**Documentation – ToDoList and related Activities**

1. **ToDoList.java**

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’. It consists of 16 methods.

* + *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.
  + Three button implementations:
    - *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).

This activity also makes use of two 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.

The primary functional parts of the activity are described below.

1. **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.
   * + *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.
       - If it contains something, the contents of the file are read in by the InputStreamReader, and each line converted to a string.
       - The contents of the input string are added to the existing input, for every input string in the file.
       - 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.
     + *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.
       - 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).
       - The OutputStreamWriter writes the ‘data’ to the file, and then closes.
2. **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.
   * + 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.
       - A debugging loop is implemented, which, if the file contains no items, will display an empty list, and the words “No Items Saved”.
       - 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.
     + 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.
     + 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.
3. **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.
   * + 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.
       - 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 only when an item in the list is long-clicked.   
         If any item in the list is long-clicked, the *changeClicker()* method is called.
       - 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.
         1. 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.
       - 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.
     + 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.
     + The checkBoxClickHandler class defines what happens when a checkbox is clicked.
       - It defines that, when a checkbox is clicked, the *onItemClick()* method should be called.
       - This method extracts, as a string, the details of the corresponding list item to the selected checkbox.
       - 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.
     + 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.
       - 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.

* + - 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:
      * 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.
      * 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.

1. **Adding of To-Do Items**  
   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.
2. **Editing of To-Do Items**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.
   * + 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).
     + 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).

1. **AddToDoItem.java**  
   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. It consists of 4 methods:
   * + *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.

1. **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.
   * + The user is presented with an EditText textbox, which hints to the user to enter the details of the to-do item in it.
     + Once the user has entered text in this field, the method converts this text to a string.
     + 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.
     + 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.
     + The string is now written to the ‘file’. A short toast is displayed using the *shortToast()* method, confirming that the item has been saved.
       - The *shortToast()* method sets a short duration on a toast, displaying the parameter ‘message’ as the toast text.
     + 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.
2. **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’.
   * + - 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.
       - The OutputStreamWriter writes the ‘data’ to the file, and then closes.
3. **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.

1. **EditToDoItem.java**  
   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.  
   The activity consists of 10 methods:

* *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.

1. **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.
   1. 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.
   2. 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.
2. **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’.
   * + - 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.
       - 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.
       - The OutputStreamWriter writes the ‘data’ to the file, and then closes.
3. **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.
   1. The item file name is identified. From this, using the *populateTimetable\_noLineBreaks()* method, the ArrayList of items already on file is generated.
   2. 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’.
   3. 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.
   4. The **EditToDoItem.java** activity sends an intent to start the **ToDoList.java** activity, and itself closes.
4. **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.