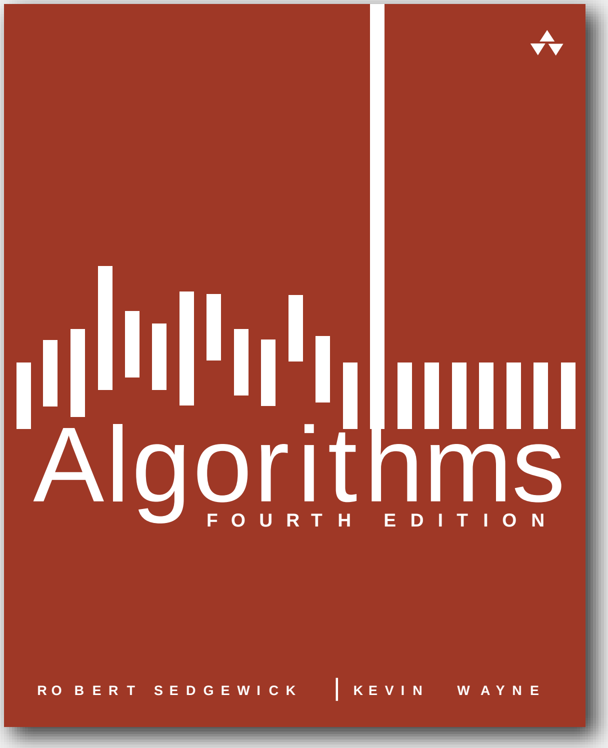
F5: Bags, Queues, and Stacks

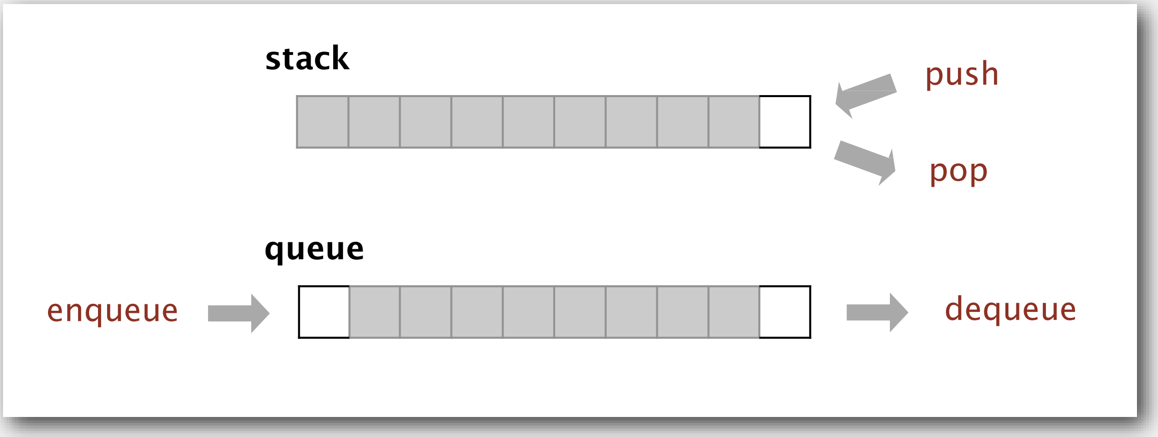


- stacks
- resizing arrays
- generics
- iterators
- applications
- queues

Stacks and queues

Fundamental data types.

- Value: collection of objects.
- Operations: insert, remove, iterate, test if empty.
- Intent is clear when we insert.
- Which item do we remove?



Stack. Examine the item most recently added. ← LIFO = "last in first out"

Queue. Examine the item least recently added. ← FIFO = "first in first out"

Client, implementation, interface

Separate interface and implementation.

Ex: stack, queue, bag, priority queue, symbol table, union-find, ….

Benefits.

- Client can't know details of implementation ⇒ client has many implementation from which to choose.
- Implementation can't know details of client needs ⇒ many clients can re-use the same implementation.
- Design: creates modular, reusable libraries.
- Performance: use optimized implementation where it matters.

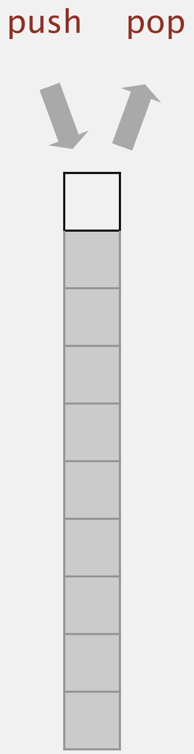
Client: program using operations defined in interface.
Implementation: actual code implementing operations.
Interface: description of data type, basic operations.

- stacks
- resizing arrays
- generics
- iterators
- applications
- queues

Stack API

Warmup API. Stack of strings data type.

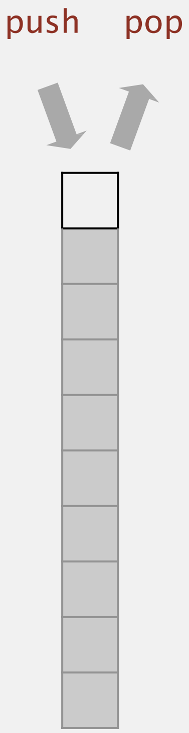
public class StackOfStrings		
	`StackOfStrings()`	*create an empty stack*
`void`	`push(String s)`	*insert a new item onto stack*
`String`	`pop()`	*remove and return the item most recently added*
`boolean`	`isEmpty()`	*is the stack empty?*
`int`	`size()`	*number of items on the stack*



Stack test client

Read strings from standard input.

- If string equals "-", pop string from stack and print.
- Otherwise, push string onto stack.



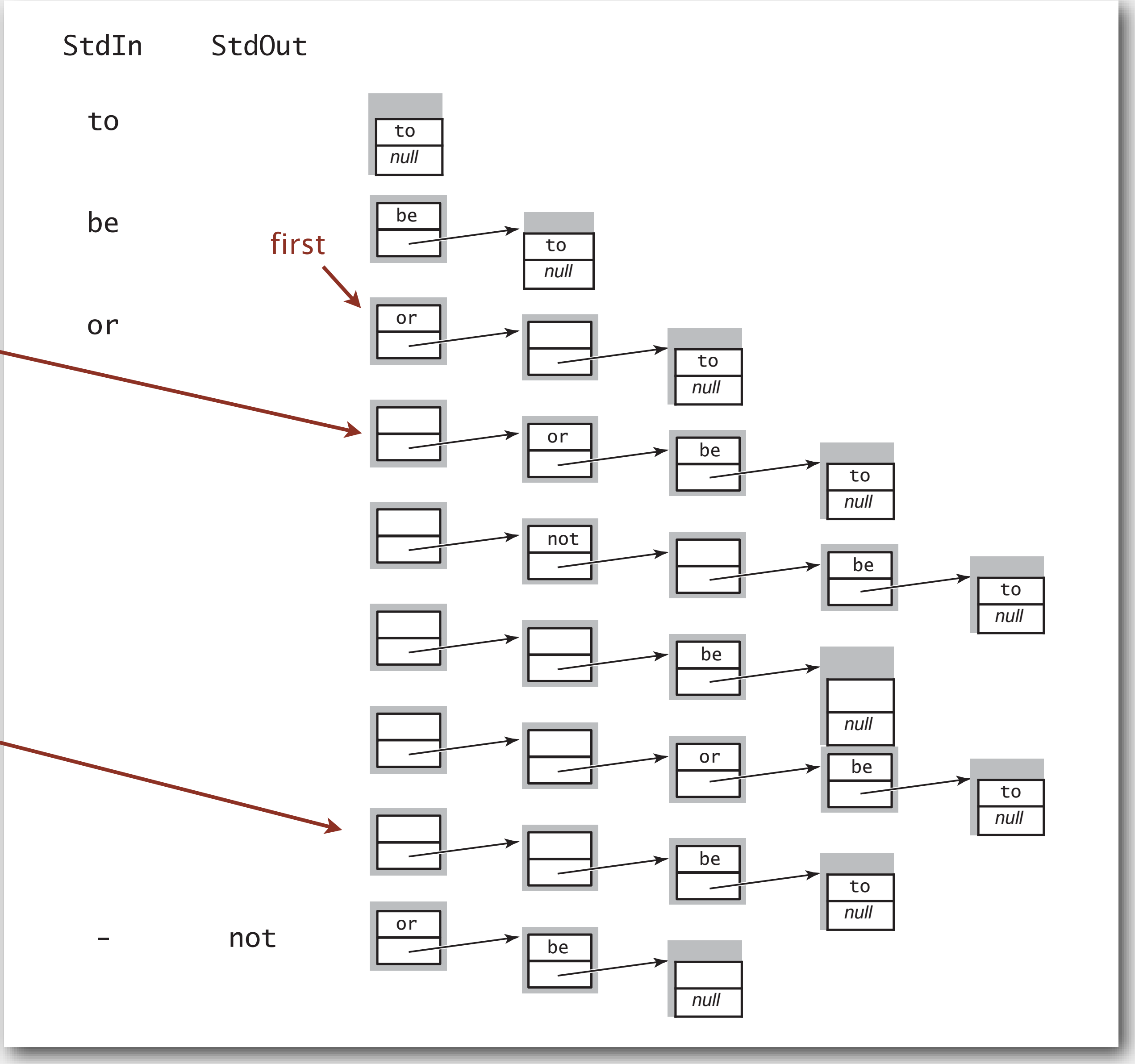
```
public static void main(String[] args)
{
   StackOfStrings stack = new StackOfStrings();
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (item.equals("-")) StdOut.print(stack.pop());
      else                  stack.push(item);
   }
}
```

```
% more tobe.txt
to be or not to - be - - that - - - is

% java StackOfStrings < tobe.txt
to be not that or be
```

Stack: linked-list representation

Stack: linked-list representation

Maintain pointer to first node in a linked list; insert/remove from front.



insert at front of linked list

remove from front

Stack **pop**: linked-list implementation

inner class

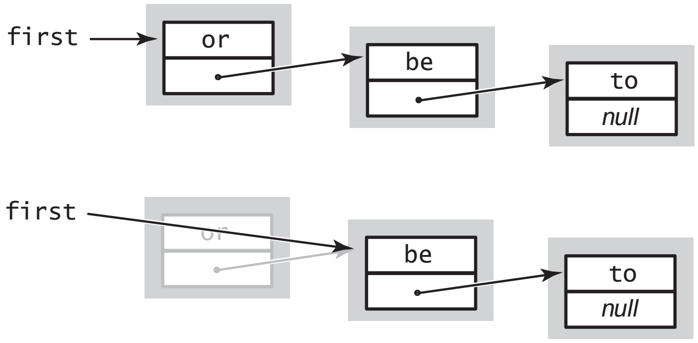
```
private class Node
{
    String item;
    Node next;
}
```

save item to return

```
String item = first.item;
```

delete first node

```
first = first.next;
```



return saved item

```
return item;
```

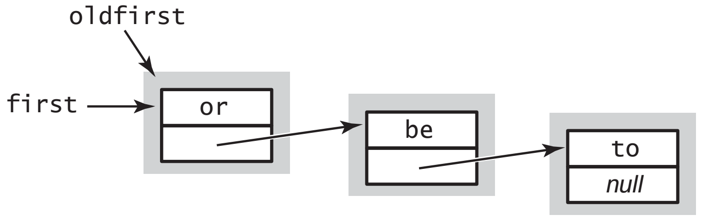
Stack **push**: linked-list implementation

inner class

```
private class Node
{
    String item;
    Node next;
}
```

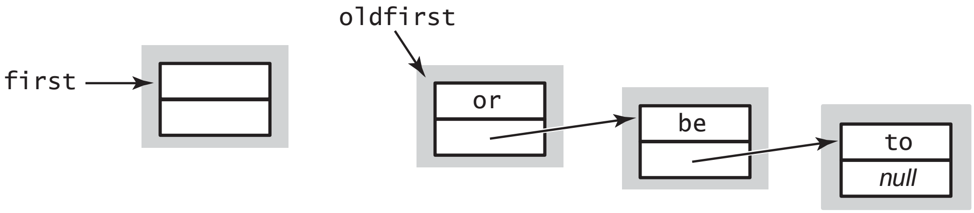
save a link to the list

```
Node oldfirst = first;
```



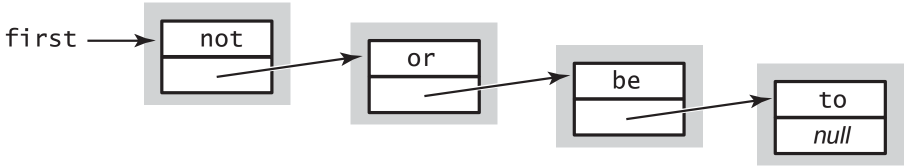
create a new node for the beginning

```
first = new Node();
```



set the instance variables in the new node

```
first.item = "not";
first.next = oldfirst;
```



Stack: linked-list implementation in Java

```
public class LinkedStackOfStrings
{
   private Node first = null;

   private class Node
   {
      String item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(String item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public String pop()
   {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

inner class

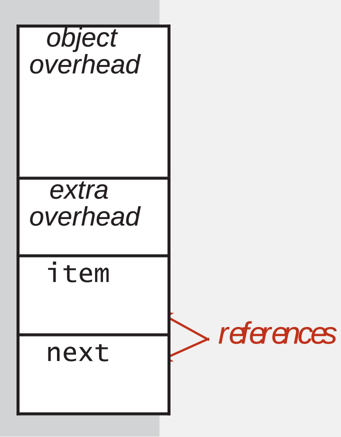
Stack: linked-list implementation performance

Proposition. Every operation takes constant time in the worst case.

Proposition. A stack with N items uses $\sim 40\ N$ bytes.

inner class

```
private class Node
{
   String item;
   Node next;
}
```



16 bytes (object overhead)

8 bytes (inner class extra overhead)

8 bytes (reference to String)

8 bytes (reference to Node)

40 bytes per stack node

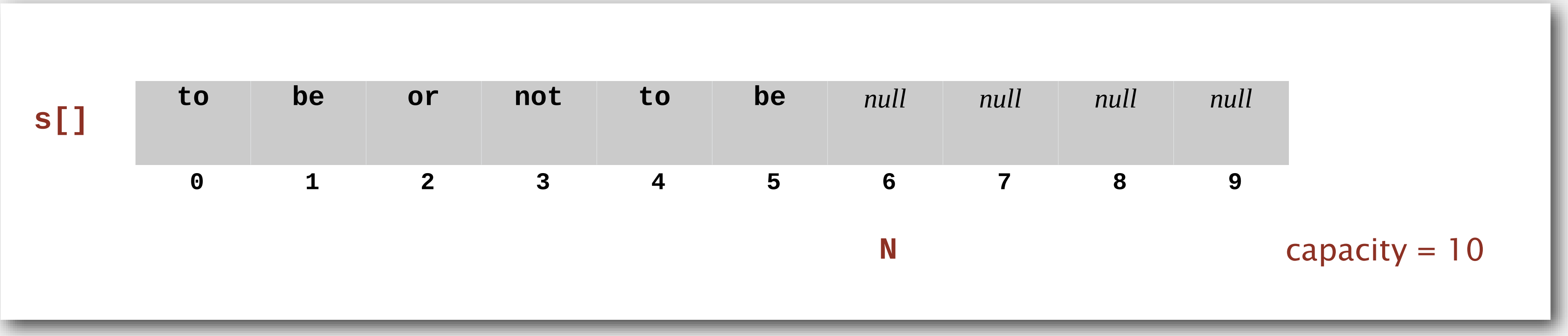
Remark. Analysis includes memory for the stack
(but not the strings themselves, which the client owns).

Stack: array implementation

Stack: array implementation

Array implementation of a stack.

- Use array `s[]` to store `N` items on stack.
- `push()`: add new item at `s[N]`.
- `pop()`: remove item from `s[N-1]`.



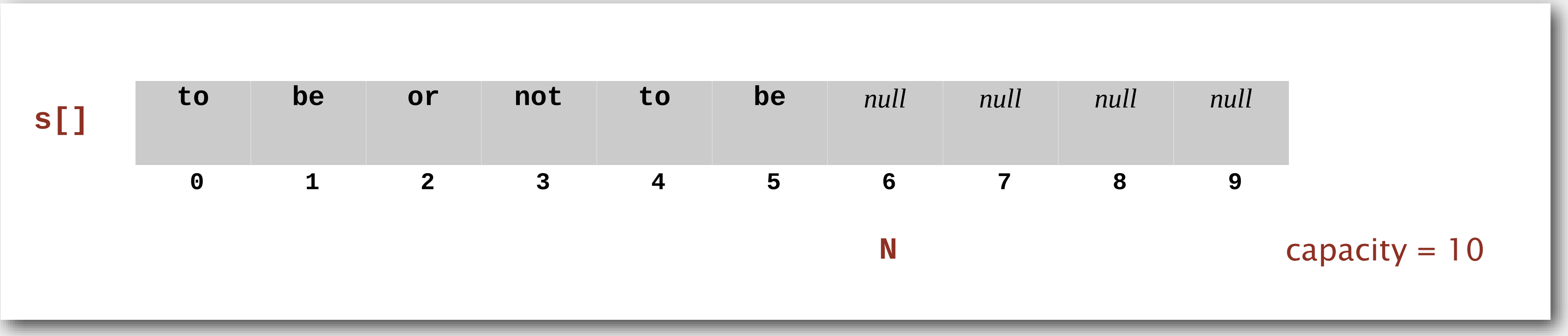
`s[]`	to	be	or	not	to	be	*null*	*null*	*null*	*null*
	0	1	2	3	4	5	6	7	8	9

Defect. ???

Stack: array implementation

Array implementation of a stack.

- Use array `s[]` to store `N` items on stack.
- `push()`: add new item at `s[N]`.
- `pop()`: remove item from `s[N-1]`.



`s[]`	to	be	or	not	to	be	*null*	*null*	*null*	*null*
	0	1	2	3	4	5	6	7	8	9
							N			

capacity = 10

Defect. Stack overflows when `N` exceeds capacity. [stay tuned]

Stack: array implementation

a cheat
(stay tuned)

```
public class FixedCapacityStackOfStrings
{
   private String[] s;
   private int N = 0;

   public FixedCapacityStackOfStrings(int capacity)
   {  s = new String[capacity];  }

   public boolean isEmpty()
   {  return N == 0;  }

   public void push(String item)
   {  s[N++] = item;  }

   public String pop()
   {  return s[--N];  }
}
```

use to index into array;
then increment N

decrement N;
then use to index into array

Stack considerations

Overflow and underflow.

- Underflow: throw exception if pop from an empty stack.
- Overflow: use resizing array for array implementation. [stay tuned]

Loitering. Holding a reference to an object when it is no longer needed.

```
public String pop()
{  return s[--N];  }
```

loitering

```
public String pop()
{
   String item = s[--N];
   s[N] = null;
   return item;
}
```

this version avoids "loitering":
garbage collector can reclaim memory
only if no outstanding references

Null items. We allow null items to be inserted.

- stacks
- **resizing arrays**
- generics
- iterators
- applications
- queues

Stack: resizing-array implementation

Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.

- **`push()`**: increase size of array **`s[]`** by 1.
- **`pop()`**: decrease size of array **`s[]`** by 1.

Stack: resizing-array implementation

Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.

- **push()**: increase size of array **s[]** by 1.
- **pop()**: decrease size of array **s[]** by 1.

Too expensive.

- Need to copy all item to a new array.
- Inserting first N items takes time proportional to $1 + 2 + \ldots + N \sim N^2 / 2$.

↑
infeasible for large N

Challenge. Ensure that array resizing happens infrequently.

Stack: resizing-array implementation

Q. How to grow array?

"repeated doubling"

A. If array is full, create a new array of twice the size, and copy items.

```
public ResizingArrayStackOfStrings()
{  s = new String[1];  }

public void push(String item)
{
   if (N == s.length) resize(2 * s.length);
   s[N++] = item;
}

private void resize(int capacity)
{
   String[] copy = new String[capacity];
   for (int i = 0; i < N; i++)
      copy[i] = s[i];
   s = copy;
}
```

see next slide

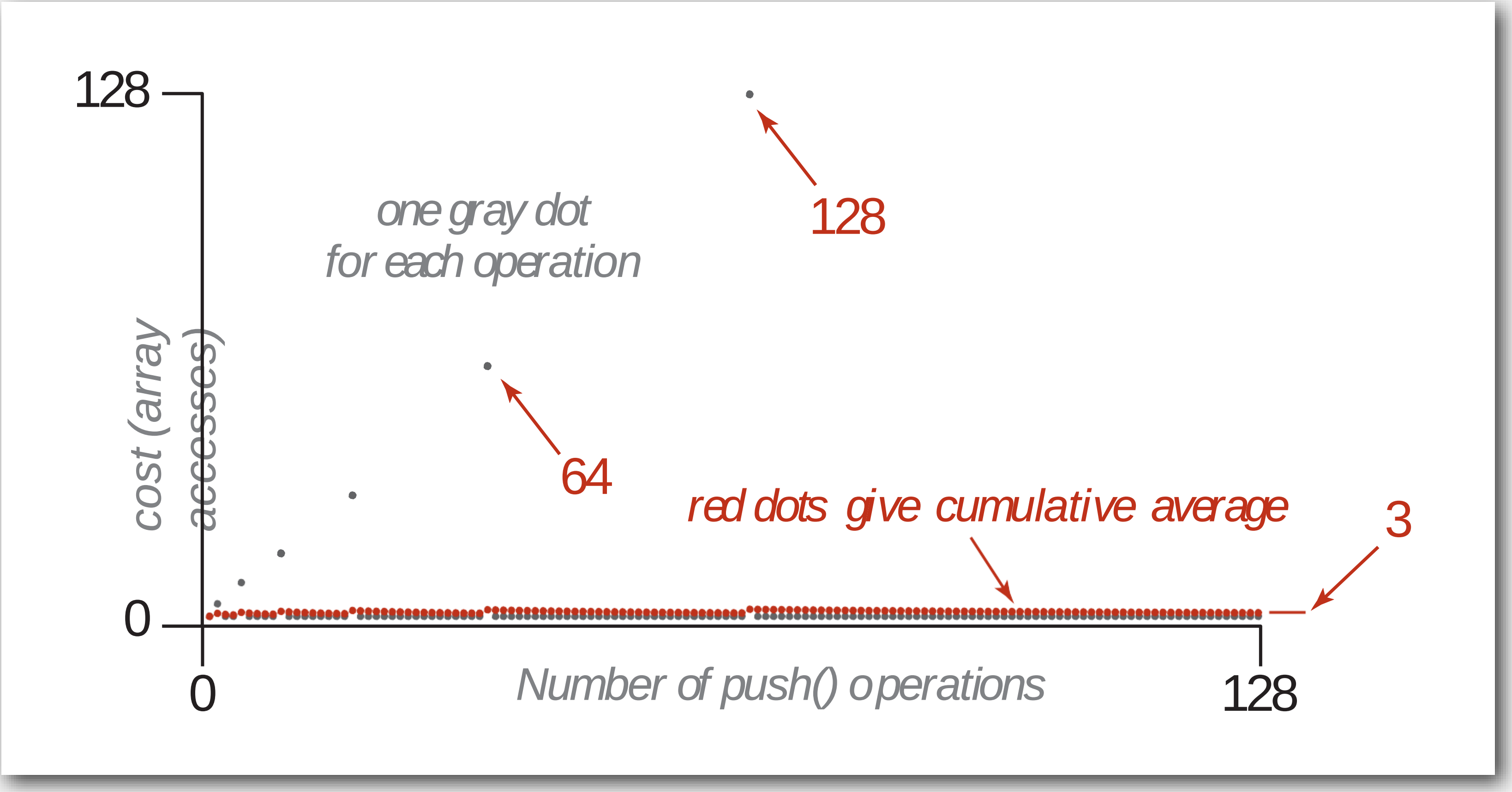
Consequence. Inserting first N items takes time proportional to N (not N^2).

Stack: amortized cost of adding to a stack

Cost of inserting first N items. $N + (2 + 4 + 8 + \ldots + N) \sim 3N.$

1 array accesses per push

k array accesses to double to size k (ignoring cost to create new array)



Stack: resizing-array implementation

Q. How to shrink array?

First try.

- **`push()`**: double size of array **`s[]`** when array is full.
- **`pop()`**: halve size of array **`s[]`** when array is one-half full.

Stack: resizing-array implementation

Q. How to shrink array?

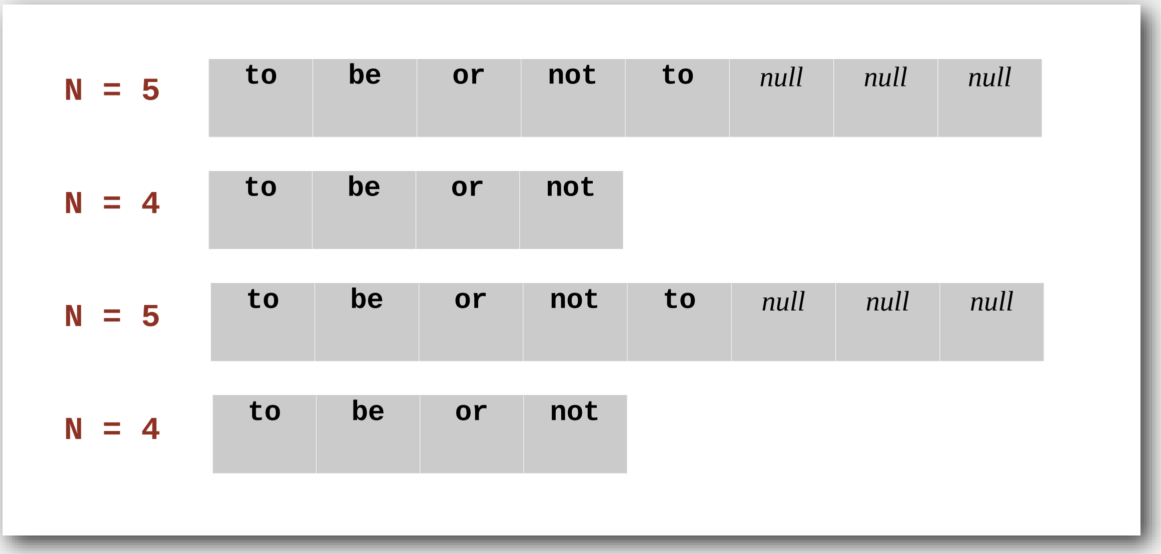
First try.

- **push()**: double size of array **s[]** when array is full.
- **pop()**: halve size of array **s[]** when array is one-half full.

Too expensive in worst case.

"thrashing"

- Consider push-pop-push-pop-... sequence when array is full.
- Each operation takes time proportional to N.



N = 5	to	be	or	not	to	*null*	*null*	*null*
N = 4	to	be	or	not				
N = 5	to	be	or	not	to	*null*	*null*	*null*
N = 4	to	be	or	not				

Stack: resizing-array implementation

Q. How to shrink array?

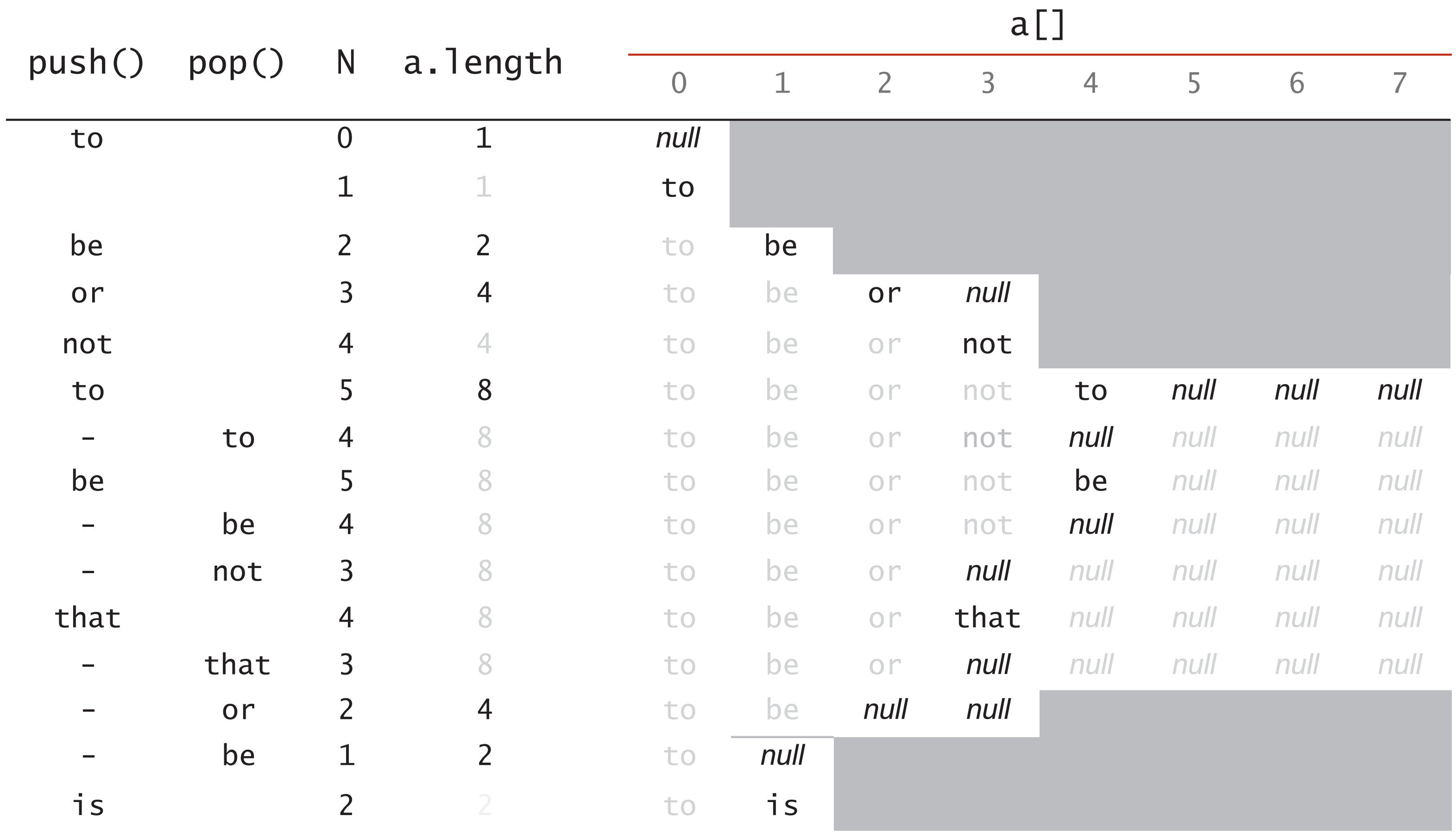
Efficient solution.

- **push()**: double size of array **s[]** when array is full.
- **pop()**: halve size of array **s[]** when array is one-quarter full.

```
public String pop()
{
   String item = s[--N];
   s[N] = null;
   if (N > 0 && N == s.length/4) resize(s.length/2);
   return item;
}
```

Invariant. Array is between 25% and 100% full.

Stack: resizing-array implementation trace



push()	pop()	N	a.length	a[]							
				0	1	2	3	4	5	6	7
to		0	1	*null*							
		1	1	to							
be		2	2	to	be						
or		3	4	to	be	or	*null*				
not		4	4	to	be	or	not				
to		5	8	to	be	or	not	to	*null*	*null*	*null*
-	to	4	8	to	be	or	not	*null*	*null*	*null*	*null*
be		5	8	to	be	or	not	be	*null*	*null*	*null*
-	be	4	8	to	be	or	not	*null*	*null*	*null*	*null*
-	not	3	8	to	be	or	*null*	*null*	*null*	*null*	*null*
that		4	8	to	be	or	that	*null*	*null*	*null*	*null*
-	that	3	8	to	be	or	*null*	*null*	*null*	*null*	*null*
-	or	2	4	to	be	*null*	*null*				
-	be	1	2	to	*null*						
is		2	2	to	is						

Trace of array resizing during a sequence of push() and pop() operations

Stack resizing-array implementation: performance

Amortized analysis. Average running time per operation over a worst-case sequence of operations.

Proposition. Starting from an empty stack, any sequence of M push and pop operations takes time proportional to M.

	best	worst	amortized
construct	1	1	1
push	1	N	1
pop	1	N	1
size	1	1	1

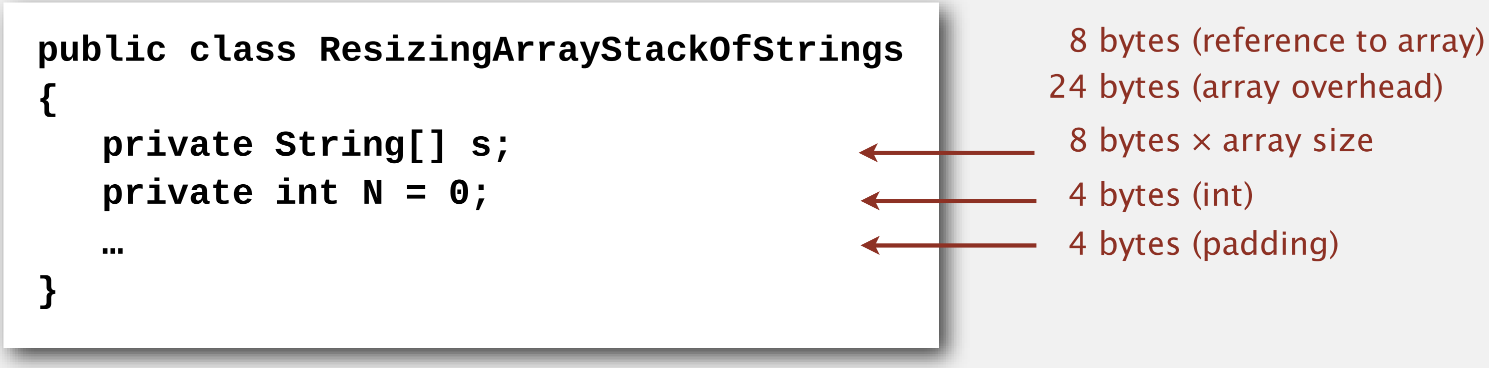
doubling and
halving operations

order of growth of running time
for resizing stack with N items

Stack resizing-array implementation: memory usage

Proposition. Uses between ~ $8N$ and ~ $32N$ bytes to represent a stack with N items.

- ~ $8N$ when full.
- ~ $32N$ when one-quarter full.



```
public class ResizingArrayStackOfStrings
{
   private String[] s;
   private int N = 0;
   …
}
```

8 bytes (reference to array)
24 bytes (array overhead)
8 bytes × array size
4 bytes (int)
4 bytes (padding)

Remark. Analysis includes memory for the stack (but not the strings themselves, which the client owns).

Stack implementations: resizing array vs. linked list

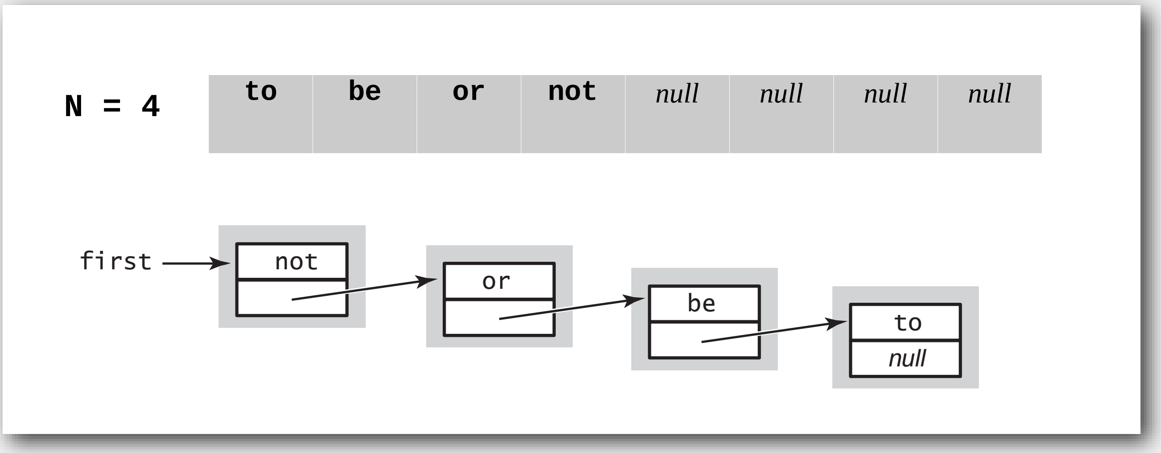
Tradeoffs. Can implement a stack with either a resizing array or a linked list; client can use interchangeably. Which one is better?

Linked-list implementation.

- Every operation takes constant time in the worst case.
- Uses extra time and space to deal with the links.

Resizing-array implementation.

- Every operation takes constant amortized time.
- Less wasted space.



Tölvuorðasafn:
generic : *stofnrænn*

Sem er notaður sem sniðmát til að mynda raunverulega máleiningu fyrir viðeigandi gagnatög í samræmi við reglur um rammtögun.

- stacks
- resizing arrays
- generics
- iterators
- applications
- queues

Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfInts`, `StackOfVans`, ….

Attempt 1. Implement a separate stack class for each type.

- Rewriting code is tedious and error-prone.
- Maintaining cut-and-pasted code is tedious and error-prone.

@#$*! most reasonable approach until Java 1.5.

Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfInts`, `StackOfVans`, ….

Attempt 2. Implement a stack with items of type `Object`.

- Casting is required in client.
- Casting is error-prone: run-time error if types mismatch.

```
StackOfObjects s = new StackOfObjects();
Apple  a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = (Apple) (s.pop());
```

run-time error

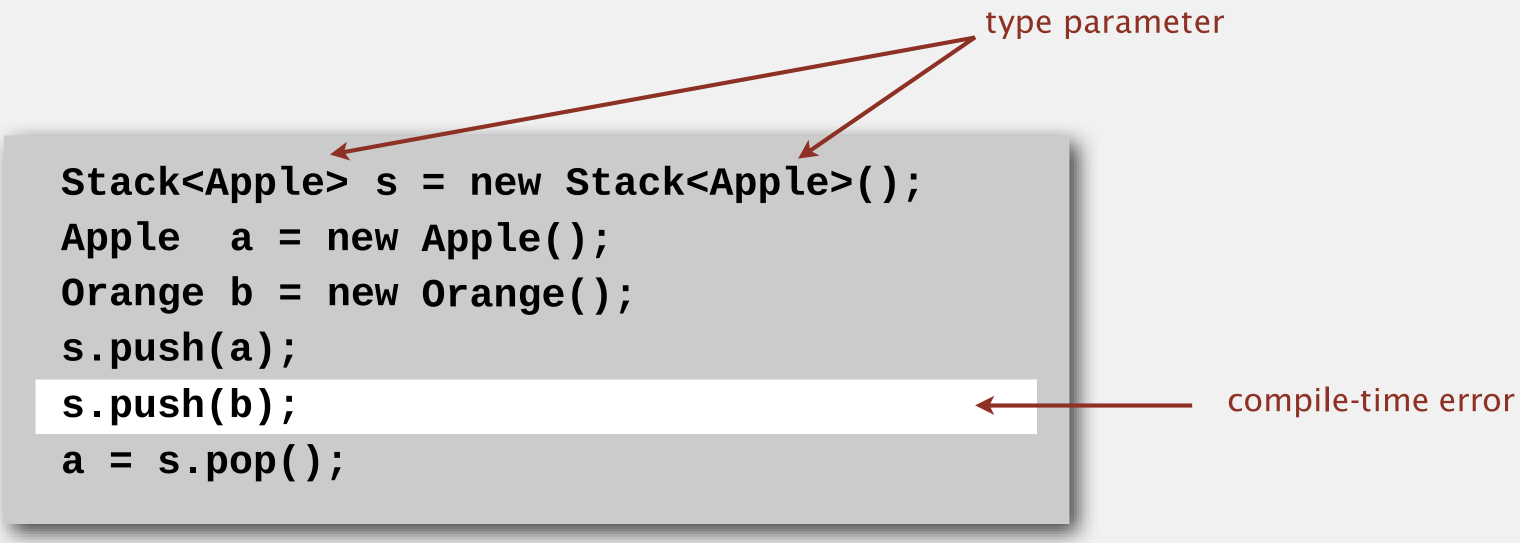
Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfInts`, `StackOfVans`, ….

Attempt 3. Java generics.

- Avoid casting in client.
- Discover type mismatch errors at compile-time instead of run-time.



```
Stack<Apple> s = new Stack<Apple>();
Apple  a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = s.pop();
```

type parameter

compile-time error

Guiding principles. Welcome compile-time errors; avoid run-time errors.

Generic stack: linked-list implementation

```
public class
LinkedStackOfStrings
{
   private Node first = null;

   private class Node
   {
      String item;
   }  Node next;

   public boolean isEmpty()
   {  return first == null;   }

   public void push(String item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public String pop()
   {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

```
public class Stack<Item>
{
   private Node first = null;

   private class Node
   {
      Item item
      Node next
   }

   public boolean isEmpty()
   {  return first == null;   }

   public void push(Item item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public Item pop()
   {
      Item item = first.item;
      first = first.next;
      return item;
   }
}
```

generic type name

Generic stack: array implementation

```
public class FixedCapacityStackOfStrings
{
   private String[] s;
   private int N = 0;

   public ..StackOfStrings(int capacity)
   {  s = new String[capacity];  }

   public boolean isEmpty()
   {  return N == 0;  }

   public void push(String item)
   {  s[N++] = item;  }

   public String pop()
   {  return s[--N];  }
}
```

the way it should be

```
public class FixedCapacityStack<Item>
{
   private Item[] s;
   private int N = 0;

   public FixedCapacityStack(int capacity)
   {  s = new Item[capacity];  }

   public boolean isEmpty()
   {  return N == 0;  }

   public void push(Item item)
   {  s[N++] = item;  }

   public Item pop()
   {  return s[--N];  }
}
```

@#$*! generic array creation not allowed in Java

Generic stack: array implementation

```
public class FixedCapacityStackOfStrings
{
   private String[] s;
   private int N = 0;

   public ..StackOfStrings(int capacity)
   {  s = new String[capacity];  }

   public boolean isEmpty()
   {  return N == 0;  }

   public void push(String item)
   {  s[N++] = item;  }

   public String pop()
   {  return s[--N];  }
}
```

the way it is

```
public class FixedCapacityStack<Item>
{
   private Item[] s;
   private int N = 0;

   public FixedCapacityStack(int capacity)
   {  s = (Item[]) new Object[capacity]; }

   public boolean isEmpty()
   {  return N == 0;  }

   public void push(Item item)
   {  s[N++] = item;  }

   public Item pop()
   {  return s[--N];  }
}
```

the ugly cast

Generic data types: autoboxing

Q. What to do about primitive types?

Generic data types: autoboxing

Q. What to do about primitive types?

Wrapper type.

- Each primitive type has a wrapper object type.
- Ex: `Integer` is wrapper type for `int`.

Autoboxing. Automatic cast between a primitive type and its wrapper.

„Syntactic sugar". Behind-the-scenes casting.

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

s.push(17);         // s.push(new Integer(17));
int a = s.pop();    // int a = s.pop().intValue();
```

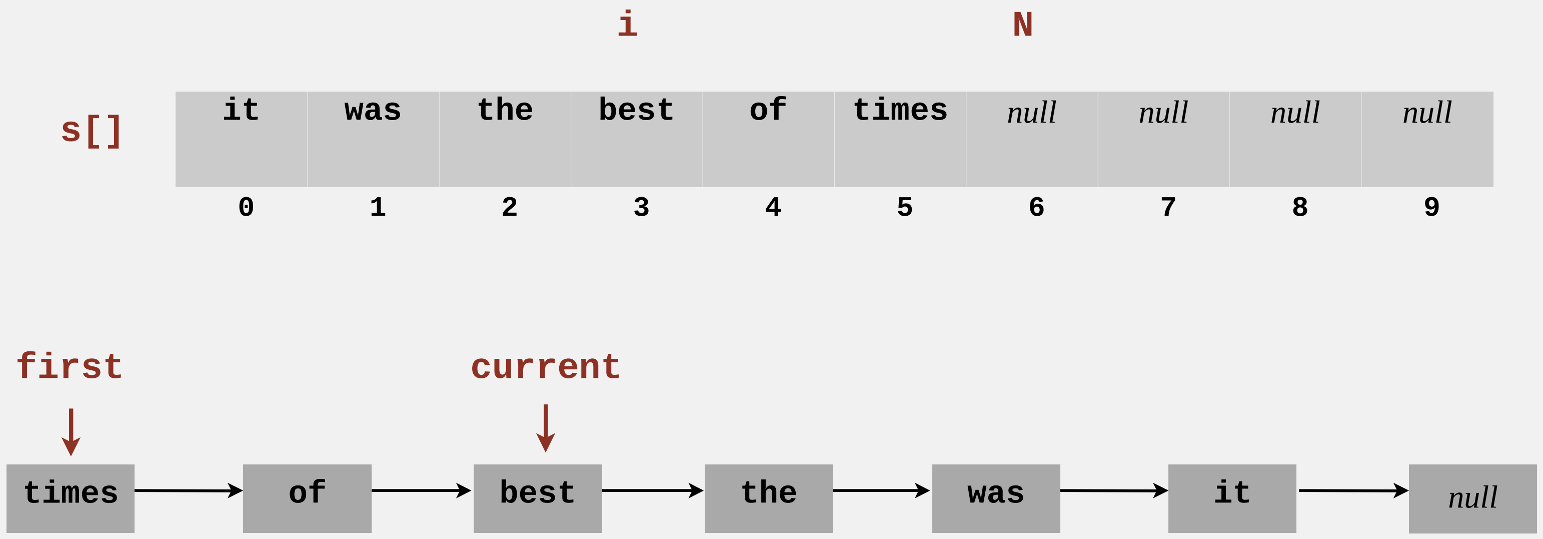
Bottom line. Client code can use generic stack for any type of data.

- stacks
- resizing arrays
- queues
- generics
- iterators
- applications

ísl.: Ítrarar, ítrekarar

Iteration

Design challenge. Support iteration over stack items by client, without revealing the internal representation of the stack.



				i			N			
s[]	it	was	the	best	of	times	*null*	*null*	*null*	*null*
	0	1	2	3	4	5	6	7	8	9

first

current

times → of → best → the → was → it → *null*

Java solution. Make stack implement the `Iterable` interface.

Java Interfaces

Restricted form of multiple inheritance.

Java doesn't allow a class to inherit from multiple parents.

Interfaces are a way around it.

Interface inheritance (subtyping).

- Java provides the interface construct for declaring a relationship between otherwise unrelated classes, by specifying a common set of methods that each implementing class must include.
- Interfaces enable us to write client programs that can manipulate objects of varying types, by invoking common methods from the interface.

Built-in interfaces

- `java.util.Comparable` (see Chapter 2 on Sorting)
- `java.util.Iterable` and `java.util.Iterator` (Here)

Iterators

Q. What is an `Iterable` ?

A. Has a method that returns an `Iterator`.

Iterable interface

```
public interface Iterable<Item>
{
   Iterator<Item> iterator();
}
```

Q. What is an `Iterator` ?

A. Has methods `hasNext()` and `next()`.

Iterator interface

```
public interface Iterator<Item>
{
   boolean hasNext();
   Item next();
   void remove();
}
```

optional; use at your own risk

Q. Why make data structures `Iterable` ?

A. Java supports elegant client code.

"foreach" statement

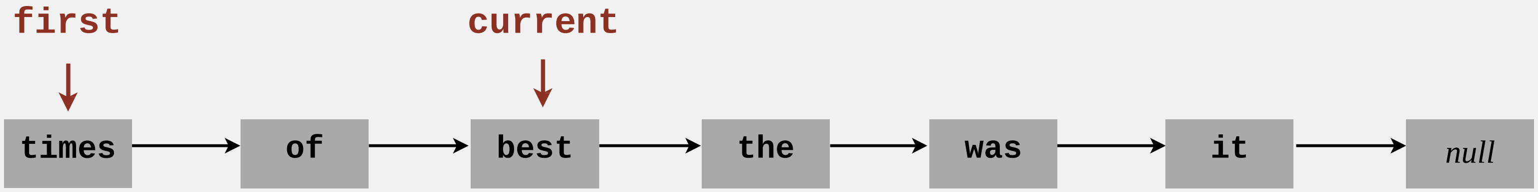
```
for (String s : stack)
   StdOut.println(s);
```

equivalent code

```
Iterator<String> i = stack.iterator();
while (i.hasNext())
{
   String s = i.next();
   StdOut.println(s);
}
```

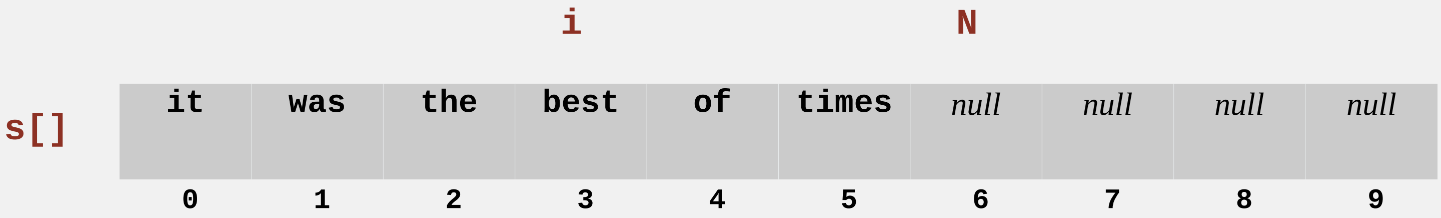
Stack iterator: linked-list implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator() { return new ListIterator(); }

    private class ListIterator implements Iterator<Item>
    {
        private Node current = first;

        public boolean hasNext() {  return current != null;   }
        public void remove()     {  /* not supported */       }
        public Item next()
        {
            Item item = current.item;
            current   = current.next;
            return item;
        }
    }
}
```



Stack iterator: array implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    …

    public Iterator<Item> iterator()
    { return new ReverseArrayIterator(); }

    private class ReverseArrayIterator implements Iterator<Item>
    {
        private int i = N;

        public boolean hasNext() {  return i > 0;          }
        public void remove()     {  /* not supported */   }
        public Item next()       {  return s[--i];         }
    }
}
```

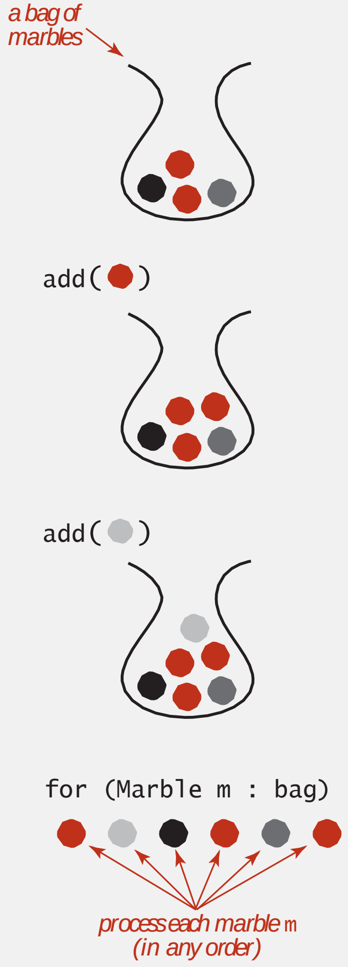


				i			N			
s[]	it	was	the	best	of	times	*null*	*null*	*null*	*null*
	0	1	2	3	4	5	6	7	8	9

Bag API

Main application. Adding items to a collection and iterating (when order doesn't matter).

public class Bag<Item> implements Iterable<Item>		
	`Bag()`	*create an empty bag*
`void`	`add(Item x)`	*insert a new item onto bag*
`int`	`size()`	*number of items in bag*
`Iterable<Item>`	`iterator()`	*iterator for all items in bag*



Implementation. Stack (without pop) or queue (without dequeue).

- stacks
- resizing arrays
- queues
- generics
- iterators
- applications

Java collections library

List interface. `java.util.List` is API for ordered collection of items.

`public interface List<Item> implements Iterable<Item>`		
	`List()`	*create an empty list*
`boolean`	`isEmpty()`	*is the list empty?*
`int`	`size()`	*number of items*
`void`	`add(Item item)`	*append item to the end*
`Item`	`get(int index)`	*return item at given index*
`Item`	`remove(int index)`	*return and delete item at given index*
`boolean`	`contains(Item item)`	*does the list contain the given item?*
`Iteartor<Item>`	`iterator()`	*iterator over all items in the list*
`...`		

Implementations. `java.util.ArrayList` uses resizing array; `java.util.LinkedList` uses linked list.

Java collections library

`java.util.Stack`.

- Supports `push()`, `pop()`, `size()`, `isEmpty()`, and iteration.
- Also implements `java.util.List` interface from previous slide, including, `get()`, `remove()`, and `contains()`.
- Bloated and poorly-designed API (why?) ⇒ don't use.

`java.util.Queue`. An interface, not an implementation of a queue.

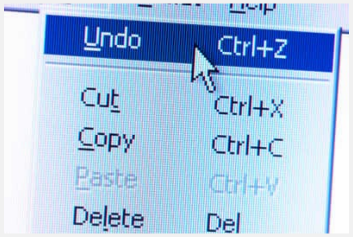
Best practices. Use our implementations of `Stack`, `Queue`, and `Bag`.

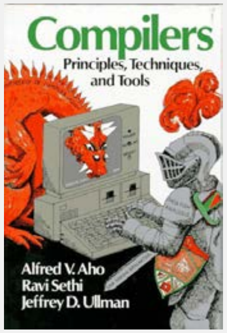
Stack applications

- Parsing in a compiler.
- Java virtual machine.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Implementing function calls in a compiler.
- ...

Java

Adobe® PostScript® 3™





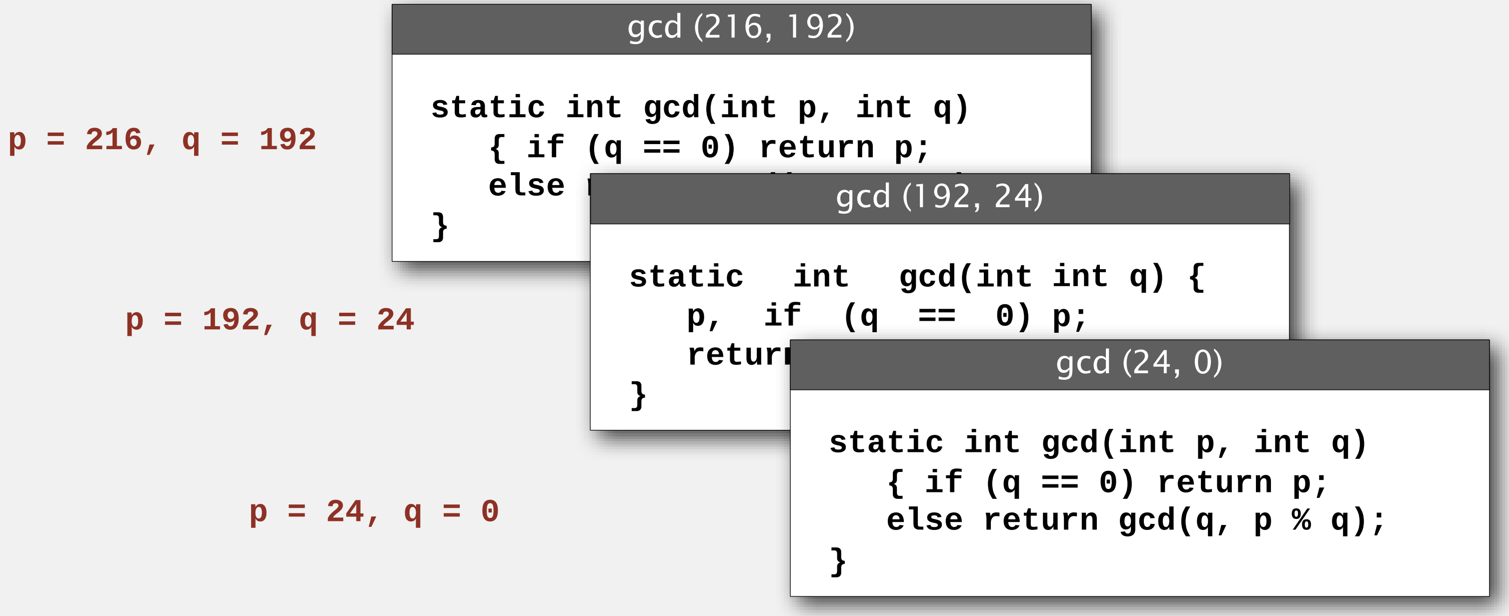
Function calls

How a compiler implements a function.

- Function call: push local environment and return address.
- Return: pop return address and local environment.

Recursive function. Function that calls itself.

Note. Can always use an explicit stack to remove recursion.



p = 216, q = 192

gcd (216, 192)

```
static int gcd(int p, int q)
   { if (q == 0) return p;
     else r
}
```

p = 192, q = 24

gcd (192, 24)

```
static  int  gcd(int int q) {
   p,  if  (q  ==  0) p;
   retur
}
```

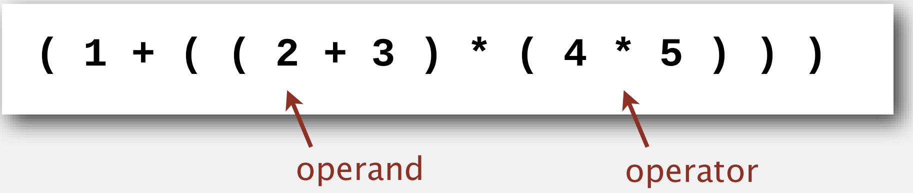
p = 24, q = 0

gcd (24, 0)

```
static int gcd(int p, int q)
   { if (q == 0) return p;
     else return gcd(q, p % q);
}
```

Arithmetic expression evaluation

Goal. Evaluate infix expressions.

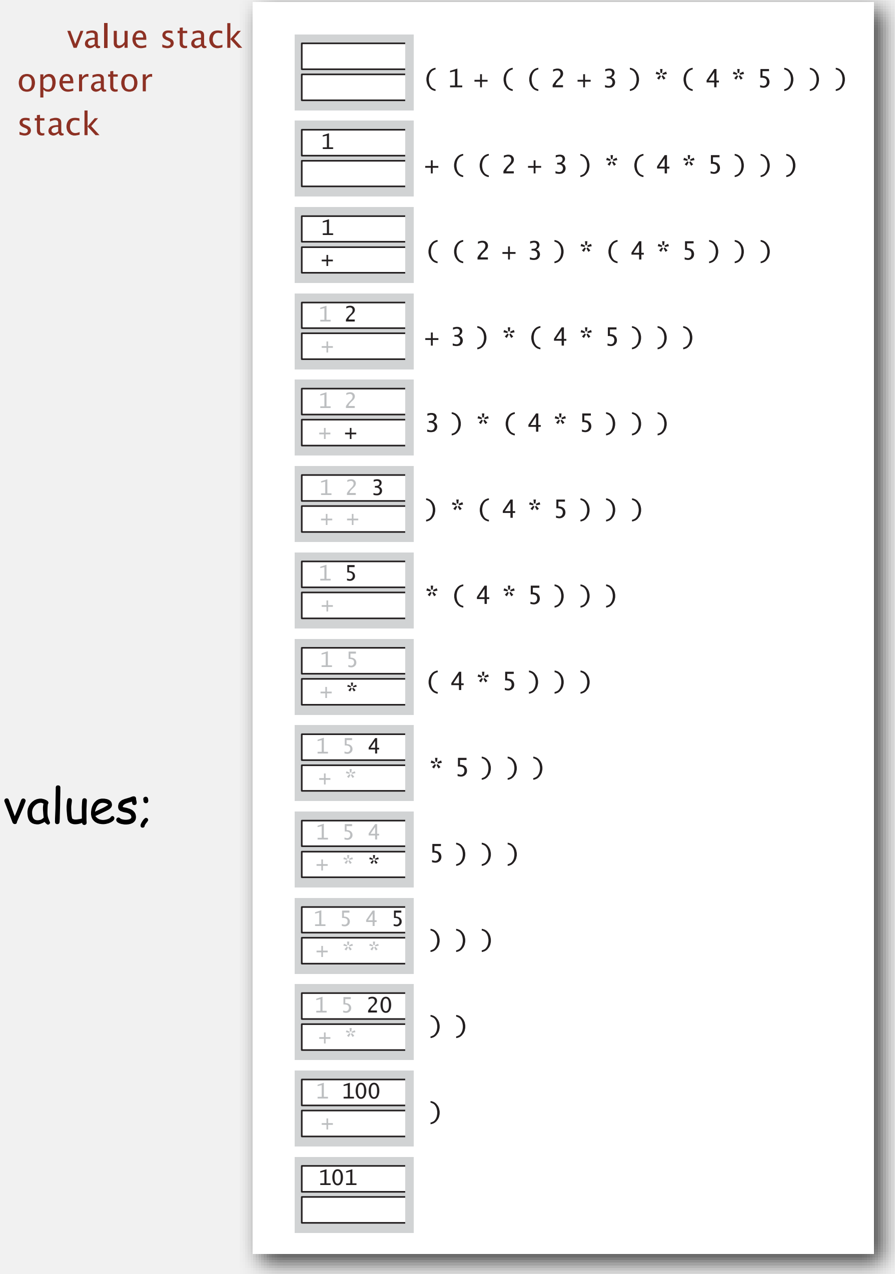


```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

Two-stack algorithm. [E. W. Dijkstra]

- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parenthesis: ignore.
- Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

Context. An interpreter!



Arithmetic expression evaluation demo

Arithmetic expression evaluation

```
public class Evaluate
{
   public static void main(String[] args)
   {
      Stack<String> ops  = new Stack<String>();
      Stack<Double> vals = new Stack<Double>();
      while (!StdIn.isEmpty()) {
         String s = StdIn.readString();
         if      (s.equals("("))               ;
         else if (s.equals("+"))    ops.push(s);
         else if (s.equals("*"))    ops.push(s);
         else if (s.equals(")"))
         {
            String op = ops.pop();
            if      (op.equals("+")) vals.push(vals.pop() + vals.pop());
            else if (op.equals("*")) vals.push(vals.pop() * vals.pop());
         }
         else vals.push(Double.parseDouble(s));
      }
      StdOut.println(vals.pop());
   }
}
```

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
```

Correctness

Q. Why correct?

A. When algorithm encounters an operator surrounded by two values within parentheses, it leaves the result on the value stack.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

as if the original input were:

```
( 1 + ( 5 * ( 4 * 5 ) ) )
```

Repeating the argument:

```
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

Extensions. More ops, precedence order, associativity.

Stack-based programming languages

Observation 1. The 2-stack algorithm computes the same value if the operator occurs after the two values.

```
( 1 ( ( 2 3 + ) ( 4 5 * ) * ) + )
```

Observation 2. All of the parentheses are redundant!

```
1 2 3 + 4 5 * * +
```

Jan Lukasiewicz

Bottom line. Postfix or "reverse Polish" notation.

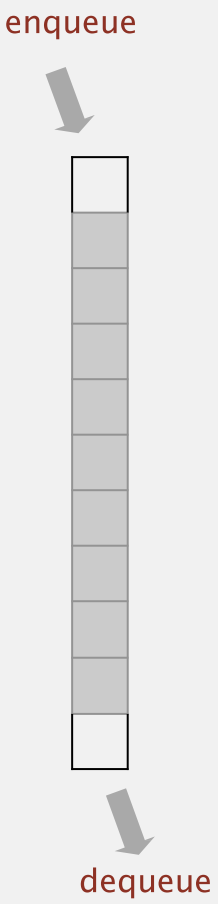
Applications. Postscript, Forth, calculators, Java virtual machine, …

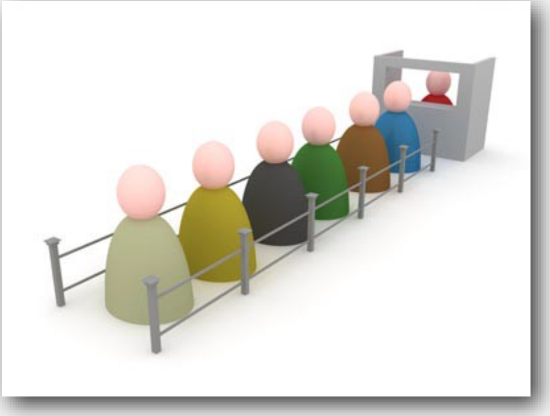
- stacks
- resizing arrays
- generics
- iterators
- applications
- queues

Queue API

`public class QueueOfStrings`

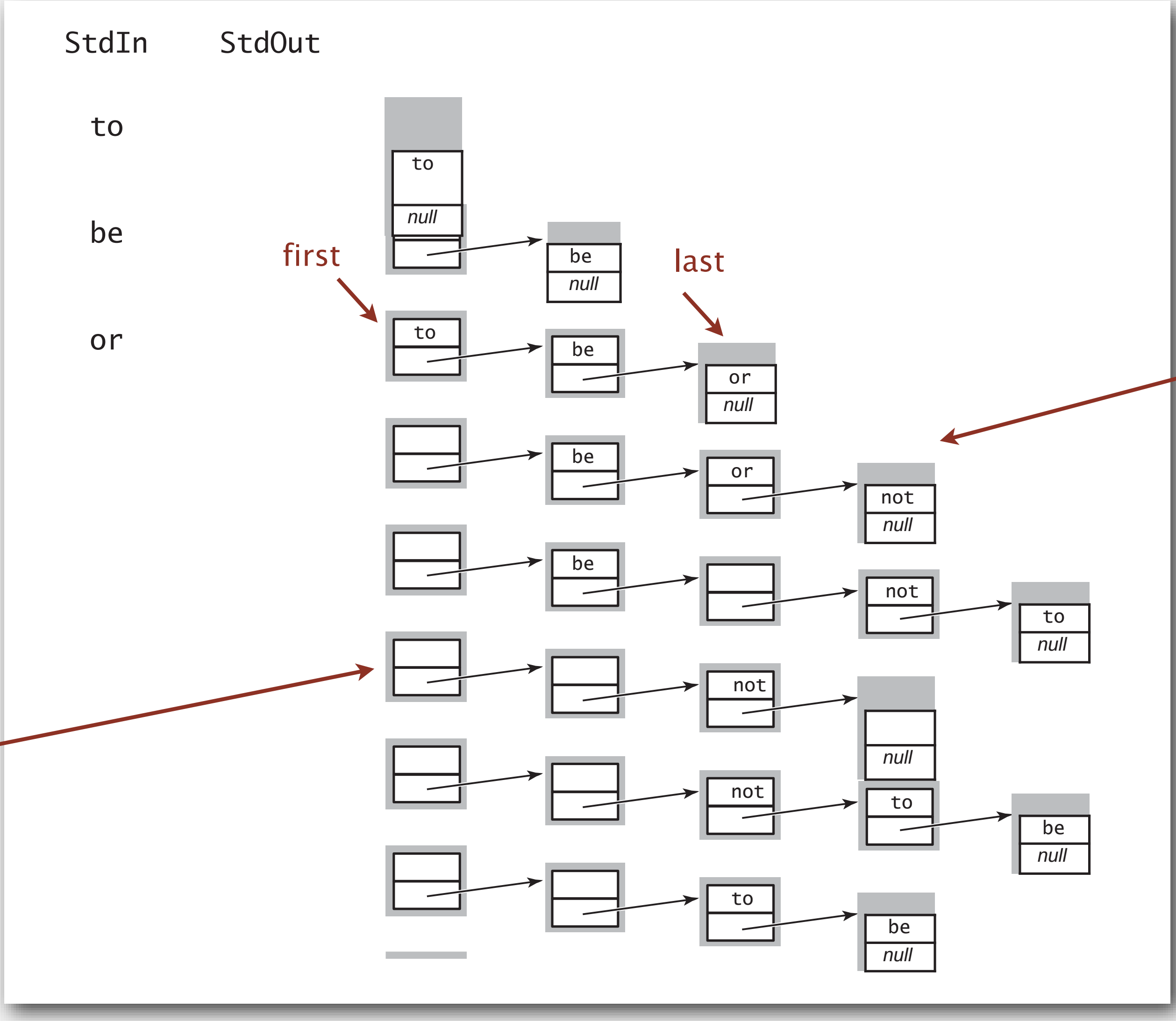
	`QueueOfStrings()`	*create an empty queue*
`void`	`enqueue(String s)`	*insert a new item onto queue*
`String`	`dequeue()`	*remove and return the item least recently added*
`boolean`	`isEmpty()`	*is the queue empty?*
`int`	`size()`	*number of items on the queue*





Queue: linked-list representation

Maintain pointer to first and last nodes in a linked list; insert/remove from opposite ends.



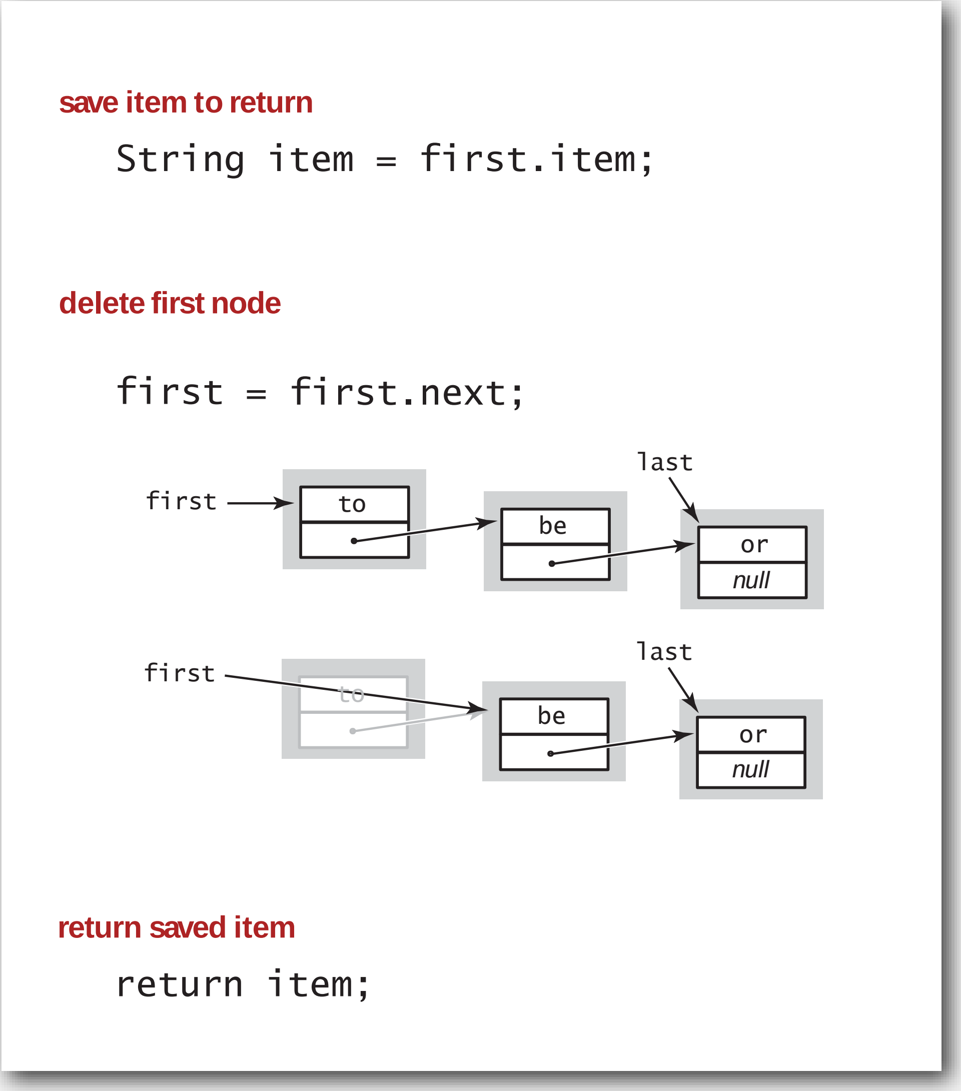
insert at end of linked list

remove from front

Queue dequeue: linked-list implementation

inner class

```
private class Node
{
    String item;
    Node next;
}
```



save item to return

```
String item = first.item;
```

delete first node

```
first = first.next;
```

return saved item

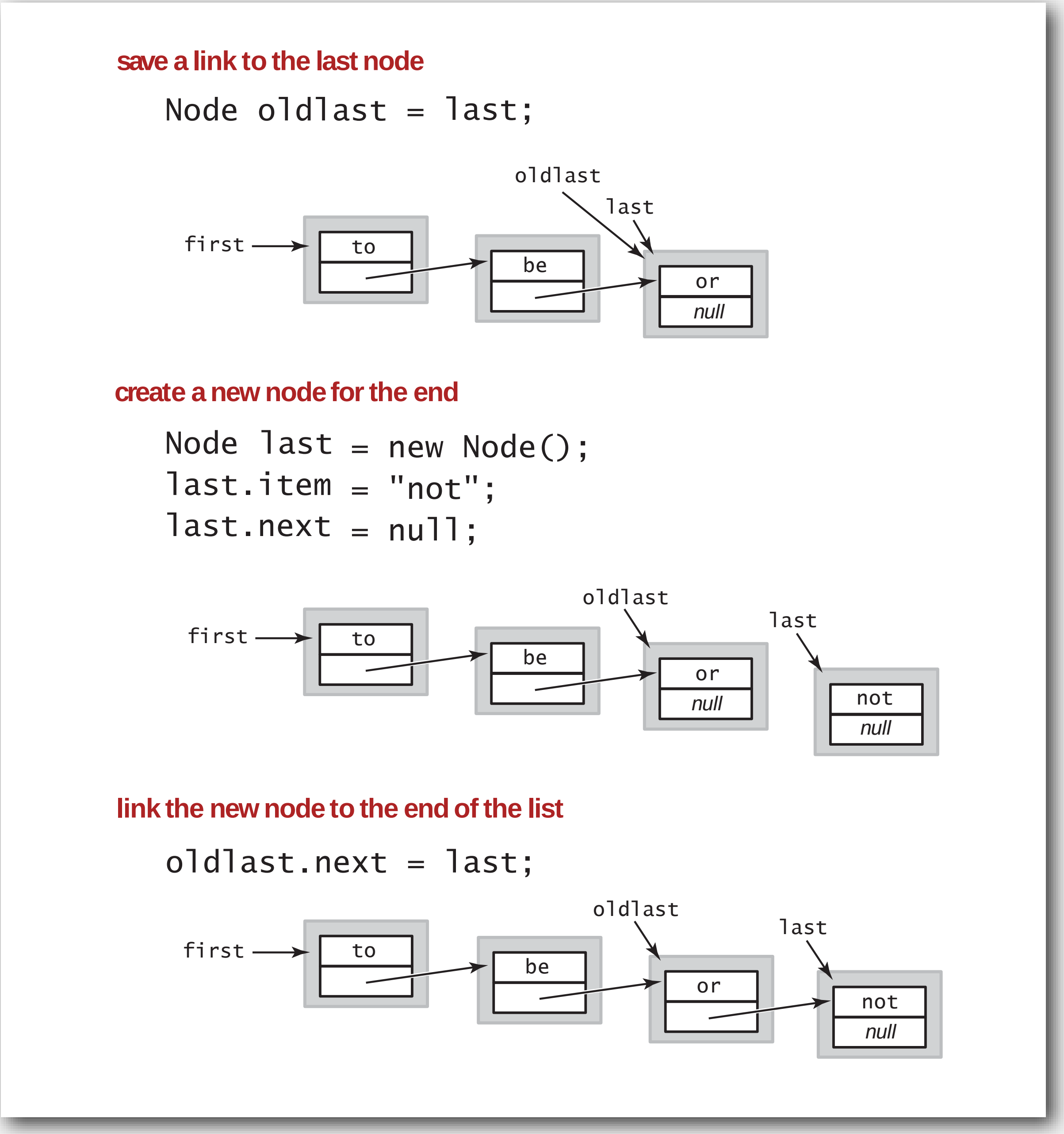
```
return item;
```

Remark. Identical code to linked-list stack `pop()`.

Queue enqueue: linked-list implementation

inner class

```
private class Node
{
    String item;
    Node next;
}
```



save a link to the last node

```
Node oldlast = last;
```

create a new node for the end

```
Node last = new Node();
last.item = "not";
last.next = null;
```

link the new node to the end of the list

```
oldlast.next = last;
```

Queue: linked-list implementation in Java

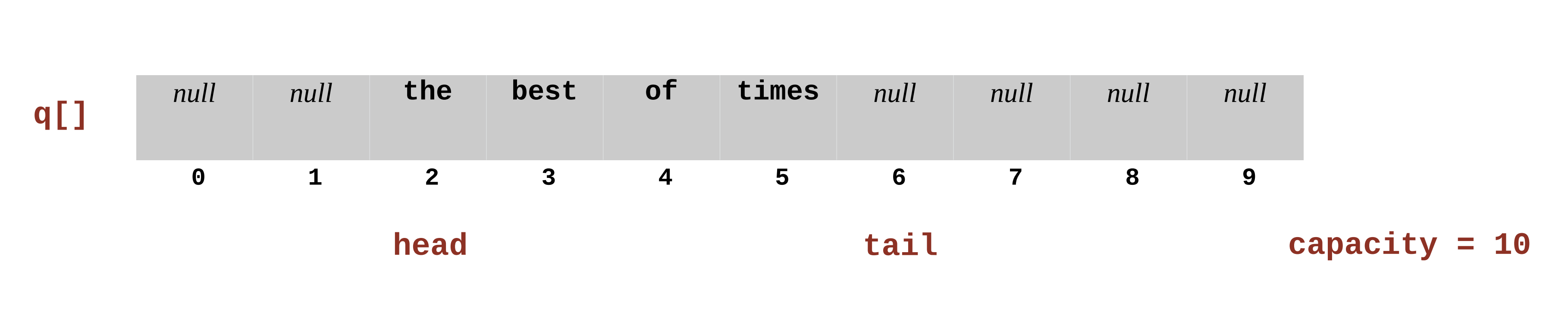
```
public class LinkedQueueOfStrings
{
   private Node first, last;

   private class Node
   {  /* same as in StackOfStrings */  }

   public boolean isEmpty()
   {  return first == null;  }

   public void enqueue(String item)
   {
      Node oldlast = last;
      last = new Node();
      last.item = item;
      last.next = null;
      if (isEmpty()) first = last;
      else           oldlast.next = last;
   }

   public String dequeue()
   {
      String item = first.item;
      first       = first.next;
      if (isEmpty()) last = null;
      return item;
   }
}
```

special cases for
empty queue

Queue: resizing array implementation

Array implementation of a queue.

- Use array **`q[]`** to store items in queue.
- **`enqueue()`**: add new item at **`q[tail]`**.
- **`dequeue()`**: remove item from **`q[head]`**.
- Update **`head`** and **`tail`** modulo the **`capacity`**.
- Add resizing array.



q[]	0	1	2	3	4	5	6	7	8	9
	null	*null*	the	best	of	times	*null*	*null*	*null*	*null*
			head				tail			

capacity = 10

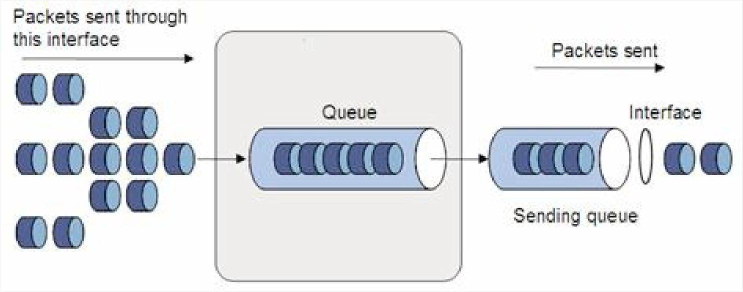
Queue applications

Familiar applications.

- iTunes playlist.
- Data buffers (iPod, TiVo).
- Asynchronous data transfer (file IO, pipes, sockets).
- Dispensing requests on a shared resource (printer, processor).

Simulations of the real world.

- Traffic analysis.
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.


Packets sent through this interface
Queue
Packets sent
Interface
Sending queue