Storage Strategies: Static Arrays

- StackADT Interface
- ArrayStack Implementation
- ArrayStack Methods with Big-O analysis
- StacklIterator Class
- StacklIterator Methods
- StacklIterator Summary
- Reading: L&C 3.6-3.8, 7.3
Stack Abstract Data Type

• A *stack* is a linear collection where the elements are added or removed from the same end
• The processing is *last in, first out* (*LIFO*)
• The last element put on the stack is the first element removed from the stack
• Think of a stack of cafeteria trays
Stack Terminology

- We *push* an element on a stack to add one
- We *pop* an element off a stack to remove one
- We can also *peek* at the top element without removing it
- We can determine if a stack is *empty* or not and how many elements it contains (its *size*)
- The StackADT interface supports the above operations and some typical class operations such as `toString()`
StackADT and Stack Classes

Since the Java Collections all extend Iterable<T>, I have added that to all my versions of the textbook examples.

Each implementing class satisfies the ADT although they each use a different internal storage strategy.
Stack Design Considerations

• Although a stack can be empty, there is no concept for it being full. An implementation must be designed to manage storage space.

• For peek and pop operation on an empty stack, the implementation would throw an exception. There is no other return value that is equivalent to “nothing to return”.

• A drop-out stack is a variation of the stack design where there is a limit to the number of elements that are retained.
ArrayStack Implementation

- We can use an array of elements as a stack
- The top is the index of the next available element in the array

```
top
integer
```

```
T [] stack
Type T reference
Type T reference
null
```

```
Object of type T
Object of type T
```
ArrayStack Methods

• An interface can’t define any constructor methods, but any implementing class needs to have one or more of them (maybe overloading the constructor)

• Default Constructor:
  ```java
  public ArrayStack()
  {
      // must be 1st statement
      this(DEFAULT_CAPACITY); // call other constructor
  } // with default capacity
  ```

• Constructor with a specified initial capacity:
  ```java
  public ArrayStack(int initialCapacity)
  {
      top = 0;
      stack = (T[]) new Object[initialCapacity];
  }
  ```
Array Stack Implementation

• push – O(1)

```java
public void push (T element)
{
    if (size() == stack.length)
        expandCapacity();  // see next slide
    stack [top++] = element;
}
```

• Because a Java array’s size cannot be changed after instantiation, the add method may need to allocate a larger array, copy the data to the new array, and release the memory of the old array
ArrayStack Methods

- expandCapacity – O(n)

```java
private void expandCapacity()
{
    T[] larger = // double the array size
        (T[]) new Object[2 * contents.length];
    for (int i = 0; i < contents.length; i++)
        larger[i] = stack[i];

    stack = larger;       // original array
                          // becomes garbage
}
```
Array Stack Implementation

• pop() – O(1)

```java
public T pop() throws EmptyStackException {
    if (isEmpty())
        throw new EmptyStackException();
    T result = stack[--top];
    stack[top] = null; // removes “stale” reference
    return result;
}
```

• The “stale” reference stored in stack[top] would prevent garbage collection on the object when the caller sets the returned reference value to null – ties up resources
ArrayStack Implementation

- peek() – O(1)

```java
public T peek() throws EmptyStackException {
    if (isEmpty())
        throw new EmptyStackException();
    return stack[top - 1];
}
```
ArrayStack Methods

• size - O(1)
  
  public int size()
  {
    return top;
  }

• isEmpty – O(1)
  
  public boolean isEmpty()
  {
    return top == 0;
  }
ArrayStack Methods

• **toString – O(n)**
  ```java
  public String toString() {
      String result = "";

      for (T obj : stack) {
          if (obj == null)  // first null is at top
              return result;
          result += obj + "\n";
      }
      return result;  // exactly full – no nulls
  }
  ```
ArrayStack Methods

• All Java Collections API classes implement (indirectly) the Iterable interface and I add that to the definition of all textbook classes

• iterator – O(1)

```java
public Iterator<T> iterator()
{
    return new StackIterator<T>();
}
```

• We need to study the StackIterator class to understand how to implement an Iterator
StackIterator Class

• The iterator method of the ArrayStack class instantiates and returns a reference to a new StackIterator object to its caller
• If an iterator class is very closely related to its collection class, it is a good candidate for implementation as an inner class
• As an inner class, the StackIterator code can access the stack and top variables of the instance of the outer class that instantiated it
StackIterator Definition/Attributes

• Class Definition/Attribute Declarations (implemented as an inner class)

  private class StackIterator<T>
      implements Iterator<T>
  {
      private int current;
  }

• Constructor:

  public StackIterator()
  {
      current = top;  // start at top for LIFO
StackIterator Methods

• **hasNext – O(1)**
  
  ```java
  public boolean hasNext()
  {
      return current > 0;
  }
  ```

• **next – O(1)**

  ```java
  public T next()
  {
      if (!hasNext())
          throw new NoSuchElementException();
      return stack[--current]; // outer class array
  }
  ```
StackIterator Methods

• remove – O(1)
• We may or may not implement real code for the remove method, but there is no return value that we can use to indicate that it is not implemented
• If we don’t implement it, we may indicate that it is not implemented by throwing an exception

```java
public void remove() throws UnsupportedOperationException {
    throw new UnsupportedOperationException();
}
```
StackIterator Methods

• If we do implement the remove method, notice that we don’t specify the element that is to be removed and we do not return a reference to the element being removed.

• It is assumed that the calling code has been iterating on condition `hasNext()` and calling `next()` and already has a reference.

• The last element returned by `next()` is the element that will be removed.
StackIterator Method Analysis

• Each of the StackIterator methods is O(1)
• However, they are usually called inside an external while loop or “for-each” loop
• Hence, the process of “iterating” through a collection using an Iterator is O(n) where n is the number of objects in the collection
ArrayListIterator Class in Textbook

- The textbook’s iterator classes detect any modification to the array and cause the iteration process to “fast-fail” with an exception
- The add and remove methods of the outer class update a variable: \texttt{modCount}
- The iterator’s constructor copies that value
- If the value of \texttt{modCount} changes during the iteration, the iterator code throws an exception
- I have not included that in my example code, but it is included in the Java Collections classes