In the previous week

Lists, Arrays, Linked Lists, . . .

List is an **Abstract Data Type**

Arrays and Linked Lists are **implementations**

A simplified memory model in the OS: `Mem[-]`

How to translate Java code into pseudo-code that uses `Mem[-]`

Let us give a name to this pseudo-code language: **OS++**

Java:

```
int[] nums = new int[4];
nums[3] = 4;
```

OS++:

```
nums = allocate_memory(4*1);
Mem[nums+3] = 4;
```

This week we will see only Java code and Pseudocode

Abstract Data Types

Abstract Data Types in Java

List is an ADT. A specific instance of a list, e.g. a List of integers, would be specified in Java by `List<int>`, a List of Strings as `List<String>` etc.

There are different implementations of the List ADT in the Java library, for example an Array based List (`ArrayList<int>`, `ArrayList<String>`, . . .) and a Linked List (`LinkedList<int>`, `LinkedList<String>`, . . .)

In Java we can declare and allocate a List, specifying which implemention we want, with the code:

```
    List<int> myArrayList = new ArrayList<int>();
    List<int> myLinkedList = new LinkedList<int>();
```

From this point on, you can use any of the predefined List methods on the `myArrayList` or `myLinkedList` variables

Abstract Data Types Revisited

Recall that An *abstract data type* is

- a type
- with associated operations
- whose representation is hidden to the user

While a *List of integers* contains the type *integer*, the type of *List of integers* is not *integer*. It is a more complex "*container type*". This is usually specified contructively: that is, we identify every possible value of type *List of integers* by specifying how to create each one. We do this by providing a list of constructor operations that create an empty *List of integers* and construct new values of type *List of integers* out of old ones

We also need to specify all other operations that any user of our ADT can depend on

List Abstract Data Type

Here is a possible list of operations for a List ADT (many variations are possible)[1]

- Constructors:
  - `EmptyList` : returns an empty List
  - `MakeList(element, list)` , adds an element at the front of a list.
- Accessors
  - `first(list)` : returns the first element of the list[2]
  - `rest(list)` : returns the list excluding the first element[2]
  - `isEmpty(list)` : reports whether the list is empty

From these, all other operations (e.g. find the nth element of the list, append one list onto another) can be implemented without requiring any other access to the List implementation details.

[1] Read chapters 1 and 2 of the module handouts

[2] Triggers error if the list is empty

List Operations: last element

in Pseudocode:

```
last(lst) {
  if ( isEmpty(lst) )
    error(”Error:␣empty␣list␣in␣last”)
  elseif ( isEmpty(rest(lst) )
    return first(lst)
  else
    return last(rest(lst))
```

List Operations: getElementByIndex

in Pseudocode:

```
getElementByIndex(index, lst) {
  if ( index < 0 or isEmpty(lst) )
    error("Error:␣index␣out␣of␣range")
  elseif ( index == 0 )
    return first(lst)
  else
    return getElementByIndex(index−1, rest(lst))
```

List Operations: append

in Pseudocode:

```
apend(lst1, lst2) {
  if ( isEmpty(lst1) )
    return lst2
  else
    return MakeList( first(lst1),
                     append(rest(lst1), lst2) )
```