CS 302

ALGORITHMS AND DATA STRUCTURES

- A famous quote: Program = Algorithm + Data Structure

General Problem

- You have some data to be manipulated by an algorithm
  - E.g., list of students in a school
    - Each student is uniquely identified by a "primary" key → student ID
    - Each student has other info associated with him/her called the satellite data
      - Name, gpa, classes taken by the student, …

General Problem (continued)

- You want organize this data set such that you can answer certain operations very efficiently
  - Insert – a new student
  - Delete – a leaving student
  - Single Item Search – a student given its ID
  - Range Searches
  - Minimum, Maximum, Median
  - Sort the data set with respect to student ID
  - …

What are data structures?

- A way of storing data in a computer so that it can be used efficiently. (Wikipedia)
- An organization of information, usually in memory, for better algorithm efficiency. (http://www.nist.gov/dads)
- A proper representation of data and the operations allowed on that data to achieve efficiency. (Weiss)

Data structures as containers

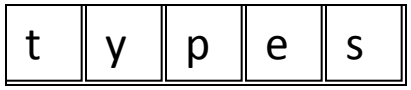
- A data structure can be thought of as a container of stuff (data).
- Some things you can do with a container:

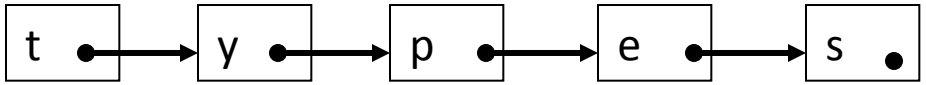
  - Add stuff to it
  - Remove stuff to it
  - Find specific stuff in it
  - Empty it (or check if its empty)

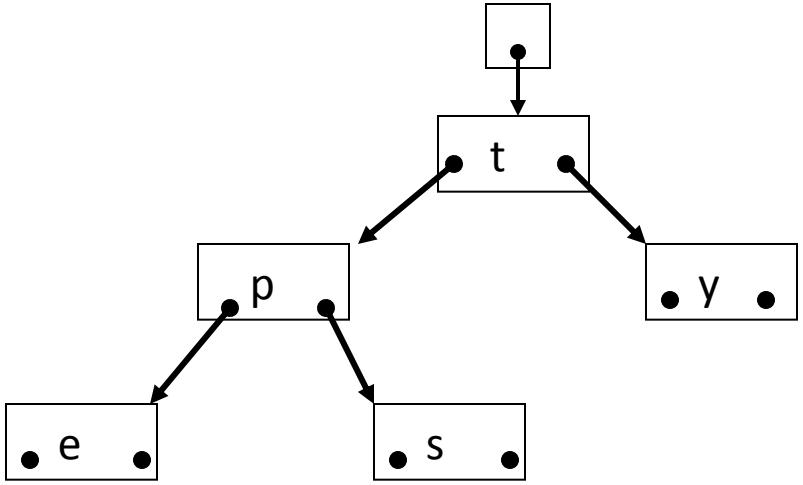
Data Structure

- A way of organizing a **collection** of Data
  - Array
  - ArrayList
  - HashMap
  - HashSet
  - …..
- The same collection of data may be represented by several data structures so there is a choice

Different Representations







- Data: types

Data Structure	Advantages	Disadvantages
Array	Quick insertion, very fast access if index known	Slow search, slow deletion, fixed size.
Ordered array	Quicker search than unsorted array.	Slow insertion and deletion, fixed size.
Stack	Provides last-in, first-out access.	Slow access to other items.
Queue	Provides first-in, first-out access.	Slow access to other items.
Linked list	Quick insertion, quick deletion.	Slow search.
Binary tree	Quick search, insertion, deletion (if tree remains balanced).	Deletion algorithm is complex.
Red-black tree	Quick search, insertion, deletion. Tree always balanced.	Complex.
2-3-4 tree	Quick search, insertion, deletion. Tree always balanced. Similar trees good for disk storage.	Complex.
Hash table	Very fast access if key known. Fast insertion.	Slow deletion, access slow if key not known, inefficient memory usage.
Heap	Fast insertion, deletion,	Slow access to other items.access to largest item.
Graph	Models real-world situations.	Some algorithms are slow and complex.

Classification of Data Structures

- Linear Data Structures
  - Unique ordering:
    - Lists, stacks, queues etc
- Hierarchical Data Structures
  - One root, internal nodes and one/many leaves
    - Trees
- Graph Data Structures
- Set Data Structures
  - No duplicates, elements occur in no fixed position

Stacks

- A Stack is a data structure where access is restricted to the most recently inserted item.
- The last item added to the stack is on top and is easily accessible (items below less so).
- Think of a pile of newspapers.

Characteristics of a Stack Structure

- A stack is a collection of elements, which can be stored and retrieved one at a time.
- Elements are retrieved in reverse order of their time of storage, i.e. the latest element stored is the next element to be retrieved.
- A stack is sometimes referred to as a Last-In-First-Out (LIFO) or First-In-Last-Out (FILO) structure. Elements previously stored cannot be retrieved until the latest element (usually referred to as the 'top' element) has been retrieved.

Stacks

Constrained version of linked list:

- New nodes can only be added to the top of the stack
- Nodes may only be removed from the top of the stack
- The depth of a stack is the number of elements it contains
- It is therefore a last-in, first-out structure (LIFO)

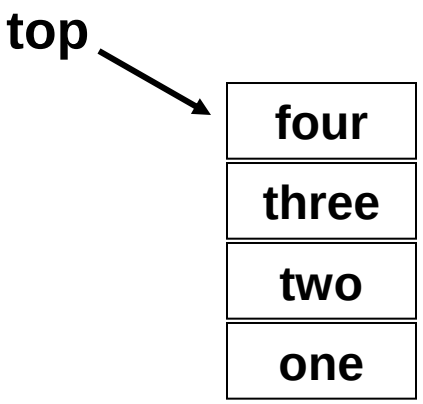
Typical Operations

OPERATION	PRE-CONDITION	POST-CONDITION
push (Object item)	stack not full	stack +1, item on top of stack
pop()	stack not empty	stack –1, top item removed
peek()	stack not empty	stack same
empty()	none	stack same
full()	none	stack same

Example

- push(one)
- push(two)
- push(three)
- push(four)
- pop()
- peek()
- push(five)
- empty()
- peek()

Manipulation of a Stack



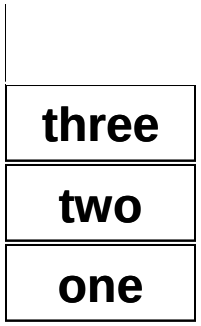
Stack with depth of 4

push(one)

push(two)

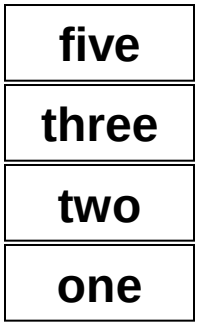
push(three)

push(four)



Stack with depth of 3

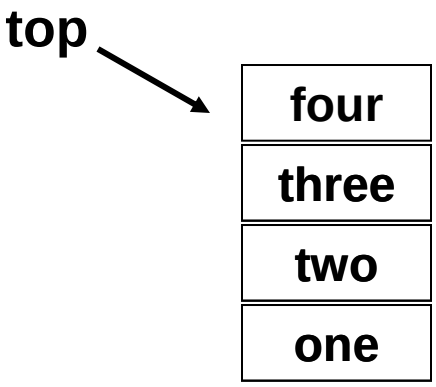
pop( )



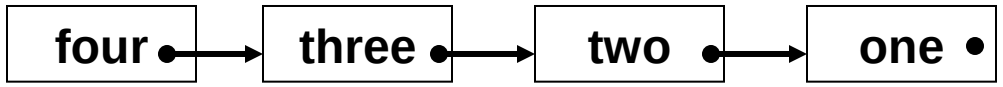
Stack with depth of 4

push(five)

List implementation of a stack



Logical view of a stack



Linked list implementation

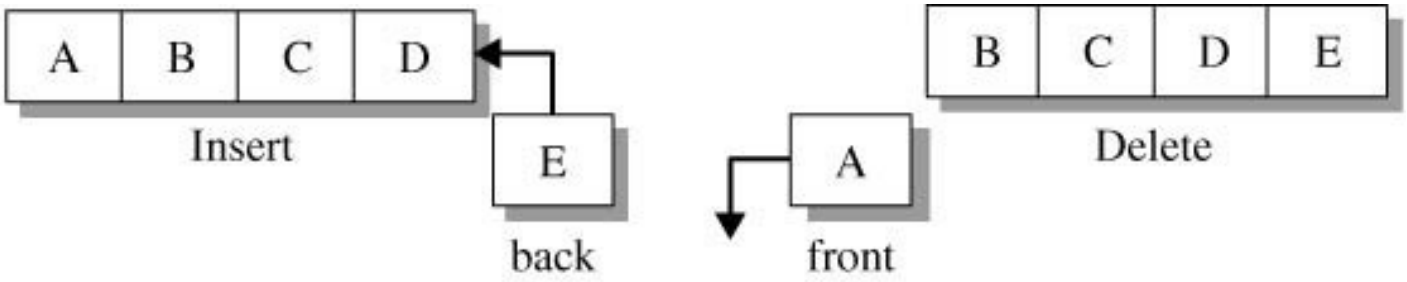
Queues

- A Queue is a data structure where access is restricted to the least recently inserted item.

- The first item added to the queue is in front and is easily accessible (items behind less so).

Queues

- A queue is a sequence of elements where;
  - Nodes can only be removed from the head/front
  - Nodes can only be added to the tail/end



- A queue is thus a first-in, first-out structure (FIFO)
- A queue has a length the number of elements it contains
- If a queue is empty it has length zero

Queue Behaviour

no queue		**front = back : 0 elements**
Fred joins queue	**Fred**	**front = back = Fred : 1 element**
Joe joins queue	**Fred Joe**	**front = Fred; back = Joe : 2 elements**
Harry joins queue	**Fred Joe Harry**	**front = Fred; back = Harry : 3 elements**
Fred catches bus	**Joe Harry**	**front = Joe; back = Harry : 2 elements**
Barney joins queue	**Joe Harry Barney**	**front = Joe; back = Barney : 3 elements**

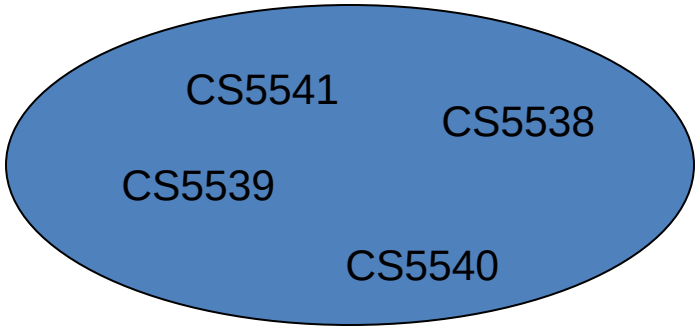
Typical Operations

OPERATION	PRE-CONDITION	POST-CONDITION
append (Object item)	q not full	queue+1, item added at end queue
remove()	q not empty	queue –1, item at front removed
front()	q not empty	queue same
isEmpty()	none	queue same
back()	q not empty	queue same
isFull()	none	queue same

Definition of a Set

A collection of objects

- No duplicates
- Ordering is unimportant



Sets: Some Terminology

- Cardinality
  - Number of members of a set
- Base type
  - The data type of the objects in the set
- Equality
  - Sets where membership is identical

set_a = {5, 3, 9, 1}

set_b = {1, 9, 3, 5}

set_a and set_b are identical sets.

Sets: Some Terminology

- Empty set (written as {}) (null set)
  - Has no members therefore cardinality of 0
- Disjoint sets
  - Sets which have no members in common
- Universal set
  - Contains all possible members of an associated data type
- Subset
  - All members in a subset also are members of the superset

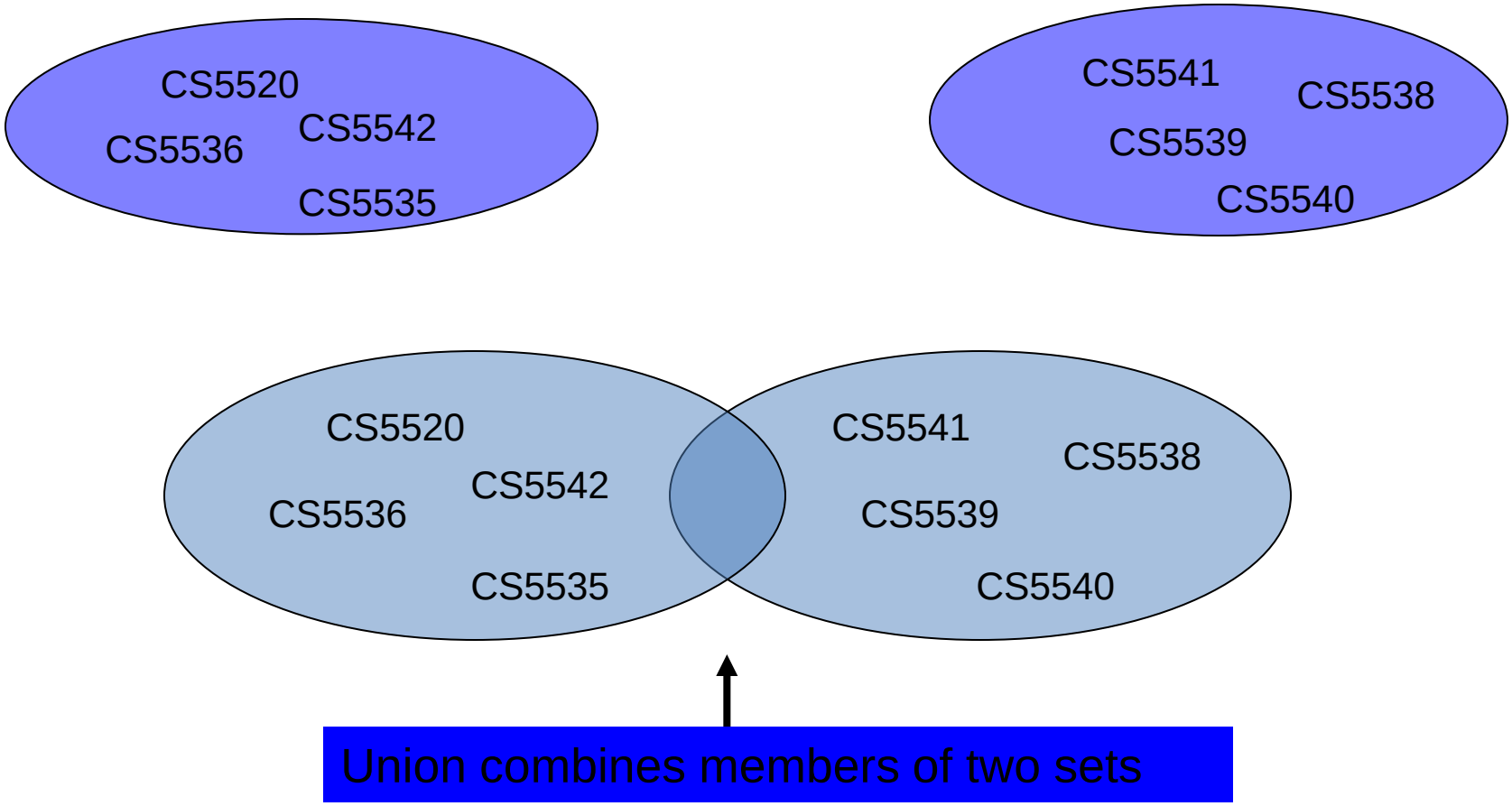
Typical Set Operations

OPERATION	PRE-CONDITION	POST-CONDITION
boolean add(Object item)	set not full, item is not null	item in set
boolean remove(Object item)	set not empty item not null	item not in set

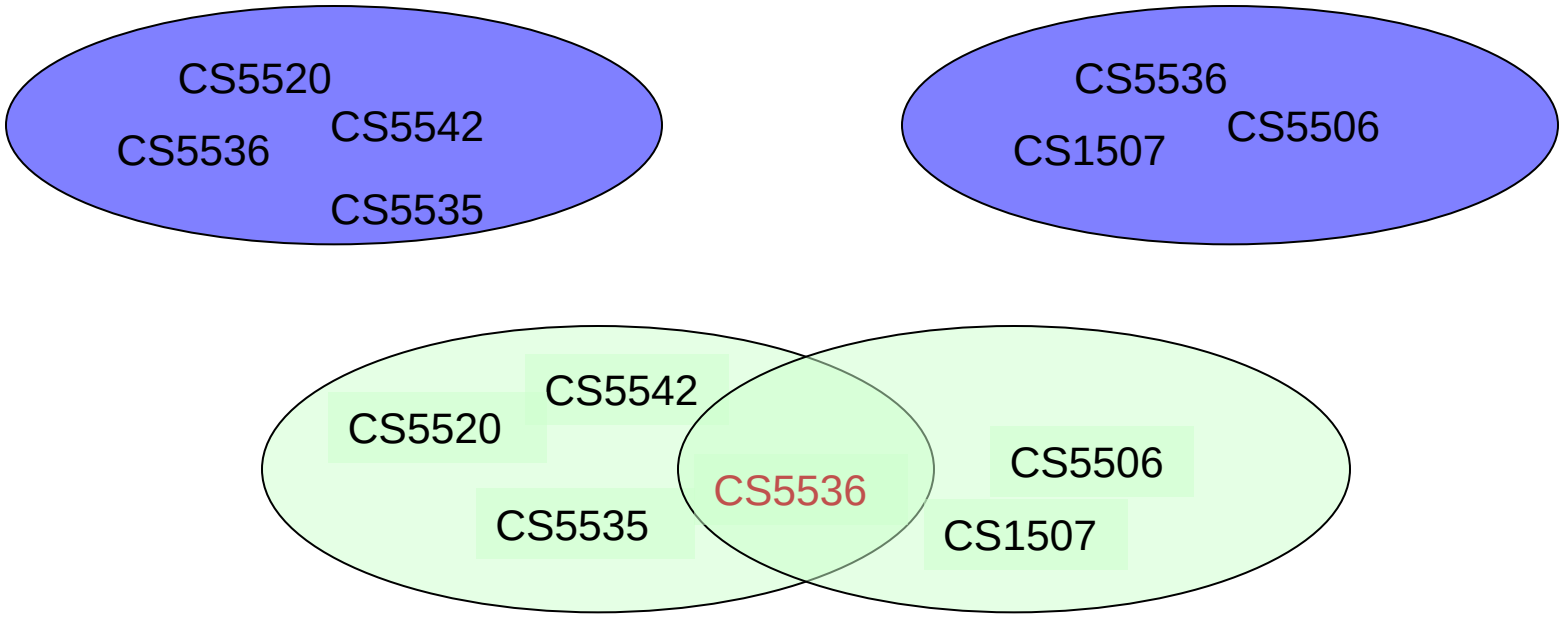
If the item is already in the set `add()` does nothing and returns false to indicate that nothing has been done.

If the item is not in the set `remove()` does nothing and returns false to indicate that nothing has been done.

Union

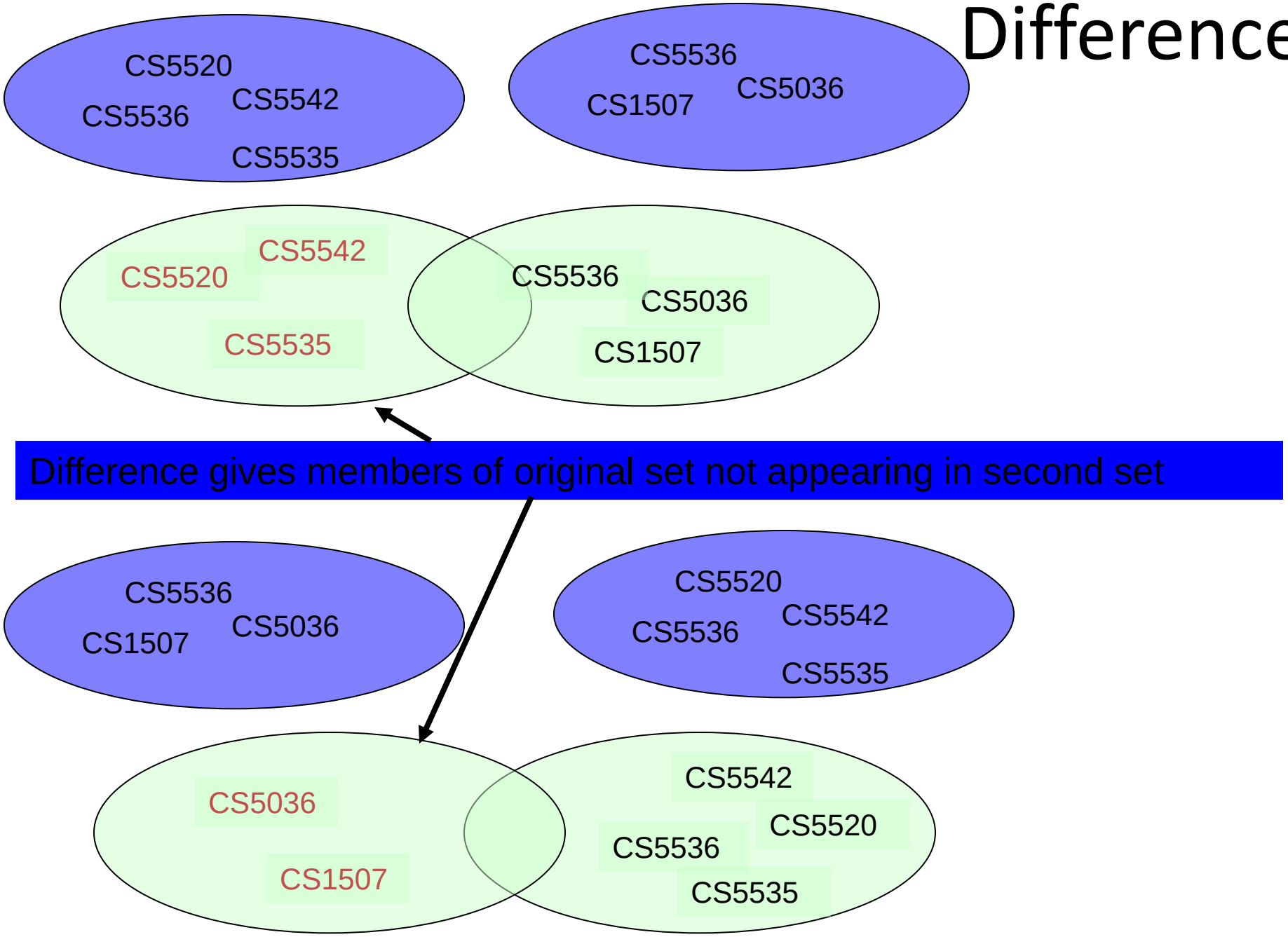

CS5520
CS5542
CS5536
CS5535
CS5541
CS5538
CS5539
CS5540
CS5520
CS5542
CS5536
CS5535
CS5541
CS5538
CS5539
CS5540
Union combines members of two sets

Intersection



Intersection gets common members of two sets

Difference


CS5520
CS5542
CS5536
CS5535
CS5536
CS5036
CS1507
CS5542
CS5520
CS5535
CS5536
CS5036
CS1507
Difference gives members of original set not appearing in second set
CS5536
CS5036
CS1507
CS5520
CS5542
CS5536
CS5535
CS5036
CS1507
CS5542
CS5520
CS5536
CS5535

Set Operations

OPERATION	PRE-CONDITION	POST-CONDITION
union(Set set2)	set2 is not null	all members of set2 added into set1
intersection(Set set2)	set2 is not null	new set contains common members
difference(Set set2)	set2 is not null	new set contains no members of set2 only members of set1

Assume

set1 is the original set

set2 is a second set

Set Operations

OPERATION	PRE-CONDITION	POST-CONDITION
boolean isEmpty()	none	set same
boolean contains (Object item)	item not null	set same
boolean subset(Set set2)	set2 not null	set same
int size()	none	set same

Examples

A = {4,3,8,6,7} B = {9,3,6,0} C = { }

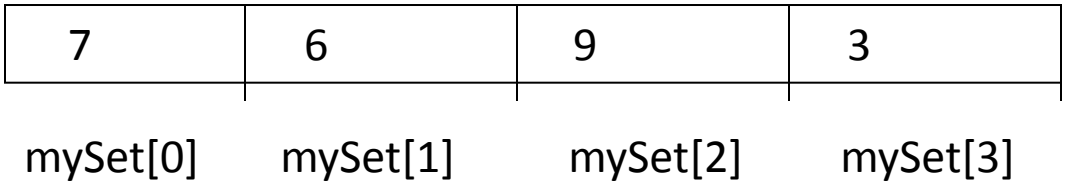
	Result	Value Returned
B.subset(A)	B = {9,3,6,0}	false
B.contains(9)	B = {9,3,6,0}	true
B.add(4)	B = {9,3,6,0,4}	true
A.remove(0)	A = {4,3,8,6,7}	false
A.difference(B)	{4,8,7}	void
B.difference(A)	{9,0}	void
C.isEmpty()	C = { }	true
A.union(B)	{4,3,8,6,7,9,0}	void
B.intersection(C)	{ }	void

Sets

- A set may be implemented as an array.

- A set may be implemented as a linked list.

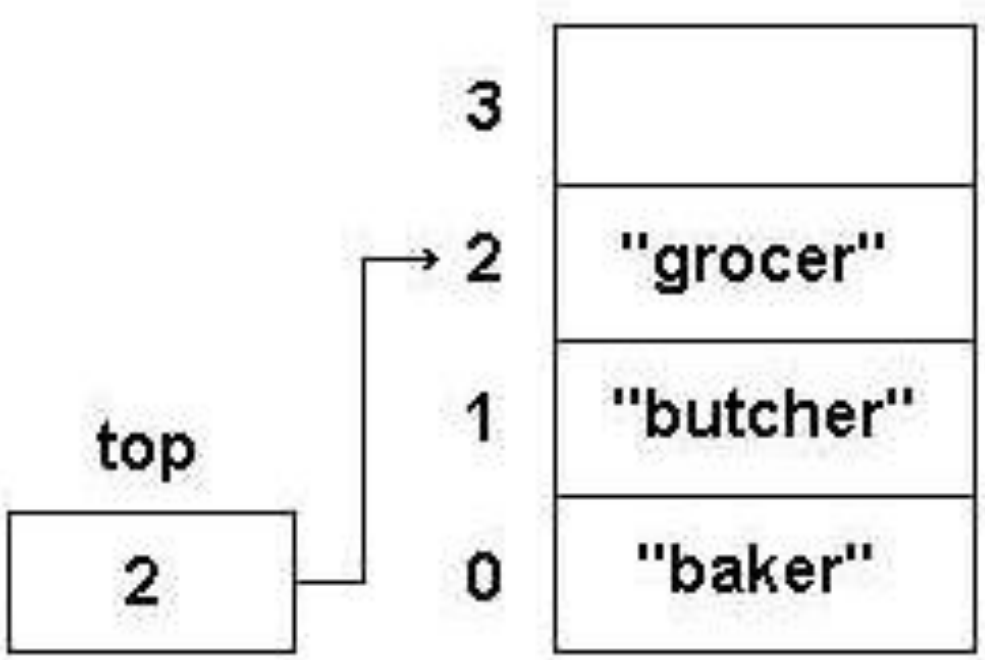
Implementation: Array



The above array is used to implement the set:
{7,6,9,3}

Implementation: Array

to represent the set { “baker”, “butcher”, “grocer” }



Lists

- A List is a collection of items in which the items have a position.
- We can access any item in a list by its index.
- Most obvious example: an array and linked list

What is a linked list?

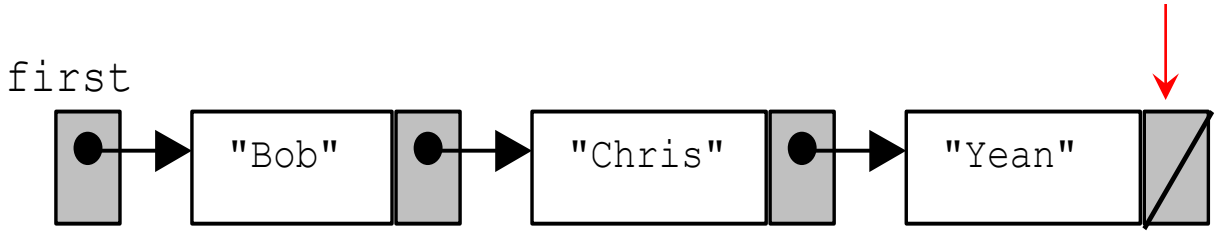
- A linked sequence of components or elements
- Each component (except the first) accessed from its predecessor
  - Predecessor: previous node
  - Successor: next node
- No direct access

Linked Lists in Java

- Each element of the same type
- Each element stored in a node, along with a reference to its successor
- Each node allocated dynamically (using new) and accessed by reference
- If there are no further nodes a null link is used
- Length of list is number of nodes
- An empty linked lists contains no nodes
- No limit on length, subject to available computer memory
- Elements may be linked backwards as well as forwards

List as a linked structure

- A linked structure provides an alternative implementation of collection classes
  - "Linked" structures are a collection of nodes
  - Each node stores a reference to an object and a reference to another node
    - **`first`** references the linked structure
    - The last element references nothing (we'll use **`null`**)

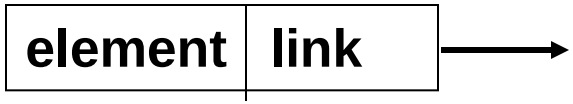


Sequential processing, not direct access like arrays

- Linked structures store a collection sequentially
  - Start at the beginning to find anything
  - Each element has a successor and predecessor
    - except the first and last element
- The collection of nodes has each storing
  - a reference to some value *it will be called* **`data`**
  - a reference to another node *it will be called* **`next`**
- Maintains memory on an as needed basis

Typical Node Declaration

```
class ListNode {
  // The element stored in the node
  private Object element;
  // Link to the next node
  private ListNode successor;
  // Methods (if any)
  ...
}
```



Different List Representations

1. Reference to first node
2. Header cell linking to first node
    (a) each node links to following node
    (b) each node links to another header cell
3. Header cell linking to first and last nodes
4. Header cell linking to spurious node
5. Header cell linking to circular list
6. Doubly linked list

Grouping Data Together

We may often find it necessary to group together a number of values or objects to be treated in the same way, e.g.

- names of students in a tutorial group
- exam marks for students in a subject
- players in a game
- CDs to be ordered

Obviously we need a better technique than creating a different variable for each value or object

Arrays

An array in Java is a **collection** of values of the same type

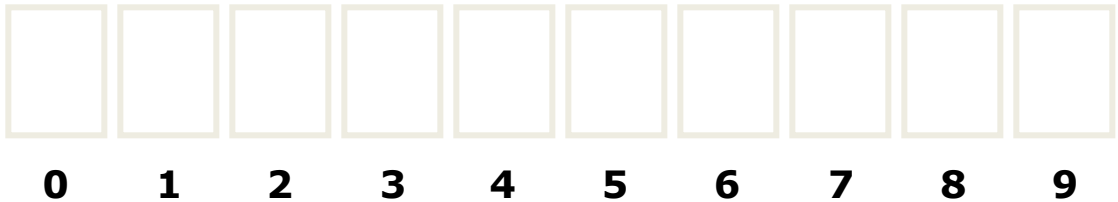
The elements in an array are held in contiguous memory locations

The array class is special - It has a small number of predefined methods, and a special notation of its own to invoke them

The notation used for arrays is a legacy notation with origins in other languages.

Arrays

- An array is like a set of pigeonholes or slots that can hold things
- The number of slots is fixed when the array is created - After creation, an array is a **fixed-length** structure
- The slots are numbered so that they can be accessed - In Java, the first slot is numbered **zero**
- The number of the last slot is therefore the **total number of slots minus 1**



Creating and Initialising an Array

- One way to create an array with some values in it is to initialize it when it is created
- In this case, the compiler will make the array as big as it needs to be to hold the values specified
- The length property is used to find the number of elements in the array.

```
public class SimpleArray
{
  // specifies an array of Strings
  // plus the initial values for the Strings are provided

  private String[] daysOfTheWeek = {"Monday",
                                    "Tuesday",
                                    "Wednesday",
                                    "Thursday",
                                    "Friday",
                                    "Saturday",
                                    "Sunday"};

  public SimpleArray()
  {
    // do nothing
  }

  public void getDaysOfTheWeek()
  {
    for(int i = 0; i < daysOfTheWeek.length; i++)
      System.out.println("Day " + (i+1) + ": " +
  daysOfTheWeek[i]);
  }
}
```

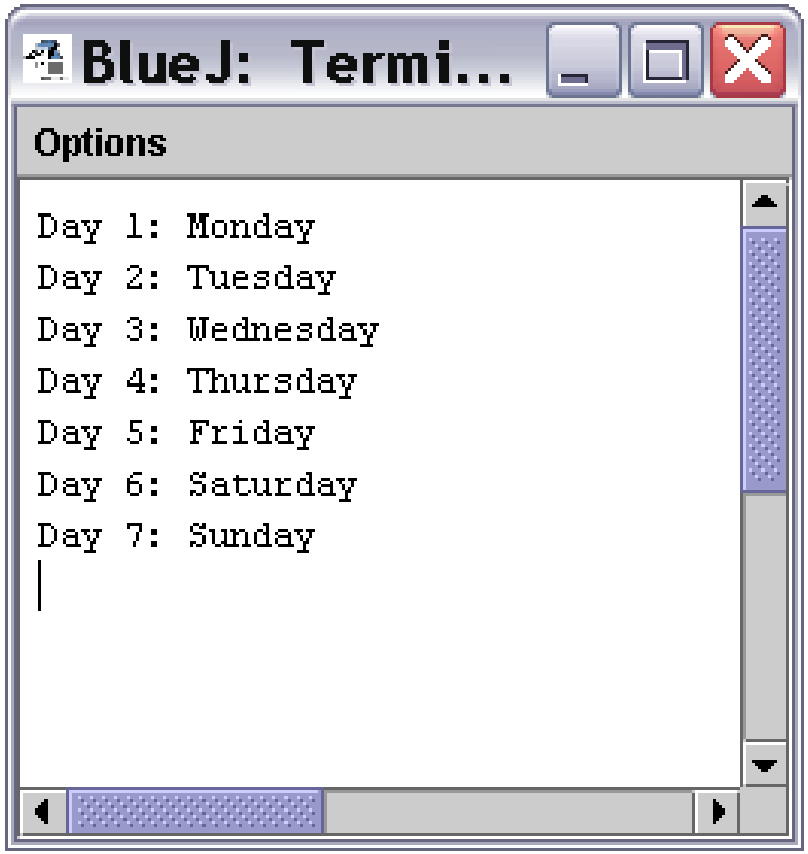
Name of the array variable

Initial values

Using a for loop to obtain the Strings from their array positions

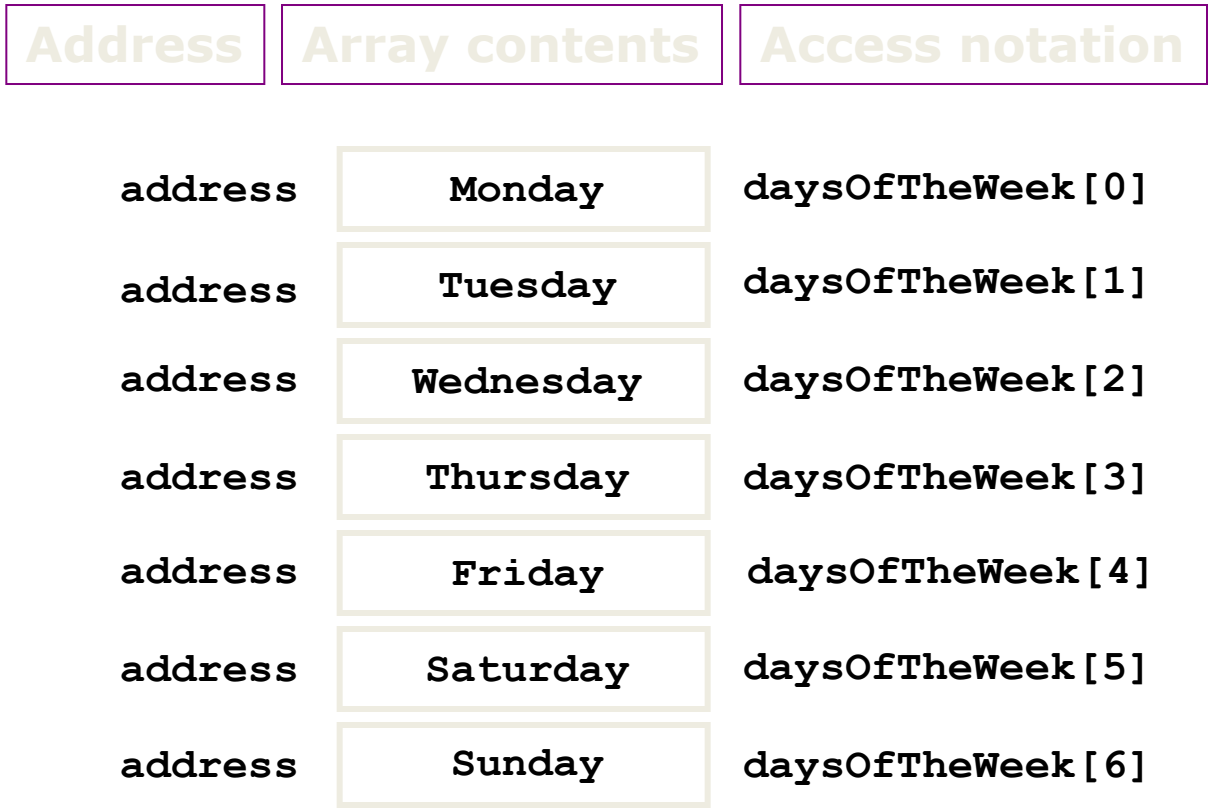
The Result…

Invoking the **getDaysOfTheWeek()** method…



Memory Allocations For Arrays

Each element of the array may be accessed using an index. It is actually the **address** of the element that is held.



Accessing Array Elements

- An array element is accessed by specifying its position in the array

  arrayName[index]

- Index is an integer value

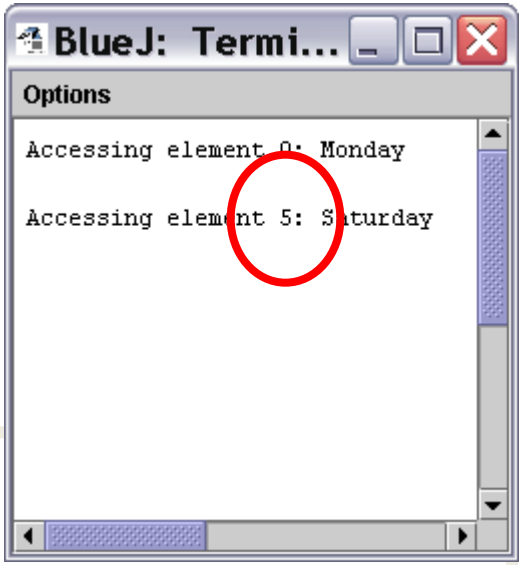
- Lowest value = **0**

- Highest value = **the size of the array -1**

10 ‘boxes’, numbering starts at 0, therefore 9 is the index of the ‘last box’

Evaluating an Index

- Index values may be obtained as the result of evaluating any expression that results in an int
- This is demonstrated below…




```
public void simpleArrayAccessDemo()
{
  System.out.println("Accessing element 0: "
                     + daysOfTheWeek[0] + "\n");
  int addMe = 3;
  System.out.println("Accessing element " + (addMe + 2)
                     + ": " + daysOfTheWeek[addMe + 2]);
}
```

2 Dimensional Array

- It is possible to have an array of **2 dimensions** to represent something that has **rows** and **columns**, e.g. seats in a theatre, pixels on a screen, positions on a chess board

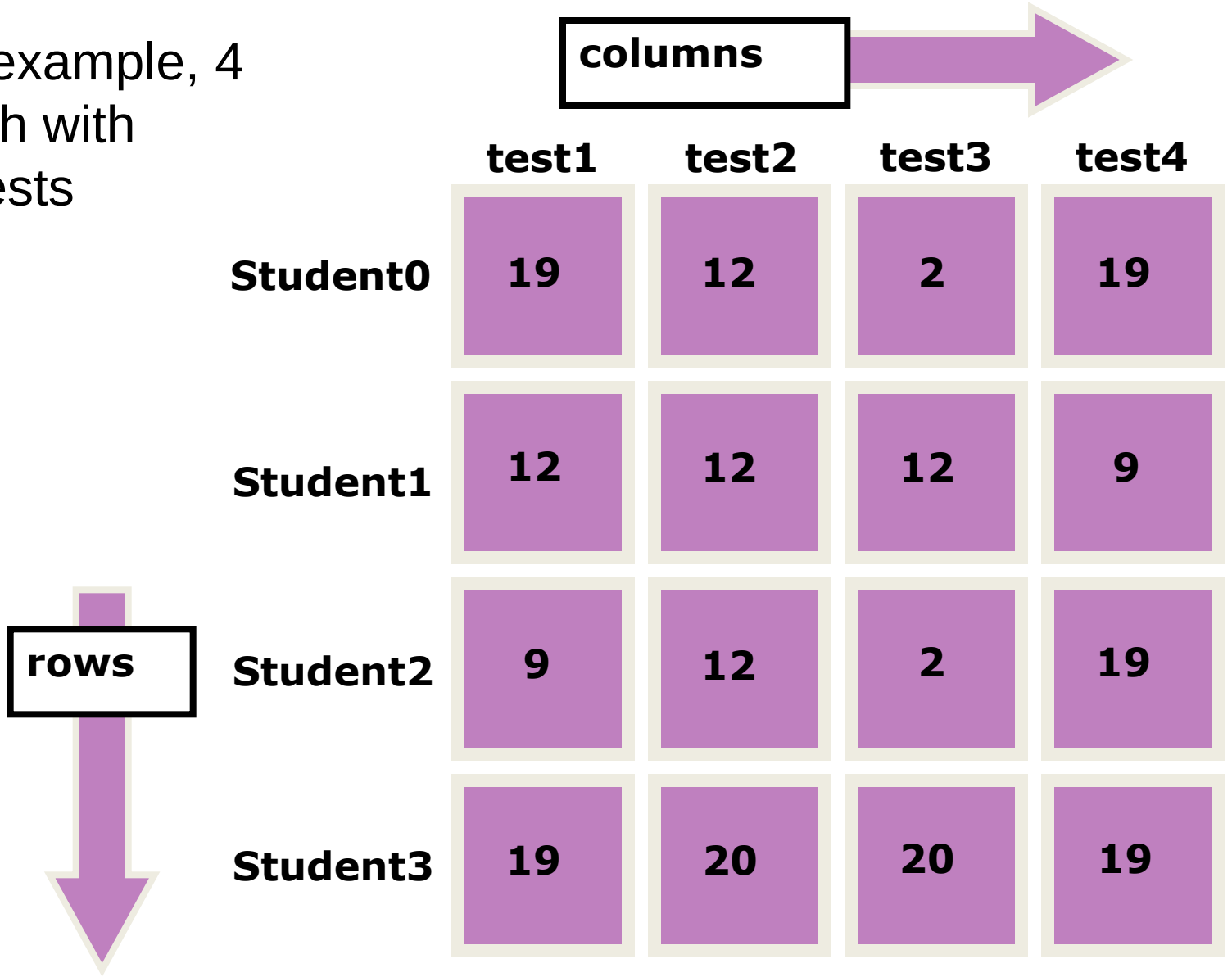
- In Java, a two-dimensional array is an array of one-dimensional arrays – ie. each element of the array is itself another one-dimensional array!

- All elements must be of the same type

- A two-dimensional array needs two indices (subscripts)

Data in the Array (Logical View)

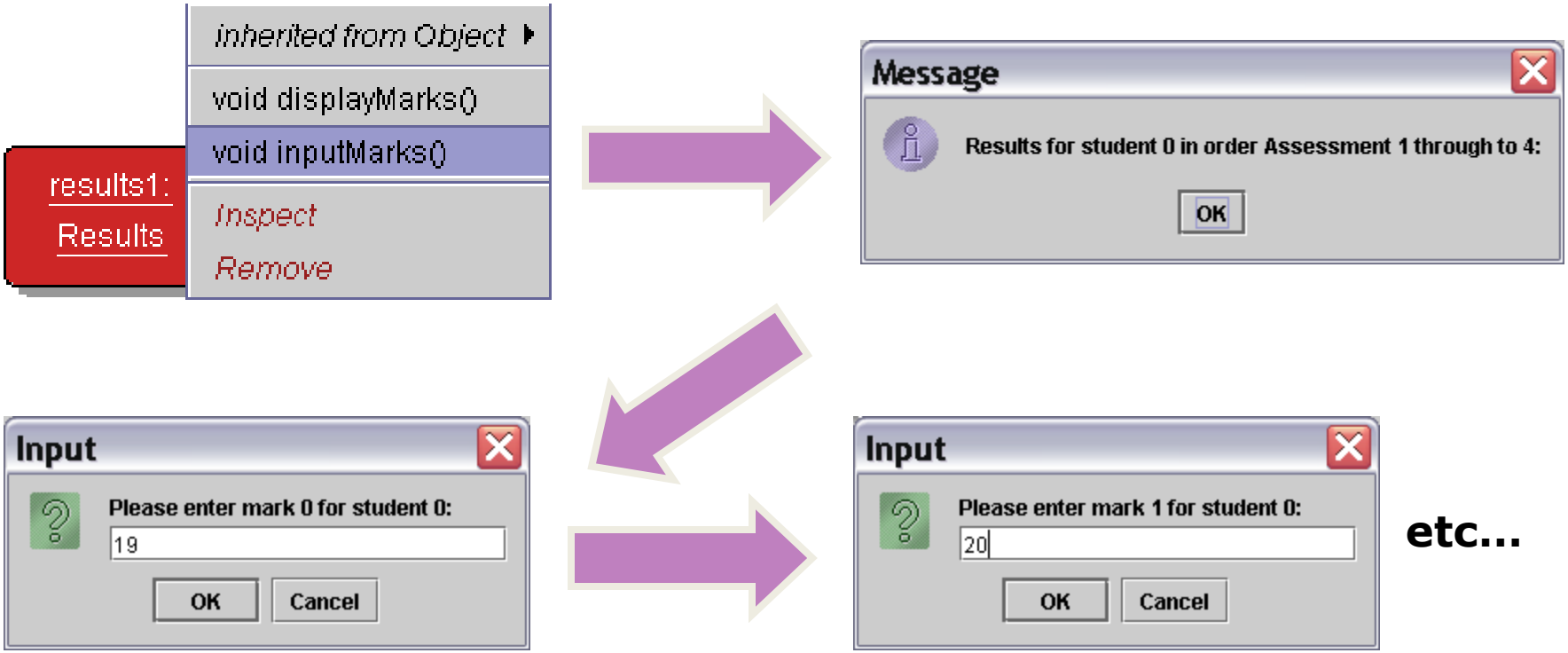
Consider for example, 4 students, each with marks for 4 tests



The Marks Application

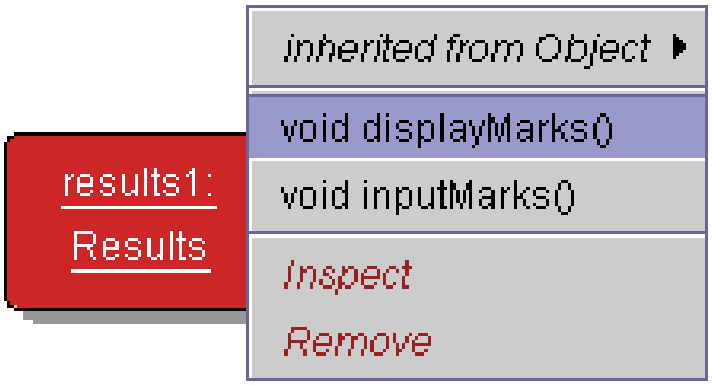
- The following class demonstrates the use of a two dimensional array for the storing of student marks

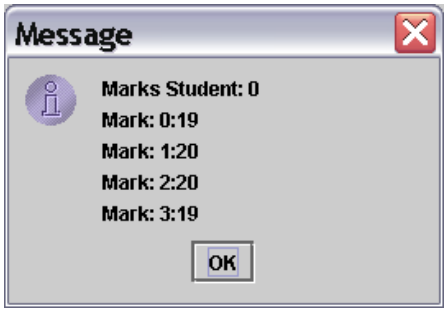
- Inputting the marks…

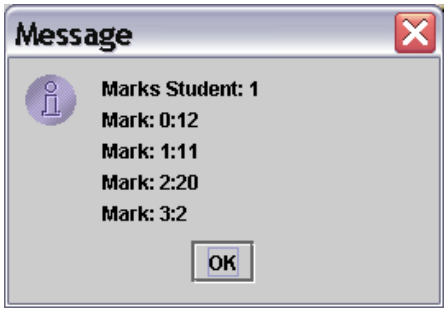


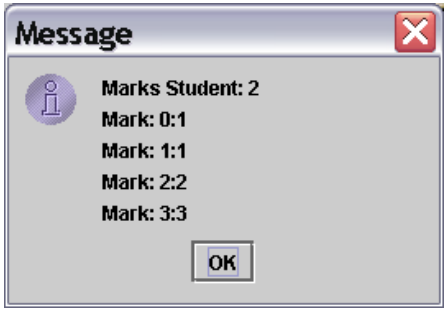
The Marks Application

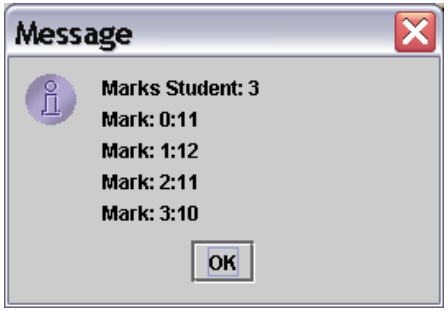
- Displaying the marks…

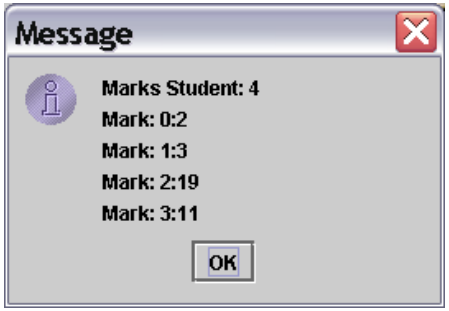






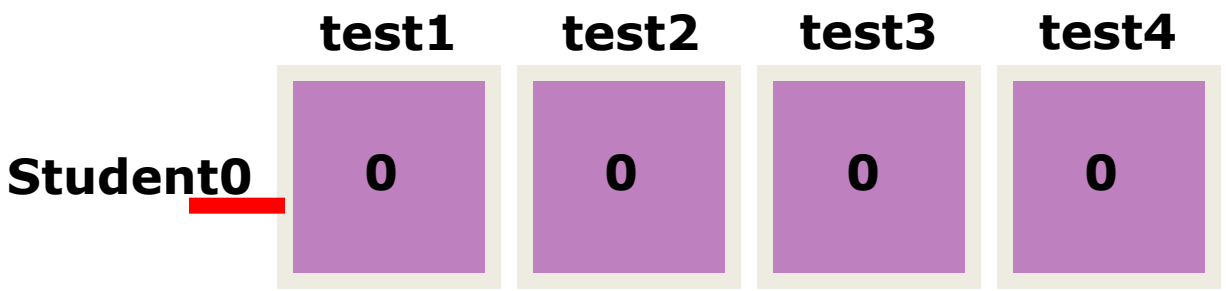






Resulting in…

First time round the loop, set the **first mark** to 0, the **second mark** to 0 and so forth



Second time round the loop, set the **first mark** to 0, the **second mark** to 0 and so forth