Prepared by Linan Qiu <lq2137@columbia.edu>, adapted from Open Data Structures (opendatastructures.org) for CS3134 Spring 2016.

Iterators

Are you one of those guys who travel between carriages (cars) on the subway by going to that really loud door at the end of cars? If you are (or you know what these people are). Here's a picture of these doors:

Figure 1: Door between train carriages

An iterator is an extreme version of that person. He will start from the first train car, and go down one by one. He can only do two things:

- Tell you what the current car contains then actually jump to the next car
- Tell you when he's at the last car

Well this is exactly what an iterator is. **You should think of it as a little machine that holds on to one element in your list at a time. It tells you if it is still holding on to an element (or has went past the last element). It can also return you the current element, then go on to hold on to the next element.**

To codify this behavior, Java provides an interface (because again, an interface specifies the **can-do** relationship) for iterators: `Iterator<T>`.

Let's implement an iterator for `AwsmArrayList`!

AwsmArrayListIterator

```java
// AwsmArrayListIterator.java

import java.util.Iterator;

public class AwsmArrayListIterator<T> implements Iterator<T> {

  private int index;
  private AwsmArrayList<T> list;

  public AwsmArrayListIterator(AwsmArrayList<T> list) {
    this.list = list;
    index = 0;
  }

  @Override
  public boolean hasNext() {
    // TODO Auto-generated method stub
    return false;
  }

  @Override
  public T next() {
    // TODO Auto-generated method stub
    return null;
  }
}
```

This is what we start with. The `hasNext` and `next` methods are required by the `Iterator<T>` interface. What about the constructor?

Well, in order to jump through a train, the train guy needs to know which train to jump right? The 1 train is different from the A train, and even 1 trains are different among themselves. Similarly, iterating through a list of pokemon names should yield different results from iterating through a list of integers. Hence, each iterator **instance** is specific to an **instance** of the list.

Hence, in the constructor for `AwsmArrayListIterator` we need a list to iterate through, and we hold on to this list. We also keep track of our position in the list via the `index` variable which we set to 0 (ie. we start at the top.)

Implementing Iterator Functions

Now that we hold on to a list, we can easily implement `hasNext` and `next`.

Since we know what index we are currently at in the list, we can check if there's a next element by asking if `index < size` (eg. in a list of 5 elements, the last possible index is 4. Hence, if current index is less than 5, there is still an element to return via `next`.)

> Notice that we are no longer caring about the length of the array or if there's sufficient space for storage. That's the whole point of making an `AwsmArrayList`! We have less to worry about!

Similarly, to get the element, we can just call `list.get(index)`, then increment the index.

```
// AwsmArrayListIterator.java

import java.util.Iterator;

public class AwsmArrayListIterator<T> implements Iterator<T> {

  private int index;
  private AwsmArrayList<T> list;

  public AwsmArrayListIterator(AwsmArrayList<T> list) {
    this.list = list;
    index = 0;
  }

  @Override
  public boolean hasNext() {
    return index < list.size();
  }

  @Override
  public T next() {
    T item = list.get(index);
    index++;
    return item;
  }
}
```

Now our iterator is almost complete. However, we are not protecting ourselves against dumb users. Users who do not check `hasNext()` before calling `next()` may end up incrementing index beyond `size`. We should include an additional check in `next()`:

```
// AwsmArrayListIterator.java
```

```java
import java.util.Iterator;

public class AwsmArrayListIterator<T> implements Iterator<T> {

  private int index;
  private AwsmArrayList<T> list;

  public AwsmArrayListIterator(AwsmArrayList<T> list) {
    this.list = list;
    index = 0;
  }

  @Override
  public boolean hasNext() {
    return index < list.size();
  }

  @Override
  public T next() {
    if (index > list.size()) {
      throw new IndexOutOfBoundsException();
    }
    T item = list.get(index);
    index++;
    return item;
  }
}
```

Now we're done writing our iterator! But how do we use it in `AwsmArrayList`?

Using an Iterator Instance

Let's see how we can actually use an iterator first.

```java
// Test.java

public class Test {
  public static void main(String[] args) {
    AwsmArrayList<Integer> listA = new AwsmArrayList<>();
    for(int i = 0; i < 100; i++) {
      listA.addLast(i);
    }
    AwsmArrayListIterator<Integer> listAIterator = new AwsmArrayListIterator<>(listA);
    while(listAIterator.hasNext()) {
```

```
      System.out.println(listAIterator.next());
    }
  }
}
```

This provides us a really neat way to go through the entire list. In each step in the while loop, we check if the iterator is still holding on to an element. If it is, we ask for it and make the iterator go to the next element.

Different instances of lists need different iterators:

```
public class Test {
  public static void main(String[] args) {
    AwsmArrayList<Integer> listA = new AwsmArrayList<>();
    for (int i = 0; i < 100; i++) {
      listA.addLast(i);
    }
    AwsmArrayListIterator<Integer> listAIterator = new AwsmArrayListIterator<>(listA);
    while (listAIterator.hasNext()) {
      System.out.println(listAIterator.next());
    }

    AwsmArrayList<String> listB = new AwsmArrayList<>();
    listB.addLast("pikachu");
    listB.addLast("mew");
    listB.addLast("bulbasaur");
    AwsmArrayListIterator<String> listBIterator = new AwsmArrayListIterator<>(listB);
    while (listBIterator.hasNext()) {
      System.out.println(listBIterator.next());
    }
  }
}
```

This makes sense, since each train jumper is tied to a specific train. You can even have two iterators iterating through the same list. (two jumpers jumping the same train). That'd totally work, and doesn't violate anything.

> The terms `next` and `hasNext` may be a little confusing. It doesn't actually mean the "next" element. `hasNext` means *can the iterator still give me something?* and `next` means *give me the current element held by the iterator*. For example, if the current index is `0`. `hasNext` actually asks if `0` is a valid index (since it checks `index < size`) and `next` actually returns the element at `0`. This is because `0` is the **next possible element**. It does not refer to the element at `1`.

Now remember that `AwsmArrayListIterator` implements `Iterator`, so we can declare it as such:

```
Iterator<Integer> listAIterator = new AwsmArrayListIterator<>(listA)
```

and we lose almost no functionality, since the two public methods are still the same.

Now since each iterator instance is always tied to a list instance, why don't we just not make the user instantiate the `AwsmArrayListIterator` object and instead just get it directly from a list? For example,

```
// instead of writing this
Iterator<Integer> listAIterator = new AwsmArrayListIterator<>(listA)

// we write
Iterator<Integer> listAIterator = listA.iterator();
```

Well, turns out it isn't difficult to accomplish that. We just need to make use of the `iterator()` method that we left empty in `AwsmArrayList`.

Returning an Iterator Instance from `AwsmArrayList`

```
// AwsmArrayList.java

import java.util.Iterator;

public class AwsmArrayList<T> implements AwsmList<T> {

  private T[] data;
  private int size;

  public static final int INITIAL_SIZE = 8;
  public static final int GROWTH_FACTOR = 2;

  @SuppressWarnings("unchecked")
  public AwsmArrayList(int length) {
    data = (T[]) new Object[length];
    size = 0;
  }

  public AwsmArrayList() {
    this(INITIAL_SIZE);
  }
```

```
  // ... other methods redacted

  @Override
  public Iterator<T> iterator() {
    // AwsmArrayList<T> currentList = this;
    return new AwsmArrayListIterator<T>(this);
  }
}
```