CS 570: Data Structures Collections Framework: Lists

Instructor: Iraklis Tsekourakis

Email: itsekour@stevens.edu

CHAPTER 2 (PART 1)

Lists and the `Collections` Framework

Chapter Objectives

- The `List` interface
- Writing an array-based implementation of `List`
- Linked list data structures:
  - Singly-linked
  - Doubly-linked
  - Circular
- Implementing the `List` interface as a linked list
- The `Iterator` interface
  - Low priority for CS 570
- The Java `Collections` framework (hierarchy)
  - Low priority for CS 570

Week 4

- Reading Assignment: Koffman and Wolfgang, Sections 2.2 – 2.4

Introduction

- 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

The `List` Interface and `ArrayList` Class

Section 2.2

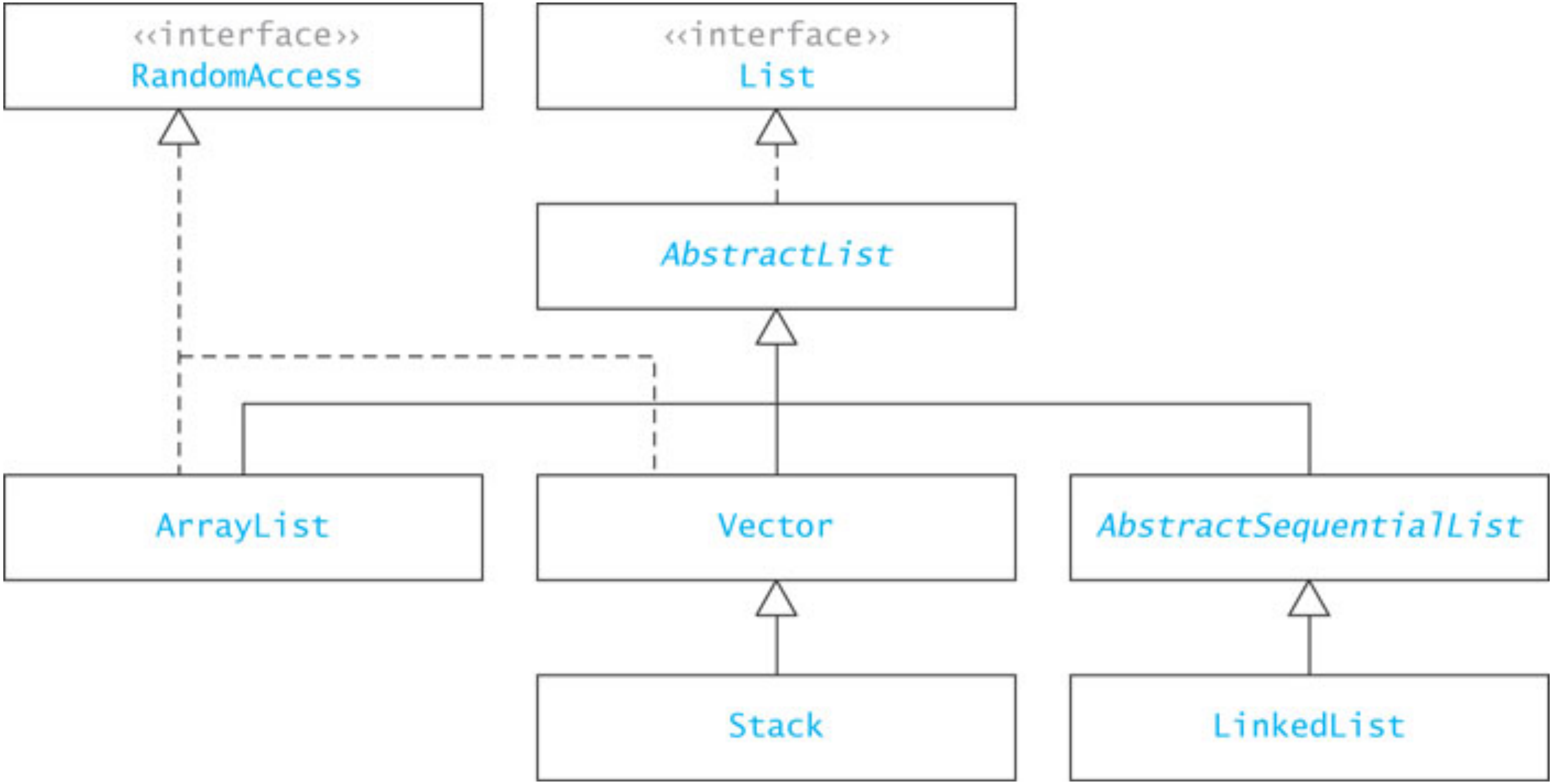
`List` Interface and `ArrayList` Class

- 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 the Java `Array` object, 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:
  - Obtain an element at a specified position
  - Replace an element at a specified position
  - Find a specified target value
  - Add an element at either end
  - Remove an element from either end
  - Insert or remove an element at any position
  - 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

`java.util.List` Interface and its Implementers



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

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

ArrayList Class (cont.)

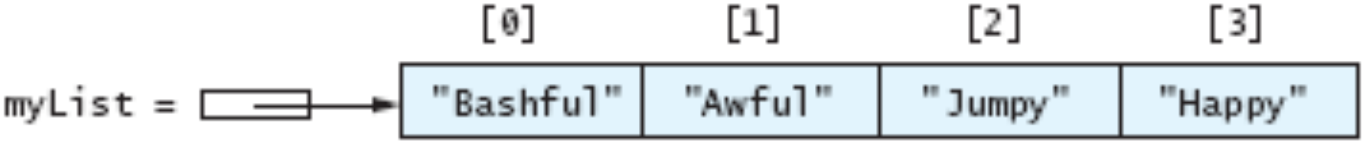
- To declare a List "object" whose elements will reference String objects:

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

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

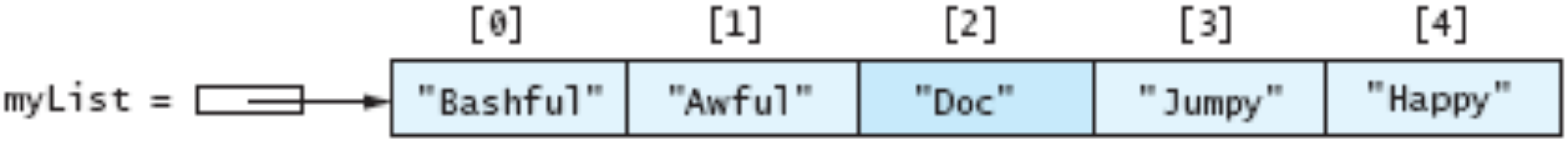
```
myList.add("Bashful");
myList.add("Awful");
myList.add("Jumpy");
myList.add("Happy");
```

ArrayList **Class** (cont.)



- Adding an element with subscript 2:

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



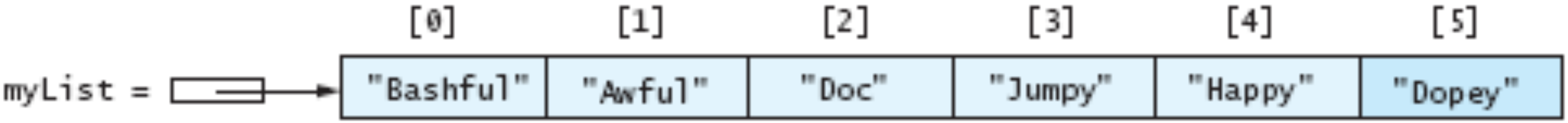
After insertion of "Doc" before the third element

- 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:

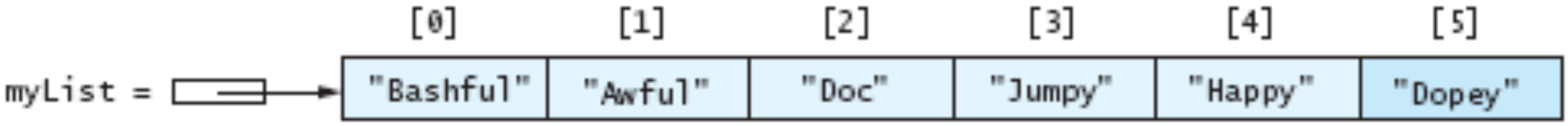
```
myList.add("Dopey");
```



After insertion of "Dopey" at the end

`ArrayList` **Class** (cont.)

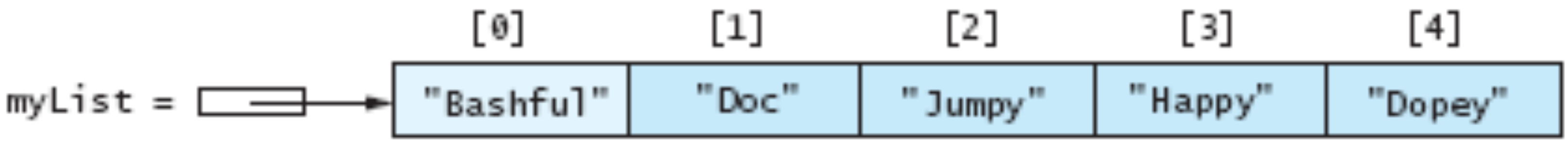
- Removing an element:



myList =

[0]	[1]	[2]	[3]	[4]	[5]
"Bashful"	"Awful"	"Doc"	"Jumpy"	"Happy"	"Dopey"

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



myList =

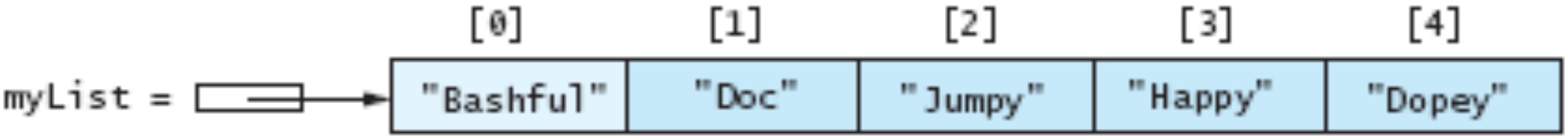
[0]	[1]	[2]	[3]	[4]
"Bashful"	"Doc"	"Jumpy"	"Happy"	"Dopey"

After removal of "Awful"

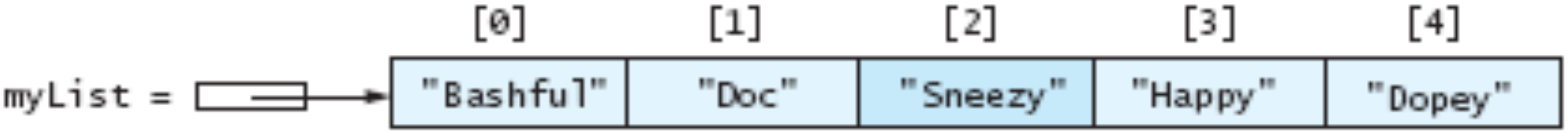
- The strings referenced by [2] to [5] have changed to [1] to [4]

ArrayList Class (cont.)

- You may also replace an element:

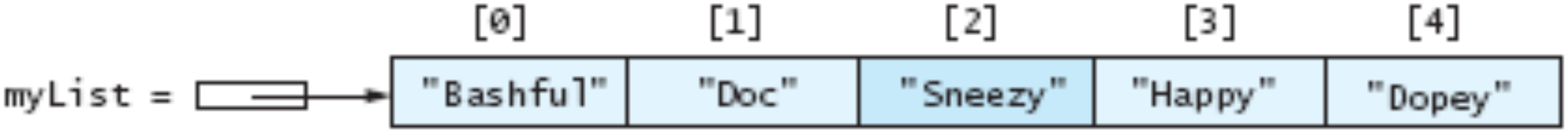


```
myList.set(2, "Sneezy");
```



After replacing "Jumpy" with"Sneezy"

ArrayList Class (cont.)

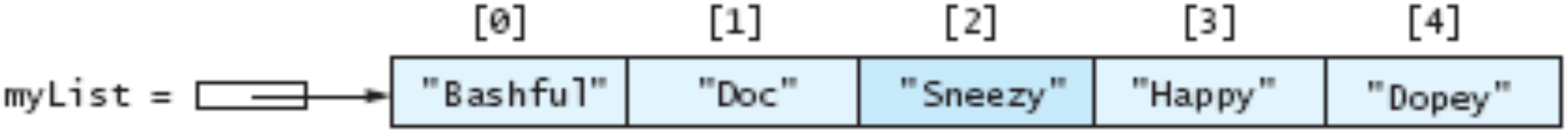


- You cannot access an element using a bracket index as you can with arrays (array[1])
- Instead, you must use the get() method:

```
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 statement

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

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

- The statement creates a `List` of `String`; only references of type `String` can be stored in the list
- `String` in this statement is called a *type parameter*
- The type parameter sets the data type of all objects stored in a collection

Generic Collections (cont.)

- The general declaration for generic collection is

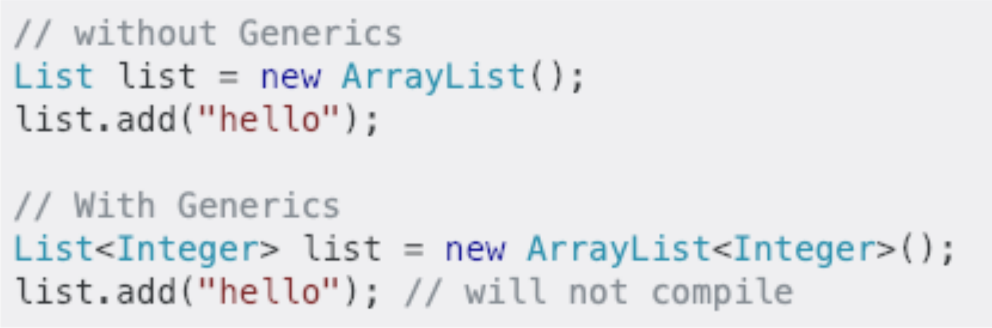
```
CollectionClassName<E> variable =
              new CollectionClassName<E>();
```

- 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<Integer>();
myList.add(new Integer(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
```

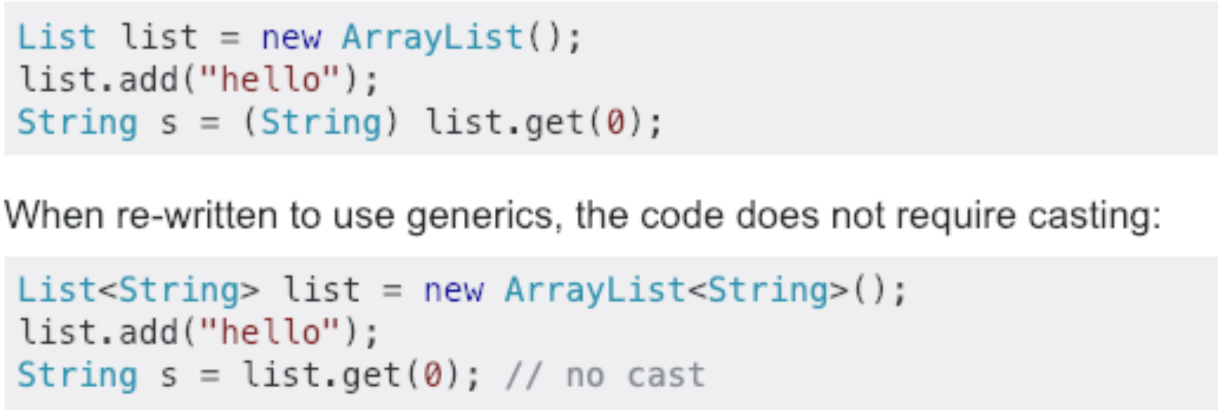
Why Use Generic Collections?

- □ Better type-checking: catch more errors, catch them earlier



```
// without Generics
List list = new ArrayList();
list.add("hello");

// With Generics
List<Integer> list = new ArrayList<Integer>();
list.add("hello"); // will not compile
```

- □ Documents intent
- □ Avoids the need to downcast from `Object`

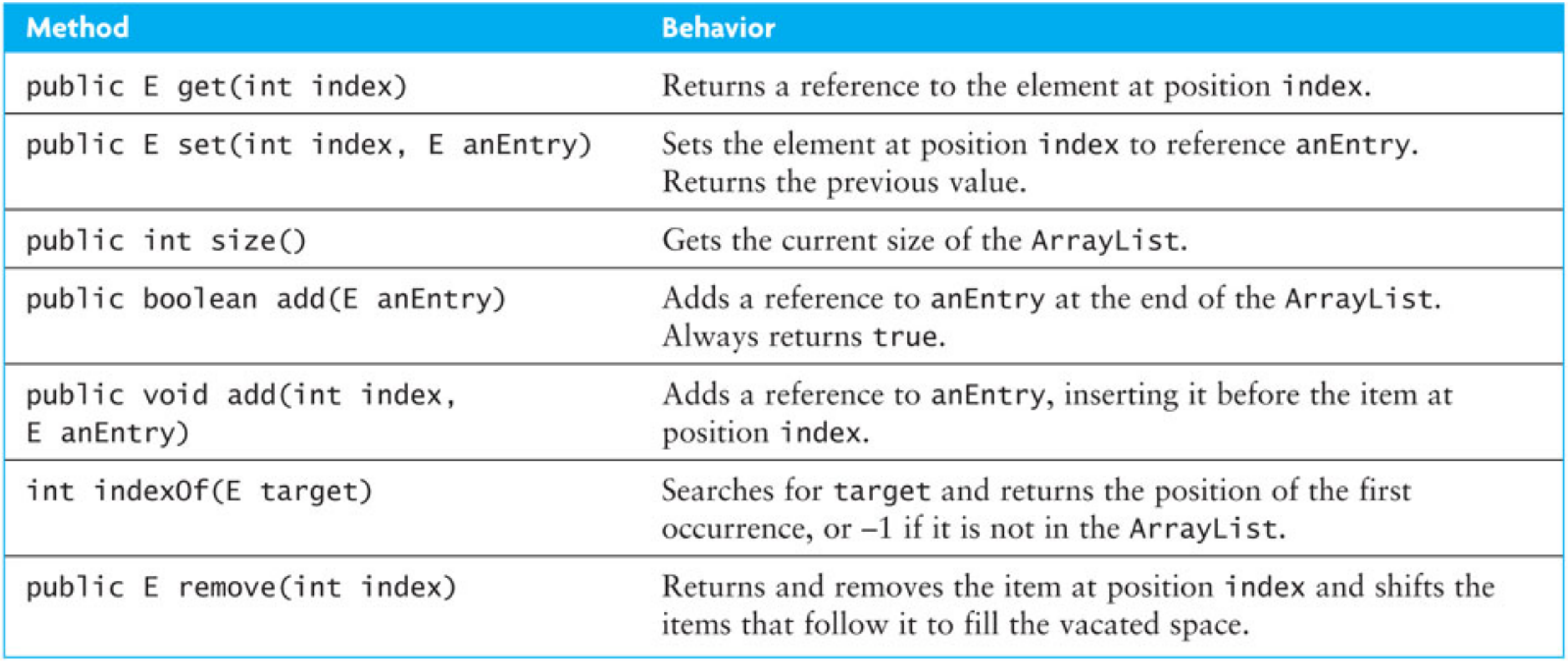


```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0); // no cast
```

Specification of the ArrayList Class



Method	Behavior
public E get(int index)	Returns a reference to the element at position index.
public E set(int index, E anEntry)	Sets the element at position index to reference anEntry. Returns the previous value.
public int size()	Gets the current size of the ArrayList.
public boolean add(E anEntry)	Adds a reference to anEntry at the end of the ArrayList. Always returns true.
public 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 ArrayList.
public E remove(int index)	Returns and removes the item at position index and shifts the items that follow it to fill the vacated space.

Applications of `ArrayList`

Section 2.3

Example Application of `ArrayList`

```
ArrayList<Integer> someInts = new ArrayList<Integer>();
int[] nums = {5, 7, 2, 15};
for (int i = 0; i < nums.length; i++) {
  someInts.add(nums[i]);
}

// Display the sum
int sum = 0;
for (int i = 0; i < someInts.size(); i++) {
  sum += someInts.get(i);
}
System.out.println("sum is " + sum);
```

Phone Directory Application

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

Create a class for objects stored in the directory

Phone Directory Application (cont.)

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


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

Create the directory

Phone Directory Application (cont.)

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


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


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

Add a `DirectoryEntry` object

Phone Directory Application (cont.)

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

```
public class Dire
  String name;
  String number;
}


private ArrayList
          new ArrayList<DirectoryEntry>();


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


int index = theDirectory.indexOf(new DirectoryEntry(aName,””));
```

Phone Directory Application (cont.)

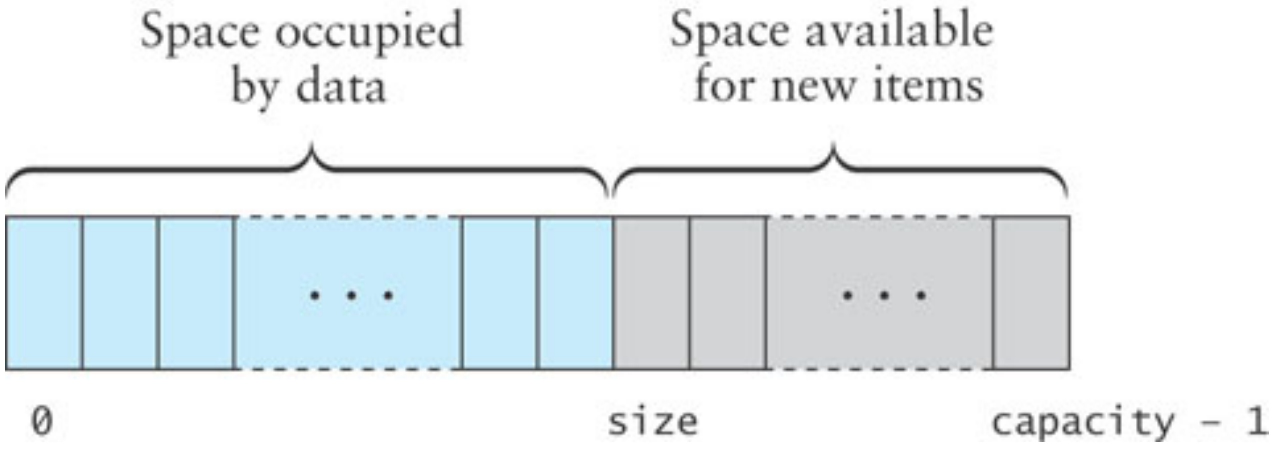
```
…

int index = theDirectory.indexOf(new
     DirectoryEntry(aName, ""));

if (index != -1)
  dE = theDirectory.get(index);
else
  dE = 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*



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[]

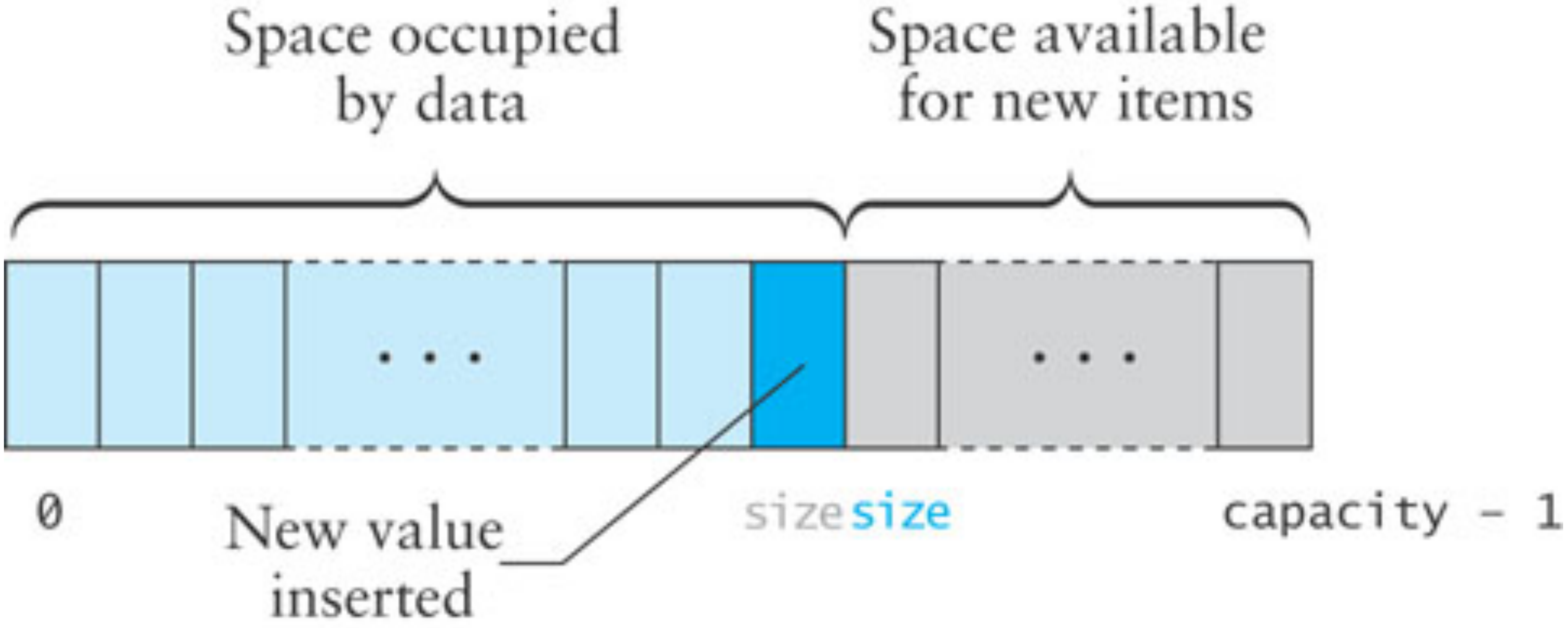
Although this may cause a compiler warning, it's ok

Implementing `ArrayList.add(E)`

- □ We will implement two add methods
- □ One will append at the end of the list
- □ The other will insert an item at a specified position

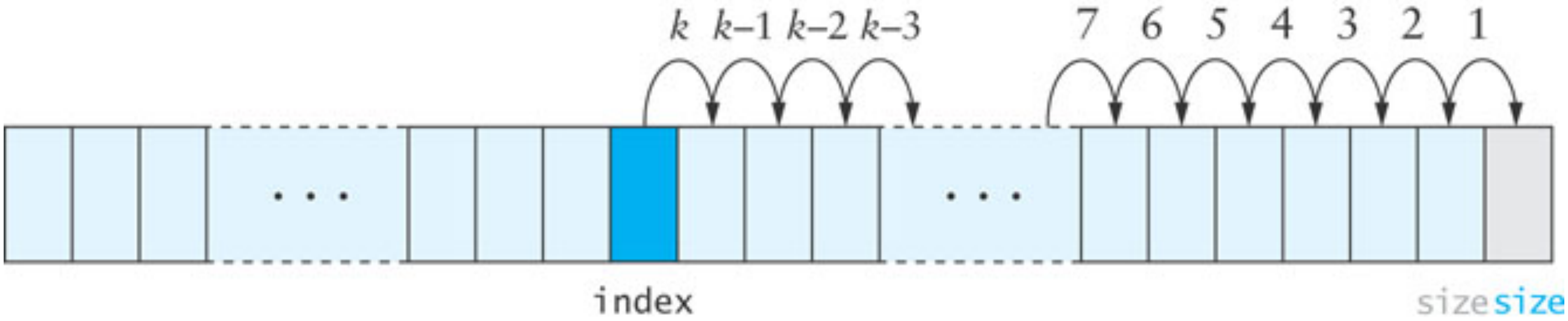
Implementing `ArrayList.add(E)` (cont.)

- 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



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 and proceeding in the indicated order



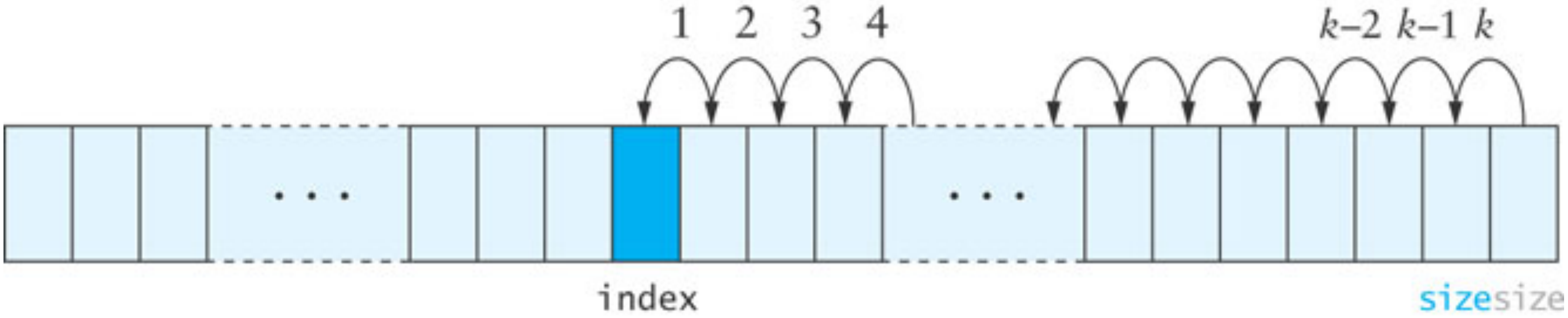
Implementing ArrayList.add(index,E)

```
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++;
}
```

`set` and `get` Methods

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

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

`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 and proceed in the indicated order

remove Method (cont.)

```
public E remove (int index) {

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

  E returnValue = theData[index];

  ?????   Fill in the blank  ???????????




  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);
}
```

`KWArrayList` as a Collection of Objects

- Earlier versions of Java did not support generics; all collections contained only `Object` elements
- To implement `KWArrayList` this way,
  - remove the parameter type `<E>` from the class heading,
  - replace each reference to data type `E` by `Object`
  - The underlying data array becomes

```
private Object[] theData;
```

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