Data Structures
&
Algorithms using Java

Dr. Abdulghani M. Al-Qasimi

Fall 2020

Introduction

Instructor:

Name: Dr. Abdulghani M. Al-Qasimi.
Office: Engineering Building A40, Room 34E15.
Hours: Only by Appointment.
E-mail: aqasimi@kau.edu.sa

Course Description:

Lecture Time: Monday and Wednesday BA: 08:00 – 09:15 am.
Lecture Place: TBA.
Lab Hours: None.
Web site: At the University's Blackboard system.

Introduction:
Pre-Requisites

- EE 305, EE 364, and good skills in Java Programming.
- This course assumes you have basic Java knowledge:
  - interfaces, classes and objects,
  - fields, static and instance variables,
  - methods, constructors and control structures,
  - standard input/output, file input/output and exceptions,
  - arrays, strings, casting, generics and exposure to inheritance.
- You must know structured program design concepts: top-down analysis, modular programming, error checking, program testing and debugging, writing clear programs, and proper program documentation.

Introduction:
The Textbook

Required Text Book:

Data Structures & Algorithms in JAVA, 6th edition – International Student Version, by M. Goodrich, R. Tamassia & M. Goldwasser, John Wiley & Sons, Inc., 2014.

Additional References:

- *Data Structures and Abstractions with Java*, 2nd edition, by Frank M. Carrano, Prentice Hall, 2007.
- *Data Structures, and Problem Solving with Java*, 3rd edition, by Mark Allen Weiss, Addison Wesley, 2006.
- Any other books on the subjects of Java programming and/or Data Structures using Java.

Introduction:
Grading System

- Theoretical Home works ………………….. 10%
- Programming Assignments …............….. 15%
- Quizzes ……………………………………..…. 15%
- Midterm Exam …..............…….….……... 20%
- Final Exam ....….……….……. 40%

Empty Slide

Introduction:
Role in the Curriculum

- This course represents a transition between "learning to program" courses (like EE202, EE364) and "content" (i.e. theory and analysis) courses.

- To do well, you must be able to handle **both**
  - Programming:
    We focus on applications with dynamic memory allocation and simple file processing
  - Content:
    We offer both algorithm theory and algorithm analysis

Introduction:
The Need for Data Structures

- More powerful computers
  $\Rightarrow$ more complex applications.

- More complex applications demand more calculations.

- Data structures organize data
  $\Rightarrow$ more efficient programs.

Introduction: Efficiency

- Choice of data structure or algorithm can make the difference between a program running in a few seconds or many days.
- A solution is said to be efficient if it solves the problem within its resource constraints.
  - Space
  - Time
- The cost of a solution is the amount of resources that the solution consumes.

Introduction: Costs and Benefits

- Each data structure has its costs and benefits.
- Rarely is one data structure better than another in all situations.
- Any data structure requires:
  - Space for each data item it stores,
  - Time to perform each basic operation,
  - Programming effort.

Introduction:
Costs and Benefits

- Each problem has constraints on available resources (space and time).
- Only after a careful analysis of the problem characteristics can we know the best data structure for the task.
- Student Record Example:
  - Creating a record: takes a few minutes
  - Transactions on a record: takes a few seconds
  - Closing a record: takes overnight

Introduction:
Selecting a Data Structure

Select a data structure as follows:

1. Analyze the problem to determine the basic operations that must be supported.
2. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

Introduction:
Some Questions to Ask

- Are all data inserted into the data structure at the beginning,
  - or are insertions interspersed with other operations?
- Can data be deleted?
- Are all data processed in some well-defined order,
  - or is random access allowed?

Empty Slide

Data Structures
&
Abstract Data Types

Definitions

Definitions:
Data Types & Data Structures

- Data, is the representation of information in a manner suitable for communication or analysis by humans or machines
- Programs and applications read, store and operate on data. They also output data.
- Data consists of Numbers, Characters, etc.
- Data is classified into:
  - Data Types
  - Data Structures

Definitions:
Data Types

- A Data Type, (e.g.: char, float, int, etc.), is defined by its logical properties:

  1. A set of data elements, which forms the domain (D) of allowed values, and

  2. A set of legal operations on these values.

Definitions:
Data Types

- Examples

Data Type	Domain (D)	Operations
boolean	{0,1}	and, or, not, etc.
char	ASCII Table Characters	==, !=, <, >, etc.
int	{-max … , -1, 0, 1, ... max}	+, -, *, /, %, etc.

Definitions:
Data Types

- The set of elements of a Data Type may be finite or infinite

Examples:

Integer:	D={…, -2, -1, 0, +1,+2, … }	infinite
Alpha:	D={A, B, C, …, Z}	finite

Definitions:
Logical vs. Physical Forms of Data

Data items have both a logical and a physical form.

- Logical form:
  Definition of the data item's logical properties (ADT)
  - ie: **domain** of data elements and the legal **operations**.
- Physical form:
  Implementation of the data item within a data structure.
  - ex: 16-bit integer **representation**, overflow indicators, …

Definitions:
The Abstract Data Type (ADT)

- Data abstraction is the separation of a data type's logical properties from its implementation details
- Abstract Data Type (called, ADT) is a data type whose properties (domain and operations) are specified independently of any particular implementation
- Each ADT operation is defined by its inputs and outputs.
- A Java interface provides the means to define ADTs.
- Encapsulation **hides** the implementation details.

Definitions:
Metaphors

- Metaphors are hierarchies of labels that manage complexity through abstraction.
- A metaphor is a label given to an assembly of objects or concepts, which can then be used in another assembly to define yet another higher-level metaphor, and so on.
  - Example:
    transistors ⇒ gates ⇒ CPU.
- In a program, implement an ADT, then think **only** about the **ADT**, **not its implementation details**.

Definitions:
Data Levels of an ADT

1. Abstract (or Logical) level:
   This is the abstract view of the data values (the domain) and the set of operations to manipulate them.

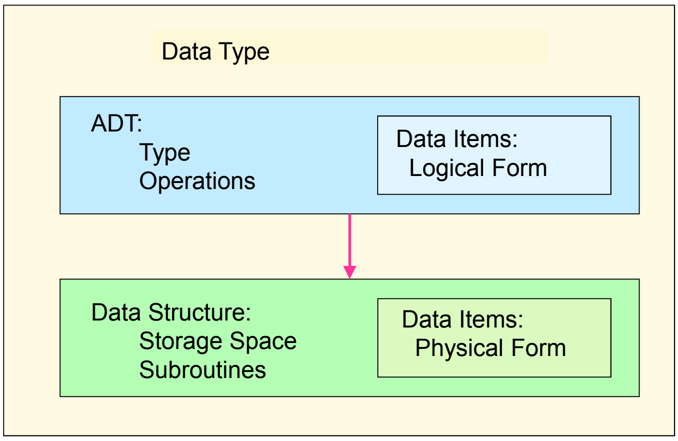
2. Implementation level:
   A specific representation of the structure to hold the data items, and the coding of the operations in a programming language.

3. User (or Application) level:
   At this level, the application programmer uses the ADT to solve a particular problem.

Definitions:
Data Structure

- A data structure is the physical implementation of an ADT.
- Data structure: usually refers to an organization for data in main memory.
- File structure: refers to an organization for data on secondary storage, such as a hard disk.

Definitions:
Data Types & Data Structures



Definitions:
ADT Implementation

- ADT implementation refers to the mapping of the abstract definition (A) into a set of other data structures (E) which actually exist in the computer.
- It involves:
  1. Specifying the representation of elements from domain D of A, by elements from the domain of E.
  2. Writing functions of A using functions of E.

Example 1:
The Integer Abstract Data Type

Type

`int`

Value range: `-2147483648 . . +2147483647`

Operations

+ prefix	identity
- prefix	negation
+ infix	addition
- infix	subtraction
* infix	multiplication
/ infix	division
% infix	remainder (modulo)

Example 1:
Integer ADT Implementation

- Problem:
  Implement the integer ADT, A, defined as:
  - Domain = {…, -2, -1, 0, +1, +2, …}
  - Operations: add, subtract, multiply, divide.

- Solution:
  Find an existing data type E,
  Using basic hardware, there is only Boolean:
  - Domain = {0,1}
  - Operations: NOT, AND, OR

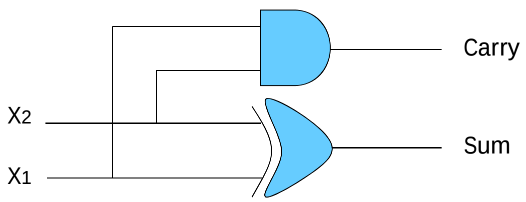
Example 1: Integer Implementation
A. Representation

1. Sign-Magnitude:
   +30 : 0001 1110, -30 : 1001 1110
2. Two's Complement:
   +30 : 0001 1110, -30 : 1110 0010
3. Binary Coded Decimal:
   +30 : 0011 0000, -30 : Not available

Example 1: Integer Implementation
B. Mapping Operations

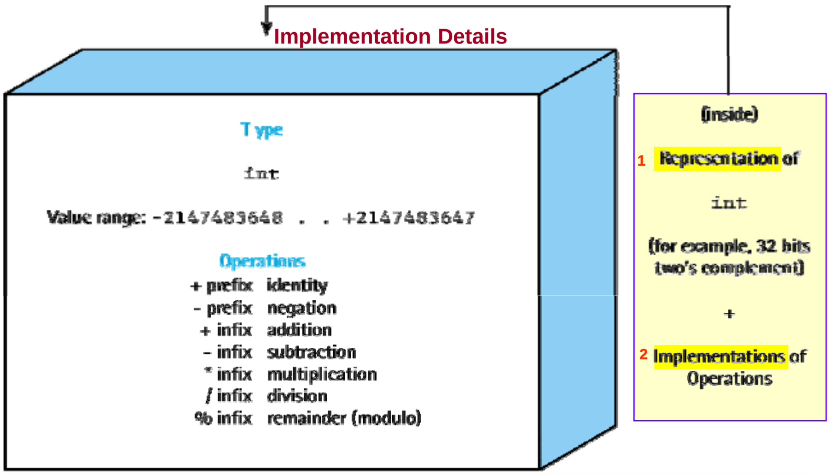
Take the Add operation for example:

- The adder circuit is shown below:
- A Full adder takes 3 inputs and produces 2 outputs



X1	X2	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Example 1:
The Integer Data Type



Example 2:
The Set Abstract Data Type

- Problem:
  Implement the Set ADT, A, defined as:
  - Domain: Elements of some base type
  - Example: [blue, red, green]
  - Operations: union (+), difference (-), intersection (*),
    subset (<=), superset (>=), and membership.

- Solution:
  Find an existing data type E,
  Using basic hardware, there is only Boolean:
  - Domain = {0,1}
  - Operations: NOT, AND, OR

Example 2: Set Implementation
A. Representation

- Any set, S, of base type T can be represented by its characteristic function, b, which is defined for all values of the domain of T.
- Example:
  If the base type is the integer in the range 1..10, then a set S=[1..3,5,9,10] can be represented by a string of 10 binary digits as follows:

1	2	3	4	5	6	7	8	9	10
1	1	1	0	1	0	0	0	1	1

For short it is written as b = {1110100011}.

Example 2: Set Implementation
B. Operation Mapping

Given the two sets S_1 and S_2, and their characteristic functions b_1 and b_2, respectively, then:

Set Operation	Its Mapping
$S_1 + S_2$	b_1 OR b_2
$S_1 * S_2$	b_1 AND b_2
$S_1 - S_2$	b_1 AND (NOT b_2)
$S_1 <= S_2$	(b_1 AND b_2) == b_1
$S_1 >= S_2$	(b_1 AND b_2) == b_2
Element x is a member of S_1	Same as [x] <= S_1

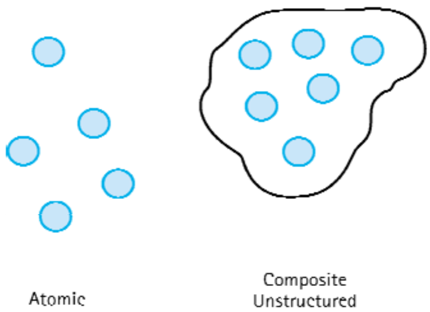
Definitions:
Primitive Data Types

- A Data Type can be:
  1. Primitive, (Simple or Atomic), where its elements are single, non-decomposable data items, like:
     - Integer numbers,
     - Real numbers,
     - Characters
  2. Composite, where its elements are composed of multiple data items.

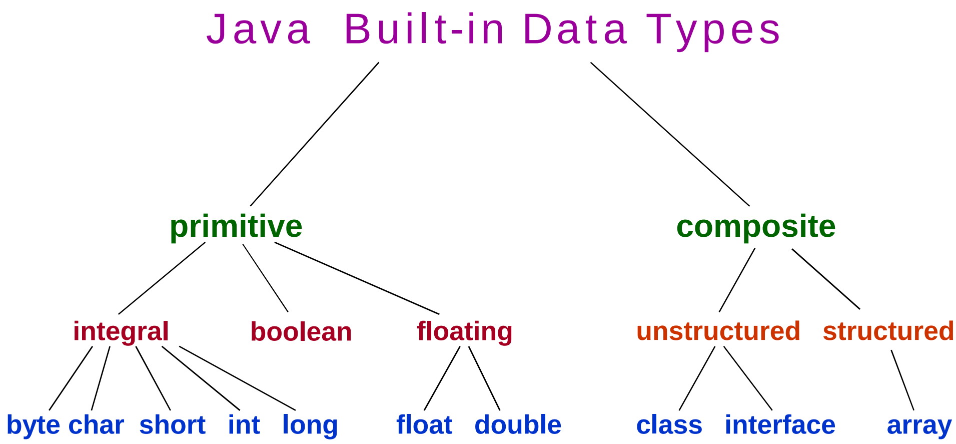
Definitions:
Composite Data Types

- With Composite data types, the main operation of interest is accessing the elements that make up the collection
- A Composite data type can be:
  1. Unstructured: A collection of components that are not organized with respect to one another
  2. Structured: An organized collection of component data in which the organization determines the means of accessing individual components or subsets of the collection

Examples:
Primitive & Composite Data Types



Examples:
Primitive & Composite Data Types



Notes:

- All Java built-in types are ADTs.
- Java programmers can use the Java class mechanism to build their own ADTs.

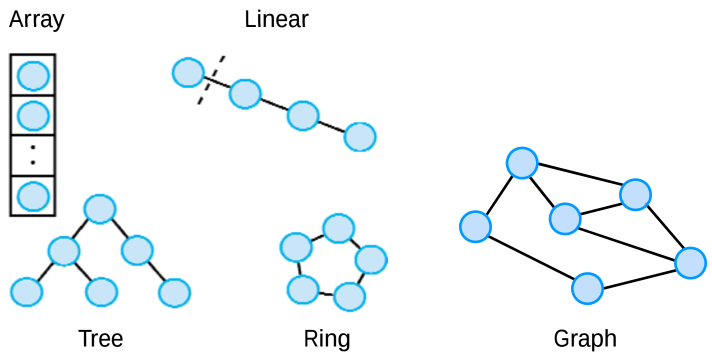
Empty Slide

Definitions:
Data Structure

- A Data Structure is also a collection of data elements (simple or composite) that have a set of relations between them, reflected by their logical organization (structure).

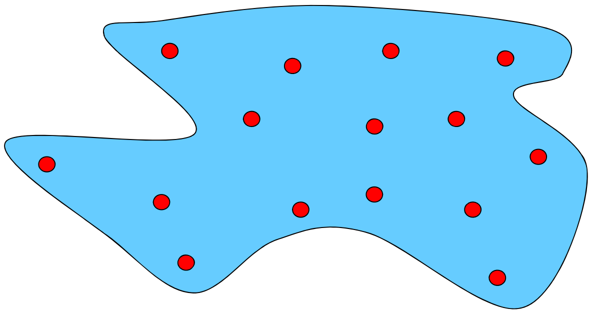
- Example data structures are:
  Arrays, Lists, Stacks, Queues, Trees, Hash tables and Graphs.

Examples:
Data Structures



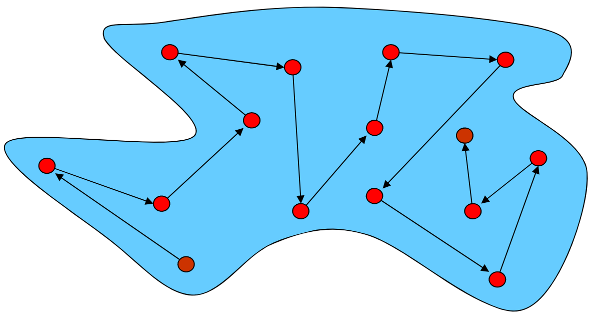
Structural Relationships:
1. Set Relationship

- There is no structure among elements of a set except their membership in the set.



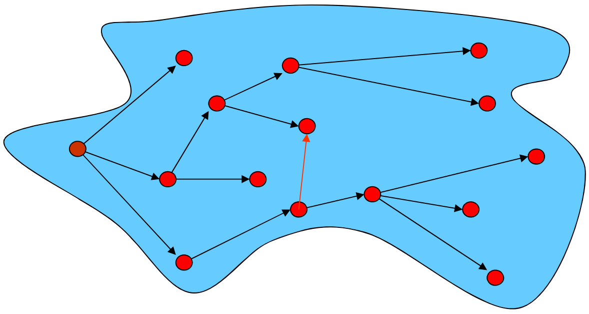
Structural Relationships:
2. Linear Relationship

- Each element is related to only one other element, defining a one-to-one relationship



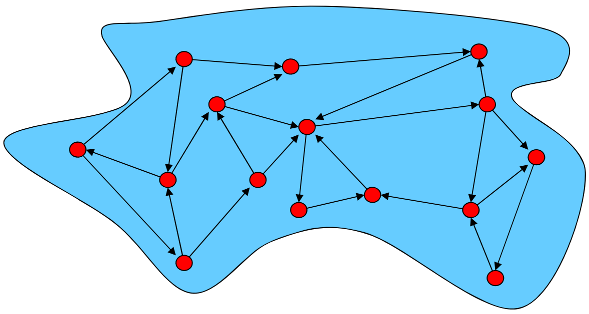
Structural Relationships:
3. Tree Relationship

- Each element is related to one or more elements in a one-to-many relationship



Structural Relationships:
4. Graph Relationship

- Each element is related to one or more elements in a many-to-many relationship



Definitions:
Order

If a, b, and c are arbitrary elements of any set of elements, S, and the relation <= is defined on pairs of elements of S, and if:

1. a <= a is true, and
2. If a <= b and b <= c then a <= c is true, and
3. If a <= b and b <= a then a = b is true, and
4. Either a <= b or b <= a is true,

Then S is said to be totally ordered by <= operator.

Order Examples

1. The set of integers, is an ordered set by the <= operator, since it satisfies all the conditions stated above.

2. The set of alphabetical characters is also ordered by the <= operator.

   The ordering in this case is called Lexicographic ordering.

Relational Operators

If a set of elements is ordered then we can specify a collection of relational operators on its elements. They are:

< , <=, >, >=, =, and !=

A relational expression produces a boolean result. i.e. True or False

Definitions: Linearity

A finite set of elements is linear if the set is empty, or if it contains a single element, or if the following four conditions are met:

1. There is a unique element called the first.
2. There is a unique element called the last.
3. Every element, except the last, has a unique successor.
4. Every element, except the first, has a unique predecessor.

Linearity Examples

1. The set of integers is linear according to the stated definition.
2. The set of alphabetical characters is also linear.
3. The set of real numbers is ordered but not linear.

   There is an infinite number of successors for each real number.

Linearity Operators

If a set of elements is linear then we can specify the following operators on its elements. They are:

Find First,
Find Last,
Find AtPosition, Find Position,
Find Next, and
Find Previous

Example 3:
Array Abstract Definition

- An array is a finite ordered set of homogeneous elements.
  - All its elements have the same size
  - Array elements occupy contiguous locations in memory
  - The ordering is defined by an index
  - All array operations involve accessing an element of the array.

Example 3: Array Implementation
One-Dimensional array

An array is stored internally in successive memory locations, starting from some address called the base-address.

Let:

base	address of first element of the array
esize	size of each element in the array
ℓ	lower bound of the array index (in Java $\ell = 0$)
i	index of current element

Then:

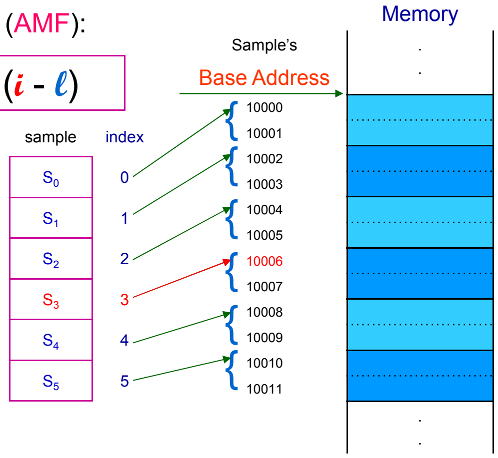
To access an array element, we need to find the memory address corresponding to the index of that element in the array.

i.e. A Mapping from index i to address a.

Example 3: Array Implementation
One-Dimensional array

The Address Mapping Function (AMF):

$$\boxed{a = base + esize * (i - \ell)}$$



Example:
For the array defined as:
int [ ] sample = new int [6];
Find the address of sample[3]:

base = 10000, esize = 2, $\ell = 0$,
$i = 3$.

$a = 10000 + 2 * (3 - 0) = 10006$

Example 3: Array Implementation
Two-Dimensional array

A two-dimensional array is stored internally in one of two ways:

- Row-Major order (row by row) or
- Column-Major order (column by column).

In successive memory locations, starting from the base-address.

Let base & esize be as defined before,

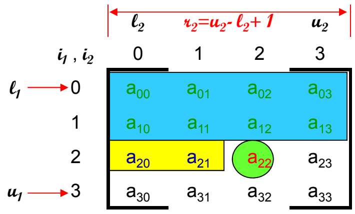
ℓ_1 and ℓ_2	lower bounds of the array indexes (in Java, $\ell_1 = \ell_2 = 0$)
u_1 and u_2	upper bounds of the array indexes (in Java, $u_1 = (size_1 - 1)$ and $u_2 = (size_2 - 1)$)
i_1 and i_2	indexes of the current element
n_2	The number of elements in each row = $(u_2 - \ell_2 + 1)$

Example 3: Array Implementation Two-Dimensional array

Then:

To access an array element, we need to find the memory address corresponding to the two indexes of that element in the array. This is a 2-D to 1-D mapping.

The Address Mapping Function (AMF):



$$a = base + esize * [\,(i_1 - \ell_1) * r_2 + (i_2 - \ell_2)\,]$$

Example 3: Array Implementation Two-Dimensional array

Example:

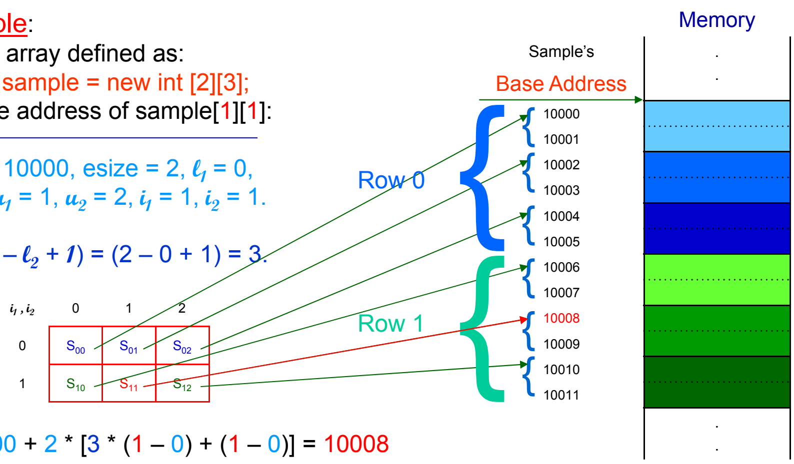
For the array defined as:

int [ ][ ] sample = new int [2][3];

Find the address of sample[1][1]:

base = 10000, esize = 2, $\ell_1 = 0$, $\ell_2 = 0$, $u_1 = 1$, $u_2 = 2$, $i_1 = 1$, $i_2 = 1$.

$r_2 = (u_2 - \ell_2 + 1) = (2 - 0 + 1) = 3.$



$a = 10000 + 2 * [3 * (1 - 0) + (1 - 0)] = 10008$

Example 3: Array Implementation Multi-Dimensional array

A Multi-dimensional array is stored internally in one of two ways:

- Row-Major order, the rightmost index changes most frequently
- Column-Major order, the leftmost index changes most frequently

using successive memory locations, starting from the base-address.

Let base & esize be as defined before,

$\ell_1, \ell_2, .., \ell_n$	lower bounds of the array indexes (in Java, $\ell_1 = \ell_2 = .. = \ell_n = 0$)
$u_1, u_2, .., u_n$	upper bounds of the array indexes (in Java, $u_1 = (size_1 - 1)$, $u_2 = (size_2 - 1)$ …)
$i_1, i_2, .., i_n$	indexes of the current element
$r_2, r_3, .., r_n$	The no. elements in each dimension $r_k = (u_k - \ell_k + 1)$

Example 3: Array Implementation Multi-Dimensional array

Then:

To access an array element, we need to find the memory address corresponding to the set of n indexes of that element in the array. This is an n-D to 1-D mapping.

The Address Mapping Function (AMF):

$$\begin{aligned} a = base + esize * [&(i_1 - \ell_1) * r_2 * r_3 * \ldots * r_{n-1} * r_n + \\ &(i_2 - \ell_2) * r_3 * r_4 * \ldots * r_{n-1} * r_n + \\ &\ldots\ldots.. + (i_{n-2} - \ell_{n-2}) * r_{n-1} * r_n + \\ &(i_{n-1} - \ell_{n-1}) * r_n + (i_n - \ell_n)] \end{aligned}$$

Example 3: Array Implementation
Multi-Dimensional array

An equivalent formula that is more efficient to evaluate is:

$$a = \text{base} + \text{esize} * [\, i_n - \ell_n + r_n * (i_{n-1} - \ell_{n-1} + r_{n-1} * (i_{n-2} - \ell_{n-2} + r_{n-2} * (\ldots\ldots + r_3 * (i_2 - \ell_2 + r_2 * (i_1 - \ell_1))\ldots\ldots)))]$$

Definitions:
Data Levels of an ADT

1. Abstract (or Logical) level:
   This is the abstract view of the data values (the domain) and the set of operations to manipulate them.

2. Implementation level:
   A specific representation of the structure to hold the data items, and the coding of the operations in a programming language.

3. User (or Application) level:
   At this level, the application programmer uses the ADT to solve a particular problem.

Definitions:
Data Encapsulation

- Data encapsulation is the separation of the representation of data from the applications that use the data at a logical level
- The Java class mechanism provides the means to encapsulate the data of an ADT.

Definitions:
Basic Operations on Encapsulated Data

- Constructors: are operations that create new instances (objects) of the data type.
- Transformers (sometimes called *mutators*): are operations that change the state of one or more of the data values.
- Observers: are operations that allow us to observe the state of one or more of the data values without changing them.
- Iterators: are operations that allow us to process all the components in a data structure sequentially.

Example 4: Specification of Collection ADT

- A Collection is a data type that is capable of holding a group of integer items.
- There can be many instances of the same item in the collection.
- Thus we can think of it as a container (a bag) with the following operations:

Operation	Action
initialize():	Creates an empty collection of fixed capacity = 10.
add(item):	Adds one item to the collection.
countOccur(item):	Checks how many occurrences of a certain item are in the collection.
remove(item):	Removes one item from the collection.
size():	checks how many items are in the collection.

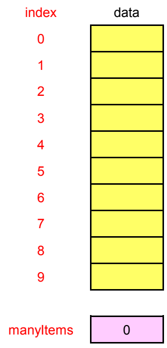
Example 4: Java Interface for The Collection ADT – Bag

```
/**
 * @(#)Bag.java
 *
 * A simple Bag interface
 * @author Dr. Abdulghani M. Al-Qasimi
 * @version 1.00 2011/7/4
 */

public interface Bag<E>  {

    public boolean add(E item);
    public int countOccur(E item);
    public boolean remove(E target);
    public int size( );

}
```

Example 4: Implementation Using Java Array

Representation:

- Use a partially filled array of fixed capacity
- Use one integer variable called manyItems, which stores the number of items currently in the bag
- An empty bag is initialized by a constructor, dynamically creating the array, and setting manyItems = 0.



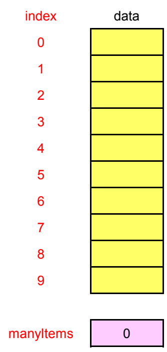
Code:

```
public class IntArrayBag implements Bag<Integer>
{
   private int[ ] data;
   private int manyItems;
}
```

Example 4: Implementing Constructor

Code:

```
/**
* Initialize an empty bag with an initial capacity of 10.
* @param - none
* @postcondition
* This bag is empty and has an initial capacity of 10.
**/

public IntArrayBag( )
{
   final int INITIAL_CAPACITY = 10;
   manyItems = 0;
   data = new int [INITIAL_CAPACITY];
}
```



Example 4: Implementing Transformer add(item)

Code:

```
/**
* If not full, then add a new item to the bag and return true.
* If the bag is full, then do not add the item and return false.
* @param item
*   the new item that is being inserted
* @postcondition
*   A new copy of the item has been added to this bag.
**/
public boolean add(Integer item)
{
   if (manyItems == data.length) return false;
   data[manyItems] = item;
   manyItems++;
   return true;
}
```

index	data
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

manyItems	0

Example 4: Applying add(item)

Example:

- add(4);

index	data
0	4
1	
2	
3	
4	
5	
6	
7	
8	
9	

manyItems	1

Example 4:
Applying add(item)

Example:

- add(4);
- add(8);

index	data
0	4
1	8
2	
3	
4	
5	
6	
7	
8	
9	

manyItems	2

Example 4:
Applying add(item)

Example:

- add(4);
- add(8);
- add(4);

index	data
0	4
1	8
2	4
3	
4	
5	
6	
7	
8	
9	

manyItems	3

Example 4:
Applying add(item)

Example:
- add(4);
- add(8);
- add(4);
- add(1);

index	data
0	4
1	8
2	4
3	1
4	
5	
6	
7	
8	
9	

manyItems	4

Example 4: Implementing Accessor countOccur(item)

Code:
```
/**
* Counts the number of occurrences of an item in this bag.
* @param item
*   the item that needs to be counted
* @return
*   the number of times that item occurs in this bag
**/

public int countOccur(Integer item)
{
   int index, answer = 0;
   for (index = 0; index < manyItems; index++)
      if (item == data[index])  answer++;
   return answer;
}
```

index	data
0	4
1	8
2	4
3	1
4	
5	
6	
7	
8	
9	

manyItems	4

Example:
- countOccur(4) ➔ 2

Example 4: Implementing Transformer remove(target)

Code:

```
/**
* Remove one copy of a specified element from this bag.
* @param target
*   the element to remove from the bag.
* @postcondition
*   If target is in the bag, one copy is removed, returns true.
*   Otherwise the bag remains unchanged, returns false.
**/
public boolean remove(Integer target)  {
   int i;                                       // Find target
   for (i = 0; (i < manyItems) && (target != data[i]); i++) ;
   if (i == manyItems) return false;  // Not found.
   data[i] = data[--manyItems];        // Found. So remove.
   return true;
}
```

index	data
0	4
1	8
2	4
3	1
4	
5	
6	
7	
8	
9	

manyItems	4

Example 4: Applying remove(target)

Example:

- remove(8);

```
public boolean remove(Integer target)  {
   int i;                                       // Find target
   for (i = 0; (i < manyItems) && (target != data[i]); i++) ;
   if (i == manyItems) return false;  // Not found.
   data[i] = data[--manyItems];        // Found. So remove.
   return true;
}
```

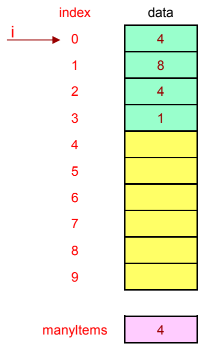
index	data
0	4
1	8
2	4
3	1
4	
5	
6	
7	
8	
9	

manyItems	4

Example 4:
Applying remove(target)

Example:

- remove(8);

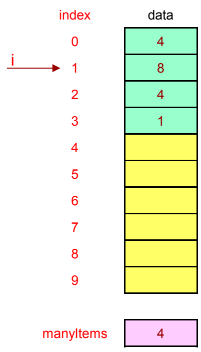



```
public boolean remove(Integer target)  {
   int i;                                    // Find target
   for (i = 0; (i < manyItems) && (target != data[i]); i++) ;
   if (i == manyItems) return false;  // Not found.
   data[i] = data[--manyItems];        // Found. So remove.
   return true;
}
```

Example 4:
Applying remove(target)

Example:

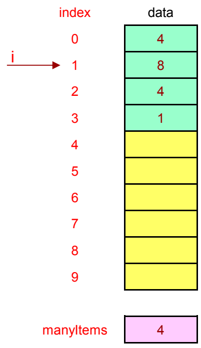
- remove(8);




```
public boolean remove(Integer target)  {
   int i;                                    // Find target
   for (i = 0; (i < manyItems) && (target != data[i]); i++) ;
   if (i == manyItems) return false;  // Not found.
   data[i] = data[--manyItems];        // Found. So remove.
   return true;
}
```

Example 4:
Applying remove(target)

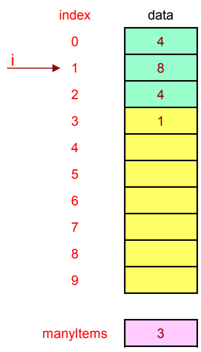
Example:
- remove(8);




```
public boolean remove(Integer target)  {
   int i;                              // Find target
   for (i = 0; (i < manyItems) && (target != data[i]); i++) ;
   if (i == manyItems) return false;  // Not found.
   data[i] = data[--manyItems];        // Found. So remove.
   return true;
}
```

Example 4:
Applying remove(target)

Example:
- remove(8);




```
public boolean remove(Integer target)  {
   int i;                              // Find target
   for (i = 0; (i < manyItems) && (target != data[i]); i++) ;
   if (i == manyItems) return false;  // Not found.
   data[i] = data[--manyItems];        // Found. So remove.
   return true;
}
```

Example 4:
Applying remove(target)

Example:

- remove(8);

	index	data
	0	4
i →	1	1
	2	4
	3	1
	4	
	5	
	6	
	7	
	8	
	9	

manyItems: 3

```
public boolean remove(Integer target)  {
   int i;                                   // Find target
   for (i = 0; (i < manyItems) && (target != data[i]); i++) ;
   if (i == manyItems) return false;  // Not found.
   data[i] = data[--manyItems];        // Found. So remove.
   return true;
}
```

Example 4:
Applying remove(target)

Example:

- remove(8);

index	data
0	4
1	1
2	4
3	1
4	
5	
6	
7	
8	
9	

manyItems: 3

```
public boolean remove(Integer target)  {
   int i;                                   // Find target
   for (i = 0; (i < manyItems) && (target != data[i]); i++) ;
   if (i == manyItems) return false;  // Not found.
   data[i] = data[--manyItems];        // Found. So remove.
   return true;
}
```

Example 4: Implementing Accessor size()

Code:

```
/**
* Determine the number of items in this bag.
* @param - none
* @return
*   the number of items in this bag
**/

public int size( )
{
   return manyItems;
}
```

index	data
0	4
1	1
2	4
3	
4	
5	
6	
7	
8	
9	

manyItems	3

Example:

- Size() ➔ 3