

1. **import** java.sql.ResultSet;
2. **import** java.sql.SQLException;
3. **import** java.sql.\*;
4. **import** java.util.ArrayList;
6. **public** **class** Dao
7. {
8. **private** **static** **final** String URL = "jdbc:mysql://www.papademas.net:3306/411labs?autoReconnect=true&useSSL=false";
9. **private** **static** **final** String USERNAME = "db411";
10. **private** **static** **final** String PASSWORD = "411";
11. **public**  **static** **final** String TABLE\_NAME = "H\_Post\_tab";
12. **private** **static** **final** **int**    MAX\_RESULTS = 600;
14. **private** **static** **final** String columns[] =
15. {"PID", "ID", "Age", "Gender",
16. "Region", "Income", "Married","Children",
17. "Car", "Savings", "Current", "Mortgage",
18. "Pep" };
20. **private** **static** **final** String columnTypes[] =
21. {"INTEGER","VARCHAR(30)","INTEGER","VARCHAR(10)",
22. "VARCHAR(30)","DOUBLE(10,4)","VARCHAR(10)","INTEGER",
23. "VARCHAR(10)","VARCHAR(10)","VARCHAR(10)","VARCHAR(10)",
24. "VARCHAR(10)"};
25. **private** **static** **final** String columnModifiers[] =
26. {"NOT NULL UNIQUE AUTO\_INCREMENT","","","",
27. "","","","",
28. "","","","",
29. ""};
31. **public** **static** ArrayList<String> command\_history = **new** ArrayList<>();
33. **private** **static** ResultSet execute\_print\_query(String string, Statement s)
34. {
35. command\_history.add(string);
36. System.out.print(string.replace('\n', ' ')+"\n");
37. Boolean wasDataManip = **null**;
39. **try**
40. {
41. **return** s.executeQuery(string);
42. }
43. **catch** (SQLException e)
44. {
45. **try**
46. {
47. System.out.println("Instead of using executeQuery(), we are going to use execute().");
48. wasDataManip = s.execute(string);
49. System.out.println("Using execute succeeded! Ignore other errors.\n");
50. }
51. **catch** (Exception e1)
52. {
53. System.out.println("Darn.\nData manipulation query results in: "+ wasDataManip);
54. e1.printStackTrace();
55. }
56. **finally**
57. {
58. **if**(wasDataManip == **null**)
59. {
60. System.out.println("Executing query \'" + string + "\' failed!");
61. e.printStackTrace();
62. }
63. }
64. }
65. **return** **null**;
67. }
69. **private** **static** **final** String NO\_SAFE\_UPDATES = "SET SQL\_SAFE\_UPDATES = 1";
71. **private** **static** Connection conn = **null**;
73. **public** Dao()
74. {
75. **try**
76. {
77. conn = DriverManager.getConnection(URL,USERNAME,PASSWORD);
78. }
79. **catch** (Exception e)
80. {
81. System.out.println("Exception happened for URL \'" + URL + "\'");
82. //          System.out.println("Exception is: "+e);
83. e.printStackTrace();
84. }
85. }
87. **public** **static** **int** getLargestColumn(ResultSet rs)
88. {
89. **int** largest = -1;
90. ResultSetMetaData rsmd = **null**;
91. **try**
92. {
93. rsmd = rs.getMetaData();

96. **for** (**int** i = 1; i < rsmd.getColumnCount(); i++)
97. {
98. **if**(rsmd.getColumnDisplaySize(i) > largest)
99. {
100. largest = rsmd.getColumnDisplaySize(i);
101. }
102. }
104. } **catch** (SQLException e)
105. {
106. e.printStackTrace();
107. }
109. **return** largest;

112. }


116. **public** **static** **void** printResultSet(ArrayList<String> ar)
117. {
118. **for**(**int** i = 0; i < ar.size(); i++)
119. {
120. System.out.println(ar.get(i));
121. }
122. **return**;
123. }

126. **public** **static** ArrayList<String> formatResultSet(ResultSet rs)
127. {
128. **int** largestColumn = getLargestColumn(rs);
129. String stringFormat =   "%" + 13 + "s |";
130. String stringFormat2 =  "%" + 15 + "s |";
132. ArrayList<String> ret = **new** ArrayList<>();
134. ResultSetMetaData rsmd;
135. **try**
136. {
137. rsmd = rs.getMetaData();
138. ret.add(String.format(stringFormat2, "Column name:"));
139. ret.add(String.format(stringFormat2, "Column type:"));

142. //          System.out.println("metadata:");
143. **for**(**int** i = 1; i < rsmd.getColumnCount(); i++)
144. {
145. //              printf("i = %2d, current thing = %10s, %10s\n",
146. //                      i,rsmd.getColumnName(i),rsmd.getColumnTypeName(i));
147. ret.set(0,
148. ret.get(0) + String.format(stringFormat,rsmd.getColumnName(i)) //tack on a name
149. );
151. ret.set(1,
152. ret.get(1) + String.format(stringFormat,rsmd.getColumnTypeName(i)) //tack on a type
153. );
154. }
156. //          System.out.println("length of my lines should be " + ret.get(0).length());
157. ret.add(stringLine("-",ret.get(0).length()));
158. ret.add(0, stringLine("\_",ret.get(0).length()));
160. **while**(rs.next())
161. {
162. String oneColumn = String.format(stringFormat2,"");
163. **for**(**int** i = 1; i < rsmd.getColumnCount(); i++)
164. {
165. //                  System.out.println("HI. ONECOLUMN = "+oneColumn);
166. oneColumn += String.format(stringFormat, rs.getString(i));
167. }
169. ret.add(oneColumn);
170. }

173. }
174. **catch** (SQLException e)
175. {
176. e.printStackTrace();
177. }


181. **return** ret;
182. }

185. **private** **static** String stringLine(String duplicate, **int** length)
186. {
187. String ret = "";
189. **for** (**int** i = 0; i < length ; i++)
190. {
191. ret += duplicate;
192. }
193. **return** ret;
194. }
196. **public** **static** **void** createTables()
197. {
198. Statement s = **null**;
199. String query = "CREATE TABLE IF NOT EXISTS " + TABLE\_NAME + "\n(";

202. **try**
203. {
204. s = conn.createStatement();
206. System.out.print("Executing this:");
207. execute\_print\_query(NO\_SAFE\_UPDATES,s);


211. **for** (**int** i = 0; i < columns.length; i++)
212. {
213. query += columns[i] + " " + columnTypes[i] + " " + columnModifiers[i];
215. **if**(i != columns.length-1)
216. {
217. query += ",\n"; //not the end, add a comma
218. }
219. **else**
220. {
221. query += ");";
222. }
223. }

226. System.out.println("executing this:");
227. execute\_print\_query(query,s);
229. query = "DESCRIBE " + TABLE\_NAME;
231. ResultSet r = execute\_print\_query(query,s);
232. ResultSetMetaData rmd = r.getMetaData();
234. System.out.println("Table \'"+TABLE\_NAME+"\' that was created: ");

237. **while**(r.next())
238. {
239. String str = "";
240. ArrayList<String> strA = **new** ArrayList<>();
242. **for**(**int** i = 1; i < rmd.getColumnCount(); i++)
243. {
244. strA.add(r.getString(i));
245. }
247. System.out.println(String.format("%-10s %-15s %-5s %-5s %-5s",
248. strA.get(0),strA.get(1),strA.get(2),strA.get(3),strA.get(4)
249. ));
250. }

253. }
254. **catch** (SQLException e)
255. {
256. e.printStackTrace();
257. }
259. **return**;
261. }
263. **public** **static** **void** resetTables()
264. {
265. String delmeTemp = "ALTER TABLE " + TABLE\_NAME + " ADD delme INTEGER";
267. Statement s = **null**;
268. System.out.println("Executing this: \'" + delmeTemp + "\'");
269. **try**
270. {
271. s = conn.createStatement();
272. execute\_print\_query(delmeTemp,s);
273. //this is added as we need a table to have at least one column.
274. }
275. **catch** (Exception e)
276. {
277. System.out.println("Executing above statement failed!");
278. }

281. String dropQuery = "ALTER TABLE " + TABLE\_NAME + " DROP COLUMN ";
282. String addQuery  = "ALTER TABLE " + TABLE\_NAME + " ADD COLUMN ";
283. String tempQuery = "";
285. **for** (**int** i = 0; i < columns.length; i++)
286. {
287. tempQuery = dropQuery + columns[i];
288. System.out.println("exec: ");
289. execute\_print\_query(tempQuery,s);


293. tempQuery = addQuery + columns[i] + " " + columnTypes[i] + " " + columnModifiers[i];
294. System.out.println("exec: "+tempQuery);
295. **try**
296. {
297. conn.createStatement().execute(tempQuery);
298. }
299. **catch**(SQLException e)
300. {
301. System.out.print("Couldn't execute adding query \'"+tempQuery+"\' because of: ");
302. System.out.println(e.getMessage() + "\n");
303. }
305. }


309. **try**
310. {
311. conn.createStatement().execute("ALTER TABLE " + TABLE\_NAME + " DROP COLUMN delme");
312. }
313. **catch** (SQLException e)
314. {
315. e.printStackTrace();
316. }
317. }
319. **public** **static** **void** insertRecords(ArrayList<BankRecords> br, Boolean execute)
320. {
321. Statement s = **null**;
322. String query = "";
323. **try**
324. {
325. s = conn.createStatement();
326. }
327. **catch** (SQLException e1)
328. {
329. e1.printStackTrace();
330. }


334. **for** (**int** i = 0; i < br.size(); i++)
335. {
336. BankRecords person = br.get(i);
337. query = "INSERT INTO " + TABLE\_NAME + " VALUES(";
338. query += i + ", ";                                  //PID,      int
339. query += "'" +  person.id + "', ";                  //ID,       String
340. query +=        person.age + ", ";                  //age,      int
341. query += "'" +  person.gender + "', ";              //gender,   String
342. query += "'" +  person.region + "', ";              //region,   String
343. query +=        person.income + ", ";               //income,   double
344. query += "'" +  person.married + "', ";             //married,  String
345. query +=        person.children + ", ";             //children, int
346. query += "'" +  person.car + "', ";                 //car,      String
347. query += "'" +  person.save\_act + "', ";            //savings,  String
348. query += "'" +  person.current\_act + "', ";         //current,  String
349. query += "'" +  person.mortgage + "', ";            //mortgage, String
350. query += "'" +  person.pep + "'";                   //pep,      String
352. query += ")";
354. System.out.println("\nExecuting INSERT query: "+query);
355. **if**(execute)
356. {
357. execute\_print\_query(query,s);       //actually executes the thing
358. }
359. }
361. }

364. **public** **static** ResultSet executeQuery(String query)
365. {
366. Statement s = **null**;
367. System.out.print("Retrieving records using this query: ");
368. **try**
369. {
370. s = conn.createStatement();
371. **return** execute\_print\_query(query, s);
372. }
373. **catch** (SQLException e)
374. {
375. e.printStackTrace();
376. }
377. **return** **null**;
378. }
380. **public** **static** ResultSet executeQuery()
381. {
382. **return** executeQuery(MAX\_RESULTS);
383. }
385. **public** **static** ResultSet executeQuery(**int** limit)
386. {
387. String query = "SELECT \* FROM " + TABLE\_NAME + " LIMIT "+limit;
389. **return** executeQuery(query);
390. }

393. **public** **static** ArrayList<ArrayList<Object>> resultSetMetadataToList(ResultSet rs)
394. {
395. ResultSetMetaData rsmd = **null**;
396. ArrayList<ArrayList<Object>> ret = **new** ArrayList<>();
398. **try**
399. {
400. rsmd = rs.getMetaData();
401. ArrayList<Object> tempRow = **new** ArrayList<>();
402. ArrayList<Object> tempRow2 = **new** ArrayList<>();
404. //add metadata for headers to ret array
405. **for** (**int** i = 1; i < rsmd.getColumnCount(); i++)
406. {
407. tempRow.add(rsmd.getColumnName(i));
408. tempRow2.add(rsmd.getColumnTypeName(i));
409. }
411. tempRow.add((Object)rsmd.getColumnName(rsmd.getColumnCount()));     // FIXME figure out why we need this...fixes last col not appearing...
412. tempRow2.add((Object)rsmd.getColumnName(rsmd.getColumnCount()));        // FIXME figure out why we need this...fixes last col not appearing...
413. ret.add(tempRow);
414. ret.add(tempRow2);


418. }
419. **catch** (SQLException e)
420. {
421. e.printStackTrace();
422. }


426. **return** ret;
427. }
429. **public** **static** ArrayList<ArrayList<Object>> resultSetToList(ResultSet rs)
430. {
431. ResultSetMetaData rsmd = **null**;
432. ArrayList<ArrayList<Object>> ret = **new** ArrayList<>();
433. ArrayList<Object> tempRow;

436. **try**
437. {
439. rsmd = rs.getMetaData();

442. System.out.printf("Width of cols are %d\n",rsmd.getColumnCount());
444. **int** col = 0;
445. **while**(rs.next())
446. {
447. //              System.out.printf("Col %d\n",col);
448. //              System.out.println("rs.getString(1) = "+rs.getString(1));
449. //              System.out.println("rs.getObject(1) = "+rs.getObject(1));
451. tempRow = **new** ArrayList<>();
453. **if**(rsmd.getColumnCount() > 1)
454. {
455. **for** (**int** i = 1; i < rsmd.getColumnCount(); i++)
456. {
457. //                      System.out.printf("about to add %s\n",rs.getObject(i));
458. tempRow.add((Object)rs.getString(i));
459. }
460. }
462. tempRow.add((Object)rs.getString(rsmd.getColumnCount())); // FIXME figure out why we need this...fixes last col not appearing...


466. //              rs.geto
467. ret.add(tempRow);
468. //              System.out.println("added this row to ret array: "+tempRow.toString());
469. col++;
470. }


474. }
475. **catch** (SQLException e)
476. {
477. e.printStackTrace();
478. }
479. //      System.out.println("Returning this: "+ret.toString());

482. **return** ret;
483. }
485. **public** **static** **void** deleteAllRecords()
486. {
487. Statement s = **null**;
488. **try**
489. {
490. s = conn.createStatement();
491. s.execute("DELETE FROM "+TABLE\_NAME + ";");
492. }
493. **catch** (SQLException e)
494. {
495. e.printStackTrace();
496. }
497. }

500. }
501. **import** java.sql.ResultSet;
502. **import** java.sql.ResultSetMetaData;
503. **import** java.sql.SQLException;
504. **import** java.time.Instant;
505. **import** java.util.ArrayList;
506. **import** java.util.Date;
507. **import** java.util.Scanner;

510. **public** **class** LoanProcessing **extends** BankRecords
511. {
513. //  public static final String prompt = " > ";
514. **public** **static** **final** **int** TIME\_DELAY\_MS = 5000; //because idk how fast ur hard drive is ;)
515. **public** **static** **final** String DEFAULT\_QUERY = "SELECT \* FROM "+Dao.TABLE\_NAME;
516. **public** **static** **final** String TEST\_FILENAME = "test.ser";
517. **public** **static** **final** String prompt = "";
518. **public** **static** **final** String QUIT = "Q";
519. **public** **static** **final** String[] OPTIONS = {
520. "Print all records",
521. "Create tables",
522. "Reset and rebuild tables",
523. "Delete all records",
524. "Insert all records",
525. "Serialize to a file (String path), relative or absolute.",
526. "Deserialize from a file (String path), relative or absolute.",
527. "See time diff between seri... and de-seri... our bankRecords!",
528. };
530. **public** **static** **final** String PROTIP = "PROTIP: I use scan.next(), so if you want to " +
531. "delete all records and ALSO insert all records, you can do it in ONE LINE like so:\n" +
532. "1 [SPACE] 2 [SPACE] [ENTER]\n" +
533. "Cool, huh? Bye.";

536. **public** **static** **void** main(String[] args) **throws** InterruptedException
537. {
538. Scanner s = **new** Scanner(System.in);
539. String input = "";
540. Dao dao = **new** Dao();
541. **int** choice = Integer.MAX\_VALUE;
543. readData();
544. processData();
546. ArrayList<BankRecords> brfsf = **new** ArrayList<>();
548. System.out.println(PROTIP);
550. **while**(choice >= 0)
551. {
553. System.out.println("\n");
554. **for** (**int** i = 0; i < OPTIONS.length; i++)
555. {
556. System.out.printf("%2d: %s\n",i,OPTIONS[i]); //print out options
557. }
559. choice = s.nextInt();
561. **switch**(choice)
562. {
564. **case** 0:
565. ResultSet rs = dao.executeQuery(DEFAULT\_QUERY);
566. dao.printResultSet(dao.formatResultSet(rs));
567. **break**;
568. **case** 1:
569. dao.createTables();
570. **break**;
572. **case** 2:
573. dao.resetTables();
574. **break**;
576. **case** 3:
577. dao.deleteAllRecords();
578. **break**;
580. **case** 4:
581. dao.insertRecords(getBRList(),**true**);
582. **break**;
584. **case** 5:
585. System.out.println("Path to file:");
586. input = s.next();
587. BankRecordsSerializer.serialize(getBRList(), input);
588. **break**;
590. **case** 6:
591. System.out.println("Path to file:");
592. input = s.next();
593. brfsf.clear();
594. brfsf.addAll(BankRecordsSerializer.deserialize(input));
596. **if**(!brfsf.isEmpty())
597. {
598. System.out.println("From what two indices do you want to see your freshly deserialized file? (int, int)");
599. **int** lower = s.nextInt();
600. **int** higher = s.nextInt();
602. System.out.println("      "+getHeaders());
603. **for**(**int** i = lower; i <= higher; i++)
604. {
605. System.out.printf("%4d: %s\n",i,brfsf.get(i));
606. }
607. }
608. **else**
609. {
610. System.out.println("Oops.");
611. }
612. **break**;
614. **case** 7:
616. Instant startSer = Instant.now();
617. System.out.printf("Started at '%s'\n\n",startSer.toString());
619. BankRecordsSerializer.serialize(getBRList(), TEST\_FILENAME);
621. Instant stopSer = Instant.now();
622. System.out.printf("\n\nStopped at '%s'\n",startSer.toString());
624. **double** diffSer = Math.abs(startSer.compareTo(stopSer));

627. System.out.printf("Time to serialize '%s': %dns\n",TEST\_FILENAME,startSer.getNano());
629. System.out.printf("Sleeping for %dms aka %ds to not explode everything.\n",TIME\_DELAY\_MS,(TIME\_DELAY\_MS/1000));
630. Thread.sleep(TIME\_DELAY\_MS);

633. startSer = Instant.now();
634. System.out.printf("Started at '%s'\n\n",startSer.toString());
636. ArrayList<BankRecords> timeBRList = BankRecordsSerializer.deserialize(TEST\_FILENAME);
638. stopSer = Instant.now();
639. System.out.printf("\n\nStopped at '%s'\n",startSer.toString());
641. System.out.printf("Time to deserialize '%s': %dns\n",TEST\_FILENAME,startSer.getNano());

644. **break**;
646. **default**:
647. **break**;
648. }
649. }









660. ResultSet rs = dao.executeQuery();
661. ResultSetMetaData rsmd = **null**;
662. **try**
663. {
664. rsmd = rs.getMetaData();
665. }
666. **catch** (SQLException e)
667. {
668. e.printStackTrace();
669. }
671. ArrayList<String> formatted = dao.formatResultSet(rs);
673. dao.printResultSet(formatted);
674. s.close();
676. }

679. **public** **static** Boolean parseYes(String word)
680. {
681. **if**(word.charAt(0) == 'y' || word.charAt(0) == 'Y')
682. {
683. **return** **true**;
684. }
685. **return** **false**;
686. }
688. **public** **static** Boolean parseQuit(String word)
689. {
690. **if**(word.charAt(0) == 'q' || word.charAt(0) == 'Q')
691. {
692. **return** **true**;
693. }
694. **return** **false**;
695. }
696. }
697. **import** java.awt.BorderLayout;
698. **import** java.awt.Color;
699. **import** java.awt.Component;
700. **import** java.awt.ComponentOrientation;
701. **import** java.awt.Dimension;
702. **import** java.awt.EventQueue;
703. **import** java.awt.Font;
704. **import** java.awt.event.ActionEvent;
705. **import** java.awt.event.ActionListener;
706. **import** java.awt.event.MouseAdapter;
707. **import** java.awt.event.MouseEvent;
708. **import** java.awt.font.FontRenderContext;
709. **import** java.awt.geom.AffineTransform;
710. **import** java.sql.ResultSet;
711. **import** java.util.ArrayList;
713. **import** javax.swing.DefaultListModel;
714. **import** javax.swing.JButton;
715. **import** javax.swing.JFrame;
716. **import** javax.swing.JLabel;
717. **import** javax.swing.JList;
718. **import** javax.swing.JPanel;
719. **import** javax.swing.JScrollPane;
720. **import** javax.swing.JTable;
721. **import** javax.swing.JTextField;
722. **import** javax.swing.KeyStroke;
723. **import** javax.swing.SpringLayout;
724. **import** javax.swing.SwingConstants;
725. **import** javax.swing.SwingUtilities;
726. **import** javax.swing.table.DefaultTableModel;

729. **import** java.awt.event.KeyEvent;
730. **import** javax.swing.JMenuBar;
731. **import** javax.swing.JMenuItem;
732. **import** javax.swing.JOptionPane;
733. **import** java.awt.event.KeyAdapter;
735. **public** **class** testframe **extends** JFrame
736. {

739. **private** **static** **final** **long** serialVersionUID = 1L;
740. **private** **static** **final** **int** PADDING = 10;
741. **private** **static** **final** **int** MAX\_WIDTH\_MULT = 3;
742. **private** **static** **final** **int** JTABLE\_COLUMN\_PADDING = 10;
744. **private** **static** **final** String TITLE = "Henry Post's ITM411 SQL Crunchifier v1.67acdefg2";
746. **private** **static** **final** String HELP\_MESSAGE = "'F1' will bring me back up.\n"
747. + "\n"
748. + "Hi, Papademas or a TA. How are you? Hopefully well.\n"
749. + " Sorry for the fake lambdas and typecasts everywhere. I didn't want to have to get java 8 or make you do the same.\n"
750. + "Anyways, here are usage tips:\n"
751. + "\n"
752. + "Double-click any command from command history to run it.\n"
753. + "Double-right-click any command from command history to delete it.\n"
754. + "\n"
755. + "That's kinda it. Have fun.\n"
756. + "Oh, also, I made a really useful function called stringDisplaySize() that \n"
757. + "renders fonts to EXACTLY determine how large the strings should be given a fontface, fontsize, etc.\n"
758. + "My serialization is in 'Serializer.java'. Have fun!\n"
759. + "-Henry.";


763. **private** **static** **final** String DEFAULT\_COMMAND = "SELECT \* FROM " + Dao.TABLE\_NAME;
765. **private** **static** String[] SUGGESTED\_COMMANDS = {
766. DEFAULT\_COMMAND,
767. "SELECT \* "+                        "FROM " + Dao.TABLE\_NAME + " WHERE pid < 10",
768. "SELECT \* "+                        "FROM " + Dao.TABLE\_NAME + " LIMIT 13",
769. "SELECT pid, id, age, gender "+     "FROM " + Dao.TABLE\_NAME + " WHERE AGE = 30",
770. "SELECT age "+                      "FROM " + Dao.TABLE\_NAME + " ORDER BY income ASC LIMIT 10",
771. "DESCRIBE " + Dao.TABLE\_NAME,
773. };
775. **private** **static** **final** Object[] DEFAULT\_FONT\_PROPERTIES = {"Dialog",Font.BOLD,12};

778. **private** **static** ArrayList<Object> LAMBDA\_PARAMS = **new** ArrayList<>();
780. **private** **static** **final** **int** L\_MODE1\_TEXT\_DISPLAY\_WIDTH =       0x0001;
781. **private** **static** **final** **int** L\_MODE1\_STRING\_NUMBER\_OF\_CHARS =   0x0002;
783. **private** **static** **final** **int** L\_MODE2\_GREATER\_THAN =             0xFF01;
784. **private** **static** **final** **int** L\_MODE2\_LESS\_THAN =                0xFF02;

787. **private** JPanel contentPane;
788. **private** JTable tableResults;
789. **private** JTextField textFieldSQLQuery;
790. **private** JLabel lblNewLabel;
791. **private** JPanel panelInfo;
792. **private** JPanel panelScrollPaneResults;
793. **private** JButton btnRun;
794. **private** JPanel panelCommandHistory;
795. **private** JMenuBar menuBar;



800. /\*\*\*
801. \* Ensure that a 2d object arraylist is n by m everywhere with no "bumpiness". <br>
802. \* Fills with {filler}.<br>
803. \* <br>
804. \* Example if {filler} were -1:<br>
805. \*
806. \* <code>
807. \* {<br>
808. \* { 1,    "hi",   2,   'a' },<br>
809. \* { 2,    "ho",   4,   'b',   'c',   'e' },<br>
810. \* { [1, 2] }<br>
811. \* }<br>
812. \* </code><br>
813. \* <br>
814. \* turns into<br>
815. \* <br>
816. \* <code>
817. \* {<br>
818. \* { 1,    "hi",   2,   'a',   -1,    -1  },<br>
819. \* { 2,    "ho",   4,   'b',   'c',   'e' },<br>
820. \* { [1, 2], -1,   -1,   -1,    -1,    -1,}<br>
821. \* }<br>
822. \* </code>
823. \*/
825. **public** ArrayList<ArrayList<Object>> ensurePadded(ArrayList<ArrayList<Object>> list)
826. {
827. ArrayList<ArrayList<Object>> ret = **new** ArrayList<>();
828. ret = list;
830. **int** largest = list.get(0).size();
832. **for** (**int** i = 0; i < list.size(); i++)
833. {
834. **if**(list.get(i).size() > largest)
835. {
836. largest = list.get(i).size();
837. }
838. }
840. **for**(**int** i = 0; i < list.size(); i++)
841. {
842. System.out.printf("unfinished. ha. %d",i);
843. }



848. **return** ret;
849. }

852. /\*\*
853. \* Launch the application.
854. \*/
855. **public** **static** **void** main(String[] args)
856. {
857. EventQueue.invokeLater(**new** Runnable()
858. {
859. **public** **void** run()
860. {
861. **try**
862. {
863. testframe frame = **new** testframe();
864. frame.setVisible(**true**);
865. }
866. **catch** (Exception e)
867. {
868. e.printStackTrace();
869. }
870. }
871. });
872. }


876. **public** **static** **void** lamb\_duh\_args(Object[] a)
877. {
878. LAMBDA\_PARAMS.clear();
879. LAMBDA\_PARAMS.add(-1);
881. **if**(a != **null**)
882. {
883. **for** (**int** i = 0; i < a.length; i++)
884. {
885. LAMBDA\_PARAMS.add(a[i]);
886. }
887. }
888. **return**;
889. }
891. /\*\*\*
892. \* Too lazy to use wrapper classes. Oh well. Here's a switch-case! Totally a higher-order function, right???
893. \* Sorry in advance to the TAs.
894. \* @param a Object a.
895. \* @param b Object b.
896. \* @param mode2 How to compare these two objects?
897. \* @return  The difference between the two objects.
898. \*/
899. **public** **static** Object lamb\_duh(Object a, Object b, **int** mode2)
900. {
901. **switch** (mode2)
902. {
903. **case** L\_MODE2\_GREATER\_THAN:
904. **return** ((Double)a > (Double)b);
905. **case** L\_MODE2\_LESS\_THAN:
906. **return** ((Double)a < (Double)b);

909. **default**:
910. **return** **null**;
911. }
912. }
914. /\*\*\*
915. \* This is a unary version of the {@link #lamb\_duh(Object, int)} function defined here: <br>{@link #lamb\_duh(Object, Object, int)}
916. \* @param a Object a.
917. \* @param mode1 How to 'size up' or 'measure' this object?
918. \* @return The 'value' of this object.
919. \*/
921. **public** **static** Object lamb\_duh(Object a, **int** mode1)
922. {
923. //      System.out.println("Passed object:"+a.toString());
925. Object ret = **null**;

928. **boolean** optArgs = **false**;
929. **if**((LAMBDA\_PARAMS.size() >= 1) && (LAMBDA\_PARAMS.get(0) != **null**))
930. {
931. optArgs = **true**;
932. }
934. **switch**(mode1)
935. {
936. **case** L\_MODE1\_TEXT\_DISPLAY\_WIDTH: //get render width
937. **if**(optArgs) //if we want to use optional args
938. {
939. ret = stringDisplaySize((String)a, (String)LAMBDA\_PARAMS.get(1), (Integer)LAMBDA\_PARAMS.get(2), (Integer)LAMBDA\_PARAMS.get(3))[0];
940. }
941. **else**
942. {
943. ret = stringDisplaySize((String)a)[0];
944. }
945. **break**;
947. **case** L\_MODE1\_STRING\_NUMBER\_OF\_CHARS: //get number of chars
948. **return** (Integer)((String)a).length();

951. }
953. //      System.out.println("Returning object: "+ret.toString());
954. **return** ret;
955. }

958. /\*\*\*
959. \* Calculates, in pixels, the width & height of a string of text.
960. \* @param   s           The string in question.
961. \* @param   fontName    The name of the font. i.e: "Tahoma", "Times New Roman", etc.
962. \* @param   fontStyle   The style of the font. i.e: "Font.PLAIN", "Font.BOLD", etc.
963. \* @param   size        The size of the font
964. \* @return  An int[] containing <code>{width, height}</code> of the font in pixels if it were to be rendered
965. \*          in the given context.
966. \*/
967. **public** **static** Double[] stringDisplaySize(String s, String fontName, **int** fontStyle, **int** size)
968. {
970. AffineTransform aft = **new** AffineTransform();
972. FontRenderContext frc = **new** FontRenderContext(aft, **true**, **true**);
974. Font font = **new** Font(fontName, fontStyle, size);
976. Double[] ret = {
977. font.getStringBounds(s, frc).getWidth(),
978. font.getStringBounds(s, frc).getHeight()
979. };
981. //      System.out.println(String.format("String %s sized %d with fontName %s and fontStyle '%d' would be:\n%fpx by %fpx",
982. //                                              s,  size,           fontName,     fontStyle,            ret[0],     ret[1]));
984. **return** ret;
985. }
987. /\*\*\*
988. \* Calculates, in pixels, the width & height of a string of text.
989. \* This method is a default version of {@link #stringDisplaySize(String, String, int, int)}.
990. \* It uses ["Tahoma", Font.PLAIN, 12] as [fontName, fontStyle, size].
991. \* @param s
992. \* @return
993. \*/
994. **public** **static** Double[] stringDisplaySize(String s)
995. {
996. **return** stringDisplaySize(s, (String)DEFAULT\_FONT\_PROPERTIES[0], (**int**)DEFAULT\_FONT\_PROPERTIES[1], (**int**)DEFAULT\_FONT\_PROPERTIES[2]);
997. }
999. /\*\*
1000. \* \*
1001. \* @param table         A 2-d array of objects.
1002. \* @param lambdaMode    Compare 2 objects by string length, by render width, by how tasty they are, etc.
1003. \*
1004. \* @return  A 1d array of the longest things in each array's column.
1005. \*          The "longness" is determined by the chars in the object's
1006. \*          .toString() method, or by pretending to render them and
1007. \*          looking at their width.
1008. \* <br>
1009. \* Example: <br>
1010. \* <code>
1011. \* table = <br>
1012. \* {{1,     "hi",   3},<br>
1013. \* {"one",  2,      4}}<br></code>
1014. \*
1015. \* And the return value would be:<br>
1016. \*  <code>{3, 2, 1}</code>
1017. \*/
1018. **public** **static** ArrayList<Integer> getLongestElements(ArrayList<ArrayList<Object>> table, **int** lambdaMode)
1019. {
1020. System.out.println("Asked to get longest elements from this table: "+table.toString());
1021. ArrayList<Integer> ret = **new** ArrayList<>();
1023. **for** (**int** i = 0; i < table.get(0).size(); i++)
1024. {
1025. ret.add(-1); //init n empty values for n columns
1026. }
1028. **for** (**int** i = 0; i < table.size(); i++) //loop through all rows
1029. {
1030. ArrayList<Object> row = table.get(i);
1031. //          System.out.printf("Row is %s\n",row.toString());
1033. **for** (**int** j = 0; j < row.size(); j++) //loop through each individual column
1034. {
1035. Object currItem = row.get(j);
1037. **if**(currItem == **null**) //take that, nullpointerexception! null.toString() shall not crash me!
1038. {
1039. currItem = "null";
1040. }
1042. //              System.out.printf("row.get(j) = %s\n",currItem.toString());
1043. Integer sizeA = ((Double)lamb\_duh(currItem.toString(), lambdaMode)).intValue();
1044. Integer sizeB = ((Double)lamb\_duh(ret.get(j).toString(), lambdaMode)).intValue();
1045. **if**(sizeA > sizeB) //if row n's length is greater than bucket[n], add it
1046. {
1047. ret.set(j, sizeA);
1048. }
1049. }
1050. }

1053. **return** ret;
1054. }
1056. /\*\*\*
1057. \*
1058. \* @param tabledata
1059. \* @return
1060. \*/
1061. **public** **static** DefaultTableModel defaultTableModelFromData(ArrayList<ArrayList<Object>> tabledata)
1062. {
1063. Object topCol[] = tabledata.get(0).toArray(**new** Object[tabledata.get(0).size()]); //arraylist<object> to object[]
1065. DefaultTableModel dtm = **new** DefaultTableModel(topCol, 0);       //add X rows
1067. **for** (**int** i = 1; i < tabledata.size(); i++)
1068. {
1069. dtm.addRow(tabledata.get(i).toArray(**new** Object[tabledata.get(i).size()]));
1070. }
1071. **return** dtm;
1072. }
1074. /\*\*
1075. \* @param table The table to be resized.
1076. \* @param largestCols An ArrayList of ints that each column's max width will be set to.
1077. \* Takes an array of integers
1078. \*
1079. \*/
1080. **public** **static** **void** resizeTableCols(JTable table, ArrayList<Integer> largestCols)
1081. {
1082. System.out.println("Using this list of sizes for table column widths: " + largestCols.toString());
1083. **for**(**int** i = 0; i < largestCols.size(); i++)
1084. {
1085. **int** width = largestCols.get(i);
1086. //          int width = largestCols.get(i) \* WIDTH\_MULT;
1087. //          System.out.println(String.format("Setting table's column %d's maxWidth to %d",i,width));
1089. table.getColumnModel().getColumn(i).setMinWidth(width + JTABLE\_COLUMN\_PADDING); //set max width to the array's val at arr[i]
1090. //          table.getColumnModel().getColumn(i).setMaxWidth(width \* MAX\_WIDTH\_MULT); //set max width to the array's val at arr[i]
1091. }
1092. **return**;
1093. }

1096. **public** **static** **int** updateJTable(Dao dao, JTable table, String query)
1097. {
1099. ResultSet rs = dao.executeQuery(query);
1101. ArrayList<ArrayList<Object>> recordData = dao.resultSetToList(rs);
1103. ArrayList<ArrayList<Object>> recordMetadata = dao.resultSetMetadataToList(rs);
1105. recordData.add(0,recordMetadata.get(0)); //add headers
1107. Object[] args = DEFAULT\_FONT\_PROPERTIES;
1109. lamb\_duh\_args(args); //pass args
1110. ArrayList<Integer> longestCols = getLongestElements(recordData, L\_MODE1\_TEXT\_DISPLAY\_WIDTH);
1111. lamb\_duh\_args(**null**);  //clear 'em
1113. table.setModel(defaultTableModelFromData(recordData));
1115. resizeTableCols(table, longestCols);
1117. **return** 1;
1119. }

1122. /\*\*
1123. \* Create the frame.
1124. \*/
1125. **public** testframe()
1126. {

1129. setTitle(TITLE);
1130. setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);
1131. setBounds(100, 100, 1015, 617);
1133. menuBar = **new** JMenuBar();
1134. setJMenuBar(menuBar);
1136. JMenuItem mntmHelp = **new** JMenuItem("Help");
1137. mntmHelp.addActionListener(**new** ActionListener()
1138. {
1139. **public** **void** actionPerformed(ActionEvent e)
1140. {
1141. JOptionPane.showMessageDialog(contentPane, HELP\_MESSAGE);
1142. }
1143. });
1145. mntmHelp.setActionCommand("(H)elp");
1147. KeyStroke F1 = KeyStroke.getKeyStroke(KeyEvent.VK\_F1,0);
1148. mntmHelp.setAccelerator(F1);
1150. menuBar.add(mntmHelp);
1151. contentPane = **new** JPanel();
1152. contentPane.setAlignmentX(0.0f);
1153. contentPane.setBackground(Color.DARK\_GRAY);
1154. contentPane.setBorder(**null**);
1155. setContentPane(contentPane);

1158. Dao dao = **new** Dao();
1160. DefaultListModel<String> listCommandHistoryModel = **new** DefaultListModel<>();
1162. dao.command\_history.add(SUGGESTED\_COMMANDS[0]);

1165. **for** (**int** i = 0; i < SUGGESTED\_COMMANDS.length; i++)
1166. {
1167. listCommandHistoryModel.addElement(SUGGESTED\_COMMANDS[i]);
1168. }


1172. SpringLayout sl\_contentPane = **new** SpringLayout();
1173. contentPane.setLayout(sl\_contentPane);
1175. panelInfo = **new** JPanel();
1176. sl\_contentPane.putConstraint(SpringLayout.WEST, panelInfo, 10, SpringLayout.WEST, contentPane);
1177. panelInfo.setMaximumSize(**new** Dimension(300, 32767));
1178. panelInfo.setComponentOrientation(ComponentOrientation.LEFT\_TO\_RIGHT);
1179. sl\_contentPane.putConstraint(SpringLayout.NORTH, panelInfo, PADDING, SpringLayout.NORTH, contentPane);
1180. sl\_contentPane.putConstraint(SpringLayout.SOUTH, panelInfo, -PADDING, SpringLayout.SOUTH, contentPane);
1181. panelInfo.setAlignmentY(Component.TOP\_ALIGNMENT);
1182. panelInfo.setAlignmentX(Component.LEFT\_ALIGNMENT);
1183. contentPane.add(panelInfo);
1184. SpringLayout sl\_panelInfo = **new** SpringLayout();
1185. panelInfo.setLayout(sl\_panelInfo);
1187. JPanel panelSQLQuery = **new** JPanel();
1188. sl\_panelInfo.putConstraint(SpringLayout.NORTH, panelSQLQuery, PADDING, SpringLayout.NORTH, panelInfo);
1189. sl\_panelInfo.putConstraint(SpringLayout.WEST, panelSQLQuery, PADDING, SpringLayout.WEST, panelInfo);
1190. panelInfo.add(panelSQLQuery);
1191. panelSQLQuery.setLayout(**new** BorderLayout(0, 0));
1193. textFieldSQLQuery = **new** JTextField();
1194. panelSQLQuery.add(textFieldSQLQuery, BorderLayout.CENTER);
1195. textFieldSQLQuery.setColumns(10);
1197. lblNewLabel = **new** JLabel("SQL Query");
1198. lblNewLabel.setHorizontalAlignment(SwingConstants.CENTER);
1199. panelSQLQuery.add(lblNewLabel, BorderLayout.NORTH);
1201. btnRun = **new** JButton("Run!");
1202. sl\_panelInfo.putConstraint(SpringLayout.EAST, panelSQLQuery, -6, SpringLayout.WEST, btnRun);
1203. sl\_panelInfo.putConstraint(SpringLayout.WEST, btnRun, 261, SpringLayout.WEST, panelInfo);
1204. sl\_panelInfo.putConstraint(SpringLayout.EAST, btnRun, -PADDING, SpringLayout.EAST, panelInfo);

1207. JLabel labelRecentCommand = **new** JLabel(DEFAULT\_COMMAND);
1208. labelRecentCommand.setDisplayedMnemonic(KeyEvent.VK\_ENTER);
1209. labelRecentCommand.setBackground(Color.LIGHT\_GRAY);
1210. labelRecentCommand.setHorizontalAlignment(SwingConstants.CENTER);
1212. sl\_panelInfo.putConstraint(SpringLayout.NORTH, btnRun, PADDING, SpringLayout.NORTH, panelInfo);
1213. panelInfo.add(btnRun);
1215. panelCommandHistory = **new** JPanel();
1216. sl\_panelInfo.putConstraint(SpringLayout.NORTH, panelCommandHistory, PADDING, SpringLayout.SOUTH, panelSQLQuery);
1217. sl\_panelInfo.putConstraint(SpringLayout.SOUTH, panelCommandHistory, -PADDING, SpringLayout.SOUTH, panelInfo);
1218. sl\_panelInfo.putConstraint(SpringLayout.SOUTH, btnRun, -PADDING, SpringLayout.NORTH, panelCommandHistory);
1219. sl\_panelInfo.putConstraint(SpringLayout.WEST, panelCommandHistory, PADDING, SpringLayout.WEST, panelInfo);
1220. sl\_panelInfo.putConstraint(SpringLayout.EAST, panelCommandHistory, -PADDING, SpringLayout.EAST, panelInfo);
1221. panelInfo.add(panelCommandHistory);
1223. JList listCommandHistory = **new** JList(listCommandHistoryModel);
1225. SpringLayout sl\_panelCommandHistory = **new** SpringLayout();
1226. sl\_panelCommandHistory.putConstraint(SpringLayout.NORTH, listCommandHistory, 16, SpringLayout.NORTH, panelCommandHistory);
1227. sl\_panelCommandHistory.putConstraint(SpringLayout.WEST, listCommandHistory, PADDING, SpringLayout.WEST, panelCommandHistory);
1228. sl\_panelCommandHistory.putConstraint(SpringLayout.SOUTH, listCommandHistory, -10, SpringLayout.SOUTH, panelCommandHistory);
1229. sl\_panelCommandHistory.putConstraint(SpringLayout.EAST, listCommandHistory, -PADDING, SpringLayout.EAST, panelCommandHistory);
1230. panelCommandHistory.setLayout(sl\_panelCommandHistory);

1233. panelCommandHistory.add(listCommandHistory);
1234. sl\_panelInfo.putConstraint(SpringLayout.NORTH, listCommandHistory, 228, SpringLayout.SOUTH, panelSQLQuery);
1235. sl\_panelInfo.putConstraint(SpringLayout.WEST, listCommandHistory, 0, SpringLayout.WEST, panelSQLQuery);
1236. sl\_panelInfo.putConstraint(SpringLayout.SOUTH, listCommandHistory, -PADDING, SpringLayout.SOUTH, panelInfo);
1237. sl\_panelInfo.putConstraint(SpringLayout.EAST, listCommandHistory, 0, SpringLayout.EAST, btnRun);
1239. JLabel lblCommandHistory = **new** JLabel("Command History");
1240. sl\_panelCommandHistory.putConstraint(SpringLayout.NORTH, lblCommandHistory, 0, SpringLayout.NORTH, panelCommandHistory);
1241. sl\_panelCommandHistory.putConstraint(SpringLayout.WEST, lblCommandHistory, 0, SpringLayout.WEST, panelCommandHistory);
1242. sl\_panelCommandHistory.putConstraint(SpringLayout.EAST, lblCommandHistory, 310, SpringLayout.WEST, panelCommandHistory);
1243. lblCommandHistory.setHorizontalAlignment(SwingConstants.CENTER);
1244. panelCommandHistory.add(lblCommandHistory);
1246. JPanel panelResults = **new** JPanel();
1247. sl\_contentPane.putConstraint(SpringLayout.NORTH, panelResults, PADDING, SpringLayout.NORTH, contentPane);
1248. sl\_contentPane.putConstraint(SpringLayout.WEST, panelResults, 6, SpringLayout.EAST, panelInfo);
1249. sl\_contentPane.putConstraint(SpringLayout.SOUTH, panelResults, -10, SpringLayout.SOUTH, contentPane);
1250. sl\_contentPane.putConstraint(SpringLayout.EAST, panelResults, -PADDING, SpringLayout.EAST, contentPane);
1251. contentPane.add(panelResults);
1252. SpringLayout sl\_panelResults = **new** SpringLayout();
1253. sl\_panelResults.putConstraint(SpringLayout.WEST, labelRecentCommand, 10, SpringLayout.WEST, panelResults);
1254. sl\_panelResults.putConstraint(SpringLayout.EAST, labelRecentCommand, -10, SpringLayout.EAST, panelResults);
1255. panelResults.setLayout(sl\_panelResults);
1257. tableResults = **new** JTable();
1259. updateJTable(dao, tableResults, DEFAULT\_COMMAND);
1261. panelScrollPaneResults = **new** JPanel();
1262. sl\_panelResults.putConstraint(SpringLayout.SOUTH, panelScrollPaneResults, -10, SpringLayout.SOUTH, panelResults);
1263. sl\_panelResults.putConstraint(SpringLayout.SOUTH, labelRecentCommand, -10, SpringLayout.NORTH, panelScrollPaneResults);
1264. sl\_panelResults.putConstraint(SpringLayout.NORTH, panelScrollPaneResults, 31, SpringLayout.NORTH, panelResults);
1265. sl\_panelResults.putConstraint(SpringLayout.WEST, panelScrollPaneResults, 10, SpringLayout.WEST, labelRecentCommand);
1266. sl\_panelResults.putConstraint(SpringLayout.EAST, panelScrollPaneResults, -PADDING, SpringLayout.EAST, panelResults);
1267. panelResults.add(panelScrollPaneResults);
1268. sl\_contentPane.putConstraint(SpringLayout.EAST, panelInfo, -PADDING, SpringLayout.WEST, panelScrollPaneResults);
1269. sl\_contentPane.putConstraint(SpringLayout.NORTH, panelScrollPaneResults, 55, SpringLayout.NORTH, contentPane);
1270. sl\_contentPane.putConstraint(SpringLayout.WEST, panelScrollPaneResults, 350, SpringLayout.WEST, contentPane);
1271. sl\_contentPane.putConstraint(SpringLayout.SOUTH, panelScrollPaneResults, -PADDING, SpringLayout.SOUTH, contentPane);
1272. sl\_contentPane.putConstraint(SpringLayout.EAST, panelScrollPaneResults, -PADDING, SpringLayout.EAST, contentPane);
1273. SpringLayout sl\_panelScrollPaneResults = **new** SpringLayout();
1274. panelScrollPaneResults.setLayout(sl\_panelScrollPaneResults);
1276. JScrollPane scrollPaneResults = **new** JScrollPane();
1277. sl\_panelScrollPaneResults.putConstraint(SpringLayout.NORTH, scrollPaneResults, PADDING, SpringLayout.NORTH, panelScrollPaneResults);
1278. sl\_panelScrollPaneResults.putConstraint(SpringLayout.WEST, scrollPaneResults, PADDING, SpringLayout.WEST, panelScrollPaneResults);
1279. sl\_panelScrollPaneResults.putConstraint(SpringLayout.SOUTH, scrollPaneResults, -PADDING, SpringLayout.SOUTH, panelScrollPaneResults);
1280. sl\_panelScrollPaneResults.putConstraint(SpringLayout.EAST, scrollPaneResults, -PADDING, SpringLayout.EAST, panelScrollPaneResults);
1281. panelScrollPaneResults.add(scrollPaneResults);
1283. tableResults.setColumnSelectionAllowed(**true**);
1284. tableResults.setCellSelectionEnabled(**true**);
1285. tableResults.setAutoResizeMode(JTable.AUTO\_RESIZE\_ALL\_COLUMNS);
1286. tableResults.setFillsViewportHeight(**true**);

1289. scrollPaneResults.setViewportView(tableResults);
1290. panelResults.add(labelRecentCommand);
1292. /\*\*\*
1293. \* If the 'run' button gets clicked.
1294. \*/
1295. btnRun.addActionListener(**new** ActionListener() {
1296. **public** **void** actionPerformed(ActionEvent e)
1297. {
1299. System.out.println("WHO KLICK RUN??");
1300. String query = textFieldSQLQuery.getText();
1301. labelRecentCommand.setForeground(Color.BLACK);
1302. **try**
1303. {
1304. updateJTable(dao, tableResults, query);
1305. labelRecentCommand.setText(query);
1306. **if**(!listCommandHistoryModel.contains(query)) //if command isn't in recent commands
1307. {
1308. listCommandHistoryModel.add(0, query);  //add it
1309. }
1310. }
1311. **catch**(Exception e1)
1312. {
1313. e1.printStackTrace();
1314. labelRecentCommand.setText("ERROR: "+query);
1315. labelRecentCommand.setForeground(Color.RED);
1316. }


1320. textFieldSQLQuery.setText(query);
1321. textFieldSQLQuery.setToolTipText(query);

1324. }
1325. });

1328. /\*\*\*
1329. \* For single/double clicks.
1330. \*/
1331. listCommandHistory.addMouseListener(**new** MouseAdapter(){
1332. @Override
1333. **public** **void** mouseClicked(MouseEvent event)
1334. {
1335. JList list = (JList)event.getSource();
1336. **int** index = list.locationToIndex(event.getPoint());
1337. **int** clicks = event.getClickCount();
1338. String query = listCommandHistoryModel.get(index);


1342. **switch** (clicks)
1343. {
1344. **case** 2:
1345. **if**(SwingUtilities.isLeftMouseButton(event))
1346. {
1347. System.out.println(String.format("Clicked list element %d which is %s",index,query));
1348. labelRecentCommand.setForeground(Color.BLACK);
1350. **try**
1351. {
1352. updateJTable(dao, tableResults, query);
1353. labelRecentCommand.setText(query);
1354. }
1355. **catch**(Exception e1)
1356. {
1357. e1.printStackTrace();
1358. labelRecentCommand.setText("ERROR: "+query);
1359. labelRecentCommand.setForeground(Color.RED);
1360. }
1362. textFieldSQLQuery.setToolTipText(query);
1363. labelRecentCommand.setText(query);
1365. }
1366. **if**(SwingUtilities.isRightMouseButton(event))
1367. {
1368. listCommandHistoryModel.remove(index);
1369. }
1371. **break**;
1373. **case** 1:
1374. **if**(SwingUtilities.isLeftMouseButton(event))
1375. {
1376. textFieldSQLQuery.setText(query);
1377. textFieldSQLQuery.setToolTipText(query);
1378. }
1379. **default**:
1380. //                          System.out.println("clicky??");
1381. **break**;
1382. }
1383. }
1384. });
1386. /\*\*\*
1387. \* This is when you press ENTER inside of the SQL query text box.
1388. \*/
1389. textFieldSQLQuery.addActionListener(**new** ActionListener()
1390. {
1391. **public** **void** actionPerformed(ActionEvent ae)
1392. {
1393. System.out.println("U PRESSIN ENTER?");
1394. }
1395. });

1398. mntmHelp.setVisible(**true**);
1400. JOptionPane.showMessageDialog(contentPane, HELP\_MESSAGE);
1402. }
1403. }
1404. **import** java.io.EOFException;
1405. **import** java.io.File;
1406. **import** java.io.FileInputStream;
1407. **import** java.io.FileOutputStream;
1408. **import** java.io.IOException;
1409. **import** java.io.ObjectInputStream;
1410. **import** java.io.ObjectOutputStream;
1411. **import** java.io.Serializable;
1412. **import** java.util.ArrayList;
1413. **import** java.util.NoSuchElementException;
1415. **public** **class** BankRecordsSerializer **implements** Serializable
1416. {
1418. **private** **static** **final** **long** serialVersionUID = 1L;
1420. **public** **static** **void** serialize(ArrayList<BankRecords> bankRecordsList, String path)
1421. {
1423. System.out.println("First 3 lines of object to be serialized:");
1425. **for** (   **int** i = 0;
1426. i < (bankRecordsList.size() >= 3 ? 3 : bankRecordsList.size());
1427. i++)
1428. {
1429. System.out.println(bankRecordsList.get(i).toString());
1430. }
1432. System.out.printf("Serializing object to this filepath: '%s'",path);


1436. ObjectOutputStream oos = **null**;
1437. **try**
1438. {
1439. oos = **new** ObjectOutputStream(**new** FileOutputStream(path));
1441. **for**(BankRecords person: bankRecordsList)
1442. {
1443. oos.writeObject(person);
1444. }
1446. }
1447. **catch** (IOException ioe)
1448. {
1449. System.err.printf("Error opening file at '%s'\n",path);
1450. ioe.printStackTrace();
1451. }
1452. **catch** (NoSuchElementException nsee)
1453. {
1454. System.err.printf("halp! \n");
1455. nsee.printStackTrace();
1456. }
1457. **finally**
1458. {
1459. **if**(oos != **null**)
1460. {
1461. **try**
1462. {
1463. oos.close();
1464. }
1465. **catch** (IOException e)
1466. {
1467. System.err.printf("Failed closing file at '%d'\n",path);
1468. e.printStackTrace();
1469. }
1470. }
1471. }
1472. }
1474. **public** **static** ArrayList<BankRecords> deserialize(String path)
1475. {
1476. ArrayList<BankRecords> ret = **new** ArrayList<>();
1478. ObjectInputStream ois = **null**;
1480. **try**
1481. {
1482. ois = **new** ObjectInputStream(**new** FileInputStream(path));
1484. **while**(**true**)
1485. {
1486. BankRecords br = (BankRecords) ois.readObject();
1487. ret.add(br);
1488. }
1490. }
1491. **catch** (EOFException eofe)
1492. {
1493. //end o' list
1494. **return** ret;
1495. }
1496. **catch** (IOException ioe)
1497. {
1498. System.out.printf("Error opening file '%s' for reading!",path);
1499. ioe.printStackTrace();
1500. }
1501. **catch** (ClassNotFoundException cnfe)
1502. {
1503. System.out.println("Object creation failed! Our list so far:");
1504. System.out.println(ret.toString());
1505. cnfe.printStackTrace();
1506. }
1508. **finally**
1509. {
1510. **if**(ois != **null**)
1511. {
1512. **try**
1513. {
1514. ois.close();
1515. }
1516. **catch** (IOException ioe2)
1517. {
1518. System.out.printf("Error closing file '%s'!",path);
1519. ioe2.printStackTrace();
1520. }
1521. }
1522. }
1523. **return** ret;
1524. }
1525. }