# GETTING STARTED WITH FIREBIRD

## Document identification

| | |
|---|---|
| | |
| **Getting started with Firebird** | |

## Status of reviews

| ***Review n.*** | ***Reason to review*** | ***Revised by*** | ***Date*** |
|---|---|---|---|
| 1 | Added some notes about isql usage | pabloj@users.sourceforge.net | 01/03/05 |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

# GETTING STARTED WITH FIREBIRD

This writing is meant to be a “quick and dirty” guide to the main features of Firebird, as I learned them day by day using the database.

### Essentials (and where to get them)

In order to follow this tutorial you’ll need to download and install the database itself plus some useful addictions:

- Firebird 1.5, the database server itself, is available for many platforms, I chose the SuperServer version.

Not mandatory, but useful:

- IBEasy+, graphical tool for management (Windows only) (http://nte-socio.univ-lyon2.fr/Marc_Grange/TeleCharger_en.htm).

All examples of SQL commands are shown as issued from the command line or the “SQL editor” interface of a graphical tool, this to teach the beginner to trust the command line (ok, not for writing stored procedures) and to get a style which can be applied to both Windows and *NIX.

### Installation

To install Firebird on Windows you just need to double-click the installer package and follow instructions. I chose to install it as a service and to use the Guardian, to ensure it keeps running (not really mandatory).

### Basic tasks

Ok, now everything seems up and running (check the service status from control panel to be shure), let’s get on with some tests.
By referencing to “Basic tasks” I mean:

- Connect to a database server
- Create a database
- Connect to the database
- Create a user
- Create structures
- Manipulate data

- Backup and restore database

**So I’ll start by connecting to Firebird Server**:

Note that there is only one security database for every database server, where all users and their profiles are kept.
The default username is “SYSDBA” and password “masterkey”, this is well known to anyone using Firebird or Interbase, so we must change it, this can be accomplished with the utility gsec, located in the ./bin subfolder of Firebird installation, from the command line type:

```
>gsec –user SYSDBA –password masterkey
```

If everything is correct it will answer with a prompt:

```
GSEC>
```

Now put in a new password, like this:

```
GSEC> modify SYSDBA –pw guess_me
```

After pressing ENTER the new password “guess_me” will substitute “masterkey” (note that all passwords are of 8 characters, meaning that the chars past the eighth will be ignored.

Now leave the GSEC utility by typing “quit” and pressing ENTER

**Create a database**:

Even if Firebird comes with a sample database, called EMPLOYEE.FDB, I’ll show you how to create a new one from scratch.
This is done by using a command line shell “isql” also located in the ./bin folder, type:

```
>isql
```

and you’ll see:

```
use CONNECT or CREATE DATABASE to specify a database
```

note that every command must end with a semicolon!!
So issue a create database command:

```
SQL> CREATE DATABASE ‘C:\firedata\first_db.fdb’ PAGE_SIZE 8192 LENGTH 1000
USER ‘SYSDBA’ PASSWORD ‘guess_me’ DEFAULT CHARACTER SET ISO8559_1;
SQL>quit;
```

You must be connected locally on the computer.

This will create a database file named “first_db.fdb” in the “C:\firedata” folder, after a few seconds the database will be created and the command prompt will be back, just type quit to leave it.
The database will be created with a length of 1000 pages of 8192 bytes each.

**Add files to database**

Usually Firebird databases seem to be made of just one file that can become very big, with many practical limitations.
The solution lies obvoiusly in adding other files to the database, it is possible to specify a size for the file to be created, the size is expressed in pages, those pages default to 1024 bytes each, with a command like:

```
SQL> ALTER DATABASE ADD FILE 'employee2.gdb'
```

Or, specifying a size:

```
SQL> ALTER DATABASE ADD FILE 'employee2.gdb' LENGTH 10000
```

Note that exclusive access to the database is needed in order to alter it.

**Connect to the database**:

Now that we have a brand new database, we can connect to it, using isql, of course:

```
>isql

SQL> CONNECT “C:\firedata\first_db.fdb” user ‘SYSDBA’ password ‘guess_me’;
```

Or

```
SQL>   CONNECT   “localhost:C:\progra~1\firebird\firebird_1_5\examples\employee.fdb”
user ‘sysdba’ password ‘masterkey’;
```

You can also specify all parameters in one line:

```
isql -u sysdba -p masterkey “c:\programmi\firebird\firebird_1_5\examples\employee.fdb"
```

If everything goes fine isql will inform you that the connection has been successfully established.

**Configure an alias**

Tired of typing something like:

```
SQL>   CONNECT   “localhost:C:\progra~1\firebird\firebird_1_5\examples\employee.fdb”
user ‘sysdba’ password ‘masterkey’;
```

Each time you need to connect to Firebird? Me too! It's time to open a file named "aliases.conf" which is found in "C:\progra~1\firebird\firebird_1_5\" and add an alias for each of your databases, every alias consists of a line like:

```
employee = C:\Programmi\Firebird\Firebird_1_5\examples\employee.fdb
```

So the connection string above will evolve into:

```
SQL> CONNECT "localhost:employee" user 'sysdba' password 'masterkey';
```

Much better, isn't it?

## The example database

Firebird comes with an example database, named "EMPLOYEE.FDB" which I'll use extensively in this tutorial, it is located in the ./examples subdirectory. You can take a quick look at it's content by connecting to it and issueing a:

```
SQL> SHOW TABLES;

    COUNTRY                         CUSTOMER
    DEPARTMENT                        EMPLOYEE
    EMPLOYEE_PROJECT                       JOB
    PHONE_LIST                       PROJECT
    PROJ_DEPT_BUDGET                      SALARY_HISTORY
    SALES
```

It will give you a list of all the tables in the database.

## The "Undefined service : gds_db/tcp" error

You usually get this error while connecting to Firebird through TCP/IP protocol, this usually means there is not an entry like "gds_db 3050/tcp" in the SERVICES file, which is located in:

| | |
|---|---|
| Unix | /etc/SERVICES |
| NT | WINNT\system32\drivers\etc\SERVICES |

## Create a user:

Of course not all users will connect to the database with administrator privileges, so we must add some other users:

```
"C:\Program Files\FireBird\bin\gsec.exe" -user sysdba -password masterkey
GSEC> ADD username_here -pw password_here -fname familyname_here -mname realname_here -lname shortname_here
GSEC> quit;
```

The options specified above have the following meaning:

| Option | Description |
|---|---|
| -pw | User's password |
| -u[id] | User ID |
| -g[id] | Group ID |
| -f[name] | User's first name |
| -mn[ame] | User's middle name |
| -l[name] | User's last name |

Users have privileges on database objects, those privileges are managed by the GRANT system, the SQL statement used are GRANT and REVOKE, here I'll show a bit about them.

First of all a list of the privileges that can be granted to a user:

| Privilege | Definition of Privileges |
|---|---|
| Insert | Allows insertion of new rows into table |
| Update | Allows existing rows to be updated |
| Delete | Allows existing rows to be deleted |
| Select | Allows user to view/query the rows in a table |
| Execute | Allows a user to execute a procedure |
| Reference | Allows server to lookup rows in a primary/foreign key relationship |
| All | Shortcut to assign insert, update, delete, select, references to a user |

Like the ALL privilege, there is a shortcut to GRANT the same privilege to all users of the database, this comes through granting the privilege to user PUBLIC, which represents all the database users.

At first only SYSDBA can grant privileges to other user, thus preventing any delegation over the privilege system, to solve this the DBA is given the option to allow others to grant privileges on objects, this is accomplished by adding the WITH GRANT OPTION clause to a normal grant statement. Note that the granted user can grant the privilege assigned only.

An example query is:

```
SQL >GRANT select ON country TO username_here WITH GRANT OPTION;
```

This allows "username_here" to give select privilege on table country to other users.

If you need to take back privileges from users you can use the REVOKE statement (this does not prevent granting privileges to users, but just removes privileges already granted), the syntax is:

```
SQL>REVOKE privilege ON objectname FROM user;
```

Of course you can REVOKE the ALL privilege as a shortcut, even if the user was not granted ALL on object. Note that revoking privileges from a user who had granted access to others (do you remember "… with grant option"?) will automatically revoke privileges to those other users.

Be careful revoking privileges from PUBLIC!!!

A more general way of managing privileges is to use ROLES, which are a general definition of privileges, a role is granted some privileges and users are given a role, thus inheriting privileges defined for that role.

The basic syntax is:

```
SQL > CREATE ROLE my_new_role;
```

By default a newly created role has no privileges and you must be SYSDBA to create a role, to grant to a role you can either be the owner of the object or a user who has been granted "with grant option".

When a role is created you can start granting privileges to it, like:

```
SQL > GRANT select ON TABLE country TO my_new_role;
```

Or

```
SQL > GRANT insert, select ON TABLE country TO my_new_role;
```

Or even

```
SQL > GRANT ALL ON country TO my_new_role;
```

After granting all needed privileges to the role, in order to make it usable, we need to grant the role to users, then each user will connect to the database specifying the role and will get the related privileges.

The syntax for granting roles is like:

```
SQL > GRANT my_new_role TO username;
```

You can also specify a comma separated list of roles or users and there is also a final clause available, the "…WITH ADMIN OPTION", which allows the user to grant role to others.

## Create structures:

### Create a table:

Creating a table in Firebird is pretty straightforward, with the usual CREATE TABLE statemet, it supports many datatypes, among the others:

1. SMALLINT
2. BIGINT
3. INTEGER
4. DECIMAL
5. FLOAT
6. DOUBLE
7. CHAR
8. VARCHAR
9. BLOB
10. TIMESTAMP

Boolean fields are not supported, but can be emulated with INTEGER or CHAR(1) fields.

A lot of clauses can be included in the CREATE TABLE, here I'll show some examples, just to show you simple table creation, but also support for DEFAULT values, CHECK constraints, PRIMARY and FOREIGN KEYS:

```
CREATE TABLE dept (dept_code BIGINT NOT NULL, dept_name VARCHAR(255) NOT
NULL);
```

You must be using SQL dialect 3 to have BIGINT datatype!!

```
CREATE TABLE emp (
emp_code BIGINT NOT NULL PRIMARY KEY,
dept_code BIGINT NOT NULL,
sal FLOAT DEFAULT 0,
emp_age INTEGER,
CHECK(emp_age BETWEEN 0 AND 150),
FOREIGN KEY (dept_code) REFERENCES dept (dept_code) ON DELETE CASCADE
ON UPDATE CASCADE);
```

In order to have foreign keys you must have declared the referenced columns as unique, so we'll alter DEPT table to accommodate this:

```
ALTER TABLE dept ADD PRIMARY KEY (dept_code);
```

Now the above statement can be run safely.
This is just a basic into to table creation in Firebird, more infos can be found at http://www.ibphoenix.com/main.nfs?a=ibphoenix&page=ibp_60_sqlref#RSf13665 .

**Where is "autoincrement" gone??**

One of the most used features of MySQL, MsSQL and MsAccess is the "autoincrement" field, while this is not one of my favourite features, it is often requested.
Firebird does not have this feature, but it has a workaround for it, which is much more powerful!!!
In this example I'll show you how to build it, let's start with a simple table:

```
CREATE TABLE new_one
(
col1 integer not null,
col2 char(10),
PRIMARY KEY (field1)
);
```

Now comes the trick, we will create a “generator”, which is a source of numbers.

```
CREATE GENERATOR my_gen_id;
```

A generator can be used to retrieve numeric values to be inserted in the table, like:

```
INSERT INTO new_one (col1, col2) VALUES (gen_id(my_gen_id, 1), 'phrase');
```

Which means inserting a value taken from the generator incrementing it’s preceding value of 1 unit.
Generators are a bit more complicated than autoincrement, but a lot more flexible, it’s easy to read it’s value (the “last” one inserted) without modifying it, with this simple query, which is the equivalent to MySQL’s `LAST_INSERT_ID()`:

```
SELECT gen_id(my_gen_id, 0) FROM ...
```

Which means that if you use one generator for each table it’s easier to keep track of the various values, another important thing is that you can manipulate the generator’s current value, just like:

```
SET GENERATOR my_gen_id TO [some_value];
```

So far generators have shown to be useful and powerful, but a bit difficult to use, now I’ll introduce another advanced function of Firebird (and other databases), a trigger, which is an action which is “triggered” by an event. Triggers can be used for a lot of tasks, but here I’ll use one to make easier to add an “autoincrement-like” functionality to Firebird.
This trigger will automatically insert a new value in the table each time a row is inserted.
Let’s create it:

```
CREATE TRIGGER autoincrementor_id FOR new_one
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
 IF (NEW.col1 IS NULL) THEN
  NEW.col1 = GEN_ID(my_gen_id,1);
END
```

This way we created a trigger on table `new_one` (the `FOR` ... clause), that will kick in before a row is inserted (the `ACTIVE BEFORE INSERT ... clause`) which will add a value coming from the generator if no value is supplied.

**Manipulate data**:

**Insert data:**

One of the most simple queries I can imagine on the Employee sample database is adding a country and currency to the standard Country table. The syntax is:

```
SQL >INSERT INTO country VALUES ('Portugal', 'Escudo');
```

But … the query seems to run fine but no data is inserted! This happens because Firebird supports transactions and so their COMMIT and ROLLBACK, for our example query this means that nothing gets written to the database until a formal COMMIT is issued.

So, in order to have our data written to the database, we must issue two commands:

```
SQL >INSERT INTO country VALUES ('Portugal', 'Escudo');
SQL >COMMIT;
```

It is also possible to fill a table with values coming from another one with a query like:

```
SQL >INSERT INTO table2 (colums) SELECT columns FROM table1;
```

**Delete data:**

We inserted new data, but how about deleting? The syntax is usual:

```
SQL >DELETE FROM country WHERE country = ‘Portugal’;
```

**Commit or Rollback?**

Support for transaction basically means that every query issued into a transaction can be reverted with a ROLLBACK, an example of this is given in the next section.
An even more interesting feature is the support for savepoints and thus the ability to create nested transactions, in other words the ability to create some sub-transactions inside a general transaction and the chance to rollback only one of those sub-transactions.

A simple example can be:

```
C:\Programmi\Firebird\Firebird_1_5\bin>isql
Use CONNECT or CREATE DATABASE to specify a database
SQL> connect "localhost:employee" user 'sysdba' password 'masterkey';
Database:  "localhost:employee", User: sysdba
SQL> create table test (name varchar(50));
SQL> commit;
SQL> insert into test values ('me');
SQL> commit;
SQL> insert into test values ('myself');
SQL> savepoint y;
SQL> delete from test;
SQL> select * from test;
SQL> rollback to y;
SQL> select * from test;

NAME
==================================================

me
myself

SQL> rollback;
SQL> select * from test;

NAME
==================================================

me

SQL>
```

Note that we created a test table, then inserted one value and committed, after that we inserted another value (thus beginning a new transaction) and then set a savepoint (the beginning of a nested transaction), after adding this savepoint we performed a delete, the select issued immediately after retrieved no results. That's the usual behaviour, but now we have a choice, to rollback to savepoint or to the beginning of the transaction, we rollback to the savepoint and issue a new select, now we get 2 rows, then, to show you the difference between a nested transaction and the main transaction we issue a rollback, (i.e. to the beginning of the main transaction) and then issue a select, the result is made of only one row, because the second insert was not followed by a commit!!!

This feature can be extremely valuable in building your own transactional applications!

**Select data:**

The SQL dialect used by Firebird is quite complete and advanced, an user can issue fairly complex queries, like this one (note support for LEFT and RIGHT OUTER JOIN, GROUP BY and ORDER BY with the ability to order ASCending or DESCending):

```
SQL>SELECT count(e.emp_no) AS emp_4_proj, d.department, p.proj_name FROM
employee e RIGHT OUTER JOIN department d ON e.dept_no = d.dept_no LEFT OUTER
JOIN employee_project ep ON e.emp_no = ep.emp_no LEFT OUTER JOIN project p ON
ep.proj_id = p.proj_id GROUP BY d.department, p.proj_name ORDER BY d.department
DESC;
```

Or this other, which has no particular meaning, but shows you support for UNION ALL and subqueries:

```
SQL >SELECT count(*) AS total FROM customer;
UNION ALL
```

```
        SELECT count(*) AS total FROM customer WHERE customer.country IN (SELECT
country FROM country);
```

Another nice feature of Firebird is the ability to combine queries, suppose you have a table with all your registered customers (named “customers”) and another one with your sales for this month (named “sales_month”), you want to see side by side the total number of customers, which will be taken from the “customers” and the number of active customers, meaning the number of registered customers that have made at least a purchase in the last month (taken from the “sales_month” table). The query will be something like:

```
SQL > select count(distinct(sm.customer_id)) as active_customers,
(select count(cm.customer_id) from customers cm) as registered_customers
from
sales_month sm
```

## Speeding up a select … indexing tables

An index can dramatically improve the speed of your queries, it summarizes the informations contained in specific columns and helps the database engine to avoid scanning the whole table for results.

A sample index can be created on a column (or several columns combined) with a simple query:

```
SQL > CREATE INDEX new_index ON my_table (my_first_col, my_second_col);
```

It’s important to be avare of other clauses, the UNIQUE and ASC or DESC, which can be inserted as “CREATE UNIQUE DESC INDEX my_index ON ….”, of course UNIQUE means that the values in the index are unique and ASC or DESC impose a preordering of values (ASC is the default value if none is specified). Be aware that BLOB columns can’t be indexed.

Index based query performance can be improved by using SET STATISTICS to recompute the index and keep it up to date, the same can be achieved by deactivating and reactivating it.

## More on speeding … the Explain plan

An extremely useful feature of relational databases is the “explain plan”,which shows us how the database engine is going to execute a query. It allows the DBA to check if indexes are used and how tables are queried. One of the most effective ways of using it is to use the Firebird option “PLANONLY” which means that the query to be analyzed is submitted to the server and the plan retrieved, but the query is not executed! Note that this is also an effective way of checking the SQL syntax of a statement, much like Oracle’s parse statement.

The planonly option can be set to ON or OFF at your option, remember to turn it off to obtain results for your queries.

The plan is accessed through ISQL this way:

```
        >isql

        SQL> CONNECT “C:\firedata\first_db.fdb” user ‘SYSDBA’ password ‘guess_me’;
…
        SQL> SET PLANONLY ON;
        SQL> SELECT a.emp_no, b.currency from employee a inner join country b on
a.job_country = b.country;
```

After submitting the query we retrieve the plan, which shows usage of the primary key of table country:

```
PLAN JOIN (A NATURAL,B INDEX (RDB$PRIMARY1))
        SQL> SET PLANONLY OFF;
```

This way we can check the execution of more complex queries and find areas of improvement

**Where is “limit” gone?**

One of the most popular features from MySQL, the LIMIT clause is selects, which makes easy record paging, is available in Firebird, under a slightly different syntax:

```
        SQL> SELECT FIRST 2 SKIP 1 country FROM country;
```

This will select total of 2 rows, skipping first 1 rows (so it will start from row 2 of the original table)

**Create a view:**

Every full fledged database supports view, so giving you the ability to organize the same data into different logical entities, without the overhead of building and loading new tables.

The syntax used to create a view is very simple:

```
        SQL >CREATE VIEW emp_dep_proj (first_name, last_name, department, proj_name)
AS
SELECT e.first_name, e.last_name, d.department, p.proj_name FROM employee e RIGHT
OUTER JOIN department d ON e.dept_no = d.dept_no LEFT OUTER JOIN employee_project ep
ON e.emp_no = ep.emp_no LEFT OUTER JOIN project p ON ep.proj_id = p.proj_id;
```

You can see the CREATE VIEW clause, followed by the view name and columns, then after the AS clause you can see the query that retrieves the data that will populate the new logical entity. Examining this SELECT you can also notice that Firebird supports aliasing of tables.

The view can be queried exactly as a table, like:

```
        SQL > SELECT * FROM emp_dep_proj;
```

And it can also be joined to other tables or views!

You can check the creation of the view with a:

```
SQL> show views;
        EMP_DEP_PROJ                   PHONE_LIST
```

The newly created view can be deleted with a simple:

```
SQL >DROP VIEW emp_dep_proj;
```

## Loading data from a DDL (*.sql) file

You can also execute a DDL file (a file made of SQL statements) this way:

```
isql -i your_script.sql
```

the script must contain login statements, or you can add them to the command line as described above.

## Spooling data to a file

Sometimes is useful to send the result of a query to an external file, this can be easily accomplished in Firebird with the OUT command, a quick example is the following:

```
SQL> out c:\firelog.txt;
SQL> select * from country;
SQL> out;
```

The first command tells Firebird to send output to a file named c:\firelog.txt, after that a query is executed (no output on screen!!) and then a second OUT command is issued without parameters in order to stop output redirection.

The resulting file will have a rough formatting, this can be improved (mainly for readability or data loading purposes) by modifying the original query:

```
SQL> out c:\firelog.txt;
SQL> select country, ',', currency, ';' from country;
SQL> out;
```

Leading to a result like:

```
COUNTRY                 CURRENCY
=============== ====== ========== ======

USA              ,      Dollar     ;
England          ,      Pound      ;
```

```
Canada        ,    CdnDlr    ;
Switzerland   ,    SFranc    ;
Japan         ,    Yen       ;
Italy         ,    Lira      ;
France        ,    FFranc    ;
Germany       ,    D-Mark    ;
Australia     ,    ADollar   ;
Hong Kong     ,    HKDollar  ;
Netherlands   ,    Guilder   ;
Belgium       ,    BFranc    ;
Austria       ,    Schilling ;
Fiji          ,    FDollar   ;
```

**Backup database**:

Firebird comes with a specific utility to backup and restore a database, called "gbak", which can be found in the ./bin subfolder (along with "isql" and others).

The backup can be performed while users are connected and active, it will simply take a consistent snapshot of data, the syntax is like:

```
>GBAK "localhost:C:\Progra~1\Firebird\Firebird_1_5\examples\employee.fdb" –USER SYSDBA –PAS masterkey "c:\somename.fdbk"
```

Notice that you have to specify the database to be backed up, a valid username and password and a destination file for the backup.

A more advanced syntax, with creation of a log file:

```
>GBAK -v -t -USER SYSDBA -PASS masterkey -y c:\employee_backup.log localhost:C:\Progra~1\Firebird\Firebird_1_5\examples\employee.fdb c:\somename.fbk
```

Options used are:

-v (verbose) verbose output of operations performed

-t (transportable) the backup will be transportable between servers and platforms

-y redirect all output messages to file (filename follows option)

**Restoring database:**

This utility allows also to restore a database, but before going into restore, I have to set some caveats:

1. you must NOT restore to a running database
2. you must NOT allow users to login while restoring a database

This two errors will probably lead to database corruption and loss of data!!!

Otherwise restoring a database is a simple task, I'll show you some practical examples that will guide you in the process:

A simple restore:

```
>GBAK -c -v -user SYSDBA -password masterkey c:\backup_name.fbk
server:/database_path/database_name.fdb
```

Restore to an already existing database:

```
>GBAK -c -r -v -user SYSDBA -password masterkey c:\backup_name.fbk
server:/database_path/database_name.fdb
```

The parameters set mean:

-c(create database) tells gbak to restore a database otherwise it works in backup mode
-r(replace_database) tells gback to restore over an existing database
-v(verbose) tells gbak to use a verbose output

There is also a –o option that is used to restore one table at a time.

**Fixing problems**

Firebird comes with an utility to repair databases, called GFIX.
Corruption can happen for various reasons, you can find a list of them at http://www.ib-aid.com/interbase/firebird/corruption/guide.html.

Gfix accepts some command line parameters:

| | |
|---|---|
| -f | Option used with –v to check all fragments of records |
| -i | Ignore checksum errors |
| -m | Mark damaged records as unavailable, will be deleted at next backup/restore |
| -n | Used with –v for read only validation without error correction |
| -pas | Set password used to connect |
| -user | Set username used to connect |
| -v | Database validation |
| -m | Write mode for database (read or read write) |
| -w(sync/async) | Turn on or off forced writes (sync turns forced writes on FW ON, async to turn off forced writes FW OFF) |

So a tipical command could be:

```
gfix –v –full –user SYSDBA –pass masterkey employee.gdb
```

In order to repair a database, after executing gfix you'll have to backup and restore il, the sequence of commands is something like:

This will show all errors:

```
gfix –v –full employee.gdb –user SYSDBA – pass masterkey
```

This will mark all corrupted rows for deletion:

```
gfix –mend –user SYSDBA –pass masterkey employee.gdb
```

Now you are ready to backup the corrupted database:

```
gbak –b –v -ig –user SYSDBA –pass masterkey employee.gdb employee.gbk
```

And if backup is successful you can now restore it:

```
gbak –c –user SYSDBA –pass masterkey employee.gbk new_employee.gdb
```