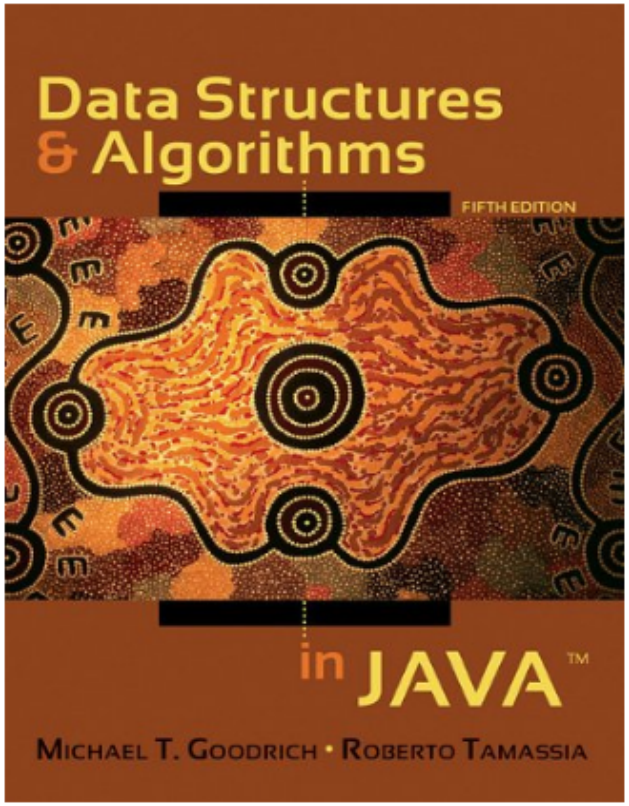
DATA STRUCTURES & ALGORITHMS

COMP H3025

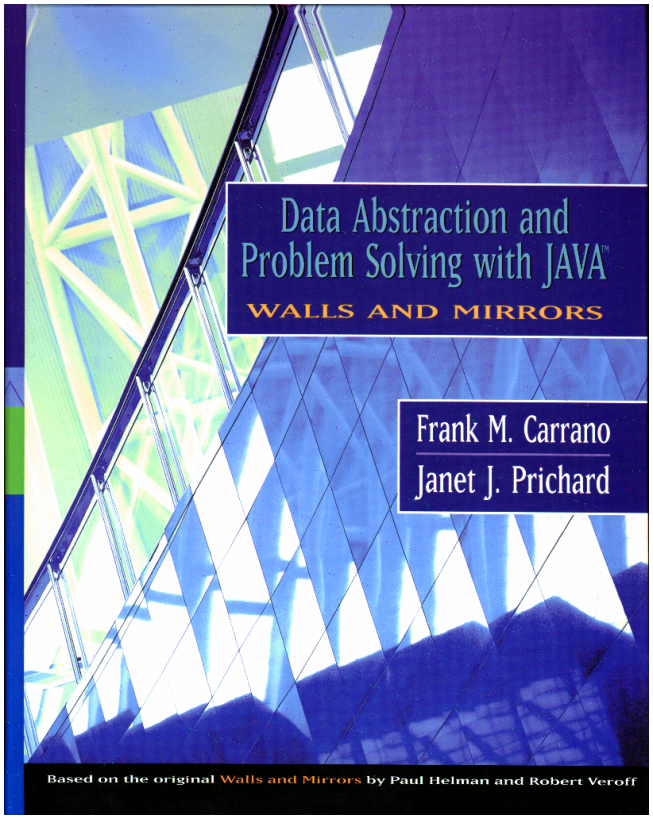
Lecture 1: Introduction

Lecturer: Stephen Sheridan

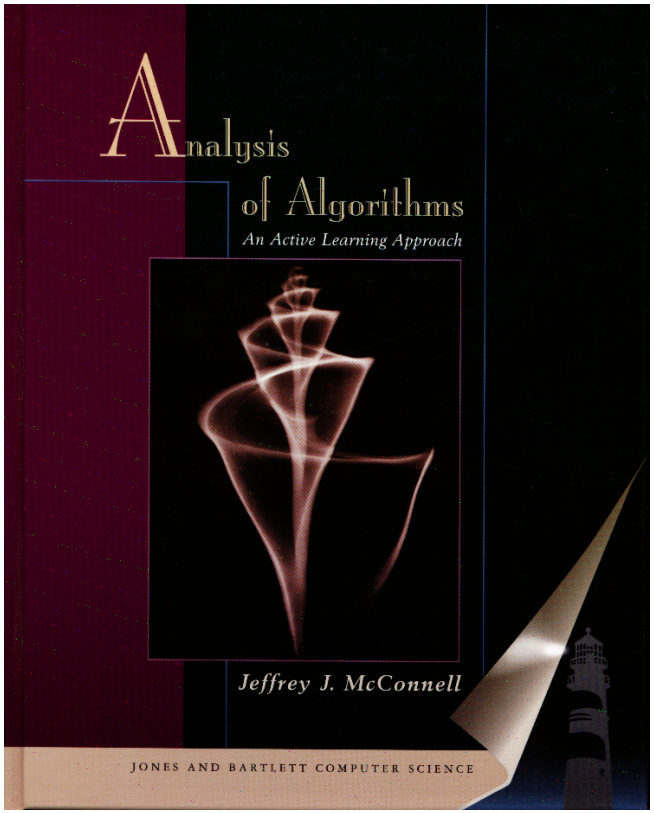
DATA STRUCTURES & ALGORITHMS



Wiley; 5th edition
(2010)



Addison Wesley
(2001)



Jones & Bartlett
(2001)

COURSE OUTLINE

- Abstract Data Types
- Lists, Stacks and Queues
- Trees
- Analysis of Algorithms
- Searching and Sorting Algorithms
- Matching Algorithms
- Graphs

ASSESSMENT

- CA 50% (Assignment 20% + Lab Test 30%)
- Written Exam 50%

PROBLEM SOLVING & SOFTWARE ENGINEERING

Q: What is problem solving in this context?

A: **[The Problem Definition]** + **[Computer Program]**

- Requires multiple phases
  - Gain understanding of problem
  - Design conceptual solution
  - Implement solution in computer code (program)

PROBLEM SOLVING & SOFTWARE ENGINEERING

Q: What is a solution in Software Engineering?

A: Typically Two Components:

(i) **Algorithms**

(ii) Ways to store the data (**Data Structures**)

- An **algorithm** is a step-by-step specification of a method to solve a problem within a finite amount of time.

- In computer science, a **data structure** is a particular way of organising data in a computer so that it can be used efficiently.

ABSTRACT DATA TYPES & DATA STRUCTURES

Q: What is an Abstract Data Type (ADT)?

A: An abstract data type is a *data structure* and a *collection of functions or procedures* which operate on the data structure.

Q: What is a Data Structure?

A: A *data structure* is a specific underlying data storage *implementation*.

ABSTRACT DATA TYPES & DATA STRUCTURES

Q: What comes first, the ADT or the Data Structure?

A: Only after fully specifying the operations of an ADT, its *Interface*, should you consider the underlying data structures required for its implementation.

- In this way an **ADT** is much like a **Java Class** in that its operations are defined by public methods and its underlying data structure is encapsulated within the class.

ABSTRACT DATA TYPES & DATA STRUCTURES

ADTs vs. Data Structures

- An abstract data type is a collection of data and a set of operations on that data.
- A data structure is a construct within a programming language that stores a collection of data.

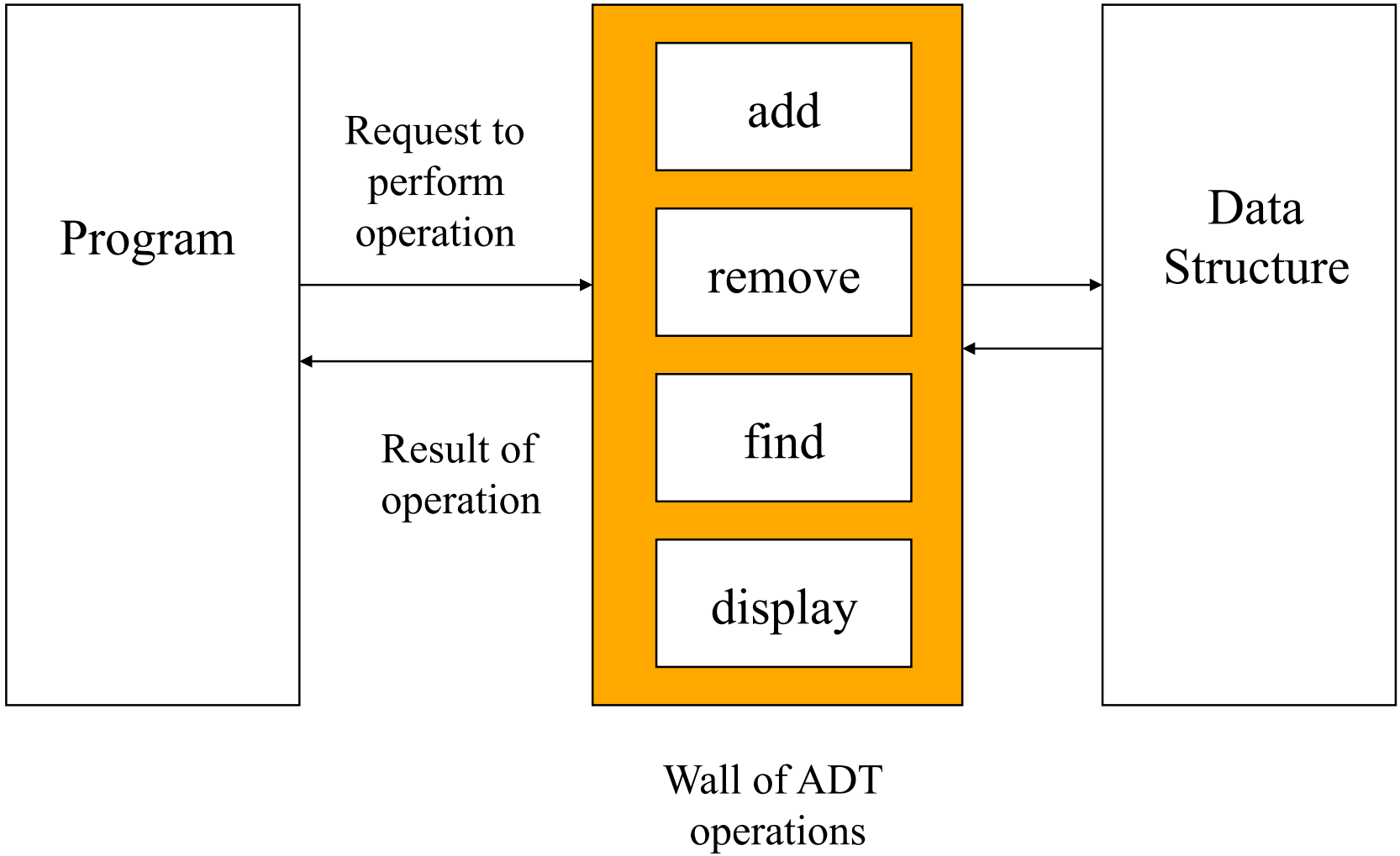
DATA ABSTRACTION (ENCAPSULATION)

- Data abstraction results in a "*wall*" of ADT operations between both **data structures** and the **program** that accesses the data with these structures.

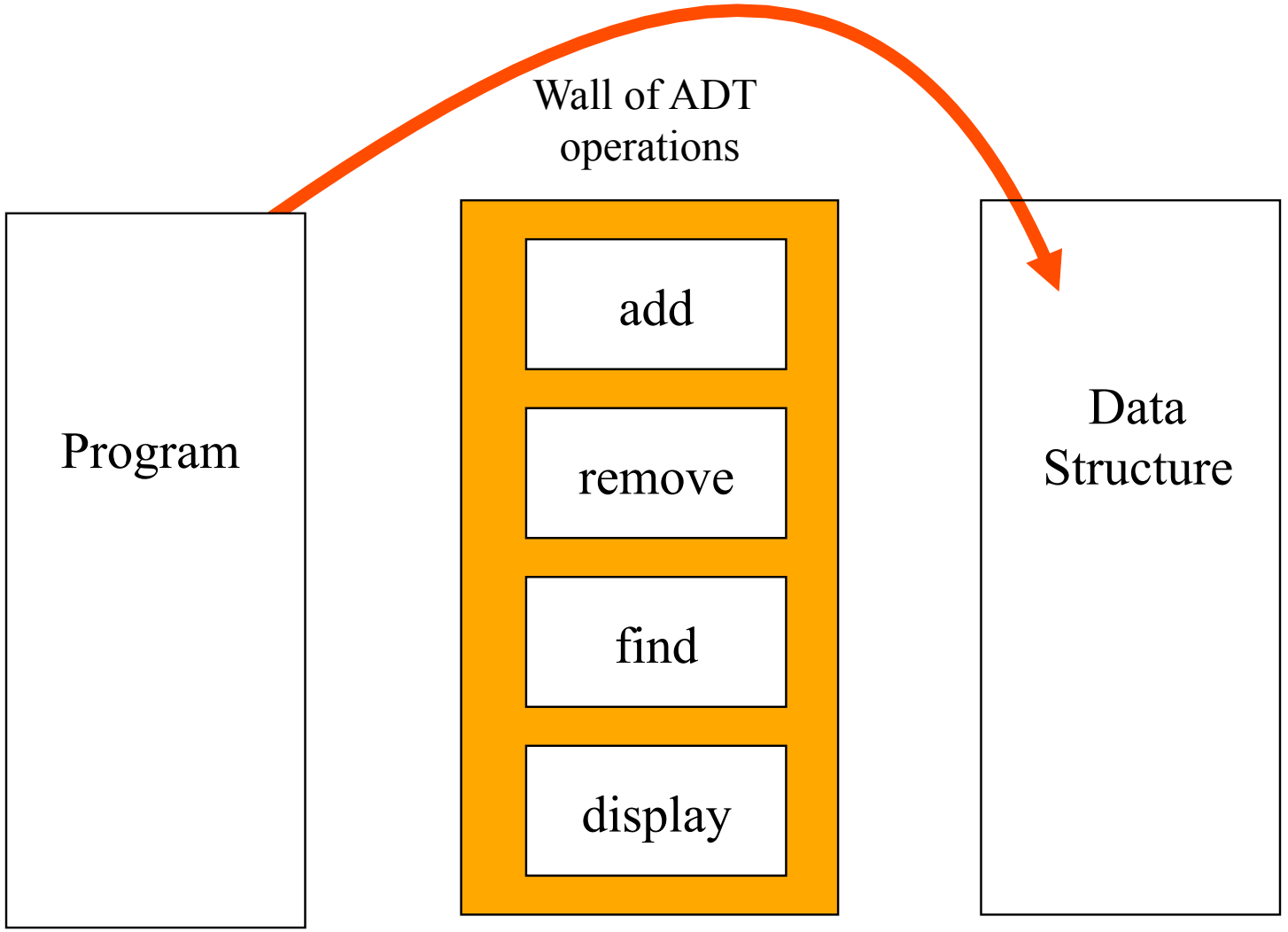
- On the program side of the wall, **interfaces** *enable communication* with the data structures.

- That is, you request the ADT operations to manipulate the data in the data structures, and they *pass the result of these manipulations back to you*.

DATA ABSTRACTION (ENCAPSULATION)



DATA ABSTRACTION (ENCAPSULATION)



Violating the wall of ADT operations

SPECIFYING AN ADT

- To elaborate on the notion of an abstract data type we will consider a typical list that you might use:
    - Shopping list
    - TODO list
    - List of addresses
- Assuming that you write in one column, most of the time we **add** new items to the **end** of the list.
- We could, however, **add** new items to the **beginning** of the list.
- We could also **sort** the list in alphabetical order, as with a list of names and addresses.

SOME COMMON LIST FEATURES

- Lists are nearly always sequential. List items appear in a sequence.
- A list has one **first** item and one **last** item.
- The first item, the **head** or front of the list, does not have a **predecessor**.
- The last item, the **tail** or end of the list, does not have a **successor**.
- Lists usually contains items of the **same type**, i.e. grocery items.

SOME COMMON LIST OPERATIONS

- **Count** list items to determine the length of the list.
- **Add** an item to the list.
- **Remove** an item from the list.
- Look at **(Retrieve)** an item from the list.
- **NOTE:** the items in the list and its operations form the ADT. We focus on specifying the operations and not on how we will implement them.

ADT LIST OPERATIONS

1. Create an empty list
2. Determine whether a list is empty
3. Determine the number of items in a list
4. Add an item at a given position in the list
5. Remove an item at a given position in the list
6. Remove all items from the list
7. Retrieve (GET) an item at a given position in the list

PSEUDO CODE FOR LIST ADT

```
createList()
//creates an empty list

isEmpty()
//determines whether a list is empty

size()
//returns the number of items that are in a list

add(index, item)
// inserts item at position index of a list, if
// 1 <= index <= size()+1
// if index <= size(), items are renumbered as
// follows: the item at index becomes the item at
// index+1, the item at index+1 becomes the item at index+2, etc.
// throws an exception when index is out of range or
// if the item cannot be placed on the list, i.e. the list is full.
```

PSEUDO CODE FOR LIST ADT

```
remove(index)
// removes the item at position index of a list,
// if 1 <= index <= size(). If index < size(), items are
// renumbered as follows: The item at index+1
// becomes the item at index, the item at index+2 becomes
// the item at index+1, etc.
// Throws an exception when index is out of range or
// if the list is empty

removeAll()
// remove all the items in the list

get(index)
// returns the item at position index of a list
// if 1<= index <= size(). The list is
// left unchanges by this operation.
// Throws an exception if index is out of range.
```

ADT LIST OPERATIONS

- So an example of how we could use the defined list operations/ interface might be as follows:

```
aList.createList();
aList.add(1, “milk”);
aList.add(2, “eggs”);
aList.add(3, “butter”);
aList.add(4, “apples”);
aList.add(5, “bread”);
```

- The notation **aList.op** indicates that an operation **op** applies to the list **aList**.

- **NOTE:** The list’s insertion operation can insert new items into any position of the list, not just at the front or end.

ADT LIST OPERATIONS

- According to the specification for the operation **add**, if an new item is inserted into position *k*, the position of each item that was at position *k* or greater is increased by 1.

- Consider
  aList.add(4, "coffee");

- The list now contains:
  milk, eggs, butter, **coffee**, apples, bread.

ADT LIST OPERATIONS

- The previous examples illustrate that an ADT can specify the effects of its operations without having to indicate how to store the data or how to implement the operations.

- The specification of the seven operations are the sole terms of the **contract** for the ADT list.

- If you request that these operations be performed, this is what they do for you!

ADT LIST OPERATIONS

- Once you have *specified the behaviour of an ADT,* you can design applications that access and manipulate the ADTs data solely in terms of its operations and *without regard for its implementation*.

- For example, suppose that you need to *display the items* in the list.

  - the wall between the implementation of the ADT and the rest of the program prevents you from seeing how the data is stored.

  - you could still write a **displayList** method that uses the operations exposed by the List ADT.

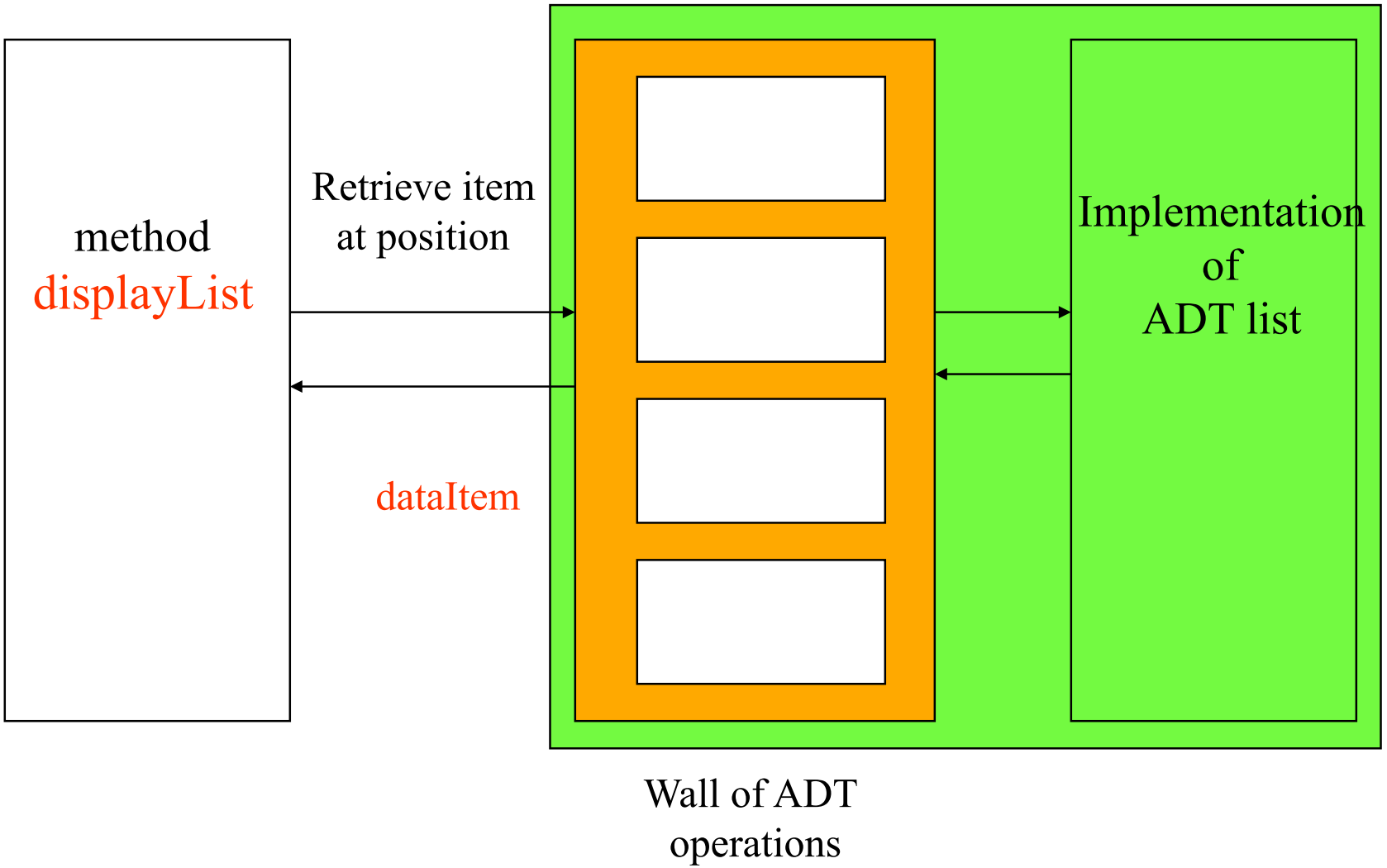
ADT LIST OPERATIONS

```
displayList(aList)
//displays the items on the list aList

for (index = 1 through aList.size() )
{  dataItem = aList.get(index)
   display dataItem
}//end for
```

- As long as the ADT list is implemented correctly, the **displayList** method will perform its task because it does not depend on how the list is implemented, only the operations its exposes.

ADT LIST OPERATIONS



The wall between displayList and the implementation that uses it

ADT LIST OPERATIONS

- As another example, suppose you need a method to **replace** an item in position *k* with an new item.

- If the *kth* item exists, **replace** deletes the item and inserts the new item at position *k*.

```
Replace(aList, k, newItem)
//replaces the kth item on the list aList with newItem

if (k >=1 and k <= aList.size() )
{
   aList.remove(k)
   aList.add(k, newItem)
} //end if
```

DESIGNING AN ADT

- The design of an abstract data type should evolve naturally during the problem solving process.
- As an example of how this process might occur, suppose that you want to create a **software based appointment book** that spans a one year period.
- Assume that appointments can only be made on the hour or half-hour between 8AM and 5PM.
- Assume that the appointments are 30 minutes in duration.
- We would also like to store some brief notation about the nature of each appointment along with a date and time.

DESIGNING AN ADT

- The data items in this ADT are the appointments, where an appointment consists of a date, time and purpose.

- So what are the ADT operations we will need?
  - **Initialise** the ADT (Create)
  - **Make an appointment** for a certain date, time and purpose. (no double bookings allowed!!)
  - **Cancel an appointment** for a certain date and time
  - **Ask whether you have an appointment** at a given date and time
  - **Determine the nature of an appointment** at a given date and time

APPOINTMENT BOOK OPERATIONS

```
CreateAppointmentBook()
// create an empty appointment book

isAppointment(date, time)
// returns true is an appointment exists for the date
// and time specified; otherwise returns false

makeAppointment(date, time, purpose)
// inserts the appointment for the date, time, and purpose
// specified as long as it does not conflict with an existing
// appointment. Returns true if successful, false otherwise

cancelAppointment(date, time)
// returns the purpose of the appointment at
// the given date/time, if one exists. Otherwise, returns null.

CheckAppointment(date, time)
// returns the purpose of the appointment at
// the given date/time, if one exists. Otherwise, returns null.
```

ARRAY-BASED LIST ADT IMPLEMENTATION

- To start our practical work off we will look at an array-based implementation of the List ADT in Java.

- The operations we will implement will be as follows:
  - createList();
  - isEmpty();
  - size();
  - add(newPosition, newItem);
  - remove(index);
  - removeAll();
  - get(index, dataItem);

ARRAY-BASED LIST ADT IMPLEMENTATION

- At first glance we might think that an array is a natural "fit" for a list implementation. *This is not quite true!*

- The ADT list has operations such as *removeAll()* that an array does not have.
  - Later we will look at a different type of implementation for the list ADT that does not use an array.

- Using an array implementation we can store a lists *kth* item in **items[k-1]**
  - We may occupy all of the array, or we may not!
  - We need to keep track of the number of items assigned to the array. We will refer to this as its **logical size**.

ARRAY-BASED LIST ADT IMPLEMENTATION

- We will implement each ADT operation as a **method of a class.**
- Each operation will require access to both the array items and the lists length **numItems**, so these need to be **data fields of the class**.
- These data fields should be **hidden** so make them **private**.
- Access to these data fields is allowed via the operations (= public methods).
- We will define a **translate(position)** operation which returns the index of the array element that contains the list item at position index. That is, translate(position) returns the index value -1.
- We should **throw exceptions** as appropriate, i.e. if an index is out of bounds.

TODO - WEEK 1

- Create an ADT List implemented using an array
- Write out the specifications for the ADT List
- Implement the ADT List as a Java class over an array such that the walls are reinforced using encapsulation/data abstraction.
- The class must implement the ADT operations as given by the Java interface on MOODLE.
- Write a small driver program that demonstrates the use of the ADT class and therefore the ADT List.
- This program should print results to screen in some simple manner.