

# **CSE 143**

# **Lecture 7**

Stacks and Queues

slides adapted from Marty Stepp

<http://www.cs.washington.edu/143/>

# Runtime Efficiency (13.2)

- **efficiency**: A measure of the use of computing resources by code.
  - can be relative to speed (time), memory (space), etc.
  - most commonly refers to run time
- Assume the following:
  - Any single Java statement takes the same amount of time to run.
  - A method call's runtime is measured by the total of the statements inside the method's body.
  - A loop's runtime, if the loop repeats  $N$  times, is  $N$  times the runtime of the statements in its body.

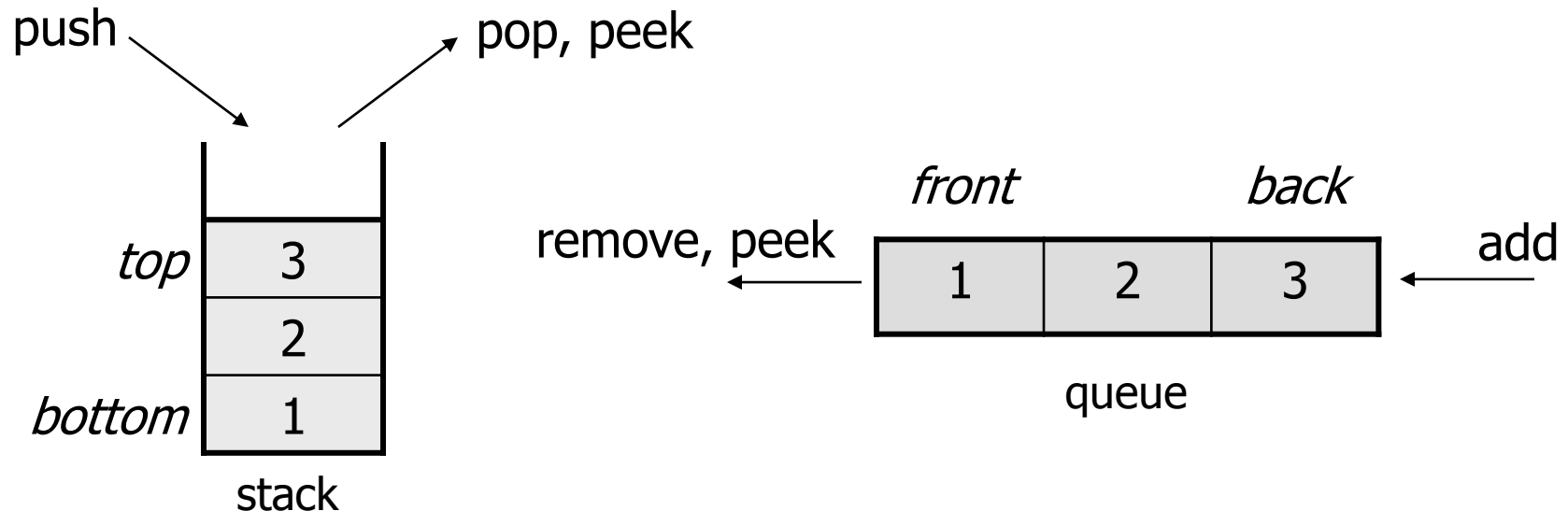
# ArrayList methods

- Which operations are most/least efficient, and why?

<code>add ( <b>value</b> )</code>	appends value at end of list
<code>add ( <b>index</b>, <b>value</b> )</code>	inserts given value at given index, shifting subsequent values right
<code>clear ()</code>	removes all elements of the list
<code>indexOf ( <b>value</b> )</code>	returns first index where given value is found in list (-1 if not found)
<code>get ( <b>index</b> )</code>	returns the value at given index
<code>remove ( <b>index</b> )</code>	removes/returns value at given index, shifting subsequent values left
<code>set ( <b>index</b>, <b>value</b> )</code>	replaces value at given index with given value
<code>size ()</code>	returns the number of elements in list
<code>toString ()</code>	returns a string representation of the list such as "[3, 42, -7, 15]"

# Stacks and queues

- Sometimes it is good to have a collection that is less powerful, but is optimized to perform certain operations very quickly.
- Today we will examine two specialty collections:
  - **stack**: Retrieves elements in the reverse of the order they were added.
  - **queue**: Retrieves elements in the same order they were added.



# Abstract data types (ADTs)

- **abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it
- We don't know exactly how a stack or queue is implemented, and we don't need to.
  - We just need to understand the idea of the collection and what operations it can perform.

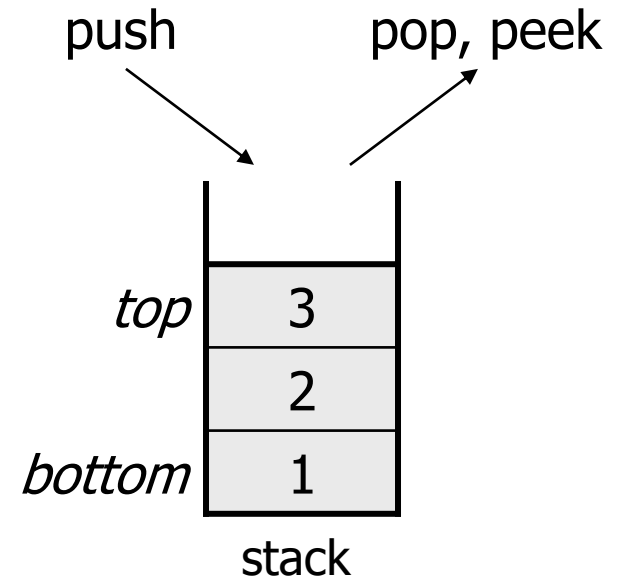
(Stacks are usually implemented with arrays; queues are often implemented using another structure called a linked list.)

# Stacks

- **stack**: A collection based on the principle of adding elements and retrieving them in the opposite order.
  - Last-In, First-Out ("LIFO")
  - The elements are stored in order of insertion, but we do not think of them as having indexes.
  - The client can only add/remove/examine the last element added (the "top").

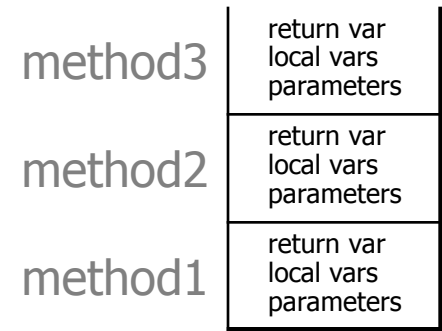


- basic stack operations:
  - **push**: Add an element to the top.
  - **pop**: Remove the top element.
  - **peek**: Examine the top element.



# Stacks in computer science

- Programming languages and compilers:
  - method calls are placed onto a stack (*call=push, return=pop*)
  - compilers use stacks to evaluate expressions
- Matching up related pairs of things:
  - find out whether a string is a palindrome
  - examine a file to see if its braces { } match
  - convert "infix" expressions to pre/postfix
- Sophisticated algorithms:
  - searching through a maze with "backtracking"
  - many programs use an "undo stack" of previous operations



# Class Stack

<code>Stack&lt;E&gt;()</code>	constructs a new stack with elements of type <b>E</b>
<code>push (value)</code>	places given value on top of stack
<code>pop ()</code>	removes top value from stack and returns it; throws <code>EmptyStackException</code> if stack is empty
<code>peek ()</code>	returns top value from stack without removing it; throws <code>EmptyStackException</code> if stack is empty
<code>size ()</code>	returns number of elements in stack
<code>isEmpty ()</code>	returns <code>true</code> if stack has no elements

```
Stack<Integer> s = new Stack<Integer>();  
s.push(42);  
s.push(-3);  
s.push(17); // bottom [42, -3, 17] top  
System.out.println(s.pop()); // 17
```

– Stack has other methods, but we forbid you to use them.



# Stack limitations/idioms

- Remember: You cannot loop over a stack in the usual way.

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

```
...
```

```
for (int i = 0; i < s.size(), i++) {  
    do something with s.get(i);  
}
```

- Instead, you must pull contents out of the stack to view them.
  - common idiom: Removing each element until the stack is empty.

```
// process (and destroy) an entire stack  
while (!s.isEmpty()) {  
    do something with s.pop();  
}
```

# Exercise

- Consider an input file of exam scores in reverse ABC order:

Yeilding	Janet	87
White	Steven	84
Todd	Kim	52
Tashev	Sylvia	95
...		

- Write code to print the exam scores in ABC order using a stack.
  - What if we want to further process the exams after printing?

# What happened to my stack?

- Suppose we're asked to write a method `max` that accepts a Stack of integers and returns the largest integer in the stack:

```
// Precondition: !s.isEmpty()
public static void max(Stack<Integer> s) {
    int maxValue = s.pop();
    while (!s.isEmpty()) {
        int next = s.pop();
        maxValue = Math.max(maxValue, next);
    }
    return maxValue;
}
```

- The algorithm is correct, but what is wrong with the code?

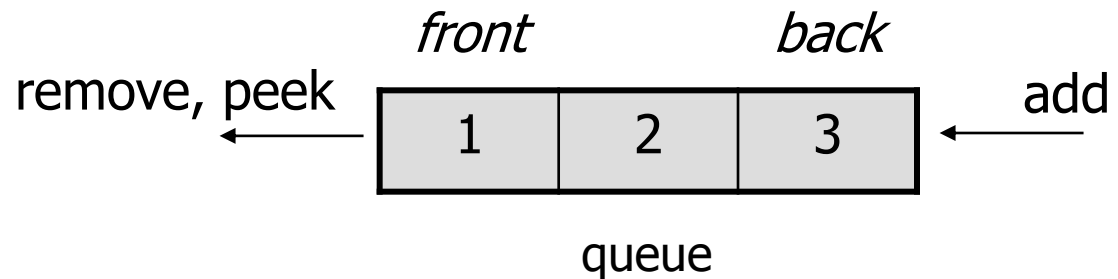
# What happened to my stack?

- The code destroys the stack in figuring out its answer.
  - To fix this, you must save and restore the stack's contents:

```
public static void max(Stack<Integer> s) {  
    Stack<Integer> backup = new Stack<Integer>();  
    int maxValue = s.pop();  
    backup.push(maxValue);  
    while (!s.isEmpty()) {  
        int next = s.pop();  
        backup.push(next);  
        maxValue = Math.max(maxValue, next);  
    }  
    while (!backup.isEmpty()) { // restore  
        s.push(backup.pop());  
    }  
    return maxValue;  
}
```

# Queues

- **queue**: Retrieves elements in the order they were added.
  - First-In, First-Out ("FIFO")
  - Elements are stored in order of insertion but don't have indexes.
  - Client can only add to the end of the queue, and can only examine/remove the front of the queue.



- basic queue operations:
  - **add** (enqueue): Add an element to the back.
  - **remove** (dequeue): Remove the front element.
  - **peek**: Examine the front element.

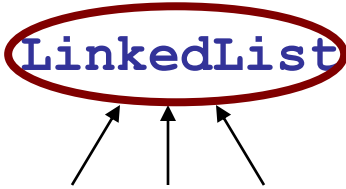
# Queues in computer science

- Operating systems:
  - queue of print jobs to send to the printer
  - queue of programs / processes to be run
  - queue of network data packets to send
- Programming:
  - modeling a line of customers or clients
  - storing a queue of computations to be performed in order
- Real world examples:
  - people on an escalator or waiting in a line
  - cars at a gas station (or on an assembly line)

# Programming with Queues

<code>add (value)</code>	places given value at back of queue
<code>remove ()</code>	removes value from front of queue and returns it; throws a <code>NoSuchElementException</code> if queue is empty
<code>peek ()</code>	returns front value from queue without removing it; returns <code>null</code> if queue is empty
<code>size ()</code>	returns number of elements in queue
<code>isEmpty ()</code>	returns <code>true</code> if queue has no elements

```
Queue<Integer> q = new LinkedList<Integer> ();  
q.add(42);  
q.add(-3);  
q.add(17);           // front [42, -3, 17] back  
System.out.println(q.remove()); // 42
```



- **IMPORTANT:** When constructing a queue you must use a new `LinkedList` object instead of a new `Queue` object.
  - This has to do with a topic we'll discuss later called *interfaces*.

# Queue idioms

- As with stacks, must pull contents out of queue to view them.

```
// process (and destroy) an entire queue
```

```
while (!q.isEmpty()) {  
    do something with q.remove();  
}
```

- another idiom: Examining each element exactly once.

```
int size = q.size();  
for (int i = 0; i < size; i++) {  
    do something with q.remove();  
    (including possibly re-adding it to the queue)  
}
```

- Why do we need the `size` variable?



# Mixing stacks and queues

- We often mix stacks and queues to achieve certain effects.
  - Example: Reverse the order of the elements of a queue.

```
Queue<Integer> q = new LinkedList<Integer>();  
q.add(1);  
q.add(2);  
q.add(3); // [1, 2, 3]
```

```
Stack<Integer> s = new Stack<Integer>();  
while (!q.isEmpty()) { // Q -> S  
    s.push(q.remove());  
}  
while (!s.isEmpty()) { // S -> Q  
    q.add(s.pop());  
}  
System.out.println(q); // [3, 2, 1]
```

# Exercise

- Modify our exam score program so that it reads the exam scores into a queue and prints the queue.
  - Next, filter out any exams where the student got a score of 100.
  - Then perform your previous code of reversing and printing the remaining students.
    - What if we want to further process the exams after printing?

# Exercises

- Write a method `stutter` that accepts a queue of integers as a parameter and replaces every element of the queue with two copies of that element.

– `front [1, 2, 3] back`

**becomes**

`front [1, 1, 2, 2, 3, 3] back`

- Write a method `mirror` that accepts a queue of strings as a parameter and appends the queue's contents to itself in reverse order.

– `front [a, b, c] back`

**becomes**

`front [a, b, c, c, b, a] back`