IT 179
7

Introduction to Lists

For Next Lecture

Read chapter 2, sections 2.5 and 2.6

Watch video on
Linked Lists in Java

Introduction to Lists

- A **list** is a collection of elements, each with a position or index
- *Iterators* facilitate sequential access to lists
- Classes `ArrayList`, `Vector`, and `LinkedList` are *subclasses* of abstract class `AbstractList` and *implement* the `List` interface

`List` Interface and `ArrayList` Class

- An array is an indexed structure
- In an indexed structure,
  - elements may be accessed in any order using subscript values
  - elements can be accessed in sequence using a loop that increments the subscript
- With a Java array, you **cannot**
  - increase or decrease its length (length is fixed)
  - add an element at a specified position without shifting elements to make room
  - remove an element at a specified position and keep the elements contiguous without shifting elements to fill in the gap

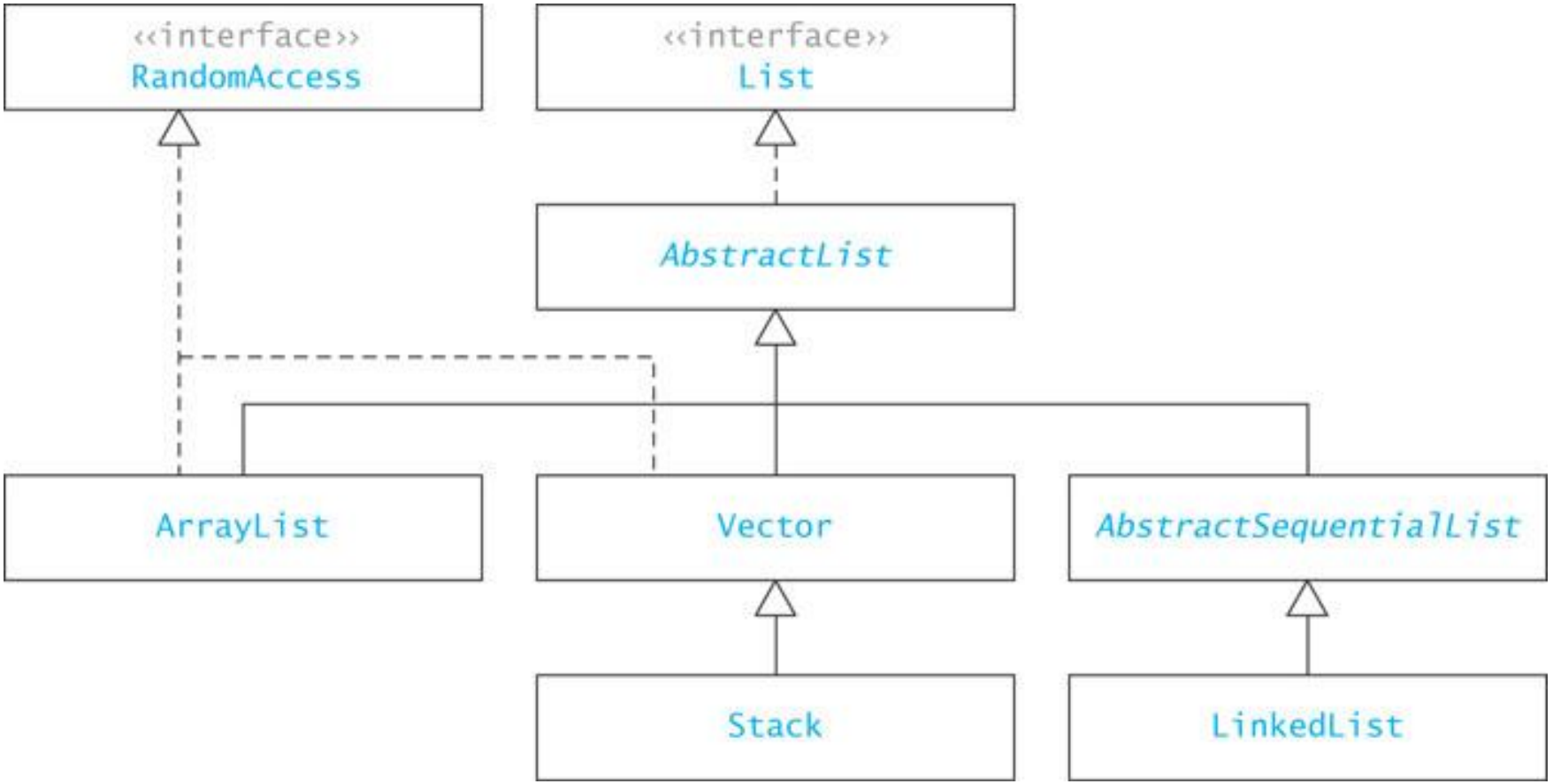
`List` Interface and `ArrayList` Class (cont.)

- Java provides a **`List` interface** as part of its API `java.util`
- Classes that implement the `List` interface provide the functionality of an indexed data structure and offer many more operations
- A sample of the operations:
  - Retrieve an element at a specified position (method get)
  - Replace an element at a specified position (method set)
  - Find a specified target value (method indexOf)
  - Add an element at either end (method add)
  - Remove an element (method remove)
  - Insert or remove an element at any position (method add)
  - Find the size of the list (method size)
  - Traverse the list structure without managing a subscript
- All classes introduced in this chapter support these operations, but they do not support them with the same degree of efficiency

Methods in the `List` Interface - E is a type parameter

Method	Behavior
`E get(int index)`	Returns the data in the element at position `index`
`E set(int index, E anEntry)`	Stores a reference to `anEntry` in the element at position `index`. Returns the data formerly at position `index`
`int size()`	Gets the current size of the `List`
`boolean add(E anEntry)`	Adds a reference to `anEntry` at the end of the List. Always returns `true`
`void add(int index, E anEntry)`	Adds a reference to `anEntry`, inserting it before the item at position `index`
`int indexOf(E target)`	Searches for `target` and returns the position of the first occurrence, or -1 if it is not in the `List`
`E remove (int index)`	Removes the entry formerly at position `index` and returns it

java.util.List Interface and Its Implementers



ArrayList al = new ArrayList();

Valid

Not Valid

ArrayList al = new List();

Valid

Not Valid

List al = new ArrayList();

Valid

Not Valid

List al = new List();

Valid

Not Valid

`List` Interface and `ArrayList` Class

- Unlike the `Array` data structure, classes that implement the `List` interface cannot store primitive types
- Classes must store values as **objects**

```
List<float> myList = new ArrayList<float>();
```

- This requires you to wrap primitive types, such as `int` and `double` in object wrappers, in this case, `Integer` and `Double`

List Interface and ArrayList Class

```
List<float> myList = new ArrayList<float>();
```

```
List<Float> myList = new ArrayList<Float>();
```

```
List<double> myList = new ArrayList<double>();
```

```
List<Double> myList = new ArrayList<Double>();
```

`ArrayList` **Class**

- The simplest class that implements the List interface
- An improvement over an array object
- Use when:
  - you will be adding new elements to the end of a list
  - you need to access elements quickly in any order

`ArrayList` Class (cont.)

- To declare a `List` object whose elements will reference `String` objects:

```
List<String> myList = new ArrayList<String>();
```

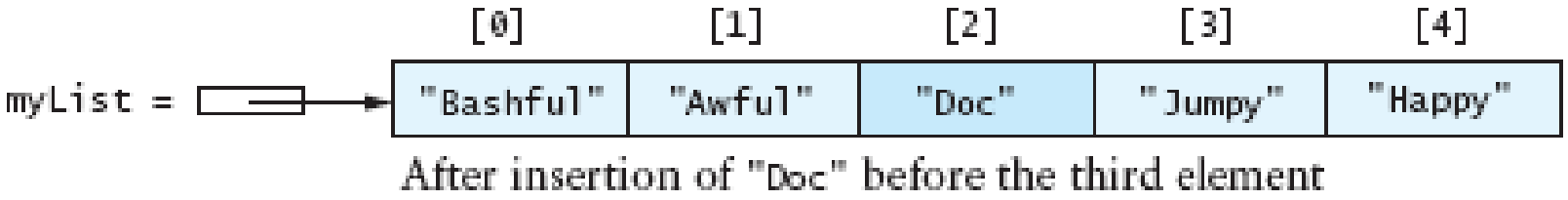
- The initial ArrayList is empty and has a default initial capacity of 10 elements
- To add strings to the list,

```
myList.add("Bashful");
myList.add("Awful");
myList.add("Jumpy");
myList.add("Happy");
```

ArrayList Class (cont.)

- Adding an element with subscript 2:

```
myList.add(2, "Doc");
```

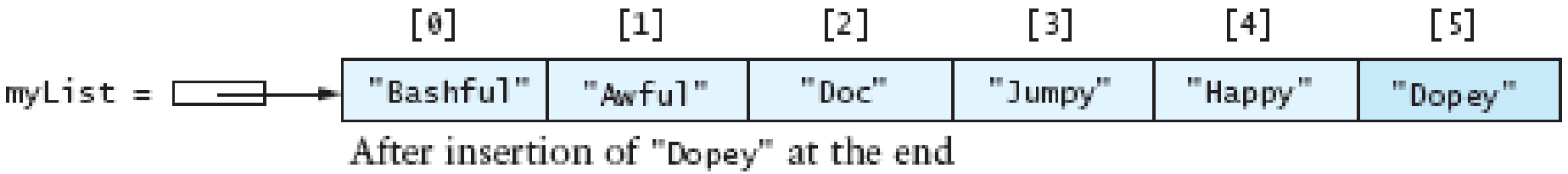


- Notice that the subscripts of "Jumpy" and "Happy" have changed from [2],[3] to [3],[4]

ArrayList Class (cont.)

- When no subscript is specified, an element is added at the end of the list:

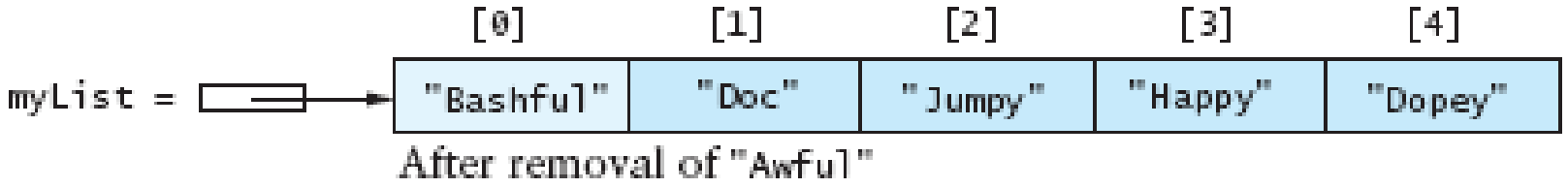
```
myList.add("Dopey");
```



ArrayList Class (cont.)

- Removing an element:


```
myList.remove(1);
```

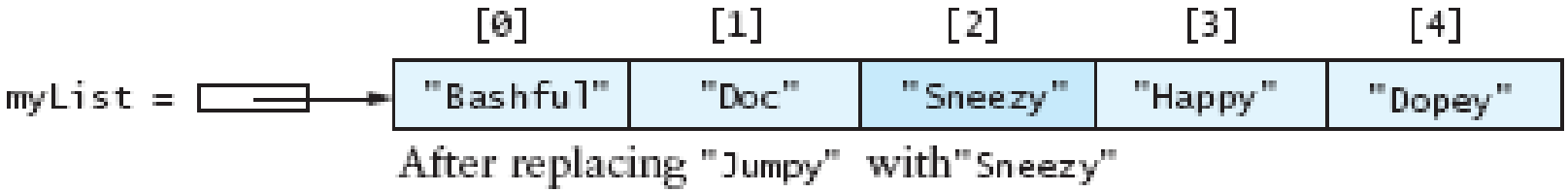


- The subscripts strings referenced by [2] to [5] have changed to [1] to [4]

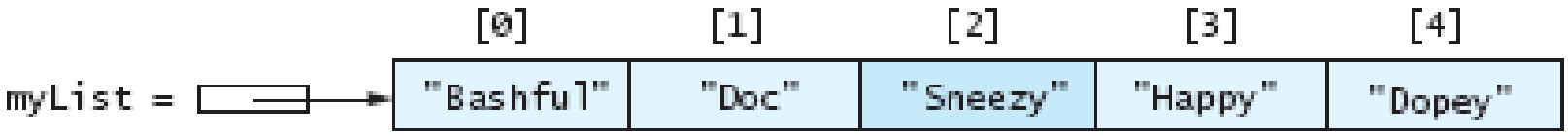
ArrayList Class (cont.)

- You may also replace an element:


```
myList.set(2, "Sneezy");
```



ArrayList Class (cont.)

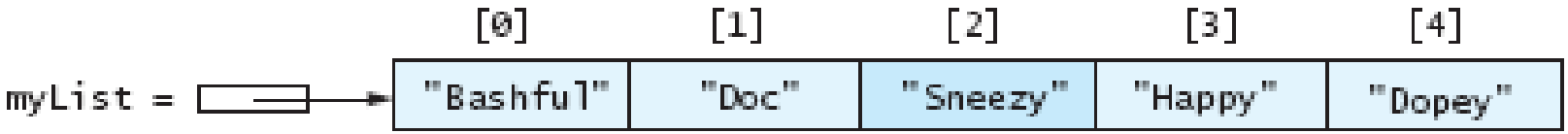


- You cannot access an element using a bracket index as you can with arrays (`array[1]`)
- Instead, you must use the `get()` method:

```
String dwarf = myList.get(2);
```

- The value of `dwarf` becomes `"Sneezy"`

ArrayList Class (cont.)



- You can also search an ArrayList:

```
myList.indexOf("Sneezy");
```

- This returns 2 while

```
myList.indexOf("Jumpy");
```

- returns -1 which indicates an unsuccessful search

Generic Collections

- The statements

```
List<String> myList = new ArrayList<String>();
List<Integer> myInts = new ArrayList<>();
var myFamily = new ArrayList<People>();
```

use a language feature called *generic collections* or *generics*

- The second statement uses the *diamond operator* <> to reduce redundancy
- The third statement uses the keyword `var` (introduced in Java 10) to simplify declarations when data type can be implied
- All 3 statements creates a `List` of objects of a specified type (`String`, `Integer`, or `People`); only references of the specified type - can be stored in the list
- The type parameter sets the data type of all objects stored in a collection

Generic Collections (cont.)

- The general declaration for generic collection is

```
CollectionClassName<E> variable = new CollectionClassName<>();
```

- The <E> indicates a type parameter
- Adding a noncompatible type to a generic collection will generate an error during compile time
- However, primitive types will be autoboxed:

```
ArrayList<Integer> myList = new ArrayList<>();
myList.add(Integer.valueOf(3)); // ok
myList.add(3); // also ok! 3 is automatically wrapped
                  in an Integer object
myList.add(new String("Hello")); // generates a type
                                    incompatability error
```

Applications of `ArrayList`

Section 2.3

Example Application of `ArrayList`

- Use of for each to access array elements in sequence

```
var someInts = new ArrayList<>();
int[] nums = {5, 7, 2, 15};
// Load ArrayList someInts from nums
for (int numNext : nums) {
    someInts.add(numNext);
}
```

`numNext` **is an** `int`**; it is automatically wrapped in an** `Integer` **object**

- Use of for each to access objects in an ArrayList in sequence

```
int sum = 0;
for (Integer sumNext : someInts) {
    sum += sumNext;
}
```

`sumNext` **is unboxed and its** `int` **value is added to** `sum`

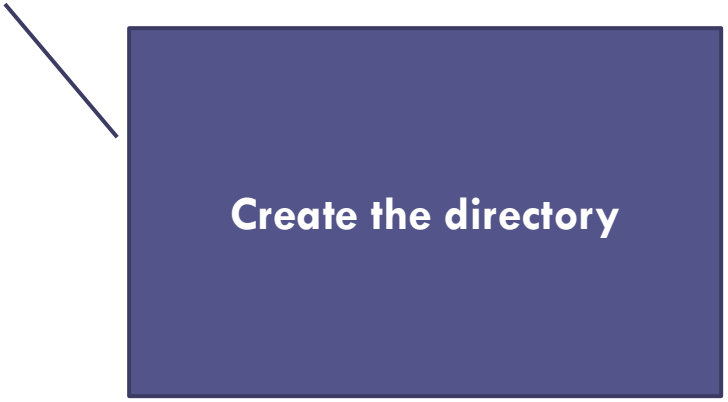
Phone Directory Application

```
public class DirectoryEntry {
   String name;
   String number;
}
```

Phone Directory Application (cont.)

```
public class DirectoryEntry {
   String name;
   String number;
}


private ArrayList<DirectoryEntry> theDirectory =
          new ArrayList<>();
```



Adding an object to the Directory

```
public class DirectoryEntry {
   String name;
   String number;
}

private ArrayList<DirectoryEntry> theDirectory =
          new ArrayList<>();

theDirectory.add(new DirectoryEntry("Jane Smith",
                                    "555-1212"));
```

Append a new `DirectoryEntry` object to the directory

Retrieving an entry if in the directory

```
int index = theDirectory.indexOf(new DirectoryEntry(aName,
                                                    ""));
if (index != -1)
  dE = theDirectory.get(index);
else
  dE = null;
```

Method `indexOf` searches `theDirectory` by applying the `equals` method for class `DirectoryEntry`. Assume `DirectoryEntry`'s `equals` method compares name fields

`dE` **references the directory entry with name** `aName`.

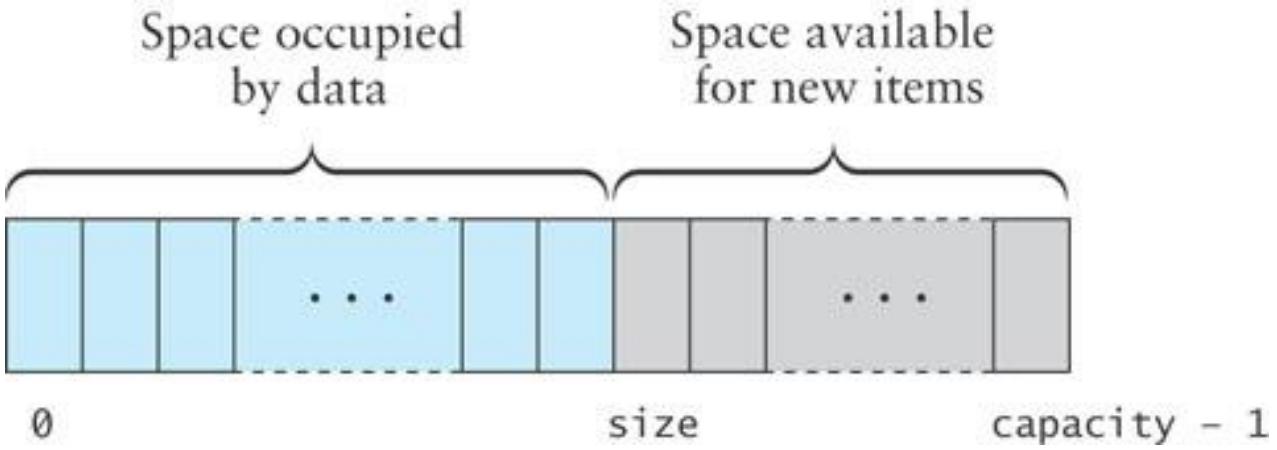
If `aName` **is not found,** `dE` **is** `null`.

Implementation of an `ArrayList` Class

Section 2.4

Implementing an `ArrayList` Class

- `KWArrayList`: a simple implementation of `ArrayList`
  - Physical size of array indicated by data field capacity
  - Number of data items indicated by the data field size



Why KWArrayList?

ELLIOT B. KOFFMAN AND PAUL A. T. WOLFGANG

KWArrayList **Fields**

```
import java.util.*;


/** This class implements some of the methods of the Java
ArrayList class
*/
public class KWArrayList<E> {
  // Data fields
  /** The default initial capacity */
  private static final int INITIAL_CAPACITY = 10;
  /** The underlying data array */
  private E[] theData;
  /** The current size */
  private int size = 0;
  /** The current capacity */
  private int capacity = 0;
}
```

KWArrayList Constructor

```
public KWArrayList () {
    capacity = INITIAL_CAPACITY;
    theData = (E[]) new Object[capacity];
}
```

This statement allocates storage for an array of type `Object` and then casts the array object to type E[]

Although this may cause a compiler warning, it's fine

Implementing `ArrayList.add(E)`

- We will implement two add methods
- One will append at the end of the list

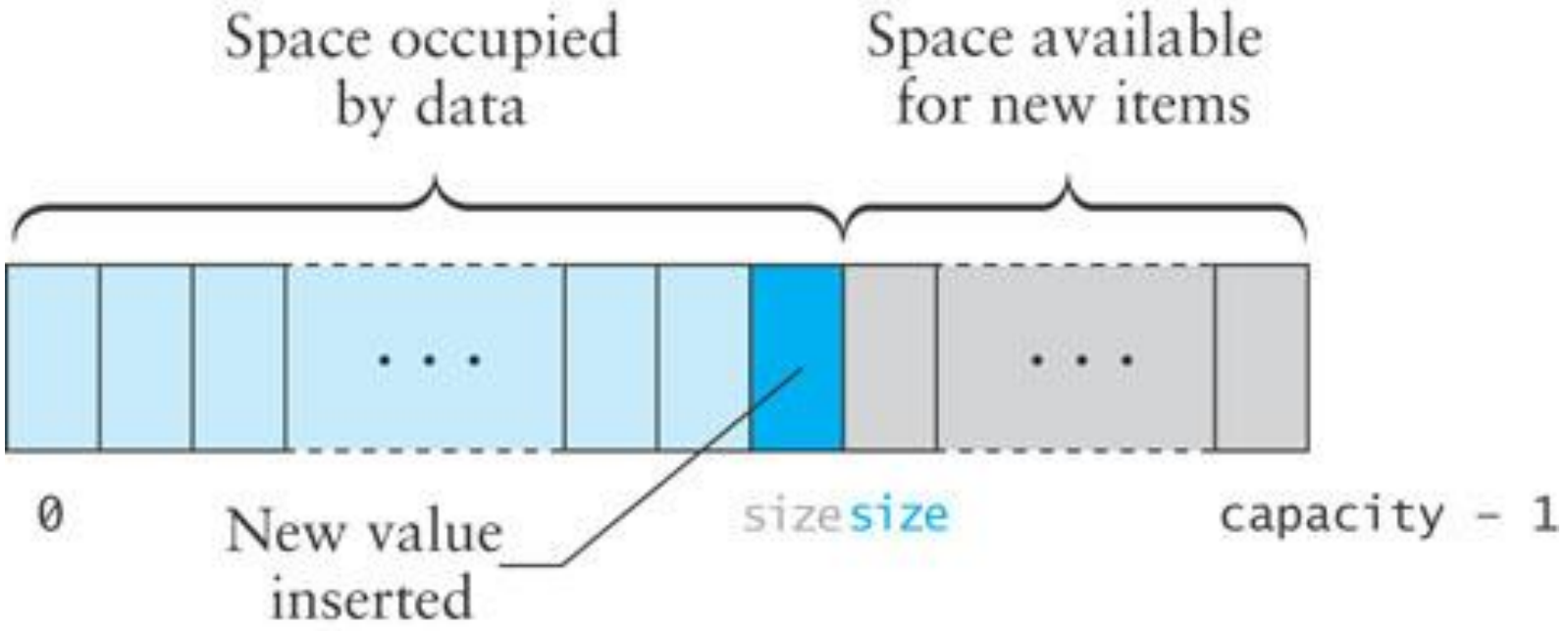
  `add(E e)`

- The other will insert an item at a specified position

  add(int index, E element)

Implementing `ArrayList.add(E)` (cont.)

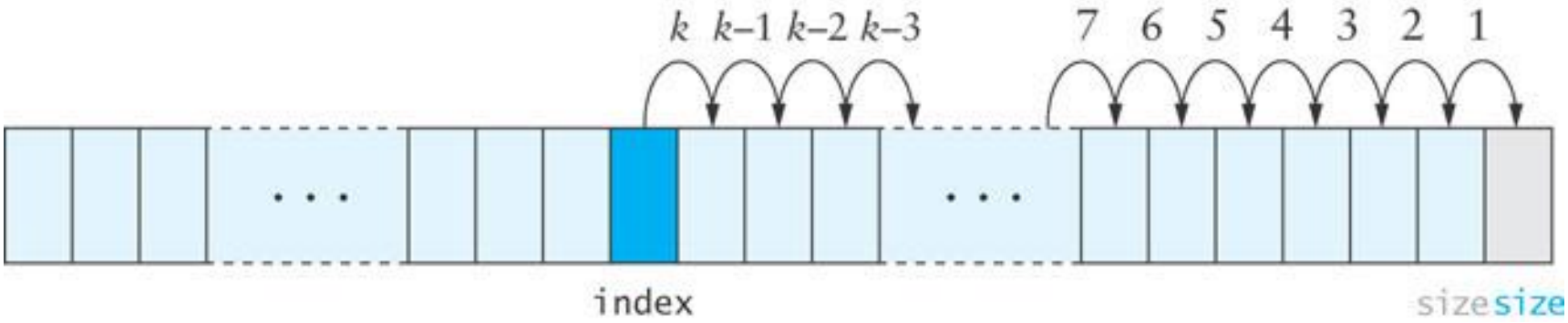
- If `size` is less than capacity, then to append a new item
  1. insert the new item at the position indicated by the value of `size`
  2. increment the value of `size`
  3. return `true` to indicate successful insertion



The method add(E e) runs in O(?)

Implementing `ArrayList.add(int index,E anEntry)`

- To insert into the middle of the array, the values at the insertion point are shifted over to make room, beginning at the end of the array



Implementing `ArrayList.add(index,E)`

```
public void add (int index, E anEntry) {

   // check bounds
   if (index < 0 || index > size) {
     throw new ArrayIndexOutOfBoundsException(index);
   }

   // Make sure there is room
   if (size >= capacity) {
     reallocate();
   }

   // shift data
   for (int i = size; i > index; i--) {
     theData[i] = theData[i-1];
   }

   // insert item
   theData[index] = anEntry;
   size++;
}
```

The method add(int index, E e) runs in O(?)

get **Method**

```
public E get (int index) {
   if (index < 0 || index >= size) {
      throw new
ArrayIndexOutOfBoundsException(index);
   }
   return theData[index];
}
```

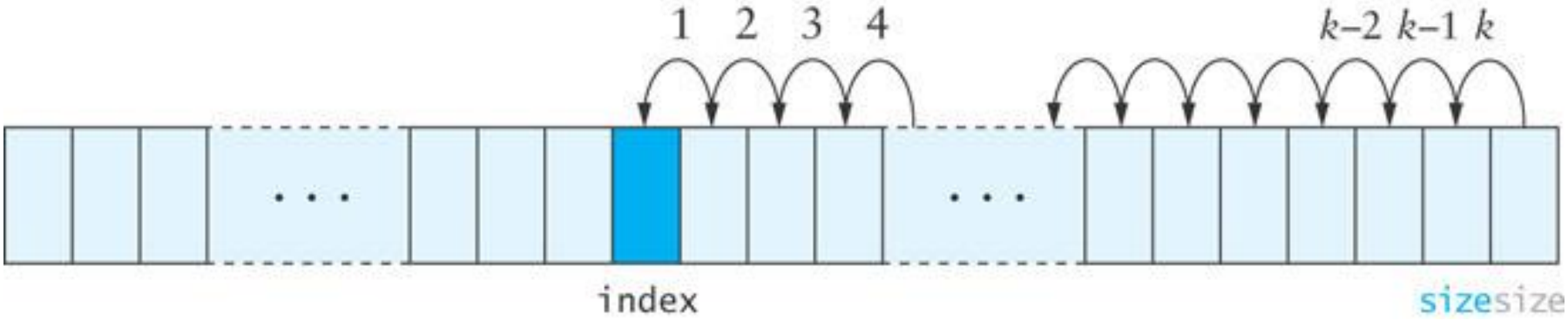
The get() method runs in O(?) time

set Method

```
public E set (int index, E newValue) {
  if (index < 0 || index >= size) {
    throw new
  ArrayIndexOutOfBoundsException(index);
  }
  E oldValue = theData[index];
  theData[index] = newValue;  #Insert the new value
  return oldValue;
}
```

The method set() runs in O() time

`remove` **Method**



- When an item is removed, the items that follow it must be moved forward to close the gap
- Begin with the item closest to the removed element

remove Method (cont.)

```
public E remove (int index) {

  if (index < 0 || index >= size) {
    throw new
  ArrayIndexOutOfBoundsException(index);
  }

  E returnValue = theData[index];

  for (int i = index + 1; i < size; i++) {
    theData[i-1] = theData[i];
  }

  size--;
  return returnValue;
}
```

`reallocate` **Method**

- Create a new array that is twice the size of the current array and then copy the contents of the new array

```
private void reallocate () {
   capacity *= 2;
   theData = Arrays.copyOf(theData,
  capacity);
}
```

reallocate Method (cont.)

```
private void reallocate () {
   capacity *= 2;
   theData = Arrays.copyOf(theData,
   capacity);
}
```

The reason for doubling capacity is to spread out the cost of copying;

Performance of `KWArrayList`

- The `set` and `get` methods execute in constant time: O(1)
- Inserting or removing general elements is linear time: O(*n*)
- Adding at the end is (usually) constant time: O(1)
  - With our reallocation technique the average is O(1)
  - The worst case is O(n) because of reallocation