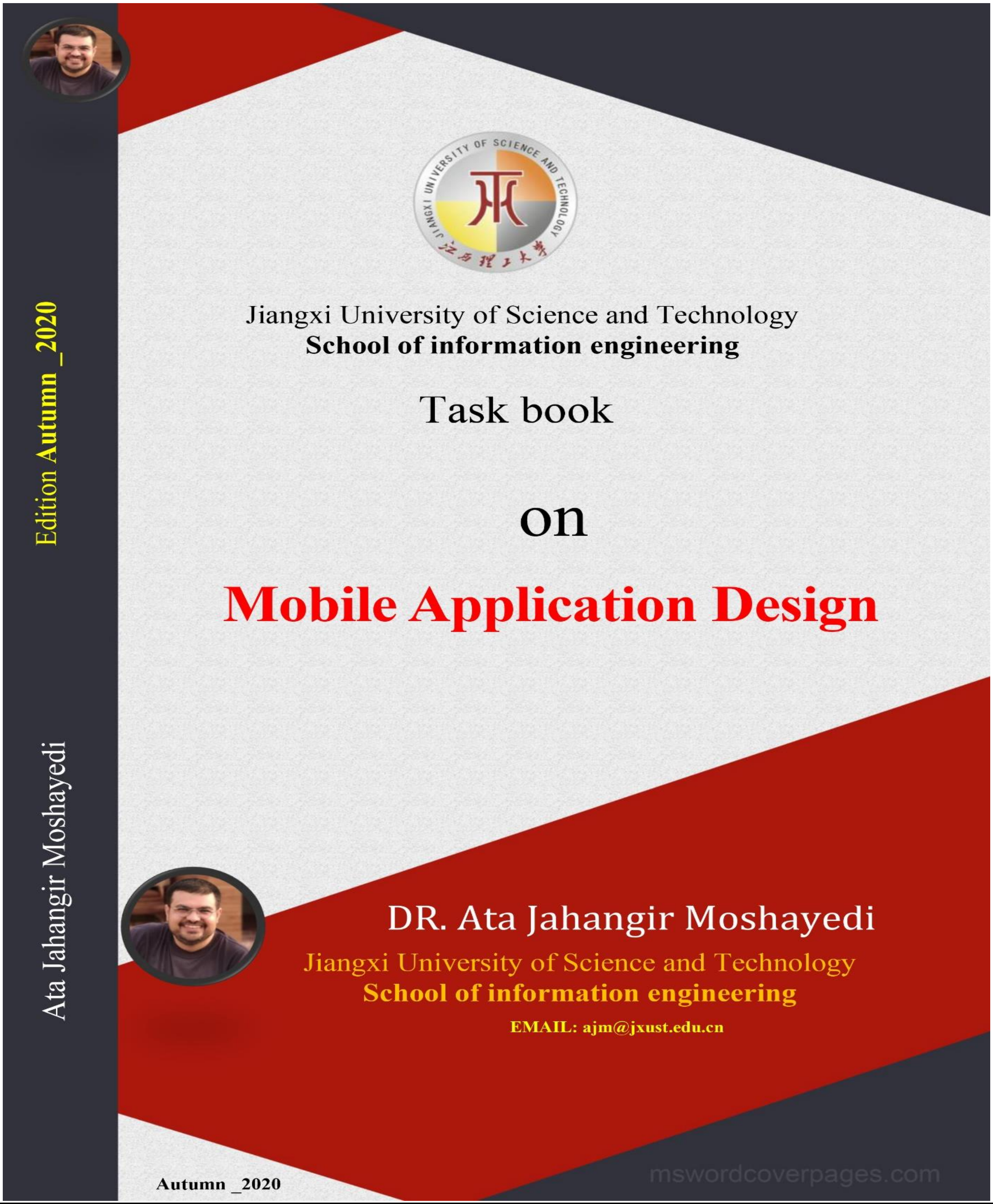


Edition Autumn _2020

Ata Jahangir Moshayedi

Jiangxi University of Science and Technology

School of information engineering

Task book

on

Mobile Application Design

DR. Ata Jahangir Moshayedi

Jiangxi University of Science and Technology

School of information engineering

EMAIL: ajm@jxust.edu.cn

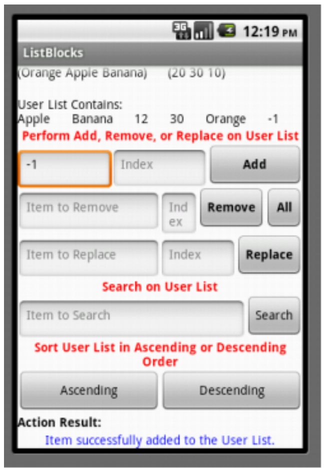
Autumn _2020

List Blocks On App Inventor

List is a necessity in almost every app regardless of what programming language you use. This is the easiest way to create and manipulate a set of values/items/elements in an ordered fashion. Please go over the reference before we start. In this tutorial, we will learn how to create a list, add new items to a list, display list items, replace an item in the list, remove an item or remove everything from a list, search for an item, and sort a list.

The snapshot below is from our tutorial app. I recommend you to download the source of this application ListBlocks and try it on an emulator or on a device so you will have better understanding of list manipulation. In the source, I have added plenty of comments for you to easily understand the implementation. You should be able to see those comments once you load the source in your block editor window.

I tried to keep the source as clean as possible without optimizing it too much for beginners to understand. Once you understand the application, trust me, you will be able to rearrange the pieces in more efficient way. Also, for some operations I tried to control it through **block editor** which actually you can control from **designer window** like checking if a text box contains only numbers which you can do by enabling the **NumbersOnly** option in the designer window for a **TextBox** component, you can specify a **ListPicker's** elements through **ElementsFromString** property in the designer window which I also implemented from block editor window. The purpose of that is to give you alternative ideas which you may need in the future. Also note that 1 and 1.25 are both numbers while list indexes are referred by only **whole** numbers.



(Orange Apple Banana) (20 30 10)

User List Contains:
Apple Banana 12 30 Orange -1

Perform Add, Remove, or Replace on User List

-1 | Index | Add
Item to Remove | Index | Remove | All
Item to Replace | Index | Replace

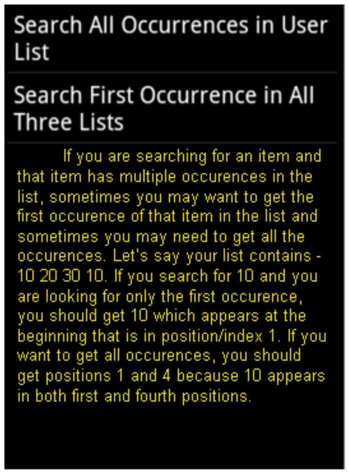
Search on User List

Item to Search | Search

Sort User List in Ascending or Descending Order

Ascending | Descending

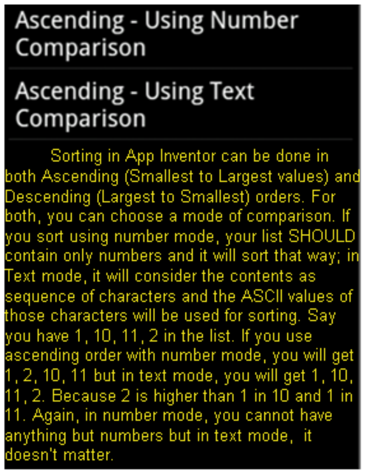
Action Result:
Item successfully added to the User List.



Search All Occurrences in User List

Search First Occurrence in All Three Lists

If you are searching for an item and that item has multiple occurences in the list, sometimes you may want to get the first occurence of that item in the list and sometimes you may need to get all the occurences. Let's say your list contains - 10 20 30 10. If you search for 10 and you are looking for only the first occurence, you should get 10 which appears at the beginning that is in position/index 1. If you want to get all occurences, you should get positions 1 and 4 because 10 appears in both first and fourth positions.



Ascending - Using Number Comparison

Ascending - Using Text Comparison

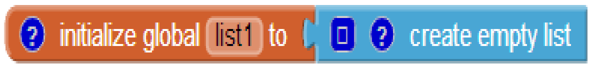
Sorting in App Inventor can be done in both Ascending (Smallest to Largest values) and Descending (Largest to Smallest) orders. For both, you can choose a mode of comparison. If you sort using number mode, your list SHOULD contain only numbers and it will sort that way; in Text mode, it will consider the contents as sequence of characters and the ASCII values of those characters will be used for sorting. Say you have 1, 10, 11, 2 in the list. If you use ascending order with number mode, you will get 1, 2, 10, 11 but in text mode, you will get 1, 10, 11, 2. Because 2 is higher than 1 in 10 and 1 in 11. Again, in number mode, you cannot have anything but numbers but in text mode, it doesn't matter.

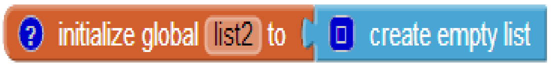
Note that the comments in yellow in the image above are not part of the application.

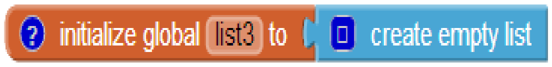
Contents:

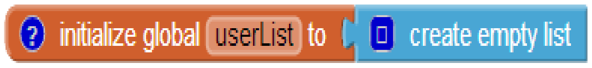
1. Create a List

Creating a list is as simple as this-

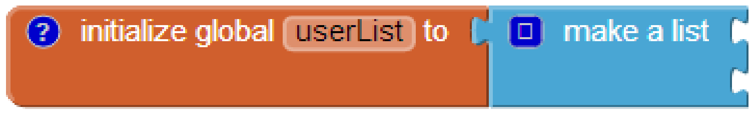




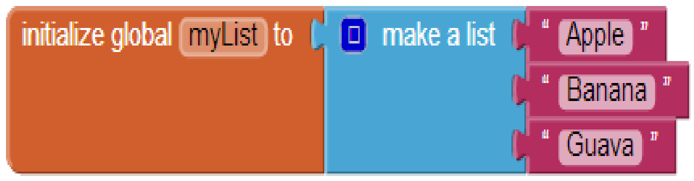




We have just created four empty lists above, named **list1**, **list2**, **list3**, and **userList**. “**create empty list**” block can be found in Lists drawer in your blocks editor. Those lists are currently empty as we haven’t added any items yet. Note, you can also use “**make a list**” block to create an empty list. Just don’t put any item when using “**make a list**” block for creating empty list like this-



What if you don’t want to create an empty list, rather you want to create a list with items in it? Well, no worries. You can use “**make a list**” block to do so.

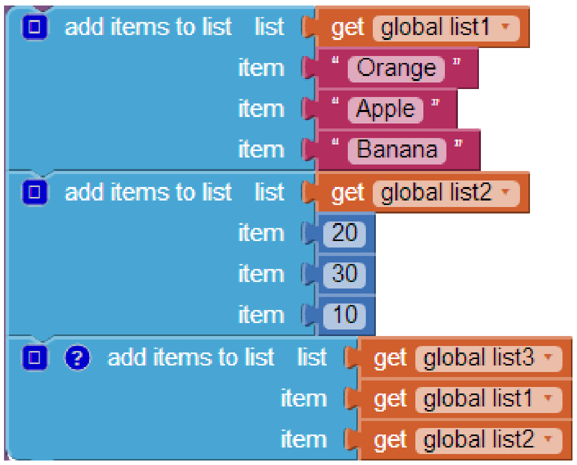


In the above image, we used “make a list” block to create a list named myList with three items – **Apple**, **Banana**, and **Guava**. If we didn’t put any item in the make a list block, it would create an empty list just like the way we did with create empty list block. Note that, you can put as many items as you want in make a list.

Tips: *When you name a variable, try naming it with a purpose. Say you need a list of prices of different fruits, you can name that list variable – fruitsPriceList so that later you don’t have to recall the purpose of that list, specially when you have lots of variables in your block editor. Also by the name of the variable, one should know that variable represents a list.*

2. Add Items to a List

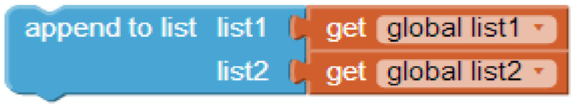
Adding an item can be done different ways.



```
add items to list  list  get global list1
                   item  "Orange"
                   item  "Apple"
                   item  "Banana"
add items to list  list  get global list2
                   item  20
                   item  30
                   item  10
add items to list  list  get global list3
                   item  get global list1
                   item  get global list2
```

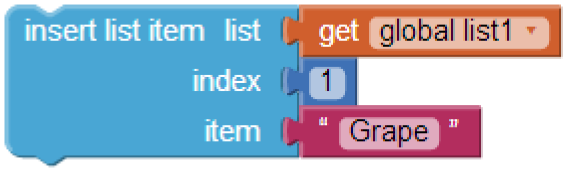
If you look at the image above, we have populated all three lists with different items. First one will have a list of three items which are Orange, Apple, and Banana; the second one is consisted of another three items – 20, 30, and 10. The last one, **list3** is consisted of two items which are actually **lists**. Yes, you can have a **list of lists**. If you take the first item from list3 which is also a **list**, you will get a list of three items which contains Orange, Apple, and Banana.

Now, look at the image below-



```
append to list  list1  get global list1
                list2  get global list2
```

The **difference** between **add** items to list and **append** to list is that for add items to list, you can add both individual items and lists, but for append to list, you can only add another list into a list. add items to list will consider a list as an item but append to list will take the contents from a list and add them as separate items. To make things clear, in the above image, list1 is the destination list where the contents of list2 will be added at the end of list1. That means, list1 will contain whatever items it had before plus the items from list2. Note that list2 will have **no change** in its contents. If things are not clear, try it for yourself and see how they behave.   Both *add items to list* and *append to list* add an item at the end of the list. What if we want to add an item in a certain **position**? That's when we need **insert list item** block. insert list item is used when you want to add an item into a list in a certain position. Let's say you want to add a new item "Grape" into list1 at position 1. This is how you will do it-



```
insert list item  list   get global list1
                  index  1
                  item   "Grape"
```

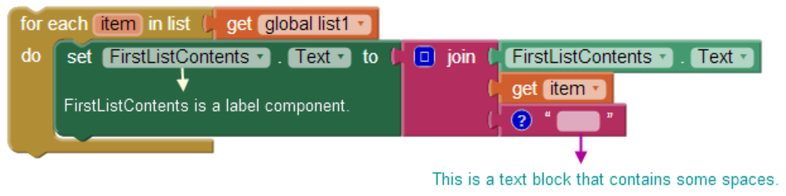
As you have noticed, insert list items takes **3 arguments** – the list where the new item will be added, the index or position where the item should be added, and the new item that should be added. Since we wanted our new item "Grape" to be added at the beginning of the list, we specified 1 as the index/position. Now our list1 contains four items – Grape, Orange, Apple, and Banana. Note that Grape is the first item in list1 and everything else got moved down by 1 because of the insert operation. Before we inserted Grape, Orange was in the first position, but now Orange is in the second position.

Tips: *Always apply whatever you have recently learned. Don't try to learn everything first and then apply.*

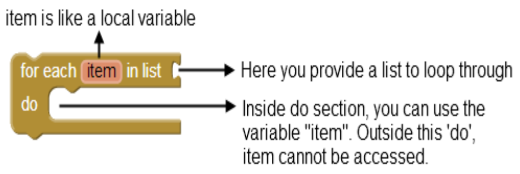
3. Display List Items

A fancy way to display list items would be using ListView component. Here's a **detail tutorial** on ListView.

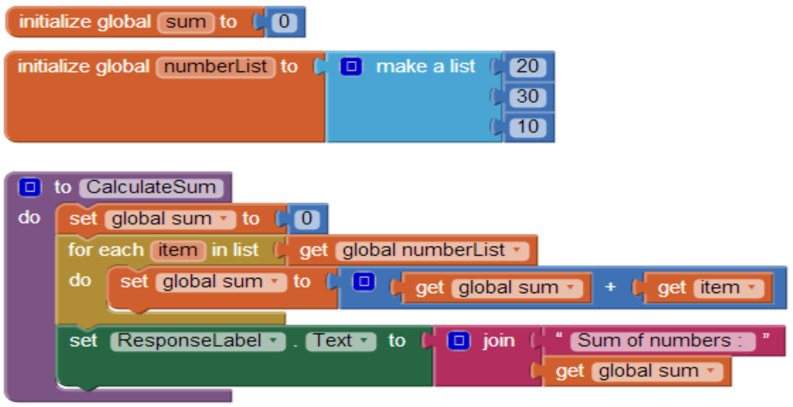
In the example below, we used a Label component named **FirstListContents** to display the contents of **list1**.


for each item in list
get global list1
do
set FirstListContents . Text to
join
FirstListContents . Text
get item
" "
FirstListContents is a label component.
This is a text block that contains some spaces.

for each item and **for each number** both can be used for iterating/looping through a list. Here we used for each item as we are not concern about the indexes/positions of an item. When we will **sort** a list, we will be using **for each number** instead as we'll need the **indexes** of different items in the list for manipulation. You can still use for each item for sorting, but you must define some local or global variables to access the indexes.   for each item is pretty simple. You provide a list to loop through and there's a local variable named "item" given to you that holds the value of an item. Note that you can change the name "item" to anything you like. Remember "item" is a **local** variable, you cannot use it outside the **scope**. In for each block, the scope is inside the **do** section of for each block.


item is like a local variable
for each item in list
do
Here you provide a list to loop through
Inside do section, you can use the variable "item". Outside this 'do', item cannot be accessed.

To give you an idea, let's say you want to get the sum of all items in a list. This is how you would do it-



```
initialize global sum to 0
initialize global numberList to make a list 20
                                            30
                                            10
to CalculateSum
do  set global sum to 0
    for each item in list get global numberList
    do  set global sum to get global sum + get item
    set ResponseLabel . Text to join "Sum of numbers : "
                                     get global sum
```

We have a list named **numberList** which contains three items – 20, 30, and 10. We wrote a procedure named **CalculateSum** that sums up the numbers in our list and then displays the sum. We declared a global variable named **sum** and initialized it to **0**. **Initializing** means assigning a value the first time when you create a variable. Inside the procedure, we again set the value of sum to 0. You might think the initial value of sum was already 0 when we defined the sum variable outside CalculateSum procedure. Well, you may want to use the same procedure again and again. In that case, it’s safe to **reinitialize** a global variable unless the previous value of that global variable should be retained. We used for each item block to loop through each item in our numberList and added to the value of sum. The execution of the above procedure would look like this-

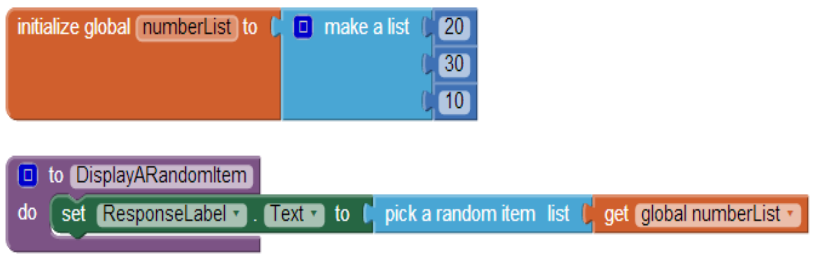
```
sum = 0

sum = sum + value at position 1 of numberList (0 + 20 = 20)
(Now the value of sum is 20)

sum = sum + value at position 2 of numberList (20 + 30 = 50)
(Now the value of sum is 50)

sum = sum + value at position 3 of numberList (50 + 10 = 60)
(Now the value of sum is 60)
```

The final value of sum will be **60**.   Let’s say you want to display a **random** item from a list. That’s how you would do it-

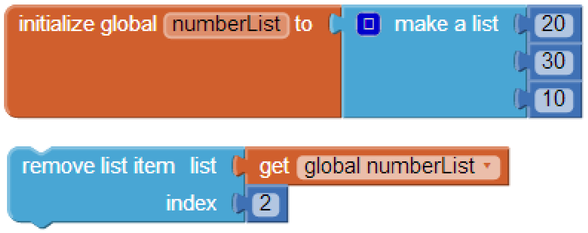


```
initialize global numberList to make a list 20
                                            30
                                            10
to DisplayARandomItem
do  set ResponseLabel . Text to pick a random item list get global numberList
```

“**pick a random item**” block gets a random item from a given list.

4. Remove List Items

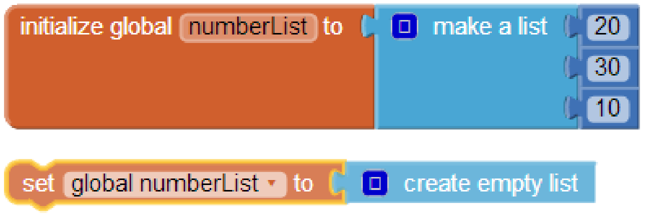
To remove an item, all you need to know is the **index/position** of the item you want to remove. If you want to remove number 30 from **numberList** below, you'll have to pass **2** as the **index** as you can see **30** is in position/**index 2** in the **numberList** below.



```
initialize global numberList to  make a list  20
                                              30
                                              10

remove list item  list  get global numberList
                  index 2
```

In a real world application, you may have to validate if a certain index/position exists in a list before you actually try to remove an item. If you have already downloaded the application that I mentioned at the top of this tutorial, you should see how I validated a given index before I try to remove an item. To remove everything from a list, simply set the **value** of the list variable to a **make a list** block without any item or just use **create empty list** block. In the image below, we **removed all** items from numberList by setting the value of numberList to create empty list block.

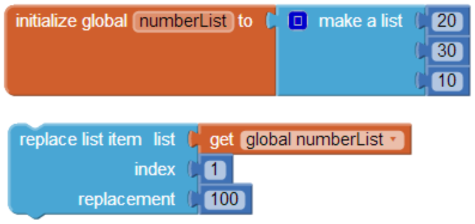


```
initialize global numberList to  make a list  20
                                              30
                                              10

set global numberList to  create empty list
```

5. Replace an Item

To replace a list item, you need to provide with **three arguments** – the list where you want to replace an item, the index of an existing item in the list, and the new item you want to replace an existing item with.



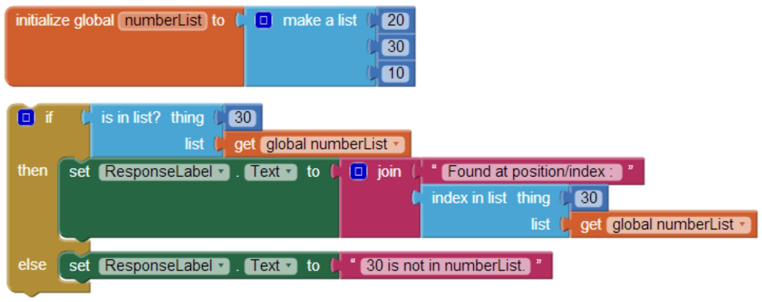
```
initialize global numberList to  make a list  20
                                              30
                                              10

replace list item  list  get global numberList
                   index 1
             replacement 100
```

In the image above, initially we had three items 20, 30, and 10 in numberList. We used replace list item block to replace the item at index 1 which was 20 with a new item 100. Now our numberList contains 100, 30, and 10.

6. Search for an Item

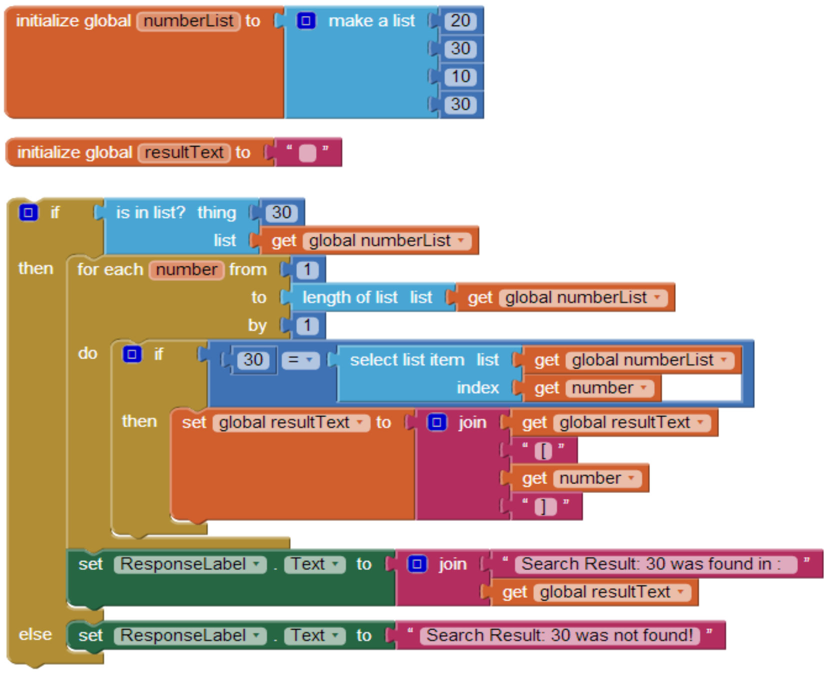
Searching for an item can be a little tricky because sometimes you will need to know if a particular item exists in a list, if it does, then you might be interested in knowing where or in which position that item exists. If you have multiple **occurrences** of the same item in a list, you may also need to know all the positions where the item exists in the list.



```
initialize global numberList to make a list 20
                                            30
                                            10

if   is in list? thing 30
                 list  get global numberList
then set ResponseLabel . Text to join "Found at position/index : "
                                      index in list thing 30
                                                    list  get global numberList
else set ResponseLabel . Text to "30 is not in numberList."
```

In the above image, you can see we are trying to search for the item 30 in numberList which contains three items 20, 30, and 10. Before we started searching, we used **is in list?** block to check if that **item exists** in **numberList**. Block **index in list** gives us the first occurrence of 30 in the list if it exists. If the item doesn't exist, it gives us 0. Since numberList above contains 30 in the second position, index in list will give us 2. As of now, we only searched for the **first occurrence**. What if our list contained 30 two times? How can we get all two positions? Well, we have to use for each block and loop through each item in the list, then compare if the item we are searching for is the same as the item currently in the loop; if equal we get and store the index for later use. We do this for all items in the list.

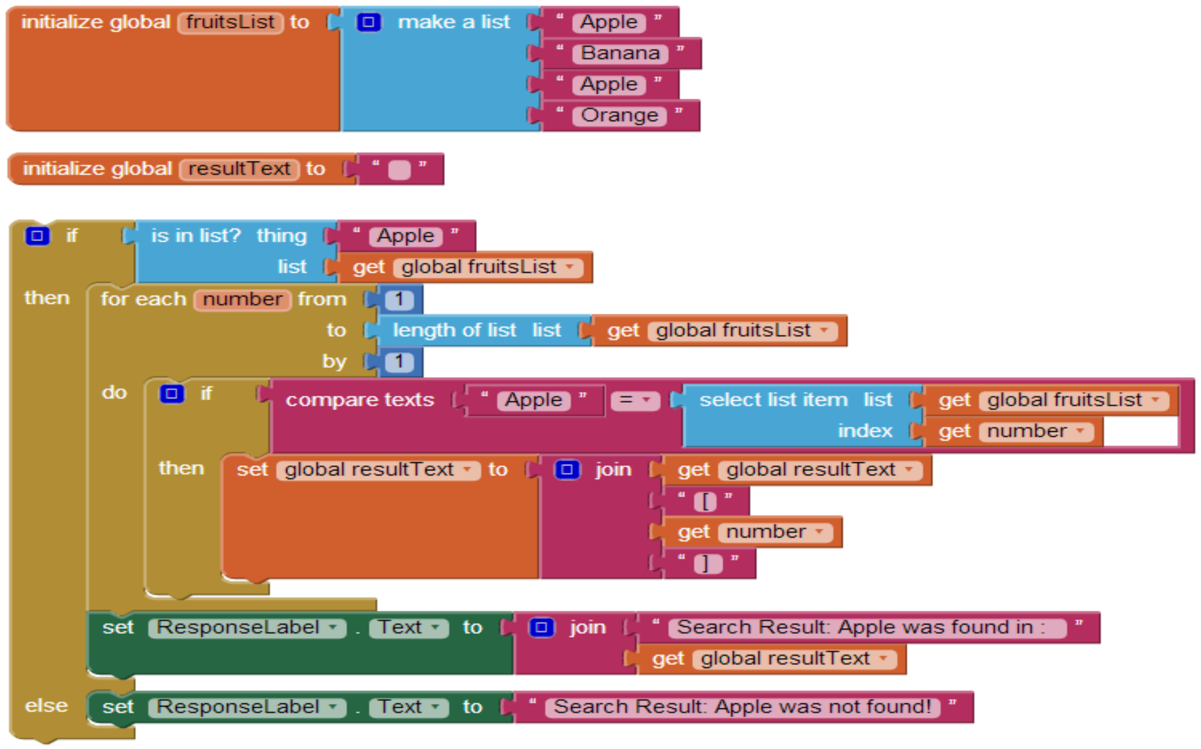


```
initialize global numberList to make a list 20
                                            30
                                            10
                                            30

initialize global resultText to " "

if   is in list? thing 30
                 list  get global numberList
then for each number from 1
                      to   length of list list get global numberList
                      by   1
     do   if   30 = select list item list  get global numberList
                                     index get number
          then set global resultText to join get global resultText
                                             " [ "
                                             get number
                                             " ] "
     set ResponseLabel . Text to join "Search Result: 30 was found in : "
                                      get global resultText
else set ResponseLabel . Text to "Search Result: 30 was not found!"
```

As you can see above, 30 appears twice in the numberList, in the second and in the fourth positions. As usual, we first check if 30 exists by using is in list block. If it does, we use a for each block and compare each item to see if the item is 30. If it is, we store it in a global variable named resultText by appending. In this case, the final value of resultText would be [2] [4] because 30 exists in 2nd and 4th positions. The above **numberList** as the name implies, contains only **numbers**. What if we had **alphanumeric/text** items? In that case, we would need to use the **compare texts** block from text drawer. See below.

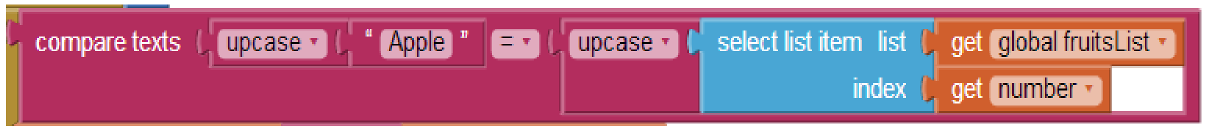


```
initialize global fruitsList to  make a list  " Apple "
                                              " Banana "
                                              " Apple "
                                              " Orange "

initialize global resultText to  " "

if    is in list?  thing  " Apple "
                   list   get global fruitsList
then  for each number from  1
                      to    length of list  list  get global fruitsList
                      by    1
      do  if    compare texts  " Apple "  =  select list item  list   get global fruitsList
                                                               index  get number
          then  set global resultText to  join  get global resultText
                                                " [ "
                                                get number
                                                " ] "
      set ResponseLabel . Text to  join  " Search Result: Apple was found in : "
                                         get global resultText
else  set ResponseLabel . Text to  " Search Result: Apple was not found! "
```

Our search above **doesn't** ignore **casing**. Meaning "**apple**" is not the same as "**Apple**". If you don't care about casing and you want "apple" to be the same as "APPLE", or "Apple", simply convert both sides of comparison to **uppercase** or **lowercase** before you compare. Choose one, if you choose upcase then use it for both sides. You can find both **upcase** and **downcase** in the Text drawer. In the above example, if you didn't want to care about casing, you would change the comparison to-



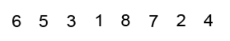
```
compare texts  upcase  " Apple "  =  upcase  select list item  list   get global fruitsList
                                                               index  get number
```

If you have noticed both sides used upcase. All we are doing is that we are converting "Apple" in the left side to "APPLE" and also converting the item in the right side to have also all uppercase letters. So, if the item in the right side was "apple", it will become "APPLE". I used upcase, you can also use downcase as long as you use the same case for both sides in comparison.

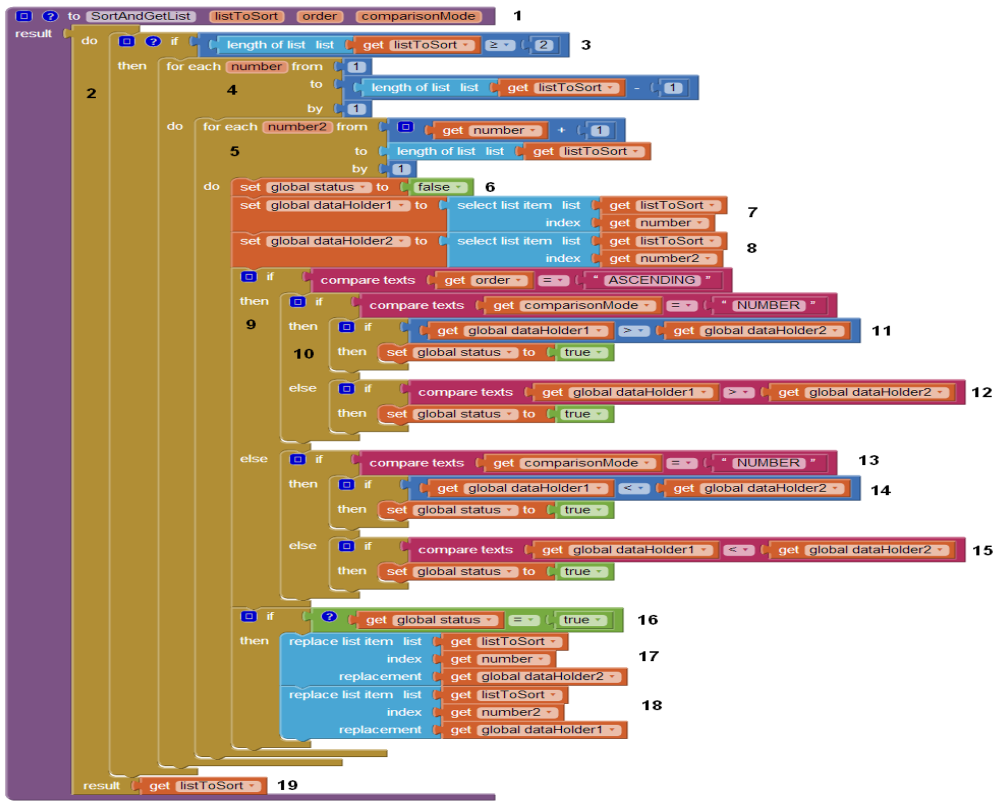
7. Sort a List

This tutorial assumes that you know the basics of list. If not, please go over the basic list tutorial. Also you need to know how to use for each number block. Sorting is arranging of items in a specific order/sequence. There are many algorithms that can be used to sort a list. Here, we will be using Bubble Sort. Bubble Sort is very easy. You start at the beginning of the list, compare each pair of adjacent items and swap if they are not in order. You stop comparing when there is no more items left to compare.

Below is a gif image taken from Wikipedia that shows comparing each pair and swap them if the latter one is smaller/lower than the former one so that the numbers are arranged from smallest to highest which is known as ascending order.



Now that you know about bubble sort, we will go explain how we came up with the procedure **SortAndGetList** below, step by step. This sort procedure is kind of **generic**, meaning you can use it for sorting **numbers** and **text** both. You can use this procedure for sorting in both **ascending** and **descending** orders. Please take note of the numbers in **black** in the image below as these will be used to describe the purpose of each set of blocks.



```
to SortAndGetList listToSort order comparisonMode                                   1
result
  do
    if length of list list get listToSort ≥ 2                                        3
    then for each number from 1
  2                  to length of list list get listToSort - 1                       4
                     by 1
         do for each number2 from get number + 1
                             to length of list list get listToSort                   5
                             by 1
            do set global status to false                                            6
               set global dataHolder1 to select list item list get listToSort
                                                          index get number           7
               set global dataHolder2 to select list item list get listToSort
                                                          index get number2          8
               if compare texts get order = "ASCENDING"
               then if compare texts get comparisonMode = "NUMBER"
                9   then if get global dataHolder1 > get global dataHolder2          11
                    10   then set global status to true
                    else if compare texts get global dataHolder1 > get global dataHolder2   12
                         then set global status to true
               else if compare texts get comparisonMode = "NUMBER"                   13
                    then if get global dataHolder1 < get global dataHolder2          14
                         then set global status to true
                    else if compare texts get global dataHolder1 < get global dataHolder2   15
                         then set global status to true
               if get global status = true                                           16
               then replace list item list get listToSort
                                      index get number                               17
                                      replacement get global dataHolder2
                    replace list item list get listToSort
                                      index get number2                              18
                                      replacement get global dataHolder1
  result get listToSort                                                              19
```

We used a procedure result block for sorting. Our procedure SortAndGetList sorts the list in a given order and returns the sorted list. It takes three arguments as follows:

1.listToSort – This is the list that we want to sort.

order – This tells us how to sort, ascending or descending.

comparisonMode – There are times we will be expecting a list to contain only numbers, sometimes only texts/strings, and sometimes a mixture of both. For numbers, we can easily use Math block's different operators (=, ≠, >, ≥, <, ≤) to **compare** two items, but we cannot use those for comparing text/strings as texts are not numbers. In that case, we should use the operators from Text block. For Text block, the comparison uses the **alphabetical representation** of each **character** in a piece of text. Please go over the documentation on Text block for better grasp. In Text mode, "**a**" is **higher** than "**A**"; "**2**" is **higher** than "**11**". Each character has a decimal/numerical representation in Text mode. For instance, the decimal value of 'A' is 65, 'B' is 66, 'a' is 97, '0' is 48, '2' is 50 etc. You can look at the ASCII table for details.

2. We used a **do** block which acts as a holder/container for **result** slot. Take a good look at it and you will notice at the bottom of that do block, there's a result slot which is used to plug the result into so that when some other procedures call this procedure above, they will **receive** the **result** plugged into **do** block.

3. First we check if the list **contains** at least **two** items, if not, then there's not really enough elements to sort as sorting of one element will return that element. Also if you remove that block and the list contains say one item, your app will encounter an **error** since in the second for each number block we have set from as "**number + 1**", but the first element in the list has a position/index of **1**, so "number + 1" will be **2** in the first iteration of the second/nested for each number block which **doesn't exist** in a list with one item.

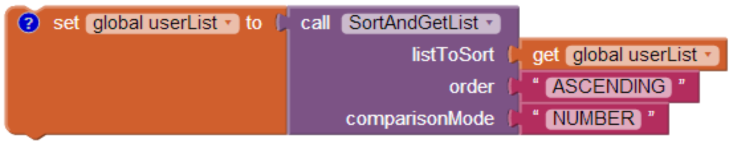
4. If you have read on bubble sort algorithm as advised above, you know that we need to **compare** each **adjacent pair**. For us to do that, we need one loop to get the items starting from the beginning of the list to the item second to the last in the list to compare. We are not going to compare the last item because there'll be no item after the last item obviously to compare with. Don't worry; by that time the last item will be in order as you will see as we move along. In the image above, you can see that in our first for each number, we have a variable named "**number**" with an initial **from** value of **1** (because first item in any list is at index 1), increments **by** 1 (because we want to go through each item) after each iteration, and our loop executes until (**to**) the value of "number" reaches length of the list minus 1.

5. In the first for each number block, we get an item at position defined by the value of "number" (e.g. if number is 1, we get the first item) and our goal is to compare the item at position pointed by "number" variable with the rest. So, we need another **for each number** block to get the rest. That is why we have the second for each number block where we have an argument named "**number2**", starts from "number + 1" because we want to compare the next item pointed by "number". We compare each until the end of list including the last item. That is why we didn't include the last item in the first for loop.

6. We just re-initialized our global variable **status** to **false** which we will need later. It's always better to **reinitialize** a global variable to its **default** value for the operation you are performing because other procedures might have used the global variable and changed the value to something else than expected.

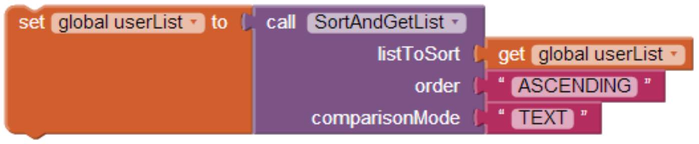
7. Storing the item at position pointed by “**number**” in the list to our global variable named **dataHolder1** for later use.
8. We are storing the item at index of “**number2**” to our global variable named **dataHolder2**.
9. We check if we should sort the items in **ascending** order. If so-
10. We need to check if the list should be sorted in **number mode**, if yes-
11. We compare the item at position “number” with the item at position “number2” using **Math** block’s “>” operator, and **swap** the items/values at index “number” and “number2” in the list if the item at position “number” is **greater**/**higher** than the item at position “number2” since we will be sorting them in **ascending** order. If item at “number” is **higher**, we set the global variable **status** to **true** so that we can **swap** later.
If **not** in number mode,
12. We do the same comparison as 11 but this time using the comparison operator from **Text** block. If item at “number” is higher, we set the global variable status to true so that we can swap later.
If we need to sort the items in **descending** order-
13. Same as **10**, check first if the comparison for Descending order should be in number mode. If yes-
14. This is opposite of **11**. We use the “<” operator from Math block since this time around we want to swap only if the item at position “number” is **less/lower** than item at position “number2” since we are sorting the items in **descending** order. If item at “number” is lower, we set the global variable status to true so that we can swap later.
If **not** in number mode,
15. We do the same comparison as **14** but this time using the **comparison** operator from **Text** block. If item at “number” is lower, we set the global variable status to true so that we can swap later.
16. We simply check if the value of our global variable **status** is **true**, if **yes**–
17. We **replace** the item at position “**number**” with the item at position “**number2**“, and-
18. We **replace** the item at position “**number2**” with the item at position “**number**“.
19. The sorting procedure is done. Hence, we **return** the **sorted** list. Note that listToSort argument in our **SortAndGetList** procedure is simply a reference to the actual list we want to sort. So if we make changes in **listToSort**, we are actually making changes in the original list that was passed when calling/using SortAndGetList. This makes it unnecessary to return the list. You can use the simple procedure block for that matter rather than using a procedure with result block. To show you how to use it, I used the latter. This tutorial on **Procedure With Result** has a better example that you should check out.
We can call our **SortAndGetList** procedure like this-

For sorting in ascending order using number mode-



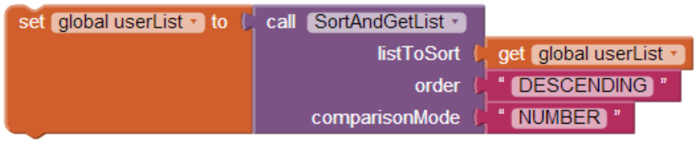
```
set global userList to call SortAndGetList
                                listToSort      get global userList
                                order           " ASCENDING "
                                comparisonMode  " NUMBER "
```

For sorting in ascending order using text mode-



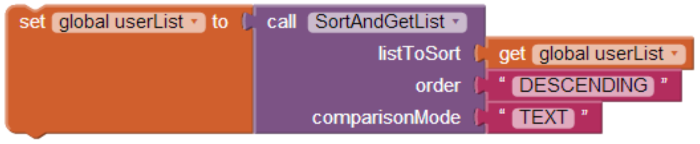
```
set global userList to call SortAndGetList
                                listToSort      get global userList
                                order           " ASCENDING "
                                comparisonMode  " TEXT "
```

For sorting in descending order uisng number mode-



```
set global userList to call SortAndGetList
                                listToSort      get global userList
                                order           " DESCENDING "
                                comparisonMode  " NUMBER "
```

For sorting in descending order using text mode-



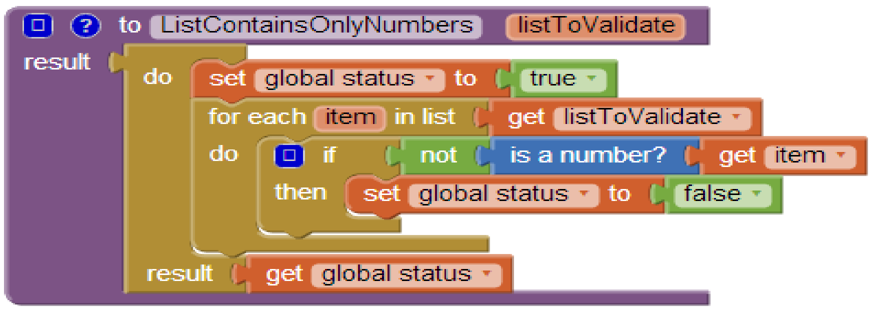
```
set global userList to call SortAndGetList
                                listToSort      get global userList
                                order           " DESCENDING "
                                comparisonMode  " TEXT "
```

In the snapshot above, **userList** is a list. As mentioned, you really don't need to assign the result of **SortAndGetList** to **userList** because when you pass the userList while calling/using SortAndGetList, the changes made in listToSort in SortAndG

etList actually changes the userList, as listToSort is just a reference to the userList.

Also, you can simplify SortAndGetList procedure depending on your needs. If you need a procedure for sorting numbers in ascending order, remove the unnecessary blocks (Numbers in black- **9, 10, 12, 13, 14, 15**). Also you can remove **order** and **comparisonMode** arguments as you don't need those arguments.

If you need to **verify** first if a list **contains** all **numbers** before you use Math block's operators for comparing (If you use Math block's operators for comparing texts, your app will throw an error), you may want to have another procedure that would check if a list contains only numbers like this-



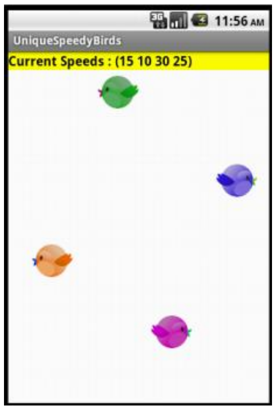
```
to ListContainsOnlyNumbers  listToValidate
result  do  set global status to true
            for each item in list  get listToValidate
            do  if    not  is a number?  get item
                then  set global status to false
        result  get global status
```

8. Shuffle a Lists & Unique Random Numbers

This tutorial will answer two questions:

1. How to shuffle a list
2. How to generate a set of random numbers that don't repeat, meaning unique numbers

We are going to make an app that will have 4 image sprites in a canvas. The sprites will be moving across the screen. In every **5 seconds**, the **speed** of the sprites will **change**. But there will never be two sprites that will have the **same** speed. We want them to always have different speeds in any given time. Our app is going to look like this-

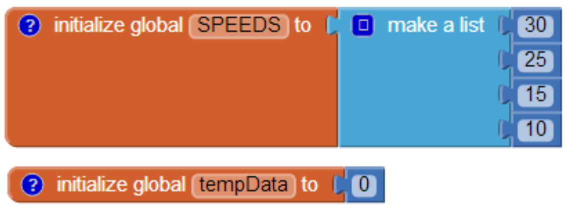


If you want to take a look at the app and play with it, download UniqueSpeedyBirds project file.

So how do you generate random numbers from a given range that are always unique?

1. *Have a list of predefined numbers you want to pick from*
2. *Shuffle the list*
3. *Get the first item in the list or get all*

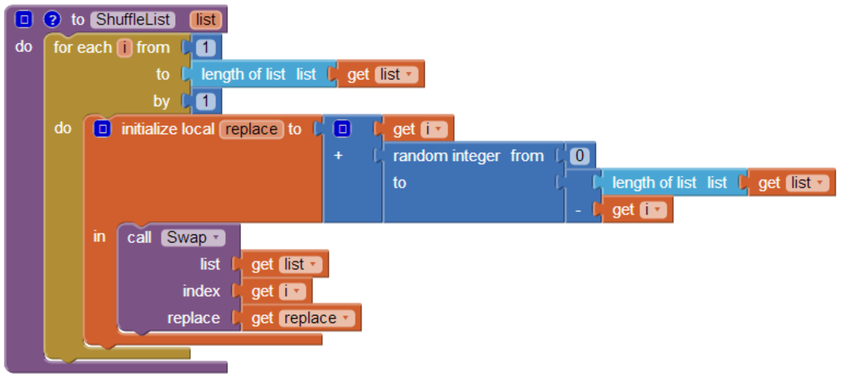
In our app, we have 4 image sprites. We assigned 4 different images to those sprites. We want them to have any of these four predefined speeds – 10, 15, 25, 30. We have a Clock component that fires after every 5 seconds. When it fires, we change the speeds of the sprites making sure that each of them is assigned a different speed.



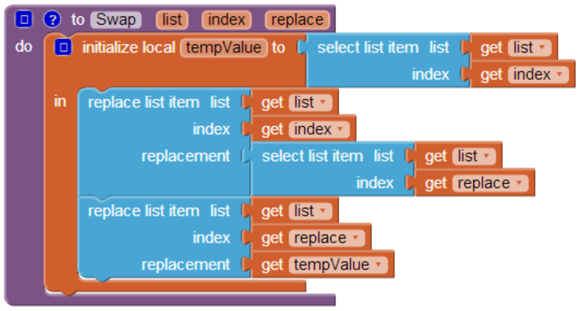
```
initialize global SPEEDS to  make a list  30
                                          25
                                          15
                                          10
initialize global tempData to  0
```

As you can see in the blocks image above, we have defined a list variable with our predefined speeds values. We also have a temporary variable that we'll need later.

Before we explain how we shuffle a list, let’s take a look at the implementation first-



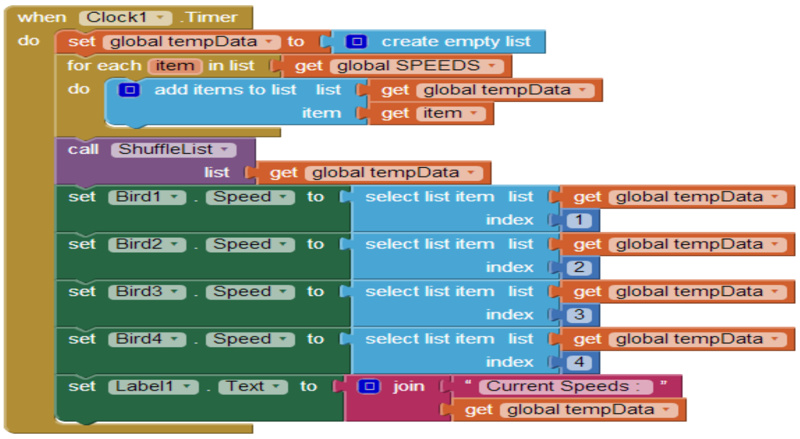
```
to ShuffleList list
do  for each i from 1
        to length of list list get list
        by 1
    do  initialize local replace to  get i
                                     + random integer from 0
                                       to length of list list get list
                                          - get i
        in  call Swap
                 list    get list
                 index   get i
                 replace get replace
```



```
to Swap list index replace
do  initialize local tempValue to select list item list get list
                                                   index get index
    in  replace list item list get list
                          index get index
                          replacement select list item list get list
                                                       index get replace
        replace list item list get list
                          index get replace
                          replacement get tempValue
```

To shuffle, we will use Fisher–Yates shuffle algorithm. You should definitely read that algorithm. We loop through each index starting with 1 until the size of the list that was passed into **ShuffleList** procedure. Current index is represented by variable **i** in for each loop. For each **i**, we select a **random** index between **0** to **length of list minus i**. We add the random index that we got, to i. The new index is represented by the local variable replace. We then swap the values in list to get a new value at index i. We do that for each index in list.

Swap procedure takes three arguments. The list where to swap, the index which we will swap at, and the replace index that we will use to swap with. We used a temporary variable named tempValue in Swap procedure to store the current value before we swap. If we don’t, we will lose current value. We use the replace list item block of list to do the swapping. This is what we do when the timer goes off-



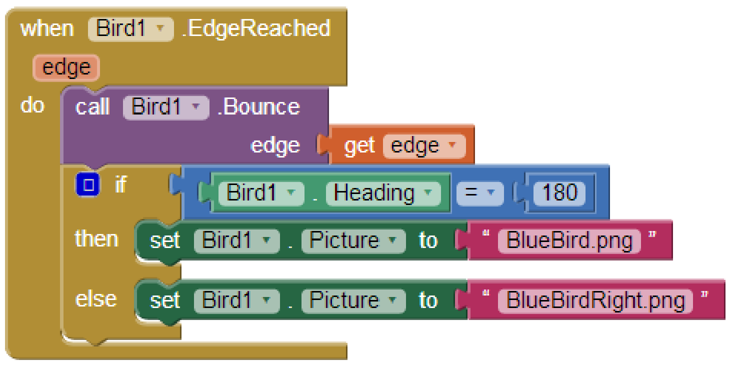
```
when Clock1 .Timer
do  set global tempData to create empty list
    for each item in list get global SPEEDS
    do  add items to list list get global tempData
                          item get item
    call ShuffleList
         list get global tempData
    set Bird1 . Speed to select list item list get global tempData
                                          index 1
    set Bird2 . Speed to select list item list get global tempData
                                          index 2
    set Bird3 . Speed to select list item list get global tempData
                                          index 3
    set Bird4 . Speed to select list item list get global tempData
                                          index 4
    set Label1 . Text to join " Current Speeds : "
                              get global tempData
```

Firstly, we don't want to change our original SPEEDS list. That is why we used a temporary variable named tempData. We added each item from SPEEDS list to that temporary variable using a for each loop. Remember we cannot just do tempData = SPEEDS. In that case, if you change anything in tempData, the changes will be also made to SPEEDS. If you are wondering why, you should better start here. After we add each item to our temporary variable, we call ShuffleList procedure and pass the temporary variable so that the shuffle only takes place to tempData, not to SPEEDS list.

After shuffling, we assign the first bird's speed to the value at index 1 of tempData. For second bird, we use the second index and so on.

Our birds move from right to left because in design view we set their Heading property to 180. Whenever they reach an **edge**, they **bounce off** meaning the **heading** changes. If they bounce from left, their heading will change to 0 heading right and if

they bounce from right, their heading will change to 180 moving leftward.



```
when Bird1 .EdgeReached
  edge
do  call Bird1 .Bounce
         edge  get edge
    if    Bird1 . Heading  =  180
    then  set Bird1 . Picture to  " BlueBird.png "
    else  set Bird1 . Picture to  " BlueBirdRight.png "
```

Also we change the pictures depending on the heading. If they are moving rightward, we want the picture where the bird is facing right.

Refrence

https://www.imagnity.com/author/admin/, by Sajal Dutta