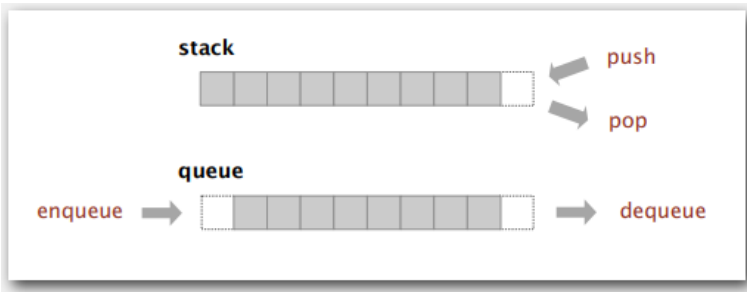
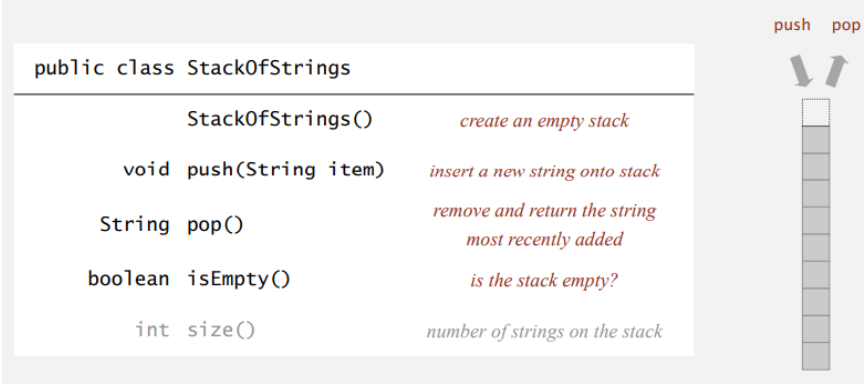
Stacks and Queues (Week 2):

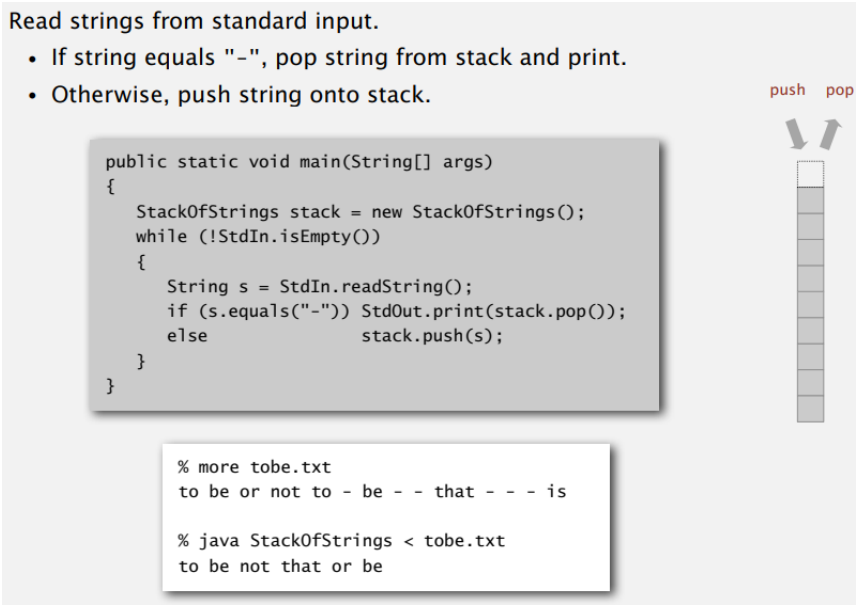
- Stacks
    - Fundamental data types
        - Value: collection of objects
        - Operations: insert, remove, iterate, test if empty.
        - Intent is clear when we insert.
        - Which item do we remove?
            - Stack
                - Remove the item that was most recently added.
                - LIFO - “last in, first out”
            - Queue
                - Remove the item that was least recently added/
                - FIFO - “first in, first out”
            - 

    - Client, implementation, interface (subtext/pre-lecture teaching definitions and concept of modular programming)
        - Definitions
            - Interface
                - The text description of a data type and its basic operations.
            - Implementation
                - Actual code that implements the described operations.
            - Client
                - The program using the operations defined in the interface.
        - Modular programming
            - The idea of modular programming is to completely separate the interface and the implementation.
                - We want to completely separate the **details of the implementation** of these data structures and data types that are precisely defined, like stacks and queues and so forth, from the client.
                    - Allows the client to have many different implementations from which to choose, but the code that the client has, only performs the fundamental operations (refer to Fundamental Data Types).
                - On the other hand, the implementation can’t know the details of the client's needs, all it's supposed to do is implement the operations (refer to Fundamental Data Types).

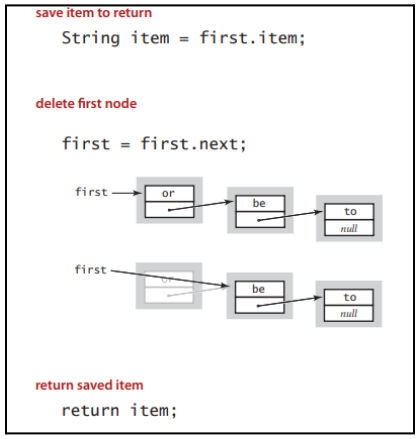
        - Allows many clients to reuse the same implementation.
    - Benefits of separating the interface and the implementation:
        - Allows us to create modular, reusable libraries of algorithms and data structures.
            - Can use these libraries to build more complicated algorithms and data structures.
        - Allows us to focus on performance when appropriate.
- Stacks
    - Stack API Example
        - Warmup API
            - Stack of string data type.



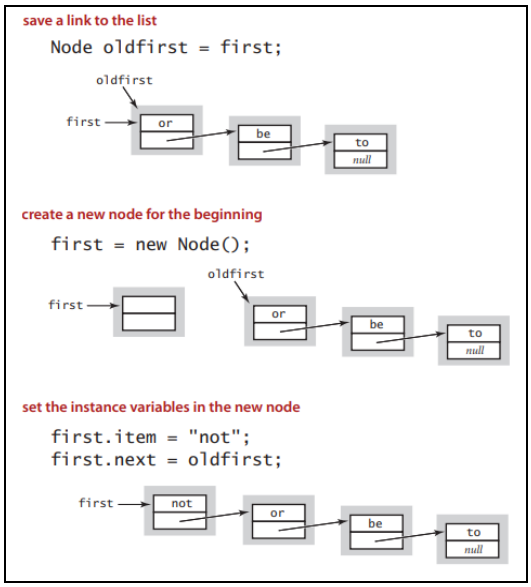
        - Warmup client
            - Reverse a sequence of strings from standard input.
        - Another example client



    - Implementing a stack
        - Linked-list representation
            - Maintain pointer to first node in a linked list
            - Insert/remove from the front of the list.

- Top of stack is at the front of the list.
  - Insert/Push creates a new node that points to what used to be the front of the linked list.
  - Remove/Pop removes the node at the front of the linked list/top of the stack.
- Code
  - Inner Class
    - For our linked list implementation we need to use an inner class.
    - An inner class is a class inside another class.
    - Since our linked-list's elements need to contain two pieces of information (the information itself, and the link to the next element), we create a node class inside our linked list class to help store these pieces of information.
    -
```
inner class
private class Node
{
   String item;
   Node next;
}
```
  - Pop operation
    - 

    - First, you save the value of the top of the stack in a temp variable, then you change the top of the stack to the node under it which also removes the current top and allows a new top. Then you return the temp variable.
  - Push operation



- First, you save a link to the list itself by creating a new node that equals the node at the top of the stack. Then you re-initialize the node at the top of the stack by making it a new node object. Then you give the new node object, that we just created, a value and make it connect to the link to the list we saved earlier. This puts it at the top of the stack effectively "pushing" a new element to the stack.

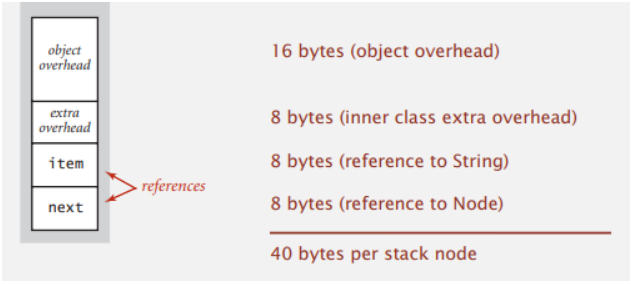
■ Full implementation

- Linked stack of strings (Stack of strings using a 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;
   }
}
```

private inner class
(access modifiers don't matter)

        - Implementation performance
            - Proposition (Time)
                - Every operation takes constant time in the worst case.
                    - No loops or iteration.
            - Proposition (Space)
                - A stack with N items uses ~40N bytes



                - Inner classes just add 8 bytes as an overhead because it is an inner class.
- Array representation
    - Array implementation of a stack
        - Use array s[] to store N items on the stack.
        - push(): add new item at s[N].
        - pop(): remove item from s[N-1].
    - Fundamental defect of using an array
        - Arrays have a preset size so when you push N past the array's declared size/capacity, the stack overflows and you get an error.
    - Code
        - Full implementation
            - Fixed capacity stack of strings

```
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];  }
}
```

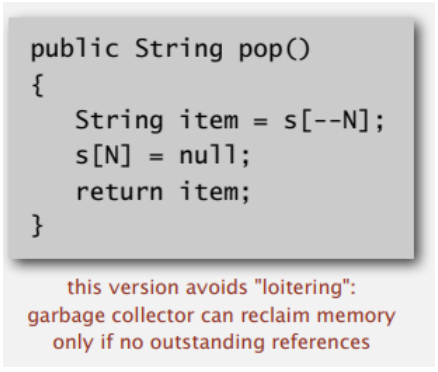
a cheat (stay tuned)

use to index into array; then increment N

decrement N; then use to index into array

- We have the parameter in the constructor that takes in the size of the stack but this is considered a cheat since the client wouldn't always know how big the stack would need to be.
- Also, note that we don't need to remove any elements in the pop() method since when we get back to that index, we write over it with a new value.

- Stack considerations
  - Overflow and underflow
    - Underflow
      - What happens if we pop from an empty stack? The code would try to return the value at index -1 which would cause an error, so we need to throw an exception.
    - Overflow
      - Since our array requires a preset size/capacity indicated at creation, it can overflow if we push elements past the max size causing an error, we should use a resizing array for array implementation.
  - Null items
    - We allow null items to be inserted.
  - Loitering
    - We hold a reference to an object when it is no longer needed (we don't remove the top of the stack in the pop() method we just return lower index values).
    - Version of pop() to avoid loitering



- Resizing-array representation
  - Problem with array representation

        - The API is not implemented correctly if the client has to provide the capacity.
            - Stack API specifies that we should be able to create a stack that is able to grow and shrink to any size.
        - So how do we grow and shrink the array?
    - First attempt
        - push()
            - Increase size of array s[] by 1.
        - pop()
            - Decrease size of array s[] by 1.
        - Problem
            - Too expensive
                - Need to copy all items to a new array.
    - Challenge
        - Since array resizing is so expensive, we need to make sure it happens infrequently.
    - Second attempt
        - How do we grow the array?
            - When the array fills up, create a new array of twice the size and copy the items over.
        - Implementation

```
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;
}
```

        - How do we shrink the array?
            - Halve the size of the array s[] when the array is one-quarter full.
                - If we were to halve it when it's half full, when you repeatedly push-pop-push-pop at the max current capacity it would constantly grow and shrink making it incredibly inefficient.
        - Implementation

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

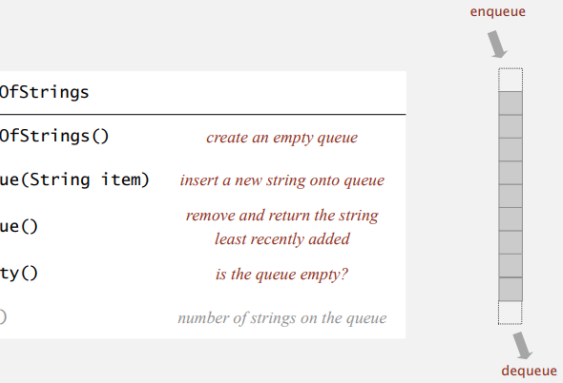
        - Invariant - the array is between 25% and 100% full.
    - Performance
        - Amortized analysis
            - Average running time per operation over a worst-case sequence of operations.
        - Proposition
            - Starting from an empty stack, any sequence of M push and pop operations takes time proportional to M.
        - Memory usage
            - Uses between ~8N and ~32N bytes to represent a stack with N items.
                - ~8N when full
                - ~32N when one-quarter full
- Resizing array vs linked list
    - The client can use either interchangeably but the choice may depend on the client's needs.
    - Linked-list implementation.
        - Every operation takes constant time in the worst case.
        - Uses extra time and space to deal with the links.

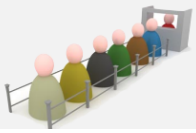
      Resizing-array implementation.
        - Every operation takes constant amortized time.
        - Less wasted space.

- Queues
    - Queue API

-
  -

public class QueueOfStrings	
QueueOfStrings()	*create an empty queue*
void enqueue(String item)	*insert a new string onto 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*





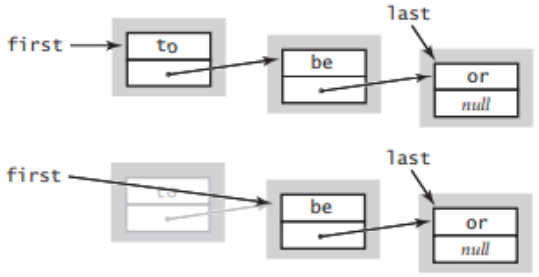
- Implementing a queue
  - Linked-list representation
    - Maintain pointer to first and last nodes in a linked list.
    - Insert/remove from opposite ends.
    - Since we are using a linked list, we are using the same Node inner class
      -
```
inner class

private class Node
{
   String item;
   Node next;
}
```
    - Dequeue()

save item to return

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

delete first node

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



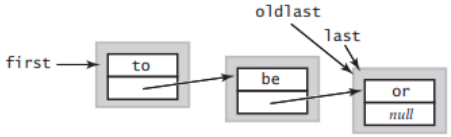
return saved item

```
return item;
```

-
  - Removes and returns the item at the front of the “line”. I.e. front/head of the list/queue.
- Enqueue()

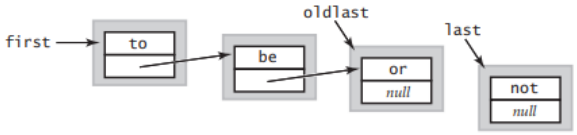
save a link to the last node

```
Node oldlast = last;
```



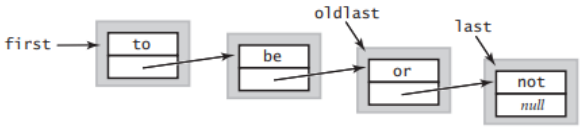
create a new node for the end

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



link the new node to the end of the list

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



-
  - Adds an item to the back of the “line”. I.e. back/tail of the list/queue.
- Full implementation of linked-list queue

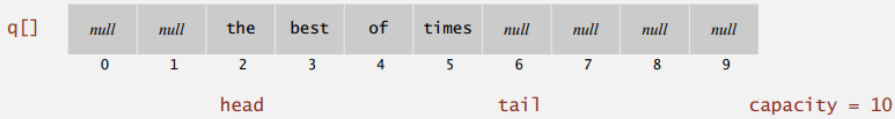
```
public class LinkedQueueOfStrings
{
   private Node first, last;

   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

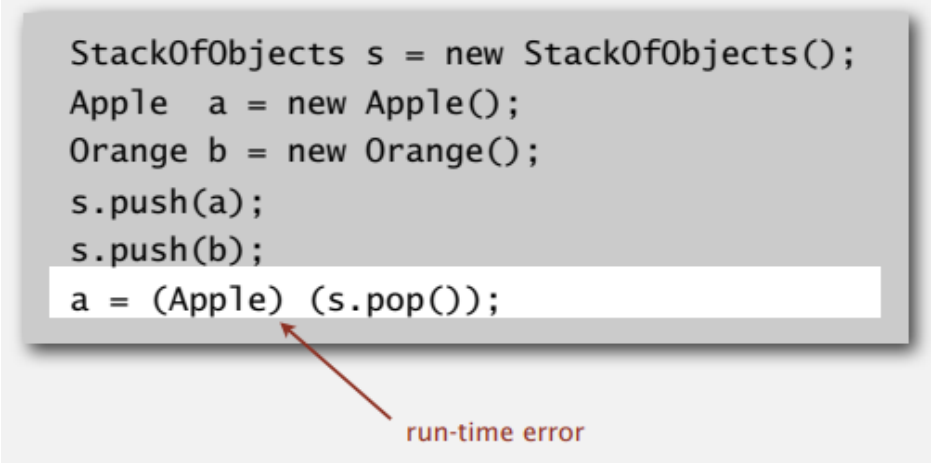
- Resizing array implementation

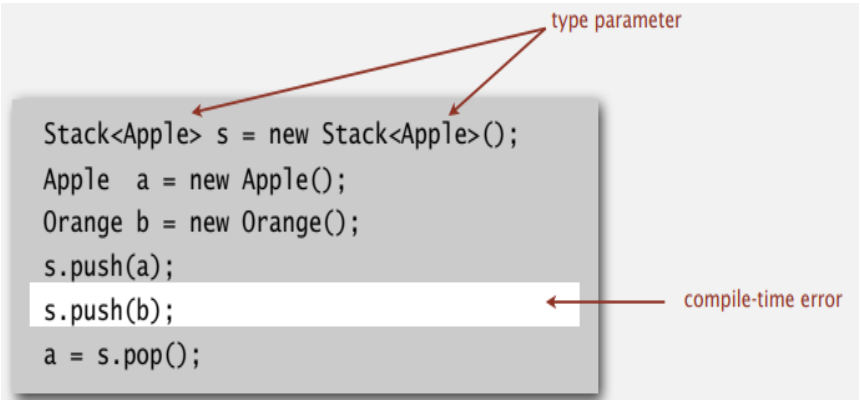
Array implementation of a queue.

- 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.

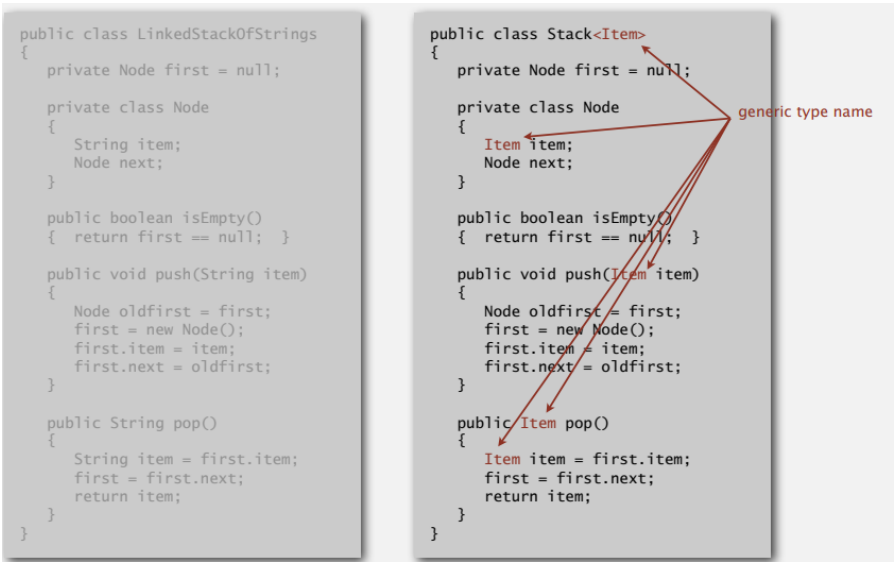


- Generics
  - Fundamental defect with our implementations so far
    - Our implementations only work for strings.
    - What if we want to have queues and stacks of other types of data?
      - This brings us to generics.
  - Generics
    - We implemented StackOfStrings but what if we want StackOfURLs, StackOfInts, StackOfVans, and so on?
    - First attempt
      - Implement a separate stack class for each type.
      - Problems

        - Rewriting code is tedious and error-prone.
        - Maintaining cut-and-pasted code is tedious and error-prone.
        - Could be a never-ending amount of data types to implement versions for.
- Second attempt
    - Implement a stack with items of type Object.
    - Problems
        - You need to cast the object to the specified data type in the client/program.
        - Casting is error-prone: run-time error if types mismatch.
        - 

        - Code has a run-time error since you try to cast an object of type Orange into an object of type Apple.
- Third attempt
    - Implement a stack using generics
    - Benefits
        - Avoid casting in the client/program.
        - Can discover mismatch errors at compile-time instead of run-time.
        - 

        - Code would have a compile-time error since you are trying to push an object of type Orange onto a stack of Apple objects.
    - Full implementation of linked-list stack

- 

- Full implementation of fixed-capacity array stack
  - 

  - Note we have to create an array of type object THEN cast it to the generic type as Java doesn't allow direct generic array creation.
  - This code does send you a warning message because of the cast but we can move past it.

■ Sidenote
- Welcome compile-time errors and avoid run-time errors.
  - If we can detect an error at compile time we can fix it and send out your product with confidence that it works with any client whereas if the error doesn't occur until during run-time, it may affect some client development years after deployment of our software.
- Usually good code has zero cast so we should try our best to avoid them at all costs unless absolutely necessary.

- Primitive data types
  - What do we do about primitive types?
    - Wrapper classes
      - Each primitive data type has a wrapper object type that allows it to be considered as a generic.
      - For example, Integer is the wrapper class for int.
    - Autoboxing
      - Java automatically casts between a primitive type and its wrapper.
      -
```
Stack<Integer> s = new Stack<Integer>();
s.push(17);        // s.push(Integer.valueOf(17));
int a = s.pop();   // int a = s.pop().intValue();
```
  - Because of the wrapper classes, the client can now use the generic stack for any type of data (both primitive and non-primitive).
- Iterators
  - Design Challenge
    - We want to allow our programs/clients to be able to iterate through the items in collections.
    - The problem is, we don't want the client to know whether we're using an array or linked list or whatever other internal representation. It's not relevant to the client. The clients just need to iterate through the stuff in the collection.
    - This is where Java provided the Iterable interface.
  - Iterables
    - What is an Iterable?
      - In Java lingo, an iterable is a class that has a method that returns an iterator.
      - What is an iterator?
        - An iterator is a class that has methods hasNext and next().
      -

Iterable interface

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

Iterator interface

```
public interface Iterator<Item>
{
   boolean hasNext();
   Item next();
   void remove();   <-- optional; use at your own risk
}
```

      - Why make data structures iterable?
        - Makes client code more elegant and readable.

- "foreach" statement (shorthand)

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

equivalent code (longhand)

```
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;
        }
    }
}
```

throw UnsupportedOperationException

throw NoSuchElementException if no more items in iteration

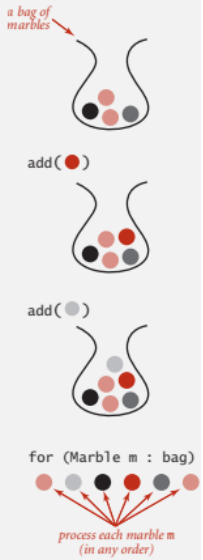
    - Array implementation

```
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];       }
    }
}
```

- Bags
  - Adding items to collections and iterating (when order doesn't matter).
  - Bag API

(which 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).
- Deques and Randomized Queues assignment grade: 100%