Building Java Programs

Chapter 14

stacks and queues

reading: 14.1-14.4

Warm up! pollev.com/cse143

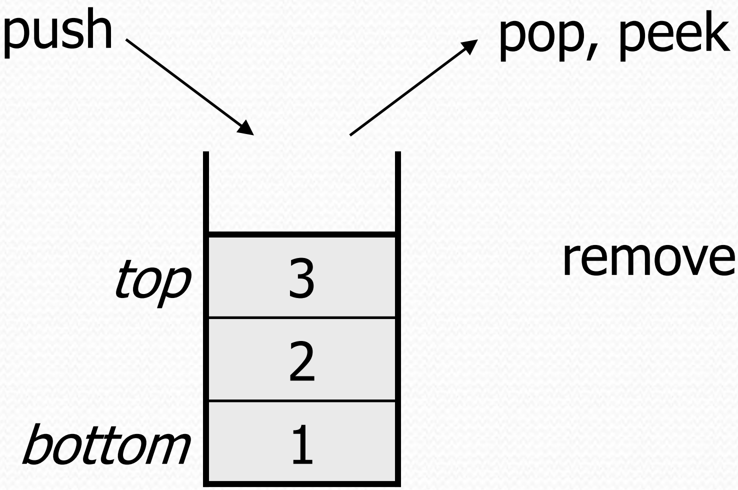
Abstract data types (ADTs)

- **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 the collections 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 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



push

pop, peek

top

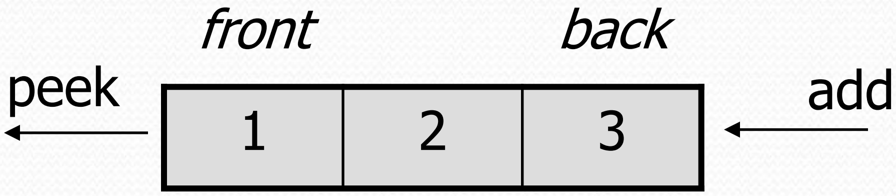
3

2

bottom

1

stack



remove, peek

front

back

1

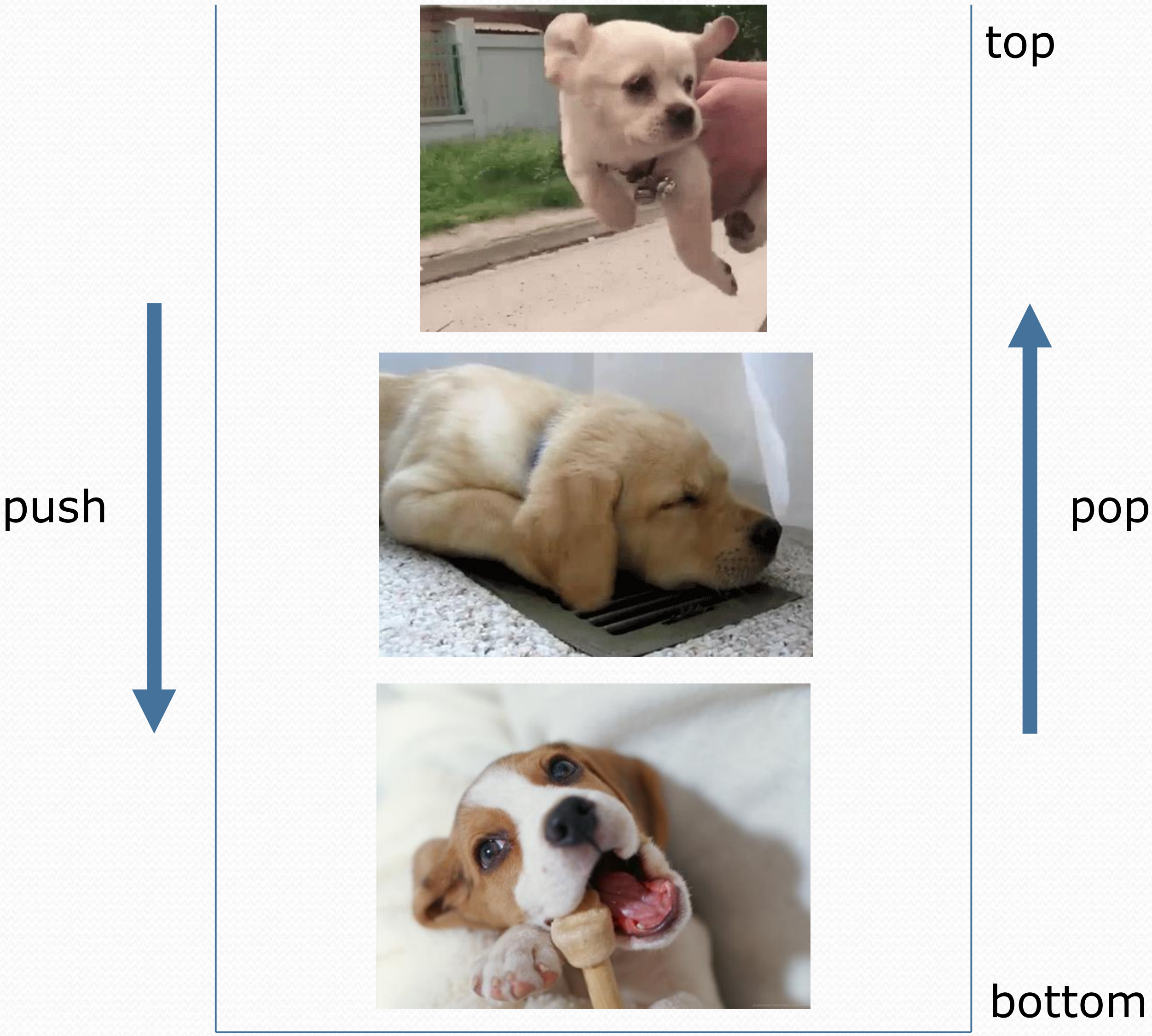
2

3

add

queue

Stack Example



top

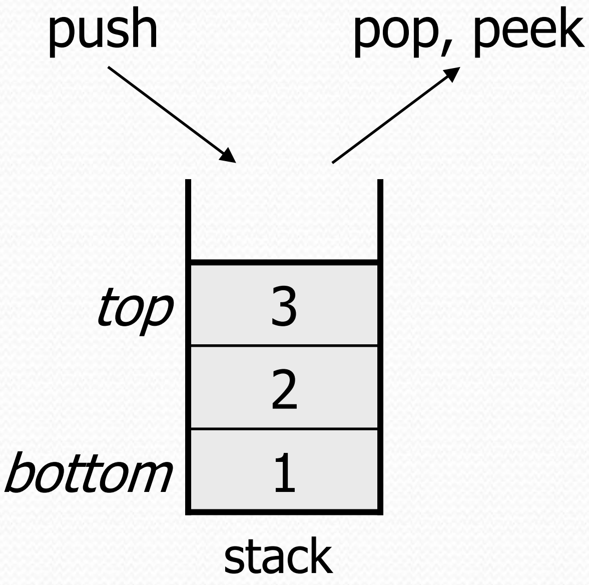
push

pop

bottom

Stacks

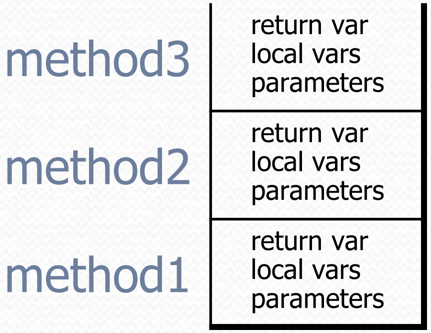
- **stack**: A collection based on the principle of adding elements and retrieving them in the opposite order.
  - Last-In, First-Out ("LIFO")
  - Elements are stored in order of insertion.
    - We do not think of them as having indexes.
  - Client can only add/remove/examine the last element added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



method3: return var, local vars, parameters

method2: return var, local vars, parameters

method1: return var, local vars, parameters

- 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

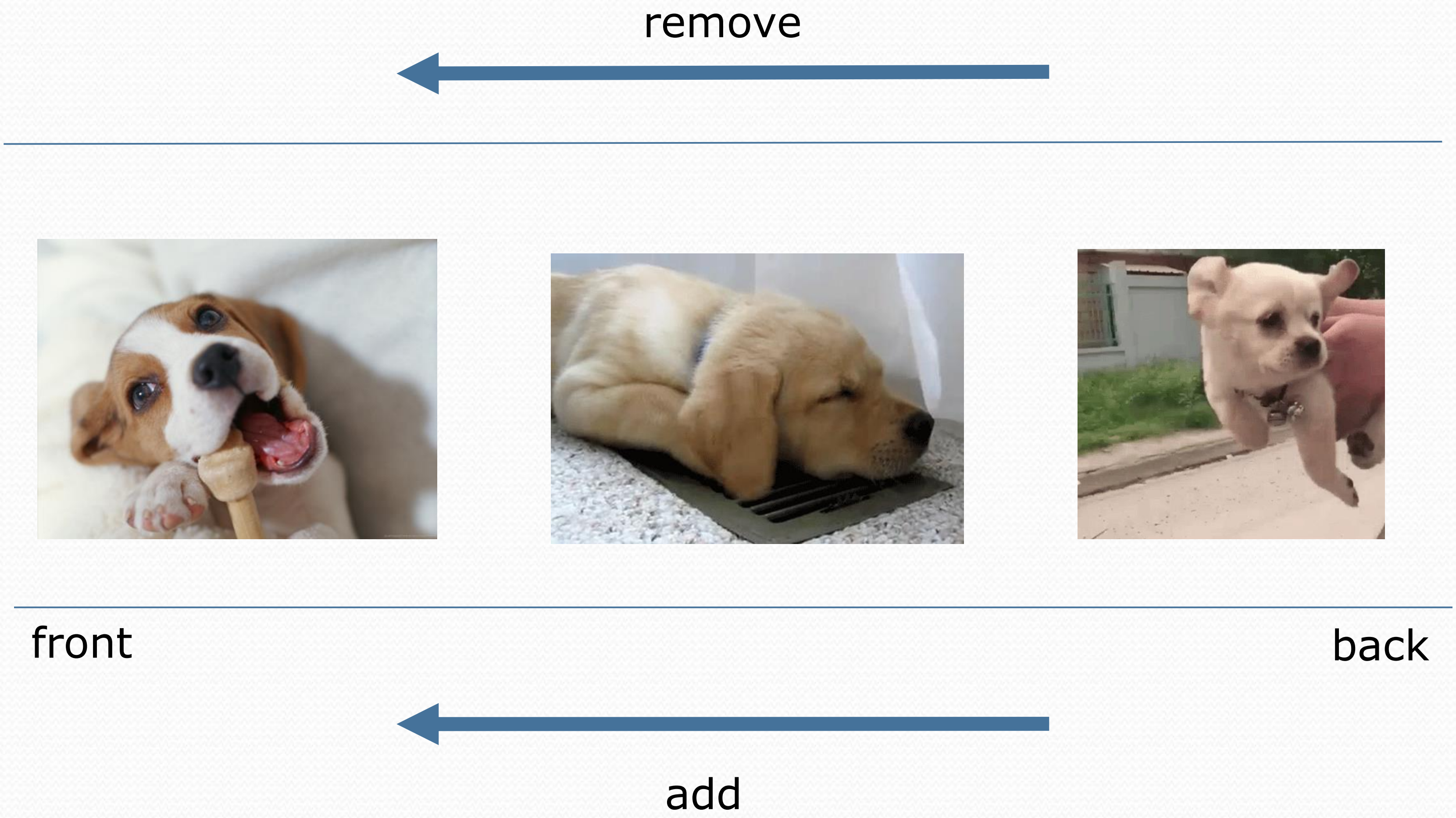
Stack<**E**>()	constructs a new stack with elements of type **E**
push(**value**)	places given value on top of stack
pop()	removes top value from stack and returns it; throws EmptyStackException if stack is empty
peek()	returns top value from stack without removing it; throws EmptyStackException if stack is empty
size()	returns number of elements in stack
isEmpty()	returns true 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)

Queue Example



remove

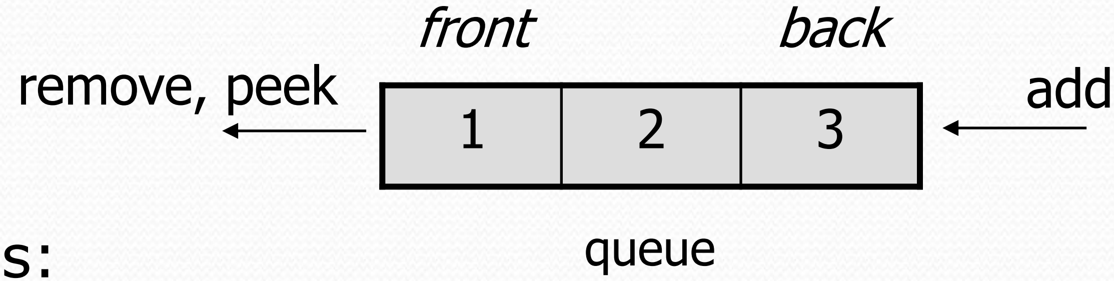
front

back

add

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.



front

back

remove, peek

1	2	3

add

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 `Queue`s

`add(`**value**`)`	places given value at back of queue
`remove()`	removes value from front of queue and returns it; throws a `NoSuchElementException` if queue is empty
`peek()`	returns front value from queue without removing it; returns `null` if queue is empty
`size()`	returns number of elements in queue
`isEmpty()`	returns `true` 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 is because `Queue` is an **interface**