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

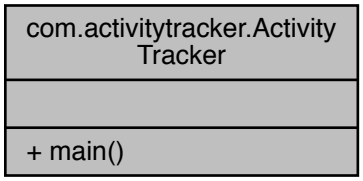
- class ActivityTracker
- class CreateUserWindow
- class DBManager
- class Iteration3Test
- class LoginWindow
- class MainWindow
- class Run
- enum RunAttribute
- class SecureString
- class User
- enum UserAttribute

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.

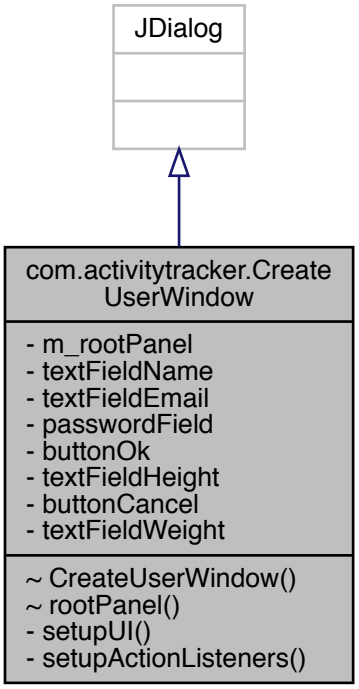
```
33                                                  {
34
35         // Create singleton instance of DBManager
36         DBManager dbManager = new DBManager();
37         if (!dbManager.init("data.db")) {
38             System.err.println("Failed to initialize DBManager");
39             System.exit(1);
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().rootPanel());
59             frame.validate();
60             frame.repaint();
61         }).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:

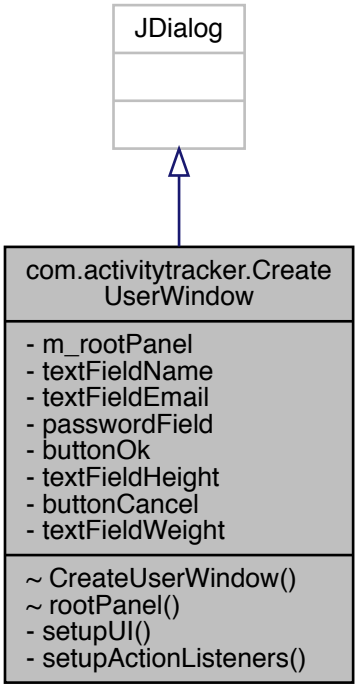
- app/src/com/activitytracker/ActivityTracker.java

7.2 com.activitytracker.CreateUserWindow Class Reference

Inheritance diagram for com.activitytracker.CreateUserWindow:



Collaboration diagram for com.activitytracker.CreateUserWindow:



Package Functions

- CreateUserWindow ()
- JPanel rootPanel ()

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 ( ) [package]`

Definition at line 20 of file CreateUserWindow.java.

```
20                          {
21
22          setupUI();
23          setupActionListeners();
24      }
```

7.2.3 Member Function Documentation

7.2.3.1 rootPanel()

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

Definition at line 53 of file CreateUserWindow.java.

```
53                          {
54          return m_rootPanel;
55      }
```

7.2.3.2 setupActionListeners()

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

Definition at line 31 of file CreateUserWindow.java.

```
31                                      {
32         buttonOk.addActionListener(new ActionListener() {
33             @Override
34             public void actionPerformed(ActionEvent e) {
35 
36                 if (textFieldName.getText().isEmpty() ||
37                     textFieldEmail.getText().isEmpty() ||
38                     passwordField.getPassword().length == 0) {
39 
40                     return;
41                 }
42             }
43         });
44 
45         buttonCancel.addActionListener(new ActionListener() {
46             @Override
47             public void actionPerformed(ActionEvent e) {
48 
49             }
50         });
51     }
```

7.2.3.3 setupUI()

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

Definition at line 26 of file CreateUserWindow.java.

```
26                          {
27         MaterialUIMovement.add(buttonCancel, MaterialColors.GRAY_100);
28         MaterialUIMovement.add(buttonOk, MaterialColors.GRAY_100);
29     }
```

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_rootPanel

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

Definition at line 11 of file CreateUserWindow.java.

7.2.4.4 passwordField

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

Definition at line 14 of file CreateUserWindow.java.

7.2.4.5 textFieldEmail

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

Definition at line 13 of file CreateUserWindow.java.

7.2.4.6 textFieldHeight

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

Definition at line 16 of file CreateUserWindow.java.

7.2.4.7 textFieldName

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

Definition at line 12 of file CreateUserWindow.java.

7.2.4.8 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 int DOBYear, final int DOBMonth, final int DOBDay, 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 int year, final int month, final int day, final float duration, final float distance, final float altitude_ascended, final float altitude_descended)
- void setRun (final int rID, final float duration, final float distance, final float altitude_↩ ascended, final float altitude_descended)
- float getRunFloatAttribute (final RunAttribute attribute, final int rID)
- boolean runExists (final int rID)

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 25 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 42 of file DBManager.java.

```
42                 {
43     }
```

7.3.3 Member Function Documentation

7.3.3.1 createUser()

```
void com.activitytracker.DBManager.createUser (
            final String name,
            final String emailAddress,
            final int DOBYear,
            final int DOBMonth,
            final int DOBDay,
            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
DOBYear	The year the user was born
DOBMonth	The month the user was born
DOBDay	The day of month 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 61 of file DBManager.java.

```
63                                                                                                                  {
64
65          if (!userExists(emailAddress)) {
66              String sqlQuery = "INSERT INTO Users (" +
67                      "email_address, " +
68                      "name, " +
69                      "date_of_birth, " +
70                      "sex, " +
71                      "height, " +
72                      "weight," +
73                      "password_hash," +
74                      "password_salt," +
75                      "created_at" +
76                      ") VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)";
77              byte sexByte = sex.equals(User.Sex.MALE) ? (byte) 1 : (byte) 0;
78              java.sql.Date currentTime = new java.sql.Date(System.currentTimeMillis());
79              Calendar c = Calendar.getInstance();
80              c.set(DOBYear, DOBMonth, DOBDay);
81              java.sql.Date dateOfBirth = new java.sql.Date(
82                      c.get(Calendar.YEAR),
83                      c.get(Calendar.MONTH),
84                      c.get(Calendar.DAY_OF_MONTH)
85              );
86
87              try {
88                  PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
89                  stmt.setString(1, emailAddress);
90                  stmt.setString(2, name);
91                  stmt.setDate(3, dateOfBirth);
92                  stmt.setByte(4, sexByte);
93                  stmt.setFloat(5, height);
94                  stmt.setFloat(6, weight);
95                  stmt.setString(7, securePassword.toString());
96                  stmt.setBytes(8, securePassword.getSalt());
97                  stmt.setDate(9, currentTime);
98
99                  if (stmt.executeUpdate() != 1) {
100                      System.err.println("User not added to database.");
101                  }
102
103                  stmt.close();
104              }
105              catch (final SQLException e) {
106                  System.err.println(e.getMessage());
```

```
107             }
108         }
109         else {
110             throw new AssertionError("User with email address '" + emailAddress + "' already exists.");
111         }
112     }
```

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 680 of file DBManager.java.

```
680                                                     {
681         ResultSet res = null;
682 
683         try {
684             Statement stmt = m_conn.createStatement();
685             res = stmt.executeQuery(sqlQuery);
686             stmt.close();
687         }
688         catch (final SQLException e) {
689             System.err.println(e.getMessage());
690         }
691 
692         return res;
693     }
```

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 708 of file DBManager.java.

```
708                                                         {
709         try {
710             Statement stmt = m_conn.createStatement();
711             stmt.executeUpdate(sqlQuery);
712             stmt.close();
713         }
714         catch (final SQLException e) {
715             System.err.println(e.getMessage());
716             return false;
717         }
718
719         return true;
720     }
```

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 298 of file DBManager.java.

```
298                                             {
299         Date DOB;
300         java.sql.Date DOBResult;
301         ResultSet res;
302         String sqlQuery = "SELECT date_of_birth FROM Users WHERE id=?";
303         try {
304             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
305             stmt.setInt(1, id);
306             res = stmt.executeQuery();
307             DOBResult = res.getDate("date_of_birth");
308 
309             stmt.close();
310         }
311         catch (final SQLException e) {
312             System.err.println(e.getMessage());
313             return null;
314         }
315         DOB = new Date(DOBResult.getYear(), DOBResult.getMonth(), DOBResult.getDay());
316 
317         return DOB;
318     }
```

7.3.3.5 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 585 of file DBManager.java.

```
585                                                                                 {
586         ResultSet res;
587         PreparedStatement stmt;
588         String sqlQuery, columnLabel;
589         float attrVal = 0.0f;
590         switch (attribute) {
591             case DURATION:
592                 columnLabel = "duration";
593                 sqlQuery = "SELECT " + columnLabel + " FROM Runs WHERE id=?";
594                 break;
595             case DISTANCE:
596                 columnLabel = "distance";
597                 sqlQuery = "SELECT " + columnLabel + " FROM Runs WHERE id=?";
598                 break;
599             case ALTITUDE_ASCENDED:
600                 columnLabel = "altitude_ascended";
601                 sqlQuery = "SELECT " + columnLabel + " FROM Runs WHERE id=?";
602                 break;
603             case ALTITUDE_DESCENDED:
604                 columnLabel = "altitude_descended";
605                 sqlQuery = "SELECT " + columnLabel + " FROM Runs WHERE id=?";
606                 break;
607             default:
608                 return attrVal;
609         }
610         if (runExists(rID)) {
611             try {
612                 stmt = m_conn.prepareStatement(sqlQuery);
613                 stmt.setInt(1, rID);
614                 res = stmt.executeQuery();
```

```
615                 attrVal = res.getFloat(columnLabel);
616             }
617             catch (final SQLException e) {
618                 System.err.println(e.getMessage());
619             }
620         }
621         else {
622             System.err.println("Run " + Integer.toString(rID) + " does not exist. Cannot get " +
      columnLabel + ".");
623         }
624 
625         return attrVal;
626 
627     }
```

7.3.3.6 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 257 of file DBManager.java.

```
257                                                                                 {
258         float attrVal;
259         ResultSet res;
260         String sqlQuery, columnLabel;
261         switch (attribute) {
262             case WEIGHT:
263                 columnLabel = "weight";
264                 sqlQuery = "SELECT " + columnLabel + " FROM Users WHERE id=?";
265                 break;
266             case HEIGHT:
267                 columnLabel = "height";
268                 sqlQuery = "SELECT " + columnLabel + " FROM Users WHERE id=?";
269                 break;
270             default:
271                 throw new AssertionError("Incorrect UserAttribute enumeration type passed to method.");
272         }
273         try {
274             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
275             stmt.setInt(1, id);
276             res = stmt.executeQuery();
277             attrVal = res.getFloat(columnLabel);
278 
279             stmt.close();
280         }
281         catch (final SQLException e) {
282             System.err.println(e.getMessage());
283             return 0.0f;
284         }
285 
286         return attrVal;
287     }
```

7.3.3.7 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 155 of file DBManager.java.

```
155                                                         {
156         int id = 0;
157         ResultSet res;
158         String sqlQuery = "SELECT id FROM Users WHERE `email_address`=?";
159         try {
160             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
161             stmt.setString(1, emailAddress);
162             res =  stmt.executeQuery();
163             id = res.getInt("id");
164 
165             stmt.close();
166         }
167         catch (final SQLException e) {
168             System.err.println(e.getMessage());
169         }
170 
171         return id;
172     }
```

7.3.3.8 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 395 of file DBManager.java.

```
395                                                 {
396         int rID = 0;
397         ResultSet res;
398         String columnLabel = "last_run";
399         String sqlQuery = "SELECT " + columnLabel + " FROM Users WHERE id=?";
400         try {
401             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
402             stmt.setInt(1, id);
403             res = stmt.executeQuery();
404             rID = res.getInt(columnLabel);
405             stmt.close();
406         }
407         catch (final SQLException e) {
408             System.err.println(e.getMessage());
409         }
410 
411         return rID;
412     }
```

7.3.3.9 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 366 of file DBManager.java.

```
366                                                 {
367         byte[] passSalt;
368         ResultSet res;
369         String sqlQuery = "SELECT password_salt FROM Users WHERE id=?";
```

```
370         try {
371             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
372             stmt.setInt(1, id);
373             res = stmt.executeQuery();
374             passSalt = res.getBytes("password_salt");
375             stmt.close();
376         }
377         catch (final SQLException e) {
378             System.err.println(e.getMessage());
379             return null;
380         }
381         return passSalt;
382     }
```

7.3.3.10 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 333 of file DBManager.java.

```
333                                             {
334         byte sex;
335         ResultSet res;
336         String sqlQuery = "SELECT sex FROM Users WHERE id=?";
337         try {
338             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
339             stmt.setInt(1, id);
340             res = stmt.executeQuery();
341             sex = res.getByte("sex");
342
```

```
343             stmt.close();
344         }
345         catch (final SQLException e) {
346             System.err.println(e.getMessage());
347             return null;
348         }
349 
350         if (sex == (byte) 1)
351             return User.Sex.MALE;
352         else
353             return User.Sex.FEMALE;
354     }
```

7.3.3.11 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 203 of file DBManager.java.

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

```
215                 break;
216             case EMAIL_ADDRESS:
217                 columnLabel = "email_address";
218                 sqlQuery = "SELECT " + columnLabel + " FROM Users WHERE id=?";
219                 break;
220             default:
221                 throw new AssertionError("Incorrect UserAttribute enumeration type passed to method.");
222         }
223         try {
224             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
225             stmt.setInt(1, id);
226             res = stmt.executeQuery();
227             name = res.getString(columnLabel);
228 
229             stmt.close();
230         }
231         catch (final SQLException e) {
232             System.err.println(e.getMessage());
233             return null;
234         }
235 
236         return name;
237     }
```

7.3.3.12 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 762 of file DBManager.java.

```
762                                     {
763         try {
764             m_conn = DriverManager.getConnection("jdbc:sqlite:" + dbURL);
765         }
766         catch (final SQLException e) {
767             System.err.println(e.getMessage());
768             return false;
769         }
770         System.out.println("Opened database successfully.");
771 
772         if (isEmpty()) {
773             System.out.println("Creating tables...");
774 
775             // Create users table
776             String sqlQuery = "CREATE TABLE USERS (" +
777                     "    id            INTEGER PRIMARY KEY ASC AUTOINCREMENT NOT NULL," +
778                     "    email_address STRING  NOT NULL UNIQUE ON CONFLICT FAIL," +
779                     "    name          STRING  NOT NULL," +
780                     "    date_of_birth DATE    NOT NULL," +
781                     "    sex           BIT(1)  NOT NULL," +
782                     "    height        REAL    NOT NULL," +
783                     "    weight        REAL    NOT NULL," +
784                     "    password_hash STRING  NOT NULL," +
785                     "    password_salt BLOB    NOT NULL," +
786                     "    last_run  INTEGER NOT NULL DEFAULT 0," +
787                     "    created_at    DATE    NOT NULL" +
788                     ")";
789 
790             if (!executeUpdate(sqlQuery)) {
791                 return false;
792             }
793 
794             // Create workouts table
795             sqlQuery = "CREATE TABLE RUNS (" +
796                     "    id                  INTEGER PRIMARY KEY ASC AUTOINCREMENT NOT NULL," +
797                     "    user_id             INTEGER NOT NULL REFERENCES USERS (id)," +
798                     "    date                DATE    NOT NULL," +
799                     "    duration            REAL    NOT NULL," + // seconds
800                     "    distance            REAL    NOT NULL," + // metres
801                     "    altitude_ascended   REAL    NOT NULL," + // metres
802                     "    altitude_descended  REAL    NOT NULL" + // metres
803                     ")";
804 
805             if (!executeUpdate(sqlQuery)) {
806                 return false;
807             }
808 
809             // Create friends table
810             sqlQuery = "CREATE TABLE FRIENDS (" +
811                     "    id            INTEGER  PRIMARY KEY ASC AUTOINCREMENT NOT NULL," +
812                     "    sender        INTEGER  NOT NULL REFERENCES USERS (id)," +
813                     "    receiver      INTEGER  REFERENCES USERS (id)," +
814                     "    send_date     DATETIME NOT NULL," +
815                     "    confirm_date  DATETIME DEFAULT NULL" +
816                     ")";
817 
818             if (!executeUpdate(sqlQuery)) {
819                 return false;
820             }
821 
822         }
823 
824         return true;
825     }
```

7.3.3.13 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 730 of file DBManager.java.

```
730                                 {
731
732         try {
733             final DatabaseMetaData dbmd = m_conn.getMetaData();
734             final String[] types = {"TABLE"};
735             final ResultSet rs = dbmd.getTables(null, null, "%", types);
736
737             return !rs.next();
738         }
739         catch (final SQLException e) {
740             System.err.println(e.getMessage());
741             return true;
742         }
743
744     }
```

7.3.3.14 newRun()

```
int com.activitytracker.DBManager.newRun (
            final int userID,
            final int year,
            final int month,
            final int day,
            final float duration,
            final float distance,
            final float altitude_ascended,
            final float altitude_descended )
```

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.
year	Year that the run was completed.
month	Month that the run was completed (1-12).
day	Day that the run was completed (1-31).
duration	Duration of the run in seconds.
distance	Distance ran in metres.
altitude_ascended	Cumulative altitude climbed in metres.
altitude_descended	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 459 of file DBManager.java.

```
461                                                         {
462         java.sql.Date RDate = new java.sql.Date(year, month, day);
463         int rID = 0;
464         ResultSet res;
465         String sqlInsertQuery = "INSERT INTO Runs (" +
466                 "user_id," +
467                 "date," +
468                 "duration," +
469                 "distance," +
470                 "altitude_ascended," +
471                 "altitude_descended" +
472                 ") VALUES (?, ?, ?, ?, ?, ?)";
473         String sqlSelectQuery = "SELECT id FROM Runs WHERE " +
474                 "user_id=? AND " +
475                 "date=? AND " +
476                 "duration=? AND " +
477                 "distance=? AND " +
478                 "altitude_ascended=? AND " +
479                 "altitude_descended=?";
480 
481         try {
482             PreparedStatement stmt = m_conn.prepareStatement(sqlInsertQuery);
483             stmt.setInt(1, userID);
484             stmt.setDate(2, RDate);
485             stmt.setFloat(3, duration);
486             stmt.setFloat(4, distance);
487             stmt.setFloat(5, altitude_ascended);
488             stmt.setFloat(6, altitude_descended);
489 
490             if (stmt.executeUpdate() != 1) {
491                 System.err.println("Run not added to database.");
492             }
493 
494             stmt.close();
495 
496             // Pass back in the stuff we just created to get the right row ID
497             // Look at a better way of doing this with OUTPUT clause of INPUT statement
```

```
498             stmt = m_conn.prepareStatement(sqlSelectQuery);
499             stmt.setInt(1, userID);
500             stmt.setDate(2, RDate);
501             stmt.setFloat(3, duration);
502             stmt.setFloat(4, distance);
503             stmt.setFloat(5, altitude_ascended);
504             stmt.setFloat(6, altitude_descended);
505 
506             res = stmt.executeQuery();
507             rID = res.getInt("id");
508         }
509         catch (final SQLException e) {
510             System.err.println(e.getMessage());
511         }
512 
513         return rID;
514     }
```

7.3.3.15 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 636 of file DBManager.java.

```
636                                             {
637         ResultSet res;
638         String sqlQuery = "SELECT COUNT(*) as count FROM Runs WHERE id=?";
639         boolean exists = false;
640         try {
641             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
642             stmt.setInt(1, rID);
643             res = stmt.executeQuery();
644             switch (res.getInt("count")) {
645                 case 0:
646                     exists = false;
647                     break;
648                 case 1:
```

```
649                     exists = true;
650                     break;
651                 default:
652                     exists = true;
653                     System.err.println("More than one run for ID " +
654                             Integer.toString(rID) + ". Something isn't right.");
655                     break;
656             }
657 
658         }
659         catch (final SQLException e) {
660             System.err.println(e.getMessage());
661         }
662 
663         return exists;
664     }
```

7.3.3.16 setRun()

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

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.
altitude_ascended	The cumulative number of metres the user climbed.
altitude_descended	The cumulative number of metres the user descended.

Definition at line 533 of file DBManager.java.

```
534                                                                             {
```

```
535         String sqlQuery = "UPDATE Runs SET " +
536                 "duration = ?, " +
537                 "distance = ?, " +
538                 "altitude_ascended=?, " +
539                 "altitude_descended=? " +
540                 "WHERE id=? ";
541         try {
542             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
543             stmt.setFloat(1, duration);
544             stmt.setFloat(2, distance);
545             stmt.setFloat(3, altitude_ascended);
546             stmt.setFloat(4, altitude_descended);
547             stmt.setInt(5, rID);
548 
549             int result = stmt.executeUpdate();
550             System.err.println(Integer.toString(result) + " rows updated in setRun().");
551             if (result != 1) {
552                 System.err.println("Run not updated in database.");
553             }
554 
555             stmt.close();
556         }
557         catch (final SQLException e) {
558             System.err.println(e.getMessage());
559         }
560 
561     }
```

7.3.3.17 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 426 of file DBManager.java.

```
426                                                                 {
427         String sqlQuery = "UPDATE Users SET last_run=? WHERE id=?";
428         try {
429             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
430             stmt.setInt(1, lastRID);
431             stmt.setInt(2, id);
432             if (stmt.executeUpdate() != 1) {
433                 System.err.println("User's last run was not updated correctly.");
434             }
435         }
436         catch (final SQLException e) {
437             System.err.println(e.getMessage());
438         }
439     }
```

7.3.3.18 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 124 of file DBManager.java.

```
124                                                                 {
125         String sqlQuery = "SELECT COUNT(*) AS count FROM Users WHERE `email_address`=?";
126         boolean exists = false;
127 
128         try {
129             PreparedStatement stmt = m_conn.prepareStatement(sqlQuery);
130             stmt.setString(1, emailAddress);
131             ResultSet res = stmt.executeQuery();
132             exists = res.getInt("count") > 0;
133 
```

```
134             stmt.close();
135         }
136         catch (final SQLException e) {
137             System.err.println(e.getMessage());
138         }
139 
140         return exists;
141     }
```

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.

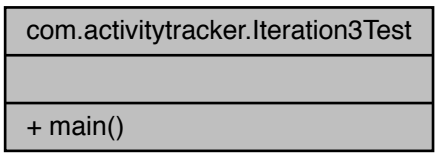
Definition at line 32 of file DBManager.java.

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 8 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 10 of file Iteration3Test.java.

```
10                                          {
11
12          // Iteration 1 begins here
13
14          User john = null;
15
16          DBManager dbManager = new DBManager();
17          if (!dbManager.init("data.db")) {
18              System.err.println("Failed to initialize DBManager");
19              System.exit(1);
20          }
21
22          System.out.println("Attempting to create user...");
23
24          if (!dbManager.userExists("jdoe@mac.com"))
25              User.createUser(
26                      dbManager,
27                      "John Doe",
28                      "jdoe@mac.com",
29                      1997,
30                      12,
31                      12,
32                      User.Sex.MALE,
33                      1.6764f,
34                      54.4310844f,
35                      "My Very Secure Password"
36              );
37          else
38              System.out.println("User already exists.");
39
40
41
```

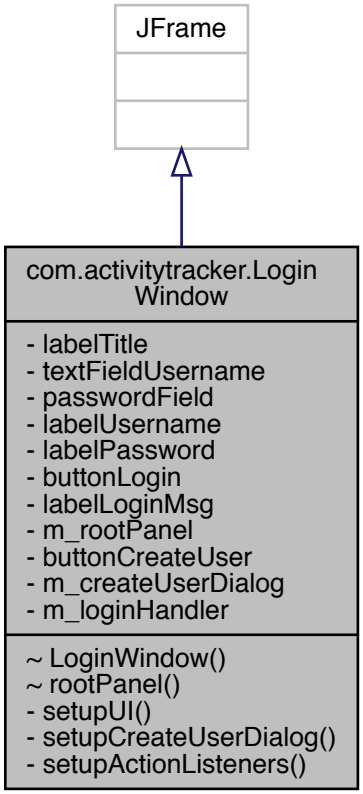
```
42          if (dbManager.userExists("jdoe@mac.com"))
43              System.out.println("John Doe was created!");
44          else
45              System.out.println("User was NOT created.");
46
47
48          System.out.println("Testing incorrect password...");
49
50          try {
51              john = new User(dbManager,"jdoe@mac.com", "Some Incorrect Password");
52          }
53          catch (final AuthenticationException e) {
54              System.out.println("Incorrect password used; authentication failed.");
55          }
56
57          System.out.println("Authenticating user...");
58
59          try {
60              john = new User(dbManager,"jdoe@mac.com", "My Very Secure Password");
61          }
62          catch (final AuthenticationException e) {
63              System.out.println("Test failed; user could not be authenticated.");
64          }
65
66          // Iteration 1 ended here
67
68          // Iteration 2 begins here
69
70          if (john !=  null) {
71              Date today = new Date();
72              try {
73                  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");
74              }
75              catch (final IOException e) {
76                  System.err.println(e.getMessage());
77              }
78          }
79          else {
80              System.out.println("John is null. Cannot execute phase 2.");
81          }
82
83          // Iteration 2 ends here
84
85      }
```

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

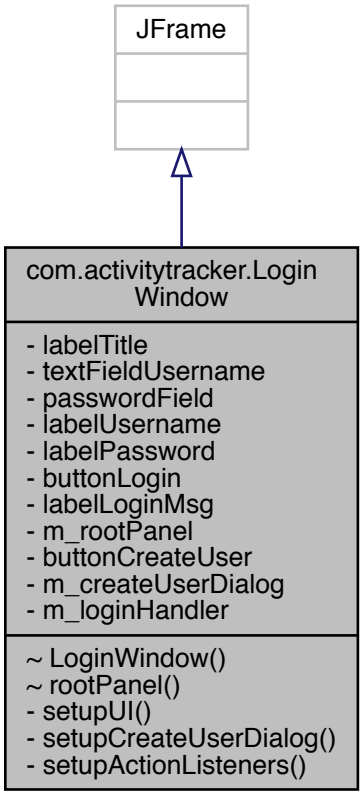
- app/src/com/activitytracker/Iteration3Test.java

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)
- 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
- 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 ) [package]
```

Definition at line 26 of file LoginWindow.java.

```
26                                                                  {
27         m_loginHandler = loginHandler;
28
29         setupUI();
30         setupCreateUserDialog();
31         setupActionListeners();
32     }
```

7.5.3 Member Function Documentation

7.5.3.1 rootPanel()

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

Definition at line 85 of file LoginWindow.java.

```
85                          {
86         return m_rootPanel;
87     }
```

7.5.3.2 setupActionListeners()

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

Definition at line 54 of file LoginWindow.java.

```
54                                          {
55
56         // Login button
57         buttonLogin.addActionListener(new ActionListener() {
58             @Override
59             public void actionPerformed(ActionEvent e) {
60
61                 // Do nothing if login fields are empty
62                 if (textFieldUsername.getText().isEmpty() ||
       passwordField.getPassword().length == 0) {
63                     return;
64                 }
65
66                 // Change to verifyLogin()
67                 if (true) {
68                     m_loginHandler.accept(null);
69                     return;
70                 }
71
72                 // Display error message
73             }
74         });
75
76         // Create user button
77         buttonCreateUser.addActionListener(new ActionListener() {
78             @Override
79             public void actionPerformed(ActionEvent e) {
80                 m_createUserDialog.setVisible(true);
81             }
82         });
83     }
```

7.5.3.3 setupCreateUserDialog()

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

Definition at line 41 of file LoginWindow.java.

```
41                                          {
42
43         // Get desktop resolution of default monitor (in case of multi-monitor setups)
44         final GraphicsDevice gd = GraphicsEnvironment.getLocalGraphicsEnvironment().getDefaultScreenDevice(
       );
45
46         m_createUserDialog = new JDialog(this, "Activity Logger | Create User", true);
47         m_createUserDialog.setContentPane(new CreateUserWindow().
       rootPanel());
48         m_createUserDialog.pack();
49         // Set window size to be 1/2 of screen dimensions
50         m_createUserDialog.setSize(gd.getDisplayMode().getWidth() / 2, gd.getDisplayMode(
       ).getHeight() / 2);
51         m_createUserDialog.setLocationRelativeTo(this); // Center window
52     }
```

7.5.3.4 setupUI()

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

Definition at line 34 of file LoginWindow.java.

```
34                              {
35         MaterialUIMovement.add(buttonLogin, MaterialColors.GRAY_100);
36         MaterialUIMovement.add(buttonCreateUser, MaterialColors.GRAY_100);
37
38         labelLoginMsg.setVisible(false);
39     }
```

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_loginHandler

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

Definition at line 24 of file LoginWindow.java.

7.5.4.9 m_rootPanel

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

Definition at line 19 of file LoginWindow.java.

7.5.4.10 passwordField

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

Definition at line 14 of file LoginWindow.java.

7.5.4.11 textFieldUsername

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

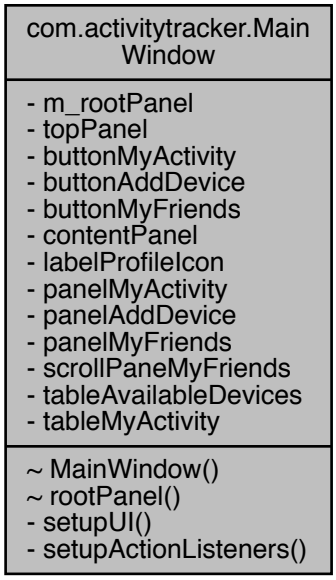
Definition at line 13 of file LoginWindow.java.

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 ()
- JPanel rootPanel ()

Private Member Functions

- void setupUI ()
- void setupActionListeners ()

Private Attributes

- JPanel m_rootPanel
- JPanel topPanel
- JButton buttonMyActivity
- JButton buttonAddDevice
- JButton buttonMyFriends
- JPanel contentPanel
- JLabel labelProfileIcon
- JPanel panelMyActivity
- JPanel panelAddDevice
- JPanel panelMyFriends
- JScrollPane scrollPaneMyFriends
- JTable tableAvailableDevices
- JTable tableMyActivity

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 ( ) [package]`

Definition at line 27 of file MainWindow.java.

```
27                 {
28         setupUI();
29         setupActionListeners();
30     }
```

7.6.3 Member Function Documentation

7.6.3.1 rootPanel()

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

Definition at line 84 of file MainWindow.java.

```
84                          {
85         return m_rootPanel;
86     }
```

7.6.3.2 setupActionListeners()

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

Definition at line 54 of file MainWindow.java.

```
54                                          {
55         // My Activity button
56         buttonMyActivity.addActionListener(new ActionListener() {
57             @Override
58             public void actionPerformed(ActionEvent actionEvent) {
59                 panelMyActivity.setVisible(true);
60                 panelAddDevice.setVisible(false);
61                 panelMyFriends.setVisible(false);
62             }
63         });
64         // Add Device button
65         buttonAddDevice.addActionListener(new ActionListener() {
66             @Override
67             public void actionPerformed(ActionEvent actionEvent) {
68                 panelMyActivity.setVisible(false);
69                 panelAddDevice.setVisible(true);
70                 panelMyFriends.setVisible(false);
71             }
72         });
73         // My Friends button
74         buttonMyFriends.addActionListener(new ActionListener() {
75             @Override
76             public void actionPerformed(ActionEvent actionEvent) {
77                 panelMyActivity.setVisible(false);
78                 panelAddDevice.setVisible(false);
79                 panelMyFriends.setVisible(true);
80             }
81         });
82     }
```

7.6.3.3 setupUI()

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

Definition at line 32 of file MainWindow.java.

```
32                          {
33
34         // Apply Material-defined hover effect to buttons
35         Color coolGrey10 = new Color(99, 102, 106);
36         Color coolGrey11 = new Color(83, 86, 90);
37         MaterialUIMovement.add(buttonMyActivity, coolGrey11);
38         MaterialUIMovement.add(buttonAddDevice, coolGrey11);
39         MaterialUIMovement.add(buttonMyFriends, coolGrey11);
40
41         // Load and scale logo into UI
42         String logoPath = "./assets/logo.png";
43         ImageIcon imageIcon = new ImageIcon(getClass().getResource(logoPath));
44         final Image image = imageIcon.getImage(); // transform it
45         final Image newimg = image.getScaledInstance(50, 50, java.awt.Image.SCALE_SMOOTH);
46         imageIcon = new ImageIcon(newimg);  // transform it back
47         labelProfileIcon.setIcon(imageIcon);
48
49         panelMyActivity.setVisible(true);
50         panelAddDevice.setVisible(false);
51         panelMyFriends.setVisible(false);
52     }
```

7.6.4 Member Data Documentation

7.6.4.1 buttonAddDevice

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

Definition at line 16 of file MainWindow.java.

7.6.4.2 buttonMyActivity

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

Definition at line 15 of file MainWindow.java.

7.6.4.3 buttonMyFriends

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

Definition at line 17 of file MainWindow.java.

7.6.4.4 contentPanel

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

Definition at line 18 of file MainWindow.java.

7.6.4.5 labelProfileIcon

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

Definition at line 19 of file MainWindow.java.

7.6.4.6 m_rootPanel

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

Definition at line 13 of file MainWindow.java.

7.6.4.7 panelAddDevice

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

Definition at line 21 of file MainWindow.java.

7.6.4.8 panelMyActivity

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

Definition at line 20 of file MainWindow.java.

7.6.4.9 panelMyFriends

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

Definition at line 22 of file MainWindow.java.

7.6.4.10 scrollPaneMyFriends

```
JScrollPane com.activitytracker.MainWindow.scrollPaneMyFriends [private]
```

Definition at line 23 of file MainWindow.java.

7.6.4.11 tableAvailableDevices

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

Definition at line 24 of file MainWindow.java.

7.6.4.12 tableMyActivity

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

Definition at line 25 of file MainWindow.java.

7.6.4.13 topPanel

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

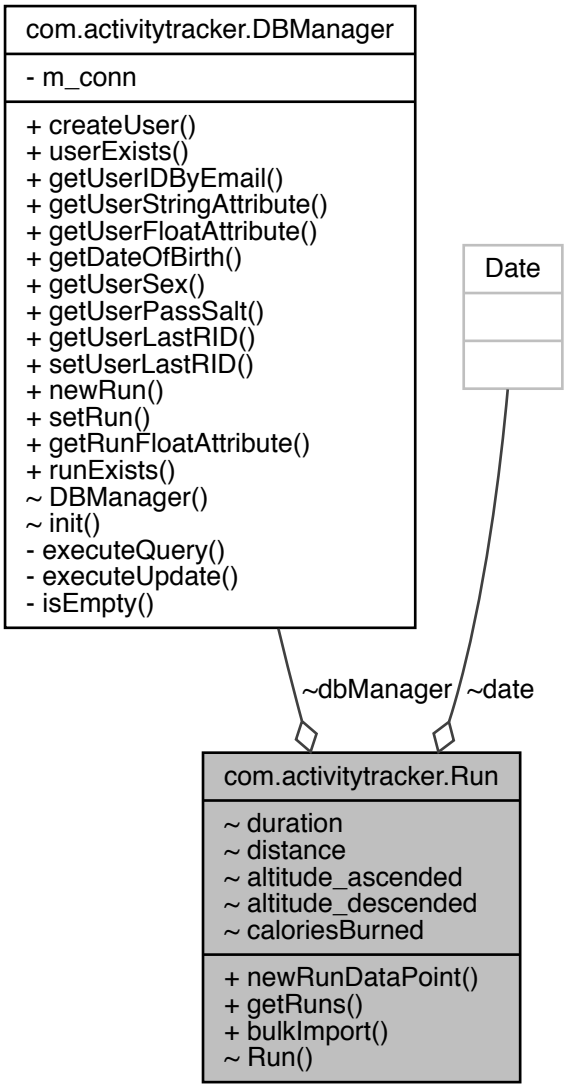
Definition at line 14 of file MainWindow.java.

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:



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

- Date date
- DBManager dbManager
- float duration
- float distance
- float altitude_ascended
- float altitude_descended
- long caloriesBurned = 0

7.7.1 Detailed Description

Used to logically instantiate a run.

Definition at line 13 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 52 of file Run.java.

```
52                                                  {
53         this.dbManager = dbManager;
54         this.duration = this.dbManager.getRunFloatAttribute(
      RunAttribute.DURATION, rID);
55         this.distance = this.dbManager.getRunFloatAttribute(
      RunAttribute.DISTANCE, rID);
56         this.altitude_ascended = this.dbManager.
      getRunFloatAttribute(RunAttribute.ALTITUDE_ASCENDED, rID);
57         this.altitude_descended = this.dbManager.
      getRunFloatAttribute(RunAttribute.ALTITUDE_DESCENDED, rID);
58     }
```

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 158 of file Run.java.

```
158                                     {
159         BufferedReader br = new BufferedReader(new FileReader(filePath));
160         String line = null;
161         while ((line = br.readLine()) != null)
162         {
163             String[] attributes = line.split(",");
164             String buffTime = attributes[0];
165             String buffDistance = attributes[1];
166             String buffAltitude = attributes[2];
167             String[] buffDate = attributes[3].split("-");
168             String buffMonth = buffDate[1];
169             String buffDay = buffDate[0];
170             String buffYear = buffDate[2];
171             Date runDate = new Date(Integer.parseInt(buffDay), Integer.parseInt(buffMonth), Integer.
    parseInt(buffYear));
172 
173             // Convert strings to floats
174             // ToDo: String date to Date date
175             float fDur = Float.parseFloat(buffTime);
176             float fDist = Float.parseFloat(buffDistance);
177             float fAlt = Float.parseFloat(buffAltitude);
178 
179             newRunDataPoint(dbManager, user, fDur, runDate, fDist, fAlt);
180         }
181     }
```

7.7.3.2 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 139 of file Run.java.

```
139
                    {
140         Vector<Run> runs = null;
141
142         return runs;
143     }
```

7.7.3.3 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 84 of file Run.java.

```
85                                                                                                  {
86         int userID = user.getID();
87         int rID;
88         float altitude_ascended;
89         float altitude_descended;
90 
91         if (duration == 0f && distance == 0f && altitude == 0f) {
92             altitude_ascended = 0f;
93             altitude_descended = 0f;
94             rID = dbManager.newRun(
95                     userID,
96                     date.getYear(),
97                     date.getMonth(),
98                     date.getDay(),
99                     duration,
100                     distance,
101                     altitude_ascended,
102                     altitude_descended
103             );
104             user.setLastRID(rID);
105             System.err.println("Run " + Integer.toString(rID) + " added to database.");
106         } else {
107             rID = user.getLastRID();
108             if (dbManager.runExists(rID)) {
109                 altitude_ascended = dbManager.
    getRunFloatAttribute(RunAttribute.ALTITUDE_ASCENDED, rID);
110                 altitude_descended = dbManager.
    getRunFloatAttribute(RunAttribute.ALTITUDE_DESCENDED, rID);
111 
112                 if (altitude < 0)
113                     altitude_descended += -1*altitude;
114                 else
115                     altitude_ascended += altitude;
116 
117                 dbManager.setRun(rID, duration, distance,
    altitude_ascended, altitude_descended);
118                 System.err.println("Run " + Integer.toString(rID) + " exists in the database; updating...")
    ;
119             } else {
120                 System.err.println("Run table and User table are inconsistent. No changes made.");
121             }
122         }
123     }
```

7.7.4 Member Data Documentation

7.7.4.1 altitude_ascended

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

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

Definition at line 33 of file Run.java.

7.7.4.2 altitude_descended

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

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

Definition at line 37 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 43 of file Run.java.

7.7.4.4 date

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

The date the run occurred.

Definition at line 17 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 21 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 29 of file Run.java.

7.7.4.7 duration

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

The length of the run in seconds.

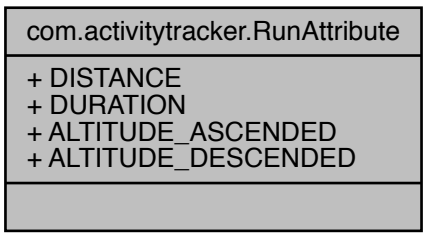
Definition at line 25 of file Run.java.

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

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.

Definition at line 24 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.

Definition at line 30 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.

Definition at line 12 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.

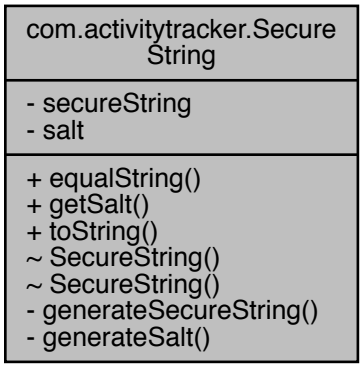
Definition at line 18 of file RunAttribute.java.

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

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

7.9 com.activitytracker.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

7.9.1 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.

7.9.2 Constructor & Destructor Documentation

7.9.2.1 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     }
```

7.9.2.2 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     }
```

7.9.3 Member Function Documentation

7.9.3.1 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     }
```

7.9.3.2 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     }
```

7.9.3.3 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     }
```

7.9.3.4 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     }
```

7.9.3.5 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     }
```

7.9.4 Member Data Documentation

7.9.4.1 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.

7.9.4.2 secureString

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

The encrypted string.

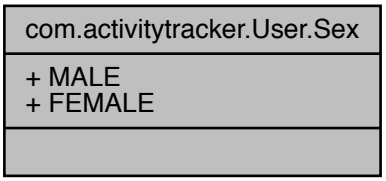
Definition at line 15 of file SecureString.java.

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

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

7.10 com.activitytracker.User.Sex Enum Reference

Collaboration diagram for com.activitytracker.User.Sex:



Public Attributes

- MALE
- FEMALE

7.10.1 Detailed Description

Used to represent whether the user is male or female.

Definition at line 12 of file User.java.

7.10.2 Member Data Documentation

7.10.2.1 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.

7.10.2.2 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*.

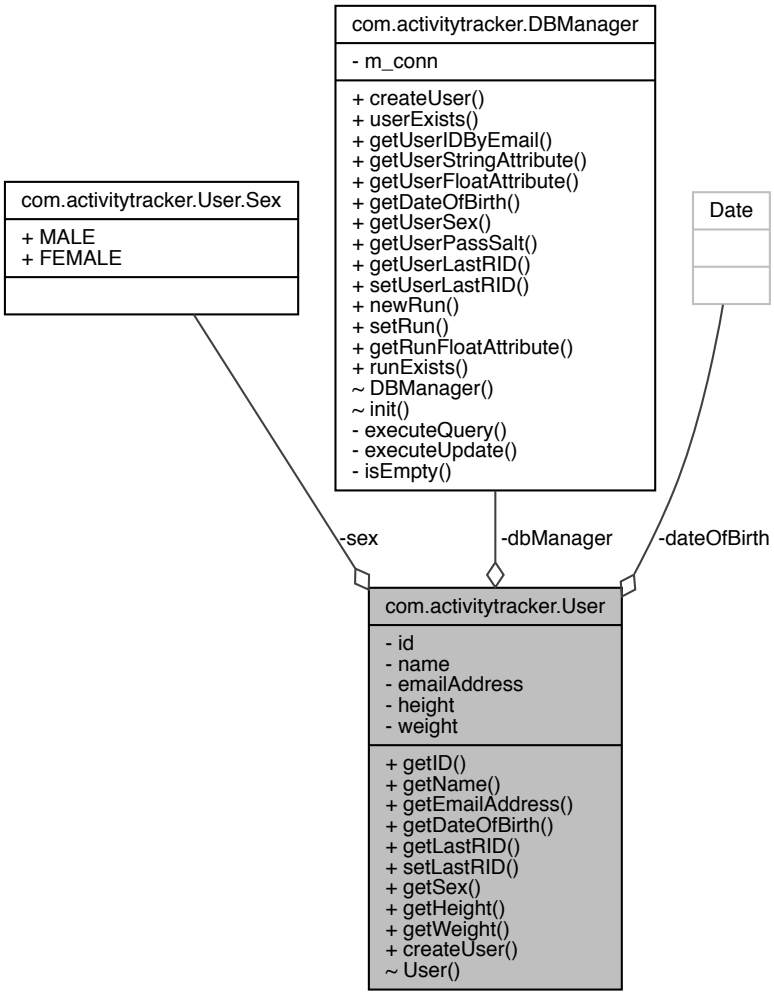
Definition at line 19 of file User.java.

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

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

7.11 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 int DOBYear, final int DOBMonth, final int DOBDay, 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

7.11.1 Detailed Description

Definition at line 8 of file User.java.

7.11.2 Constructor & Destructor Documentation

7.11.2.1 User()

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

Definition at line 38 of file User.java.

```
38
                        {
39
40          this.dbManager = dbManager;
41          if (this.dbManager.userExists(emailAddress)) {
42
43              this.id = dbManager.getUserIDByEmail(
      emailAddress);
44
45              String passHash = this.dbManager.getUserStringAttribute(
      UserAttribute.PASSWORD, this.id);
46              byte[] passSalt = this.dbManager.getUserPassSalt(this.id);
47
48              SecureString candidatePassword = new SecureString(plaintextPassword, passSalt);
49
50              if (candidatePassword.equalString(passHash)) {
51
52
53                  this.name = this.dbManager.getUserStringAttribute(
      UserAttribute.NAME, this.id);
54 //                 this.emailAddress = this.dbManager.getEmailAddress(this.id);
55                  this.emailAddress = emailAddress;
56                  this.dateOfBirth = this.dbManager.
      getDateOfBirth(this.id);
57                  this.sex = this.dbManager.getUserSex(this.id);
58                  this.height = this.dbManager.
      getUserFloatAttribute(UserAttribute.HEIGHT, this.id);
59                  this.weight = this.dbManager.
      getUserFloatAttribute(UserAttribute.WEIGHT, this.id);
60
61                  System.out.println("Authentication succeeded for " + this.name);
62
63              }
64              else {
65
66                  throw new AuthenticationException("Incorrect password.");
67
68              }
69          }
70          else {
71
72              throw new NoSuchElementException("No such user exists.");
73
74          }
75
76      }
```

7.11.3 Member Function Documentation

7.11.3.1 createUser()

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

Definition at line 78 of file User.java.

```
80                                                                                  {
81
82          SecureString securePassword = new SecureString(plaintextPassword);
83
84
85          dbManager.createUser(
86                  name,
87                  emailAddress,
88                  DOBYear,
89                  DOBMonth,
90                  DOBDay,
91                  sex,
92                  height,
93                  weight,
94                  securePassword
95          );
96
97      }
```

7.11.3.2 getDateOfBirth()

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

Definition at line 111 of file User.java.

```
111                                 {
112          return this.dateOfBirth;
113      }
```

7.11.3.3 getEmailAddress()

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

Definition at line 107 of file User.java.

```
107                                     {
108         return this.emailAddress;
109     }
```

7.11.3.4 getHeight()

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

Definition at line 123 of file User.java.

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

7.11.3.5 getID()

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

Definition at line 99 of file User.java.

```
99                          {
100         return this.id;
101     }
```

7.11.3.6 getLastRID()

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

Definition at line 115 of file User.java.

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

7.11.3.7 getName()

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

Definition at line 103 of file User.java.

```
103                             {
104         return this.name;
105     }
```

7.11.3.8 getSex()

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

Definition at line 119 of file User.java.

```
119                         {
120         return this.sex;
121     }
```

7.11.3.9 getWeight()

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

Definition at line 127 of file User.java.

```
127                             {
128         return this.weight;
129     }
```

7.11.3.10 setLastRID()

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

Definition at line 117 of file User.java.

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

7.11.4 Member Data Documentation

7.11.4.1 dateOfBirth

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

Definition at line 32 of file User.java.

7.11.4.2 dbManager

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

Definition at line 36 of file User.java.

7.11.4.3 emailAddress

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

Definition at line 31 of file User.java.

7.11.4.4 height

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

Definition at line 34 of file User.java.

7.11.4.5 id

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

Definition at line 29 of file User.java.

7.11.4.6 name

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

Definition at line 30 of file User.java.

7.11.4.7 sex

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

Definition at line 33 of file User.java.

7.11.4.8 weight

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

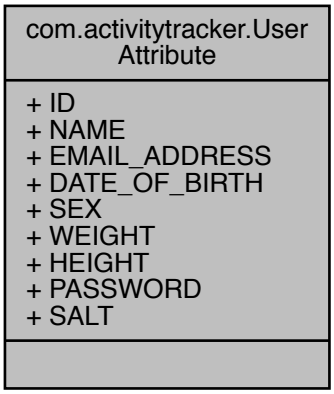
Definition at line 35 of file User.java.

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

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

7.12 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

7.12.1 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.

7.12.2 Member Data Documentation

7.12.2.1 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.

7.12.2.2 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.

7.12.2.3 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.

7.12.2.4 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.

7.12.2.5 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.

7.12.2.6 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.

7.12.2.7 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.

7.12.2.8 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.

7.12.2.9 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/SecureString.java File Reference

Classes

- class com.activitytracker.SecureString

Packages

- package com.activitytracker

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

Classes

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

Packages

- package com.activitytracker

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

Classes

- enum com.activitytracker.UserAttribute

Packages

- package com.activitytracker

Index