# CS 371 Laboratory: Hash Tables

This laboratory will give you some practice using Java’s Hashtable class. (You will want to refer to the Java API documentation during this lab.) The lab has a total of 100 points.

## Preliminaries

# Go to https://github.com/cs301up/soccer. Download it as a zip file.

* Unzip it into your cs371 folder.
* Run Android Studio and browse to the project you just unzipped.
* Post the project to your own GitHub repository.

## Overview

In this laboratory, you will implement a simple database that keeps track of statistics about soccer players. Several classes have already been defined for you:

* **SoccerPlayer** (in package **soccerPlayer**), which defines a soccer player. Its state includes the player's first and last names, the name of his team, his uniform number and certain statistics (e.g., goals scored, fouls committed). You may want to look at the code for this class, but you should not modify it.
* **MainActivity**, which defines a GUI behavior that allows a player to add, remove and query the database. Again, you may want to look at this code, but you should not modify it.
* **SoccerDB** , which is an interface that the **MainActivity** object interacts with. It defines the database operations that must be available. You should not modify this code.
* **SoccerDatabase**, which is a class that implements the **SoccerDB** interface. It presently is dummied up. This is the file that you will modify.

The layout file (and other resource files) are also part of the project. You should not need to modify these.

Your job will be to implement all of the operations for the **SoccerDatabase** class so that the **MainActivity** works correctly when it runs. Its operations will include:

* adding a player to the database
* looking up a player in the database
* removing a player from the database
* modifying a player's statistics (e.g., increment number of goals scored)
* counting the number of players, either on a given team or in the entire database
* find the *n*th player in the database (or on a given team)

The fundamental thing that your **MainActivity** object needs to keep track of is the association between player names and **SoccerPlayer** objects. For this, you should use an instance of Java’s built-in **Hashtable** class. The **MainActivity** will expect you to add, remove and look up data based on the name of a player. Java’s **Hashtable** class is ideal for this. *You will likely find it useful to refer to the Java API documentation for the* **Hashtable** *class.*

All of the public **SoccerDatabase***'s* methods should return an indication of whether the operation was successful:

* Most operations return a boolean. For such methods, returning *true* indicates a successful operation; returning *false* indicates an unsuccessful operation.
* A handful of methods return either an object (where a *null* return value indicates failure) or an integer (where a negative value indicates failure).

Whenever a failure occurs, the GUI indicates this by flashing the screen.

In order to test anything meaningful in this app, you need to enter data for one or more players each time the app is run. To save you some effort, the Populate button calls the **addPlayer** method on a handful of predefined players (listed below), scattering them among several teams.

* John Silver
* Lionel Messi
* Fred Flintstone
* George Jetson
* Christine Sinclair
* Megan Rapinoe
* Stephanie Lopez
* Lindsey Huie
* Mandy Clemens

## Laboratory

### Part 1A: Implement the “add a player to the database” operation

Before implementing this functionality, you should understand the GUI sequence for adding a player:

1. The user fills in the first name, last name, team and uniform number fields in the top row of the GUI. All fields are required, and the uniform number must be positive to be valid.
2. The user presses the “Add Player” button to add a player. This will cause a call to your **addPlayer** method, which is the method you'll be modifying for this checkpoint. The four values from the text fields will be passed as parameters to **addPlayer**.

*Note: if you want to delete a word that you have typed—for example, the name of the team from the currently displayed player—you can long-click on the word. That will highlight the entire word; typing onto the virtual keyboard will replace it rather than appending to it.*

Begin by making these changes to the **SoccerDatabase** class (in package **main** ):

* Declare and instantiate a **Hashtable** instance variable to hold the associations between a player's name and the object that contains all the player's stats. Initially, your hashtable should be empty.

Hashtables require a single object as the key, but we want to do a lookup based on the combination of first and last name. To do this, I suggest that you append the two together (separated by a character or two that would not be part of a name) and use that as a key. For example, you might construct a key that is the first name and last name, separated by the string **“ ## ”.**

* Now, implement **addPlayer** method with this functionality:
  + Check whether the player is already in the database; if so, return *false*, because the player is already in the database.
  + If the player is not already in the database, create a SoccerPlayer object with the appropriate values, put it into the hashtable (using the key as described above), and return *true*.

Because the GUI uses the (presently dummied-up) **getPlayer** method to access the data (e.g., goals scored) for each player, these values will not yet show up in the GUI. For now, about the only thing you can do to test your program is to confirm that an error is reported (by the flashing screen) if an attempt is made to add a player to the database whose name is already there.

### Part 1B: Implement the “find a player in the database” operation.

Before implementing this functionality, you should understand the GUI sequence for looking a player up:

* Fill in the first name and last name fields in the GUI.
* Press the “Find Player” button. This will cause your **getPlayer** method to be called with the first name and last name fields passed as parameters.

Your **getPlayer** method should:

* Create a hashtable key, as above.
* Look up the object for this key in the hashtable.
  + If it is there, return the object.
  + If it is not there, return null.

At this point, you should be able to insert items into the database, and then look them up. *Even if you modify the ‘Team’ and ‘Uniform number’ fields, the ‘Find player’ operation should restore them to the values that were stored*. Data such as goals scored should now also be reported for each player, but you won’t be able to change this information yet.

#### **checkpoint 1 (20 points):** Demonstrate the app’s behavior to your lab instructor/assistant. A toast message is not sufficient for demonstrating that it works properly. You must add the player and find the player to show that both work as expected.

### Part 2: Implement the “remove a player from the database” operation

Before implementing this functionality, you should understand the GUI-sequence for removing a player:

* Fill in the first name and last name fields in the GUI.
* Press the “Remove player” button. This will cause a call to your *removePlayer()* method.

Your **removePlayer** method should

* Verify that the player is in the database.
* If it’s there, remove it from the hashtable and return *true*.
* If it’s not there, return *false*.
  + This should cause a red “flash” under both of the following conditions:
    - If the “Remove Player” button is pressed a second time in succession (or any time there is no player name specified).
    - If the “Remove Player” button is pressed after specifying the name of a player who doesn’t exist or has already been removed.

You should now be able to do combinations of “New player”, “Find Player” and “Remove player”, and see flashing and not flashing at appropriate times.

#### **checkpoint 2 (10 points):**Demonstrate your code to your lab instructor/assistant. A toast message is not sufficient for demonstrating that it works properly. You must demonstrate that the removed player is no longer in the database.

### Part 3: Implement the operations that increment the goals, yellow cards and red cards.

Before implementing this functionality, you should understand the GUI sequence to be used:

* Fill in the first name and last name fields in the GUI.
* Press the appropriate button (e.g., “+goal” to increment the number of goals). If the player is in the database, his data (e.g., uniform number, number of goals, etc.) should be displayed in the GUI. If not, the GUI should flash, and player data other than first and last name should be cleared.

The **bumpGoals** method in the SoccerDatabase class (and similarly for the others) should look up the name in the database, apply the appropriate method (e.g., **bumpGoals** in the SoccerPlayer class) to the resulting object, and return true. If the object was not in the database, then *false* should be returned.

If things are working properly, you should be able to see the number of goals (and such) increment as you press the various “+” buttons. You should also be able to come back to this player after doing other things and see that the statistics are preserved (because they are stored in the **SoccerPlayer** object).

#### **checkpoint 3 (10 points):** Demonstrate your app and show your code to your lab instructor/assistant.

### Part 4: Count the number of players

Before implementing this functionality, you should understand its GUI sequence:

* In the menu near the bottom-center (labeled “Team:”, select the team you want to count. (Or if you want to count all the players, select “### ALL ###”.)
* Press the “count players” button; it will call your **numPlayers** method.
* The number of players is reported in the field immediately to the right of the “Count players” button.

Your **numPlayers** method should:

* If the team name is *null* (meaning that “### ALL ###” was selected), it can simply return the number of associations in the Hashtable.
* If the team name is not null, you will have to iterate through all the keys (or values) in the Hashtable and count the ones that correspond to the desired team.
* In either case, the number of elements should be returned by the method.

#### **Be sure to test that it works for ALL and for selected teams**.

#### **checkpoint 4 (10 points):** Demonstrate your app and show your code to your lab instructor/assistant.

### Part 5: Implement the operation to find the nth player

Before implementing this functionality, you should understand the GUI sequence that accesses it:

* In the menu near the bottom-center (labeled “Team:”, select the team you want to count. (Or if you want to count all the players, select “### ALL ###”.)
* Press the “first player” button to get the first player. That player's name, team, uniform number and statistics should appear in the corresponding windows.
* Press the “next player” button to successively get the remaining players.
* When there a no more players, the screen will flash.
* Pressing the “First player” button will call the **playerIndex** method with an index of 0. Pressing the “Next player” button will subsequently call the **playerIndex** method with increasing indices (i.e., 1, 2, 3, etc).

Your **playerIndex** method should:

* If the given index is greater than the number of players on the team, return null.
* Begin iterating through all the players in the database
* If a particular team name is specified, look for the *n*th player with that team name. Otherwise (i.e., the team name is *null*), look for the *n*th player in the database as a whole.

#### Important note: The players will likely not appear in uniform-number (or alphabetical) order—that’s OK. You can just use the order in which the hashtable gives them (which will seem to be random).

#### **Be sure to test that it works for ALL and for selected teams.**

#### **checkpoint 5 (15 points):** Demonstrate your app and show your code to your lab instructor/assistant.

### Part 6: Modify a player's data using the debugger

Without changing the code, use the debugger to change data in a running program. Do this by following these steps:

1. In Android Studio, locate the **onClick** method of the **FindButtonListener** inner class (**MainActivity**.**java** ). This method will be called whenever the “Find player” button is pressed by the user.
2. Set a breakpoint on the first executable line of the **onClick** method. This will cause a “break” to occur when the “Find Player” button is pressed.
3. Run the program *in debug mode*, and add the following player:

* Name: John Silver
* Team: Waves
* Uniform number: 4

1. Now press the “Find Player” button. Your program should stop at the breakpoint. Step through until the *temp* variable is initialized, which should then contain John Silver’s **SoccerPlayer** object. Open that object in the debugger by clicking on the small triangle in the left margin that corresponds to the variable.
2. Change the uniform number to -37:

* Right-click on the appropriate instance variable; select “Set Value…” from the resulting menu.
* Type the new value into the pop-up and then press the enter *key*.

1. Change the number of goals scored to -7 in a similar manner.
2. Resume execution of the program. If you did this right, the GUI should show both the uniform number and goals scored with their modified values.

Subsequent increments to John’s goal-count should change its value to -6, -5, etc.

Even when you access Mr. Silver’s data through other means (e.g., first-player, next-player), the modified data should persist.

Warning: changing data in a GUI object can have cache/refresh issues, so you might not see the effects right away. Doing so could also corrupt the GUI’s data, causing the program to crash.

#### **checkpoint 6 (10 points):** Show your lab instructor/assistant that you made the modification to John’s uniform number and goals scored demonstrate that you can increment the number of goals to -6 and -5.

### Part 7: Write the data out to a text file so that it can be read back in later.

Before implementing this functionality, you should understand its GUI:

* The user types the filename in the text field marked labeled “File:”.
* The user then presses the “Write” button.

When the user presses the Write button, the *writeData* method is called, with a *File* parameter that tells the name (and path) of the file. Modify this method (currently a stub) so that it actually creates a file, writes the contents of the database (all of the player data) to it, and closes it.

First, you must decide on a file format. For readability, you should use a textual format. A simple format is for each player to store each item on a separate line. (That would be 7 items, if I'm counting correctly—so 7 lines per player) A *PrintWriter* object would be a reasonable one to use for writing the file, as it will allow you to use operations such as *println*.

Implement the *writeData* operation which:

* Opens the given file.
* Writes out all the data in the database and closes the file.
* Returns *true* if the operation was successful, *false* otherwise.

Android’s internal files are difficult to view directly, because it hides them. So that you (and your instructor) can see what is being written, the starter file contains a private method, *logString*, which takes a string parameter, writes a LogCat message, and then returns the string. So instead of writing, for example,

**pw.println(player.getFirstName());**

instead, write

**pw.println(logString(player.getFirstName()));**

It will have the same effect on the file but will allow you to see the data being written by looking at the *LogCat* pane.

The “Read” button has been set up so that it does nothing if the file exists, and flashes if the file does not exist, so you can test that the file has been created by pressing the “Read” button.

#### **checkpoint 7 (10 points):** Demonstrate your app and show your code to your lab instructor/assistant. A toast message is not sufficient for demonstrating that it works properly. You must show the contents of the database printed to the LogCat.

### Part 8: Read the data back in from a text file.

When the user presses the “Read” button, the **readData** method is called on the file named in the “File:” text field. Modify this method (currently a stub that tests for the file existing) so that it actually reads all the player data from the file, recreates the players, and adds them to the database. You can assume that the file is in the format that you wrote to it in the “**writeData**” method above.

**If the name of an existing player conflicts with one that is read in the file, the player read in from the file should replace the existing player’s data.**

In order for new teams to show up properly in the team-selection list, you will need to implement the **getTeams** method (presently dummied up), which should return a **HashSet<String>** of all the team names for the players in the database.

There is no way to directly set the number of goals (etc.) for a *SoccerPlayer* object. You will therefore need to call the *bumpGoals* method (etc.) the appropriate number of times when you read the object in.

You will likely find the *Scanner* class useful.

At this point, you should comment out the line of code in the *logString* method that sends the information to *LogCat*:

* A convenient way to comment out one or more lines is to select them, and then to type Ctrl + / (forward slash).
* A group of already-commented lines can be uncommented with the same keystroke.

Demonstrate your code works for both of the following scenarios:

* 1. Reading values into an empty database.
  2. Overwriting one or more existing players.

#### **checkpoint 8 (15 points):** Demonstrate your app and show your code to your lab instructor/assistant. A toast message is not sufficient for demonstrating that it works properly. You must prove that the contents of the file were actually read in. If the contents of the existing database and the file are the same, how can you prove that the database was loaded with the file contents? How can you prove that if a player exists in the database already, the contents of the file overwrote the existing database content?

### Part 9: Finish up

* Upload your completed work to GitHub.

### Extra Credit (10 points)

### You must complete all of the following and get it checked off before the due date for the lab in order to receive the extra credit.

* 1. Add buttons and text fields for one or more additional player stats (i.e. Shots, Assists, Fouls, Saves).
  2. Update all the necessary places in the code to make them work like the increment buttons previously implemented.
  3. Demonstrate that they all work properly.