Australian National University

Structured Programming

COMP1110/6710

Needs ANU Account!

pollev.com/albertofmartin963
Register for Engagement

Image Courtesy NASA/JPL-Caltech.

Admin

- U1 due this Friday, 3PM (COMP1110)
- P1 code walks this week (Thu, Fri) in Hancock Library (COMP6710). You should have received an e-mail notification with the 1-hour slot allocated to you
- P2 out now (COMP6710)
- You MUST select P1 commit ID you want us to mark/code walk and (optionally) your choice for assignment variables by Tue, 23:55PM
- New Standard Library Version: 2025S1-6 (**Download it now!**)
- (*Note:* 2025S1-6 includes the full set of list features required to complete: (1) P2; (2) this and tomorrow's workshops. While not strictly required for U1, you may also use these features if desired)
- Drop-in schedule for weeks 3-6 updated (in particular, we added many more!)
- We already have class representatives (contact details available in Wattle)

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Recap: The Design Recipe

A **systematic approach** to computer program design

Step 1. Problem Analysis and Data Design

Step 2. Function Signature and Purpose Statement

Step 3. Examples

Step 4. Design Strategy

Step 5. Implementation

Step 6. Tests

Fixed-Sized vs Arbitrarily Large Data

- Every data definition we covered so far describes data of **fixed size**
- This includes (*recap*): (1) **basic data types** (e.g., `int`, `float`, or `String`); (2) **enumerations;** (3) **records**; and (4) **general itemizations.**
- Although these data definitions can be combined to create deeply nested data structures, we always know the exact number of data pieces in (that is, **the size of**) any specific instance of such data definitions
- In many programming problems, though, **we need to process an undetermined (but finite) number of pieces of information** (e.g., keep track of an arbitrary number of objects in a world program; *today's demo*)
- Frow now own, we incorporate to our repertoire data definitions that will allow us to describe and process **arbitrarily large pieces of information**

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Flagship example: Lists

- In this workshop, we introduce **lists** as a flagship example for arbitrarily large data definitions
- Arranging information in the form of lists is an ubiquitous part of our life (e.g., TO-DO lists, files to be submitted for an assignment, invitees to an event, items to purchase in the supermarket, etc.)
- Given that information frequently comes in the form of lists, **we must learn how to represent such lists in computer programs**
- Lists are very appealing to introduce **two key concepts** in which we are going to insist over again and again and that are here to stay:

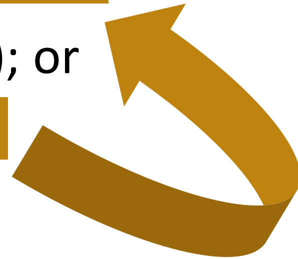
Self-referential data definitions

Recursion (Recursive thinking)

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Lists, self-references, and recursive thinking

- We will work with lists of elements of a given data type (such as, e.g., `Integer`, `String`, a user-defined `record`, etc.)
- We assume all the list elements to be of the same type. However, with Java generics (*type abstraction*), a single data definition is sufficient to have lists of different types (e.g., a list of `Strings`, a list of `Images`, etc.)
- *Conceptually*, we conceive **a list of elements** as one of either:
    1. An empty list (i.e., no elements); or
    2. An element + **a list of elements**



Self-reference

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Lists, self-references, and recursive thinking

Example: a list of `Integers` (pictorial representation)

8 -2 7 13 42 6 9 [ ]

Empty list (no elements)

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Lists in functional Java (I)

- The functional Java library offers an implementation of lists
- We will use such implementation in our functional Java programs
- You should not (and MUST NOT) write the Java code with the data definition of lists in your program
- (Note: this also applies to ANY other data definition readily available in the library; we saw people trying to do so in their code!)
- ***Any idea of how lists might be defined in the library?*** (Hint: the data definition for lists uses tools that we have already covered so far)

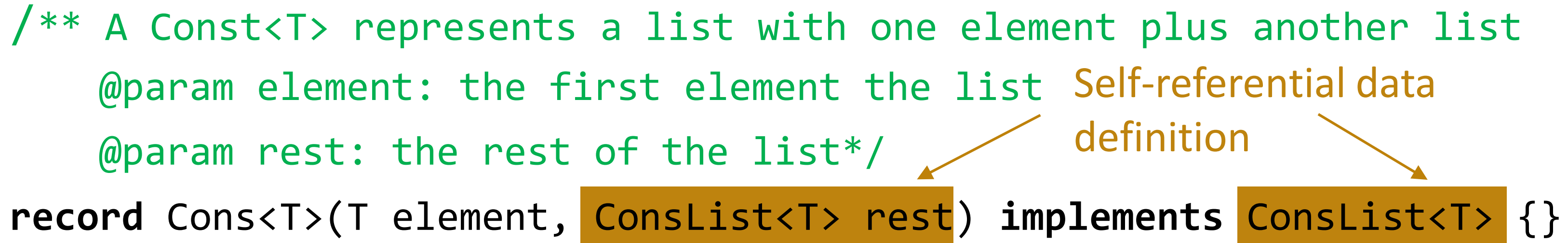
TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Lists in functional Java (II)

Do you recognize the type of data definition?

```
/** A ConsList<T> represents a list of elements of type T and is one of:
* - Nil<T> representing the empty list; or
* - Cons<T> representing an elem of type T + list of elems of type T */
sealed interface ConsList<T> permits Nil, Cons {}

/** A Nil <T> represents the empty list */
record Nil<T>() implements ConsList<T> {}
```



TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Remember COMP1100?

- Haskell has built-in support for lists
- Indeed, one can also use `Nil` and `Cons` in a similar way to functional Java
- However, in COMP1100, you might have preferably used `[]` to denote the empty list (instead of `Nil`) and `x : y` (instead of `Cons(x, y)`)

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Creating lists (example)

- Design a function `newList` that given: (1) an `Integer`, and (2) a list of `Integers`, returns a new list of `Integers` containing (1) and (2)
- Using this function, write a program that creates a list with the elements 8, -2, 7, 13, 42, 6, and 9 (see picture of this list in a previous slide)
- After creating the list, print the list on screen using a call to `println`
- *Note*: the standard library `MakeList` function offers a much more amenable alternative to create lists (we will explore it after working with `newList`)

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Programming with lists (code template)

Mostly the same code template for general itemizations with **slight new addition** that the function might be called **recursively** in the Cons case (several examples of this covered later)

```
// { ...
//   return ... switch(listOfIntegers) {
//     case Nil<Integer>() ->   ... ;
//     case Cons<Integer>(var element, var rest)->
//           ... element ... [recursiveCall](... rest ... )  ... ;
//   } ...;
// }
```

Key Motto:

The shape of the data determines the shape of the code!

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Programming with lists (easy example)

- Design a function, called `first`, that given a list of `Integers`, returns the first element of the list, if there is one. The function should deal with the empty list without generating an error on screen.
- *Example.* Given: `Cons(42, Cons(6, Nil())).` Expect: `Something<Integer>(42)`
- *Question*: Has `Nil()` any element? How can we deal with this case?
- *Hint*: you should use a data definition provided by the standard library that we covered in the previous workshop
- *Note:* the standard library `First` function is similar to this function, except for the fact that the it generates an error if called with an empty list
- *Note:* above, we used the `println`-like representation of lists. However, for convenience, you may also use `[42, 6]` instead in your documentation (indeed the standard library documentation follows this convention)

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Programming with lists (harder example)

- Design a function, called `len`, that given a list of `Integers`, returns how many elements there are in the list
- *Example*: Given: `Cons(42, Cons(6, Nil()));` Expect: 2
- *Question*: Which value should `len(Nil())`return?
- *Spoiler*: in contrast to the previous example, solution to this one requires the use of **recursion** (the main key concept we are introducing in this workshop)
- Note: the standard library contains the `Length` function, which provides the same functionality as `len`, except for the fact that, via generics, it can be applied to lists of any type (not only `Integers`*)*

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

```
len(Cons(42, Cons(6, Cons(9, Nil()))))
return switch(list) {
  case Nil()  -> 0;
  case Cons(_,var rest) -> 1 + len(rest);
};
```

3

root function call output

2 Recursive call

Each box represents a different call to the `len(…)` function with an smaller list as we move downwards

```
len(Cons(6, Cons(9, Nil())))
return switch(list) {
  case Nil()  -> 0;
  case Cons(_,var rest) -> 1 + len(rest);
};
```

1 Recursive call

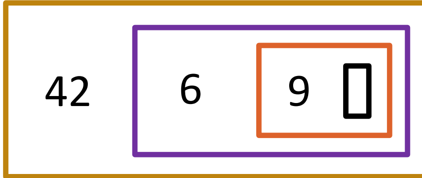
```
len(Cons(9, Nil()))
return switch(list) {
  case Nil()  -> 0;
  case Cons(_,var rest) -> 1 + len(rest);
};
```

0 Recursive call

```
len(Nil())
return switch(list) {
  case Nil()  -> 0;
  case Cons(_,var rest) -> 1 + len(rest);
};
```

Base case: recursion stops!

Illustration of all steps involved in a call to the `len(…)` function with the input list:



TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Programming with lists (exercise)

- Design a function, called `sum`, that given a list of `Integers`, returns the sum of the elements in the list
- *Example.* Given: `Cons(42, Cons(6, Nil())).` Expect: 48
- *Question*: Which value should `sum(Nil())`return?
- *Hint:* Copy and paste the definition of the `len` function and modify it to implement the `sum` function
- *Note*: `sum` can be very easily implemented using the so-called `Fold` higher-order function in the standard library (tomorrow's workshop)

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

What is recursion?

“A function that calls to itself”

- It is **a fundamental technique for solving problems in computer science**
- It finds the solution of a problem by first splitting it into smaller versions of the same problem and then combining the solutions to these smaller subproblems
- A recursive function provides a solution for **a base case**, and for a case other than the base case, the **function calls to itself** to solve a divided and smaller case
- Expectation is that any recursive call will ultimately return to the initial place it was called to produce the correct result
- In general, a recursive function can call to itself multiple times (as, e.g., in trees and graphs; studied later in the course). Besides, recursion can be used to solve problems beyond programming with lists (e.g., computing the factorial of a number, Fibonacci series, etc.; more examples later in the course)

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

How recursion is implemented in practice? ("The call stack")

- Recursion relies on a data structure called **call stack** to handle function calls
- In a **stack,** elements are inserted and deleted only at one end (top of the stack)
- An stack follows the LIFO (*Last In First Out*) policy for insertion/removal (i.e., the last element introduced is the first one to be removed)
- On each function call, an **activation record** is pushed to the stack, and on return of the function call, the top of the stack is popped from the stack
- **An activation record** mainly consists of the function call parameters, space required to store the local variables of the function being called, and the return address so that the control flow can continue right after the function call
- The compiler and the operating system handle the call stack under the hood, you do **NOT** have to handle it explicitly in your programs

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Call stack dynamics example

The call stack stores **activation records**.
Such records include return address to the caller

```
len(Cons(42,      TOP
    Cons(6,
      Cons(9,
         Nil()))))
void main(...)
```

call to len(…) → ← return

```
len(Cons(6,       TOP
    Cons(9,
      Nil())))
len(Cons(42,
    Cons(6,
      Cons(9,
         Nil()))))
void main(…)
```

call to len(…) → ← return

```
len(Cons(9,       TOP
      Nil()))
len(Cons(6,
    Cons(9,
      Nil())))
len(Cons(42,
    Cons(6,
      Cons(9,
         Nil()))))
void main(…)
```

call to len(…) → ← return

```
len(Nil())        TOP
len(Cons(9,
      Nil()))
len(Cons(6,
    Cons(9,
      Nil())))
len(Cons(42,
    Cons(6,
      Cons(9,
         Nil()))))
void main(…)
```

State of stack right after `main(…)` calls `len(…)`

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Common issue with wrong use of recursion

Do you foresee any problem with the following function?

```
/**
* … your application of the design recipe goes here …
*/
int len(ConsList<Integer> list){
   return switch(list) {
        case Nil<Integer>() -> 0;
        case Cons<Integer>(var first, var rest) -> 1 + len(list);
   };
}
```


TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Common issue with wrong use of recursion

Java runtime flags the issue with the following runtime error message
“Exception in thread "main" java.lang.StackOverflowError”

```
/**
* … your application of the design recipe goes here …
*/
int len(ConsList<Integer> list){
    return switch(list) {
        case Nil<Integer>() -> 0;
        case Cons<Integer>(var first, var rest) -> 1 + len(list);
    };
}
```

Infinite recursion!!!
(base case never reached)

“**Stack Overflow**” means that the call stack runs out of memory

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

(Some recursive thinking) Exercises

1. Design a function, called `copyStringList`, that given a list of `Strings`, returns a new list with the same elements as the input list
2. Design a function `exists`, that given (1) a list of `Strings`, and (2) a `String`, returns `true` if the element is in the list, and `false` otherwise
3. Design a function, called `last,` that given a list of `Strings`, returns the last element of the list
4. Design a function, called append, that given: (1) a list of `Strings`, and (2) an String, returns a new list resulting from appending (2) at the end of (1)
5. Design a function, called `get`, that given: (1) a list of `Strings`, and (2) the index position of an element in the list, returns the element located in such position. The position of the first element is 0, and that of the last element, `len(list)-1`. Assume that the user always provides a valid index position.

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Practice

Fork and clone the comp1110-2025s1-workshops project.
Create a folder “ws3a”, and work in “FallingBalloons.java” in there.
Commit and push when you are done.

Following the design recipe, design a world program that runs on a 800x800 pixels WHITE background square and behaves as follows. Each time the user left clicks with the mouse on the screen, the program draws a new balloon (e.g., a circle of say, radius 20, or otherwise an image of your preference) with center located at the position of the click and randomly chosen colour among the following 5 possibilities: RED, GREEN, BLUE, MAGENTA, or BLACK. The balloons must fall down at a constant speed of 5 pixels/step and disappear from the window once the reach the bottom of the background image.

Note: In a first version of the program it is ok if memory consumption grows arbitrarily as we left click with the mouse. However, in a second stage, you may also want to develop an improved version which reduces memory consumption by removing those balloons which disappear from the screen.

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C