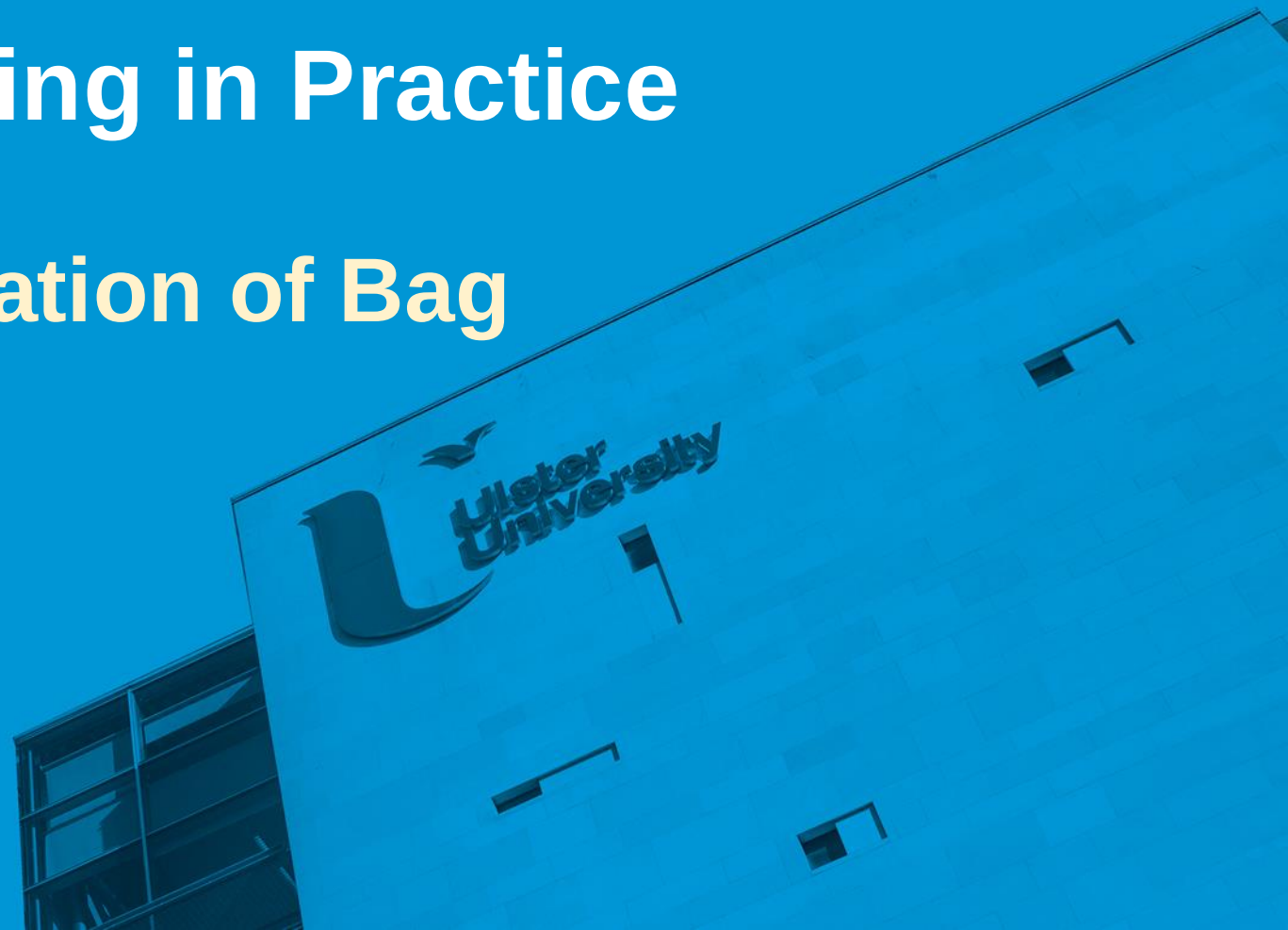
Ulster University

COM410 Programming in Practice

A3.3 Array Implementation of Bag



Fixed-Size Array to Implement Bag

- One way to implement the `Bag` ADT is to use a fixed-size array using a Java array to represent the entries in a bag (bag can become full like a real bag)
- Task is to define the methods previously specified in `BagInterface` where each public method corresponds to an operation on an instance of the `Bag` ADT
- Will result in the class `ArrayBag` that implements the interface

Fixed-Size Array Implementation

- Assuming a collection of `Building` objects

```
• int getCurrentSize()
• boolean isEmpty()
• boolean addNewEntry(Building newEntry)
• Building remove()
• boolean remove(Building anEntry)
• void clear()
• int getFrequencyOf(Building anEntry)
• boolean contains(Building anEntry)
• String toString()
```

- Definition for the class will be fairly involved (lots of methods), so should not define the entire class then attempt to test it
- Instead, identify a group of core methods to implement and test first

Core Methods

- When dealing with a collection such as a bag, you cannot test most methods until you have created the collection

```
int getCurrentSize()
boolean isEmpty()
boolean addNewEntry(Building newEntry)
Building remove()
boolean remove(Building anEntry)
void clear()
int getFrequencyOf(Building anEntry)
boolean contains(Building anEntry)
String toString()
```

- If `addNewEntry()` doesn't work, the `remove()` methods can't be tested
- To see if `add()` works you need a method that shows the contents of the bag
- Constructors are also needed, along with any methods used by the core methods

- The set of core methods should allow you to construct a bag, add objects to the bag, and look at the result

Core Methods

```
Building[] bag
int numberOfEntries
int DEFAULT_CAPACITY
```

```
int getCurrentSize()
boolean isEmpty
boolean addNewEntry(Building newEntry)
Building remove()
boolean remove(Building anEntry)
void clear()
int getFrequencyOf(Building anEntry)
boolean contains(Building anEntry)
String[] toString
```

```
boolean isArrayFull()
```

- We identify `addNewEntry()` and its helper method `isArrayFull()` as core methods – along with `toString()`
- These (plus a constructor) provide the minimum facilities to test a working application
- Also, identified an array to hold objects, the current number of entries and the length of the array as essential data components

Defining the Constructor

- Our constructor has 3 tasks
    1. Specify the array length
    2. Create the array (of whatever type the bag is to hold)
    3. Initialise the current number of entries to zero (initially the bag is empty)

Flexible Constructor

- Since our constructor needs to specify the length of the array (max size of the bag), a good approach is to provide 2 versions - one that sets a default size if the client expresses no preference and another to set a size specified by the client

```
private Building[] bag;
private int numberOfEntries;
private static final int DEFAULT_CAPACITY = 25;
```

Instance and class variables

```
public ArrayBag() {
   this(ArrayBag.DEFAULT_CAPACITY);
}
```

Constructor 1

```
public ArrayBag(int capacity) {
   this.bag = new Building[capacity];
   this.numberOfEntries = 0;
}
```

Constructor 2

Scenario

- Implement a skeleton for the class `ArrayBag` to store a collection of `Building` objects.
    - Provide the class header to implement the `BagInterface` previously defined and populate it with the definition of the class and instance variables, the constructors and empty methods for each of the public methods defined in the interface class.

Core Methods: `addNewEntry()`

```
Algorithm addNewEntry (newEntry)
// Add a new entry into the bag, returning true if space available and false otherwise

if the array is full
   return false
else add the newEntry into the array at the position pointed at by numberOfEntries
   increment numberOfEntries
   return true
```

```
Algorithm isArrayFull ()
// Returning true if all array elements are occupied and false otherwise

if numberOfEntries equals the size of the bag array
   return true
else return false
```

Core Methods: `addNewEntry()`

- Adding entries to an array that represents a bag with capacity 6 (until full):

- After each addition to the bag, increment the data field `numberOfEntries`

Scenario

- Add the core methods to the class `ArrayBag`. Include an implementation for `toString()`.
- Test your implementation by creating a new class `ArrayBagTest`. Copy the code from the `main()` method in `AnytownTest` and modify it to…
    i. Replace the existing `buildings` array with a new `ArrayBag` with capacity 13
    ii. Read the Buildings data from the input file and add each into the `ArrayBag` as it is created. Print the value returned by `addNewEntry()` after each addition.
    iii. Print the `ArrayBag` to the console and verify its contents
    iv. Using data of your choice, add three more Buildings into the bag, again reporting the value returned by the `addNewEntry()` method. Note that the value returned for the final `addNewEntry()` should be `false`, as the bag will have reached capacity
    v. Print the `ArrayBag` to the console and verify its contents, ensuring that only the first 13 objects were added.

Implementing More Methods

- Once the core methods are successfully implemented and tested the remaining methods can be defined (starting with the easiest ones)

```
public int getCurrentSize() {
    return this.numberOfEntries;
}

public boolean isEmpty() {
    return this.numberOfEntries == 0;
}
```

Implementing More Methods

```
Algorithm getFrequencyOf(anEntry)
// returns the number of times that anEntry appears in the bag

set counter to zero
for array elements from position 0 to position numberOfEntries - 1
   if current array element equals anEntry add 1 to counter
end for
return counter
```

- To do the comparison, the `equals()` method must be used (instead of ==)

  `anEntry.equals(bag[index])` NOT `anEntry == bag[index]`

- We assume the class to which the objects belong defines its own version of `equals()`

Implementing More Methods

```
Algorithm contains(anEntry)
// returns true if anEntry appears in the bag, false otherwise

set found to false
set index to 0
while not found and there are more elements to check
   if array element at index equals newEntry set found to true
   increment index
end while
return found
```

- Similar to `getFrequencyOf()` but search can stop when the first match is found

Scenario

- Java defines an `equals()` method for most built-in object types that returns `true` when the object passed as a parameter has the same value as the object that is the subject of the method, and `false` otherwise.
    - In our `Anytown` project, we will consider two buildings to be equal if they have the same address. Implement and test a suitable `equals()` method for your `Building` class.
    - Now, add the code for the `contains()` and `getFrequencyOf()` methods in the `ArrayBag` class and implement suitable tests in the `ArrayBagTest` class.

Methods That Remove Entries

```
public void clear() {
   while(!isEmpty()) remove();
}
```

- Keep removing objects until the bag is empty

```
public boolean isEmpty() {
   return this.numberOfEntries == 0;
}
```

- An empty bag contains no elements

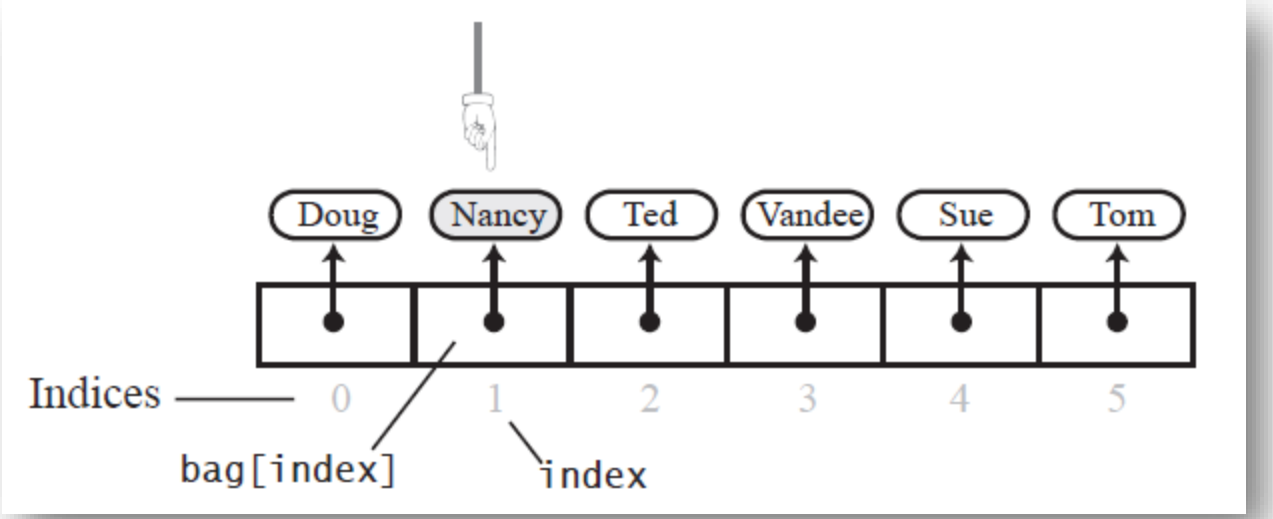
```
public Building remove() {
   Building result = null;
   if (numberOfEntries > 0) {
      result = this.bag[this.numberOfEntries - 1];
      this.bag[this.numberOfEntries - 1] = null;
      this.numberOfEntries--;
   }
   return result;
}
```

- First version – we will return to this later!
- Doesn't matter which object is removed, so remove the easiest one

Methods That Remove Entries

- Remove an occurrence of a specific named object from the bag
  - If an entry occurs more than once, we simply remove the first occurrence

- Assuming the bag is not empty:
  - Search the array bag
  - If *anEntry* equals bag[index] then note the value of index
- Now remove the entry at the index

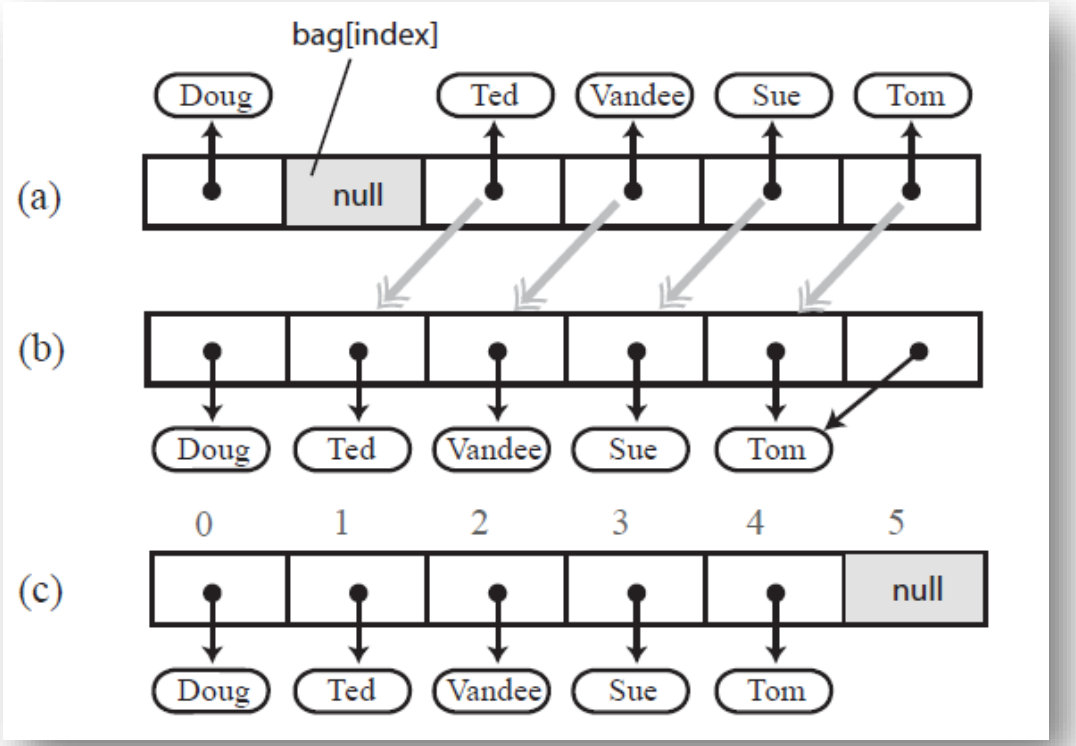


Methods That Remove Entries

- Setting the entry in the array to null will remove the reference to the entry, but it will leave a gap in the array (bag no longer stored in consecutive array locations)

- Could get rid of the gap by shifting the following entries and replacing the duplicate reference to the last entry with `null`

    a) A gap in the array bag
    b) Array after shifting entries (after the gap)
    c) Replace duplicate reference to the last entry with `null`

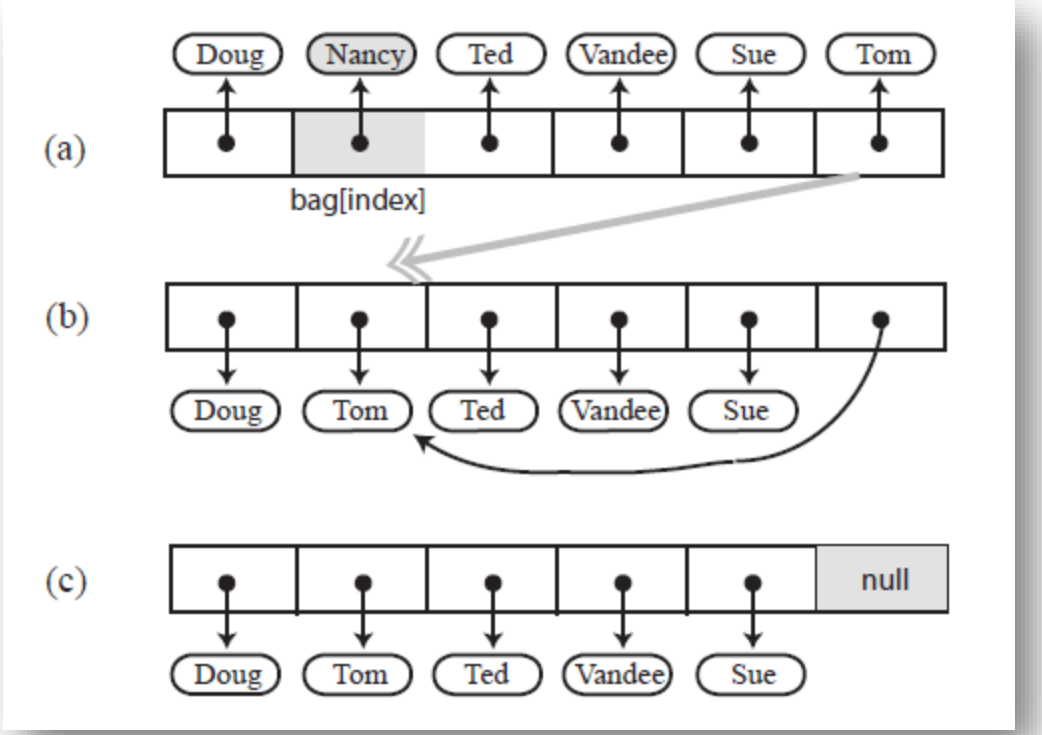


Methods That Remove Entries

- Previous approach would work but is somewhat time consuming!
- Remember that we aren't required to maintain any particular order for the elements

- Instead, we can replace the entry being removed with a copy of the last entry in the array and remove the duplicate:

    a) Locate the entry to be removed
    b) Copy the last entry in the array to the index of the entry to be removed
    c) Replace the last entry in the array with `null`



Methods That Remove Entries

- Both `remove()` and `remove(anElement)` can make use of a private helper method to remove and return an element from a specific position

```
public Building remove() {
    return removeElementAt(this.numberOfEntries - 1);
}
```

```
private Building removeElementAt(int index) {
    Building result = null;
    int lastIndex = this.numberOfEntries - 1
    if (!isEmpty() && index >= 0 && index <= lastIndex) {
        result = this.bag[index];
        this.bag[index] = this.bag[lastIndex];
        this.bag[lastIndex] = null;
        this.numberOfEntries--;
    }
    return result;
}
```

```
public boolean remove(Building anEntry) {
    boolean found = false;
    int index = 0;
    while (!found && index < this.numberOfEntries) {
        if (this.bag[index].equals(anEntry)) found = true;
        else index++;
    }
    if (found) removeElementAt(index);
    return found;
}
```

Scenario

- Add the remaining methods (i.e., those to remove elements, plus the helper method) to your `ArrayBag` class. All methods outlined in the interface class should now be implemented. Now verify the operation of the `ArrayBag` class by the following

    - Test your implementation by adding the file ***BagOfBuildingsTest.java*** to your **Anytown** project and providing suitable code at the 5 locations marked as `// TODO`
    - Run the `main()` method in ***BagOfBuildingsTest.java*** and trace the diagnostic comments provided to ensure that your `ArrayBag` implementation is working as expected.

Pros and Cons of Using an Array

- Adding an entry to the bag is fast
- Removing an unspecified entry is fast
- Removing a particular entry requires time to locate the entry
- A fixed-size array is limited in its capacity
- Approaches to dynamically increase the size of the array are possible (but increasing the size of the array requires time to copy its entries)

Challenge

- Create a new Java project called `Classroom` and populate with a class called `ArrayBag` that provides storage and functionality for a collection of `String` objects. In the `main()` method of the `Classroom` class, provide the following functionality.
    - Create an array of 10 student first names
    - Populate the bag with 100 random selections from the list of names and display the bag contents on the console.
    - Find the name most frequently held in the bag and report it and its number of occurrences. If there are multiple names with the joint-top number of occurrences, report all of the jointly most-popular names
    - Remove all occurrences of these names from the bag and display the number of names that remain in the bag
    - Generate a table of all remaining names with the number of times that they appear. Only names that appear in the bag at least once should be included.