Activity Tracker

1.0

Generated by Doxygen 1.8.14

Contents

Chapter 1

COMP-2005 Activity Logger Documentation

This website contains documentation for all source code contained in the *Activity Logger* application. Class and method documentation may be accessed in HTML format using the left-hand side navigation bar, or the search box at the top right-hand side of the page.

For offline viewing, a precompiled PDF of this documentation has been made available `here` Note, however, that this document does *not* contain the full source code which is included in formatted HTML on this website.

More detailed information about contributions, repository branches, and commit history is available by browsing the `GitHub repository` for this project.

Chapter 2

Namespace Index

2.1 Packages

Here are the packages with brief descriptions (if available):

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

Chapter 6

Namespace Documentation

6.1 Package com

Packages

- package activitytracker

6.2 Package com.activitytracker

Classes

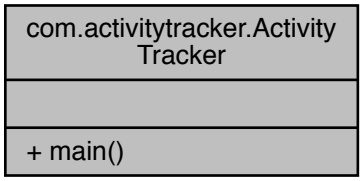
- class ActivityTracker
- class CreateUserWindow
- class DBManager
- class Iteration3Test
- class LoginWindow
- class MainWindow
- class Run
- enum RunAttribute
- class RunStats
- class SecureString
- class User
- enum UserAttribute

Chapter 7

Class Documentation

7.1 com.activitytracker.ActivityTracker Class Reference

Collaboration diagram for com.activitytracker.ActivityTracker:



Static Public Member Functions

- static void main (final String[ ] args)

7.1.1 Detailed Description

The main program class.

Definition at line 28 of file ActivityTracker.java.

7.1.2 Member Function Documentation

7.1.2.1 main()

```
static void com.activitytracker.ActivityTracker.main (
          final String [] args ) [static]
```

The main program entry point.

Definition at line 33 of file ActivityTracker.java.

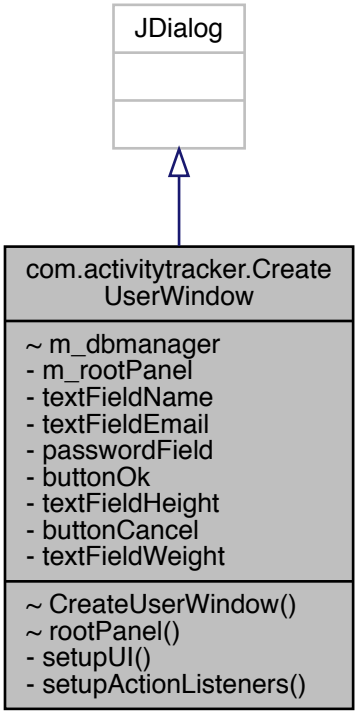
```
33                                                  {
34
35         // Create singleton instance of DBManager
36         DBManager dbManager = new DBManager();
37         if (!dbManager.init("data.db")) {
38             System.err.println("Failed to initialize DBManager");
39             System.exit(1);
40         }
41
42         // Set Look and Feel
43         try {
44             UIManager.setLookAndFeel(new MaterialLookAndFeel());
45         }
46         catch (final UnsupportedLookAndFeelException e) {
47             e.printStackTrace();
48         }
49         // Get desktop resolution of default monitor (in case of multi-monitor setups)
50         final GraphicsDevice gd = GraphicsEnvironment.getLocalGraphicsEnvironment().getDefaultScreenDevice(
     );
51
52         final JFrame frame = new JFrame("Activity Logger");
53
54         final String logoPath = "./assets/logo.png";
55         ImageIcon imgIcon = new ImageIcon(ActivityTracker.class.getResource(logoPath));
56         frame.setIconImage(imgIcon.getImage());
57         frame.setContentPane(new LoginWindow((Void) -> {
58             frame.setContentPane(new MainWindow(dbManager).rootPanel());
59             frame.validate();
60             frame.repaint();
61         }, dbManager).rootPanel());
62         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
63         frame.pack();
64
65         // Set window size to be 1/2 of screen dimensions
66         frame.setSize(gd.getDisplayMode().getWidth() / 2, gd.getDisplayMode().getHeight() / 2);
67         frame.setLocationRelativeTo(null); // Center window
68         frame.setVisible(true);
69     }
```

The documentation for this class was generated from the following file:

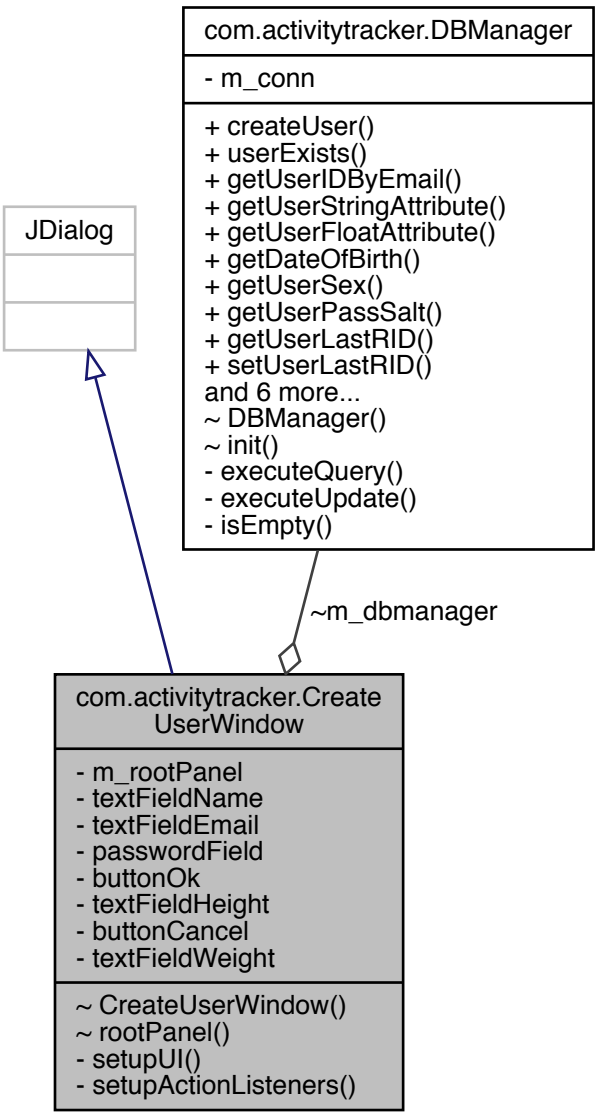
- app/src/com/activitytracker/ActivityTracker.java

7.2 com.activitytracker.CreateUserWindow Class Reference

Inheritance diagram for com.activitytracker.CreateUserWindow:



Collaboration diagram for com.activitytracker.CreateUserWindow:



Package Functions

- CreateUserWindow (DBManager dbmanager)
- JPanel rootPanel ()

Package Attributes

- DBManager m_dbmanager = null

Private Member Functions

- void setupUI ()
- void setupActionListeners ()

Private Attributes

- JPanel m_rootPanel
- JTextField textFieldName
- JTextField textFieldEmail
- JPasswordField passwordField
- JButton buttonOk
- JTextField textFieldHeight
- JButton buttonCancel
- JTextField textFieldWeight

7.2.1 Detailed Description

Definition at line 10 of file CreateUserWindow.java.

7.2.2 Constructor & Destructor Documentation

7.2.2.1 CreateUserWindow()

```
com.activitytracker.CreateUserWindow.CreateUserWindow (
            DBManager dbmanager ) [package]
```

Definition at line 22 of file CreateUserWindow.java.

```
22                                          {
23         m_dbmanager = dbmanager;
24
25         setupUI();
26         setupActionListeners();
27     }
```

7.2.3 Member Function Documentation

7.2.3.1 rootPanel()

```
JPanel com.activitytracker.CreateUserWindow.rootPanel ( )  [package]
```

Definition at line 56 of file CreateUserWindow.java.

```
56                      {
57         return m_rootPanel;
58     }
```

7.2.3.2 setupActionListeners()

```
void com.activitytracker.CreateUserWindow.setupActionListeners ( )  [private]
```

Definition at line 34 of file CreateUserWindow.java.

```
34                                      {
35         buttonOk.addActionListener(new ActionListener() {
36             @Override
37             public void actionPerformed(ActionEvent e) {
38 
39                 if (textFieldName.getText().isEmpty() ||
40                     textFieldEmail.getText().isEmpty() ||
41                     passwordField.getPassword().length == 0) {
42 
43                     return;
44                 }
45             }
46         });
47 
48         buttonCancel.addActionListener(new ActionListener() {
49             @Override
50             public void actionPerformed(ActionEvent e) {
51 
52             }
53         });
54     }
```

7.2.3.3 setupUI()

```
void com.activitytracker.CreateUserWindow.setupUI ( ) [private]
```

Definition at line 29 of file CreateUserWindow.java.

```
29                          {
30         MaterialUIMovement.add(buttonCancel, MaterialColors.GRAY_100);
31         MaterialUIMovement.add(buttonOk, MaterialColors.GRAY_100);
32     }
```

7.2.4 Member Data Documentation

7.2.4.1 buttonCancel

```
JButton com.activitytracker.CreateUserWindow.buttonCancel [private]
```

Definition at line 17 of file CreateUserWindow.java.

7.2.4.2 buttonOk

```
JButton com.activitytracker.CreateUserWindow.buttonOk [private]
```

Definition at line 15 of file CreateUserWindow.java.

7.2.4.3 m_dbmanager

```
DBManager com.activitytracker.CreateUserWindow.m_dbmanager = null [package]
```

Definition at line 20 of file CreateUserWindow.java.

7.2.4.4 m_rootPanel

```
JPanel com.activitytracker.CreateUserWindow.m_rootPanel [private]
```

Definition at line 11 of file CreateUserWindow.java.

7.2.4.5 passwordField

```
JPasswordField com.activitytracker.CreateUserWindow.passwordField [private]
```

Definition at line 14 of file CreateUserWindow.java.

7.2.4.6 textFieldEmail

```
JTextField com.activitytracker.CreateUserWindow.textFieldEmail [private]
```

Definition at line 13 of file CreateUserWindow.java.

7.2.4.7 textFieldHeight

```
JTextField com.activitytracker.CreateUserWindow.textFieldHeight [private]
```

Definition at line 16 of file CreateUserWindow.java.

7.2.4.8 textFieldName

```
JTextField com.activitytracker.CreateUserWindow.textFieldName [private]
```

Definition at line 12 of file CreateUserWindow.java.

7.2.4.9 textFieldWeight

```
JTextField com.activitytracker.CreateUserWindow.textFieldWeight [private]
```

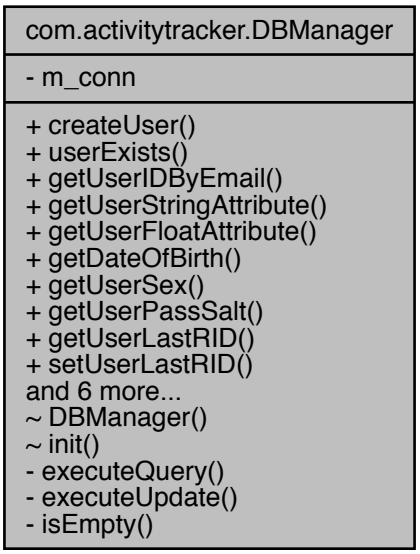
Definition at line 18 of file CreateUserWindow.java.

The documentation for this class was generated from the following file:

- app/src/com/activitytracker/CreateUserWindow.java

7.3 com.activitytracker.DBManager Class Reference

Collaboration diagram for com.activitytracker.DBManager:



Public Member Functions

- void createUser (final String name, final String emailAddress, final java.util.Date dateofBirth, final User.Sex sex, final float height, final float weight, final SecureString securePassword) throws AssertionError

- boolean userExists (final String emailAddress)
- int getUserIDByEmail (final String emailAddress)
- String getUserStringAttribute (final UserAttribute attribute, final int id)
- float getUserFloatAttribute (final UserAttribute attribute, final int id)
- Date getDateOfBirth (final int id)
- User.Sex getUserSex (final int id)
- byte [ ] getUserPassSalt (final int id)
- int getUserLastRID (final int id)
- void setUserLastRID (final int id, final int lastRID)
- int newRun (final int userID, final java.util.Date date, final float duration, final float distance, final float altitudeAscended, final float altitudeDescended)
- void setRun (final int rID, final float duration, final float distance, final float altitudeAscended, final float altitudeDescended)
- float getRunFloatAttribute (final RunAttribute attribute, final int rID)
- java.util.Date getRunDate (final int runID)
- boolean runExists (final int rID)
- Vector< Integer > getRuns (final int userID, final java.util.Date startDate, final java.util.Date endDate)

Package Functions

- DBManager ()
- boolean init (final String dbURL)

Private Member Functions

- ResultSet executeQuery (final String sqlQuery)
- boolean executeUpdate (final String sqlQuery)
- boolean isEmpty ()

Private Attributes

- Connection m_conn = null

7.3.1 Detailed Description

Singleton class for the database. All classes and methods that interact with the database will use a method in this class.

Many times we are faced with the "chicken and egg" problem where we wish to create an object that is populated with information from the database. So the question one faces is, "does the object's constructor query the database (through the DBManager class, of course) for each attribute of the object that it wishes to retrieve, or do we directly interact with a DBManager method which will then return a User or Run object, for example?" We have decided to use the former methodology, with DBManager methods being as general as possible, and often accepting enum types which then are put into a switch to create the specific SQL query we wish to execute. This works best when all data returned is of the same data type (for example, the Workout class will have three float attributes at the time of writing so we use one method with return type of float for returning Workout attributes). This does not work as well when the object requires data of multiple types — for example, the User class. In this case, we have split the DBManager methods into a single method for each attribute being returned.

Polymorphism could theoretically be used here to simply have a return type of Object, however this is not flexible and requires casting *all* returned data to the correct type in the invoking method.

Definition at line 30 of file DBManager.java.

7.3.2 Constructor & Destructor Documentation

7.3.2.1 DBManager()

`com.activitytracker.DBManager.DBManager ( ) [package]`

Creates a new DBManager object.

This should only be called once, from the main program, as DBManager is meant to be a *singleton* class.

This constructor takes no parameters as verification of the SQLite database is done in the init() method of this class, which returns information about whether the initialization was successful or not.

Definition at line 47 of file DBManager.java.

```
47                  {
48     }
```

7.3.3 Member Function Documentation

7.3.3.1 createUser()

```
void com.activitytracker.DBManager.createUser (
            final String name,
            final String emailAddress,
            final java.util.Date dateofBirth,
            final User.Sex sex,
            final float height,
            final float weight,
            final SecureString securePassword ) throws AssertionError
```

Adds a row for a user to the Users table in the SQLite database for the app.

Requires that the database tables exist and are in the correct format. If the user exists in the database this method raises an AssertionError exception.

Parameters

name	User's name.
emailAddress	User's email address; used to authenticate.
dateofBirth	The date the user was born.
sex	The user's sex; is either User.Sex.MALE or User.Sex.FEMALE.
height	Floating point number of the user's height in metres.
weight	Floating point number of the user's weight in kilograms.
securePassword	A SecureString object containing the user's password, encrypted.

Definition at line 64 of file DBManager.java.

```
66                                                                                  {
67
68         if (!userExists(emailAddress)) {
69             String sqlQuery = "INSERT INTO Users (" +
70                     "email_address, " +
71                     "name, " +
72                     "date_of_birth, " +
73                     "sex, " +
74                     "height, " +
75                     "weight," +
76                     "password_hash," +
77                     "password_salt," +
78                     "created_at" +
79                     ") VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)";
```

```
80              byte sexByte = sex.equals(User.Sex.MALE) ? (byte) 1 : (byte) 0;
81              java.sql.Date currentTime = new java.sql.Date(System.currentTimeMillis());
82
83              try {
84                  PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
85                  stmt.setString(1, emailAddress);
86                  stmt.setString(2, name);
87                  stmt.setLong(3, dateofBirth.getTime());
88                  stmt.setByte(4, sexByte);
89                  stmt.setFloat(5, height);
90                  stmt.setFloat(6, weight);
91                  stmt.setString(7, securePassword.toString());
92                  stmt.setBytes(8, securePassword.getSalt());
93                  stmt.setDate(9, currentTime);
94
95                  if (stmt.executeUpdate() != 1) {
96                      System.err.println("User not added to database.");
97                  }
98
99                  stmt.close();
100             }
101             catch (final SQLException e) {
102                 System.err.println(e.getMessage());
103             }
104         }
105         else {
106             throw new AssertionError("User with email address '" + emailAddress + "' already exists.");
107         }
108     }
```

7.3.3.2 executeQuery()

```
ResultSet com.activitytracker.DBManager.executeQuery (
            final String sqlQuery ) [private]
```

A wrapper method for processing *safe* SQL queries.

By safe we mean that the SQL query string is entirely hard-coded in the program source code. In other words, no user input is added. This is an important distinction as the former may leave the application vulnerable to SQL injection.

In such cases, a SQL PreparedStatement should be used.

Parameters

sqlQuery	The SQL code to be executed. Must be a *SELECT* statement.

Returns

This method returns a ResultSet containing the returned row(s) and/or column(s) of the SQL query that was executed.

Definition at line 732 of file DBManager.java.

```
732                                                         {
733         ResultSet res = null;
734
735         try {
736             Statement stmt = m_conn.createStatement();
737             res = stmt.executeQuery(sqlQuery);
738             stmt.close();
739         }
740         catch (final SQLException e) {
741             System.err.println(e.getMessage());
742         }
743
744         return res;
745     }
```

7.3.3.3 executeUpdate()

```
boolean com.activitytracker.DBManager.executeUpdate (
           final String sqlQuery ) [private]
```

A wrapper method for processing *safe* SQL queries.

By safe we mean that the SQL query string is entirely hard-coded in the program source code. In other words, no user input is added. This is an important distinction as the former may leave the application vulnerable to SQL injection.

In such cases, a SQL PreparedStatement should be used.

Parameters

sqlQuery	The SQL code to be executed. Must be an *INSERT* or *UPDATE* statement.

Returns

This method returns a boolean indicating if the query was successful.

Definition at line 760 of file DBManager.java.

```
760                                                         {
761         try {
762             Statement stmt = m_conn.createStatement();
763             stmt.executeUpdate(sqlQuery);
764             stmt.close();
```

```
765         }
766         catch (final SQLException e) {
767             System.err.println(e.getMessage());
768             return false;
769         }
770 
771         return true;
772     }
```

7.3.3.4 getDateOfBirth()

```
Date com.activitytracker.DBManager.getDateOfBirth (
           final int id )
```

Retrieves the user's date of birth (DOB) from the database.

At the time of writing, this method is only being used in the User constructor.

Parameters

id	Unique ID used to associate information in the database to this user.

Returns

This method returns a Date object containing the user's DOB (i.e., year, month, day).

Definition at line 294 of file DBManager.java.

```
294                                             {
295         Date DOB;
296         java.sql.Date DOBResult;
297         ResultSet res;
298         String sqlQuery = "SELECT date_of_birth FROM Users WHERE id=?";
299         try {
300             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
301             stmt.setInt(1, id);
302             res = stmt.executeQuery();
303             DOBResult = res.getDate("date_of_birth");
304 
305             stmt.close();
306         }
307         catch (final SQLException e) {
308             System.err.println(e.getMessage());
309             return null;
310         }
311         DOB = new Date(DOBResult.getYear(), DOBResult.getMonth(), DOBResult.getDay());
312 
313         return DOB;
314     }
```

7.3.3.5 getRunDate()

```
java.util.Date com.activitytracker.DBManager.getRunDate (
            final int runID )
```

Retrieves a run's date from the database using its unique ID

Parameters

runID	Unique ID corresponding to the row in the Runs table that we wish to query.

Definition at line 627 of file DBManager.java.

```
627                                                     {
628         ResultSet res;
629         long fromEpoch;
630         java.util.Date date = null;
631         try {
632             PreparedStatement stmt = m_conn.prepareStatement("SELECT date FROM Runs WHERE id=?");
633             stmt.setInt(1, runID);
634             res = stmt.executeQuery();
635             fromEpoch = res.getLong("date");
636             date = new java.util.Date(fromEpoch);
637         }
638         catch (final SQLException e) {
639             System.err.println(e.getMessage());
640         }
641 
642         return date;
643     }
```

7.3.3.6 getRunFloatAttribute()

```
float com.activitytracker.DBManager.getRunFloatAttribute (
            final RunAttribute attribute,
            final int rID )
```

Retrieves a run's attribute as a floating point number, where applicable, from the database.

This method accepts a RunAttribute enumeration type to specify what attribute it is returning from the database. Only certain attributes are accepted by this method, namely those that are stored as real values. Attributes stored as other data types should use the appropriate accessor method.

Parameters

attribute	The attribute that the method is supposed to query the DB for and return the value of. Note that only certain RunAttribute types are supported in this method. • When *attribute* is RunAttribute.DURATION, the run's duration is returned. • When *attribute* is RunAttribute.DISTANCE, the run's cumulative distance is returned in metres. • When *attribute* is RunAttribute.ALTITUDE_ASCENDED, the run's cumulative altitude climbed is returned in metres • When *attribute* is RunAttribute.ALTITUDE_DESCENDED, the run's cumulative altitude descended is returned in metres
rID	Unique ID corresponding to the row in the Runs table that we wish to query. If such an ID does not exist, *0.0f* will be returned.

Returns

This method returns a float containing run attribute as specified by the *attribute* parameter.

Definition at line 578 of file DBManager.java.

```
578                                                                                 {
579         ResultSet res;
580         PreparedStatement stmt;
581         String sqlQuery, columnLabel;
582         float attrVal = 0.0f;
583         switch (attribute) {
584             case DURATION:
585                 columnLabel = "duration";
586                 sqlQuery = "SELECT " + columnLabel + " FROM Runs WHERE id=?";
587                 break;
588             case DISTANCE:
589                 columnLabel = "distance";
590                 sqlQuery = "SELECT " + columnLabel + " FROM Runs WHERE id=?";
591                 break;
592             case ALTITUDE_ASCENDED:
593                 columnLabel = "altitude_ascended";
594                 sqlQuery = "SELECT " + columnLabel + " FROM Runs WHERE id=?";
595                 break;
596             case ALTITUDE_DESCENDED:
597                 columnLabel = "altitude_descended";
598                 sqlQuery = "SELECT " + columnLabel + " FROM Runs WHERE id=?";
599                 break;
600             default:
601                 return attrVal;
602         }
603         if (runExists(rID)) {
604             try {
605                 stmt = m_conn.prepareStatement(sqlQuery);
606                 stmt.setInt(1, rID);
607                 res = stmt.executeQuery();
```

```
608                 attrVal = res.getFloat(columnLabel);
609             }
610             catch (final SQLException e) {
611                 System.err.println(e.getMessage());
612             }
613         }
614         else {
615             System.err.println("Run " + Integer.toString(rID) + " does not exist. Cannot get " +
      columnLabel + ".");
616         }
617 
618         return attrVal;
619 
620     }
```

7.3.3.7 getRuns()

```
Vector<Integer> com.activitytracker.DBManager.getRuns (
           final int userID,
           final java.util.Date startDate,
           final java.util.Date endDate )
```

Queries the database for all runs by a user with user ID *userID* between *startDate* and *endDate*.

Parameters

userID	The ID of the user whose runs we wish to retrieve.
startDate	The lower bound of the interval we wish to retrieve runs for.
endDate	The uppper bound of the interval we wish to retrieve runs for.

Returns

Returns a vector containing run IDs for each run that meets the search criteria.

Definition at line 691 of file DBManager.java.

```
691 
           {
692         ResultSet res;
693         Vector<Integer> runs = new Vector<>();
694         try {
695             PreparedStatement stmt = m_conn.prepareStatement(
696                     "SELECT id FROM Runs WHERE user_id=? AND date BETWEEN ? AND ?;");
697             stmt.setInt(1, userID);
698             stmt.setLong(2, startDate.getTime());
699             stmt.setLong(3, endDate.getTime());
700 
```

```
701             res = stmt.executeQuery();
702
703             if (res.isClosed())
704                 System.err.println("Result set closed; cannot get any data?");
705
706             while (res.next()) {
707                 runs.add(res.getInt("id"));
708             }
709             stmt.close();
710         }
711         catch (final SQLException e) {
712             System.err.println(e.getMessage());
713         }
714
715         return runs;
716     }
```

7.3.3.8 getUserFloatAttribute()

```
float com.activitytracker.DBManager.getUserFloatAttribute (
            final UserAttribute attribute,
            final int id )
```

Retrieves a user's attribute in floating point format, when applicable, from the database's Users table.

This method accepts a UserAttribute enumeration type to specify what attribute it is returning from the database. Only certain attributes are accepted by this method, namely those that are stored as real values. Attributes stored as other data types should use the appropriate accessor method.

Parameters

attribute	The attribute that the method is supposed to query the DB for and return the value of. Note that only certain UserAttribute types are supported in this method. • When *attribute* is UserAttribute.WEIGHT, this method retrieves the user's weight from the database. • When *attribute* is UserAttribute.HEIGHT, this method retrieves the user's height from the database.
id	Unique ID used to associate information in the database to this user.

Returns

Returns a floating point number corresponding to the UserAttribute passed to the method, for the user specified by *id*.

Definition at line 253 of file DBManager.java.

```
253                                                                                 {
254         float attrVal;
255         ResultSet res;
256         String sqlQuery, columnLabel;
257         switch (attribute) {
258             case WEIGHT:
259                 columnLabel = "weight";
260                 sqlQuery = "SELECT " + columnLabel + " FROM Users WHERE id=?";
261                 break;
262             case HEIGHT:
263                 columnLabel = "height";
264                 sqlQuery = "SELECT " + columnLabel + " FROM Users WHERE id=?";
265                 break;
266             default:
267                 throw new AssertionError("Incorrect UserAttribute enumeration type passed to method.");
268         }
269         try {
270             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
271             stmt.setInt(1, id);
272             res = stmt.executeQuery();
273             attrVal = res.getFloat(columnLabel);
274 
275             stmt.close();
276         }
277         catch (final SQLException e) {
278             System.err.println(e.getMessage());
279             return 0.0f;
280         }
281 
282         return attrVal;
283     }
```

7.3.3.9 getUserIDByEmail()

```
int com.activitytracker.DBManager.getUserIDByEmail (
            final String emailAddress )
```

As we are using the user's email address as their identifying attribute, they will supply this when they log in. Hence, as the database relates everything to the user's unique ID, we must retrieve this ID given the email address.

The logic behind this method relies on the database Users table structure making *email_address* a unique field.

Parameters

emailAddress	The user's email address with which they authenticate.

Returns

This method returns a unique integer corresponding to the row in the database's Users table that stores user information for user with email address *emailAddress*.

Definition at line 151 of file DBManager.java.

```
151                                                         {
152         int id = 0;
153         ResultSet res;
154         String sqlQuery = "SELECT id FROM Users WHERE `email_address`=?";
155         try {
156             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
157             stmt.setString(1, emailAddress);
158             res =  stmt.executeQuery();
159             id = res.getInt("id");
160 
161             stmt.close();
162         }
163         catch (final SQLException e) {
164             System.err.println(e.getMessage());
165         }
166 
167         return id;
168     }
```

7.3.3.10 getUserLastRID()

```
int com.activitytracker.DBManager.getUserLastRID (
            final int id )
```

Retrieves the last workout ID that the user added as an integer from the database.

This is used because of the format in which the data is supplied. As the only way to denote a new workout is by recieving (0, 0, 0) in the input file, if the input is *not* (0, 0, 0), we need to update the previously added workout with the latest line. Hence we need some way of storing an identifier for this workout. As this is unique to each user, we have chosen to store this in the Users table of the database.

Parameters

id	Unique ID used to associate information in the database to this user.

Returns

An integer corresponding to the last row in the Workouts table that the user created.

Definition at line 391 of file DBManager.java.

```
391                                                 {
392         int rID = 0;
393         ResultSet res;
394         String columnLabel = "last_run";
395         String sqlQuery = "SELECT " + columnLabel + " FROM Users WHERE id=?";
396         try {
397             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
398             stmt.setInt(1, id);
399             res = stmt.executeQuery();
400             rID = res.getInt(columnLabel);
401             stmt.close();
402         }
403         catch (final SQLException e) {
404             System.err.println(e.getMessage());
405         }
406 
407         return rID;
408     }
```

7.3.3.11 getUserPassSalt()

```
byte [] com.activitytracker.DBManager.getUserPassSalt (
            final int id )
```

Retrieves a byte array containing the salt used to encrypt the user's password from the database.

This is necessary because to compare a candidate password supplied by a user to a known (encrypted) password stored in the database, we must encrypt the new candidate password using the same salt as was originally used.

Parameters

id	Unique ID used to associate information in the database to this user.

Returns

This method returns a byte array containing the user's password encryption salt.

Definition at line 362 of file DBManager.java.

```
362                                                 {
363         byte[] passSalt;
364         ResultSet res;
365         String sqlQuery = "SELECT password_salt FROM Users WHERE id=?";
```

```
366         try {
367             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
368             stmt.setInt(1, id);
369             res = stmt.executeQuery();
370             passSalt = res.getBytes("password_salt");
371             stmt.close();
372         }
373         catch (final SQLException e) {
374             System.err.println(e.getMessage());
375             return null;
376         }
377         return passSalt;
378     }
```

7.3.3.12 getUserSex()

```
User.Sex com.activitytracker.DBManager.getUserSex (
            final int id )
```

Retrieves the user's gender from the database.

We have chosen to represent gender in the SQLite database with the data type BIT(1), where 1 denotes male and 0 denotes female. Hence, if the database contains 1 this method returns User.Sex.MALE and if the database contains 0 then this method returns User.Sex.FEMALE.

At the time of writing, this method is only being used in the User constructor.

Parameters

id	Unique ID used to associate information in the database to this user.

Returns

This method returns a User.Sex enumeration type corresponding to the user's gender.

Definition at line 329 of file DBManager.java.

```
329                                             {
330         byte sex;
331         ResultSet res;
332         String sqlQuery = "SELECT sex FROM Users WHERE id=?";
333         try {
334             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
335             stmt.setInt(1, id);
336             res = stmt.executeQuery();
337             sex = res.getByte("sex");
338
```

```
339             stmt.close();
340         }
341         catch (final SQLException e) {
342             System.err.println(e.getMessage());
343             return null;
344         }
345 
346         if (sex == (byte) 1)
347             return User.Sex.MALE;
348         else
349             return User.Sex.FEMALE;
350     }
```

7.3.3.13 getUserStringAttribute()

```
String com.activitytracker.DBManager.getUserStringAttribute (
            final UserAttribute attribute,
            final int id )
```

This method retrieves a string, varchar, text, or char field, when applicable, from the database's Users table.

This method accepts a UserAttribute enumeration type to specify what attribute it is returning from the database. Only certain attributes are accepted by this method, namely those that are stored as string-like values. Attributes stored as other data types should use the appropriate accessor method.

Parameters

attribute	The attribute that the method is supposed to query the DB for and return the value of. Note that only certain UserAttribute types are supported in this method. • When *attribute* is UserAttribute.PASSWORD, this method retrieves the user's encrypted password from the database. Typically this will be used in the following sequence of calls: 1. User attempts to authenticate with email and password 2. Their unique ID is retrieved from the database using DBManager::getUserIDByEmail() 3. Their ID is used to retrieve the hash of their password (i.e., this method is called) 4. The returned string from this method is compared a SecureString generated from the candidate password supplied by the user when authenticating. • When *attribute* is UserAttribute.NAME, this method retrieves the user's full name from the database (e.g., "John Doe"). • When *attribute* is UserAttribute.EMAIL_ADDRESS, this method retrieves the user's email address from the database. Note that this is likely somewhat redundant as the user will always be required to authenticate by providing their email address and hence it will already be available to the User constructor, which is likely what is invoking this method.
id	Unique ID used to associate information in the database to this user.

Returns

This method returns a string containing attribute specified by the *attribute* parameter for the user specified by the *id* parameter.

Definition at line 199 of file DBManager.java.

```
199                                                                                 {
200         String name;
201         ResultSet res;
202         String sqlQuery, columnLabel;
203         switch (attribute) {
204             case PASSWORD:
205                 columnLabel = "password_hash";
206                 sqlQuery = "SELECT " + columnLabel + " FROM Users WHERE id=?";
207                 break;
208             case NAME:
209                 columnLabel = "name";
210                 sqlQuery = "SELECT " + columnLabel + " FROM Users WHERE id=?";
```

```
211                 break;
212             case EMAIL_ADDRESS:
213                 columnLabel = "email_address";
214                 sqlQuery = "SELECT " + columnLabel + " FROM Users WHERE id=?";
215                 break;
216             default:
217                 throw new AssertionError("Incorrect UserAttribute enumeration type passed to method.");
218         }
219         try {
220             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
221             stmt.setInt(1, id);
222             res = stmt.executeQuery();
223             name = res.getString(columnLabel);
224 
225             stmt.close();
226         }
227         catch (final SQLException e) {
228             System.err.println(e.getMessage());
229             return null;
230         }
231 
232         return name;
233     }
```

7.3.3.14 init()

```
boolean com.activitytracker.DBManager.init (
           final String dbURL ) [package]
```

Initializes a connection to the SQLite database.

As no work is done in the DBManager() constructor, this method should be called immediately after creating the single instance of DBManager that the application is to use.

This method will attempt to connect to the database file specified by the *dbURL* parameter, creating the file and all required tables if it/they do not exist. You are encouraged to view the source code of this method for more information about the database schema used.

If all of the above is successful, the method returns True. Otherwise, False is returned.

Parameters

dbURL	A file system path to the SQLite database file.

Returns

This method returns True if the database can be initialized, or False otherwise.

Definition at line 814 of file DBManager.java.

```
814                                     {
815         try {
816             m_conn = DriverManager.getConnection("jdbc:sqlite:" + dbURL);
817         }
818         catch (final SQLException e) {
819             System.err.println(e.getMessage());
820             return false;
821         }
822         System.out.println("Opened database successfully.");
823 
824         if (isEmpty()) {
825             System.out.println("Creating tables...");
826 
827             // Create users table
828             String sqlQuery = "CREATE TABLE USERS (" +
829                     "    id            INTEGER PRIMARY KEY ASC AUTOINCREMENT NOT NULL," +
830                     "    email_address STRING  NOT NULL UNIQUE ON CONFLICT FAIL," +
831                     "    name          STRING  NOT NULL," +
832                     "    date_of_birth DATETIME    NOT NULL," +
833                     "    sex           BIT(1)  NOT NULL," +
834                     "    height        REAL    NOT NULL," +
835                     "    weight        REAL    NOT NULL," +
836                     "    password_hash STRING  NOT NULL," +
837                     "    password_salt BLOB    NOT NULL," +
838                     "    last_run      INTEGER NOT NULL DEFAULT 0," +
839                     "    created_at    DATETIME    NOT NULL" +
840                     ")";
841 
842             if (!executeUpdate(sqlQuery)) {
843                 return false;
844             }
845 
846             // Create workouts table
847             sqlQuery = "CREATE TABLE RUNS (" +
848                     "    id                  INTEGER PRIMARY KEY ASC AUTOINCREMENT NOT NULL," +
849                     "    user_id             INTEGER NOT NULL REFERENCES USERS (id)," +
850                     "    date                DATETIME    NOT NULL," +
851                     "    duration            REAL    NOT NULL," + // seconds
852                     "    distance            REAL    NOT NULL," + // metres
853                     "    altitude_ascended   REAL    NOT NULL," + // metres
854                     "    altitude_descended  REAL    NOT NULL" + // metres
855                     ")";
856 
857             if (!executeUpdate(sqlQuery)) {
858                 return false;
859             }
860 
861         }
862 
863         return true;
864     }
```

7.3.3.15 isEmpty()

`boolean com.activitytracker.DBManager.isEmpty ( ) [private]`

Returns a boolean value depending on whether or not the database is populated.

This is done by retrieving tables in the database and checking if this iterator has a next(). If not then there are no tables in the database and we consider it to be empty.

Returns

Returns True if there are tables in the database, False otherwise.

Definition at line 782 of file DBManager.java.

```
782                                 {
783
784         try {
785             final DatabaseMetaData dbmd = m_conn.getMetaData();
786             final String[] types = {"TABLE"};
787             final ResultSet rs = dbmd.getTables(null, null, "%", types);
788
789             return !rs.next();
790         }
791         catch (final SQLException e) {
792             System.err.println(e.getMessage());
793             return true;
794         }
795
796     }
```

7.3.3.16 newRun()

```
int com.activitytracker.DBManager.newRun (
            final int userID,
            final java.util.Date date,
            final float duration,
            final float distance,
            final float altitudeAscended,
            final float altitudeDescended )
```

Creates a new row in the Runs table with the attributes provided as parameters.

In particular, this method will be called when Run::newRunDataPoint() receives (0, 0, 0) for (*duration*, *distance*, *altitude*).

Parameters

userID	Unique ID used to associate information in the database to this user.
date	Date that the run was completed.
duration	Duration of the run in seconds.
distance	Distance ran in metres.
altitudeAscended	Cumulative altitude climbed in metres.
altitudeDescended	Cumulative altitude descended in metres.

Returns

Returns a unique integer corresponding to the new row in the SQLite Workouts table by which the new entry can be identified.

Definition at line 453 of file DBManager.java.

```
454                                                                                 {
455
456         int rID = 0;
457         ResultSet res;
458         String sqlInsertQuery = "INSERT INTO Runs (" +
459                 "user_id," +
460                 "date," +
461                 "duration," +
462                 "distance," +
463                 "altitude_ascended," +
464                 "altitude_descended" +
465                 ") VALUES (?, ?, ?, ?, ?, ?)";
466         String sqlSelectQuery = "SELECT id FROM Runs WHERE " +
467                 "user_id=? AND " +
468                 "date=? AND " +
469                 "duration=? AND " +
470                 "distance=? AND " +
471                 "altitude_ascended=? AND " +
472                 "altitude_descended=?";
473
474         try {
475             PreparedStatement stmt = m_conn.prepareStatement(sqlInsertQuery);
476             stmt.setInt(1, userID);
477             stmt.setLong(2, date.getTime());
478             stmt.setFloat(3, duration);
479             stmt.setFloat(4, distance);
480             stmt.setFloat(5, altitudeAscended);
481             stmt.setFloat(6, altitudeDescended);
482
483             if (stmt.executeUpdate() != 1) {
484                 System.err.println("Run not added to database.");
485             }
486
487             stmt.close();
488
489             // Pass back in the stuff we just created to get the right row ID
490             // Look at a better way of doing this with OUTPUT clause of INPUT statement
491             stmt = m_conn.prepareStatement(sqlSelectQuery);
492             stmt.setInt(1, userID);
493             stmt.setLong(2, date.getTime());
494             stmt.setFloat(3, duration);
495             stmt.setFloat(4, distance);
496             stmt.setFloat(5, altitudeAscended);
497             stmt.setFloat(6, altitudeDescended);
498
499             res = stmt.executeQuery();
500             rID = res.getInt("id");
501         }
502         catch (final SQLException e) {
503             System.err.println(e.getMessage());
504         }
505
506         return rID;
507     }
```

7.3.3.17 runExists()

```
boolean com.activitytracker.DBManager.runExists (
          final int rID )
```

Determines if a given run ID exists in the database.

Parameters

rID	Unique ID corresponding to the row in the Runs table that we wish to check exists.

Returns

This method returns True if the run row with ID *WOID* exists in the database, or False otherwise.

Definition at line 652 of file DBManager.java.

```
652                                             {
653         ResultSet res;
654         String sqlQuery = "SELECT COUNT(*) as count FROM Runs WHERE id=?";
655         boolean exists = false;
656         try {
657             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
658             stmt.setInt(1, rID);
659             res = stmt.executeQuery();
660             switch (res.getInt("count")) {
661                 case 0:
662                     exists = false;
663                     break;
664                 case 1:
665                     exists = true;
666                     break;
667                 default:
668                     exists = true;
669                     System.err.println("More than one run for ID " +
670                             Integer.toString(rID) + ". Something isn't right.");
671                     break;
672             }
673 
674         }
675         catch (final SQLException e) {
676             System.err.println(e.getMessage());
677         }
678 
679         return exists;
680     }
```

7.3.3.18 setRun()

```
void com.activitytracker.DBManager.setRun (
            final int rID,
            final float duration,
            final float distance,
            final float altitudeAscended,
            final float altitudeDescended )
```

Updates a run entry in the database as new information becomes available from the input file.

In particular, this method is called when Run::newRunDataPoint() receives non-(0, 0, 0) input for (*duration*, *distance*, *altitude*).

This method will not be called directly by the application, rather it is called from Run::newRunDataPoint(). Hence that method will take care of adding/subtracting to/from the current stored values for *duration*, *distance*, and *altitude* — here we just take the input and put it in the database.

Parameters

rID	Unique ID used to identify a run in the database.
duration	The number of seconds the user's run lasted.
distance	The cumulative number of metres the user ran.
altitudeAscended	The cumulative number of metres the user climbed.
altitudeDescended	The cumulative number of metres the user descended.

Definition at line 526 of file DBManager.java.

```
527                                                                                                 {
528         String sqlQuery = "UPDATE Runs SET " +
529                 "duration = ?, " +
530                 "distance = ?, " +
531                 "altitude_ascended=?, " +
532                 "altitude_descended=? " +
533                 "WHERE id=? ";
534         try {
535             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
536             stmt.setFloat(1, duration);
537             stmt.setFloat(2, distance);
538             stmt.setFloat(3, altitudeAscended);
539             stmt.setFloat(4, altitudeDescended);
540             stmt.setInt(5, rID);
541 
542             int result = stmt.executeUpdate();
543             System.err.println(Integer.toString(result) + " rows updated in setRun().");
544             if (result != 1) {
545                 System.err.println("Run not updated in database.");
546             }
547 
548             stmt.close();
549         }
```

```
550         catch (final SQLException e) {
551             System.err.println(e.getMessage());
552         }
553 
554     }
```

7.3.3.19 setUserLastRID()

```
void com.activitytracker.DBManager.setUserLastRID (
            final int id,
            final int lastRID )
```

Updates a user's last run ID in the database.

This method will be used to update the run that a particular user last created. This is used when creating new run as the format of the input file requires that we maintain a record of what run we must update if the next line in the file is *not* (0, 0, 0).

See getUserLastRID() for more information on the user of the *last_run* field in the database.

Parameters

id	Unique ID used to associate information in the database to this user.
lastRID	Integer corresponding to the last row in the Workouts table that the user with ID *id* created.

Definition at line 422 of file DBManager.java.

```
422                                                         {
423         String sqlQuery = "UPDATE Users SET last_run=? WHERE id=?";
424         try {
425             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
426             stmt.setInt(1, lastRID);
427             stmt.setInt(2, id);
428             if (stmt.executeUpdate() != 1) {
429                 System.err.println("User's last run was not updated correctly.");
430             }
431         }
432         catch (final SQLException e) {
433             System.err.println(e.getMessage());
434         }
435     }
```

7.3.3.20 userExists()

```
boolean com.activitytracker.DBManager.userExists (
          final String emailAddress )
```

The DBManager::userExists() method is designed to facilitate the user experience (UX) design choice of users creating one account to the app and logging in with an existing account for future use. This maintains saved (persistent) data and helps enforce the unique constraint placed on the *email_address* field in the database (again, as users are authenticating using their email address as a user name to identify themselves).

Parameters

emailAddress	The user's email address for which we are checking existence. We use email address here because this is what the user uses to log in to the app.

Returns

True if the user exists in the database, false otherwise.

Definition at line 120 of file DBManager.java.

```
120                                                         {
121         String sqlQuery = "SELECT COUNT(*) AS count FROM Users WHERE `email_address`=?";
122         boolean exists = false;
123 
124         try {
125             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
126             stmt.setString(1, emailAddress);
127             ResultSet res = stmt.executeQuery();
128             exists = res.getInt("count") > 0;
129 
130             stmt.close();
131         }
132         catch (final SQLException e) {
133             System.err.println(e.getMessage());
134         }
135 
136         return exists;
137     }
```

7.3.4 Member Data Documentation

7.3.4.1 m_conn

```
Connection com.activitytracker.DBManager.m_conn = null [private]
```

The *m_conn* variable in the DBManager class is initially assigned the value of *null*.

When DBManager::init() is invoked, it is made to be the connection to the database and is subsequently used each time a new SQL statement is created.

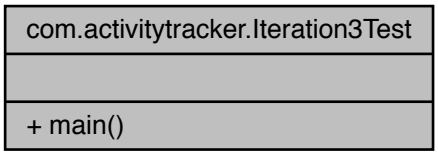
Definition at line 37 of file DBManager.java.

The documentation for this class was generated from the following file:

- app/src/com/activitytracker/DBManager.java

7.4 com.activitytracker.Iteration3Test Class Reference

Collaboration diagram for com.activitytracker.Iteration3Test:



Static Public Member Functions

- static void main (String[] args)

7.4.1 Detailed Description

Definition at line 12 of file Iteration3Test.java.

7.4.2 Member Function Documentation

7.4.2.1 main()

```
static void com.activitytracker.Iteration3Test.main (
          String [] args ) [static]
```

Definition at line 14 of file Iteration3Test.java.

```
14                                          {
15
16          // Iteration 1 begins here
17
18          User john = null;
19
20          DBManager dbManager = new DBManager();
21          if (!dbManager.init("data.db")) {
22              System.err.println("Failed to initialize DBManager");
23              System.exit(1);
24          }
25
26          System.out.println("Attempting to create user...");
27
28          if (!dbManager.userExists("jdoe@mac.com")) {
29              Date dob = null;
30              DateFormat sourceFormat = new SimpleDateFormat("dd-MM-yyyy");
31              try {
32                  dob = sourceFormat.parse("03-04-1978");
33              } catch (final ParseException e) {
34                  System.err.println(e.getMessage());
35              }
36
37              User.createUser(
38                      dbManager,
39                      "John Doe",
40                      "jdoe@mac.com",
41                      dob,
42                      User.Sex.MALE,
43                      1.6764f,
44                      54.4310844f,
45                      "My Very Secure Password"
46              );
47          }
48          else
49              System.out.println("User already exists.");
50
51
52
53          if (dbManager.userExists("jdoe@mac.com"))
54              System.out.println("John Doe was created!");
55          else
56              System.out.println("User was NOT created.");
57
58
59          System.out.println("Testing incorrect password...");
60
61          try {
```

```
62              john = new User(dbManager,"jdoe@mac.com", "Some Incorrect Password");
63          }
64          catch (final AuthenticationException e) {
65              System.out.println("Incorrect password used; authentication failed.");
66          }
67
68          System.out.println("Authenticating user...");
69
70          try {
71              john = new User(dbManager,"jdoe@mac.com", "My Very Secure Password");
72          }
73          catch (final AuthenticationException e) {
74              System.out.println("Test failed; user could not be authenticated.");
75          }
76
77          // Iteration 1 ended here
78
79          // Iteration 2 begins here
80
81          if (john !=  null) {
82              Date today = new Date();
83              try {
84                  Run.bulkImport(dbManager, john, "/Users/jacobhouse/Google Drive File Stream/My
        Drive/Documents/Courses/Computer Science/COMP-2005 Software Engineering/Final
        Project/comp2005-activity-tracker/app/InputWO.csv");
85              }
86              catch (final IOException e) {
87                  System.err.println(e.getMessage());
88              }
89          }
90          else {
91              System.out.println("John is null. Cannot execute phase 2.");
92          }
93
94          // Iteration 2 ends here
95          // Iteration 3 begins here
96
97          Date date = null;
98          DateFormat sourceFormat = new SimpleDateFormat("dd-MM-yyyy");
99
100         try {
101             date = sourceFormat.parse("01-01-2018");
102         }
103         catch (final ParseException e) {
104             System.err.println(e.getMessage());
105         }
106
107         Vector<Run> runs = Run.getRuns(dbManager, john, date, new Date());
108
109         if (runs == null) {
110             System.out.println("Runs is null.");
111         } else if (runs.size() == 0)
112             System.err.println("No runs in vector.");
113         else {
114             for (Run run : runs) {
115                 System.out.println("Retrieved run with ID " + Integer.toString(run.getID()));
116                 System.out.println("Run duration: " + Float.toString(run.getDuration()));
117                 System.out.println("Run distance: " + Float.toString(run.getDistance()));
118                 System.out.println("Run speed: " + Float.toString(run.getSpeed()));
119                 System.out.println("Run altitude ascended: " + run.getAltitudeAscended());
120                 System.out.println("Run altitude descended: " + run.getAltitudeDescended());
121                 System.out.println("Run date: " + run.getRunDate().toString());
122                 System.out.println();
123             }
124         }
125
126         RunStats stats = new RunStats(runs);
127         if (!stats.isEmpty()) {
```

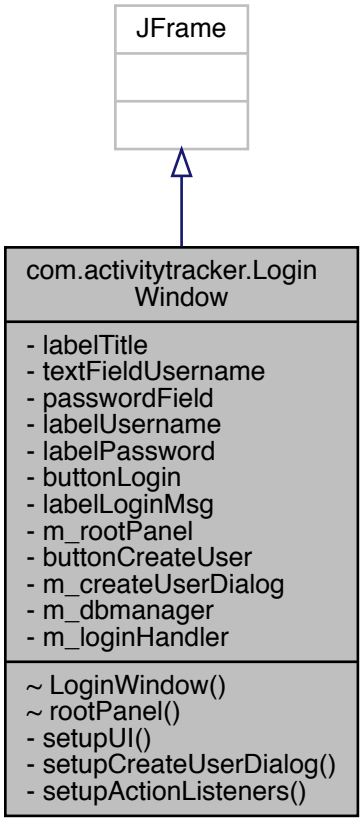
```
128             System.out.println("Average run speed: " + stats.getMeanSpeed());
129             System.out.println("Average run duration: " + stats.getMeanDuration());
130             System.out.println("Average run distance: " + stats.getMeanDistance());
131             System.out.println("Total run distance: " + stats.getTotalDistance());
132             System.out.println("Average altitude gained: " + stats.getMeanAltitudeAscended());
133             System.out.println("Total altitude gained: " + stats.getTotalAltitudeAscended());
134             System.out.println("Average altitude lost: " + stats.getMeanAltitudeDescended());
135             System.out.println("Total altitude lost: " + stats.getTotalAltitudeDescended());
136         }
137         else {
138             System.out.println("RunStats is empty. No stats to show.");
139         }
140 
141         // Iteration 3 ends here
142 
143     }
```

The documentation for this class was generated from the following file:

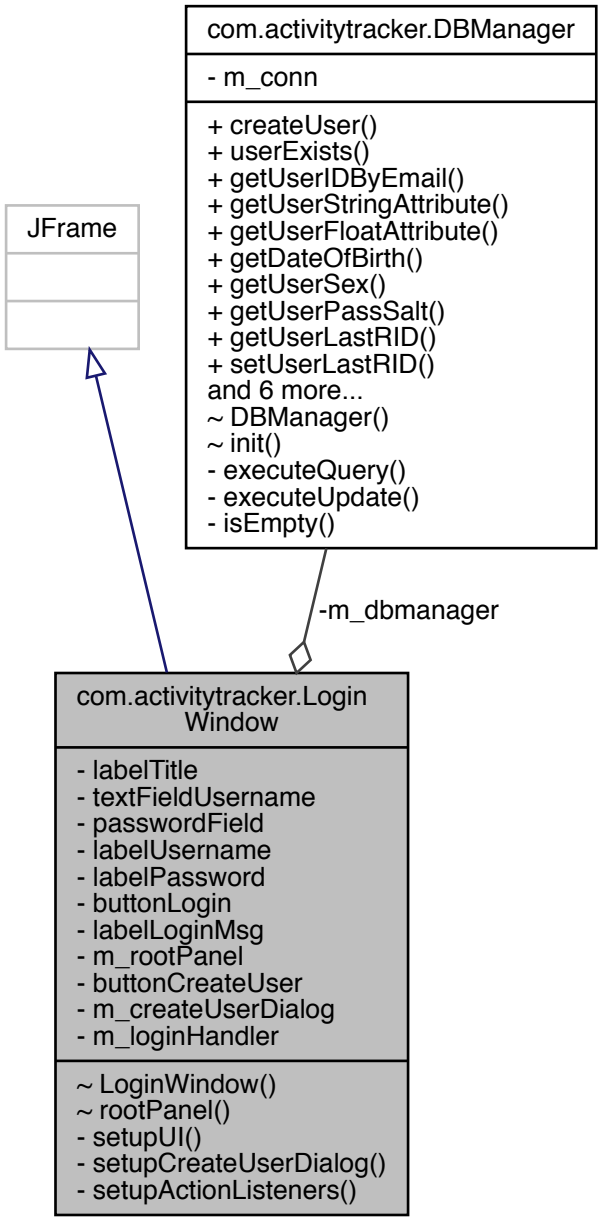
- app/src/com/activitytracker/Iteration3Test.java

7.5 com.activitytracker.LoginWindow Class Reference

Inheritance diagram for com.activitytracker.LoginWindow:



Collaboration diagram for com.activitytracker.LoginWindow:



Package Functions

- LoginWindow (java.util.function.Consumer< Void > loginHandler, DBManager dbmanager)

- JPanel rootPanel ()

Private Member Functions

- void setupUI ()
- void setupCreateUserDialog ()
- void setupActionListeners ()

Private Attributes

- JLabel labelTitle
- JTextField textFieldUsername
- JPasswordField passwordField
- JLabel labelUsername
- JLabel labelPassword
- JButton buttonLogin
- JLabel labelLoginMsg
- JPanel m_rootPanel
- JButton buttonCreateUser
- JDialog m_createUserDialog = null
- DBManager m_dbmanager = null
- java.util.function.Consumer< Void > m_loginHandler

7.5.1 Detailed Description

Definition at line 11 of file LoginWindow.java.

7.5.2 Constructor & Destructor Documentation

7.5.2.1 LoginWindow()

```
com.activitytracker.LoginWindow.LoginWindow (
            java.util.function.Consumer< Void > loginHandler,
            DBManager dbmanager ) [package]
```

Definition at line 28 of file LoginWindow.java.

```
28                                                                                  {
29          m_loginHandler = loginHandler;
30
31          setupUI();
32          setupCreateUserDialog();
33          setupActionListeners();
34      }
```

7.5.3 Member Function Documentation

7.5.3.1 rootPanel()

```
JPanel com.activitytracker.LoginWindow.rootPanel ( ) [package]
```

Definition at line 87 of file LoginWindow.java.

```
87                          {
88          return m_rootPanel;
89      }
```

7.5.3.2 setupActionListeners()

`void com.activitytracker.LoginWindow.setupActionListeners ( ) [private]`

Definition at line 56 of file LoginWindow.java.

```
56                                      {
57
58         // Login button
59         buttonLogin.addActionListener(new ActionListener() {
60             @Override
61             public void actionPerformed(ActionEvent e) {
62
63                 // Do nothing if login fields are empty
64                 if (textFieldUsername.getText().isEmpty() ||
       passwordField.getPassword().length == 0) {
65                     return;
66                 }
67
68                 // Change to verifyLogin()
69                 if (true) {
70                     m_loginHandler.accept(null);
71                     return;
72                 }
73
74                 // Display error message
75             }
76         });
77
78         // Create user button
79         buttonCreateUser.addActionListener(new ActionListener() {
80             @Override
81             public void actionPerformed(ActionEvent e) {
82                 m_createUserDialog.setVisible(true);
83             }
84         });
85     }
```

7.5.3.3 setupCreateUserDialog()

`void com.activitytracker.LoginWindow.setupCreateUserDialog ( ) [private]`

Definition at line 43 of file LoginWindow.java.

```
43                                       {
44
45         // Get desktop resolution of default monitor (in case of multi-monitor setups)
46         final GraphicsDevice gd = GraphicsEnvironment.getLocalGraphicsEnvironment().getDefaultScreenDevice(
       );
47
48         m_createUserDialog = new JDialog(this, "Activity Logger | Create User", true);
49         m_createUserDialog.setContentPane(new CreateUserWindow(
       m_dbmanager).rootPanel());
50         m_createUserDialog.pack();
51         // Set window size to be 1/2 of screen dimensions
52         m_createUserDialog.setSize(gd.getDisplayMode().getWidth() / 2, gd.getDisplayMode(
       ).getHeight() / 2);
53         m_createUserDialog.setLocationRelativeTo(this); // Center window
54     }
```

7.5.3.4 setupUI()

```
void com.activitytracker.LoginWindow.setupUI ( ) [private]
```

Definition at line 36 of file LoginWindow.java.

```
36                          {
37         MaterialUIMovement.add(buttonLogin, MaterialColors.GRAY_100);
38         MaterialUIMovement.add(buttonCreateUser, MaterialColors.GRAY_100);
39 
40         labelLoginMsg.setVisible(false);
41     }
```

7.5.4 Member Data Documentation

7.5.4.1 buttonCreateUser

```
JButton com.activitytracker.LoginWindow.buttonCreateUser [private]
```

Definition at line 20 of file LoginWindow.java.

7.5.4.2 buttonLogin

```
JButton com.activitytracker.LoginWindow.buttonLogin [private]
```

Definition at line 17 of file LoginWindow.java.

7.5.4.3 labelLoginMsg

```
JLabel com.activitytracker.LoginWindow.labelLoginMsg [private]
```

Definition at line 18 of file LoginWindow.java.

7.5.4.4 labelPassword

```
JLabel com.activitytracker.LoginWindow.labelPassword [private]
```

Definition at line 16 of file LoginWindow.java.

7.5.4.5 labelTitle

```
JLabel com.activitytracker.LoginWindow.labelTitle [private]
```

Definition at line 12 of file LoginWindow.java.

7.5.4.6 labelUsername

```
JLabel com.activitytracker.LoginWindow.labelUsername [private]
```

Definition at line 15 of file LoginWindow.java.

7.5.4.7 m_createUserDialog

```
JDialog com.activitytracker.LoginWindow.m_createUserDialog = null [private]
```

Definition at line 22 of file LoginWindow.java.

7.5.4.8 m_dbmanager

```
DBManager com.activitytracker.LoginWindow.m_dbmanager = null [private]
```

Definition at line 24 of file LoginWindow.java.

7.5.4.9 m_loginHandler

```
java.util.function.Consumer<Void> com.activitytracker.LoginWindow.m_login↩
Handler [private]
```

Definition at line 26 of file LoginWindow.java.

7.5.4.10 m_rootPanel

```
JPanel com.activitytracker.LoginWindow.m_rootPanel [private]
```

Definition at line 19 of file LoginWindow.java.

7.5.4.11 passwordField

```
JPasswordField com.activitytracker.LoginWindow.passwordField [private]
```

Definition at line 14 of file LoginWindow.java.

7.5.4.12 textFieldUsername

```
JTextField com.activitytracker.LoginWindow.textFieldUsername [private]
```

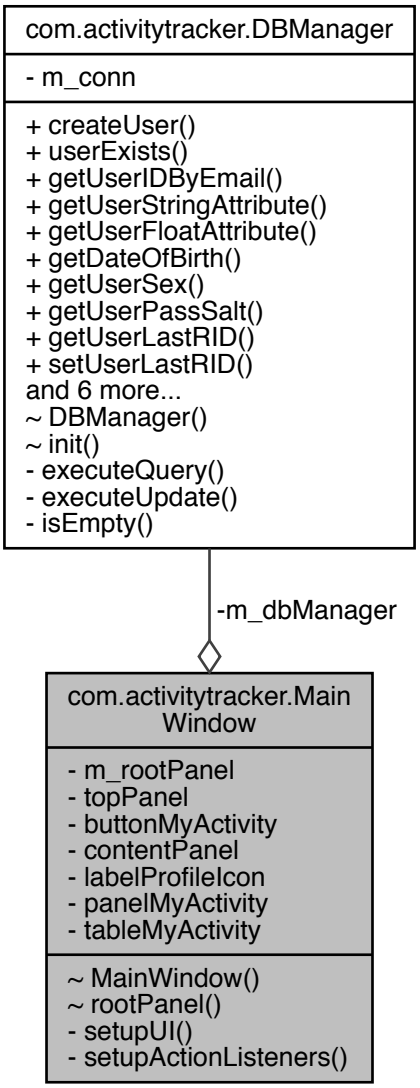
Definition at line 13 of file LoginWindow.java.

The documentation for this class was generated from the following file:

- app/src/com/activitytracker/LoginWindow.java

7.6 com.activitytracker.MainWindow Class Reference

Collaboration diagram for com.activitytracker.MainWindow:



Package Functions

- MainWindow (DBManager dbmanager)
- JPanel rootPanel ()

Private Member Functions

- void setupUI ()
- void setupActionListeners ()

Private Attributes

- JPanel m_rootPanel
- JPanel topPanel
- JButton buttonMyActivity
- JPanel contentPanel
- JLabel labelProfileIcon
- JPanel panelMyActivity
- JTable tableMyActivity
- DBManager m_dbManager = null

7.6.1 Detailed Description

Definition at line 12 of file MainWindow.java.

7.6.2 Constructor & Destructor Documentation

7.6.2.1 MainWindow()

```
com.activitytracker.MainWindow.MainWindow (
            DBManager dbmanager ) [package]
```

Definition at line 23 of file MainWindow.java.

```
23                                  {
24         m_dbManager = dbmanager;
25
26         setupUI();
27         setupActionListeners();
28     }
```

7.6.3 Member Function Documentation

7.6.3.1 rootPanel()

`JPanel com.activitytracker.MainWindow.rootPanel ( ) [package]`

Definition at line 58 of file MainWindow.java.

```
58                          {
59          return m_rootPanel;
60      }
```

7.6.3.2 setupActionListeners()

`void com.activitytracker.MainWindow.setupActionListeners ( ) [private]`

Definition at line 48 of file MainWindow.java.

```
48                                          {
49          // My Activity button
50          buttonMyActivity.addActionListener(new ActionListener() {
51              @Override
52              public void actionPerformed(ActionEvent actionEvent) {
53                  panelMyActivity.setVisible(true);
54              }
55          });
56      }
```

7.6.3.3 setupUI()

`void com.activitytracker.MainWindow.setupUI ( ) [private]`

Definition at line 30 of file MainWindow.java.

```
30                              {
31
32          // Apply Material-defined hover effect to buttons
33          Color coolGrey10 = new Color(99, 102, 106);
34          Color coolGrey11 = new Color(83, 86, 90);
35          MaterialUIMovement.add(buttonMyActivity, coolGrey11);
36
37          // Load and scale logo into UI
38          String logoPath = "./assets/logo.png";
39          ImageIcon imageIcon = new ImageIcon(getClass().getResource(logoPath));
40          final Image image = imageIcon.getImage(); // transform it
41          final Image newimg = image.getScaledInstance(50, 50, java.awt.Image.SCALE_SMOOTH);
42          imageIcon = new ImageIcon(newimg);  // transform it back
43          labelProfileIcon.setIcon(imageIcon);
44
45          panelMyActivity.setVisible(true);
46      }
```

7.6.4 Member Data Documentation

7.6.4.1 buttonMyActivity

```
JButton com.activitytracker.MainWindow.buttonMyActivity [private]
```

Definition at line 15 of file MainWindow.java.

7.6.4.2 contentPanel

```
JPanel com.activitytracker.MainWindow.contentPanel [private]
```

Definition at line 16 of file MainWindow.java.

7.6.4.3 labelProfileIcon

```
JLabel com.activitytracker.MainWindow.labelProfileIcon [private]
```

Definition at line 17 of file MainWindow.java.

7.6.4.4 m_dbManager

```
DBManager com.activitytracker.MainWindow.m_dbManager = null [private]
```

Definition at line 21 of file MainWindow.java.

7.6.4.5 m_rootPanel

```
JPanel com.activitytracker.MainWindow.m_rootPanel [private]
```

Definition at line 13 of file MainWindow.java.

7.6.4.6 panelMyActivity

```
JPanel com.activitytracker.MainWindow.panelMyActivity [private]
```

Definition at line 18 of file MainWindow.java.

7.6.4.7 tableMyActivity

```
JTable com.activitytracker.MainWindow.tableMyActivity [private]
```

Definition at line 19 of file MainWindow.java.

7.6.4.8 topPanel

```
JPanel com.activitytracker.MainWindow.topPanel [private]
```

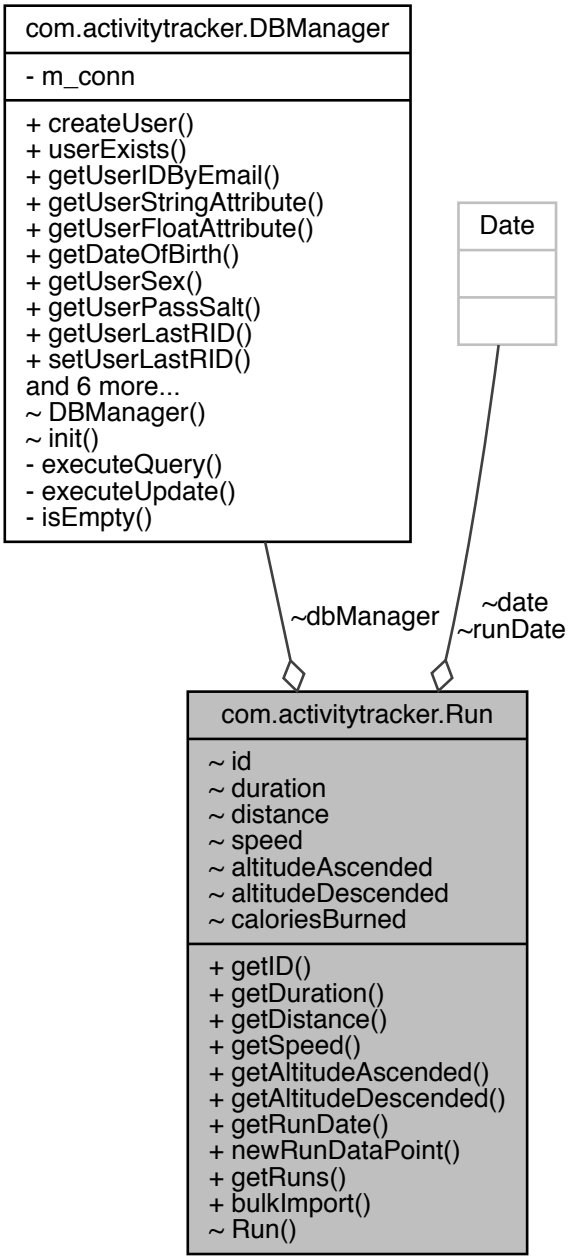
Definition at line 14 of file MainWindow.java.

The documentation for this class was generated from the following file:

- app/src/com/activitytracker/MainWindow.java

7.7 com.activitytracker.Run Class Reference

Collaboration diagram for com.activitytracker.Run:



Public Member Functions

- int getID ()
- float getDuration ()
- float getDistance ()
- float getSpeed ()
- float getAltitudeAscended ()
- float getAltitudeDescended ()
- Date getRunDate ()

Static Public Member Functions

- static void newRunDataPoint (final DBManager dbManager, final User user, final float duration, final Date date, final float distance, final float altitude)
- static Vector< Run > getRuns (final DBManager dbManager, final User user, final Date startDate, final Date endDate)
- static void bulkImport (final DBManager dbManager, final User user, final String filePath) throws FileNotFoundException, IOException

Package Functions

- Run (final DBManager dbManager, final int rID)

Package Attributes

- int id
- Date date
- DBManager dbManager
- float duration
- float distance
- float speed
- float altitudeAscended
- float altitudeDescended
- Date runDate
- long caloriesBurned = 0

7.7.1 Detailed Description

Used to logically instantiate a run.

Definition at line 17 of file Run.java.

7.7.2 Constructor & Destructor Documentation

7.7.2.1 Run()

```
com.activitytracker.Run.Run (
            final DBManager dbManager,
            final int rID ) [package]
```

The Run() constructor is used to retrieve workout information from the database and instantiate each row of the Runs table in a logical format.

Parameters

dbManager	The connection to the database.
rID	The run ID used to retrieve information from the database.

Definition at line 68 of file Run.java.

```
68                                                  {
69          this.id = rID;
70          this.dbManager = dbManager;
71          this.runDate = this.dbManager.getRunDate(rID);
72          this.duration = this.dbManager.getRunFloatAttribute(
      RunAttribute.DURATION, rID);
73          this.distance = this.dbManager.getRunFloatAttribute(
      RunAttribute.DISTANCE, rID);
74          this.speed = this.distance / this.duration;
75          this.altitudeAscended = this.dbManager.
      getRunFloatAttribute(RunAttribute.ALTITUDE_ASCENDED, rID);
76          this.altitudeDescended = this.dbManager.
      getRunFloatAttribute(RunAttribute.ALTITUDE_DESCENDED, rID);
77      }
```

7.7.3 Member Function Documentation

7.7.3.1 bulkImport()

```
static void com.activitytracker.Run.bulkImport (
            final DBManager dbManager,
            final User user,
            final String filePath ) throws FileNotFoundException, IOException [static]
```

Opens and iterates through a file. The Run::newRunDataPoint() method is called for each line.

Parameters

dbManager	Database connection with with the method interacts.
user	A User object corresponding to the use whose run(s) is/are being retrieved from the database.
filePath	The file to be iterated through

Exceptions

FileNotFoundException	Thrown if the file path given does not exist.
IOException	Thrown if there is an error reading or opening the file.

Definition at line 184 of file Run.java.

```
185                                                     {
186         BufferedReader br = new BufferedReader(new FileReader(filePath));
187         String line = null;
188         Date date = null;
189         while ((line = br.readLine()) != null)
190         {
191             String[] attributes = line.split(",");
192             String buffTime = attributes[0];
193             String buffDistance = attributes[1];
194             String buffAltitude = attributes[2];
195             String buffDate = attributes[3];
196             DateFormat sourceFormat = new SimpleDateFormat("dd-MM-yyyy");
197             try {
198                 date = sourceFormat.parse(buffDate);
199             }
200             catch (final ParseException e) {
201                 System.err.println(e.getMessage());
202             }
203 
204             // Convert strings to floats
205             float fDur = Float.parseFloat(buffTime);
206             float fDist = Float.parseFloat(buffDistance);
207             float fAlt = Float.parseFloat(buffAltitude);
208 
209             newRunDataPoint(dbManager, user, fDur, date, fDist, fAlt);
210         }
211     }
```

7.7.3.2 getAltitudeAscended()

```
float com.activitytracker.Run.getAltitudeAscended ( )
```

Retrieves a Run object's altitude ascended (in metres).

Returns

The Run's altitude ascended as defined in the database.

Definition at line 254 of file Run.java.

```
254                                     {
255         return altitudeAscended;
256     }
```

7.7.3.3 getAltitudeDescended()

```
float com.activitytracker.Run.getAltitudeDescended ( )
```

Retrieves a Run object's altitude descended (in metres).

Returns

The Run's altitude descended as defined in the database.

Definition at line 263 of file Run.java.

```
263                                      {
264         return altitudeDescended;
265     }
```

7.7.3.4 getDistance()

```
float com.activitytracker.Run.getDistance ( )
```

Retrieves a Run object's distance (in metres).

Returns

The Run's distance as defined in the database.

Definition at line 236 of file Run.java.

```
236                                 {
237         return distance;
238     }
```

7.7.3.5 getDuration()

```
float com.activitytracker.Run.getDuration ( )
```

Retrieves a Run object's duration (in seconds).

Returns

The Run's duration as defined in the database.

Definition at line 227 of file Run.java.

```
227                                 {
228         return duration;
229     }
```

7.7.3.6 getID()

```
int com.activitytracker.Run.getID ( )
```

Retrieves a Run object's ID.

Returns

The Run's ID as defined in the database.

Definition at line 218 of file Run.java.

```
218                         {
219         return this.id;
220     }
```

7.7.3.7 getRunDate()

```
Date com.activitytracker.Run.getRunDate ( )
```

Retrieves a Run object's date.

Returns

The Run's date.

Definition at line 272 of file Run.java.

```
272                             {
273         return runDate;
274     }
```

7.7.3.8 getRuns()

```
static Vector<Run> com.activitytracker.Run.getRuns (
            final DBManager dbManager,
            final User user,
            final Date startDate,
            final Date endDate ) [static]
```

Retrieves a set of runs from the database. Returns the result as a vector of Run objects.

Parameters

dbManager	Database connection with with the method interacts.
user	A User object corresponding to the use whose run(s) is/are being retrieved from the database.
startDate	The beginning of the interval for which we are retrieving workouts.
endDate	The end of the interval for which we are retrieving workouts.

Returns

A vector containing instances of Run corresponding to all entered workouts between the start and end dates specified.

Definition at line 153 of file Run.java.

```
154                                                                                 {
155         Vector<Run> runs = new Vector<>();
156         int rID;
157         Vector<Integer> rIDs = dbManager.getRuns(user.getID(), startDate, endDate);
158 
159         if (rIDs != null) {
160             Iterator<Integer> runIDIter = rIDs.iterator();
161             while (runIDIter.hasNext()) {
162                 rID = runIDIter.next();
163                 runs.add(new Run(dbManager, rID));
164             }
165             return runs;
166         }
167         else {
168             System.err.println("DBManager.getRuns() returned null.");
169             return null;
170         }
171     }
```

7.7.3.9 getSpeed()

`float com.activitytracker.Run.getSpeed ( )`

Retrieves a Run object's average speed (in metres per second).

Returns

The Run's average speed as computed in the Run() constructor.

Definition at line 245 of file Run.java.

```
245                             {
246         return speed;
247     }
```

7.7.3.10 newRunDataPoint()

```
static void com.activitytracker.Run.newRunDataPoint (
            final DBManager dbManager,
            final User user,
            final float duration,
            final Date date,
            final float distance,
            final float altitude ) [static]
```

Adds a new workout to the database or updates an existing workout with new information that the user imported from the log file.

If (*duration*, *distance*, *altitude*) passed to this method is (0, 0, 0) then the intended assumption is that this is the beginning of a new workout. As such, this input will cause a new row to be added to the Runs table in the database and the user's last run ID attribute will be updated accordingly. If the input is non-(0, 0, 0), then three things take place:

1. The *duration* in the database is overwritten by the *duration* provided as input;
2. The *distance* in the database is overwritten by the *distance* provided as input; and
3. Existing values for *altitude_ascended* and *altitude_descended* are retrieved from the database, their difference is compared to the current relative altitude, and depending whether this difference is positive or negative, the appropriate field in the database is updated to reflect the change.

Parameters

dbManager	Database connection with with the method interacts.
user	A User object corresponding to the use whose run is being added to the database.
duration	The length of time in seconds that the user's run lasted.
date	The date the run occurred.
distance	The cumulative distance (in metres) that the user ran as of the current time passed to the method.
altitude	The relative current altitude (in metres) of the user at the time point being entered. Used to compute cumulative altitude ascended and descended throughout the run.

Definition at line 103 of file Run.java.

```
104                                                                                 {
105         int userID = user.getID();
106         int rID;
107         float altitude_ascended;
```

```
108         float altitude_descended;
109
110         if (duration == 0f && distance == 0f && altitude == 0f) {
111             altitude_ascended = 0f;
112             altitude_descended = 0f;
113             rID = dbManager.newRun(
114                     userID,
115                     date,
116                     duration,
117                     distance,
118                     altitude_ascended,
119                     altitude_descended
120             );
121             user.setLastRID(rID);
122             System.err.println("Run " + Integer.toString(rID) + " added to database.");
123         } else {
124             rID = user.getLastRID();
125             if (dbManager.runExists(rID)) {
126                 altitude_ascended = dbManager.getRunFloatAttribute(
    RunAttribute.ALTITUDE_ASCENDED, rID);
127                 altitude_descended = dbManager.getRunFloatAttribute(
    RunAttribute.ALTITUDE_DESCENDED, rID);
128
129                 if (altitude < 0)
130                     altitude_descended += -1*altitude;
131                 else
132                     altitude_ascended += altitude;
133
134                 dbManager.setRun(rID, duration, distance, altitude_ascended,
     altitude_descended);
135                 System.err.println("Run " + Integer.toString(rID) + " exists in the database; updating...")
    ;
136             } else {
137                 System.err.println("Run table and User table are inconsistent. No changes made.");
138             }
139         }
140     }
```

7.7.4 Member Data Documentation

7.7.4.1 altitudeAscended

`float com.activitytracker.Run.altitudeAscended [package]`

The altitude (in metres) that the user climed throughout the run.

Definition at line 45 of file Run.java.

7.7.4.2 altitudeDescended

```
float com.activitytracker.Run.altitudeDescended [package]
```

The altitude (in metres) that the user descended throughout their run.

Definition at line 49 of file Run.java.

7.7.4.3 caloriesBurned

```
long com.activitytracker.Run.caloriesBurned = 0 [package]
```

The number of calories that the user burned throughout their run.

Currently this is not being used; it is for future features.

Definition at line 59 of file Run.java.

7.7.4.4 date

```
Date com.activitytracker.Run.date [package]
```

The date the run occurred.

Definition at line 25 of file Run.java.

7.7.4.5 dbManager

```
DBManager com.activitytracker.Run.dbManager [package]
```

The run's connection to the database. This is used to add data points and retrieve workout metadata.

Definition at line 29 of file Run.java.

7.7.4.6 distance

```
float com.activitytracker.Run.distance [package]
```

The distance (in metres) that the user ran.

Definition at line 37 of file Run.java.

7.7.4.7 duration

```
float com.activitytracker.Run.duration [package]
```

The length of the run in seconds.

Definition at line 33 of file Run.java.

7.7.4.8 id

```
int com.activitytracker.Run.id [package]
```

The run's unique ID.

Definition at line 21 of file Run.java.

7.7.4.9 runDate

```
Date com.activitytracker.Run.runDate [package]
```

The date the run took place.

Definition at line 53 of file Run.java.

7.7.4.10 speed

```
float com.activitytracker.Run.speed [package]
```

The average speed (in metres per second) that the user ran.

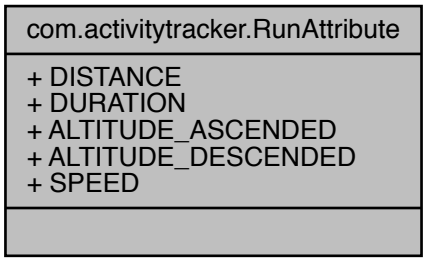
Definition at line 41 of file Run.java.

The documentation for this class was generated from the following file:

- app/src/com/activitytracker/Run.java

7.8 com.activitytracker.RunAttribute Enum Reference

Collaboration diagram for com.activitytracker.RunAttribute:



Public Attributes

- DISTANCE
- DURATION
- ALTITUDE_ASCENDED
- ALTITUDE_DESCENDED
- SPEED

7.8.1 Detailed Description

This enumeration type is used to specify the behaviour of generalized methods, particularly in the DBManager class.

Definition at line 6 of file RunAttribute.java.

7.8.2 Member Data Documentation

7.8.2.1 ALTITUDE_ASCENDED

`com.activitytracker.RunAttribute.ALTITUDE_ASCENDED`

The cumulative altitude (in metres) that the user has climbed throughout their run.

Used in DBManager::getRunFloatAttribute to specify that ascended altitude should be returned and RunStats::computeMean().

Definition at line 27 of file RunAttribute.java.

7.8.2.2 ALTITUDE_DESCENDED

`com.activitytracker.RunAttribute.ALTITUDE_DESCENDED`

The cumulative altitude (in metres) that the user has descended throughout their run.

Used in DBManager::getRunFloatAttribute to specify that descended altitude should be returned and RunStats::computeMean().

Definition at line 34 of file RunAttribute.java.

7.8.2.3 DISTANCE

`com.activitytracker.RunAttribute.DISTANCE`

The cumulative distance the user has run (in metres).

Used in DBManager::getRunFloatAttribute to specify that distance should be returned and RunStats::computeMean().

Definition at line 13 of file RunAttribute.java.

7.8.2.4 DURATION

`com.activitytracker.RunAttribute.DURATION`

The duration of the user's run (in seconds).

Used in DBManager::getRunFloatAttribute to specify that duration should be returned and RunStats::computeMean().

Definition at line 20 of file RunAttribute.java.

7.8.2.5 SPEED

`com.activitytracker.RunAttribute.SPEED`

The average speed the user ran.

Used in RunStats::computeMean().

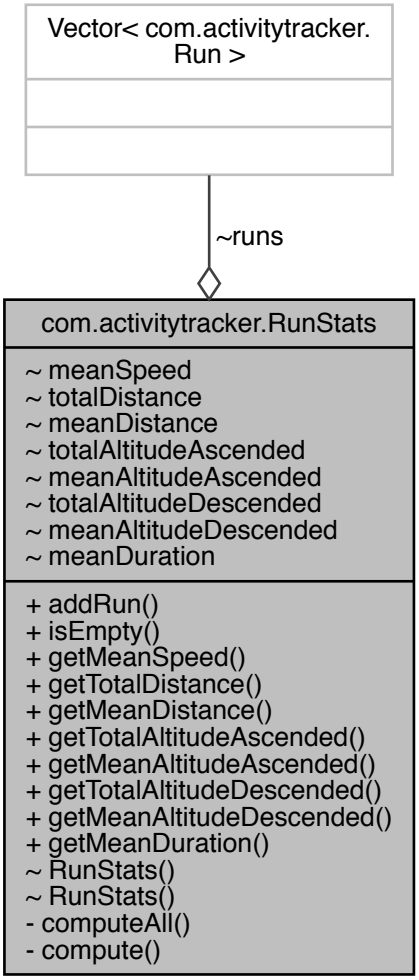
Definition at line 40 of file RunAttribute.java.

The documentation for this enum was generated from the following file:

- app/src/com/activitytracker/RunAttribute.java

7.9 com.activitytracker.RunStats Class Reference

Collaboration diagram for com.activitytracker.RunStats:



Public Member Functions

- void addRun (Run run)
- boolean isEmpty ()
- float getMeanSpeed ()
- float getTotalDistance ()

- float getMeanDistance ()
- float getTotalAltitudeAscended ()
- float getMeanAltitudeAscended ()
- float getTotalAltitudeDescended ()
- float getMeanAltitudeDescended ()
- float getMeanDuration ()

Package Functions

- RunStats (final Vector< Run > runs)
- RunStats ()

Package Attributes

- Vector< Run > runs
- float meanSpeed
- float totalDistance
- float meanDistance
- float totalAltitudeAscended
- float meanAltitudeAscended
- float totalAltitudeDescended
- float meanAltitudeDescended
- float meanDuration

Private Member Functions

- void computeAll ()
- void compute (final RunAttribute attribute)

7.9.1 Detailed Description

The RunStats class is used to compute statistics for a selection of past runs.

It is intended to be integrated with the Run::getRuns method, as that returns a vector of Run objects that match search parameters, and the constructor of this class accepts a vector of Run objects, or nothing.

Definition at line 12 of file RunStats.java.

7.9.2 Constructor & Destructor Documentation

7.9.2.1 RunStats() [1/2]

```
com.activitytracker.RunStats.RunStats (
            final Vector< Run > runs ) [package]
```

Stores the *runs* parameter in RunStats::runs and computes all statistics based on the contents of this vector.

Parameters

runs	A vector of Run objects with which we compute statistics.

Definition at line 56 of file RunStats.java.

```
56                                      {
57         this.runs = runs;
58         this.computeAll();
59     }
```

7.9.2.2 RunStats() [2/2]

```
com.activitytracker.RunStats.RunStats ( ) [package]
```

An overloaded constructor which takes no argument. Here we assign to RunStats::runs an empty vector. Computations are attempted but since the vector has size of zero, RunStats::compute() assigns all stats the value *0.0f*.

You may want to use RunStats::addRun() to add Runs to a (possibly empty) RunStats object.

Definition at line 68 of file RunStats.java.

```
68                 {
69         this(new Vector<>());
70     }
```

7.9.3 Member Function Documentation

7.9.3.1 addRun()

```
void com.activitytracker.RunStats.addRun (
          Run run )
```

Adds a Run object to the RunStats::runs vector and re-computes statistics by invoking RunStats::computeAll().

Parameters

run	A Run object to be appended to the set of runs that the RunStats statistics are based on.

Definition at line 161 of file RunStats.java.

```
161                                 {
162         this.runs.addElement(run);
163         this.computeAll();
164     }
```

7.9.3.2 compute()

```
void com.activitytracker.RunStats.compute (
          final RunAttribute attribute ) [private]
```

Computes and stores averages for the RunAttribute passed as *attribute*, and stores a total sum where applicable.

Parameters

attribute	The RunAttribute for which we are computing statistics.

Definition at line 91 of file RunStats.java.

```
91                                                  {
```

```
92          float sum = 0.0f;
93          int numRuns = runs.size();
94
95          // If there are no runs, set everything to 0
96          if (numRuns == 0) {
97              this.meanSpeed = 0.0f;
98              this.totalDistance = 0.0f;
99              this.meanDistance = 0.0f;
100              this.totalAltitudeAscended = 0.0f;
101              this.meanAltitudeAscended = 0.0f;
102              this.totalAltitudeDescended = 0.0f;
103              this.meanAltitudeDescended = 0.0f;
104              this.meanDuration = 0.0f;
105              return;
106          }
107
108          switch (attribute) {
109              case DISTANCE:
110                  for (Run run : this.runs)
111                      sum += run.getDistance();
112                  this.totalDistance = sum;
113                  break;
114              case DURATION:
115                  for (Run run : this.runs)
116                      sum += run.getDuration();
117                  break;
118              case ALTITUDE_ASCENDED:
119                  for (Run run : this.runs)
120                      sum += run.getAltitudeAscended();
121                  this.totalAltitudeAscended = sum;
122                  break;
123              case ALTITUDE_DESCENDED:
124                  for (Run run : this.runs)
125                      sum += run.getAltitudeDescended();
126                  this.totalAltitudeDescended = sum;
127                  break;
128              case SPEED:
129                  for (Run run : this.runs)
130                      sum += run.getSpeed();
131                  break;
132          }
133
134          float mean = sum / numRuns;
135
136          switch (attribute) {
137              case DISTANCE:
138                  this.meanDistance = mean;
139                  break;
140              case DURATION:
141                  this.meanDuration = mean;
142                  break;
143              case ALTITUDE_ASCENDED:
144                  this.meanAltitudeAscended = mean;
145                  break;
146              case ALTITUDE_DESCENDED:
147                  this.meanAltitudeDescended = mean;
148                  break;
149              case SPEED:
150                  this.meanSpeed = mean;
151          }
152      }
```

7.9.3.3 computeAll()

```
void com.activitytracker.RunStats.computeAll ( ) [private]
```

A wrapper for the compute() method in this class that invokes all calculations.

Basically, it's the lazy man's way of calling the full computation.

Definition at line 77 of file RunStats.java.

```
77                                {
78         this.compute(RunAttribute.SPEED);
79         this.compute(RunAttribute.DISTANCE);
80         this.compute(RunAttribute.ALTITUDE_ASCENDED);
81         this.compute(RunAttribute.ALTITUDE_DESCENDED);
82         this.compute(RunAttribute.DURATION);
83     }
```

7.9.3.4 getMeanAltitudeAscended()

```
float com.activitytracker.RunStats.getMeanAltitudeAscended ( )
```

Definition at line 194 of file RunStats.java.

```
194                                             {
195         return meanAltitudeAscended;
196     }
```

7.9.3.5 getMeanAltitudeDescended()

```
float com.activitytracker.RunStats.getMeanAltitudeDescended ( )
```

Definition at line 202 of file RunStats.java.

```
202                                              {
203         return meanAltitudeDescended;
204     }
```

7.9.3.6 getMeanDistance()

```
float com.activitytracker.RunStats.getMeanDistance ( )
```

Definition at line 186 of file RunStats.java.

```
186                                     {
187         return meanDistance;
188     }
```

7.9.3.7 getMeanDuration()

```
float com.activitytracker.RunStats.getMeanDuration ( )
```

Definition at line 206 of file RunStats.java.

```
206                                     {
207         return meanDuration;
208     }
```

7.9.3.8 getMeanSpeed()

```
float com.activitytracker.RunStats.getMeanSpeed ( )
```

Definition at line 178 of file RunStats.java.

```
178                                   {
179         return meanSpeed;
180     }
```

7.9.3.9 getTotalAltitudeAscended()

```
float com.activitytracker.RunStats.getTotalAltitudeAscended ( )
```

Definition at line 190 of file RunStats.java.

```
190                                         {
191         return totalAltitudeAscended;
192     }
```

7.9.3.10 getTotalAltitudeDescended()

```
float com.activitytracker.RunStats.getTotalAltitudeDescended ( )
```

Definition at line 198 of file RunStats.java.

```
198                                          {
199         return totalAltitudeDescended;
200     }
```

7.9.3.11 getTotalDistance()

```
float com.activitytracker.RunStats.getTotalDistance ( )
```

Definition at line 182 of file RunStats.java.

```
182                                    {
183         return totalDistance;
184     }
```

7.9.3.12 isEmpty()

```
boolean com.activitytracker.RunStats.isEmpty ( )
```

Checks if an instance of RunStats has any runs with with statistics have been computed.

If this method returns *False* then all statistics are set to *0.0f* (*i.e.*, they are useless).

Returns

True if the RunStats::runs vector has a size of zero, false otherwise.

Definition at line 174 of file RunStats.java.

```
174                                {
175         return this.runs.size() == 0;
176     }
```

7.9.4 Member Data Documentation

7.9.4.1 meanAltitudeAscended

```
float com.activitytracker.RunStats.meanAltitudeAscended [package]
```

Mean altitude climbed per run, computed using the runs in RunStats::runs.

Definition at line 36 of file RunStats.java.

7.9.4.2 meanAltitudeDescended

```
float com.activitytracker.RunStats.meanAltitudeDescended [package]
```

Mean altitude descended per run, computed using the runs in RunStats::runs.

Definition at line 44 of file RunStats.java.

7.9.4.3 meanDistance

```
float com.activitytracker.RunStats.meanDistance [package]
```

Mean distance per run, computed using the runs in RunStats::runs.

Definition at line 28 of file RunStats.java.

7.9.4.4 meanDuration

```
float com.activitytracker.RunStats.meanDuration [package]
```

Mean duration per run, computed using the runs in RunStats::runs.

Definition at line 48 of file RunStats.java.

7.9.4.5 meanSpeed

```
float com.activitytracker.RunStats.meanSpeed [package]
```

The average speed for all runs in RunStats::runs.

Definition at line 20 of file RunStats.java.

7.9.4.6 runs

```
Vector<Run> com.activitytracker.RunStats.runs [package]
```

The runs for which the statistics in the RunStats object pertain.

Definition at line 16 of file RunStats.java.

7.9.4.7 totalAltitudeAscended

```
float com.activitytracker.RunStats.totalAltitudeAscended [package]
```

Cumulative altitude climbed for all runs in RunStats::runs.

Definition at line 32 of file RunStats.java.

7.9.4.8 totalAltitudeDescended

```
float com.activitytracker.RunStats.totalAltitudeDescended [package]
```

Cumulative altitude descended for all runs in RunStats::runs.

Definition at line 40 of file RunStats.java.

7.9.4.9 totalDistance

```
float com.activitytracker.RunStats.totalDistance [package]
```

Cumulative distance ran for all runs in RunStats::runs.

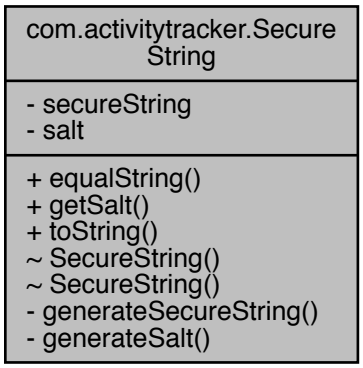
Definition at line 24 of file RunStats.java.

The documentation for this class was generated from the following file:

- app/src/com/activitytracker/RunStats.java

7.10 com.activitytracker.SecureString Class Reference

Collaboration diagram for com.activitytracker.SecureString:



Public Member Functions

- boolean equalString (final String other)
- byte [ ] getSalt ()
- String toString ()

Package Functions

- SecureString (final String plaintext)
- SecureString (final String plaintext, final byte[ ] salt)

Private Member Functions

- String generateSecureString (final String strToSecure, final byte[ ] salt)

Static Private Member Functions

- static byte [ ] generateSalt () throws NoSuchAlgorithmException

Private Attributes

- String secureString
- byte [] salt

7.10.1 Detailed Description

This class is used to securely store sensitive string-like information such as user passwords.

Definition at line 10 of file SecureString.java.

7.10.2 Constructor & Destructor Documentation

7.10.2.1 SecureString() [1/2]

```
com.activitytracker.SecureString.SecureString (
            final String plaintext ) [package]
```

The SecureString() constructor takes as an argument a plain text string, encrypts it, and stores the encrypted string in the variable SecureString::secureString.

Salt is generated using SecureString::generateSalt().

Parameters

plaintext	The string to be encrypted. May contain sensitive information.

Definition at line 29 of file SecureString.java.

```
29                                          {
30
31         try {
32             this.salt = generateSalt();
33         }
34         catch (final NoSuchAlgorithmException e) {
35             System.err.println(e.getMessage());
36         }
37         this.secureString = generateSecureString(plaintext, this.
       salt);
38
39     }
```

7.10.2.2 SecureString() [2/2]

```
com.activitytracker.SecureString.SecureString (
            final String plaintext,
            final byte [] salt ) [package]
```

The SecureString() constructor takes as an argument a plain text string and a previously-generated salt, encrypts the plain text string with the provided salt, and stores the encrypted string in the variable SecureString::secureString.

Parameters

plaintext	The string to be encrypted. May contain sensitive information.
salt	Salt that is used to encrypt *plaintext*. This parameter is used whenever we wish to encrypt using a previously-generated salt for the purpose of encrypted string comparison.

Definition at line 50 of file SecureString.java.

```
50                                                          {
51
52          this.salt = salt;
53          this.secureString = generateSecureString(plaintext,
      salt);
54
55      }
```

7.10.3 Member Function Documentation

7.10.3.1 equalString()

```
boolean com.activitytracker.SecureString.equalString (
            final String other )
```

Compares the secure string to the *other* parameter for equality.

This method will likely be used to authenticate a user from a password hash existing in the database.

Parameters

other	A (previously encrypted) string with with we compare SecureString::secureString.

Returns

This method returns True if the hashes of both strings are the same, and False otherwise.

Definition at line 66 of file SecureString.java.

```
66                                                  {
67
68         return this.secureString.equals(other);
69
70     }
```

7.10.3.2 generateSalt()

```
static byte [] com.activitytracker.SecureString.generateSalt ( ) throws No↩
SuchAlgorithmException [static], [private]
```

This method generates salt for encryption of a plain text string.

Returns

Returns a byte array of length sixteen (16) containing the encryption salt.

Exceptions

NoSuchAlgorithmException	Required as *SecureRandom.getInstace()* may throw this exception and we would like the invoking method to decide how to handle it rather than catching and dismissing it here.

Definition at line 83 of file SecureString.java.

```
83                                                                      {
84         SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
85         byte[] salt = new byte[16];
86         sr.nextBytes(salt);
87         return salt;
88     }
```

7.10.3.3 generateSecureString()

```
String com.activitytracker.SecureString.generateSecureString (
          final String strToSecure,
          final byte [] salt ) [private]
```

Encrypt string and return secure version.

Due to the importance of securely storing passwords, a "tried and true" method for encrypting passwords found at this link has been used.

Parameters

strToSecure	The plain text string we wish to encrypt.
salt	The salt with which we will encrypt *strToSecure*.

Returns

This private method returns the encrypted string to the SecureString() constructor.

Definition at line 102 of file SecureString.java.

```
102                                                                                 {
103         String generatedPassword = null;
104         try {
105             MessageDigest md = MessageDigest.getInstance("SHA-512");
106             md.update(salt);
107             byte[] strBytes = md.digest(strToSecure.getBytes());
108             StringBuilder sb = new StringBuilder();
109             for (int i = 0; i < strBytes.length; i++) {
110                 sb.append(Integer.toString((strBytes[i] & 0xff) + 0x100, 16).substring(1));
111             }
112             generatedPassword = sb.toString();
113         }
114         catch (final NoSuchAlgorithmException e) {
115             System.err.println(e.getMessage());
116         }
117         return generatedPassword;
118     }
```

7.10.3.4 getSalt()

```
byte [] com.activitytracker.SecureString.getSalt ( )
```

Returns

Returns the byte array-type salt used to encrypt the text given to the object's constructor.

Definition at line 123 of file SecureString.java.

```
123                             {
124         return this.salt;
125     }
```

7.10.3.5 toString()

```
String com.activitytracker.SecureString.toString ( )
```

Overrided method to return the object as a Java String.

The encrypted string will be returned, though it should be noted for completeness that this is not a full representation of the object since the salt is crucial in arriving at SecureString::secureString being returned.

Returns

Returns the encrypted string.

Definition at line 136 of file SecureString.java.

```
136                              {
137         return this.secureString;
138     }
```

7.10.4 Member Data Documentation

7.10.4.1 salt

```
byte [] com.activitytracker.SecureString.salt [private]
```

The salt that was used to encrypt the plain text string.

Definition at line 19 of file SecureString.java.

7.10.4.2 secureString

```
String com.activitytracker.SecureString.secureString [private]
```

The encrypted string.

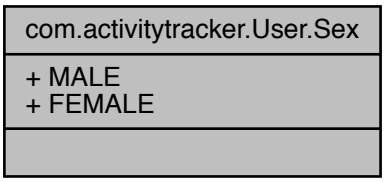
Definition at line 15 of file SecureString.java.

The documentation for this class was generated from the following file:

- app/src/com/activitytracker/SecureString.java

7.11 com.activitytracker.User.Sex Enum Reference

Collaboration diagram for com.activitytracker.User.Sex:



Public Attributes

- MALE
- FEMALE

7.11.1 Detailed Description

Used to represent whether the user is male or female.

Definition at line 12 of file User.java.

7.11.2 Member Data Documentation

7.11.2.1 FEMALE

`com.activitytracker.User.Sex.FEMALE`

Used to represent that the user is female.

Recall from the source code included in DBManager::init() that sex is stored in the database using a data type of BIT(1). If the user is female, we store this in the database by populating this field with a *0*.

Definition at line 26 of file User.java.

7.11.2.2 MALE

`com.activitytracker.User.Sex.MALE`

Used to represent that the user is male.

Recall from the source code included in DBManager::init() that sex is stored in the database using a data type of BIT(1). If the user is female, we store this in the database by populating this field with a *1*.

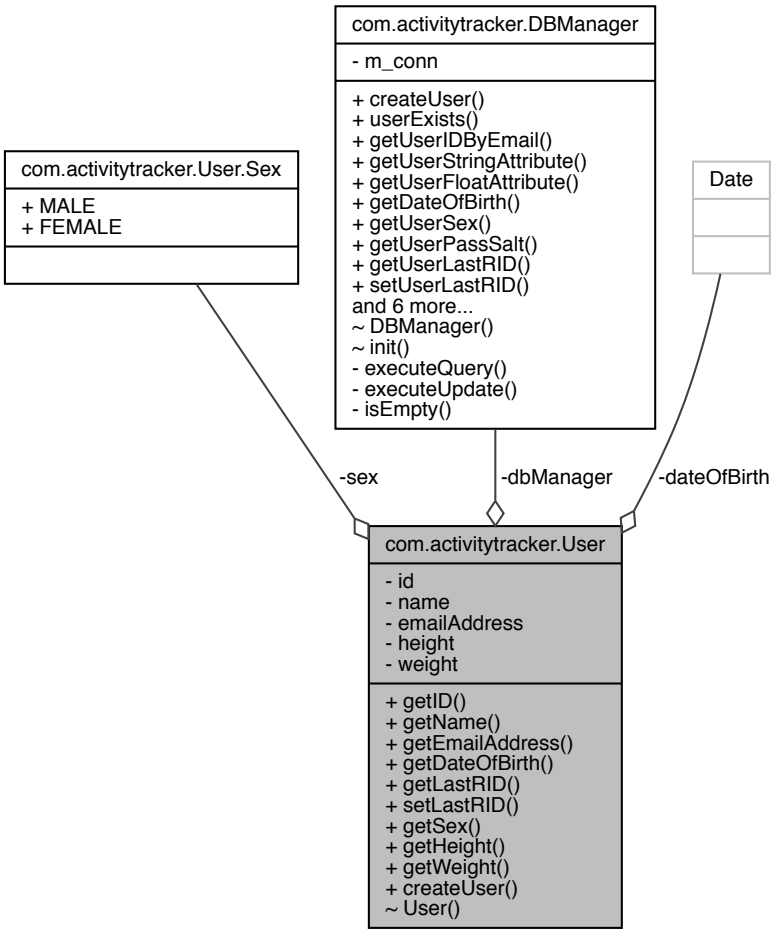
Definition at line 19 of file User.java.

The documentation for this enum was generated from the following file:

- app/src/com/activitytracker/User.java

7.12 com.activitytracker.User Class Reference

Collaboration diagram for com.activitytracker.User:



Classes

- enum Sex

Public Member Functions

- int getID ()

- String getName ()
- String getEmailAddress ()
- Date getDateOfBirth ()
- int getLastRID ()
- void setLastRID (final int rID)
- Sex getSex ()
- float getHeight ()
- float getWeight ()

Static Public Member Functions

- static void createUser (final DBManager dbManager, final String name, final String emailAddress, final Date dateOfBirth, final User.Sex sex, final float height, final float weight, final String plaintextPassword)

Package Functions

- User (final DBManager dbManager, final String emailAddress, final String plaintext↩ Password) throws AuthenticationException

Private Attributes

- int id
- String name
- String emailAddress
- Date dateOfBirth
- Sex sex
- float height
- float weight
- DBManager dbManager = null

7.12.1 Detailed Description

Definition at line 8 of file User.java.

7.12.2 Constructor & Destructor Documentation

7.12.2.1 User()

```
com.activitytracker.User.User (
            final DBManager dbManager,
            final String emailAddress,
            final String plaintextPassword ) throws AuthenticationException [pack-
age]
```

Definition at line 64 of file User.java.

```
64
                        {
65
          this.dbManager = dbManager;
66
          if (this.dbManager.userExists(emailAddress)) {
67
68
              this.id = dbManager.getUserIDByEmail(
69        emailAddress);
70
              String passHash = this.dbManager.getUserStringAttribute(
71        UserAttribute.PASSWORD, this.id);
              byte[] passSalt = this.dbManager.getUserPassSalt(this.id);
72
73
              SecureString candidatePassword = new SecureString(plaintextPassword, passSalt);
74
75
              if (candidatePassword.equalString(passHash)) {
76
77
78
                  this.name = this.dbManager.getUserStringAttribute(
79        UserAttribute.NAME, this.id);
80 //                 this.emailAddress = this.dbManager.getEmailAddress(this.id);
                  this.emailAddress = emailAddress;
81
                  this.dateOfBirth = this.dbManager.
82        getDateOfBirth(this.id);
                  this.sex = this.dbManager.getUserSex(this.id);
83
                  this.height = this.dbManager.
84        getUserFloatAttribute(UserAttribute.HEIGHT, this.id);
                  this.weight = this.dbManager.
85        getUserFloatAttribute(UserAttribute.WEIGHT, this.id);
86
                  System.out.println("Authentication succeeded for " + this.name);
87
88
              }
89
              else {
90
91
                  throw new AuthenticationException("Incorrect password.");
92
93
              }
94
          }
95
          else {
96
97
              throw new NoSuchElementException("No such user exists.");
98
99
           }
100
101
      }
102
```

7.12.3 Member Function Documentation

7.12.3.1 createUser()

```
static void com.activitytracker.User.createUser (
            final DBManager dbManager,
            final String name,
            final String emailAddress,
            final Date dateOfBirth,
            final User.Sex sex,
            final float height,
            final float weight,
            final String plaintextPassword ) [static]
```

Generates a SecureString for the plain text password provided by the user and passes this, along with the rest of the user's information, to DBManager for entry in the database.

Parameters

dbManager	An instance of DBManager with which we access the database to store the new user.
name	The new user's full name (*e.g.*, Johnathan Doe).
emailAddress	The new user's email address (*e.g.*, jondoe@mac.com) used to register.
dateOfBirth	
sex	
height	
weight	
plaintextPassword	

Definition at line 117 of file User.java.

```
119                                                                                 {
120
121         SecureString securePassword = new SecureString(plaintextPassword);
122
123
124         dbManager.createUser(
125                 name,
126                 emailAddress,
127                 dateOfBirth,
128                 sex,
129                 height,
130                 weight,
131                 securePassword
132         );
133
134     }
```

7.12.3.2 getDateOfBirth()

```
Date com.activitytracker.User.getDateOfBirth ( )
```

Definition at line 148 of file User.java.

```
148                                 {
149         return this.dateOfBirth;
150     }
```

7.12.3.3 getEmailAddress()

```
String com.activitytracker.User.getEmailAddress ( )
```

Definition at line 144 of file User.java.

```
144                                    {
145         return this.emailAddress;
146     }
```

7.12.3.4 getHeight()

```
float com.activitytracker.User.getHeight ( )
```

Definition at line 160 of file User.java.

```
160                                {
161         return this.height;
162     }
```

7.12.3.5 getID()

```
int com.activitytracker.User.getID ( )
```

Definition at line 136 of file User.java.

```
136                         {
137         return this.id;
138     }
```

7.12.3.6 getLastRID()

```
int com.activitytracker.User.getLastRID ( )
```

Definition at line 152 of file User.java.

```
152 { return this.dbManager.getUserLastRID(this.id); }
```

7.12.3.7 getName()

```
String com.activitytracker.User.getName ( )
```

Definition at line 140 of file User.java.

```
140                             {
141         return this.name;
142     }
```

7.12.3.8 getSex()

```
Sex com.activitytracker.User.getSex ( )
```

Definition at line 156 of file User.java.

```
156                         {
157         return this.sex;
158     }
```

7.12.3.9 getWeight()

```
float com.activitytracker.User.getWeight ( )
```

Definition at line 164 of file User.java.

```
164                                {
165         return this.weight;
166     }
```

7.12.3.10 setLastRID()

```
void com.activitytracker.User.setLastRID (
           final int rID )
```

Definition at line 154 of file User.java.

```
154 { this.dbManager.setUserLastRID(this.id, rID); }
```

7.12.4 Member Data Documentation

7.12.4.1 dateOfBirth

```
Date com.activitytracker.User.dateOfBirth [private]
```

The user's date of birth (*e.g.*, 12-12-1998).

Definition at line 45 of file User.java.

7.12.4.2 dbManager

```
DBManager com.activitytracker.User.dbManager = null [private]
```

An instance of DBManager with which we perform any DB accesses required to retrieve or set user attributes in the database.

Definition at line 62 of file User.java.

7.12.4.3 emailAddress

```
String com.activitytracker.User.emailAddress [private]
```

The email address (*e.g.*, jondoe@mac.com) that the user entered when registering for the app. This is used to authenticate the user at login.

Definition at line 41 of file User.java.

7.12.4.4 height

```
float com.activitytracker.User.height [private]
```

The user's height in metres.

Definition at line 53 of file User.java.

7.12.4.5 id

```
int com.activitytracker.User.id [private]
```

The user's ID, used by the database to associate workouts with a user.

Definition at line 32 of file User.java.

7.12.4.6 name

```
String com.activitytracker.User.name [private]
```

The user's full name (*e.g.*, Johnathan Doe).

Definition at line 36 of file User.java.

7.12.4.7 sex

```
Sex com.activitytracker.User.sex [private]
```

The user's sex. Can be one of User.Sex.MALE or User.Sex.FEMALE.

Definition at line 49 of file User.java.

7.12.4.8 weight

```
float com.activitytracker.User.weight [private]
```

The user's weight in kilograms.

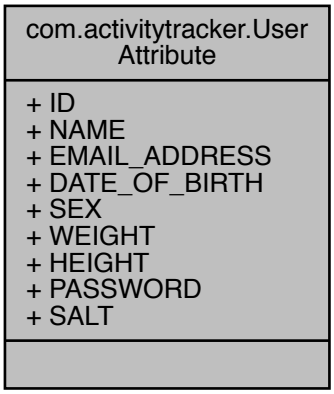
Definition at line 57 of file User.java.

The documentation for this class was generated from the following file:

- app/src/com/activitytracker/User.java

7.13 com.activitytracker.UserAttribute Enum Reference

Collaboration diagram for com.activitytracker.UserAttribute:



Public Attributes

- ID
- NAME
- EMAIL_ADDRESS
- DATE_OF_BIRTH
- SEX
- WEIGHT
- HEIGHT
- PASSWORD
- SALT

7.13.1 Detailed Description

This enumeration type is used to specify the behaviour of generalized methods, particularly in the DBManager class.

Definition at line 6 of file UserAttribute.java.

7.13.2 Member Data Documentation

7.13.2.1 DATE_OF_BIRTH

```
com.activitytracker.UserAttribute.DATE_OF_BIRTH
```

Currently not used as no generalized method retrieves the user's DOB.

Definition at line 26 of file UserAttribute.java.

7.13.2.2 EMAIL_ADDRESS

```
com.activitytracker.UserAttribute.EMAIL_ADDRESS
```

The user's email address.

Used in DBManager::getUserStringAttribute to specify that the user's email address should be returned.

Definition at line 22 of file UserAttribute.java.

7.13.2.3 HEIGHT

```
com.activitytracker.UserAttribute.HEIGHT
```

The user's height (in metres).

Used in DBManager::getUserFloatAttribute to specify that the user's email height should be returned.

Definition at line 42 of file UserAttribute.java.

7.13.2.4 ID

`com.activitytracker.UserAttribute.ID`

Currently not used as no generalized method retrieves the user's ID.

Definition at line 10 of file UserAttribute.java.

7.13.2.5 NAME

`com.activitytracker.UserAttribute.NAME`

The user's full name.

Used in DBManager::getUserStringAttribute to specify that the user's name should be returned.

Definition at line 16 of file UserAttribute.java.

7.13.2.6 PASSWORD

`com.activitytracker.UserAttribute.PASSWORD`

The user's encrypted password hash.

Used in DBManager::getUserStringAttribute to specify that the user's password hash should be returned.

Definition at line 48 of file UserAttribute.java.

7.13.2.7 SALT

`com.activitytracker.UserAttribute.SALT`

Currently not used as no generalized method retrieves the user's password encryption salt.

Definition at line 52 of file UserAttribute.java.

7.13.2.8 SEX

`com.activitytracker.UserAttribute.SEX`

Currently not used as no generalized method retrieves the user's sex.

Definition at line 30 of file UserAttribute.java.

7.13.2.9 WEIGHT

`com.activitytracker.UserAttribute.WEIGHT`

The user's weight (in kilograms).

Used in DBManager::getUserFloatAttribute to specify that the user's email weight should be returned.

Definition at line 36 of file UserAttribute.java.

The documentation for this enum was generated from the following file:

- app/src/com/activitytracker/UserAttribute.java

Chapter 8

File Documentation

8.1 app/src/com/activitytracker/ActivityTracker.java File Reference

Classes

- class com.activitytracker.ActivityTracker

Packages

- package com.activitytracker

8.2 app/src/com/activitytracker/CreateUserWindow.java File Reference

Classes

- class com.activitytracker.CreateUserWindow

Packages

- package com.activitytracker

8.3 app/src/com/activitytracker/DBManager.java File Reference

Classes

- class com.activitytracker.DBManager

Packages

- package com.activitytracker

8.4 app/src/com/activitytracker/Iteration3Test.java File Reference

Classes

- class com.activitytracker.Iteration3Test

Packages

- package com.activitytracker

8.5 app/src/com/activitytracker/LoginWindow.java File Reference

Classes

- class com.activitytracker.LoginWindow

Packages

- package com.activitytracker

8.6 app/src/com/activitytracker/MainWindow.java File Reference

Classes

- class com.activitytracker.MainWindow

Packages

- package com.activitytracker

8.7 app/src/com/activitytracker/Run.java File Reference

Classes

- class com.activitytracker.Run

Packages

- package com.activitytracker

8.8 app/src/com/activitytracker/RunAttribute.java File Reference

Classes

- enum com.activitytracker.RunAttribute

Packages

- package com.activitytracker

8.9 app/src/com/activitytracker/RunStats.java File Reference

Classes

- class com.activitytracker.RunStats

Packages

- package com.activitytracker

8.10 app/src/com/activitytracker/SecureString.java File Reference

Classes

- class com.activitytracker.SecureString

Packages

- package com.activitytracker

8.11 app/src/com/activitytracker/User.java File Reference

Classes

- class com.activitytracker.User
- enum com.activitytracker.User.Sex

Packages

- package com.activitytracker

8.12 app/src/com/activitytracker/UserAttribute.java File Reference

Classes

- enum com.activitytracker.UserAttribute

Packages

- package com.activitytracker

Index