Stacks & Queues

Why do computer scientists come up with their own definitions for common words?
List, Tree, Type, Class, Bug, Escape

To make a list of the types of bugs escaping up the tree (class).

Questions From Last Time

1

Drawings

2

What Are We Doing Again?

We're learning some new data structures (we're going to be the client of them!).

Today's Main Goals:

- Finish up last time
- To understand the difference between an interface and an implementation
- To understand what stacks and queues are
Duplicated Code: Constructors

We’d like to have two constructors for ArrayIntList:

- One that uses a default size
- One that uses a size given by the user

This a lot of redundant code! How can we fix it?

Fixed Constructor

Java allows us to call one constructor from another using this(. . .):

```java
1 public ArrayIntList() {
2   this(10);
3 }
```

Class CONSTANTS

Looking back at the constructor, what’s ugly about it?

```java
1 public ArrayIntList() {
2   this(10);
3 }
```

The 10 is a “magic constant”; this is really bad style! We can use:

```
public static final int DEFAULT_CAPACITY = 10;
```

to declare a class constant.

So, for instance:

```
public static final int DEFAULT_CAPACITY = 10;
```

This Code Is Redundant

1 List<String> list = new List<String>(); // BAD : WON’T COMPILE
List is a description of what a collection of data can do. We usually specify these with interfaces.

Abstract Data Types (ADT)

An abstract data type is a description of what a collection of data can do. We usually specify these with interfaces.

List ADT

In Java, a List can add, remove, size, get, set.

List Implementations

An ArrayList is a particular type of List. Because it is a list, we promise it can do everything a List can. A LinkedList is another type of List.

Even though we don’t know how it works, we know it can do everything a List can, because it’s a List.

Using the List ADT

This is INVALID CODE

1 List<String> list = new ArrayList<String>();
2 // BAD : WON’T COMPILE
List is a description of methods. It doesn’t specify how they work.

This Code Is Redundant

```java
1 ArrayList<Integer> list = new ArrayList<Integer>();
2 list.add(5);
3 list.add(6);
4 for (int i = 0; i < list.size(); i++) {
5   System.out.println(list.get(i));
6 }
7 }
8 ArrayList<Integer> list = new ArrayList<Integer>();
9 list.add(5);
10 list.add(6);
11 for (int i = 0; i < list.size(); i++) {
12   System.out.println(list.get(i));
13 }
```

We can’t condense it any more when written this way, because ArrayList and LinkedList are totally different things.

NOT Using the List ADT

Instead, we can use the List interface and swap out different implementations of lists:

```java
1 List<Integer> list = new ArrayList<Integer>();
2 // = new LinkedList<Integer>();
3 // We can choose which implementation
4 // And the code below will work the
5 // same way for both of them!
6 list.add(5);
7 list.add(6);
8 for (int i = 0; i < list.size(); i++) {
9   System.out.println(list.get(i));
10 }
```

The other benefit is that the code doesn’t change based on which implementation we (or a client!) want to use!
Queues

A queue is a collection which orders the elements first-in-first-out ("FIFO"). Note that, unlike lists, queues do not have indices.
- Elements are stored internally in order of insertion.
- Clients can ask for the first element (dequeue/peek).
- Clients can ask for the size.
- Clients can add to the back of the queue (enqueue).
- Clients may only see the first element of the queue.

```
Client: ← /leftrightline 7 7 7 7 7 ← /leftrightline dequeue() Client: ← /leftrightline 2 2 2 2 2 ← /leftrightline
Impl: ← /leftrightline 7 7 7 7 7 ← /leftrightline / / / / / / |
```

Queue Reference

Queue is an interface. So, you create a new Queue with:
```
Queue<Integer> queue = new LinkedList<Integer>();
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue(val)</td>
<td>Adds val to the back of the queue</td>
</tr>
<tr>
<td>dequeue()</td>
<td>Removes the first value from the queue; throws a NoSuchElementException if the queue is empty</td>
</tr>
<tr>
<td>peek()</td>
<td>Returns the first value in the queue without removing it; throws a NoSuchElementException if the queue is empty</td>
</tr>
<tr>
<td>size()</td>
<td>Returns the number of elements in the queue</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>Returns true if the queue has no elements</td>
</tr>
</tbody>
</table>

Okay; Wait; Why?

A queue seems like what you get if you take a list and remove methods.

```
Client: ← /leftrightline 7 7 4 2 3 ← /leftrightline enqueue(9) Client: ← /leftrightline 2 2 2 2 2 ← /leftrightline
Impl: ← /leftrightline 7 7 4 2 3 ← /leftrightline / / / / / / |
```

Applications Of Queues

- Queue of print jobs to send to the printer
- Queue of programs / processes to be run
- Queue of keys pressed and not yet handled
- Queue of network data packets to send
- Queue of button/keyboard/etc. events in Java
- Modeling any sort of line
- Queuing Theory (subfield of CS about complex behavior of queues)

Stacks

A stack is a collection which orders the elements last-in-first-out ("LIFO"). Note that, unlike lists, stacks do not have indices.

```
Client: ← /leftrightline 9 9 7 7 7 7 ← /leftrightline push(3) Client: ← /leftrightline 9 9 7 7 7 7 ← /leftrightline
Impl: ← /leftrightline 9 9 7 7 7 7 ← /leftrightline / / / / / / |
```

Applications of Stacks

- Your programs use stacks to run:
  ```java
  (pop = return, method call = push)
  ```

```
1 public static fun1() {
2    fun2(5);
3 }
4 public static fun2(int i) {
5    return 2*i; //At this point!
6 }
7 public static void main(String[] args) {
8    System.out.println(fun1());
9 }
```

- Compilers parse expressions using stacks
- Stacks help convert between infix (3 + 2) and postfix (3 2 +)
  - (This is important, because postfix notation uses fewer characters.)
- Many programs use “undo stacks” to keep track of user operations.
Stack is an interface. So, you create a new Stack with:

```java
Stack<Integer> stack = new Stack<>();
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(val)</td>
<td>Places val on top of the stack</td>
</tr>
<tr>
<td>pop()</td>
<td>Removes top value from the stack and returns it; throws NoSuchElementException if stack is empty</td>
</tr>
<tr>
<td>peek()</td>
<td>Returns top value from the stack without removing it; throws NoSuchElementException if stack is empty</td>
</tr>
<tr>
<td>size()</td>
<td>Returns the number of elements in the stack</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>Returns true if the stack has no elements</td>
</tr>
</tbody>
</table>

Consider the code we ended with for ReverseFile from the first lecture:

Print out words in reverse, then the words in all capital letters

```java
ArrayList<String> words = new ArrayList<>();
Scanner input = new Scanner(new File("words.txt"));
while (input.hasNext()) {
    String word = input.next();
    words.add(word);
}
for (int i = words.size() - 1; i >= 0; i--){
    System.out.println(words.get(i));
    System.out.println(words.get(i).toUpperCase());
}
```

We used an ArrayList, but then we printed in reverse order. A Stack would work better!

This is the equivalent code using Stacks instead:

Doing it with Stacks

```java
Stack<String> words = new Stack<>();
Scanner input = new Scanner(new File("words.txt"));
while (input.hasNext()) {
    String word = input.next();
    words.push(word);
}
Stack<String> copy = new Stack<>();
while (!words.isEmpty()) {
    copy.push(words.pop());
    System.out.println(words.peek());
}
while (!copy.isEmpty()) {
    System.out.println(copy.pop().toUpperCase());
}
```

Illegal Stack Operations

You may NOT use get on a stack!

```java
Stack<Integer> s = new Stack<Integer>();
for (int i = 0; i < s.size(); i++) {
    System.out.println(s.get(i));
}
```

get, set, etc. are not valid stack operations.

Instead, use a while loop

```java
Stack<Integer> s = new Stack<Integer>();
while (!s.isEmpty()) {
    System.out.println(s.pop());
}
```

Note that as we discovered, the while loop destroys the stack.