Stacks and Queues

- *Stacks*
- *Resizing arrays*
- *Queues*
- *Generics*
- *Iterators (optional)*
- *Applications*

Teams Code

01d0tx9

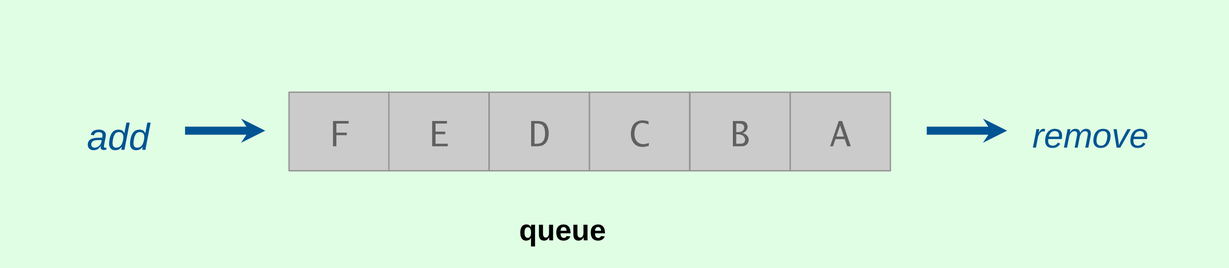
Grading Policy

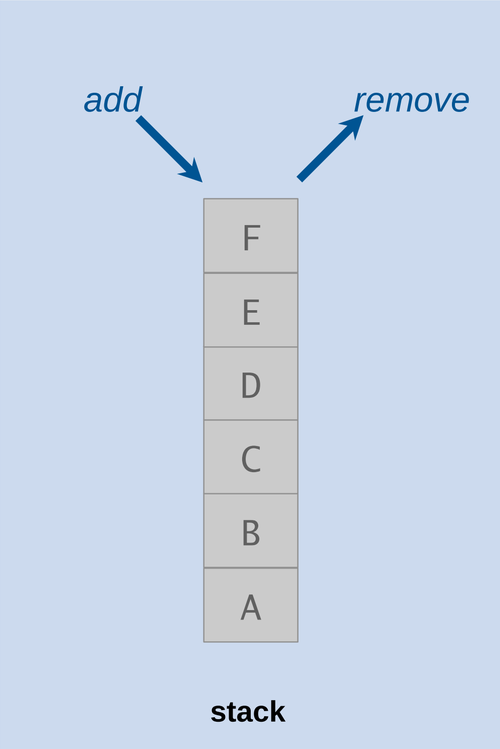
Mid-term Exam	20%
Project	10%
Quizzes	10%
Assignments	5%
Participation	5%
Final Exam	50%

Stacks and queues

Fundamental data types.

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



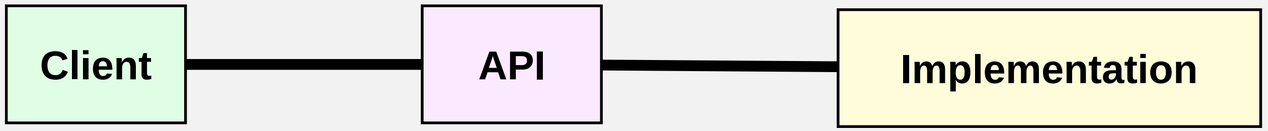


Stack. Examine the item most recently added. ← LIFO = "last in first out"

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

Client, implementation, API

Separate client and implementation via API.



API: operations that characterize the behavior of a data type.

Client: program that uses the API operations.

Implementation: code that implements the API operations.

Benefits.

- Client cannot know details of implementation ⇒ client has many implementation from which to choose.
- Implementation cannot know details of client needs ⇒ many clients can re-use the same implementation.
- Design: create modular, reusable libraries.
- Performance: substitute faster implementations.

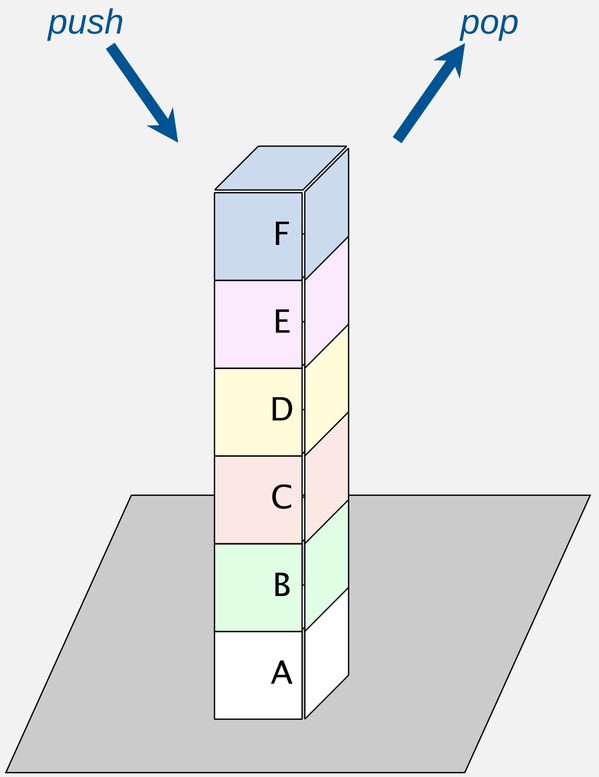
2. Stacks and Queues

- *Stacks*
- *Resizing arrays*
- *Queues*
- *Generics*
- *Iterators (optional)*
- *Applications*

Stack API

Warmup API. Stack of strings data type.

public class StackOfStrings	
StackOfStrings()	*create an empty stack*
void push(String item)	*add a new string to stack*
String pop()	*remove and return the string most recently added*
boolean isEmpty()	*is the stack empty?*
int size()	*number of strings on the stack*



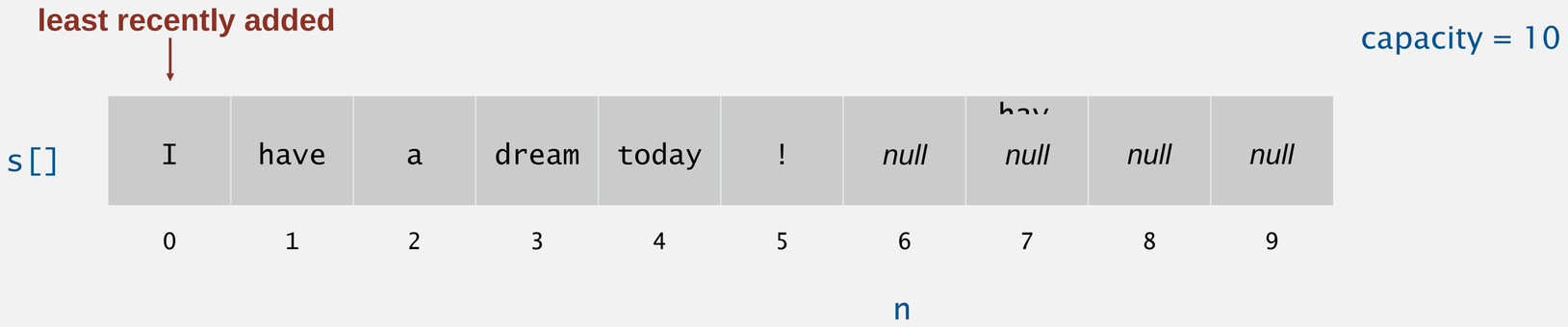
Performance goal. Every operation takes O(1) time.

Warmup client. Reverse a stream of strings from standard input.

Fixed-capacity stack: array implementation

LO 2.1

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



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

Fixed-capacity stack: array implementation


```
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];  }
}
```

post-increment operator:
use as index into array;
then increment n

pre-decrement operator:
decrement n;
then use as index into array

Stack considerations

Overflow and underflow.

- Underflow: throw exception if `pop()` called on an empty stack.
- Overflow: use "resizing array" for array implementation. [stay tuned]

Null items. We allow `null` items to be added.

Duplicate items. We allow an item to be added more than once.

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;
}
```

no loitering

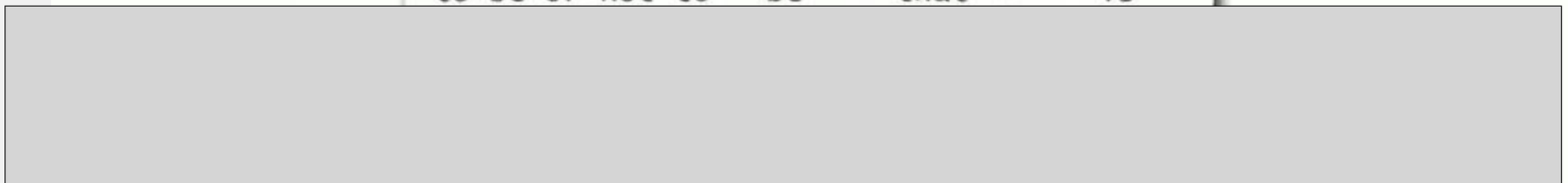
Stack test client

Read strings from standard input.

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

push pop

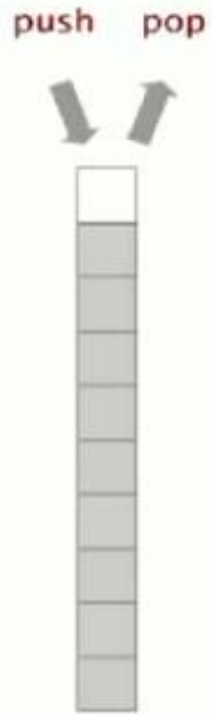
```
% more tobe.txt
to be or not to - be - - that - - - is
```



Stack test client

Read strings from standard input.

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



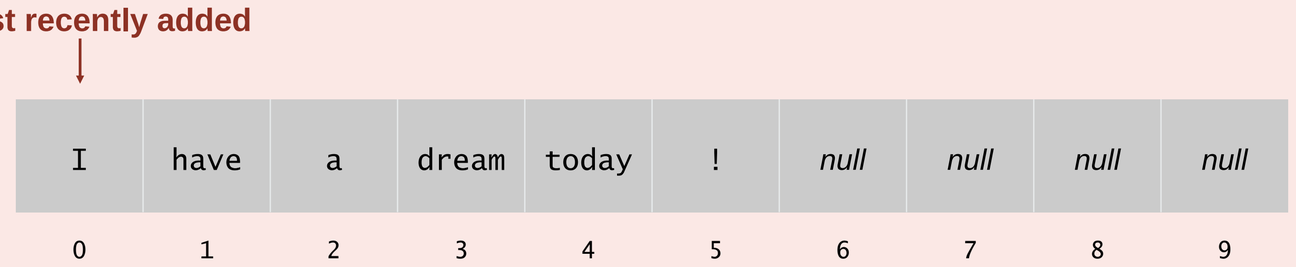

```
public static void main(String[] args)
{
   StackOfStrings stack = new StackOfStrings();
   while (!StdIn.isEmpty())
   {
      String s = StdIn.readString();
      if (s.equals("-")) StdOut.print(stack.pop());
      else               stack.push(s);
   }
}
```

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

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

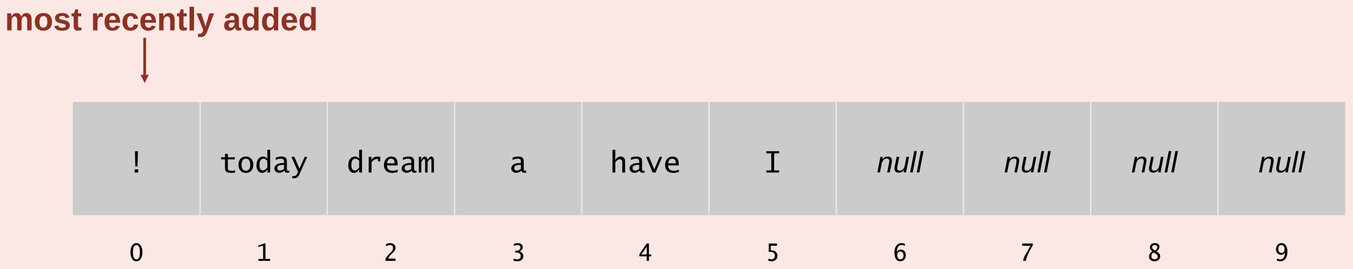
Stacks: quiz

How to implement a fixed-capacity stack with an array?

A.



B.



C. *Both A and B.*

D. *Neither A nor B.*

Answer = A

2. Stacks and Queues

- *Stacks*
- *Resizing arrays*
- *Queues*
- *Generics*
- *Iterators (optional)*
- *Applications (optional)*

Stack: resizing-array implementation

LO 2.3

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 items to a new array, for each operation.
- Array accesses to add first n items = $n + (2 + 4 + 6 + \ldots + 2(n-1)) \sim n^2$.

infeasible for large n

1 array access per push

$2(k-1)$ array accesses to expand to size k (ignoring cost to create new array)

Challenge. Ensure that array resizing happens infrequently.

Stack: resizing-array implementation

LO 2.3

Q. How to grow array?

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

"repeated doubling"

```
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;
}
```

Array accesses to add first $n = 2^i$ items. $n + (2 + 4 + 8 + \ldots + n) \leftarrow 3n.$

feasible for large n

1 array access per push

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

Stack: resizing-array implementation

LO 2.3

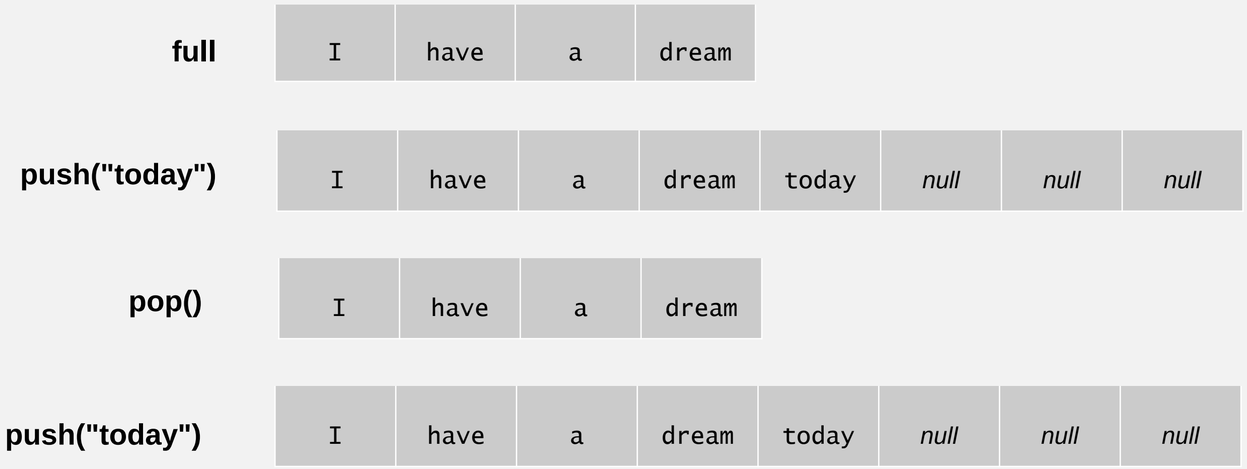
Q. How to shrink array?

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.

- Consider alternating sequence of push and pop operations when array is full.
- Each operation takes O(*n*) time.



Stack: resizing-array implementation

LO 2.3

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.

2. Stacks and Queues

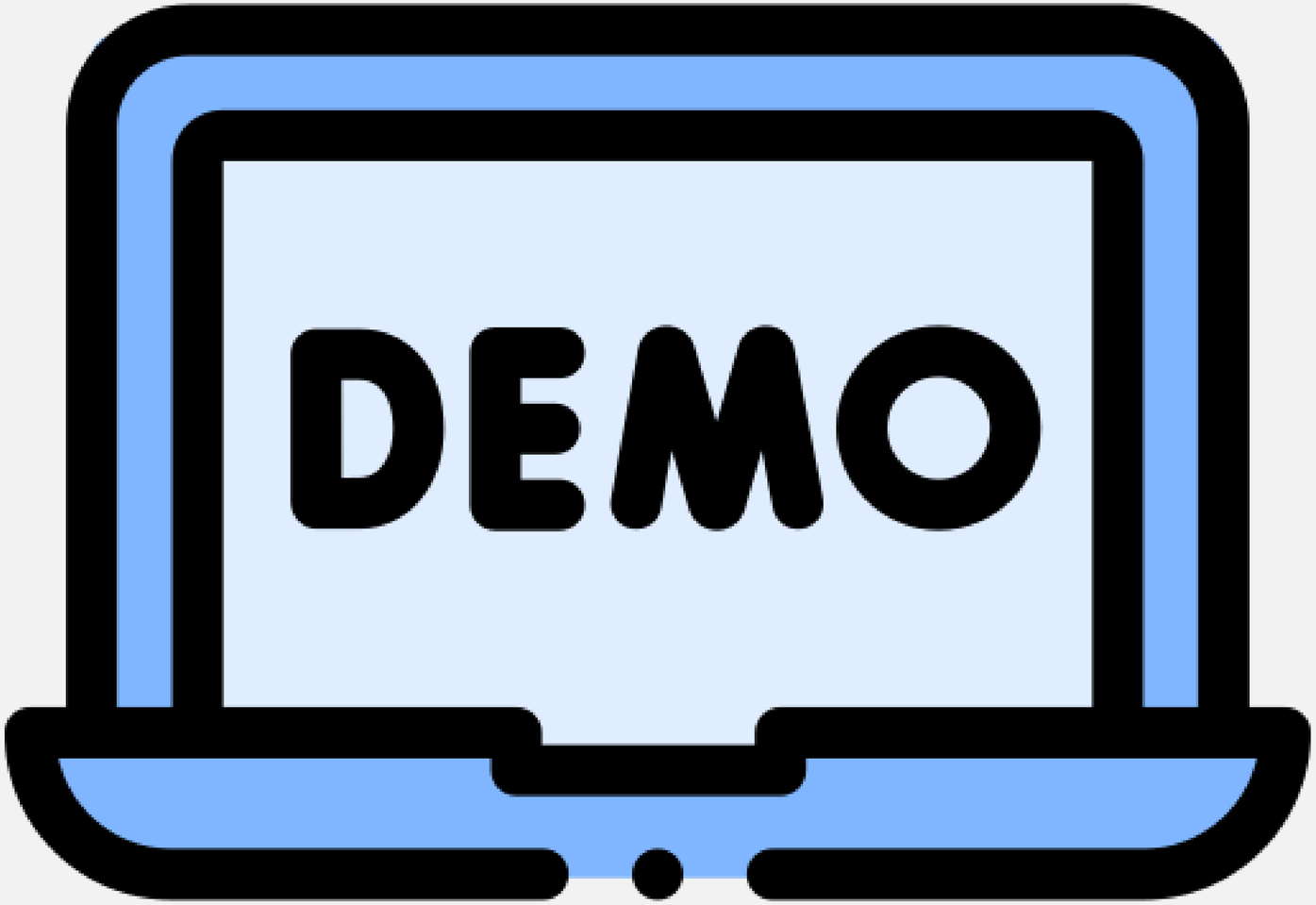
- *Stacks*
- *Resizing arrays*
- *Queues*
- *Generics*
- *Iterators (optional)*
- *Applications*

Queue of strings API

public class QueueOfStrings	
QueueOfStrings()	*create an empty queue*
void enqueue(String item)	*add a new string to queue*
String dequeue()	*remove and return the string least recently added*
boolean isEmpty()	*is the queue empty?*
int size()	*number of strings on the queue*

Performance goal. Every operation takes O(1) time.

Queue: Array Implementation



Queue: resizing-array implementation

LO 2.1

- 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`.

Q. How to resize?

QUEUE: RESIZING-ARRAY IMPLEMENTATION

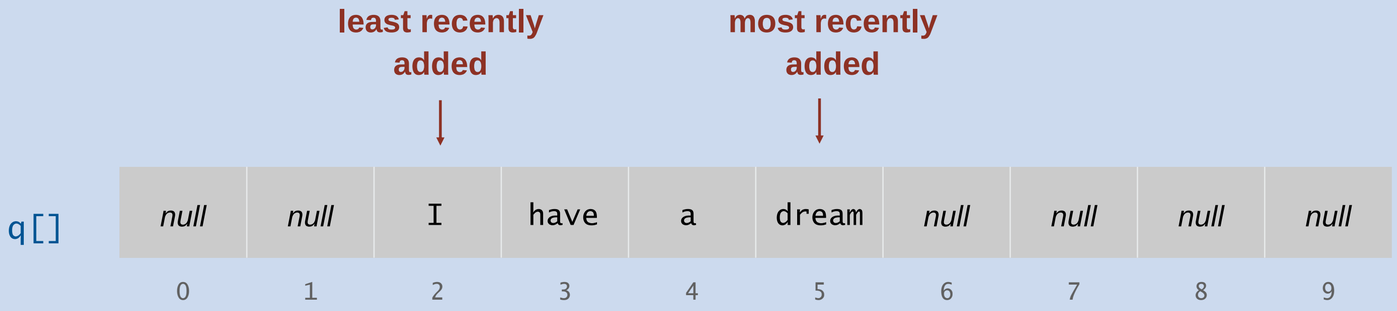
LO 2.1

Goal. Implement a queue using a resizing array so that, starting from an empty queue, any sequence of any sequence of m enqueue and dequeue operations takes $O(m)$ time.

QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

Goal. Implement a queue using a resizing array so that, starting from an empty queue, any sequence of any sequence of m enqueue and dequeue operations takes $O(m)$ time.



QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- 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`.

QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- 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`.

enqueue today

`q[]`

null	*null*	I	have	a	dream	*null*	*null*	*null*	*null*
0	1	2	3	4	5	6	7	8	9

`head` `tail`

QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- 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`.

enqueue !

`q[]`

null	*null*	I	have	a	dream	today	*null*	*null*	*null*
0	1	2	3	4	5	6	7	8	9

`head` `tail`

QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- 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`.

dequeue

`q[]`

null	*null*	I	have	a	dream	today	!	*null*	*null*
0	1	2	3	4	5	6	7	8	9

`head` `tail`

QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- 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`.

dequeue

`q[]`

null	*null*	*null*	have	a	dream	today	!	*null*	*null*
0	1	2	3	4	5	6	7	8	9

`head` `tail`

I

QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- 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`.

enqueue I

`q[]`

null	*null*	*null*	*null*	a	dream	today	!	*null*	*null*
0	1	2	3	4	5	6	7	8	9

`head` `tail`

have

QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- 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`.

enqueue have

`q[]`

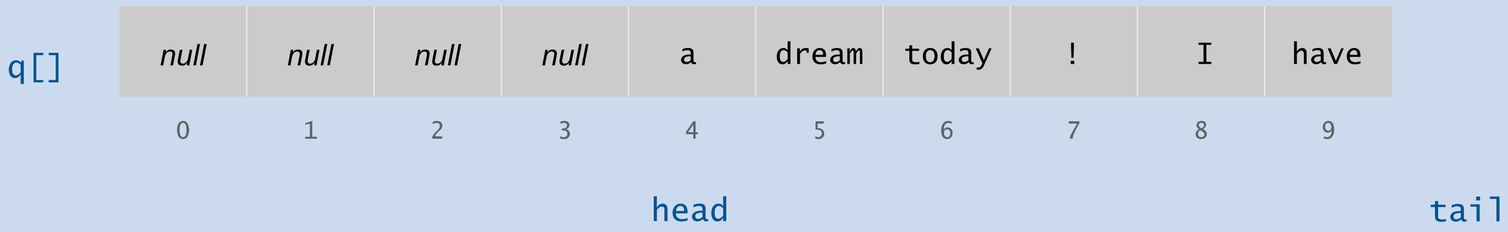
null	*null*	*null*	*null*	a	dream	today	!	I	*null*
0	1	2	3	4	5	6	7	8	9

`head` `tail`

QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- 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`.



QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- 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`.

enqueue a

`q[]`

null	*null*	*null*	*null*	a	dream	today	!	I	have
0	1	2	3	4	5	6	7	8	9

`tail` `head`

QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- 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`.

`q[]`

a	*null*	*null*	*null*	a	dream	today	!	I	have
0	1	2	3	4	5	6	7	8	9

`tail` `head`

QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- 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`.

Q. How to resize?

`q[]`

a	*null*	*null*	*null*	a	dream	today	!	I	have
0	1	2	3	4	5	6	7	8	9

`tail` `head`

2. Stacks and Queues

- *Stacks*
- *Resizing arrays*
- *Queues*
- *Generics*
- *Iterators (optional)*
- *Applications*

Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfInts`, `StackOfApples`, `StackOfOranges`, ...

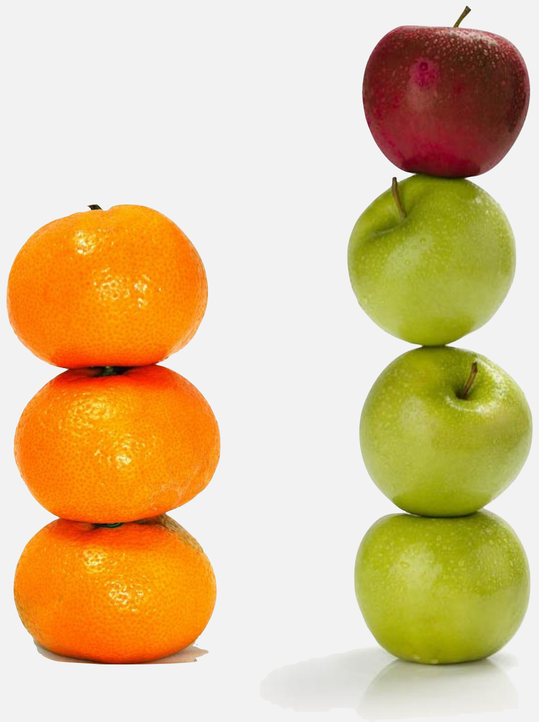
Solution in Java: generics.

type parameter

(use syntax both to specify type and to call constructor)

```
Stack<Apple> stack = new Stack<Apple>();
Apple apple = new Apple();
Orange orange = new Orange();
stack.push(apple);
stack.push(orange);
...
```

compile-time error



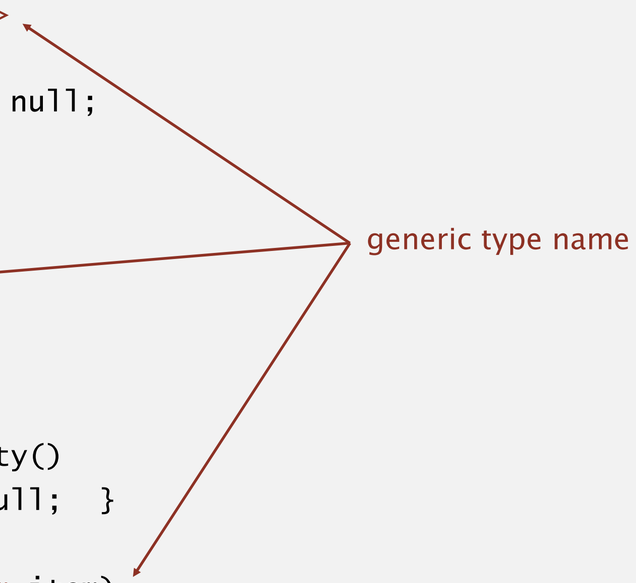
Generic stack: linked-list implementation

stack of strings (linked list)

```
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;
   }
}
```

generic stack (linked list)

```
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 stack: array implementation

The way it should be.

stack of strings (fixed-length array)

```
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];  }
}
```

Generic stack of strings (fixed-length array)

```
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

The way it should be.

stack of strings (fixed-length array)

```
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];  }
}
```

generic stack (fixed-length array)

```
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

Unchecked cast

```
~/Desktop/queues> javac -Xlint:unchecked FixedCapacityStack.java
FixedCapacityStack.java:26: warning: [unchecked] unchecked cast
        s = (Item[]) new Object[capacity];
                     ^
  required: Item[]
  found:    Object[]
  where Item is a type-variable:
    Item extends Object declared in class FixedCapacityStack
1 warning
```

Q. Why does Java require a cast (or reflection)?

Short answer. Backward compatibility.

Long answer. Need to learn about type erasure and covariant arrays.

Generic data types: autoboxing and unboxing

Q. What to do about primitive types?

Wrapper type.

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

Autoboxing. Automatic cast from primitive type to wrapper type.

Unboxing. Automatic cast from wrapper type to primitive type.

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);          // stack.push(Integer.valueOf(17));
int a = stack.pop();     // int a = stack.pop().intValue();
```

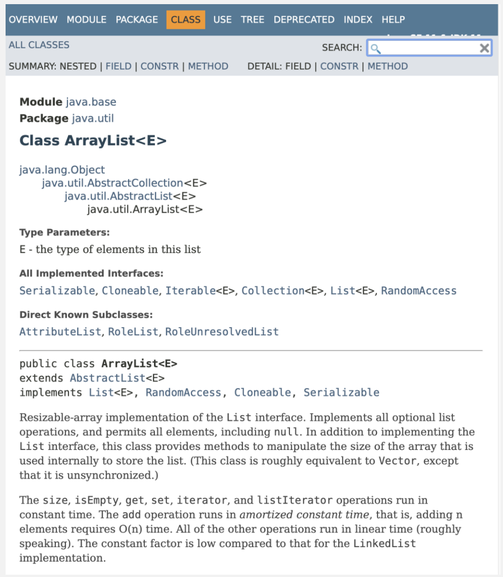
Bottom line. Client code can use generic stack for any type of data.

(but substantial overhead for primitive types)

Java collections framework

Java's library of collection data types.

- `java.util.ArrayList` [resizing array]
- `java.util.LinkedList` [doubly linked list]
- `java.util.ArrayDeque`



OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

ALL CLASSES SEARCH:

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Module java.base
Package java.util

Class ArrayList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>

Type Parameters:

E - the type of elements in this list

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

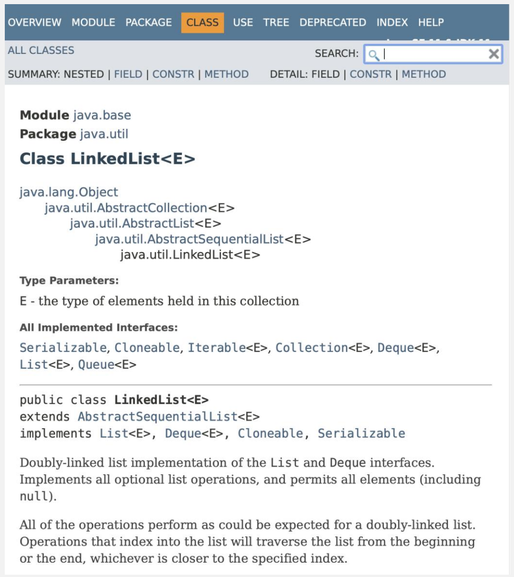
Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.



OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

ALL CLASSES SEARCH:

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Module java.base
Package java.util

Class LinkedList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.AbstractSequentialList<E>
        java.util.LinkedList<E>

Type Parameters:

E - the type of elements held in this collection

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

This course. Implement from scratch (once).

Beyond. Basis for understanding performance guarantees.

Best practices.

- Use our `Stack` and `Queue` for stacks and queues to improve design and efficiency.
- Use Java's `ArrayList` or `LinkedList` when other ops needed
  (but remember that some ops are inefficient).

Stacks and queues summary

Fundamental data types.

- Value: collection of objects.
- Operations: add, remove, iterate, test if empty

Stack. Examine the item most recently added (LIFO).

Queue. Examine the item least recently added (FIFO).

Efficient implementations.

- Singly linked list.
- Resizing array.

Next time. Advanced Java (including iterators for collections).

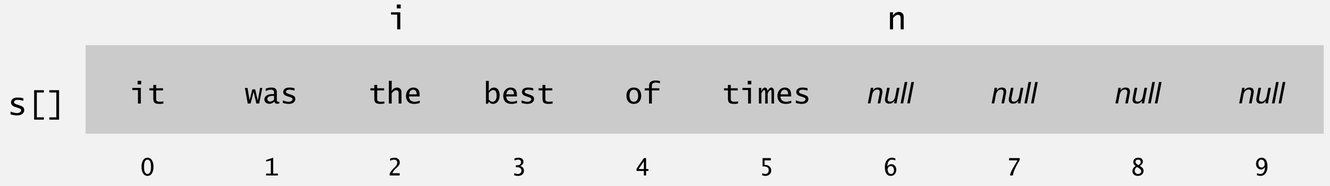
2. Stacks and Queues

- *Stacks*
- *Resizing arrays*
- *Queues*
- *Generics*
- *Iterators (optional)*
- *Applications*

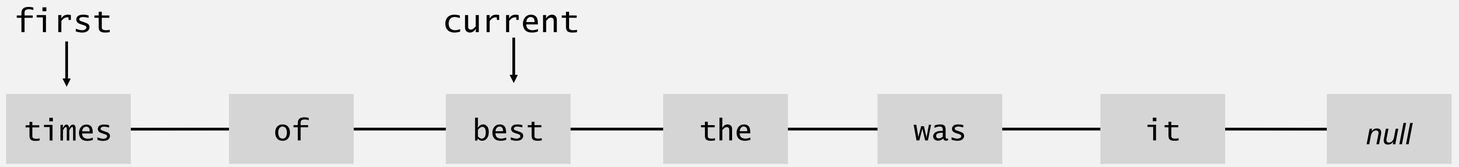
Iteration

Design challenge. Allow client to iterate over the stack items, without exposing the internal representation of the stack.

resizing-array representation



linked-list representation



Java solution. Use a foreach loop.

Foreach loop

Java provides elegant syntax for iterating over items in a collection.

“foreach” loop (shorthand)

```
Stack<String> stack;
...

for (String s : stack)
    ...
```

equivalent code (longhand)

```
Stack<String> stack;
...

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

To make user-defined collection support foreach loop:

- Data type must have a method named `iterator()`.
- The `iterator()` method returns an object that has two core method:
  - the `hasNext()` methods returns `false` when there are no more items
  - the `next()` method returns the next item in the collection

Iterators

To support foreach loops, Java provides two interfaces.

- `Iterator` interface: `next()` and `hasNext()` methods.
- `Iterable` interface: `iterator()` method that returns an `Iterator`.
- Both should be used with generics.

java.util.Iterator interface

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

java.lang.Iterable interface

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

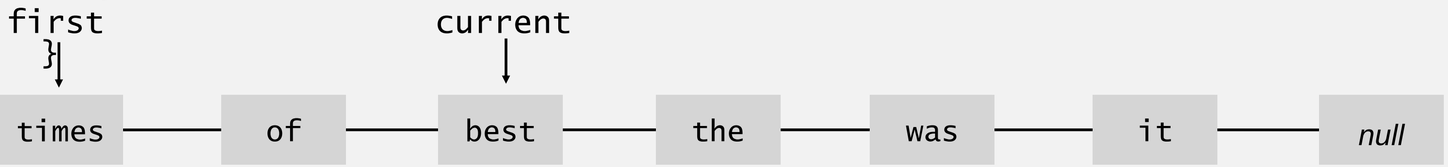
Type safety.

- Implementation must use these interfaces to support foreach loop.
- Client program won't compile unless implementation do.

Stack iterator: linked-list implementation

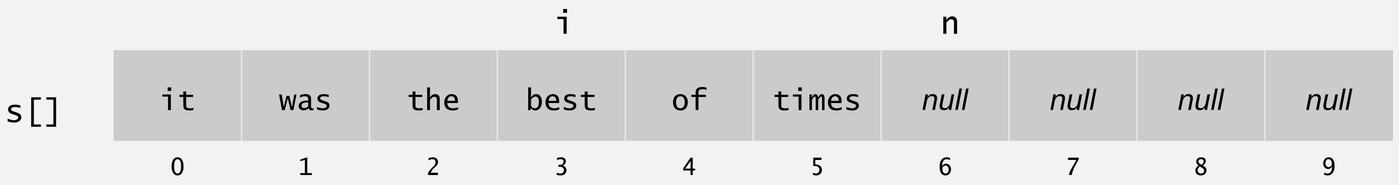
```
import java.util.Iterator;

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

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

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

      public boolean hasNext()  {  return current != null;  }

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

throw NoSuchElementException if no more items in iteration



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 Item next()       {  return s[--i];  }
   }

}
```



ITERATION: CONCURRENT MODIFICATION

Q. What if client modifies the data structure while iterating?

A. A fail-fast iterator throws a `java.util.ConcurrentModificationException`.

concurrent modification

```
for (String s : stack)
   stack.push(s);
```

Q. How to detect concurrent modification?

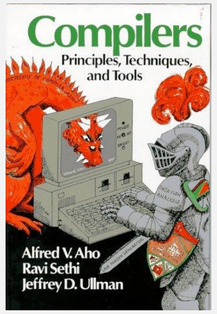
A.

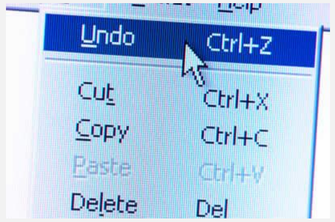
2. Stacks and Queues

- *Stacks*
- *Resizing arrays*
- *Queues*
- *Generics*
- *Iterators (optional)*
- *Applications*

Stack applications

- Parsing in a compiler.
- Undo in a word processor.
- Back button in a Web browser.
- Function-call stack during execution of a program.





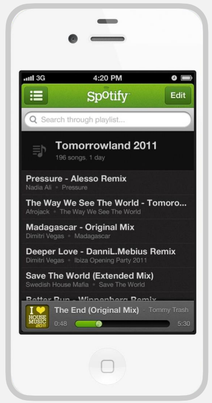
Queue applications

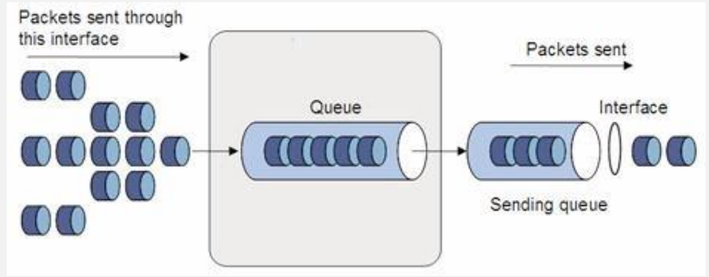
Familiar applications.

- Spotify playlist.
- Data buffers (iPod, TiVo, sound card, streaming video, …).
- Asynchronous data transfer (file I/O, 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.





Java collections library

Optional content

List interface. `java.util.List` is API for a sequence of items.

public interface List<Item> extends Iterable<Item>	
List()	*create an empty list*
boolean isEmpty()	*is the list empty?*
int size()	*number of items*
void add(Item item)	*add item to the end*
Iterator<Item> iterator()	*iterator over all items in the list*
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?*
⋮	

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

Caveat: not all operations are efficient!

Java collections library

`java.util.Stack`.

- Supports `push()`, `pop()`, and iteration.
- Inherits from `java.util.Vector`, which implements `java.util.List` interface.

Java collections library

`java.util.Stack`.

- Supports `push()`, `pop()`, and iteration.
- Inherits from `java.util.Vector`, which implements `java.util.List` interface.

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

Best practices. Use our `Stack` and `Queue` for stacks and queues;
use `java.util.ArrayList` or `java.util.LinkedList` when appropriate.