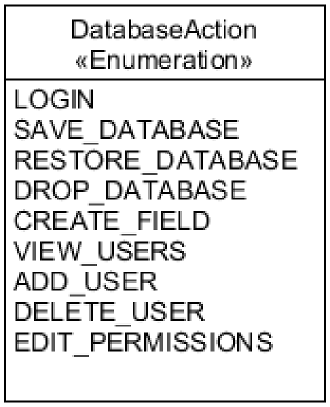
Databases cannot be created by calling the constructor directly, but must be built using the newDatabase() static method. Existing databases can be loaded using the getDatabase() static method. Loaded databases can be deleted using the dropDatabase() method, assuming that the current user has permission to drop the database.

The database object has a series of methods to save and load databases to and from binary files. Calling the saveDatabase() method stores all of the data to a file, and calling restoreDatabase() replaces all of the information in the current database with the data in the file. The static method loadDatabase() adds a database stored in a file to the list of active databases, thus creating a new database.

Fields in the database are created using the newField() method (see the Field object for the list of FieldTypes). Not all parameters are required by all fields, and can be left NULL for fields that don't require them. The FieldType enumeration has metadata that explains what parameters are needed by each field type. Existing fields can be retrieved by calling getField(), or all Fields can be returned as an array by calling allFields().

The database object also has a series of methods to create and delete users, as well as update user permissions in the database. This is to enforce access control limits. If the current user does not have permission to view users, then calls to allUsers(), deleteUser(), and getUserACL() will only work for the users that the current user created.

Database Action



This enumeration defines the list of all actions that a user can execute on a database. The actions are all summarized below:

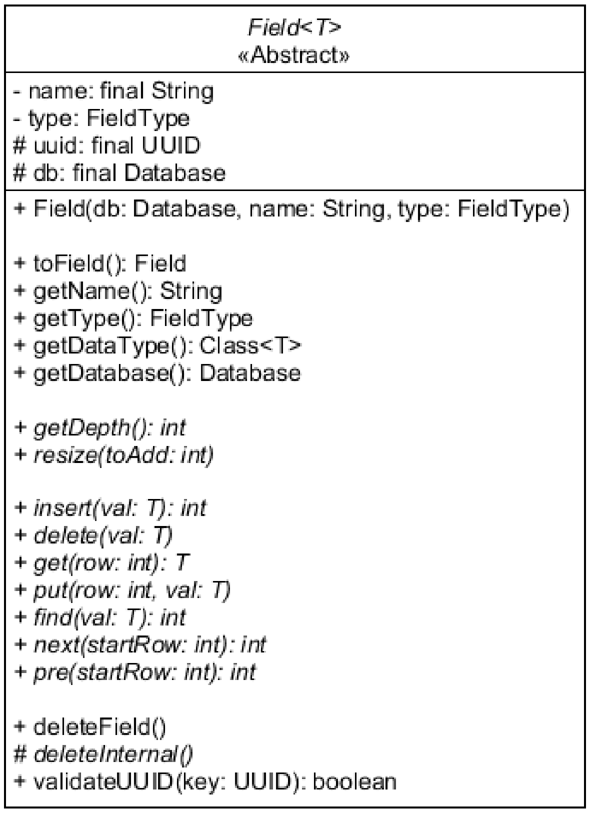
- **Login** – Providing the right username and password returns a valid connection to a JBase database.
- **Save Database** – Writing the database to a binary file
- **Restore Database** – Loading a backup of the database from a binary file, and replacing the current version of the database.
- **Drop Database** – Deleting the database from memory
- **Create Field** – Adding a new field to the database
- **View Users** – Ability to view the list of all usernames for users who can connect to this database.
- **Add User** – Creating a new user who can connect to the database
- **Delete User** – Removing a user from the list of users
- **Edit Permissions** – Update the ACL permissions for a given user

Fields

Field Object

Implements: JBaseField

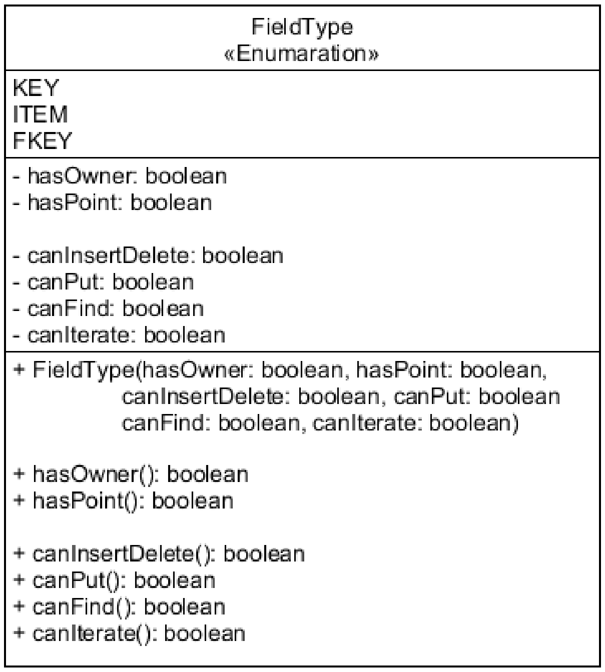
The Field object is an abstract way to represent a "column" in the database. The field object uses Java generics to store potentially any type of complex object in the database. Every field has a variety of actions that can be used to insert, delete, update, or iterate over values in the database. The Field class is extended to create concrete field types (Key, Item, and Foreign Key).



All fields have several universal properties:

- **Type** – The sub type of this field (either key, item, or foreign key)
- **Data Type** – The actual data stored in the field. Because fields use Java generics, the getDataType() method returns a Class<T> object using Java reflection.
- **Depth** – Number of records stored in the field. The depth of an item or foreign key is dependent on the depth of the key
- **Rows** – Every record in a field is assigned a unique row number

FieldType



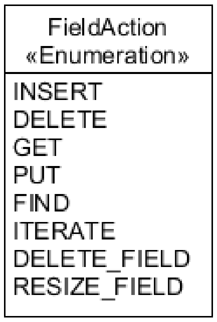
The FieldType enumeration stores all possible types of fields in JBase, as well as some metadata about each type of field:

- **Has Owner** – Does this field have an owner field? (ParentField)
- **Has Point** – Does this field point to another field? (PointableField)
- **Can Insert Delete** – Can you call the insert() and delete() methods on this field?
- **Can Put** – Can you call the put() method on this field?
- **Can Find** – Can you call the find() method on this field?
- **Can Iterate** – Can you call the next() and pre() methods on this field?

The types of fields are summarized here:

- **Key Field** – Each element in the field must be unique, and the elements are sorted for easy lookup. Key fields own item or foreign key fields, and can be pointed to by foreign keys.
- **Item Field** – Owned by a key field. Unlike a key field, the elements in an item can have duplicate values.
- **Foreign Key Field** – Also owned by a key field, but unlike an item, points to another key field.

FieldAction



This enumeration defines all of the actions that can happen on a field. All of the actions are explained below:

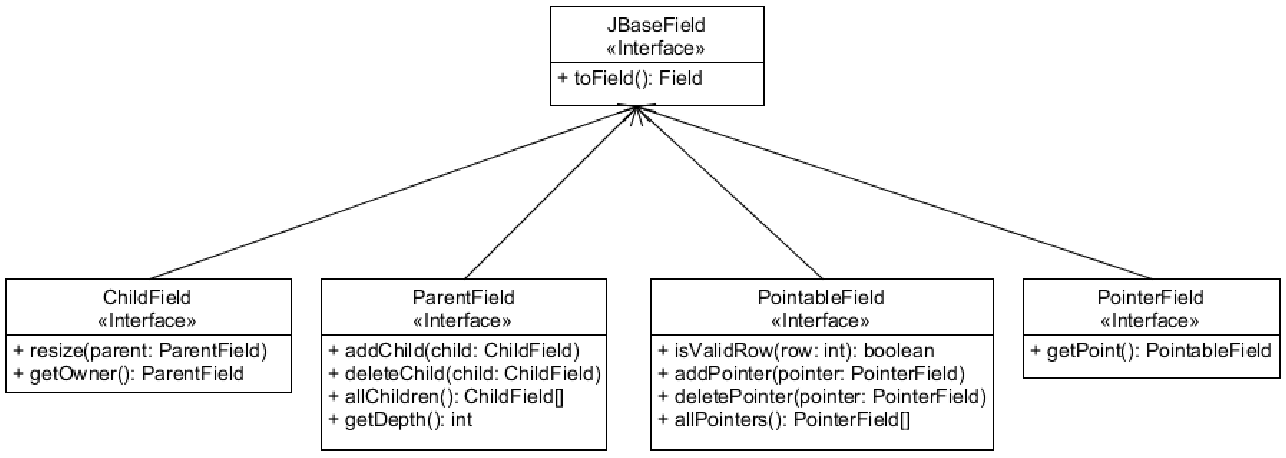
- **Insert** – Add another value to this field. Used with keys to add new unique records.
- **Delete** – Remove a value from this field. Used with keys to delete unique records.

- **Get** – Get the value of a record in the field given the row
- **Put** – Update the value of an existing record at a given row in the field
- **Find** – Search for a unique value in a key field
- **Iterate** – Loop over the values in the field in order, by either calling next or pre on the field
- **Delete Field** – Remove this field from the database. If the field is a key, then it also deletes all items, foreign keys, and any fields that point to it.
- **Resize Field** – Increase the number of records allowed in a key field. Also resizes all of the item fields

Field Interfaces

JBase defines several field interfaces for abstracting the behavior of the concrete field types. These interfaces are summarized below:

- **JBaseField** – Ensures that this interface can be converted to a Field object
- **ParentField** – Field serves as a parent for one or more child fields
- **ChildField** – Field is owned by a parent field
- **PointableField** – Pointer Field can point to this field
- **PointerField** – This field stores a pointer to a row in another field


JBaseField
«Interface»
+ toField(): Field
ChildField
«Interface»
+ resize(parent: ParentField)
+ getOwner(): ParentField
ParentField
«Interface»
+ addChild(child: ChildField)
+ deleteChild(child: ChildField)
+ allChildren(): ChildField[]
+ getDepth(): int
PointableField
«Interface»
+ isValidRow(row: int): boolean
+ addPointer(pointer: PointerField)
+ deletePointer(pointer: PointerField)
+ allPointers(): PointerField[]
PointerField
«Interface»
+ getPoint(): PointableField

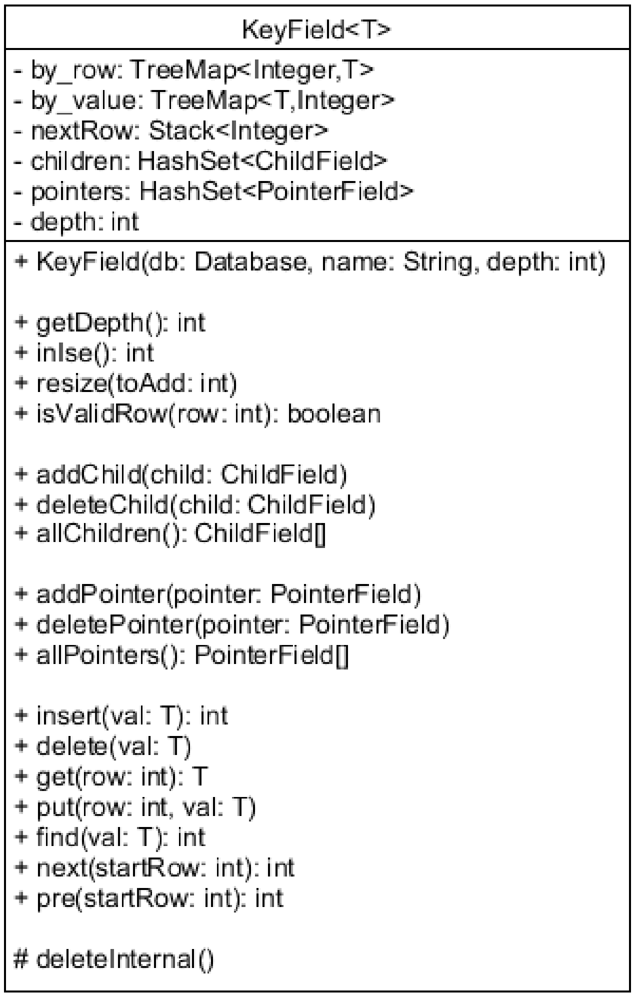
Key Field

Extends: Field

Implements: ParentField, PointableField

Keys essentially serve the purpose of a primary key for JBase. Each entry in a key must be unique, and are added or removed using the insert() and delete() methods. Keys cannot be updated by calling put(). Because the values are all unique, the row for a given value can be found by calling find(). You can also iterate over keys in order using next() and pre(), which return the row of the next element.

Because keys can own child fields, keys internally store a list of children fields. Since keys can be pointed to by pointer fields, keys also store a list of pointer fields. When a key is resized or deleted, all children (and pointers) are also resized or deleted to match the state of the key.

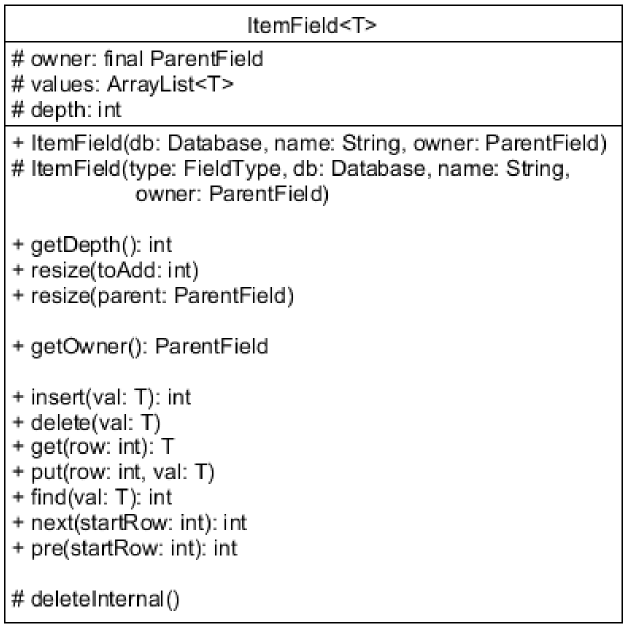

KeyField<T>
- by_row: TreeMap<Integer,T>
- by_value: TreeMap<T,Integer>
- nextRow: Stack<Integer>
- children: HashSet<ChildField>
- pointers: HashSet<PointerField>
- depth: int
+ KeyField(db: Database, name: String, depth: int)
+ getDepth(): int
+ inIse(): int
+ resize(toAdd: int)
+ isValidRow(row: int): boolean
+ addChild(child: ChildField)
+ deleteChild(child: ChildField)
+ allChildren(): ChildField[]
+ addPointer(pointer: PointerField)
+ deletePointer(pointer: PointerField)
+ allPointers(): PointerField[]
+ insert(val: T): int
+ delete(val: T)
+ get(row: int): T
+ put(row: int, val: T)
+ find(val: T): int
+ next(startRow: int): int
+ pre(startRow: int): int
# deleteInternal()

Item Field

Extends: Field

Implements: ChildField

Items serve the purpose of columns that don't have unique data. All item fields are owned by a parent field, and have the same depth as the parent field. Items are also automatically resized whenever a key is resized. You cannot call insert(), delete(), or find() on an item. Rather, values in an item are modified using put() and get(). You also cannot iterate over an item with next() and pre().



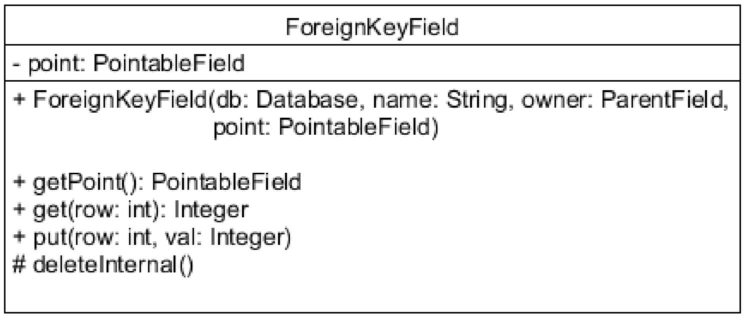
```
ItemField<T>
# owner: final ParentField
# values: ArrayList<T>
# depth: int
+ ItemField(db: Database, name: String, owner: ParentField)
# ItemField(type: FieldType, db: Database, name: String,
            owner: ParentField)

+ getDepth(): int
+ resize(toAdd: int)
+ resize(parent: ParentField)

+ getOwner(): ParentField

+ insert(val: T): int
+ delete(val: T)
+ get(row: int): T
+ put(row: int, val: T)
+ find(val: T): int
+ next(startRow: int): int
+ pre(startRow: int): int

# deleteInternal()
```

Foreign Key Field

Extends: ItemField

Implements: PointerField

Foreign keys have many of the same properties as an item field, but can only store a row of a pointable field. Hence, all calls to put() first validate the row by calling PointableField.isValidRow(). Other than these small differences, foreign keys operate exactly the same as item fields.



```
ForeignKeyField
- point: PointableField
+ ForeignKeyField(db: Database, name: String, owner: ParentField,
                  point: PointableField)

+ getPoint(): PointableField
+ get(row: int): Integer
+ put(row: int, val: Integer)
# deleteInternal()
```

Access Control Lists

Overview

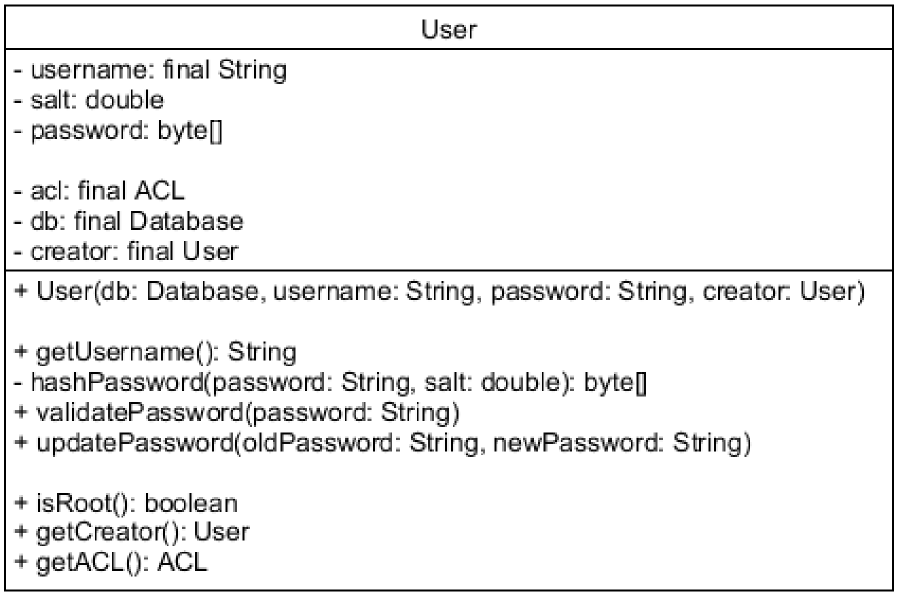
Access Control Lists, or ACL's, are used to protect the integrity of the database by ensuring that users can only do certain actions. JBase implements Access Control Lists (ACL) using both a User and an ACL class.

User Object

This object represents an entity that can access the database. Each user in the database has an ACL object, which stores the list of actions that the user can and cannot do in the database. Performing an unauthorized action results in a permission exception being throw in the database.

For ensuring security, the user password is not stored as a string, bur rather as an array of bytes obtained by hashing the password (using SHA-256). The password is also salted to help protect against reverse hash tables.

Each database has one root user, which implicitly can do any action and not remove any permissions from itself. The root user ditinguished by being the only user with a null creator.

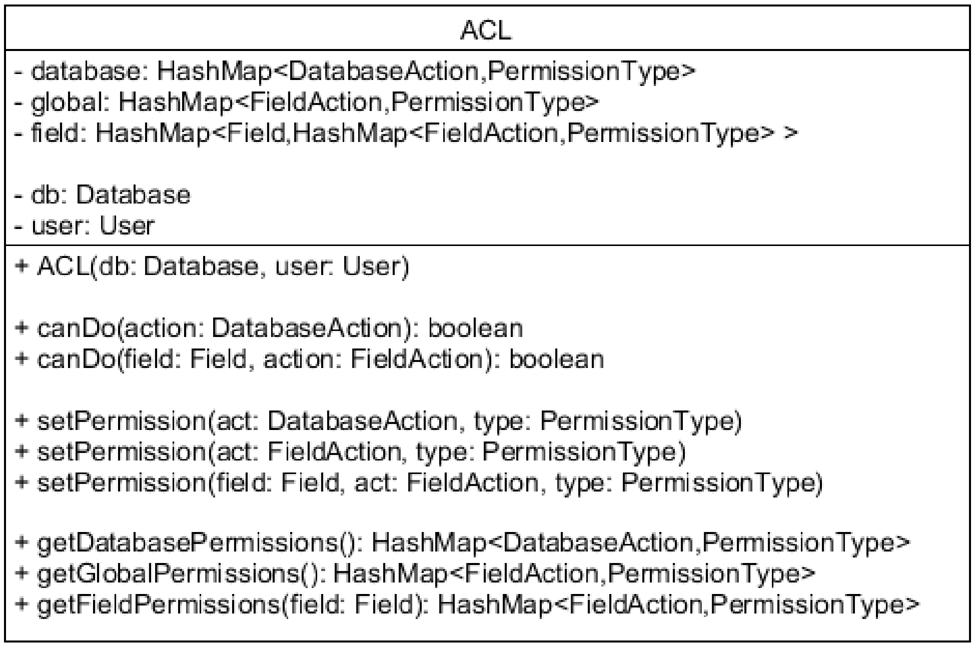


```
User

- username: final String
- salt: double
- password: byte[]

- acl: final ACL
- db: final Database
- creator: final User

+ User(db: Database, username: String, password: String, creator: User)

+ getUsername(): String
- hashPassword(password: String, salt: double): byte[]
+ validatePassword(password: String)
+ updatePassword(oldPassword: String, newPassword: String)

+ isRoot(): boolean
+ getCreator(): User
+ getACL(): ACL
```

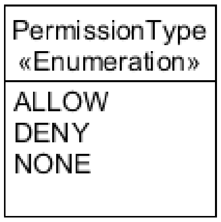
ACL Object (Access Control List)

This object is used to limit the actions that a user can do in the database. The ACL uses a variety of enumerations to define actions in the datbase or on a field, as well as whether or not the action is allowed or denied.

Database permissions deal with database-wide actions, such as modifying users or saving the database to a file. Global and field permissions are about manipulating fields in the database. Field specific permissions override global permissions. Checking the ACL is as simple as calling ACL.canDo() (all of the logic for permissions is handled internally).



```
ACL
- database: HashMap<DatabaseAction,PermissionType>
- global: HashMap<FieldAction,PermissionType>
- field: HashMap<Field,HashMap<FieldAction,PermissionType> >

- db: Database
- user: User
+ ACL(db: Database, user: User)

+ canDo(action: DatabaseAction): boolean
+ canDo(field: Field, action: FieldAction): boolean

+ setPermission(act: DatabaseAction, type: PermissionType)
+ setPermission(act: FieldAction, type: PermissionType)
+ setPermission(field: Field, act: FieldAction, type: PermissionType)

+ getDatabasePermissions(): HashMap<DatabaseAction,PermissionType>
+ getGlobalPermissions(): HashMap<FieldAction,PermissionType>
+ getFieldPermissions(field: Field): HashMap<FieldAction,PermissionType>
```



```
PermissionType
«Enumeration»
ALLOW
DENY
NONE
```

Exception Handling

Overview

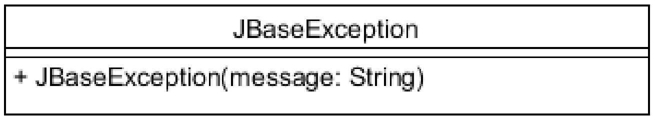
JBase has a variety of exceptions to abstract the different types of actions that can go wrong in the database. All exceptions derive from JBaseException, and can be further derived from DatabaseException, FieldException, or PermissionException. All of the exceptions are listed below, following this hierarchy order.

Exceptions

JBaseException

Extends: java.lang.RuntimeException

Generic exception in a JBase database

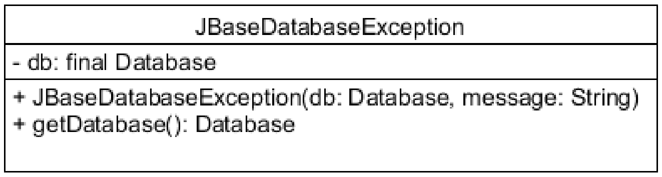


Database Exceptions

JBaseDatabaseException

Extends: JBaseException

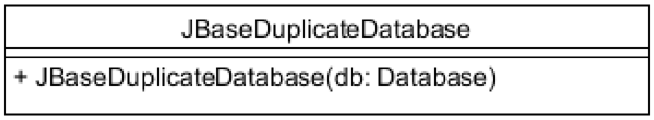
Generic exception that happens on a specific JBase database instance


JBaseDatabaseException
- db: final Database
+ JBaseDatabaseException(db: Database, message: String)
+ getDatabase(): Database

JBaseDuplicateDatabase

Extends: JBaseDatabaseException

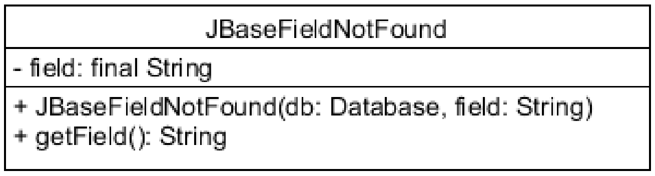
Trying to load a database that already exists in memory



JBaseFieldNotFound

Extends: JBaseDatabaseException

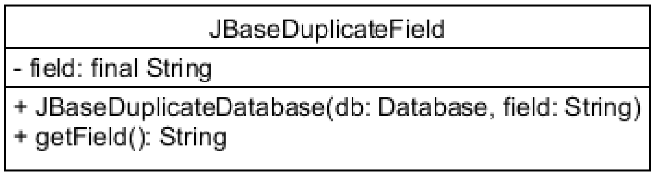
Trying to call Database.getField() on a field that doesn't exist in the database



JBaseDuplicateField

Extends: JBaseDatabaseException

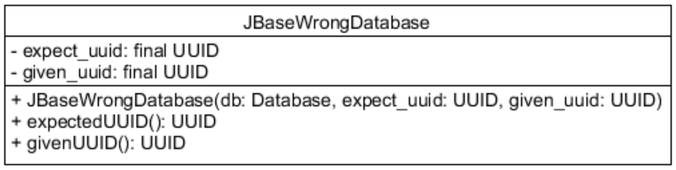
Trying to create a field in the database that already exists



JBaseWrongDatabase

Extends: JBaseDatabaseException

Database being restored doesn't match the UUID of the current database

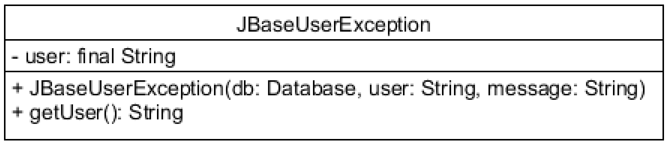

JBaseWrongDatabase
- expect_uuid: final UUID
- given_uuid: final UUID
+ JBaseWrongDatabase(db: Database, expect_uuid: UUID, given_uuid: UUID)
+ expectedUUID(): UUID
+ givenUUID(): UUID

User Exceptions

JBaseUserException

Extends: JBaseDatabaseException

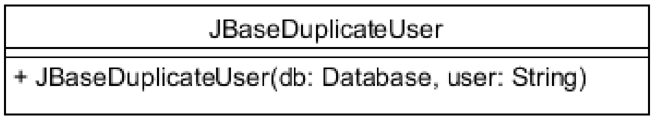
Generic exception that happens with a given user in the database


JBaseUserException
- user: final String
+ JBaseUserException(db: Database, user: String, message: String)
+ getUser(): String

JBaseDuplicateUser

Extends: JBaseUserException

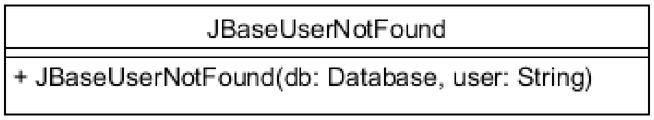
Trying to create a user that already exists in the database



JBaseUserNotFound

Extends: JBaseUserException

Trying to get a user that doesn't exist in the database

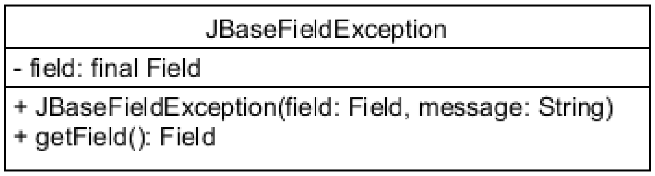


Field Exceptions

JBaseFieldException

Extends: JBaseException

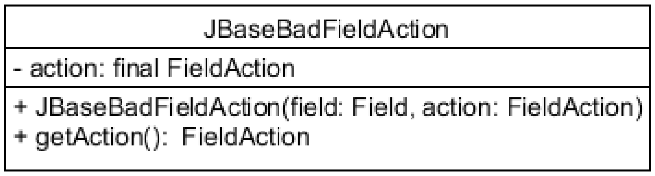
Generic exception that happens with a field in the database



JBaseBadFieldAction

Extends: JBaseFieldException

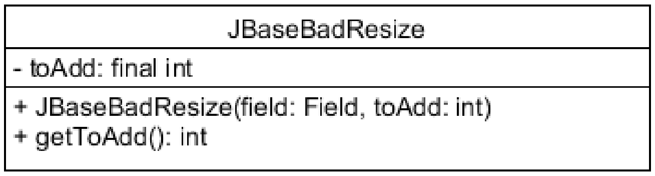
Sub field does not support this action call (such as trying to call insert on an item, or put on a key)


JBaseBadFieldAction
- action: final FieldAction
+ JBaseBadFieldAction(field: Field, action: FieldAction)
+ getAction(): FieldAction

JBaseBadResize

Extends: JBaseFieldException

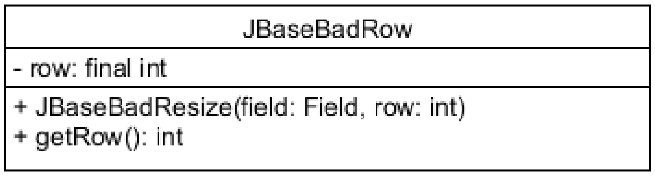
Invalid amount passed to resize function



JBaseBadRow

Extends: JBaseFieldException

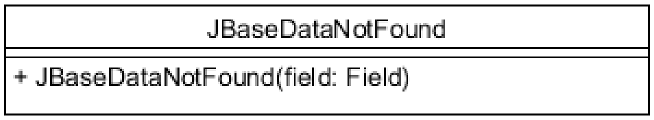
Bad row passed to get, put, next or pre method



JBaseDataNotFound

Extends: JBaseFieldException

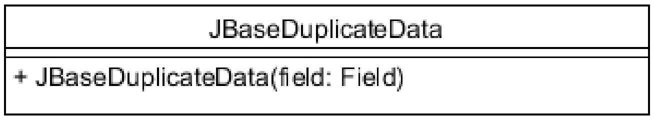
Data doesn't exist when trying to do a delete or find on a field



JBaseDuplicateData

Extends: JBaseFieldException

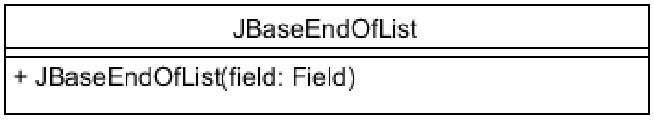
Trying to insert a value that already exists into a key field



JBaseEndOfList

Extends: JBaseFieldException

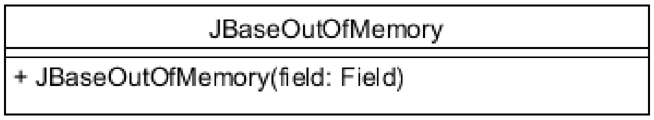
Call to next or pre has no next or previous entry



JBaseOutOfMemory

Extends: JBaseFieldException

Trying to insert into a key that has no empty slots remaining

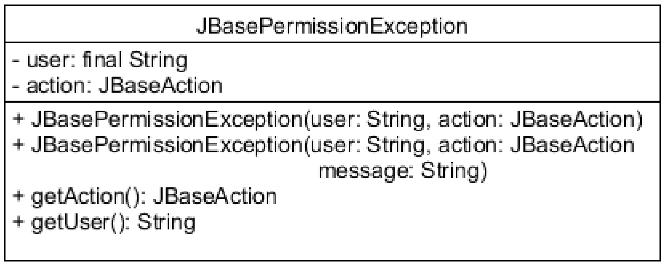


Permission Exceptions

JBasePermissionException

Extends: JBaseException

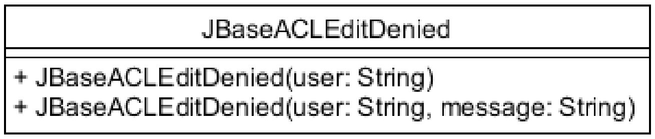
Generic exception thrown when a user ACL permission is violated


JBasePermissionException
- user: final String
- action: JBaseAction
+ JBasePermissionException(user: String, action: JBaseAction)
+ JBasePermissionException(user: String, action: JBaseAction message: String)
+ getAction(): JBaseAction
+ getUser(): String

JBaseACLEditDenied

Extends: JBasePermissionException

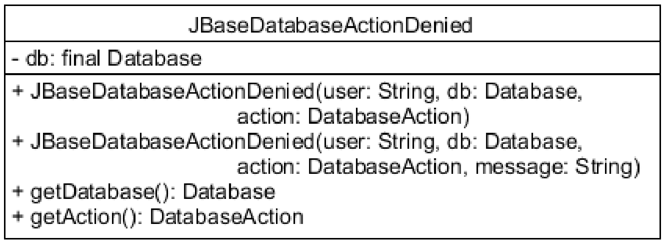
Trying to update the permissions of an ACL when the user doesn't have permission to do so


JBaseACLEditDenied
+ JBaseACLEditDenied(user: String)
+ JBaseACLEditDenied(user: String, message: String)

JBaseDatabaseActionDenied

Extends: JBasePermissionException

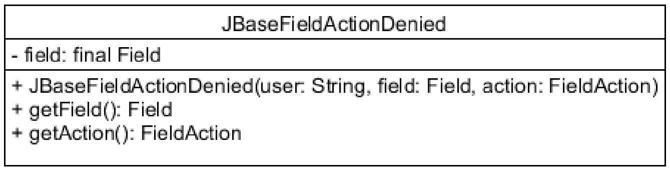
Trying to perform a database action that the user doesn't have permission to do so


JBaseDatabaseActionDenied
- db: final Database
+ JBaseDatabaseActionDenied(user: String, db: Database, action: DatabaseAction)
+ JBaseDatabaseActionDenied(user: String, db: Database, action: DatabaseAction, message: String)
+ getDatabase(): Database
+ getAction(): DatabaseAction

JBaseFieldActionDenied

Extends: JBasePermissionException

Trying to perform a field action that the user doesn't have permission to do so

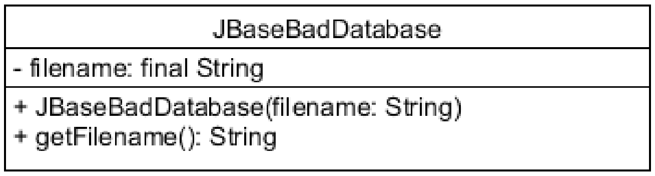

JBaseFieldActionDenied
- field: final Field
+ JBaseFieldActionDenied(user: String, field: Field, action: FieldAction)
+ getField(): Field
+ getAction(): FieldAction

Other Exceptions

JBaseBadDatabase

Extends: JBaseException

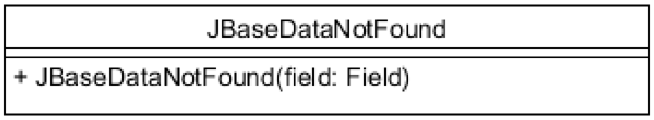
Trying to load a database from an invalid file



JBaseDatabaseNotFound

Extends: JBaseException

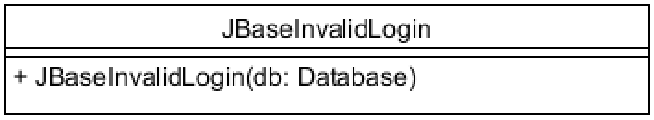
Trying to open a database that doesn't exist



JBaseInvalidLogin

Extends: JBaseException

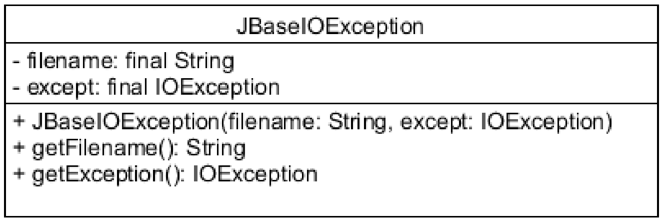
Bad credentials given when trying to log into an existing database



JBaseIOException

Extends: JBaseException

File I/O problem when saving, restoring, or loading a database


JBaseIOException
- filename: final String
- except: final IOException
+ JBaseIOException(filename: String, except: IOException)
+ getFilename(): String
+ getException(): IOException

Terminal Interface Design

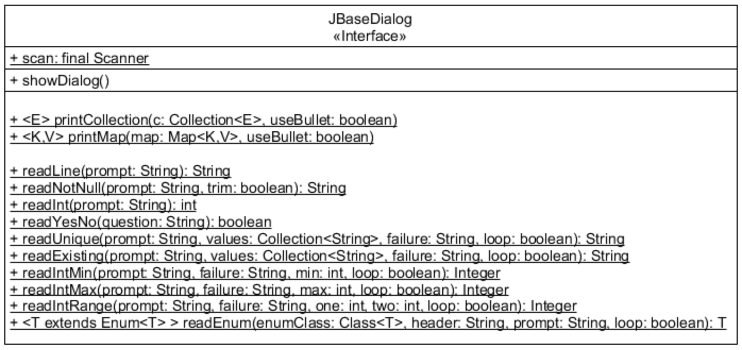
Overview

The JBase terminal interface is composed of classes that implement the JBaseDialog interface. Each class functions as a different dialog in the database, making it easier to abstract the parts of the interface as a whole. In addition, many dialogs have sub-functions that implement a small feature of the dialog (such as saving a file or creating a new field). The dialogs are listed below:

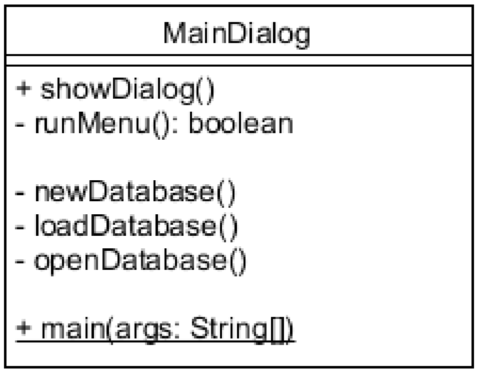
- **Main Dialog** – Allows the user to create a new database, log in to an existing database, or load a database from a file.
- **Database Dialog** – Allows the user to open the field dialog, open the user dialog, save the database to a file, restore the database from a file, or drop the database from memory.
- **Select Key Dialog** – Allows the user to create new key fields, delete key fields, or open the key dialog for a key in the database.
- **Key Dialog** – Allows the user to create new items or foreign keys, delete a child field, resize the key field, insert new records, update existing records, or delete records in a key field.
- **User Dialog** – Allows the user to create new users, delete existing users, and update user permissions in the database.
- **ACL Dialog** – Allows the user to modify the access control list for a given user in the database

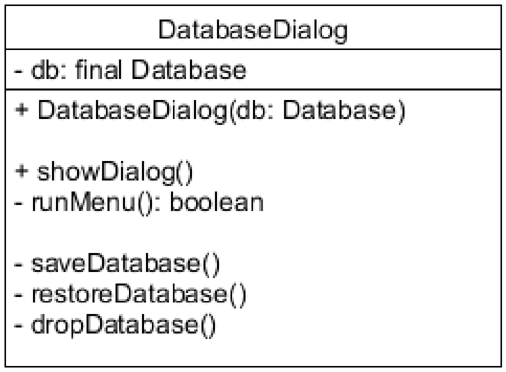
JBaseDialog

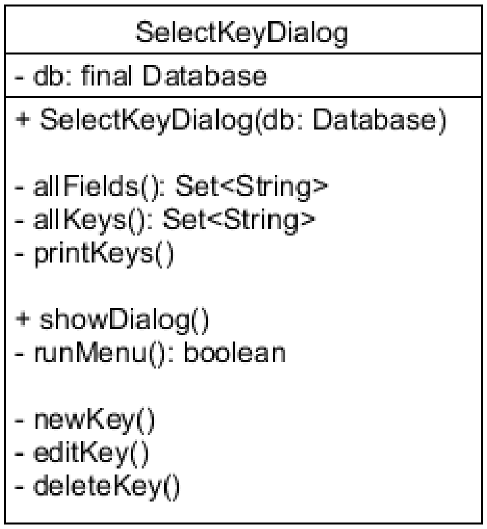
This is the interface that requires all terminal dialogs to implement the showDialog() method. This interface also has a variety of static user input functions (readLine, readInt, readYesNo, etc.) to make it easier to abstract terminal input.

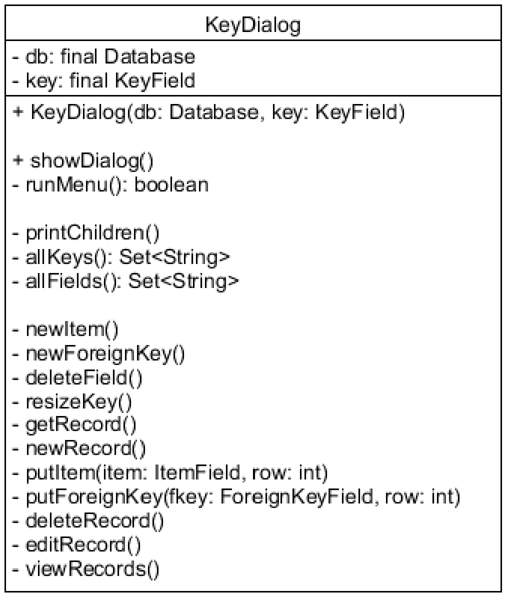

JBaseDialog
«Interface»
+ scan: final Scanner
+ showDialog()
+ <E> printCollection(c: Collection<E>, useBullet: boolean)
+ <K,V> printMap(map: Map<K,V>, useBullet: boolean)
+ readLine(prompt: String): String
+ readNotNull(prompt: String, trim: boolean): String
+ readInt(prompt: String): int
+ readYesNo(question: String): boolean
+ readUnique(prompt: String, values: Collection<String>, failure: String, loop: boolean): String
+ readExisting(prompt: String, values: Collection<String>, failure: String, loop: boolean): String
+ readIntMin(prompt: String, failure: String, min: int, loop: boolean): Integer
+ readIntMax(prompt: String, failure: String, max: int, loop: boolean): Integer
+ readIntRange(prompt: String, failure: String, one: int, two: int, loop: boolean): Integer
+ <T extends Enum<T> > readEnum(enumClass: Class<T>, header: String, prompt: String, loop: boolean): T

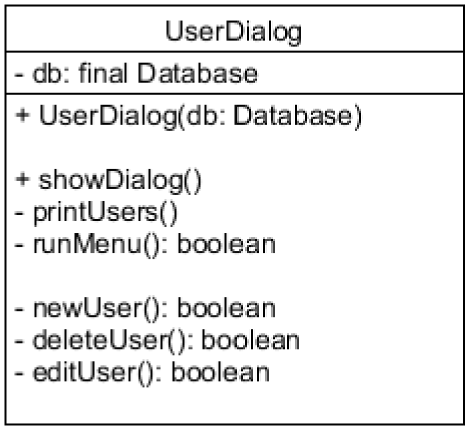
Interface UML Diagrams

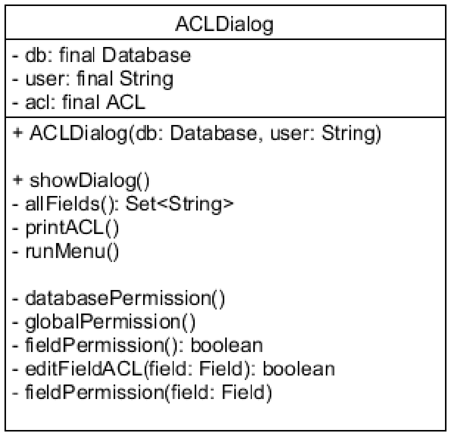











ACLDialog
- db: final Database
- user: final String
- acl: final ACL
+ ACLDialog(db: Database, user: String)
+ showDialog()
- allFields(): Set<String>
- printACL()
- runMenu()
- databasePermission()
- globalPermission()
- fieldPermission(): boolean
- editFieldACL(field: Field): boolean
- fieldPermission(field: Field)

Algorithm

Overview

Due to the complexity of JBase, the algorithm for all of the classes is stored using javadocs written in the code and not explicitly in this document. However, the provided makefile has a target to build the javadocs into a browsable website. At the terminal, run the following command:

```
make docs
```

This command will automatically generate the javadocs, and store them in a folder called "docs." To use the documents, simply open docs/index.html in a browser. The documents contain all information about how each function works, as well as a description of the input and output of each function (and what exceptions a function might throw).