Redis Laboratory

This laboratory is dedicated to Redis, an in-memory key-value store that belongs to the NoSQL family.
The main goal for this laboratory is to gain familiarity with the data structures supported by Redis, selecting the one that suits the problem to solve. The interested student can understand the details of Redis by "playing" with the command line interface that interacts with the Redis server. On the official website (www.redis.io) it is possible to find the list of commands supported by Redis, and additional useful documentation.

Installing and running the Redis server

Redis can be downloaded and installed by typing the following commands from the terminal

```
wget http://download.redis.io/releases/redis-2.8.3.tar.gz
tar xzf redis-2.8.3.tar.gz
cd redis-2.8.3
make
```

Then, some useful binaries should be copied to the bin directory

```
sudo cp redis-server /usr/local/bin/
sudo cp redis-cli /usr/local/bin/
```

Once Redis is installed, the server can be launched with

```
redis-server
```

Redis is ready to accept connections on port 6379.

Interacting with the Redis server

In order to interact with the Redis server, a client is necessary. The client will connect to the server on port 6379 and it will send the commands (SET, GET, …) through the connection. The server will reply to each command.

Redis comes with a built-in command line interface client. The client can be launched on a terminal with

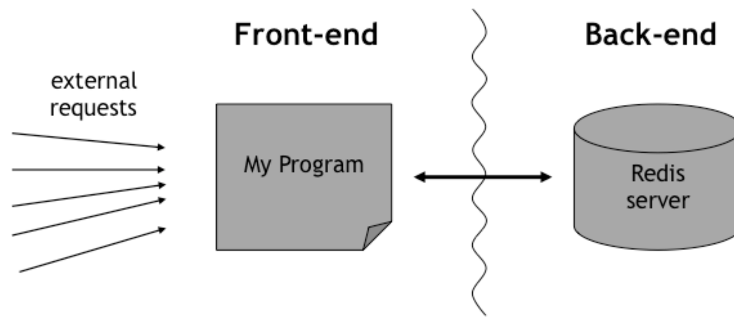
```
redis-cli
```

The client will automatically connect to the instance of Redis server running on the same machine. At this point, is it possible to issue the different commands and observe the Redis server replies.

There exists many clients written in different languages that can be used to interact with the Redis server. The interested student can find the complete list here: www.redis.io/clients
One client in Java is ***Jedis***, tha can be find here: https://github.com/xetorthio/jedis
In practice, these are the steps to follow for using Jedis:

- Download the jar of the Jedis client: https://github.com/downloads/xetorthio/jedis/jedis-2.1.0.jar
- Put the jar in the same directory of your Java program;
- In your Java program, import the Jedis class:
  ```
  import redis.clients.jedis.Jedis;
  ```
- In your Java program, use the Jedis client, e.g.:
  ```
  Jedis jedis = new Jedis("localhost");
  jedis.set("my first key", "This is the value");
  String value = jedis.get("my first key");
  ```
- Compile your Java program:
  ```
  javac -cp jedis-2.1.0.jar myProgram.java
  ```
- Run your Java program (that includes the Redis client):
  ```
  java -cp '.:jedis-2.1.0.jar' myProgram
  ```

Now your program will be the interface (front-end) between the external world and the Redis server (back-end).



Exploring the Redis data structures

We will use the *command line interface* to understand the basic data structures. Please, refer to the official documentation for a complete reference.
Note that other clients (e.g., Jedis) have different syntax, so refer to the API for the details on how to use them.

Strings

A string can be stored with:
```
SET "my key" "my string to store"
```
Multiple strings can be stored with one command:
```
MSET "my key 1" "my string1" "my key 2" "my string 2" …
```
To retrieve a string:
```
GET "my key"
```
If the string is an integer, it can be modified with the command "INCR", "INCRBY", "DECR" and "DECRBY":
```
SET "my counter" 150
INCR "my counter"
INCRBY "my counter" 12
DECR "my counter"
DECRBY "my counter" 10
```

Lists

An object identified by a key can contain a set of values organized in a list. Elements can be added at the beginning:
```
LPUSH "my key" "first element" ["second element" "third element" …]
```
Elements can be added at the end:
```
RPUSH "my key" "last element" ["previous than last element" …]
```
Elements can be inserted in the middle:
```
LINSERT "my key" BEFORE|AFTER pivot value
```
Element can be retrieved from the top:
```
LPOP "my key"
```
Or from the end:
```
RPOP "my key"
```
To know all the elements in a list:
```
LRANGE "my key" 0 -1
```

Sorted Sets

An object identified by a key can contain a set of values organized in an ordered list. The order is given by a score given to the element of the list; the scores can be modified (incremented, decremented). Elements can be added with:
```
ZADD "my key" score element [score element ...]
```
The score can be modified with:
```
ZINCRBY key increment member
```
The set can be queried by referring to the score:
```
ZRANGE key start stop [WITHSCORES]
```

Hashes

An extremely interesting data structure: each key is associated to a value, which is organized in a set of field-value pairs. The fields can be specified one by one:

```
HSET key field1 value
HSET key field2 value
```

Or with a single command:

```
HMSET key field1 value [field2 value …]
```

Given a key, the value of a single field can be retrieved with:

```
HGET key field
```

Given a key, the value of multiple fields can be retrieved with:

```
HMGET key field1 [field2 …]
```

Given a key, all the fields and the values can be retrieved with:

```
HGETALL key
```

If a field contains a numeric value, it can be modified with:

```
HINCRBY key field increment
```

Other useful commands

It is possible to obtain a list of keys that match a pattern with the command "`KEYS`". See the following example:

```
SET user.name Alice
SET user.age 21
SET user.name Bob
SET user.age 22
KEYS user.n*
```

The last command will return all the keys that starts with "user.n…".
Keys can also be sorted: see the "`SORT`" command on the official documentation.
At each key can be associated an expiration time: see the official documentation.

Exercise – Design of a simple URL shortener service

Design a simple program that manages the translation between a long URL and a short one. The program should allow for three distinct operations:

- **Insertion**: the user provides a long URL and obtains a short URL from the system;
- **Query**: the user provides a short URL and obtains a long URL from the system, if it exists;
- **Statistics**: the systems provides a set of statistics on the service.

For the **Insertion** operation, the system:

- takes as an input a string (e.g., http://www.di.univr.it/?ent=persona&id=6412&lang=it);
- asks for identification of the user (e.g., user email);
- checks if the string has already been recorded;
- generates a random string composed by 6 random lower case letters and numbers (e.g., rv4b6n);
- saves the association between the two strings.

For the **Query** operation, the system:

- takes as input a (short) string;
- provides a long URL associated to the string, if it exists, otherwise returns an error;
- registers the number of times that a short string has been asked.

For the **Statistics** operations, the system should provide:

- the number of insertions made by each user;
- the average number of times the short URLs have been asked.

The student can use as a starting point the file "tinyURL.java" provided on the course web page.