College Calendar App Documentaion

Group members:

Éamon Dunne

Amber Higgins

Grace Carandang

Tushti Singla

Arwa Aldakheel

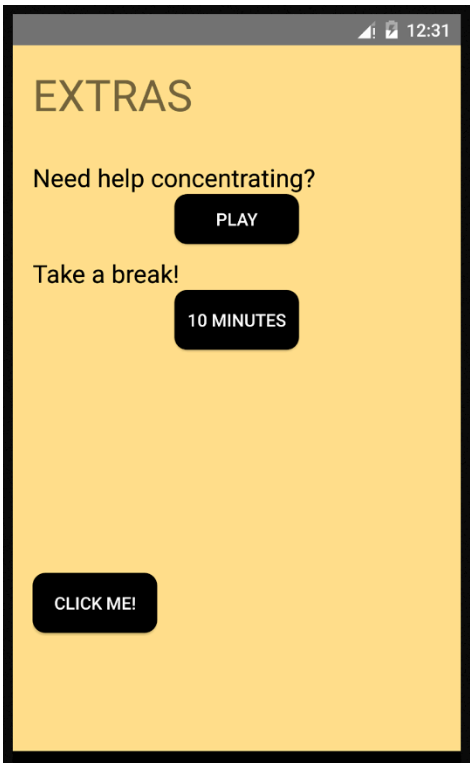
The CollegeCalendar App was made with the aim of making the lives of students easier and to allow them to be more organized because they can have all their college related tasks in one app. It allows them to add a weekly schedule, which they can use to add their set timetable of classes and tutorials and laboratory sessions. It also gives the option to add events, which can be used to set due dates or any college or non-college related activities to their timetable. There is also the option to add as many tasks as the user wishes to a do list, which can be ticket off and deleted when the task is done. To make it stand out from other calendar apps, an 'Extras' Activity as an additional content for the user is available from the Main Menu, it provides access to a music player, stress releasing puppies pictures and 10 minutes break button because taking breaks from studying sessions is highly beneficial for students.

Extras Activity - Architecture and Design

With the 'Extras' Activity, there are three options:

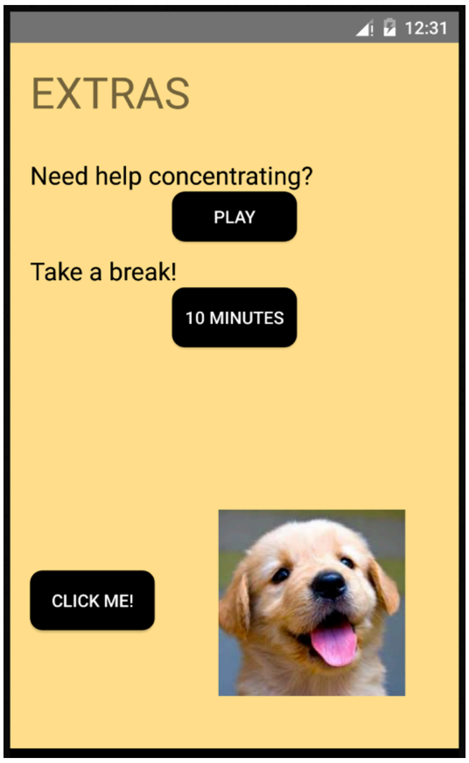
- **Play a song**
  This feature was added because there are many students who prefer listening to music while studying to help them concentrate.
- **Set up a 10-minute alarm**
  It is ideal to have 10-minute breaks in between study session blocks.
- **Look at puppies**
  During exam period, where students are most stressed, TCD had set up 'Puppy Rooms' to help students relax. To imitate that, we added a puppy-picture generator.

Each option has a button to start the task.



When 'Click Me'

button is clicked on ->



The **'PLAY'** button:

In the xml file of this activity, the 'Play' button's onClick event ="startMusic"

In the java file of this activity, inside the **onCreate()** function, a static method **create()** of the **MediaPlayer** class is called.

```
mySound = MediaPlayer.create(this,R.raw.music);
```

This method returns an instance of **MediaPlayer** class. ***`music`*** is the name of the file that will be played. This file is placed in a folder called **raw** in the **res** folder.

Inside the **public class Activity {}** function, a function called 'startMusic' is made. This function starts whenever the 'Play' button is clicked on.

```
public void startMusic(View view) {
    if (mySound.isPlaying()) {      //returns true/false whether the
                                    //song is playing or not
```

```
            mySound.pause();            //if true -> pause song
        }
        else {
            mySound.start();            //if false ->start song
        }
    }
```

The **’CLICK ME!’** button,:

This feature uses the event listener **OnClickListener()**.

```
Button btn =(Button) findViewById(R.id.button2);
btn.setOnClickListener(new View.OnClickListener() {
       //code that generates photos
}
```

Inside the OnClickListener(), there is an OnClick function. On each click, it generates a random number between 1 and 5 (the amount of photos stored) and stores this number in **mystring**.

```
int min= 1;
int max = 5;
Random r = new Random();
int number = r.nextInt(max - min +1) +min;
String mystring = String.valueOf(number);
```

The function then goes to the **if** functions. If the number stored in **mystring** is equal to “1”, show ***images1***, if the number stored in **mystring** is equal to “2”, show ***images2*** and so on. The classes **ImageView** and **bitmap** and the method **setImageBitmap()** are used to display the images on the ImageView widget. The images are stored in a folder called **mipmap** inside the **res** folder.

```
final ImageView imgtable = (ImageView)findViewById(R.id.imageView3);


if (mystring.equals("1")){

    final Bitmap img1 = BitmapFactory.decodeResource(getResources(),
R.mipmap.images1);
    imgtable.setImageBitmap(img1);

}
```

The **’10 MINUTES’** button:

In the xml file of this activity, the ’10 Minutes’ button’s onClick event =”onetimeTimer”

In the java file of this activity, inside the **public class Activity {}** function, a function called 'onetimeTimer' is made. This function starts whenever the '10 Minutes' button is clicked on.

```
public void onetimeTimer(View view){
    Context context = this.getApplicationContext();
    if(alarm != null){
        alarm.setOnetimeTimer(context);
    }

}
```

This function calls up the method **setOnetimeTimer()** in **AlarmManagerBroadcastReceiver.java** which extends as a BroadcastReceiver.

```
public void setOnetimeTimer(Context context){
    AlarmManager am =
(AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
    Intent intent = new Intent(context,
AlarmManagerBroadcastReceiver.class);
    intent.putExtra(ONE_TIME, Boolean.TRUE);
    PendingIntent pi = PendingIntent.getBroadcast(context, 0, intent,
0);
    am.set(AlarmManager.RTC_WAKEUP, System.currentTimeMillis() + 10 *
1000, pi);
}
```

This function sets up the alarm using the class AlarmManager and sets up the alarm so that the phone wakes up after 10 minutes (10 seconds for demonstration). ***RTC_WAKEUP*** delivers the PendingIntent after 10 seconds regardless of whether the phone is sleeping or if the app was exited or not.

The createNotification() method creates the notification which will be activated after 10 seconds is finished. It defines what the notification will look like, what it will say and the alarm sound.

```
//Open the MainMenu Activity when notification is clicked on

PendingIntent notificIntent = PendingIntent.getActivity(context, 0,
        new Intent(context, MainMenu.class), 0);
```

`<uses-permission android:name="android.permission.WAKE_LOCK"/>` is typed in the manifests file to ask for permission to use the alarm of the phone. Also, make sure that the AlarmManagerBroadcastReceiver is labeled as a receiver and not an activity in the java file.

```
<receiver android:name=".AlarmManagerBroadcastReceiver"/>
```

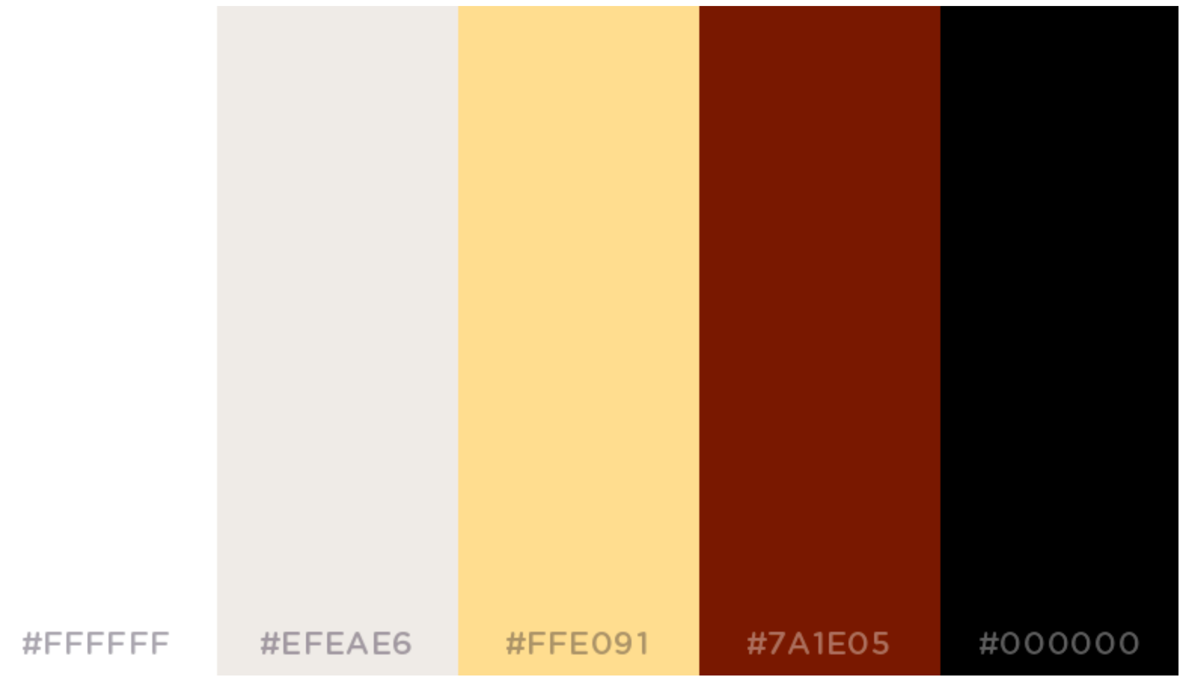
LAYOUT

The layout is done with the mindset of making it clean and easy to use.

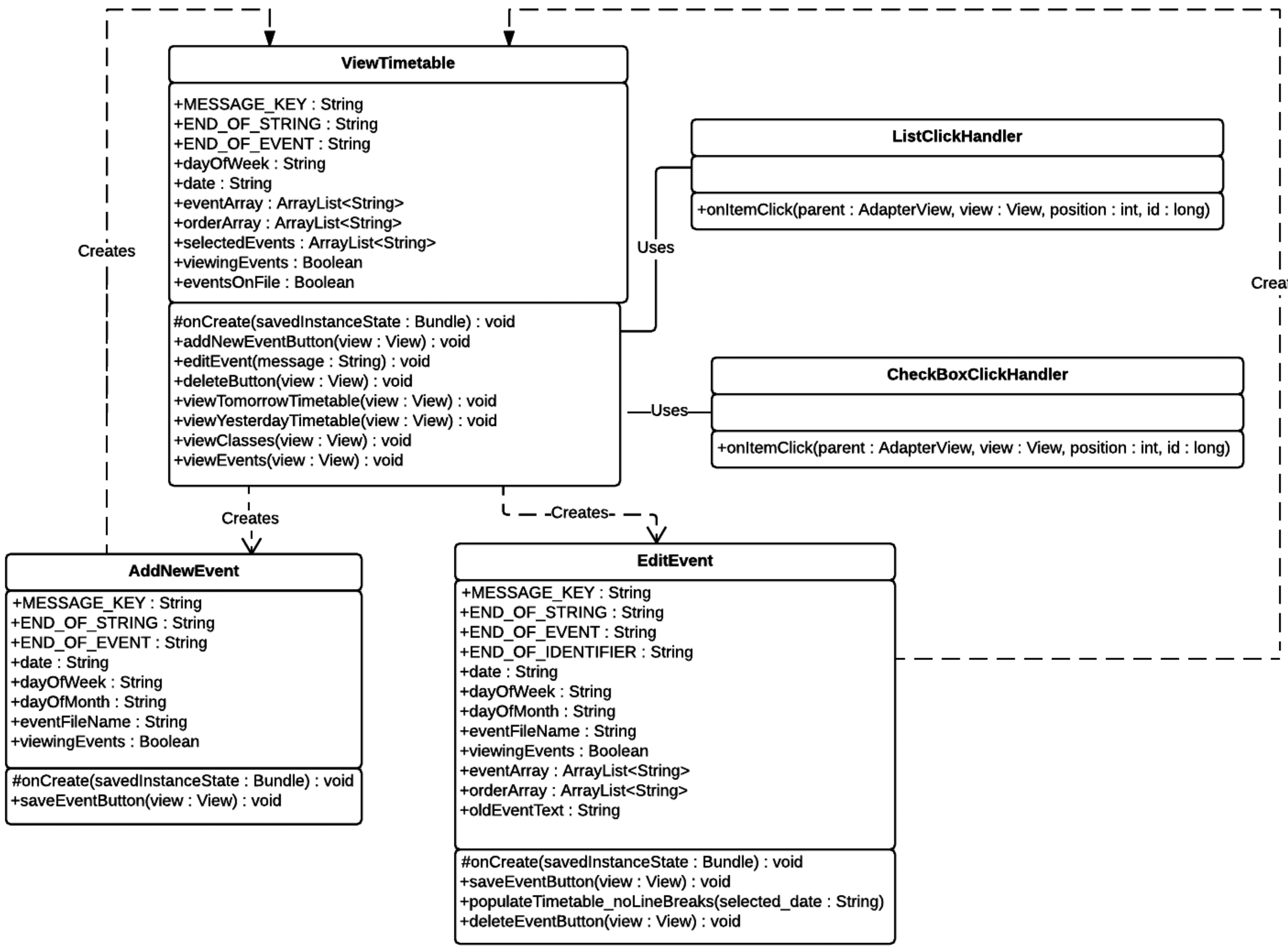
The buttons are edited by using an xml file as the background. The xml file is played inside the **drawable** folder and contains the code to change the colour and to round up the corners of the button.

We also made sure that the buttons are consistent in where they are placed ie the ‘Cancel’ button is always on the right, the ‘Save’ button on the left.

- The colour scheme used:



Timetable Display


ViewTimetable
+MESSAGE_KEY : String
+END_OF_STRING : String
+END_OF_EVENT : String
+dayOfWeek : String
+date : String
+eventArray : ArrayList<String>
+orderArray : ArrayList<String>
+selectedEvents : ArrayList<String>
+viewingEvents : Boolean
+eventsOnFile : Boolean
#onCreate(savedInstanceState : Bundle) : void
+addNewEventButton(view : View) : void
+editEvent(message : String) : void
+deleteButton(view : View) : void
+viewTomorrowTimetable(view : View) : void
+viewYesterdayTimetable(view : View) : void
+viewClasses(view : View) : void
+viewEvents(view : View) : void
ListClickHandler
+onItemClick(parent : AdapterView, view : View, position : int, id : long)
CheckBoxClickHandler
+onItemClick(parent : AdapterView, view : View, position : int, id : long)
Uses
Uses
Creates
Creates
Creates
-Creates-
AddNewEvent
+MESSAGE_KEY : String
+END_OF_STRING : String
+END_OF_EVENT : String
+date : String
+dayOfWeek : String
+dayOfMonth : String
+eventFileName : String
+viewingEvents : Boolean
#onCreate(savedInstanceState : Bundle) : void
+saveEventButton(view : View) : void
EditEvent
+MESSAGE_KEY : String
+END_OF_STRING : String
+END_OF_EVENT : String
+END_OF_IDENTIFIER : String
+date : String
+dayOfWeek : String
+dayOfMonth : String
+eventFileName : String
+viewingEvents : Boolean
+eventArray : ArrayList<String>
+orderArray : ArrayList<String>
+oldEventText : String
#onCreate(savedInstanceState : Bundle) : void
+saveEventButton(view : View) : void
+populateTimetable_noLineBreaks(selected_date : String)
+deleteEventButton(view : View) : void

ViewTimetable - Architecture and Design

Displays the timetables of classes and events for a particular date. Events are unique to a particular date, and are not repeated. Classes are assigned to a day of the week, and occur week after week.

The events for each date are saved to a different text file in internal storage. The date being viewed determines which file is opened. Files are deleted when all of the events for that particular date are deleted by the user.

Event Flow:

1. The activity displays a timetable for the date being viewed, if save data for this date exists.
2. If the user clicks on the "New Event" button, the *AddNewEvent* activity is started, and the *ViewTimetable* activity finishes.
3. If the user clicks on an item in the list, the *editEvent* activity is started, and the *ViewTimetable* activity finishes.

4. If the user long presses an item in the list, checkboxes appear next to each item in the list. Clicking on list items at this point checks or unchecks their corresponding checkboxes.
   - If the "Delete" button is pressed, all items that have been checked are deleted. The activity is restarted to refresh the timetable, and account for the changes.
   - If the "Cancel" button is pressed, the activity reverts to normal behaviour.
5. If the user clicks either of the arrow buttons at the top of the program, the activity restarts and generates either the previous or next day's timetable.
6. If the user clicks the "View Events" or "View Classes" button, the activity restarts and generates the corresponding choice of timetable.

Classes:

public class ViewTimetable extends ActionBarActivity
Implements entire activity, with the exception of handling clicks on items in the list.

- **Subclasses:**

  public class ListClickHandler implements AdapterView.OnItemClickListener
  Implements editing of a list item when it is clicked on by the user.

  public class checkBoxClickHandler implements AdapterView.OnItemClickListener
  Implements checking and unchecking of checkboxes.

Methods of class ViewTimetable:

protected void onCreate(Bundle savedInstanceState)
Parses the intent that started the program to determine the date to be viewed, the corresponding day of the week, and whether to show the event or class timetable. Calls *populateTimetable* and *setDayOfWeekBox* to generate the necessary visual elements. Also sets item-click listeners for list elements.

public void populateTimetable(final String selected_date)
Sets the TextView element that displays the date. Calls *readFromFile* to search for a text file corresponding to the "*selected_date*" string. If the file exists, its contents are parsed and added to the ordered "*eventArray*" ArrayList. Newline characters are interspersed throughout (after each time, location and title) so that the contents of the timetable are formatted correctly.

The *eventArray* is ordered by storing the time corresponding to each event in a separate ArrayList of integers, "*orderArray".* Each element added to *eventArray* is placed in the same position as its corresponding time is placed in *orderArray*.

For example, if an event took place at 11:30, a value of 1130 would be placed in *orderArray*. If another event took place at 14:00, a value of 1400 would be placed in *orderArray*. As *orderArray* is an ArrayList of integers, it is very simple to order it. The corresponding events are then added to the same position in *eventArray*.

For example, if a certain time is placed in position 5 in *orderArray*, the event corresponding to that time will be place in position 5 in *eventArray*. If two or more events share the same time, they will be added one after another in both ArrayLists.

If *eventArray* contains any elements (i.e. if the string specified by *selected_date* matched one of the files in internal storage), it is displayed to the user as a ListView element.

public void setDayOfWeekBox(String myDay)
This method simply sets the TextView element that displays the day of the week.

private String readFromFile(String dateFileName)
This method searches for a text file corresponding to the specified file name. It returns all of the text from said file if it exists, and returns a blank string otherwise.

public void addNewEventButton(View view)
Implements the "New Event" and "New Class" buttons. Finishes the *ViewTimetable* activity and starts the *AddNewEvent* activity. The intent message passed to the activity contains the date, the day of the week and an instruction on whether to add to events or classes.

public void editEvent(String message)
Invoked when the user clicks on an item in the list. Finishes the *ViewTimetable* activity and starts the *editEvent* activity. The contents of the list element that was clicked are passed to the *editEvent* activity as part of the intent message, as well as information on the date, the day of the week and whether the item being edited is an event or a class.

public void changeClicker()
Invoked by the ListView's long-click listener. Puts the timetable into "deletion mode" – checkboxes are set to be visible and the list's click listener is set to check and uncheck boxes.

public void revertClicker()
Takes the timetable out of "deletion mode" – checkboxes are made to vanish, and the list's click listener is set to implement the *editEvent* method once more. Invoked when the user clicks the "Delete" or "Cancel" button when in deletion mode.

public void addToSelectedEvents(String addition)
Marks an event for deletion. Called when the user checks the checkbox next to a list element. Adds the contents of the list element to a "*selectedEvents*" string array.

public void removeFromSelectedEvents(String removal)
Removes an event from the list of those to be deleted. Called when the user un-checks the checkbox next to a list element. Removes any elements from the list array that perfectly match the list element that was clicked on.
<u>Aside</u>: A break should have been added to the if statement in this method. When more than two perfect duplicates were present in a list, more than one would be deleted. The aforementioned break would have resolved this relatively minor bug.

public void deleteButton(View view)
Implement the "Delete" button. Calls the *deleteSelectedEvents* activity.

public void deleteSelectedEvents()
Called when the user presses the "Delete" button when in "deletion mode". To begin, this method calls the *populateTimetable_noLineBreaks* method, which generates two ArrayLists, "*eventArray2*" and "*orderArray2*". Unlike "*eventArray"*, the elements of *eventArray2* do not contain newline characters interspersed throughout. This makes it easier to compare them with the items selected by the user.

These two ArrayLists are used as replacements for the ones used on creation. If any element of *eventArray2* matches an element of *selectedEvents*, the element is removed from *eventArray2* and its corresponding time element is removed from *orderArray2*.

After every item in *selectedEvents* has been examined, *eventArray2* only contains the items the user wishes to keep. The program loops through the remaining elements of *eventArray2* and concatenates the entirety of its contents into a single string. This string is then passed to the *writeToFile_Overwrite* method, which overwrites the contents of that date's save file with this new string.

If *eventArray2* is empty (i.e. all elements in the list have been deleted), then the *deleteEvent* method is called and the save file corresponding to this date is deleted from internal storage.

public void deleteEvent(String myEventFileName)
Deletes the file saved in internal storage that corresponds to the specified file name. Re-populates the timetable afterwards. Called when the activity detects that the last remaining element in a timetable has been deleted.

public void cancelDeletion(View view)
Called when the user clicks the "Cancel" button when the app is in "deletion mode". Calls *revertClicker* to resume normal operation, and makes the "Delete" and "Cancel" buttons disappear.

public void shortToast(String message)
Displays a short toast message to the user. Used multiple times throughout the activity.

public void viewTomorrowTimetable(View view)
Restarts the activity to display the following date's timetable. Called when the user clicks one of the two arrow buttons at the top of the activity.

Determines the date and day of the week, then increments the date by one. Restarts the activity, passing this information as part of an intent. Event/Class view is preserved - if the user is viewing events, the next day's timetable opens to events, and vice-versa.

public void viewYesterdayTimetable(View view)
The inverse of *viewTomorrowTimetable*. Displays the timetable from the day before. Functions very similarly to *viewTomorrowTimetable*.

public void viewClasses(View view)
Called when the user clicks the "View Classes" button. Restarts the activity to display classes instead of events. This button is made to disappear when the user is already viewing classes.

public void viewEvents(View view)
Twin of *viewClasses*. Works in the exact same way. Made to disappear when the user is already viewing events.

public void populateTimetable_noLineBreaks(final String selected_date)
Modified version of *populateTimetable*. Functions very similarly. The key difference is that newline characters are not added to array elements to split up the time, title and location.

Used by the *deleteSelectedEvents* method. Generates two arrays that can be easily compared with data marked for deletion by the user.

private void writeToFile_Overwrite(String data, String eventFileName)
Overwrites the contents of the specified text file with the provided data string. Used by the *deleteSelectedEvents* method.

[DISABLED] public void viewMonthButton(View view)
A method that was disabled in the final build of the application. Takes the user to the *viewMonth* activity, which displays every event the user had on for that month, in a single list. Originally called when the user clicked a "View All Events" button.

Methods of subclass ListClickHandler:

public void onItemClick(AdapterView parent, View view, int position, long id)
Called when the user clicks on an item in the timetable. Passes the contents of the item clicked to the *editEvent* method.

Methods of subclass CheckBoxClickHandler:

public void onItemClick(AdapterView parent, View view, int position, long id)
Called when the user clicks a list element while the timetable is in "deletion mode". Stores the contents of the list element in a string. Also checks/unchecks the associated checkbox in the list.

If the box is being checked, this element is now marked for deletion - the string is passed to the *addToSelectedEvents* method. If the box is being unchecked, this element should not be marked for deletion - the string is passed to the *removeFromSelectedEvents* method.

AddNewEvent - Architecture and Design

Allows the user to add a new event or class to the day's timetable. Always started from the *ViewTimetable* activity. All of the information needed on which file to save to is provided in the intent message that starts this activity.

Event Flow:

1. The user can click either of the "Location" or "Add a Task" fields to add text to them. The user can also click either of the two time spinners to change the selected hours and minutes.

2. If the user clicks "Save", the entered data is added to the day's timetable, and saved to internal storage. The *AddNewEvent* activity then finishes, and the *ViewTimetable* activity restarts.
    - If the user clicks "Cancel", the *AddNewEvent* activity finishes, and the *ViewTimetable* activity is restarted. No data is saved.

Classes:

public class AddNewEvent extends ActionBarActivity
Implements entire activity.

Methods of class AddNewEvent:

protected void onCreate(Bundle savedInstanceState)
Sets layout elements and populates time spinners with set values (1-24 and 1-60 for the hours and minutes spinners, respectively). Parses the message of the intent that started the activity to determine which date is being added to (and hence which file to save user input to). Also determines from the intent message whether the information to be saved is an event or a class.

If the information being saved is an event, the file name that will be saved to will be the date in question (with the ".txt" suffix). In the case of a class, the file name will be the day of the week (along with the ".txt" suffix).

public void saveEventButton(View view)
Implements the "Save" button. Concatenates the contents of the two text fields and the two spinners into a single string, and passes it to the *writeToFile* method, along with the appropriate file name. Uses the *cleanInput* method when reading the text fields, to make sure the data being saved to file will not cause errors in the application.

Afterwards, this method finishes the *AddNewEvent* activity and restarts the *ViewTimetable* activity, placing all appropriate information (date, day of week, event view or class view) in the intent message.

public void writeToFile(String data, String fileName)
Saves the provided data string to the specified text file. If the file does not exist, it is created first. If the file already exists, the data string is appended to the end of its existing contents. Called by the *saveEventButton* method.

public void cancelButton(View view)
Implements the "Cancel" button. Ends the *AddNewEvent* activity and restarts the *ViewTimetable* activity, placing all appropriate information (date, day of week, event view or class view) in the intent message.

public void shortToast(String message)
Displays a short toast message to the user.

public string cleanInput(String input)

Searches for two particular strings in the specified input string and removes them if they are present. The two strings are global variables used by the program to parse user input, referred to as *END_OF_STRING* and *END_OF_EVENT*.

This method is designed to remove them from user input in the extremely unlikely occurrence that the user manages to enter either of the 14-character sequences perfectly. It returns the "cleaned" version of the input, with any occurrences of *END_OF_STRING* or *END_OF_EVENT* removed.

If the *END_OF_EVENT* string is saved to file as part of the user's input, the *ViewTimetable* activity may crash on creation. This method was developed to counteract this.

EditEvent - Architecture and Design

Modified version of the *AddNewEvent* activity. Allows the user to edit or delete an event or class from the day's timetable. The information provided by the intent which starts this activity also provides the detail on which event or class from a particular timetable is being edited.

This activity functions similarly to *AddNewEvent*. They share some methods, but *EditEvent* differs in several others to account for the fact that it deals with existing save data being altered, not new save data being added.

Event Flow:

1. The text fields and time spinners are updated to display the information corresponding to the event or class being edited.
2. The user can click either of the "Location" or "Add a Task" fields to edit their text. The user can also click either of the two time spinners to change the selected hours and minutes.
3. If the user clicks "Save", the entered data is added to the day's timetable, and saved to internal storage. The *editEvent* activity then finishes, and the *ViewTimetable* activity restarts.
    - If the user clicks "Delete", the event or class being edited is deleted from the day's timetable. The *editEvent* activity then finishes, and the *ViewTimetable* activity restarts.
    - If the user clicks "Cancel", the *editEvent* activity finishes, and the *ViewTimetable* activity is restarted. No changes are made.

Classes:

public class EditEvent extends AppCompatActivity
Implements entire activity.

Methods of class EditEvent:

protected void onCreate(Bundle savedInstanceState)
Very similar to the method of the same name from the *AddNewEvent* activity. Difference between the two is that the *EditEvent* version parses the message of the intent that started the activity to find the information corresponding to the event or class being edited. It then sets the contents of the text fields and time spinners to match.

It also saves this information to a global string, "*oldEventText*" (when modifying or deleting an existing item, the application must have a copy of the original text so it can identify which item is to be changed).

public void saveEventButton(View view)
Modified version of the method of the same name from the *AddNewEvent* activity. Concatenates the contents of the text fields and time spinners into a single string. Calls the *cleanInput_specialIdentifier* method to remove any invalid input from the entered values.

Calls the *populateTimetable_noLineBreaks* method to generate ArrayLists of the items already saved to file. Then navigates through the ArrayLists to find the old version of the data being saved, by comparing the ArrayList elements to the "*oldEventText"* global string. Once the old version of the data is found, it is removed and the new version (with the user's edits) is inserted in its place.

The *writeToFile_Overwrite* method is called to save the changes. The *editEvent* activity is then finished, and the *ViewTimetable* activity is restarted, with all relevant information being passed as part of the intent message.

public void populateTimetable_noLineBreaks(final String selected_date)
Calls *readFromFile* to search for a text file corresponding to the "*selected_date*" string. If the file exists, its contents are parsed and added to the ordered "*eventArray*" ArrayList. Newline characters are interspersed throughout (after each time, location and title) so that the contents of the timetable are formatted correctly.

The *eventArray* ArrayList is ordered in the exact same manner as the ArrayList of the same name in the *ViewTimetable* activity.

private string readFromFile(String dateFileName)
This method searches for a text file corresponding to the specified file name. It returns all of the text from said file if it exists, and returns a blank string otherwise.

public string cleanInput_specialIdentifier(String input)
Searches for particular strings in the specified input string and removes them if they are present.

Identical to the *cleanInput* method used by the *addNewEvent* activity, except that it also removes any instances of a third string, *END_OF_IDENTIFIER*, which is also used to parse the string.

private void writeToFile_Overwrite(String data)
Overwrites the contents of the specified text file with the provided data string.

public void deleteEventButton(View view)
Implements the "Delete" button. Calls *populateTimetable_noLineBreaks* to generate ArrayLists of the items already saves to file. Then navigates through the ArrayLists to find the event being edited, and removes it.

The *writeToFile_Overwrite* method is called to save the deletion. If the event deleted was the only item in the timetable, the *deleteEvent* method is called, and the save file itself is deleted.

The *editEvent* activity is then finished, and the *ViewTimetable* activity is restarted, with all relevant information being passed as part of the intent message.

public void deleteEvent(String myEventFileName)
Deletes the file saved in internal storage that corresponds to the specified file name.

public void shortToast(String message)
Displays a short toast message to the user.

[DISABLED] viewMonth - Architecture and Design

An activity that was disabled in the final build of the application. It functioned correctly, but was found to become unstable if the user had saved a large number of events for a given month.

Hence, it was decided to disable this activity. The "View all Events" button in the *ViewTimetable* activity that provided access to *viewMonth* was prevented from being displayed.

Functions similarly to *viewTimetable*. Displays all of the events for a particular month, rather than for one day. Shows the dates on which the events occur, as well as the events themselves. Does not allow the user to add or edit events, but takes them to the timetable for a particular date if they click on the date itself in the list.

Event Flow:

1. The activity displays a list of events for the month being viewed, along with the dates on which they occur, if save data for this date exists.
2. If the user clicks on a date in the list, the *ViewTimetable* activity is started, and the *viewMonth* activity finishes.
3. If the user clicks either of the arrow buttons at the top of the program, the activity restarts and generates either the previous or next month's timetable.
4. If the user clicks the “View Day" button, the *ViewTimetable* activity is started, and the *viewMonth* activity finishes.

Classes:

public class viewMonth extends ActionBarActivity
Implements entire activity, except for handling of clicks on list items.

- **Subclasses:**

  public class ListClickHandler implements AdapterView.OnItemClickListener
  Implements handling of clicks on list items.

Methods of class viewMonth:

protected void onCreate(Bundle savedInstanceState)

Parses the intent that started the program to determine the month to be viewed. Sets layout elements and calls *populateTimetable* to generate and display the list of events saved to file. Also sets item-click listeners for list elements.

public void populateTimetable(final String selected_month)
Uses the *readFromFile* method to search for save data corresponding to all possible dates for this month, going from 1-mm-yyyy to 31-mm-yyyy. If save data is found for a particular day, the date itself is added to the ArrayList of items to be displayed, called "*eventArray*". Then, the save data is parsed to split it into individual events for that day. These are then added to the list of items to be displayed as well, placed after the date they belong to.

The *eventArray* ArrayList is ordered by adding the times corresponding to each event to another ArrayList, "*orderArray*". This ArrayList contains all of the times for each event in the month. The ordering process is similar to that used by the *populateTimetable* method of the *ViewTimetable* activity, only the array positions in *eventArray* are incremented by a certain offset, to account for the fact that times corresponding to multiple dates are being added.

For example, an event on 2/1/2015 should go after all events occurring on 1/1/2015. The offset when adding events from 2/1/2015 to *eventArray* would account for all of the items in *eventArray* from 1/1/2015. The offset is always the size of *eventArray* before any events are added from the date in question - in other words, all items from previous dates.

private String readFromFile(String dateFileName)
This method searches for a text file corresponding to the specified file name. It returns all of the text from said file if it exists, and returns a blank string otherwise.

public void viewNextMonthButton(View view)
Called when the user clicks one of the arrow buttons at the top of the screen. Restarts the *viewMonth* activity to display the month following the one being viewed.

public void viewPreviousMonthButton(View view)
Inverse of the *viewNextMonthButton*. Restarts the *viewMonth* activity to display the month before the one being viewed.

public void viewDayButton(View view)
Implements the "View Day" button. Takes the user back to the day-to-day timetable. Finishes the *viewMonth* activity and starts the *ViewTimetable* activity. Passes all relevant information as part of the intent message that starts the activity.

public void viewTimetable(String selected_date)
Called when the user clicks on a date in the list. Finishes the *viewMonth* activity and starts the *ViewTimetable* activity, taking the user to the timetable corresponding to the date they clicked.

Aside: There appears to be a relatively minor error in *viewNextMonthButton*, *viewPreviousMonthButton*, *viewDayButton* and *viewTimetable*. When creating messages to be passed as part of an intent, they do not reconfigure the day of the week correctly.

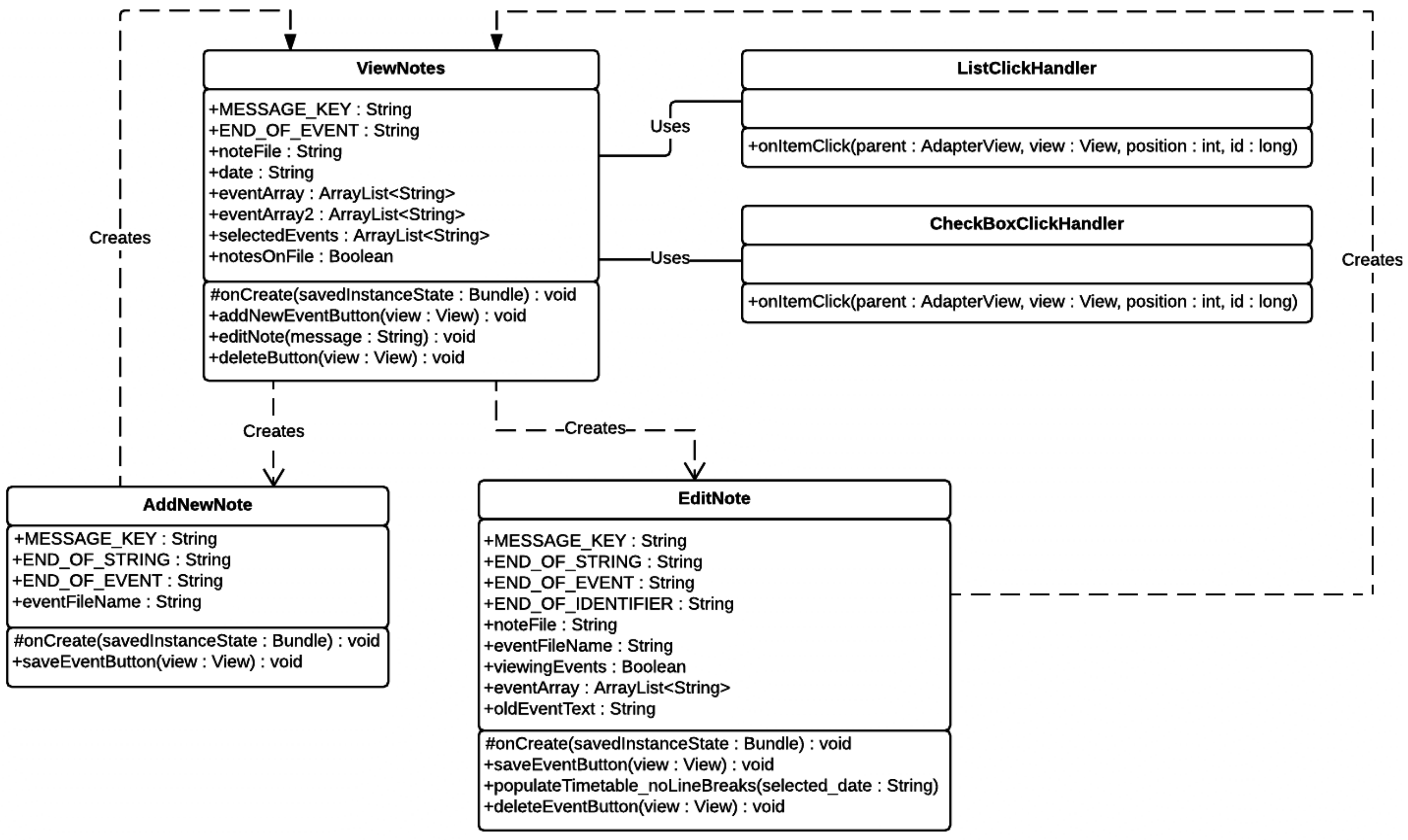
Each of these methods just passes the same day of the week without making any changes to it. It does not account for the change in the day of the week that a change in the date may cause.

/To rectify this error, a new value for the day of the week should be determined, based on the date passed as part of the intent. The exact process by which this is done is quite simple, and features in the *onSelectedDayChange* method of the *DisplayCalendar* activity.

Methods of subclass ListClickHandler:

public void onItemClick(AdapterView parent, View view, int position, long id)
Called when the user clicks on an item in the list. If the contents of the item match a date, the contents are passed to the *viewTimetable* method, which finishes the *viewMonth* activity and starts the *ViewTimetable* activity. If the contents of the item do not correspond to a date, then nothing is done.

Notes Display



ViewNotes
+MESSAGE_KEY : String
+END_OF_EVENT : String
+noteFile : String
+date : String
+eventArray : ArrayList<String>
+eventArray2 : ArrayList<String>
+selectedEvents : ArrayList<String>
+notesOnFile : Boolean
#onCreate(savedInstanceState : Bundle) : void
+addNewEventButton(view : View) : void
+editNote(message : String) : void
+deleteButton(view : View) : void
ListClickHandler
+onItemClick(parent : AdapterView, view : View, position : int, id : long)
CheckBoxClickHandler
+onItemClick(parent : AdapterView, view : View, position : int, id : long)
Uses
Uses
Creates
Creates
Creates
Creates
AddNewNote
+MESSAGE_KEY : String
+END_OF_STRING : String
+END_OF_EVENT : String
+eventFileName : String
#onCreate(savedInstanceState : Bundle) : void
+saveEventButton(view : View) : void
EditNote
+MESSAGE_KEY : String
+END_OF_STRING : String
+END_OF_EVENT : String
+END_OF_IDENTIFIER : String
+noteFile : String
+eventFileName : String
+viewingEvents : Boolean
+eventArray : ArrayList<String>
+oldEventText : String
#onCreate(savedInstanceState : Bundle) : void
+saveEventButton(view : View) : void
+populateTimetable_noLineBreaks(selected_date : String)
+deleteEventButton(view : View) : void

ViewNotes - Architecture and Design

Displays all of the user's notes. A simplified version of the *ViewTimetable* activity. Unlike *ViewTimetable*, *ViewNotes* does not use multiple files to store the user's saved data - it uses a single file, the name of which is hard-coded into the application.

Another key difference is that the notes are not ordered. The most recently changed and added notes are simply placed at the end of the list.

This activity operates in a broadly similar fashion to *ViewTImetable*, but the methods used differ in the fine detail.

As this activity is a modification of *ViewTimetable*, occasional errors made their way into variable and method names (the word "event" is used in place of "note" a number of times). Unfortunately, these mistakes were not rectified before the demonstration of the app during the final lab. To comply with the code saved to the online repository, this document uses the erroneous names, rather than the names that would have been used, had the mistakes been spotted in time.

<u>Event Flow:</u>

1. If the user clicks on the "New Note" button, the *AddNewNote* activity is started, and the *ViewNotes* activity finishes.
2. If the user clicks on an item in the list of notes, the *editNote* activity is started, and the *ViewNotes* activity finishes.
3. If the user long-presses on an item in the list, checkboxes appear next to each of the list items. The user can click on items in the list to check or un-check the corresponding box.
   - If the user clicks the "Delete" button at this point, the items whose boxes have been checked are deleted. The activity restarts to refresh the list and account for the changes.
   - If the user clicks the "Cancel" button at this point, the checkboxes disappear and no changes are made.

<u>Classes:</u>

public class ViewNotes extends ActionBarActivity
Implements entire activity, except for the handling of clicks on items in the list.

- **<u>Subclasses</u>:**

  public class ListClickHandler implements AdapterView.OnItemClickListener
  Implements editing of a list item when it is clicked on by the user.

  public class checkBoxClickHandler implements AdapterView.OnItemClickListener
  Implements checking and unchecking of checkboxes.

<u>Methods of class ViewNotes:</u>

protected void onCreate(Bundle savedInstanceState)
Sets layout elements and calls the *populateNotes* method to display the user's saved data.

public void populateNotes(final String fileName)

Calls the *readFromFile* method to search for save data corresponding to the given file name. The string returned by *readFromFile* is parsed to split it into individual notes, which are then added to an ArrayList.

At this point, if the ArrayList is not empty (i.e. the save file was not blank or non-existent), a ListView is generated to display the list of notes to the user.

private String readFromFile(String dateFileName)
Searches for save data corresponding to the given file name. If any such data exists, it is returned in the form of a string.

public void addNewEventButton(View view)
Causes the *ViewNotes* activity to finish and starts the *AddNewNote* activity.

public void editNote(String message)
Causes the *ViewNotes* activity to finish and the *editNote* activity to start. Passes the contents of the note to be edited as part of an intent message.

public void changeClicker()
Puts the list into "deletion mode". Called when the user long-presses on any item in the list. Makes checkboxes appear next to all list items, and changes the layout so that the "New Note" button is replaced with the "Delete" and "Cancel" buttons. Also sets a new item-click listener, that handles the clicking of the checkboxes.

public void revertClicker()
Takes the list out of "deletion mode", and resumes normal operation. Called when the user clicks either the "Delete" or "Cancel" buttons. Removes checkboxes and sets the normal item-click listener, that handles the editing of list items.

public void addToSelectedEvents(String addition)
Marks an event for deletion. Called when the user checks the checkbox next to a list element. Adds the contents of the list element to a "*selectedEvents"* string array.

public void removeFromSelectedEvents(String removal)
Removes an event from the list of those to be deleted. Called when the user un-checks the checkbox next to a list element. Removes any elements from the list array that perfectly match the list element that was clicked on.

public void deleteSelectedEvents()
Called when the user clicks the "Delete" button while the activity is in "deletion mode". Uses the *populateNotes_noLineBreaks* method to generate an array of the notes saved to internal storage. Then loops through this array, and removes any that match the notes selected by the user for the deletion.

After, the notes that remain are concatenated into a single string. The *writeToFile_Overwrite* method is called, and these notes are saved to internal storage, overwriting any previous save data.

If there were no notes left after the deletion, the save file itself is deleted, using the *deleteEvent* method.

Aside: Like with the *ViewTimetable* activity, the if statement in this method is missing a break. This method suffered from the same problem with multiple duplicates of the same item, and a break would likely have solved this bug.

public void populateNotes_noLineBreaks
Modified version of the *populateNotes* method. Generates a different ArrayList of the notes on file, and does not display this to the user. Otherwise the same. Used by the *deleteSelectedEvents* method.

private void writeToFile_Overwrite(String data, String eventFileName)
Overwrites the contents of the specified text file with the provided data string. Used by the *deleteSelectedEvents* method.

public void deleteEvent(String myNoteFileName)
Deletes the file saved in internal storage that corresponds to the specified file name. Called when the activity detects that the last remaining note on file has been deleted.

public void cancelDeletion(View view)
Called when the user clicks the "Cancel" button while the activity is in deletion mode. Uses *revertClicker* to restore normal operation, and reverts the layout to normal. Clears the array of notes marked for deletion.

public void shortToast(String message)
Displays a short toast message to the user.

Methods of subclass ListClickHandler:

public void onItemClick(AdapterView parent, View view, int position, long id)
Called when the user clicks on an item in the list. Passes the contents of the item clicked to the *editNote* method.

Methods of subclass CheckBoxClickHandler:

public void onItemClick(AdapterView parent, View view, int position, long id)
Called when the user clicks a list element while the activity is in "deletion mode". Stores the contents of the list element in a string. Also checks/unchecks the associated checkbox in the list.

If the box is being checked, this element is now marked for deletion - the string is passed to the *addToSelectedEvents* method. If the box is being unchecked, this element should not be marked for deletion - the string is passed to the *removeFromSelectedEvents* method.

AddNewNote - Architecture and Design

Simplified version of the *AddNewEvent* activity. Allows the user to add a new note to the list displayed by the *ViewNotes* activity.

As with ViewNotes, errors in the naming convention are preserved, to comply with the code used in the final demonstration.

Event Flow:

1. The user can click and add text to the text field.
2. If the user clicks "Save", the entered data is added to the list of notes, and saved to internal storage. The *AddNewNote* activity then finishes, and the *ViewNotes* activity restarts.
   - If the user clicks "Cancel", the *AddNewNotes* activity finishes, and the *ViewNotes* activity is restarted. No data is saved.

Classes:

public class AddNewNote extends ActionBarActivity
Implements entire activity.

Methods of class AddNewNote:

protected void onCreate(Bundle savedInstanceState)
Sets layout elements.

public void saveEventButton(View view)
Implements the "Save" button. Passes the contents of the text field to the *writeToFile* method. Uses the *cleanInput* method when reading the text field, to make sure the data being saved to file will not cause errors in the application.

Afterwards, this method finishes the *AddNewNote* activity and restarts the *ViewNotes* activity.

public void writeToFile(String data, String fileName)
Saves the provided data string to the specified text file. If the file does not exist, it is created first. If the file already exists, the data string is appended to the end of its existing contents. Called by the *saveEventButton* method.

public void cancelButton(View view)
Implements the "Cancel" button. Ends the *AddNewNote* activity and restarts the *ViewNotes* activity.

public void shortToast(String message)
Displays a short toast message to the user.

public string cleanInput(String input)
Searches for two particular strings in the specified input string and removes them if they are present.

editNote - Architecture and Design

Modified version of the *EditEvent* activity, which is itself a modified version of the *AddNewEvent* activity. Allows the user to make changes to the note being viewed, or delete it entirely.

As with ViewNotes and AddNewNote, errors in the naming convention are preserved, to comply with the code used in the final demonstration.

Event Flow:

1. The text field is updated to display the information corresponding to the note being edited.
2. The user can click and add text to the text field.
3. If the user clicks "Save", the entered data is added to the list of notes, and saved to internal storage. The *editNote* activity then finishes, and the *ViewNotes* activity restarts.
   - If the user clicks "Delete", the note being edited is deleted from the list displayed by *ViewNotes*. The *editNote* activity then finishes, and the *ViewNotes* activity restarts.
   - If the user clicks "Cancel", the *editNote* activity finishes, and the *ViewNotes* activity is restarted. No changes are made.

Classes:

public class editNote extends AppCompatActivity
Implements entire activity.

Methods of class editNote:

protected void onCreate(Bundle savedInstanceState)
Sets layout elements. Parses the message of the intent that started the activity to obtain the contents of the note being edited. Makes a copy of the original text.

public void saveEventButton(View view)
Modified version of the method of the same name from the *AddNewNote* activity. Gets the user's input from the text field, and calls the *cleanInput* method to remove any invalid parts of the string.

Calls the *populateTimetable_noLineBreaks* method to generate an ArrayList of the items already saved to file. Then navigates through the ArrayList to find the old version of the data being saved, and replaces it with the version that was edited by the user.

The *writeToFile_Overwrite* method is called to save the changes. The *editNote* activity is then finished, and the *ViewNotes* activity is restarted.

public void populateTimetable_noLineBreaks(final String selected_date)
Calls *readFromFile* to search for a text file corresponding to the "*selected_date*" string. If the file exists, its contents are parsed and added to the ordered "*eventArray*" ArrayList.

private string readFromFile(String dateFileName)
This method searches for a text file corresponding to the specified file name. It returns all of the text from said file if it exists, and returns a blank string otherwise.

public string cleanInput (String input)
Searches for two particular strings in the specified input string and removes them if they are present.

private void writeToFile_Overwrite(String data)
Overwrites the contents of the specified text file with the provided data string.

public void deleteEventButton(View view)
Implements the "Delete" button. Calls *populateTimetable_noLineBreaks* to generate an ArrayList of the items already saves to file. Then navigates through the ArrayList to find the event being edited, and removes it.

The *writeToFile_Overwrite* method is called to save the deletion. If the event deleted was the only item note the user had saved to file, the *deleteEvent* method is called, and the save file itself is deleted.

The *editNote* activity is then finished, and the *ViewNotes* activity is restarted.

public void deleteEvent(String myEventFileName)
Deletes the file saved in internal storage that corresponds to the specified file name.

public void shortToast(String message)
Displays a short toast message to the user.

DisplayCalendar - Architecture and Design

This activity is encapsulated by an overall class, '*public class DisplayCalendar extends ActionBarActivity*'. This activity displays a calendar to the user, using the predefined calendarwidget xml layout element. Upon selection of a day in this calendar using a single click, the user will be brought to the **ViewTimetable.java** activity, where they can see the events for that date.

Methods of class DisplayCalendar:

public void onSelectedDayChange(CalendarView _view, int year, int month, int dayOfMonth)
which defines what should be done when a date in the calendar is clicked on.

protected void onCreate(Bundle savedInstanceState)
which is called upon opening the activity to initialise it (layout, variables, onClickListeners, etc).

The primary functional parts of the activity are described below.

Sending a Date Intent

In order to view the timetable for a given day, the application must create a file for that date/day if one does not already exist. If one does exist, the existing contents should display. This is the principle behind how the activity works.

a. In the *onCreate()* method, a calendarwidget is created from the layout file activity_display_calendar.xml. An intent is created, which will go from this activity to the **ViewTimetable.java** activity.
b. The *onSelectedDayChange()* method is called by the *onDateChangeListener*, which is set in the *onCreate()* method.
    i. In this method, a Calendar instance is created, and set to have date *dayOfMonth*, *month*, *year*. From this, the day of the week is extracted and stored in variable *dayOfWeek* as a string.
    ii. A switch loop is used to indicate what should happen depending on the day number of the week, e.g. "2" = Monday. In this case, if *dayOfWeek* string = "2", the *dayOfWeek* string is now set to "Monday".
    iii. As the calendarwidget goes from months 0-11 rather than 1-12, the month must be incremented to reflected the correct month number, e.g. if month = 10, corresponding to November, must change to month++ = 11 to be consistent with common convention. This complete, a string *finalExtra* is defined to be *dayOfMonth*"-*"month*"-"*year*"-END-"*dayOfWeek*"-END-viewingEvents".
    iv. This string is sent as part of an intent to the **ViewTimetable.java** activity.
c. The **ViewTimetable.java** activity is started using the intent. The string which is sent as part of the intent will be used as the title of the file in which all of the timetable events will be stored for that day.

ToDoList and related Activities

1. ToDoList - Architecture and Design

This activity is encapsulated by an overall class, '*public class ToDoList extends ActionBarActivity*'. All content displayed from this activity is loaded into a ListView array, from a single file stored in the local memory of the device. The file can be changed using methods in this activity to reflect deletion of items. The activity also has links to **EditToDoItem.java** and **AddToDoItem.java**, two separate activities allowing the file to be further updated with new or changed 'To-Do Items'.

Subclasses:

public class ListClickHandler implements AdapterView.OnItemClickListener
This subclass implements the editing of items, by allowing the user to click on a list item to bring them to the **EditToDoItem.java** activity.

public class checkBoxClickHandler implements AdapterView.OnItemClickListener
This subclass implements the selection and deselection of checkboxes corresponding to each item in the list, which allows multiple items to be selected for deletion.

<u>Methods of class ToDoList:</u>

public void changeClicker()
which puts the list into checkbox-selection mode.

public void revertClicker()
which puts the list back into normal clicking mode.
public void editToDoItem(String message)
which brings the user to the **EditToDoItem.java** activity.

public void addToSelectedItems(String addition)
which adds the string from an item to the selectedItems array.

public void removeFromSelectedItems(String removal)
which removes the string from an item, from the selectedItems array.

public void deleteSelectedItems()
which deletes the items that have been checked (those which are in the selectedItems array), using the *deleteItem()* method.

public void deleteItem(String myItemFileName)
which deletes an item’s file in memory and updates the list to account for any changes.

public void populateToDoList(final String selected_title)
which populates the ListView to reflect the items in the ‘file’ at that time.

public void populateToDoList_noLineBreaks(final String selected_title)
which populates the to-do list to match saved values (used in the *deleteSelectedItems()* method to update the list).

private void writeToFile_Overwrite(String data, String itemFileName)
which saves the ‘data’ string to the file ‘itemFileName’.

private String readFromFile(String itemFileName)
which reads and sorts the strings from the ‘file’. Used in *populateToDoList()* to fill the list with items.

public void shortToast(String message)
which implements a short toast to display the ‘message’ text.

public void deleteButton(View view)
which implements a button that calls the *deleteSelectedItems()* method and refreshes the activity to reflect changes.

public void addNewItemButton(View view)
which brings the user to the **AddToDoItem.java** activity and allows a new item to be created.

public void cancelDeletion(View view)
which implements the 'Cancel' button, which exits checkbox selection mode and reverts back to normal clicking mode.

protected void onCreate(Bundle savedInstanceState)
which is called upon opening the activity to initialise it (layout, variables, onClickListeners, etc).

The primary functional parts of the activity are described below.

Reading and Writing to Memory
Each 'To-Do Item' is stored in the file (described above) as a string containing the item's relevant information. Each individual string has appropriate identifiers and sentinels, to allow for correct identification of relevant item data, identification of the start and end of each item string, and the implementation of debugging measures.
Two methods, *readFromFile()*, and *writeToFile_Overwrite()*, are used for reading from/writing to the file.

i. *readFromFile()* method is used by methods *populateToDoList()*, and *populateToDoList_noLineBreaks()*. It takes one parameter – a file name. The method begins by finding the file in memory, and checking whether it contains anything.
    1. If it contains something, the contents of the file are read in by the InputStreamReader, and each line converted to a string.
    2. The contents of the input string are added to the existing input, for every input string in the file.
    3. Once all of the file has been read, the input buffer is closed and all input that has been read in, is converted to a string – the format required for interpretation by the program.

ii. *writeToFile_Overwrite()* method is used in the *deleteSelectedItems()* method, where it is used to rewrite the contents of the file to reflect the updated list of to-do items, i.e. to reflect any deletions of items from the list.
    1. An OutputStreamWriter is used to write the 'data' passed in as a parameter, to the file 'itemFileName', also passed in as a parameter. In this instance, the 'data' is a string which corresponds to a list of all to-do items in the list, in the correct string format for file storage (i.e. including all sentinels and identifiers required for reading from file later).
    2. The OutputStreamWriter writes the 'data' to the file, and then closes.

Display of To-Do Items
The **ToDoList.java** activity uses a ListView array to display the items in the To-Do List in a simple,

ordered list. The implementation of this load-and-display feature of the activity is achieved using one primary method, *populateToDoList()* method. As can be deduced from the name, this method is called to read in the items stored in the ToDoList file, and display them in a list on the screen.

iii. The *populateToDoList()* method takes one parameter, a title as a string. This title corresponds to the title of the file from which it should read the items – in this case, the title should be "myToDoList". The method first searches the internal memory of the device for myToDoList.txt, in order to confirm that the file exists.
    1. A debugging loop is implemented, which, if the file contains no items, will display an empty list, and the words "No Items Saved".
    2. A String array, *myOtherSplit[]*, is implemented, which stores the value before the split "END_OF_ITEM", for each "END_OF_ITEM" encountered.
       An iterative loop, with i from 0 to the length of the split, is implemented, with the input to the array of items (to be displayed in the ListView) being identified each time as the ith element of the
       myOtherSplit[] array.

iv. The *populateToDoList()* method uses an ArrayAdapter to display the itemArray as elements of a ListView. This list format is what the user will interact with, as it will be displayed on the screen when the activity is opened.

v. A second version of the *populateToDoList()* method exists – the *populateToDoList_noLineBreaks()* method. This method populates the contents of a second itemArray list, itemArray2, to compare it with the original itemArray so that any previously saved values do not need to be added again.
   This method is used in the *deleteSelectedItems()* method.

Multiple Selection of Items to Delete

The **ToDoList.java** activity includes methods which allow the real-time selection of checkboxes, in order to delete multiple items as well as individual ones.

vi. The user is able to enter item-selection mode (where all to-do items have a corresponding checkbox), by long-clicking on any item in the ListView. This is done this using a method called the *changeClicker()* method.
    1. The *changeClicker()* method is called initially by the *onCreate()* method. The *onCreate()* method sets an *onItemLongClickListener()* for the ListView 'myList', which displays the to-do items on the screen. The *onItemLongClickListener()* instructions execute <u>only when an item in the list is long-clicked</u>.
       If any item in the list is long-clicked, the *changeClicker()* method is called.
    2. The *changeClicker()* method works by setting an *onItemClickListener()* for the ListView that displays the to-do items on the screen. Initially, an 'unchecked' checkbox is

then assigned to each element of the list.
Once each element has been assigned a checkbox for selection purposes, the 'Cancel Deletion' and 'Delete' buttons are made visible, and the 'Add New Item' button is set to 'gone' visibility.

a. The 'Delete' button is implemented using a separate method, the *deleteButton()* method. This method switches the clicking mode back to normal, and calling the *deleteSelectedItems()* method to remove all of the items currently in the selectedItems array.

3. The *changeClicker()* method is also called by the *revertClicker()* method as part of an *onItemLongClickListener()*, just as in the *onCreate()* method. This listener waits for a long-click, and when it is received, the clicker mode switches to checkbox-selection mode once again.

vii. The **ToDoList.java** activity keeps track of the items that have been clicked by identifying whether the corresponding checkbox for each item has been clicked, and adding the item (i.e. the 'addition'), to an array of 'item' strings. This addition of 'item' strings to the array is done using the method *addToSelectedItems()*. This method has a debugging measure which checks whether the current item has already been added to the list. If so, the item is not added again. There is also a similarly implemented *removeFromSelectedItems()* method, which allows the user to uncheck a box (i.e. deselect an item, the 'removal'), which had been checked, hence removing it from the array of 'item' strings.

viii. The checkBoxClickHandler class defines what happens when a checkbox is clicked.

1. It defines that, when a checkbox is clicked, the *onItemClick()* method should be called.
2. This method extracts, as a string, the details of the corresponding list item to the selected checkbox.
3. The method then checks whether the checkbox is checked or unchecked. If checked, the item is sent to the *addToSelectedItems()* method. If unchecked, the item is sent to the *removeFromSelectedItems()* method.

ix. After selecting a number of items using the checkboxes, the user has the option to delete the checked items. The deletion of items is achieved by implementation of a *deleteSelectedItems()* method, which removes strings associated with each selected item, from the file described above. It then repopulates the screen display to reflected the updated contents of the file, with the previously selected items now removed.

1. The deletion of individual items is achieved using the *deleteItem()* method, which searches the internal memory for the item file name it was given, and if found, deletes it.

The method *shortToast()* is also used, displaying “Items deleted.” if successful, and “Could not delete items” if unsuccessful.

x. The problem of a user deciding that they no longer want to be in selection mode, is dealt with using the *cancelDeletion()* method. This method uses a “sub-method”, *revertClicker()*, in order to return to normal clicking mode, i.e. exiting the checkbox-selection mode. The *cancelDeletion(View view)* method works by calling the *revertClicker()* method, which is designed to bring the user back to normal clicking mode. It does this by:
    1. Setting an *onItemLongClickListener()*, which listens for a long-click by the user. If the long-click occurs, the *changeClicker()* method is called, which brings the user to the multiple-selection, checkbox-clicking mode as before.
    2. Setting the visibility of the buttons ‘Cancel Deletion’ and ‘Delete All’ to *gone*, and the visibility of the ‘Add New Item’ button to *visible*, until the next long-click occurs.

The *changeClicker()* method, described above, is used by the *revertClicker()* method to listen for the next long-click, which will return the user to checkbox-selection mode.

<u>Adding of To-Do Items</u>
The **ToDoList.java** activity includes a method for the definition of an ‘Add To Do Item’ button - the *addNewItemButton()* method.
This method works by defining an intent, which, when the button is clicked by the user, will send the user to the **AddNewToDoItem.java** activity.

<u>Editing of To-Do Items</u>
The activity includes a method, which allows the user to edit an item by clicking on it as it is displayed in the ListView layout on the screen. It is implemented using a ListClickHandler method.

xi. The ListClickHandler method identifies the list item which has been clicked, and extracts its details as a string (the format in which the file stores all ‘item’ details).

xii. This string of ‘item’ details is sent to an *editToDoItem()* method, which sends an Intent to another activity, **EditToDoItem.java**, containing the string. This means that the string can now be operated on by the **EditToDoItem.java** activity and changed appropriately, before being written back to the ‘file’ described in a). Once the edited item string is written back to the ‘file’, the ToDoList.java screen can be repopulated with the information from the updated ‘file’.

Upon launching of the **ToDoList.java** activity, the onCreate() method executes, initialising the activity as follows:

- A predefined layout for the activity is loaded from an xml layout file (activity_to_do_list.xml). A title textbox is also loaded to display at the top of the screen ('Items').
- The ListView is filled with items from the file using the *populateToDoList()* method.
- If there are items in the file that can be loaded into the list, the list layouts are loaded for each list item.
- OnItemClickListeners are set (so that a single click on an item will allow the editing of the selected item), as well as onItemLongClickListeners (to bring the user to checkbox-selection mode).

2. AddToDoItem – Architecture and Design

This activity is encapsulated by the class '*public class AddToDoItem extends AppCompatActivity'*. This activity is opened when the user clicks the 'Add To Do Item' button in the **ToDoList.java** activity screen, which sends an intent to start this activity.

Methods of class AddToDoItem:

public void saveItemButton(View view)
which implements the 'Save' button

private void writeToFile(String data, String itemFileName)
which saves the 'data' string to the file 'itemFileName'

public void cancelButton(View view)
which implements the 'Cancel' button

public void shortToast(String message)
which implements a short toast to display the 'message' text

protected void onCreate(Bundle savedInstanceState)
which is called upon opening the activity to initialise it.

The primary functional parts of the activity are described below.

Adding a New To-Do Item
New to-do items must have some sort of content to display to the user – in this case, the details of the to-do item. For example, a to-do item might be "Send notes to Katie", or "Apply to Amazon before application deadline 21st March 2016", for example. These details consist of a string of

characters, which must be provided by the user. The *saveItemButton(View view)* method is used to achieve this in this activity.

i. The user is presented with an EditText textbox, which hints to the user to enter the details of the to-do item in it.
ii. Once the user has entered text in this field, the method converts this text to a string.
iii. If the string is empty, the string is set to a default value of "No Item Saved". When in the **ToDoList.java** activity, the user can simply click on this item to edit it and add a value to this field.
iv. The string containing the details of the to-do item has "END_OF_STRING" appended onto the end of it as a sentinel, which will be used as an identifier by the *readFromFile()* method of the **ToDoList.java** activity, enabling extraction of the item details.
v. The string is now written to the 'file'. A short toast is displayed using the *shortToast()* method, confirming that the item has been saved.
    1. The *shortToast()* method sets a short duration on a toast, displaying the parameter 'message' as the toast text.
vi. An intent is then used to return to the **ToDoList.java** activity, which should now update itself with the contents of the updated 'file', i.e. with the new item included.

Writing to Memory

Each 'To-Do Item' is stored in the file (described above) as a string containing the item's relevant information. The information is stored in the string in the format described in part a) above, with appropriate identifiers and sentinels. Storing the information in this way means that when reading from the file later on, the data can be extracted appropriately.
The *writeToFile()* method is called by the *saveItemButton()* method, where it is used to write the entered details for a new item to the 'file'.

1. An OutputStreamWriter is used to write the 'data' passed in as a parameter, to the file 'itemFileName', also passed in as a parameter. In this instance, the 'data' is a string which corresponds the item whose details have just been entered, in the correct string format for file storage.
2. The OutputStreamWriter writes the 'data' to the file, and then closes.

Cancelling the Addition of a New Item

The activity includes a 'Cancel' button, which is used to return the user to the **ToDoList.java** activity without adding the new item.
The method works simply by sending an intent to start the **ToDoList.java** activity, hence closing the **AddToDoItem.java** activity.

Upon launching of the **AddToDoItem.java** activity, the onCreate() method executes, initialising the activity as follows:

- A predefined layout for the activity is loaded from an xml layout file (activity_add_to_do_item.xml).

- This layout includes the ‘Save’ and ‘Cancel’ buttons, as well as an EditText textbox, set to maximum input length 200 characters, where the user can enter in the details of the to-do item. There is also a title box above this text field, stating ‘Title’.
- As described in previous steps, the ‘Cancel’ and ‘Save’ buttons are implemented by the *cancelButton()* and *saveButton()* methods respectively, which can be selected by a single click.

3. EditToDoItem – Architecture and Design

This activity was added to the project at a late stage in the development, and so many of the variable names are unintuitive for the purpose (e.g. ‘event’ instead of ‘item’, ‘Timetable’ instead of ‘ToDoList’) , being taken from the **editEvent.java** activity originally without being changed. These have been left in place for consistency with the version demonstrated on 18th December 2015. The activity is encapsulated by an overall class, ‘*public class EditToDoItem extends AppCompatActivity’*. It started from an intent sent from the **ToDoList.java** activity, which occurs when the user single-clicks on an item in the list. The item details are loaded into the text field, and can be edited from this screen. Any changes saved overwrite what was present in the place of the original item in the file before editing. The user can delete the item entirely from this activity if they wish.

Methods of class EditToDoItem:

public void shortToast(String message)
which displays a short toast with the ‘message’ as its text.

public void cancelButton(View view)
which brings the user back to the **ToDoList.java** activity without making any changes to the ‘file’.

public void saveEventButton(View view)
which takes the edited input from the user, deletes the existing item details from the string, and adds the new input in its place, writing these changes to the ‘file’. The user is then returned to the updated **ToDoList.java** activity.

private String readFromFile(String dateFileName)
which reads in the values from the previous version of the ‘file’ whose title is dateFileName (used in the *populateTimetable_NoLineBreaks()* method).

public void populateTimetable_noLineBreaks(final String selected_date)
which populates the contents of the list to match the saved items, if any exist.

private void writeToFile_Overwrite(String data)

which saves the specified ‘data’ string to the file specified when the **EditToDoItem.java** activity was started.

public void deleteEventButton(View view)
which removes the item from the itemArray that needs to be deleted, and updates the list. The activity is then closed, and the user is returned to the **ToDoList.java** activity.

public void deleteEvent(String myEventFileName)
which deletes the item whose title is passed in as a parameter, and updates the list to account for the change (used by the *deleteEventButton()* method).

public String cleanInput(String input)
used as a debugging measure to remove the input if the user enters ‘END_OF_STRING’ or ‘END_OF_EVENT’ as the item details – an extremely unlikely but possible outcome.

protected void onCreate(Bundle savedInstanceState)
which is called upon opening the activity to initialise it.

The primary functional parts of the activity are described below.

Editing an Existing To-Do Item
The entire purpose of this activity is to enable the user to edit and resave a to-do item.

i. The user can enter and delete text in the EditText text field, as with in the **AddToDoItem.java** activity, up to a maximum of 200 characters.
ii. When finished, the user can click on the ‘Save’ button to save the edited item to the ‘file’, which will be used in the **ToDoList.java** activity to repopulate the list displayed to the user. When this button is clicked, the *saveEventButton()* method is called.
    - The input provided by the user is converted to string format.
    - This string is then run through the *cleanInput()* method, which checks to make sure that the input does not contain either “END_OF_STRING” or “END_OF_EVENT”, as these inputs result in unpredictable behaviour. If the input does contain either of these strings, they are removed.
    - If the input field is empty, the string is set to a default value of “No Item Saved”. Appropriate identifiers are then added to the string to be used by the *readFromFile()* method in the **ToDoList.java** activity.
    - The *populateTimetable_noLineBreaks()* method is called to generate an ArrayList of items saved to the file previously. The old version of the to-do item string is removed from the ArrayList by iteration, and the new version added, i.e. the old string is effectively replaced by the new string. The changes are then written to the file, replacing the previous version. This means that when the **ToDoList.java** activity is reopened, it will be updated with the correct values.
    - The **EditToDoItem.java** activity sends an intent to start the **ToDoList.java** activity, and itself closes.

Writing to Memory

Each 'To-Do Item' is stored in the file (described above) as a string containing the item's relevant information. The information is stored in the string in the format described above, with appropriate identifiers and sentinels. Storing the information in this way means that when reading from the file in the **ToDoList.java** activity, the data can be extracted appropriately.

The *writeToFile_Overwrite()* method is called by the *saveEventButton()* method, where it is used to overwrite the entered details for an edited item to the 'file'.

3. The file name, provided by the **ToDoList.java** activity when the **EditToDoItem.java** activity is opened, is identified as the file to be written to.
4. An OutputStreamWriter is used to write the 'data' passed in as a parameter, to the 'file'. In this instance, the 'data' is a string which corresponds the item whose details have just been entered, in the correct string format for file storage.
5. The OutputStreamWriter writes the 'data' to the file, and then closes.

Deleting the Item

The item sent from the **ToDoList.java** activity to be edited can be deleted from within the **EditToDoItem.java** activity, by selecting the 'Delete' button. When the 'Delete' button is selected by the user, the *deleteEventButton()* activity is called.

iii. The item file name is identified. From this, using the *populateTimetable_noLineBreaks()* method, the ArrayList of items already on file is generated.

iv. From the ArrayList, the item corresponding to the item file name is removed. The *populateTimetable_noLineBreaks()* method is called to generate an ArrayList of items which were saved to the file previously. The to-do item string is removed from the ArrayList by iteration through the ArrayList, using the *deleteEvent()* method.
    - The *deleteEvent()* method works by taking the file name of the item as a parameter, and identifying it in the internal memory of the device.
    - If the file exists, the file is deleted. The populateTimetable_noLineBreaks() method is called to generated the updated ArrayList, which will not include the deleted file.
    - If the file doesn't exist, the *shortToast()* method is called, and displays the message 'Item Not Deleted'.

v. The changes are then written to the file, replacing the previous version. This means that when the **ToDoList.java** activity is reopened, it will be updated with the correct values, i.e. with the item removed.

vi. The **EditToDoItem.java** activity sends an intent to start the **ToDoList.java** activity, and itself closes.

Cancelling the Editing of an Item

The activity includes a 'Cancel' button, which is used to return the user to the **ToDoList.java** activity without editing the item.

The method works simply by sending an intent to start the **ToDoList.java** activity, hence closing the

EditToDoItem.java activity.

Upon launching of the **EditToDoItem.java** activity, the onCreate() method executes, initialising the activity as follows:

- The title of the file that is being edited is received as an intent from the **ToDoList.java** activity, and saved to a local variable.
- A predefined layout for the activity is loaded from an xml layout file (activity_edit_to_do_item.xml). This layout includes the 'Save', 'Delete' and 'Cancel' buttons. As described in previous steps, these buttons are implemented by the *cancelButton()*, *deleteEventButton()* and *saveButton()* methods respectively, which can be selected by a single click.
- An EditText textbox, set to maximum input length 200 characters, is displayed where the user can enter in the edited details of the to-do item. There is also a title box above this text field, stating 'Title'. The existing text content of the file before editing, is saved in a local variable for referencing to by other methods.

[DISABLED] NavigationDrawer - Architecture and Design

An activity that was meant to be included in the app but due to shortage of time was deleted from the final version of the App is a side Navigation Drawer, it can be found on GitLab as a separate project where it is running successfully but when trying to implement it with the College Calendar App many issues have arisen, therefore it was decided to focus on enhancing the features we already have in the App rather than waste more time trying to figure out how to implement the Navigation. Below is a short describtion of the activity's classes, subclasses and methods which would have been used given the Navigation Drawer was successfully working.

When a navigation (left) drawer is present, the host activity should detect presses of the action bar's Up affordance as a signal to open and close the navigation drawer. TheActionBarDrawerToggle facilitates this behavior. Items within the drawer should fall into one of two categories:

- <li><strong>View switches</strong>. A view switch follows the same basic policies as list or tab navigation in that a view switch does not create navigation history. This pattern should only be used at the root activity of a task, leaving some form of Up navigation active for activities further down the navigation hierarchy.

- <li><strong>Selective Up</strong>. The drawer allows the user to choose an alternate parent for Up navigation. This allows a user to jump across an app's navigation hierarchy at will. The application should treat this as it treats Up navigation from a different task, replacing the current task stack using TaskStackBuilder or similar. This is the only form of navigation drawer that should be used outside of the root

Right side drawers should be used for actions, not navigation. This follows the pattern established by the Action Bar that navigation should be to the left and actions to the right. An action should be an operation performed on the current contents of the window, for example enabling or disabling a data overlay on top of the current content.

Classes:

public class NavigationDrawer extends Activity

Implements entire activity, except for handling of clicks on list items.

- **Subclasses**:

  public class DrawerItemClickListener implements ListView.OnItemClickListener
  Implements handling of clicks on list items.

Methods of class NavigationDrawer:

protected void onCreate(Bundle savedInstanceState)
Set a custom shadow that overlays the main content when the drawer opens
Set up the drawer's list view with items and click listener
Enable ActionBar app icon to behave as action to toggle navigation drawer
ActionBarDrawerToggle ties together the proper interactions between the sliding drawer and the action bar app icon

public void onDrawerClosed(View view)
Creates call to onPrepareOptionsMenu()

public void onDrawerOpened(View drawerView)
Creates call to onPrepareOptionsMenu()

public boolean onCreateOptionsMenu(Menu menu)
Called whenever we call invalidateOptionsMenu()

public boolean onPrepareOptionsMenu(Menu menu)
If the navigation drawer is open, hide action items related to the content view

public boolean onOptionsItemSelected(MenuItem item)
The action bar home/up action should open or close the drawer.
ActionBarDrawerToggle will take care of this.

private void selectItem(int position)
Update the main content by replacing fragments

public void setTitle(CharSequence title)

protected void onPostCreate(Bundle savedInstanceState)
When using the ActionBarDrawerToggle, you must call it during onPostCreate() and onConfigurationChanged()
Sync the toggle state after onRestoreInstanceState has occurred.

public void onConfigurationChanged(Configuration newConfig)
Pass any configuration change to the drawer toggls

<u>Methods of subclass *DrawerItemClickListener*:</u>

public void onItemClick(AdapterView parent, View view, int position, long id)

The click listener for ListView in the navigation drawer