CMPSCI 187: Programming With Data Structures

Lecture #11: Implementing Stacks With Arrays
David Mix Barrington
28 September 2012

Implementing Stacks With Arrays

- The Idea of the Implementation
- Data Fields for ArrayStack
- Constructors for ArrayStack
- Transformers: Pushing and Popping
- Observers for ArrayStack
- The `ArrayListStack` Class
- Application: Reversing Some Strings

The Idea of the Implementation

- In our first implementation of the Stack idea, we will use an array. Like ArrayStringLog, we will have a capacity limit and thus will implement `BoundedStackInterface` rather than `UnboundedStackInterface`.

- The `ArrayStack<T>` generic class will have an array of T objects as its central data structure. Again as with ArrayStringLog, the objects in the stack will occupy a **consecutive initial sequence** of the locations in the array.

- We will throw a `StackOverflowException` if a client asks us to push a new item when the array is full. We thus need an `isFull` method to allow the client to check for this condition.

- As with any stack, popping or topping when the stack is empty causes a `StackUnderflowException`. We need an `isEmpty` method to allow guarding against this, and a `size` method to say how many items we have.

Data Fields for ArrayStack

- We begin the code for the class with declarations of the data fields. We need a named constant for the default capacity, an array of T elements for the actual data storage, and an int variable pointing to the top of the stack.

- With the stack empty, again as with ArrayStringLog, we set `topIndex` to -1 even though trying to access this array element would throw an exception. This choice preserves other invariants like "all the indices x with x > topIndex are unused and have null contents". As with other empty sets, correct phrasing of the invariants tells us how to represent an empty stack.

```
public class ArrayStack<T> implements
                            BoundedStackInterface<T> {
   protected final int DEFCAP = 100; //default capacity
   protected T[ ] stack; // holds stack elements
   protected int topIndex = -1; // index of top element
```

Constructors for ArrayStack

- The only field to initialize in our constructors is the array itself, which should have and will have all its indices pointing to `null` when it is created.

- We would like to create an array of type `T[ ]` but we can't -- there are no constructors for class variables, because we can't create an array of objects without knowing their types. So we create an array of Objects, knowing that our T items will all be Objects, and **cast** the resulting array into the type `T[ ]`. The compiler warns us that we can't be sure that there are T's in the array, but we ignore this because we know that the array is all nulls anyway.

- Note that we could use the `this` constructor though DJW choose not to.

```
public ArrayStack( ) {
   stack = (T[ ]) new Object[DEFCAP];}

public ArrayStack(int maxSize) {
   stack = (T[ ]) new Object[maxSize];}
```

Transformers: Pushing

- Pushing a new element onto the stack works exactly like the insert method of ArrayStringLog. The size of the array, and thus also the index of the last used location, will increase by 1. So we increment `topIndex` and put the new element into `stack[topIndex]`.

- Pushing onto a full array throws an exception. We don't need a `throws` clause because the exception is unchecked. The exception may still be caught by a method that calls `push`, directly or indirectly. If it is not caught by any method up to the original `main` method, the program crashes.

```
public void push (T element) {
   if (!isFull( )) {
      topIndex++;
      stack[topIndex] = element;}
   else throw new StackOverflowException("push to full stack");}
```

Transformers: Popping

- Remember that DJW, unlike the Java `Stack<T>` class, separate the transformer `pop` from the observer `top`. So `pop` just discards the top element, while `top` returns it (that is, makes another pointer to it).

- We spend some time setting the abandoned location to null, which one could argue is unnecessary because we will never look at that location again before putting a new element there with a push. But we do free up the memory used by the object, since it will be garbage-collected once the array no longer points to it.

```
public void pop( ) {
   if (!isEmpty( )) {
      stack[topIndex] = null;
      topIndex--;}
   else throw new StackUnderflowException("pop empty stack");}
```

Observers for ArrayStack

- The size method is not in the book. Note again that the top method (called "peek" by `java.util.Stack<T>`) returns the element without removing it. I've saved space by replacing DJW's "if (b) then return true; else return false;" with the equivalent "return b;" where b is a boolean.

```
public boolean isEmpty( ) {
   return (topIndex == -1);}

public boolean isFull( ) {
   return (topIndex == (stack.length - 1));}

public int size( ){// number of items in the stack
   return topIndex + 1;}

public T top( ) {
   T topOfStack = null;
   if (!isEmpty( ))
      topOfStack = stack[topIndex];
   else throw new StackUnderflowException("top of empty stack");
   return topOfStack;}
```

The `ArrayListStack` Class

- The Collections package includes a variable-length generic array data structure called `ArrayList<T>`. It begins with some fixed capacity, but when you "add" another element to a full one, it copies itself into another array of twice the size. Thus it appears to be an array of unlimited size.

- DJW give code for an `ArrayListStack<T>` class on pages 192-3. It uses the `add` and `remove` methods of `ArrayList` to implement `push` and `pop` respectively, and its code is otherwise similar to that of `ArrayStack<T>`.

- The constructor of `ArrayListStack<T>` can call the constructor of `ArrayList<T>` to create an ArrayList of the desired size. It doesn't need to create an Object array and then cast it into a T array as `ArrayStack` did, because `ArrayList<T>` does have a constructor. Of course, *that* structure is implemented with an array, so *that* constructor must create an Object array and do the cast, as must the code that resizes the array by making a new one.

Application: Reversing Some Strings

- Here's a simple example that gets three strings from the user and prints them out in reverse order. In Discussion #3 we had an example where we used two stacks to sort the strings in a StringBag, shifting strings from one stack to the other.

```
public class ReverseStrings {
   public static void main (String[ ] args) {
      Scanner conIn = new Scanner(System.in);
      BoundedStackInterface<String> stack;
      stack = new ArrayStack<String>(3);
      String line;
      for (int i = 1; i <= 3; i++) {
         System.out.println("Enter a line of text:");
         line = conIn.nextLine( );
         stack.push(line);}
      System.out.println("\nReverse is:\n");
      while (!stack.isEmpty( )) {
         line = stack.top( );
         stack.pop( );
         System.out.println(line);}}}
```