CHAPTER 7

Lists

Topics

- Creating a list
- Iterating over a list with the `for each` loop
- Selecting an item
- Inserting and appending items
- Removing items
- Replacing items
- Searching for an item
- Other list operations

Creating a List

- A list is a single object that contains multiple items of related data.
- To create a list, first need to create a *variable*.
- The variable will hold the list of multiple items.
- To create a list by plugging the `make a list` block into the list variable.
- The `make a list` method is located in the *List* drawer.

Figure 7-1 Create a Variable that Holds a List (*Source:* MIT App Inventor 2)

Creating a List

- Next, you can begin adding items to your list.
- To add a text item to your list, drag a text block (from the *Text* drawer).
- Change the value to the data that you wish to add and plug it in.

Figure 7-2 Add a Text Item to a List (*Source:* MIT App Inventor 2)

Creating a List

There are two steps to make your list visible.

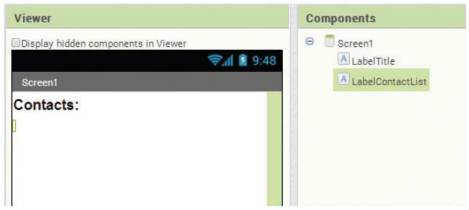
1. Use a component such as a Label to display your list.
2. You must have an event that populates the Label once the list is created.

The `Screen.Initialize` event will work if you want to show the list when your application loads.

Creating a List

Figure 7-3 shows the app in the Designer.

Figure 7-3 Create a List–Design View (*Source:* MIT App Inventor 2)

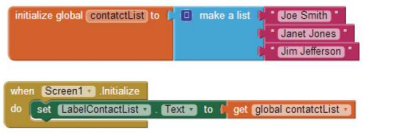


Creating a List

- In Figure a 7-4, we have created a list of names using the `make a list` block.
- We put the names in simple text blocks, and plug those into the `make a list` block.
- Store the entire list in a variable named `ContactList`.
- We use the `Screen1.Initialize` event to set the `LabelContactList.Text` property to the value of the `ContactList` variable.
- Using the `Screen.Initialize` event, we populate the three names as soon as the application loads.

Creating a List

Figure 7-4 Creating a List–Blocks Editor (*Source:* MIT App Inventor 2)



Iterating Over a List with the `for each` Loop

Iteration means to repeat the same process over and over until you reach the result you're looking for. To iterate a list generally means to step through all of the list items, one at a time, until you reach the end.

Iterating Over a List with the `for each` Loop

- The `for each` loop is designed to work with a list.
- When the loop executes, it iterates once for each item in the list.

Figure 7-16 The `for each` Loop (Source: MIT App Inventor 2)

- The `for each` block has a variable named `item` after the words "for each".

Iterating Over a List with the `for each` Loop

The `for each` loop executes in the following manner:

- The `item` variable is assigned the first value in the list.
- The blocks that appear inside the `for each` block are executed.
- The `item` variable is assigned to the next value in the list.
- The blocks that appear inside `for each` block are executed again.
- This continues until the `item` variable has been assigned the last value in the list.

Iterating Over a List with the `for each` Loop

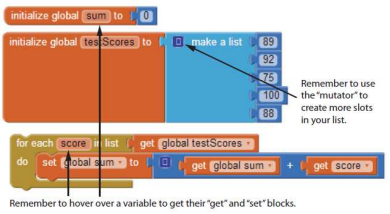
Test Scores Example

- In the Test Scores example, a list of a class's test scores is created and stored to a variable named `TestScores`.
- We will use a `for each` loop to iterate through the list and accumulate the sum of the scores and calculate the class average.

Figure 7-17 Test Scores App Design (Source: MIT App Inventor 2)

Iterating Over a List with the `for each` Loop

Figure 7-18 Test Scores Code in the Blocks Editor (*Source:* MIT App Inventor 2)



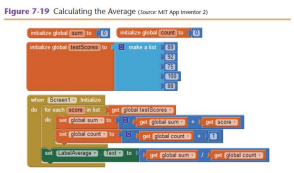
Iterating Over a List with the `for each` Loop

Test Scores Example

1. We create a variable `testScores` to hold the list.
2. We make the list and populate the scores with `number` blocks rather than `text` blocks.
3. We create a variable `sum` to hold the sum of numbers.
4. The `for each` block has an `item` variable and also requires a block, which is the variable that holds a list, to be plugged into it.
5. The block `get global testScores` represents the list that holds the scores, and its "get" block is plugged into the `for each` block.

Iterating Over a List with the `for each` Loop

- After the `for each` iterates through the list, the `Sum` variable will equal all the test scores added together.
- Last, we will need to divide the sum by the number of tests and place that back into the `Text` property of the label as in Figure 7-19



Iterating Over a List with the `for each` Loop

To complete this example:

1. We added a variable named `count`, for the count.
2. We plugged the `for each` loop into the `Screen1.Initialize` event.
3. We added a statement to count by one in each iteration and assigned the results to the `count` variable.
4. We set the results of the some variable divided by the `count` variable to the `LabelAverage.Text` property.

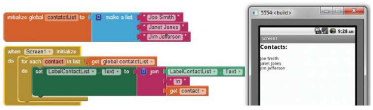
Iterating Over a List with the `for each` Loop

Contact List Example

We are going to print out each list item's value with the return character `\n`, one at a time.

The backslash (\ ) is also known as an *escape sequence*.

Figure 7-20 Adding the Carriage Return (Source: MIT App Inventor 2)



Use the `join text` block to add the item plus the return character `\n` to the label's `Text` property.

Selecting an Item

If you would like to choose a particular item in the list to work with, you can use the `select list item` block.

The first item is at index 1.

Figure 7-26 `select list item` Block (Source: MIT App Inventor 2)

Selecting an Item

- If you only have 10 items in a list and you try to select the item at position 11, your app will crash.
- To avoid this, use the list's length of list function then use if/then logic to stop the attempt if it is out of range.

Selecting an Item

To demonstrate the out of range concept we are going to modify the Contact List app.

We will:

1. Add a number to the left of each name to show the index or place in the list.
2. Add a label and text box for the user to select a contact by entering the index of the person they would like to select.
3. Add a select button to the design and create an event trigger to do the selection.

Selecting an Item

4. Display back to the user the contact they selected.
5. Add logic to check the length of the list before trying to select an item so that we can avoid a crash if the selection is out of range.

Inserting and Appending Items

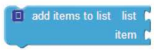
Adding items to a list comes into different forms, adding and inserting.

- ***Appending*** – you can add items to the bottom of the list.
- ***Inserting*** – you can add items somewhere in the middle of the list.

Inserting and Appending Items

Use the `add items to list` block to add a single item at a time.

Figure 7-45 `add items to list` Block (*Source:* MIT App Inventor 2)



Inserting and Appending Items

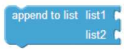
The insert list item block requires you specify the list to insert into, the index (or position) of where you want to insert, and the new item you want to insert.

Figure 7-46 `insert list item` Block (*Source:* MIT App Inventor 2)

Inserting and Appending Items

To append an entire list to the end of a list, use the `append to list` function.

Figure 7-47 `append to list` Block (*Source:* MIT App Inventor 2)

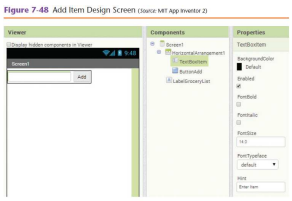


Inserting and Appending Items

Consider an app that allows the user to enter grocery items to a list.

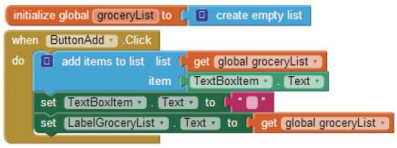
In Figure 7-48 there is a `Textbox` for the grocery item, an Add `Button`, and a label (`LabelGroceryList`) to output the list.

Figure 7-48 Add Item Design Screen (*Source:* MIT App Inventor 2)



Inserting and Appending Items

Figure 7-49 Add Item Blocks Editor (*Source:* MIT App Inventor 2)



Inserting and Appending Items

1. Create a global variable, `groceryList`, and plug in the `create empty list` block.
2. Next, use the `when ButtonAdd.Click do` event handler to call the `add items to list` function.
3. Plug `get global groceryList` and `TextBoxItem.Text` (this is what the user typed in) into the `additems to list` arguments slots.
4. Clear out the `TextBoxItem.Text` property so the user can add another time if they choose.

Removing Items

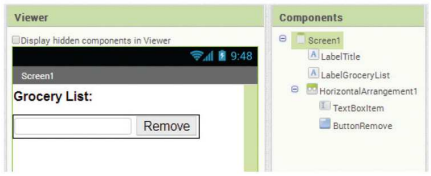
- Each item in a list has an index.
- When an item is removed, the indexes are recalculated starting from position one.
- To remove the third item in a list, use the `remove list items` block.

Figure 7-60 Remove a List Item (*Source:* MIT App Inventor 2)

Removing Items

- Let's look at another grocery list example.
- Add a text box so that the user can choose what item to remove, then provide a *Remove* button. Figure 7-61 shows an example of the app's design.

Figure 7-61 Remove Item Design (*Source:* MIT App Inventor 2)

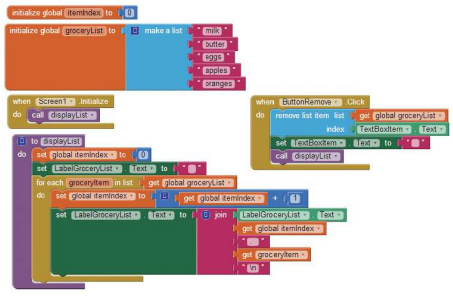


Removing Items

- In Figure 7-61 we have a label, `LabelTitle`, for the "Grocery List" prompt.
- We also have a text box and a button so that the user can indicate which item (by the index number) they would like to remove.
- The Blocks Editor workspace is shown in figure 7-62.

Removing Items

Figure 7-62 Remove Item and Display List (*Source:* MIT App Inventor 2)



Removing Items

1. Two variables are created: one to store the list, and one to store the index.
2. Two events are used: `Screen1.Initialize` and the `ButtonRemove.Click`.
3. Both the initialize and button click events will need to display the list item by item.
4. The procedure will first clear out the `LabelGroceryList` component's `Text` property and set the index to zero.

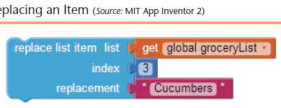
Removing Items

5. The `for each` loop iterates through the list.
6. The `ButtonRemove.Click` event will remove an item from a list, clear out the input field (`TextReplaceItem`), and then call the `displayList` procedure to redisplay they new list.

Replacing Items

- Replacing an item in a list means to change the value of one item to a new value.
- The index positions are unchanged.
- The `replace list items` block requires a list variable, an indexed item to replace, and a new value.

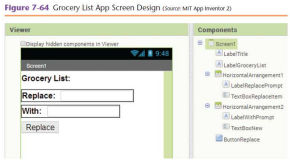
Figure 7-63 Replacing an Item (Source: MIT App Inventor 2)



Replacing Items

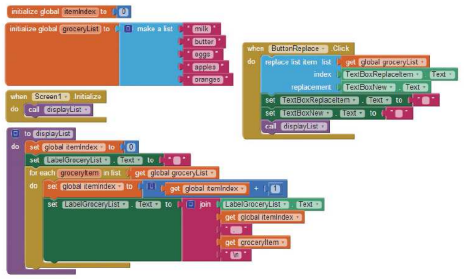
- In Figure 7-64 we have a place for the user to indicate the index, and a text box field for the value of the new item.
- There is also a button that causes the item to be replaced.

Figure 7-64 Grocery List App Screen Design (Source: MIT App Inventor 2)



Replacing Items

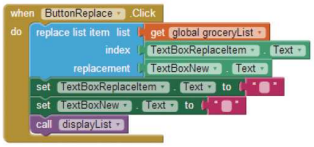
Figure 7-65 Blocks in the Grocery List App (*Source:* MIT App Inventor 2)



Replacing Items

This figure contains essentially the same blocks as in section 7.5, removing items. The only difference is the when `ButtonReplaced.Click` `do` event handler shown in Figure 7-66.

Figure 7-66 `ButtonReplace.Click` Event Handler (*Source:* MIT App Inventor 2)



Replacing Items

- Here we use the replace list item function block.
- The `TextReplaceItem.Text` for the index (this is the number that the user will type in).
- The `TextBoxNew.Text` for the replacement.
- Once we replace the item, we should clear out the `TextReplaceItem.Text` and `TextBoxNewItem.Text` and call the procedure to re-display the list.

Searching for an Item

- When we search for an item in a list, generally we are interested in two things: does the item exist in the list? If so, where or in what position is it?
- To search a list use `is in list?` and `index in list` from the *List* drawer.

Figure 7-79 Searching Blocks (*Source:* MIT App Inventor 2)

Searching for an Item

- The `is in list?` Block will return a true or false value.
- The `index in list` will return an integer, representing the index.

Searching for an Item

- In Figure 7-80, we have a variable named `position`, a list with a few items as `List1`, and the `when Screen1.Initialize do` event handler.
- `Screen1.Initialize` is used to find a position of *Jam*, which is not in the List.
- When this example runs, the result is 0.
- It does not cause an error. It may be beneficial to check whether or not the items exist first as shown in figure 7-81.

Searching for an Item

Figure 7-81 Checking if an Item Exists (*Source:* MIT App Inventor 2)

By adding the check `is in list?` before trying to find a position, we avoid any processing time if the value is return false.

Other List Functions

Other blocks can be used to create and write lists such as a comma-separated value file (CSV).

Figure 7-93 (*Source:* MIT App Inventor 2)

Other List Functions

- The first, list to *csv row* block, will take a list and return text that represents a single row of comma-separated values.
- The `list to csv table` assumes that each item in the list is a text block of comma-separated values and that each list item will represent an entire row.
- The `list from csv row` block will return a list made from comma-separated values.
- The `list from csv table` block will make a list that holds an entire row of the table in each list item.