Chapter 7

A List ADT

Goals

- Introduce the List ADT
- Implement an interface
- Have methods throw exceptions and test that the methods do

This chapter defines an interface that will represent a List abstract data type. The class that implements this interface uses an array data structure to store the elements. In the next chapter, we will see how this interface can also be implemented using the linked structure shown in the previous chapter.

Outline

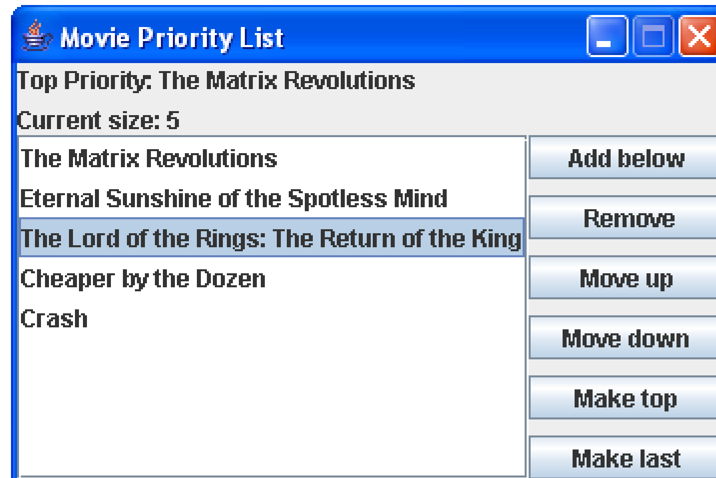
- Define an abstract data type to represent a list
- Provide a Java interface to implement list collection classes in different ways
- Discuss generic classes that are capable of storing elements of any reference type
- Implement a list using an array data structure

7.1 Lists

A **list** is a collection where each element has a specific position—each element has a distinct predecessor (except the first) and a distinct successor (except the last). A list allows access to elements through an index. The list interface presented here supports operations such as the following:

- add, get, or remove an element at specific location in the list
- find or remove an element with a particular characteristic

From an *application* point of view, a list may store a collection of elements where the index has some importance. For example, the following interface shows one view of a list that stores a collection of DVDs to order. The DVD at index 0, “The Matrix Revolutions”, has the top priority. The DVD at index 4 has a lower priority than the DVD at index 3. By moving any “to do” item up or down in the list, users reprioritize what movies to get next. Users are able to add and remove DVDs or rearrange priorities.



From an *implementation* point of view, your applications could simply use an existing Java collection class such as `ArrayList<E>` or `LinkedList<E>`. As is customary in a second level course in computer science, we will be implementing our own, simpler version, which will

- enhance your ability to use arrays and linked structures (required in further study of computing).
- provide an opportunity to further develop programming skills: coding, testing, and debugging.
- help you understand how existing collection classes work, so you can better choose which one to use in your programs.

Specifying ADTs as Java Interfaces

To show the inner workings of a collection class (first with an array data structure, and then later with a linked structure), we will have the same interface implemented by two different classes. This interface, shown below, represents one abstract view of a list that was designed to support the goals mentioned above.

The interface specifies that implementing classes must be able to store any type of element through Java generics—`List<E>`, rather than `List`. One alternative to this design decision is to write a List class each time you need a new list of a different type (which could be multiple classes that are almost the same). You could implement a class for each type of the following objects:

```
// Why implement a new class for each type?
StringList stringList = new StringList();
BankAccountList bankAccountList = new BankAccountList();
DateList dateList = new DateList();
```

Another alternative was shown with the `GenericList` class shown in the previous chapter. The method heading that adds an element would use an `Object` parameter and the `get` method to return an element would have an `Object` return type.

```
// Add any reference type of element (no primitive types)
public void add(Object element);

// Get any reference type of element (no primitive types)
public Object get(int index);
```

Collections of this type require the extra effort of casting when getting an element. If you wanted a collectin of primitives, you would have to wrap them. Additionally, these types of collections allow you to add any mix of types. The output below also shows that runtime errors can occur because any reference type can be added as an element. The compiler approves this code, but an exception is thrown at runtime.

```
GenericList list = new GenericList();
list.add("Jody");
list.add(new BankAccount("Kim", 100));

for (int i = 0; i < list.size(); i++) {
  String element = (String) list.get(i); // cast required
  System.out.println(element.toUpperCase());
}
```

Output:

```
JODY
Exception in thread "main" java.lang.ClassCastException: BankAccount
```

The preferred option is to focus on classes that have a type parameter in the heading like this

```
public class OurList<E>  // E is a type parameter
```

Now `<E>` represents the type of elements to be stored in the collection. Generic classes provide the same services as the raw type equivalent with the following advantages:

- requires less casting.
- can store collections of any type, including primitives (at least give the appearance of).
- generates the errors at compile time, when they are much easier to deal with.
- this approach is used in the new version of Java's collection framework

Generic collections need a type argument at construction to let the compiler know which type E represents. When an `OurList` object is constructed with a `<String>` type argument, every occurrence of `E` in the class will be seen as `String`.

```
  // Add a type parameter such as <E> and implement only one class
  OurList<String> sl = new OurArrayList<String>();
  OurList<BankAccount> bl = new OurArrayList<BankAccount>();
  OurList<Integer> dl = new OurArrayList<Integer>();
```

Now an attempt to add a `BankAccount` to a list constructed to only store strings

```
  sl.add(0, new BankAccount("Jody", 100));
```

results in this compiletime error:

The method add(int, String) in the type OurList<String> is not applicable for the arguments (int, BankAccount)

A List ADT Specified as a Java interface

Interface `OurList` specifies a reduced version of Java's `List` interface (7 methods instead of 25). By design, these methods match the methods of the same name found in the two Java classes that implement Java's `List` interface: `ArrayList<E>` and `LinkedList<E>`.

```
/**
 * This interface specifies the methods for a generic List ADT.
 * It is designed to be generic so any type element can be stored.
 * It will be implemented with an array in this chapter and then a
 * linked structure in the chapter that follows. These 7 methods
 * are a subset of the 25 methods specified in java.util.List<E>
 */
 public interface OurList<E>  {

  // Return the number of elements currently in the list
  public int size();

  // Insert an element at the specified location
  // Precondition: insertIndex >= 0 and insertIndex <= size()
  public void add(int insertIndex, E element) throws IllegalArgumentException;

  // Get the element stored at a specific index
  // Precondition: insertIndex >= 0 and insertIndex < size()
  public E get(int getIndex) throws IllegalArgumentException;

  // Replace the element at a specific index with element
  // Precondition: insertIndex >= 0 and insertIndex < size()
  public void set(int insertIndex, E element) throws IllegalArgumentException;

  // Return a reference to element in the list or null if not found.
  public E find(E search);

  // Remove element specified by removalIndex if it is in range
  // Precondition: insertIndex >= 0 and insertIndex < size()
  public void removeElementAt(int removalIndex) throws IllegalArgumentException;

  // Remove the first occurrence of element and return true or if the
  // element is not found leave the list unchanged and return false
  public boolean remove(E element);
}
```

OurArrayList<E> implements OurList<E>

The following class implements `OurList` using an array as the structure to store elements. The constructor ensures the array has the capacity to store 10 elements. (The capacity can change). Since `n` is initially set to 0, the list is initially empty.

```
public class OurArrayList<E> implements OurList<E> {

 /**
   * A class constant to set the capacity of the array.
   * The storage capacity increases if needed during an add.
   */
  public static final int INIITIAL_CAPACITY = 10;
```

```
 /**
   * A class constant to help control thrashing about when adding and
   * removing elements when the capacity must increase or decrease.
   */
  public static final int GROW_SHRINK_INCREMENT = 20;

  // --Instance variables
  private Object[] data; // Use an array data structure to store elements
  private int n;         // The number of elements (not the capacity)

  /**
   * Construct an empty list with an initial default capacity.
   * This capacity may grow and shrink during add and remove.
   */
  public OurArrayList() {
    data = new Object[INIITIAL_CAPACITY];
    n = 0;
  }
```

Whenever you are making a generic collection, the type parameter (such as <E>) does not appear in the constructor. Since the compiler does not know what the array element type will be in the future, it is declared to be an array of Objects so it can store any reference type.

The initial capacity of `OurList` object was selected as 10, since this is the same as Java's `ArrayList<E>`. This class does not currently have additional constructors to start with a larger capacity, or a different grow and shrink increment, as does Java's `ArrayList.` (Enhancing this class in this manner is left as an exercise).

size

The `size` method returns the number of elements in the list which, when empty, is zero.

```
public void testSizeWhenEmpty() {
  OurList<String> emptyList = new OurArrayList<String>();
  assertEquals(0, emptyList.size());
}
```

Because returning an integer does not depend on the number of elements in the collection, the `size` method executes in constant time.

```
/**
  * Accessing method to determine how many elements are in this list.
  * Runtime: O(1)
  * @returns the number of elements in this list.
  */
  public int size() {
    return n;
  }
```

get

`OurList` specifies a `get` method that emulates the array square bracket notation `[]` for getting a reference to a specific index. This implementation of the get method will throw an `IllegalArgumentException` if argument `index` is outside the range of `0` through `size()-1`. Although not specified in the interface, this design decision will cause the correct exception to be

thrown in the correct place, even if the index is in the capacity bounds of the array. This avoids returning null or other meaningless data during a "get" when the index is in the range of 0 through `data.length-1` inclusive.

```
/**
  * Return a reference to the element at the given index.
  * This method acts like an array with [] except an exception
  * is thrown if index >= size().
  * Runtime: O(1)
  * @returns Reference to object at index if 0 <= index < size().
  * @throws IllegalArgumentException when index<0 or index>=size().
  */
  public E get(int index) throws IllegalArgumentException {
    if (index < 0 || index >= size())
      throw new IllegalArgumentException("" + index);

    return data[index];
  }
```

Exception Handling

When programs run, errors occur. Perhaps an arithmetic expression results in division by zero, or an array subscript is out of bounds, or there is an attempt to read a file from a floppy disk, but there is no disk in the floppy disk drive, or perhaps a file with a specific name simply does not exist. Or perhaps, the `get` method receives an argument 5 when the size was 5. These types of errors that occur while a program is running are known as exceptions.

The `get` method throws an exception to let the programmer using the method know that an invalid argument was passed during a message. At that point, the program terminates indicating the file name, the method name, and the line number where the exception was thrown. When `size` is 5 and the argument 5 is passed, the `get` method throws the exception and Java prints this information:

```
java.lang.IllegalArgumentException: 5
at OurArrayListTest.get(OurArrayListTest.java:108)
```

Programmers have at least two options for dealing with these types of errors:

- Ignore the exception and let the program terminate
- Handle the exception

Java allows you to *try* to execute methods that may throw an exception. The code exists in a **`try`** block—the keyword `try` followed by the code wrapped in a block, { }.

```
try {
    code that may throw an exception when an exception is thrown
}
catch(Exception anException) {
    code that executes only if an exception is thrown from code in the above try block.
}
```

A `try` block must be followed by a `catch` block—the keyword `catch` followed by the anticipated exception as a parameter and code wrapped in a block. The `catch` block contains the code that executes when the code in the `try` block causes an exception to be thrown (or called a method that throws an exception).

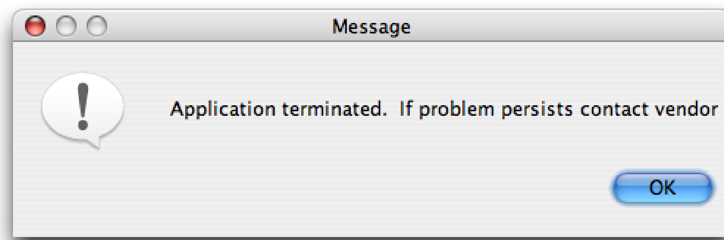
Because all exception classes extend the `Exception` class, the type of exception in as the parameter to catch could always be `Exception`. In this case, the `catch` block would catch any type of exception that can be thrown. However, it is recommended that you use the specific exception that is expected to be thrown by the code in the `try` block, such as `IllegalArgumentException`.

The following example will always throw an exception since the list is empty. Any input by the user for `index` will cause the get method to throw an exception.

```
Scanner keyboard = new Scanner(System.in);
OurArrayList<String> list = new OurArrayList<String>();
int index = keyboard.nextInt();

try {
  String str = list.get(index); // When size==0, get always throws an exception
}
catch (IllegalArgumentException iobe) {
  JOptionPane.showMessageDialog(null, "Application terminated. "
                          + " If problem persists contact vendor");
}
```

Output



If the size were greater than 0, the user input may or may not cause an exception to be thrown.

To successfully handle exceptions, a programmer must know if a method might throw an exception, and if so, the type of exception. This is done through documentation in the method heading.

```
public E get(int index) throws IllegalArgumentException {
```

A programmer has the option to put a call to get in a `try` block or the programmer may call the method without placing it in a `try` block. The option comes from the fact that `IllegalArgumentException` is a `RuntimeException` that needs not be handled. Exceptions that don't need to be handled are called unchecked exceptions. The unchecked exception classes are those that extend `RuntimeException`, plus any `Exception` that you write that also extends `RuntimeException`. The unchecked exceptions include the following types (this is not a complete list):

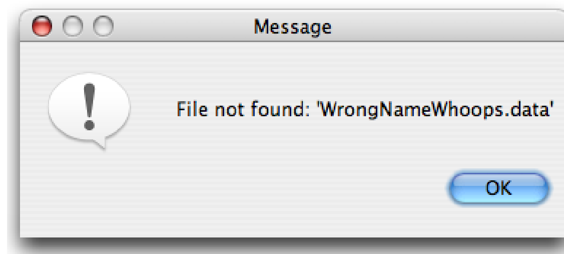
- `ArithmeticException`
- `ClassCastException`
- `IllegalArgumentException`
- `IllegalArgumentException`
- `NullPointerException`

Other types of exceptions require that the programmer handle them. These are called checked exceptions. There are many checked exceptions when dealing with file input/output and networking

that *must* be surrounded by a try catch. For example when using the `Scanner` class to read input from a file, the constructor needs a `java.io.File` object. Because that constructor can throw a `FileNotFoundException`, the Scanner must be constructed in a try bock.

```
Scanner keyboard = new Scanner(System.in);
String fileName = keyboard.nextLine();
Scanner inputFile = null;
try {
  // Throws exception if file with the input name can not be found
  inputFile = new Scanner(new File(filename));
}
catch (FileNotFoundException fnfe) {
  JOptionPane.showMessageDialog(null, "File not found: '" +
                                       fileName + "'");
}
```

Output assuming the user entered *WrongNameWhoops.data* and that file name does not exist.



Self-Check

7-1 Which of the following code fragments throws an exception?

-a `int j = 7 / 0;`

-b
```
String[] names = new String[5];
names[0] = "Austen";
System.out.println(names[1].toUpperCase());
```

-c
```
String[] names;
names[0] = "Austen";
```

7.2 Write a method that reads and prints all the lines in the file.

Testing that the Method throws the Exception

The `get` method is supposed to throw an exception when the index is out of bounds. To make sure this is happening, the following test method will `fail` if the `get` method does *not* throw an exception when it is expected:

```
@Test
public void testEasyGetException() {
   OurArrayList<String> list = new OurArrayList<String>();
   try {
     list.get(0); // We want get to throw an exception . . .
     fail();      // Show the red bar only if get did NOT throw the exception
   }
   catch (IllegalArgumentException iobe) {
     // . . . and then skip fail() to execute this empty catch block
   }
 }
```

This rather elaborate way of testing—to make sure a method throws an exception without shutting down the program—depends on the fact that the empty catch block will execute rather than the `fail()` method. The `fail` method of class `TestCase` automatically generates a failed assertion. The assertion will fail only when your method does not throw an exception at the correct time.

JUnit 4.0 proves an easier technique to ensure a method throws an exception. The `@Test` annotation takes a parameter, which can be the type of the Exception that the code in the test method should throw. The following test method will `fail` if the `get` method does *not* throw an exception when it is expected:

```
 @Test(expected = IllegalArgumentException.class)
 public void testEasyGetException() {
   OurArrayList<String> list = new OurArrayList<String>();

   // We want get to ensure this does throws an exception.
   list.get(0);
 }
```

We will use this shorter technique.

add(int, E)

An element of any type can be inserted into any index as long as it in the range of 0 through `size()` inclusive. Any element added at 0 is the same as adding it as the first element in the list.

```
@Test
public void testAddAndGet() {
  OurList<String> list = new OurArrayList<String>();
  list.add(0, "First");
  list.add(1, "Second");
  list.add(0, "New first");
  assertEquals(3, list.size());
  assertEquals("New first", list.get(0));
  assertEquals("First", list.get(1));
  assertEquals("Second", list.get(2));
}

@Test(expected = IllegalArgumentException.class)
public void testAddThrowsException() {
  OurArrayList<String> list = new OurArrayList<String>();
    list.add(1, "Must start with 0");
}
```

The `add` method first checks to ensure the parameter `insertIndex` is in the correct range. If it is out of range, the method throws an exception.

```
/**
 * Place element at insertIndex.
 * Runtime: O(n)
 *
 * @param element The new element to be added to this list
 * @param insertIndex The location to place the new element.
 * @throws IllegalArgumentException if insertIndex is out of range.
 */
public void add(int insertIndex, E element) throws IllegalArgumentException {
```

```
  // Throw exception if insertIndex is not in range
  if (insertIndex < 0 || insertIndex > size())
    throw new IllegalArgumentException("" + insertIndex);

  // Increase the array capacity if necessary
  if (size() == data.length)
    growArray();

  // Slide all elements right to make a hole at insertIndex
  for (int index = size(); index > insertIndex; index--)
    data[index] = data[index - 1];

  // Insert element into the "hole" and increment n.
  data[insertIndex] = element;
  n++;
}
```

If the index is in range, the method checks if the array is full. If so, it calls the private helper method `growArray` (shown later) to increase the array capacity. A `for` loop then slides the array elements one index to the right to make a "hole" for the new element. Finally, the reference to the new element gets inserted into the array and n (size) increases by +1. Here is a picture of the array after five elements are added with the following code:

```
OurList<String> list = new OurArrayList<String>();
list.add(0, "A");
list.add(1, "B");
list.add(2, "C");
list.add(3, "D");
list.add(4, "E");
```

n: 5
data:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
"A"	"B"	"C"	"D"	"E"	null	null	null	null	null

At this point, an `add(0, "F")` would cause all array elements to slide to the right by one index. This leaves a "hole" at index 0, which is actually an unnecessary reference to the `String` object `"A"` in `data[0]`. This is okay, since this is precisely where the new element `"F"` should be placed.

n: 5
data:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
"A"	**"A"**	**"B"**	**"C"**	**"D"**	**"E"**	null	null	null	null

After storing a reference to `"F"` in `data[0]` and increasing `n`, the instance variables should look like this:

n: **6**
data:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
"F"	"A"	"B"	"C"	"D"	"E"	null	null	null	null

An `add` operation may require that every reference slide to the bordering array location. If there are 1,000 elements, the loop executes 1,000 assignments. Assuming that an insertion is as likely to be at

index 1 as at index n-1 or any other index, the loop will likely average n/2 assignments to make a "hole" for the new element. With the possibility of `growArray` executing O(n), `add`, for all other cases, f(n) = n/2 + n or 1.5n. After dropping the coefficient 1.5, the runtime of `add` would still be O(n). The tightest upper bound is still O(n) even if `growArray` is called, since it too is O(n).

`growArray()`, a private helper method

If there is not enough room to insert the element, the array capacity will increase by sending a `growArray` message before the insertion. `OurArrayList` allocates another `GROW_SHRINK_INCREMENT` number of array locations when an `add` message discovers that there is no more room to add a new element. The `growArray` method changes the instance variable `data` to reference a new array a larger capacity by calling `growArray`. It leaves the original contents (`x[0]` through `x[n-1]`) intact, with the elements in the same indexes as before. With an array implementation, the object should grow and shrink the array at the appropriate times. If the list frequently changes in size, the `add` and `remove` methods can ensure that not too much memory is wasted at one time.

```
// Make the array have greater capacity to store more elements.
// The original elements are retained in indexes 0..size()-1.
private void growArray() {
  Object[] temp = new Object[data.length + GROW_SHRINK_INCREMENT];
  for (int j = 0; j < size(); j++) {
    temp[j] = data[j];
  }
  data = temp;
  // Reference to temp disappears--that memory will be garbage collected
}
```

When the array `data` is filled to capacity, the instance variables look like this:

```
n: 10
data (data.length == 10):
```

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
"F"	"A"	"B"	"C"	"D"	"E"	"G"	"H"	"I"	"J"

During the message `add(10, "Z");` the `add` method sends a `growArray message` to itself to make the instance variables look like this:

```
n: 11
data (data.length == 30):
```

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	**[10]**	[11]	[12]	…	…	[28]	**[29]**
"F"	"A"	"B"	"C"	"D"	"E"	"G"	"H"	"I"	"J"	"Z"	null	null	…	…	null	**null**

`toString()`

Although not specified in interface `OurList`, each Java class should implement a `toString` method to provide an easy way to visualize the state of its objects. This `toString` method overrides the Object `toString` method. The output will look just like the output from Java's collection classes. Square brackets surround all elements that are separated by commas.

Although it is not necessary to test `toString` methods, the following test shows the desired return result from `toString`.

```
@Test
public void testToString() {
  OurList<String> list = new OurArrayList<String>();
  assertEquals("[]", list.toString());

  list.add(0, "A");
  assertEquals("[A]", list.toString());

  list.add(1, "B");
  assertEquals("[A, B]", list.toString());

  list.add(0, "1st");
  assertEquals("[1st, A, B]", list.toString());
}
```

Since the `toString` method concatenates all n elements in the list, the runtime of this method grows with n, and so is O(n).

```
/**
  * Return a string with all elements in this list.
  * Runtime: O(n)
  * @returns A String that is the concatenation of the toString version
  * of all elements separated by ", " and bracketed with "[" and "]".
  */
  @Override
  public String toString() {
    String result = "[";
    // Concatenate all but the last
    for (int index = 0; index < size() - 1 ; index++) {
      result = result + data[index].toString() + ", ";
    }
    if (size() > 0) { // Avoid placing , after the last element.
      result += data[size()-1];
    }
    result += "]";   // Always concatenate the closing square bracket
    return result;
  }
```

set(int, E)

`OurList` specifies the `set` method to emulate the array square bracket notation `[]` for setting an element at a specific index. Like the `get` method, `set` will also throw an `IllegalArgumentException` if the `index` is outside the range of `0` through `size()-1`; and for the same reasons. Since the size of the list does not affect the runtime of these two algorithms, `get` and `set` are O(1).

```
 /** Change the element referenced by index in this list.
   * @param index Location where a newElement replaces existing one.
   * @param newElement A reference to the object to be inserted.
   * @throws IllegalArgumentException if index < 0 or index >= size().
   */
  public void set(int index, E newElement) throws IllegalArgumentException {
    if (index < 0 || index >= size())
      throw new IllegalArgumentException("" + index);
    else
      data[index] = newElement;
  }
```

The following test method will generate a failure if the `set` method does not throw an exception when expected:

```
@Test(expected = IllegalArgumentException.class)
public void testThrowExceptionsOnAnEmptyList() {
  OurList<String> list = new OurArrayList<String>();
  list.set(0, "Cannot change any element in an empty list");
}
```

Summarize the Behavior of `OurArrayList` so Far

The following test method demonstrates the methods just described. It must be assumed that the other methods of interface `OurList` have already been implemented at least as stubs (correct headings, but only returns) because `OurArrayList` will not compile until it implements all seven method headings of interface `OurList`.

```
@Test
public void testOurArrayListAddsGetsSetsAndToString() {
  OurList<String> list = new OurArrayList<String>();
  list.add(0, "A");              // [A]
  list.add(1, "B");              // [A, B]
  assertEquals("[A, B]", list.toString());

  // Insert elements at indexes 0, 3, and list.size()
  list.add(0, "1st");             // [1st, A, B]
  list.add(list.size(), "4th");  // [1st, A, B, 4th]
  assertEquals("[1st, A, B, 4th]", list.toString());

  // Change two elements with set
  list.set(1, "2nd");
  list.set(2, "3rd");
  assertEquals("[1st, 2nd, 3rd, 4th]", list.toString());

  // Check every element to ensure get returns the correct value
  assertEquals("1st", list.get(0));
  assertEquals("2nd", list.get(1));
  assertEquals("3rd", list.get(2));
  assertEquals("4th", list.get(3));
}
```

Self-Check

7-3 What is the runtime of the test method above, assuming `assertEquals` is O(1)?

7-4 Which `assertEquals` will pass: a, b, or c?

```
OurArrayList<String> list = new OurArrayList<String>();
list.add(0, "A");
list.add(0, "B");
list.add(0, "C");
assertEquals("B", list.get(0));  // a.
assertEquals("B", list.get(1));  // b.
assertEquals("B", list.get(2));  // c.
```

find

The `find` method returns a reference to the first element in the list that matches the argument. It uses the `equals` method that is defined for that type. The `find` method returns `null` if the argument does not match any list element, again using the `equals` method for the type of elements being stored. Any class of objects that you store should override the `equals` method such that the state of the objects are compared rather than references.

Searching for a `String` in a list of strings is easy, since the `String` equals method does compare the state of the object. You can simply ask to get a reference to a `String` by supplying the string you seek as an argument.

```
@Test
public void testFindWithStrings() {
  OurList<String> list = new OurArrayList<String>();
  list.add(0, "zero");
  list.add(1, "one");
  list.add(2, "two");
  assertNotNull(list.find("zero"));
  assertNotNull(list.find("one"));
  assertNotNull(list.find("two"));
}
```

A test should also exist to make sure `null` is returned when the string does not exist in the list

```
@Test
public void testFindWhenNotHere() {
  OurList<String> names = new OurArrayList<String>();
  names.add(0, "Jody");
  names.add(1, "Devon");
  names.add(2, "Nar");
  assertNull(names.find("Not Here"));
}
```

However, for most other types, searching through an `OurArrayList` object (or an `ArrayList` or `LinkedList` object) requires the creation of a faked temporary object that "`equals`" the object that you wish to query or whose state you wish to modify. Consider the following test that establishes a small list for demonstration purposes. Using a small list of BankAccounts, the following code shows a deposit of 100.00 made to one of the accounts.

```
@Test
public void testDepositInList() {
  OurList<BankAccount> accountList = new OurArrayList<BankAccount>();
  accountList.add(0, new BankAccount("Joe", 0.00));
  accountList.add(1, new BankAccount("Ali", 1.00));
  accountList.add(2, new BankAccount("Sandeep", 2.00));

  String searchID = "Ali";
  BankAccount searchAccount = new BankAccount(searchID, -999);
  BankAccount ref = accountList.find(searchAccount);
  assertNotNull(ref);
  ref.deposit(100.00);
  // Make sure the correct element was really changed
  ref = accountList.find(searchAccount);
  assertEquals(101.00, ref.getBalance(), 1e-12);
}
```

The code constructs a "fake" reference (`searchAccount`) to be compared to elements in the list. This temporary instance of `BankAcccount` exists solely for aiding the search process. To make an appropriate temporary search object, the programmer must know how the `equals` method returns true for this type when the IDs match exactly. (You may need to consult the documentation for `equals` methods of other type.) The temporary search account need only have the ID of the searched-for object—`equals` ignores the balance. They do not need to match the other parts of the real object's state.[1] The constructor uses an initial balance of -999 to emphasize that the other parameter will not be used in the search.

The `find` method uses the sequential search algorithm to search the unsorted elements in the array structure. Therefore it runs O(n).

```
/**
 * Return a reference to target if it "equals" an element, or null if
 * it is not found. Since you will rarely be requesting a reference to
 * an object that you already have a reference to, this method will
 * often be called with a "fake" object that has the correct state to
 * equal the real object being sought. For example
 *
 *    list.find(new JukeboxAccount("SearchID", -999));
 *
 * returns a reference to the first element in this list that has the
 * ID of "SearchID". The other argument -999 is not used in the
 * equals method for the type (BankAccount in this example)
 *
 * Runtime: O(n)
 *
 * @param target The object that will be compared to list elements.
 * @returns Reference to first object that equals target (or null).
 */
public E find(E target) {
  // Get index of first element that equals target
  for (int index = 0; index < n; index++)  {
    if (target.equals(data[index]))
      return data[index];
  }
  return null; // Did not find target in this list
}
```

The following test method builds a list of two `BankAccount` objects and asserts that both can be successfully found.

```
@Test
public void testFindWithBankAccounts() {
  // Set up a small list of BankAccounts
  OurList<BankAccount> list = new OurArrayList<BankAccount>();
  list.add(0, new BankAccount("zero", 0.00));
  list.add(1, new BankAccount("one", 1.00));

  // Find one
```

[1] This is typical when searching through indexed collections. However, there are better ways to do the same thing. Other collections presented later will map a key to a value. All the programmer needs to worry about is the key, such as an account number or student ID. There is no need to construct a temporary object or worry about how a particular `equals` method works for many different classes of objects.

```
    BankAccount fakedToFind = new BankAccount("zero", -999);
    BankAccount withTheRealBalance = list.find(fakedToFind);

    // The following assertions expect a reference to the real account
    assertNotNull(withTheRealBalance);
    assertEquals("zero", withTheRealBalance.getID());
    assertEquals(0.00, withTheRealBalance.getBalance(), 1e-12);

    // Find the the other
    fakedToFind = new BankAccount("one", +234321.99);
    withTheRealBalance = list.find(fakedToFind);

    // The following assertions expect a reference to the real account
    assertNotNull(withTheRealBalance);
    assertEquals("one", withTheRealBalance.getID());
    assertEquals(1.00, withTheRealBalance.getBalance(), 1e-12);
  }
```

And of course we should make sure the `find` method returns null when the object does not "equals" any element in the list:

```
  @Test
  public void testFindWhenElementIsNotThere() {
    OurList<BankAccount> list = new OurArrayList<BankAccount>();
    list.add(0, new BankAccount("zero", 0.00));
    list.add(1, new BankAccount("one", 1.00));
    list.add(2, new BankAccount("two", 2.00));
    BankAccount fakedToFind = new BankAccount("Not Here", 0.00);
    // The following assertions expect a reference to the real account
    assertNull(list.find(fakedToFind));
  }
```

removeElementAt(int) and **remove(E)**

The `removeElementAt` method will first check to make sure the removal `index` is in range. Instead of sliding elements to the right to make a hole as in `add`, `removeElementAt` will slide all successive elements one index to the left. The first loop iteration destroys the reference to the element to be removed. The loop continues until the last element has been moved to the old `size()-1`. Then `n` decreases by 1. Here is a picture of an array based list object when it has six elements (`"F"` is first, and `"E"` is last):

n: 6
data:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
"F"	"A"	"B"	"C"	"D"	"E"	null	null	null	null

The message `removeElementAt(2)` ("B") would shift all elements from index 2 through size-1 one location to the left. In this case, there are now two references to a `String` object with the value "E".

n: 5
data:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
"F"	"A"	~~B~~"C"	"D"	"E"	"E"	null	null	null	null

The unnecessary reference to "E" in `data[5]` could be set to `null`, but it is not necessary. Since there still is a reference to the String object "E", the object would remain anyway. The "extra" reference to "E" (in `data[5]` above) would effectively be removed from the list simply by decreasing `n` by 1.

The remove method could now use the tested `removeElement`. Here is one algorithm for `remove` that will be called with a message such as `list.remove("B")`.

Algorithm to remove an element that "equals" target

```
index = indexOf(target)
if (index  >=  0)
  removeElementAt(index)
```

In this algorithm, the remove method first asks the existing `indexOf` method for the index of the object to be removed. If the object is not found, the collection remains unchanged. If the element is in the list, a `removeElementAt` message removes the element from that index in the list.

Whereas the `add` method of `OurArrayList` increased the capacity of the array instance variable `data`, a similar method should decrease the capacity when appropriate.

Self-Check

7-5 What is the tightest big-O runtime of the `removeElementAt` algorithm?

7-6 What is the tightest big-O runtime of the `remove` operation?

7-7 What would be the tightest big-O runtime of a `removeAll` algorithm that removes the element in index 0 as long as there is an element? Here is the code that accomplishes this:

```
  public void removeAll() {
    while(! list.isEmpty())
      list.removeElementAt(0);
}
```

7-8 What would be the runtime of a new `removeAll` method that effectively removes all elements like this:

```
public void removeAll() {
  data = new Object[INIITIAL_CAPACITY];
  n = 0;
}
```

7-9 Which algorithm would be preferred if you assume someone will use `ArrayList` objects with tens of thousands of elements: the algorithm that is O(n) or the algorithm that is $O(n^2)$?

7-10 Write the removeElementAt method as if it were in class OurArrayList. Make sure the method throws an IllegalArgumentException if the index is not in the correct range.

```
/**
  *  Remove element at the given index leaving
  *  the other elements in the same order.
  */
 public void removeElementAt(int index) {
```

7-11 Write the `remove` method as if it were in `class OurArrayList`. Do not bother to shrink the array.

```
// Remove the first occurrence of element leaving the other elements in the same
// order and return true. If element is not found, return false.
public boolean remove(E element) {
```