

# Building Java Programs

Appendix Q

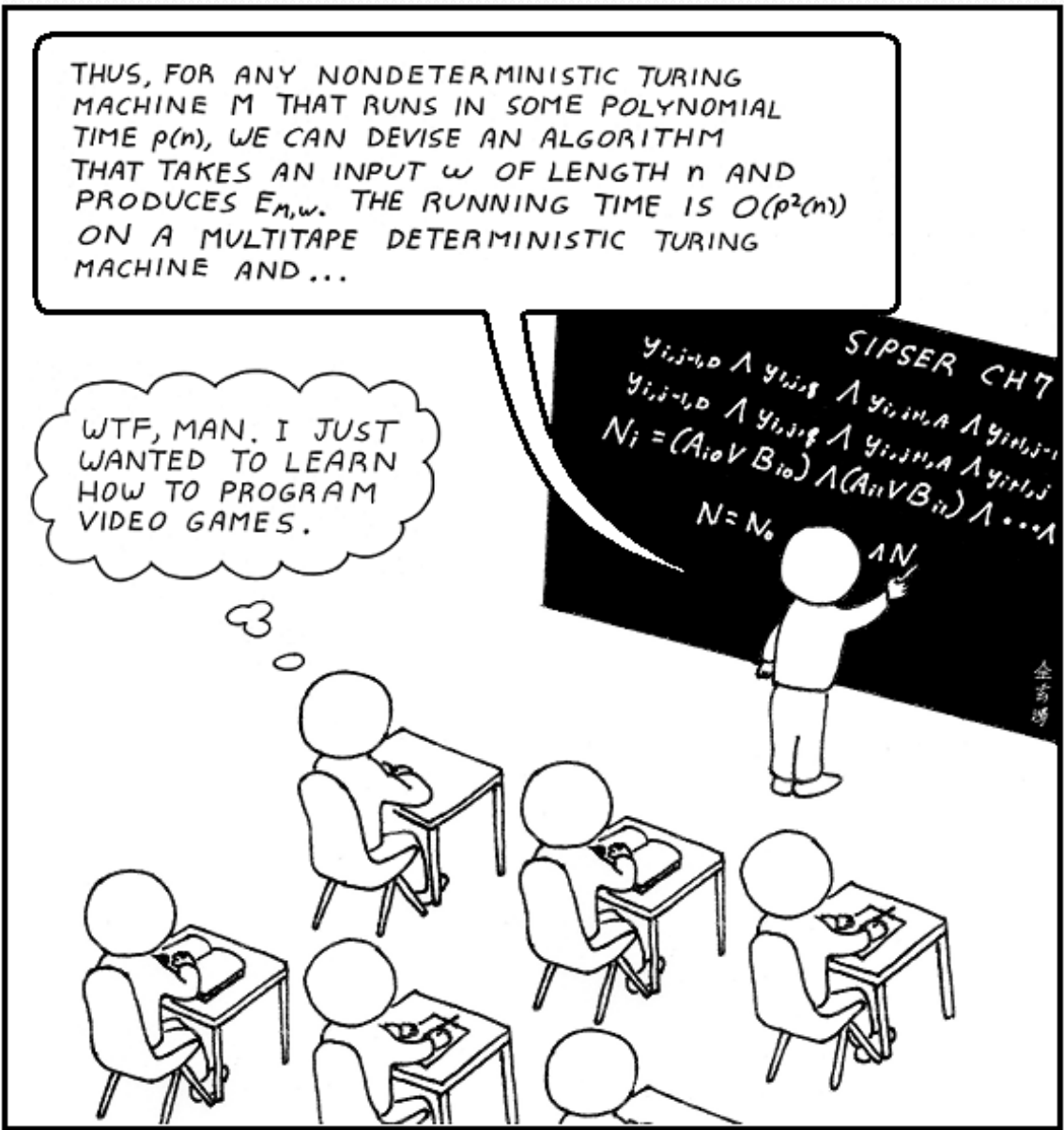
Lecture Q-1: stacks and queues

**reading: [appendix Q](#)**

THUS, FOR ANY NONDETERMINISTIC TURING MACHINE  $M$  THAT RUNS IN SOME POLYNOMIAL TIME  $p(n)$ , WE CAN DEVISE AN ALGORITHM THAT TAKES AN INPUT  $w$  OF LENGTH  $n$  AND PRODUCES  $E_{M,w}$ . THE RUNNING TIME IS  $O(p^2(n))$  ON A MULTITAPE DETERMINISTIC TURING MACHINE AND...

WTF, MAN. I JUST WANTED TO LEARN HOW TO PROGRAM VIDEO GAMES.

SIPSER CH7  
 $y_{i,j-1,0} \wedge y_{i,j,0} \wedge y_{i,j,1} \wedge y_{i,j,2}$   
 $y_{i,j-1,0} \wedge y_{i,j,0} \wedge y_{i,j,1} \wedge y_{i,j,2}$   
 $N_i = (A_{i0} \vee B_{i0}) \wedge (A_{i1} \vee B_{i1}) \wedge \dots \wedge$   
 $N = N_0 \wedge \dots \wedge N_n$



# Runtime Efficiency (13.2)

- **efficiency**: measure of computing resources used 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 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.

# Collection efficiency

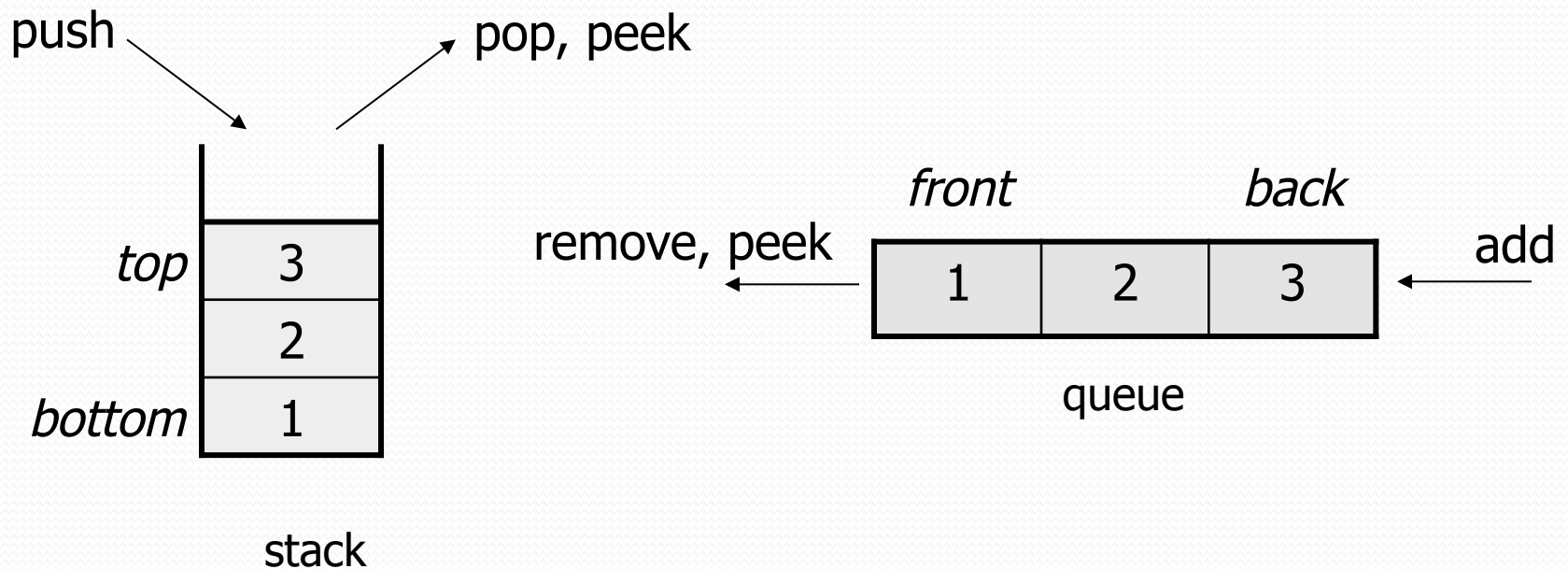
- Efficiency of our `ArrayIntList` or Java's `ArrayList`:

Method	<code>ArrayList</code>
<code>add</code>	$O(1)$
<code>add(index, value)</code>	$O(N)$
<code>indexOf</code>	$O(N)$
<code>get</code>	$O(1)$
<code>remove</code>	$O(N)$
<code>set</code>	$O(1)$
<code>size</code>	$O(1)$

- Which operations should we try to avoid?

# Stacks and queues

- Some collections are constrained so clients can only use optimized operations
  - **stack**: retrieves elements in reverse order as added
  - **queue**: retrieves elements in same order as added



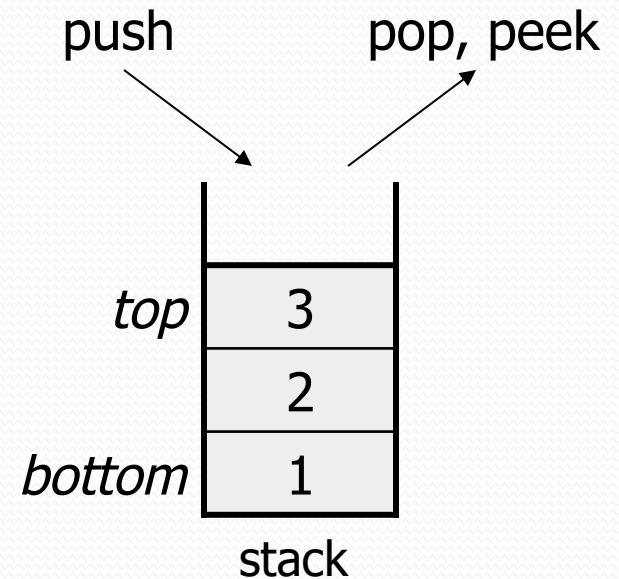
# Abstract data types (ADTs)

- **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

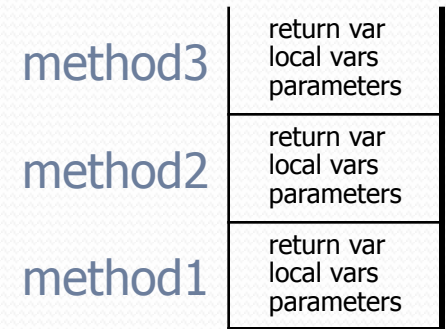
- **stack**: A collection based on the principle of adding elements and retrieving them in the opposite order.
  - Last-In, First-Out ("LIFO")
  - Elements are stored in order of insertion.
    - We do not think of them as having indexes.
  - Client can only add/remove/examine the last element added (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<String> s = new Stack<String> ();  
s.push ("a");  
s.push ("b");  
s.push ("c"); // bottom ["a", "b", "c"] top  
System.out.println (s.pop ()); // "c"
```

- Stack has other methods that are off-limits (not efficient)

# Collections of primitives

- The type parameter specified when creating a collection (e.g. `ArrayList`, `Stack`, `Queue`) must be an object type

```
// illegal -- int cannot be a type parameter
```

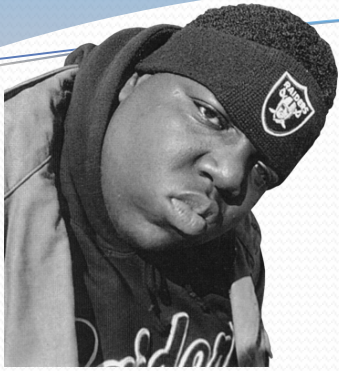
```
Stack<int> s = new Stack<int>();
```

```
ArrayList<int> list = new ArrayList<int>();
```

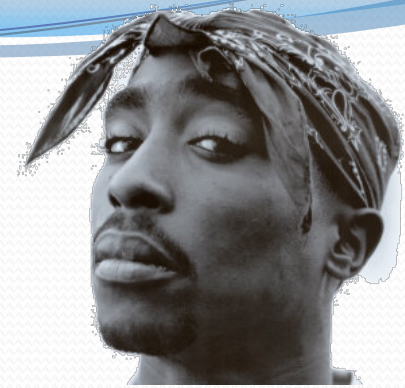
- Primitive types need to be "wrapped" in objects

```
// creates a stack of ints
```

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



# Wrapper classes



Primitive Type	Wrapper Type
int	Integer
double	Double
char	Character
boolean	Boolean

- Wrapper objects have a single field of a primitive type
- The collection can be used with familiar primitives:

```
ArrayList<Double> grades = new ArrayList<Double>();  
grades.add(3.2);  
grades.add(2.7);  
...  
double myGrade = grades.get(0);
```

# Stack limitations/idioms

- 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 pull elements out of the stack one at a time.
  - common idiom: Pop each element until the stack is empty.

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

# 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?

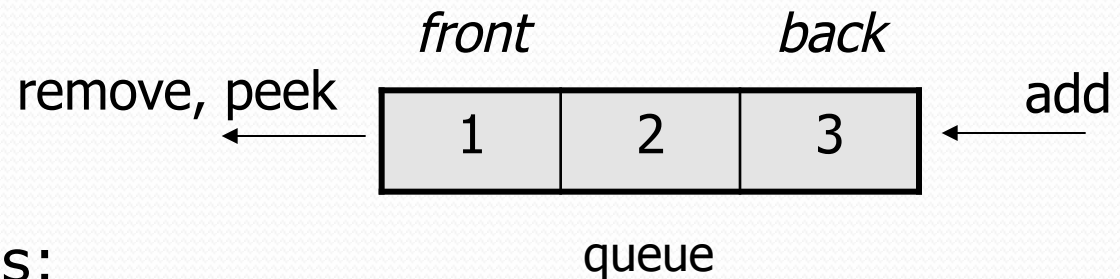
# What happened to my stack?

- 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]
```

# 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`