Adam Blank | Lecture 4 | Autumn 2016

CSE 143

Computer Programming II

CSE 143: Computer Programming II

Stacks & Queues

- Why do Computer Scientists come up with their own definitions for common words?

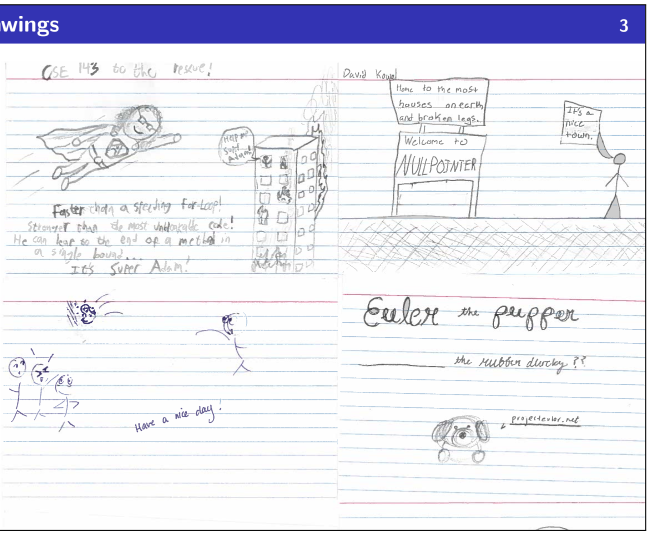
List, Tree, Type, class, Bug, Escape

To make a list of the types of bugs escaping up the tree. Classy.

Questions From Last Time

Drawings

Drawings


CSE 143 to the rescue!
Help me Super Adam!
Faster than a speeding For-Loop!
Stronger than the most unbreakable code!
He can leap to the end of a method in a single bound...
It's Super Adam!
David Kowal
Home to the most houses on earth and broken legs.
Welcome to
NULLPOINTER
It's a nice town.
Have a nice day!
Euler the pupper
the rubber ducky??
projecteuler.net

What Are We Doing Again?

What Are We Doing...?

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 betweeen 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

Redundant Constructors

```java
/* Inside the ArrayIntList class... */
public ArrayIntList() {
   this.data = new int[10];
   this.size = 0;
}

public ArrayIntList(int capacity) {
   this.data = new int[capacity];
   this.size = 0;
}
```

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

Fixed Constructor

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

```java
public ArrayIntList() {
   this(10);
}
```

Class CONSTANTS

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

```java
public ArrayIntList() {
   this(10);
}
```

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

`public static final` **type name** = **value**

to declare a **class constant**.

So, for instance:

```java
public static final int DEFAULT_CAPACITY = 10.
```

Class CONSTANT

A class constant is a **global**, **unchangable** value in a class. Some examples:

- `Math.PI`
- `Integer.MAX_VALUE`, `Integer.MIN_VALUE`
- `Color.GREEN`

Outline

1 Interfaces

2 Queues

3 Stacks

Abstract Data Types (ADT)

Abstract Data Type

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

```java
List<String> list = new List<String>(); // BAD : WON'T COMPILE
```

List is a description of methods. It doesn't specify **how they work**.

This Code Is Redundant

```java
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(5);
list.add(6);

for (int i = 0; i < list.size(); i++) {
   System.out.println(list.get(i));
}

LinkedList<Integer> list = new LinkedList<Integer>();
list.add(5);
list.add(6);

for (int i = 0; i < list.size(); i++) {
   System.out.println(list.get(i));
}
```

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:

This Uses Interfaces Correctly!

```java
List<Integer> list = new ArrayList<Integer>();
                // = new LinkedList<Integer>();
                // We can choose which implementation
                // And the code below will work the
                // same way for both of them!
list.add(5);
list.add(6);

for (int i = 0; i < list.size(); i++) {
   System.out.println(list.get(i));
}
```

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

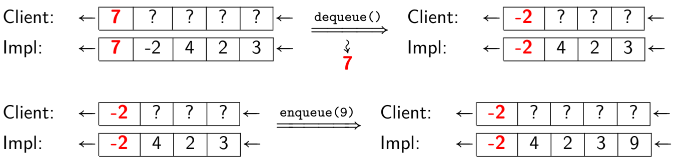
Queues

Queue

Real-world queues: a service line, printer jobs

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**.



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)

Queue Reference

Queue is an interface. So, you create a new Queue with:

```
Queue<Integer> queue = new LinkedList<Integer>();
```

enqueue(**val**)	Adds **val** to the back of the queue
dequeue()	Removes the first value from the queue; throws a NoSuchElementException if the queue is empty
peek()	Returns the first value in the queue without removing it; throws a NoSuchElementException if the queue is empty
size()	Returns the number of elements in the queue
isEmpty()	Returns true if the queue has no elements

Okay; Wait; Why?

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

Well. . . yes. . .

- This prevents the client from doing something they shouldn't.
- This ensures that all valid operations are fast.
- Having fewer operations makes queues easy to reason about.

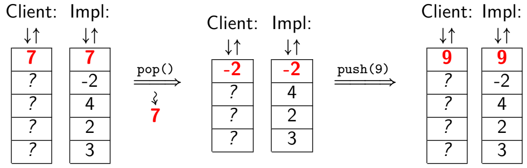
Stacks

Stack

Real-world stacks: stock piles of index cards, trays in a cafeteria

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

- Elements are stored internally in order of insertion.
- Clients can ask for the top element (**pop**/**peek**).
- Clients can ask for the size.
- Clients can add to the top of the stack (**push**).
- Clients **may only see the top element of the stack**

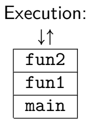


Applications of Stacks

- Your programs use stacks to run:

  (pop = return, method call = push)!

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



- 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 Reference

Stack is an interface. So, you create a new Stack with:

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

Stack<E>()	Constructs a new stack with elements of type **E**
push(**val**)	Places **val** on top of the stack
pop()	Removes top value from the stack and returns it; throws NoSuchElementException if stack is empty
peek()	Returns top value from the stack without removing it; throws NoSuchElementException if stack is empty
size()	Returns the number of elements in the stack
isEmpty()	Returns true if the stack has no elements

Stack Reference

Back to ReverseFile

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

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

```
ArrayList<String> words = new ArrayList<String>();

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));
}
for (int i = words.size() − 1; i >= 0; i−−) {
   System.out.println(words.get(i).toUpperCase());
}
```

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

ReverseFile with Stacks

This is the equivalent code using Stacks instead:

Doing it with Stacks

```
Stack<String> words = new Stack<String>();

Scanner input = new Scanner(new File("words.txt"));

while (input.hasNext()) {
   String word = input.next();
   words.push(word);
}

Stack<String> copy = new Stack<String>();
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!

```
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

```
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**.