Convenience methods

- Implement the following methods:
  - `indexOf` - returns the first index an element is found, or -1 if not
  - `isEmpty` - returns true if list has no elements
  - `contains` - returns true if the list contains the given int value

- Why do we need `isEmpty` and `contains` when we already have `indexOf` and `size`?
  - These methods provide convenience to the client of our class.

```java
if (myList.size() == 0) {
  if (myList.isEmpty()) {
    if (myList.indexOf(42) >= 0) {
      if (myList.contains(42)) {
```
Let's add some new features to our `ArrayIntList` class:

1. A method that allows client programs to print a list's elements
2. A constructor that accepts an initial capacity

(By writing these we will recall some features of objects in Java.)

Printing lists: You may be tempted to write a `print` method:

```java
// client code
ArrayIntList list = new ArrayIntList();
...
list.print();
```

- Why is this a bad idea? What would be better?
The `toString` method

- Tells Java how to convert an object into a `String`
  
  ```java
  ArrayIntList list = new ArrayIntList();
  System.out.println("list is " + list);
  // ("list is " + list.toString());
  ```

- Syntax:
  ```java
  public String toString() {
      code that returns a suitable String;
  }
  ```

- Every class has a `toString`, even if it isn't in your code.
  - The default is the class's name and a hex (base-16) number:
    ```java
    ArrayIntList@9e8c34
    ```
// Returns a String representation of the list.
public String toString() {
    if (size == 0) {
        return "[]";
    } else {
        String result = "[" + elementData[0];
        for (int i = 1; i < size; i++) {
            result += ", " + elementData[i];
        }
        result += "]";
    return result;
    }
}
Multiple constructors

• existing constructor:
  ```java
  public ArrayIntList() {
      elementData = new int[10];
      size = 0;
  }
  ```

• Add a new constructor that accepts a capacity parameter:
  ```java
  public ArrayIntList(int capacity) {
      elementData = new int[capacity];
      size = 0;
  }
  ```

  – The constructors are very similar. Can we avoid redundancy?
**this keyword**

- **this**: A reference to the *implicit parameter* (the object on which a method/constructor is called)

**Syntax:**

- To refer to a field: `this.field`
- To call a method: `this.method(parameters);`
- To call a constructor from another constructor: `this(parameters);`
Revised constructors

class ArrayIntList {  
    public ArrayIntList(int capacity) {  
        elementData = new int[capacity];  
        size = 0;  
    }  
    
    public ArrayIntList() {  
        this(10); // calls (int) constructor  
    }  
}
Size vs. capacity

- What happens if the client tries to access an element that is past the size but within the capacity (bounds) of the array?
  - Example: `list.get(7)`; on a list of size 5 (capacity 10)

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>8</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>size</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Answer: Currently the list allows this and returns 0.
  - Is this good or bad? What (if anything) should we do about it?
**Preconditions**

- **precondition**: Something your method *assumes is true* at the start of its execution.
  - Often documented as a comment on the method's header:

    ```java
    // Returns the element at the given index.
    // Precondition: 0 <= index < size
    public void remove(int index) {
        return elementData[index];
    }
    ```

  - Stating a precondition doesn't really "solve" the problem, but it at least documents our decision and warns the client what not to do.

  - What if we want to actually enforce the precondition?
What is wrong with the following way to handle violations?

```java
// Returns the element at the given index.
// Precondition: 0 <= index < size
public void remove(int index) {
    if (index < 0 || index >= size) {
        System.out.println("Bad index! " + index);
        return -1;
    }
    return elementData[index];
}
```

- returning -1 is no better than returning 0 (could be a legal value)
- `println` is not a very strong deterrent to the client (esp. GUI)
Throwing exceptions (4.5)

```java
throw new ExceptionType();
throw new ExceptionType("message");
```

- Causes the program to immediately crash with an exception.

- Common exception types:
  - ArithmeticException, ArrayIndexOutOfBoundsException, FileNotFoundException, IllegalArgumentException, IllegalStateException, IOException, NoSuchElementException, NullPointerException, RuntimeException, UnsupportedOperationException

- Why would anyone ever want the program to crash?
public void get(int index) {
    if (index < 0 || index >= size) {
        throw new ArrayIndexOutOfBoundsException(index);
    }
    return elementData[index];
}

- Exercise: Modify the rest of `ArrayIntList` to state preconditions and throw exceptions as appropriate.
Postconditions

- **postcondition**: Something your method *promises will be true* at the *end* of its execution.
  - Often documented as a comment on the method's header:

    ```java
    // Makes sure that this list's internal array is large enough to store the given number of elements.
    // Postcondition: elementData.length >= capacity
    public void ensureCapacity(int capacity) {
      // double in size until large enough
      while (capacity > elementData.length) {
        elementData = Arrays.copyOf(elementData, 2 * elementData.length);
      }
    }
    ```

- If your method states a postcondition, clients should be able to rely on that statement being true after they call the method.
• Some programs are written specifically to test other programs.

• If we wrote ArrayIntList and want to give it to others, we must make sure it works adequately well first.

• Write a client program with a main method that constructs several lists, adds elements to them, and calls the various other methods.
Tips for testing

• You cannot test every possible input, parameter value, etc.
  – Even a single \( \texttt{int} \) method has \( 2^{32} \) different possible values!
  – So you must think of a limited set of tests likely to expose bugs.

• Think about boundary cases
  – positive, zero, negative numbers
  – right at the edge of an array or collection's size

• Think about empty cases and error cases
  – 0, -1, null; an empty list or array
  – an array or collection that contains null elements

• Write helping methods in your test program to shorten it.
More testing tips

• Focus on **expected** vs. **actual** behavior

• the test shouldn't just call methods and print results; it should:
  – call the method(s)
  – compare their results to a known correct expected value
  – if they are the same, report that the test "passed"
  – if they differ, report that the test "failed" along with the values

• test behavior in combination
  – maybe `add` usually works, but fails after you call `remove`
  – what happens if I call `add` then `size`? `remove` then `toString`?
  – make multiple calls; maybe `size` fails the second time only
public static void main(String[] args) {
    int[] a1 = {5, 2, 7, 8, 4};
    int[] a2 = {2, 7, 42, 8};
    int[] a3 = {7, 42, 42};
    helper(a1, a2);
    helper(a2, a3);
    helper(new int[] {1, 2, 3, 4, 5}, new int[] {2, 3, 42, 4});
}

public static void helper(int[] elements, int[] expected) {
    ArrayIntList list = new ArrayIntList(elements);
    for (int i = 0; i < elements.length; i++) {
        list.add(elements[i]);
    }
    list.remove(0);
    list.remove(list.size() - 1);
    list.add(2, 42);
    for (int i = 0; i < expected.length; i++) {
        if (list.get(i) != expected[i]) {
            System.out.println("fail; expect " + Arrays.toString(expected) + ", actual " + list);
        }
    }
}