Here, we say `List` is a **generic interface** with a **type variable** named `Element`.

A **generic type** can be used to represent an arbitrary number of types created by filling in any object type for the **type variable**.

```
interface List<Element> {
  void add(Element s);
  Element get(int index);
  int size();
}
```

```
public interface List<String> {
  void add(String s);
  Integer get(int index);
  int size();
}
```

```
public interface List<Integer> {
  void add(Integer s);
  Integer get(int index);
  int size();
}
```

...

Here, we say `AList` is a **generic class** with a **type variable** named `E`.

Another term for generics is **parametric polymorphism.**

```
public class AList<E> implements List<E> {

  E[] elements;
  int size;

  @SuppressWarnings("unchecked")
  public AList() {
    this.elements = (E[])(new Object[2]);
    this.size = 0;
  }

  public void add(E s) {
    expandCapacity();
    this.elements[this.size] = s;
    this.size += 1;
  }

  public E get(int index) {
    // TODO: Check for out-of-bound
    // throw IndexOutOfBoundsException
    return this.elements[index];
  }

  public int size() {
    return this.size;
  }



  private void expandCapacity() {
    int currentCapacity = this.elements.length;
    if(this.size < currentCapacity) { return; }

    // How to construct new array here?



    for(int i = 0; i < this.size; i += 1) {
      expanded[i] = this.elements[i];
    }
    this.elements = expanded;
  }
}
```

```
class AList<String> implements List<String> {
  void add(String s) { ... }
  String get(int index) { ... }
  int size() { ... }

 }z
```

```
class AList<Integer> implements List<Integer> {
  void add(Integer s) { ... }
  Integer get(int index) { ... }
  int size() { ... }

 }
```

...

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class TestList {

  @Test
  public void testAdd() {
    List<String> slist = new AList<String>();
    slist.add("banana"); slist.add("apple");
    assertEquals("banana", slist.get(0));
    assertEquals("apple", slist.get(1));
  }

  @Test
  public void testAddThenSize() {
    List<Integer> ilist = new AList<Integer>();
    ilist.add(500); ilist.add(12);
    assertEquals(2, ilist.size());
  }

  @Test
  public void testListOfLists() {
    // Fill in declaration of bllist



    bllist.add(new AList<String>());
    bllist.add(new AList<String>());
    bllist.get(0).add("a");
    bllist.get(0).add("b");
    bllist.get(1).add("c");
    bllist.get(1).add("d");

    assertEquals("a", bllist.get(0).get(0));
    assertEquals("b", bllist.get(0).get(1));
    assertEquals("c", bllist.get(1).get(0));
    assertEquals("d", bllist.get(1).get(1));
  }
}
```

`public class` **`IndexOutOfBoundsException`**
`extends RuntimeException`

Thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range.

```
public class AList<E> implements List<E> {
  ... code from other side ...

  public E get(int index) {
    // TODO: Check for out-of-bounds









    return this.elements[index];
  }

}
```

Design principle: at the start of CSE12 we will mostly be writing code to **throw an informative exception** and **test for it** more than trying to **catch and handle exceptions**.

If the program is going wrong, it should probably stop before it can do any harm, and tell the user what happened. Catching errors and continuing is a whole-system design concern that shouldn't be handled by a data structure.

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class TestList {

  @Test(expected = IndexOutOfBoundsException.class)
  public void testNegativeIndex() {
    List<String> slist = new AList<String>();
    slist.add("banana");
    slist.get(-1);
  }

  // more tests for index out of bounds here!









}
```