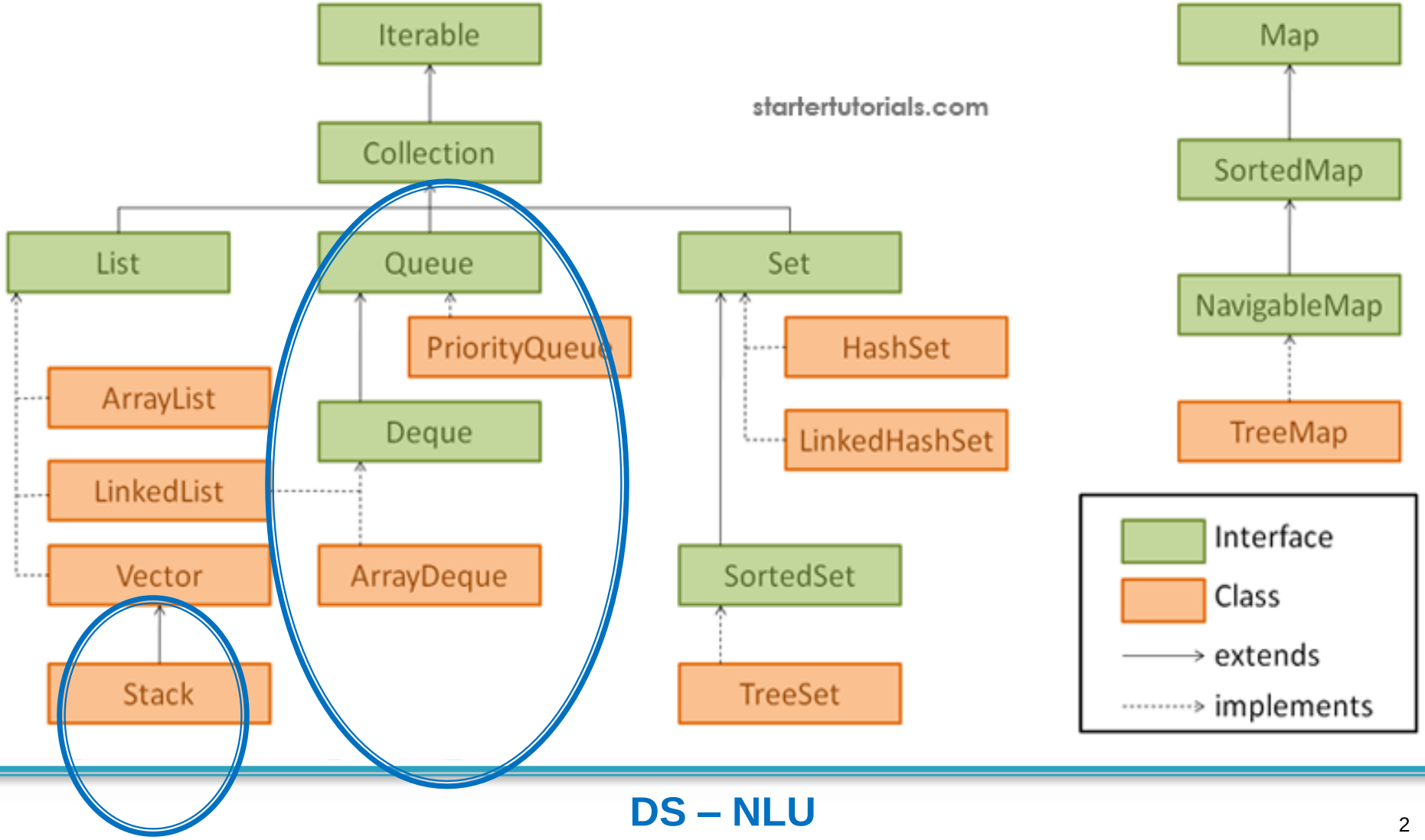
FACULTY OF INFORMATION TECHNOLOGY

DATA STRUCTURES (CTDL)

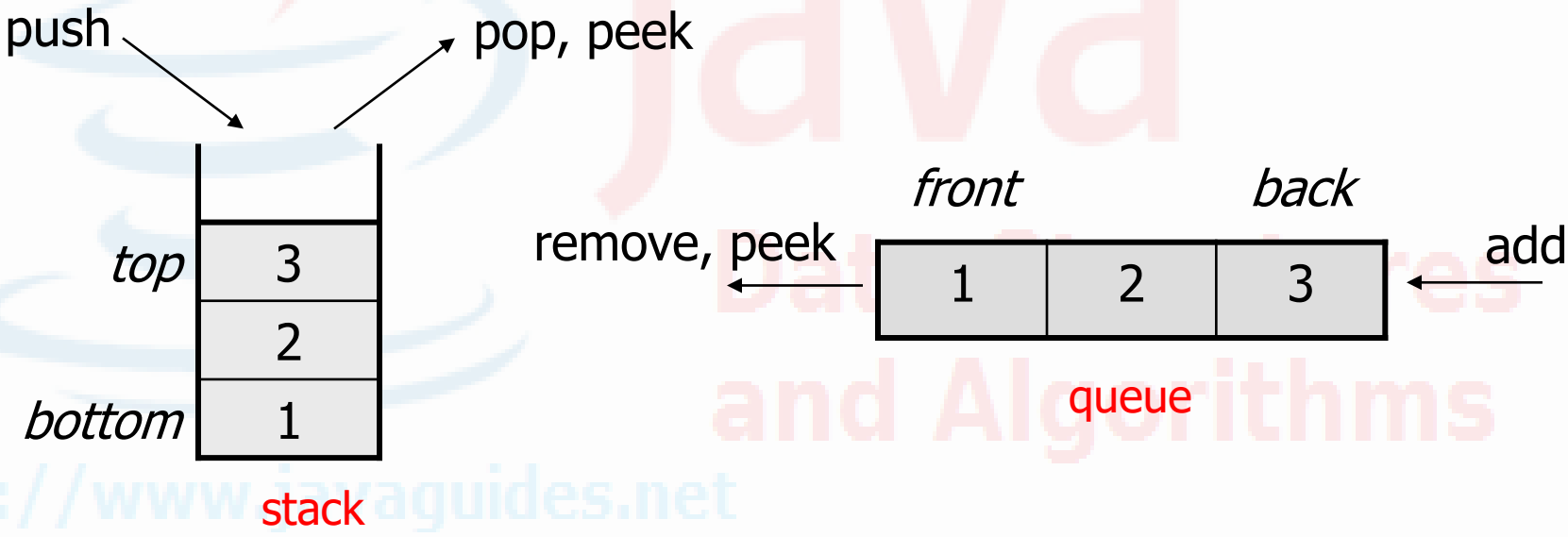
Semester 1, 2021/2022

Java Collection framework

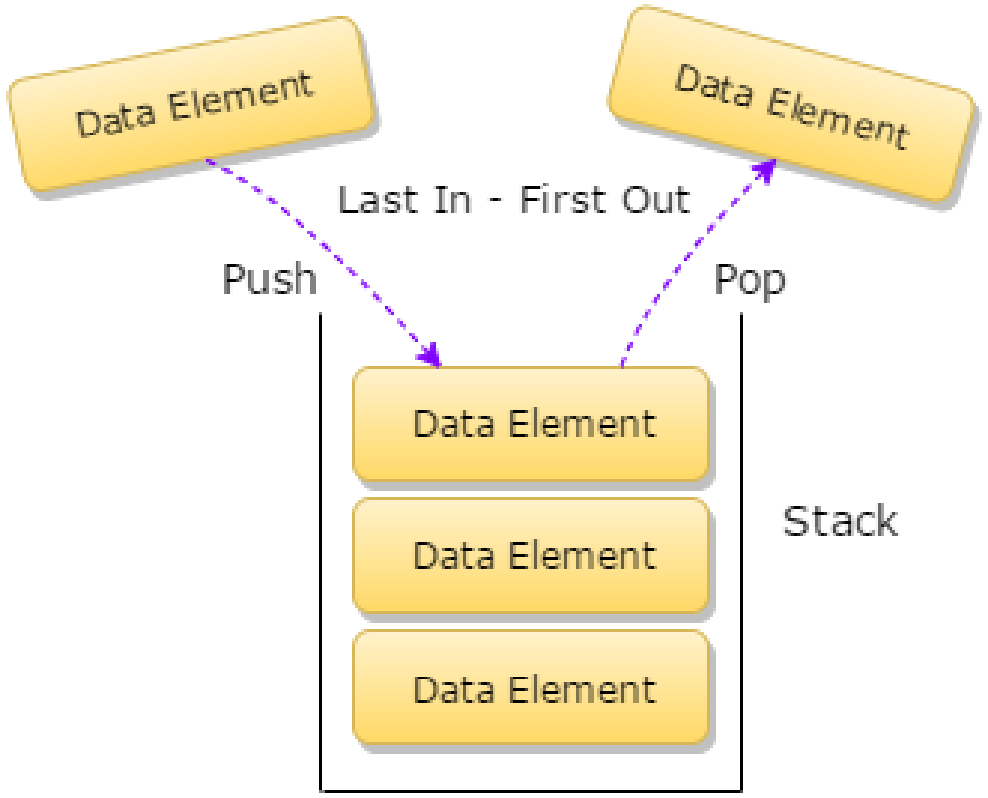


Stacks and queues

- 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



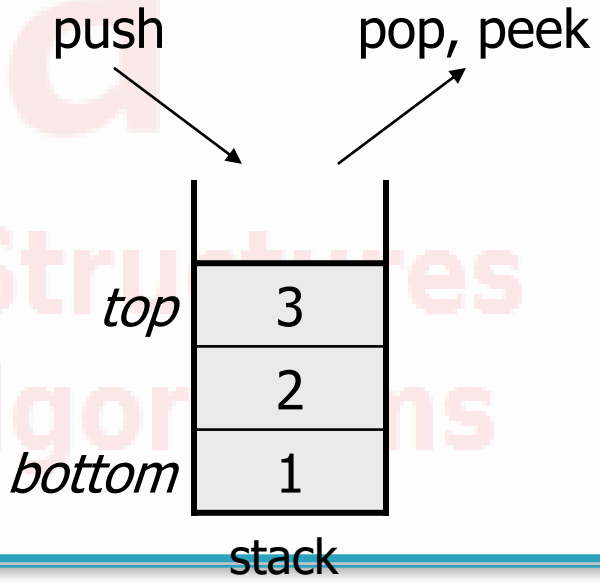
Stack



Definition

- **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.



Stack ADT methods

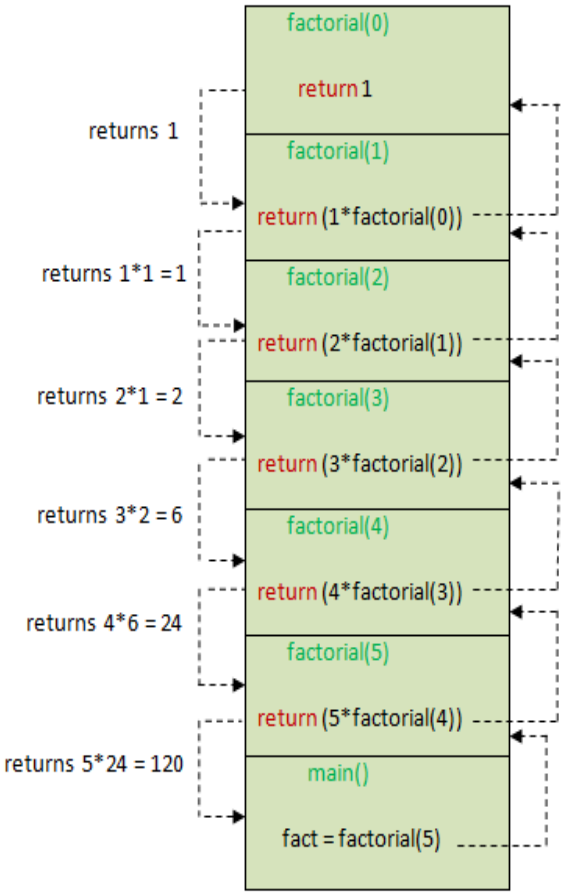
push(e):	Adds element e to the top of the stack.
pop( ):	Removes and returns the top element from the stack (or null if the stack is empty).
top( ):	Returns the top element of the stack, without removing it (or null if the stack is empty).
size( ):	Returns the number of elements in the stack.
isEmpty( ):	Returns a boolean indicating whether the stack is empty.

Stack operations (cont.)

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size( )	2	(5, 3)
pop( )	3	(5)
isEmpty( )	false	(5)
pop( )	5	( )
isEmpty( )	true	( )
pop( )	null	( )
push(7)	–	(7)
push(9)	–	(7, 9)
top( )	9	(7, 9)
push(4)	–	(7, 9, 4)
size( )	3	(7, 9, 4)
pop( )	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop( )	8	(7, 9, 6)

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the JVM
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures



Stack Presentation of Recursion

Stack Implementation

Stack interface

```
1  /**
2   * A collection of objects that are inserted and removed according to the last-in
3   * first-out principle. Although similar in purpose, this interface differs from
4   * java.util.Stack.
5   *
6   * @author Michael T. Goodrich
7   * @author Roberto Tamassia
8   * @author Michael H. Goldwasser
9   */
10 public interface Stack<E> {
11
12   /**
13    * Returns the number of elements in the stack.
14    * @return number of elements in the stack
15    */
16   int size();
17
18   /**
19    * Tests whether the stack is empty.
20    * @return true if the stack is empty, false otherwise
21    */
22   boolean isEmpty();
```

Stack interface (cont.)

```
23
24      /**
25       * Inserts an element at the top of the stack.
26       * @param e   the element to be inserted
27       */
28      void push(E e);
29
30      /**
31       * Returns, but does not remove, the element at the top of the stack.
32       * @return top element in the stack (or null if empty)
33       */
34      E top( );
35
36      /**
37       * Removes and returns the top element from the stack.
38       * @return element removed (or null if empty)
39       */
40      E pop( );
41  }
```

Stack Implementation?

Which data structures can be used to implement Stack?

Array-Based

Array-Based Stack Implementation

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

Array-Based Stack Implementation

- Elements are stored in an array with capacity N for some fixed N.
- The bottom element of the stack is always stored in cell data[0],
- The top element of the stack in cell data[t] for index t (t<= the current size of the stack).

Array-Based Stack

```
1  public class ArrayStack<E> implements Stack<E> {
2    public static final int CAPACITY=1000;  // default array capacity
3    private E[ ] data;                      // generic array used for storage
4    private int t = −1;                     // index of the top element in stack
5    public ArrayStack( ) { this(CAPACITY); } // constructs stack with default capacity
6    public ArrayStack(int capacity) {       // constructs stack with given capacity
7      data = (E[ ]) new Object[capacity];   // safe cast; compiler may give warning
8    }
9    public int size( ) { return (t + 1); }
10   public boolean isEmpty( ) { return (t == −1); }
11   public void push(E e) throws IllegalStateException {
12     if (size( ) == data.length) throw new IllegalStateException("Stack is full");
13     data[++t] = e;                        // increment t before storing new item
14   }
```

Array-Based Stack (cont.)

```
15    public E top( ) {
16      if (isEmpty( )) return null;
17      return data[t];
18    }
19    public E pop( ) {
20      if (isEmpty( )) return null;
21      E answer = data[t];
22      data[t] = null;                    // dereference to help garbage collection
23      t--;
24      return answer;
25    }
26  }
```

Array-Based Stack (cont.)

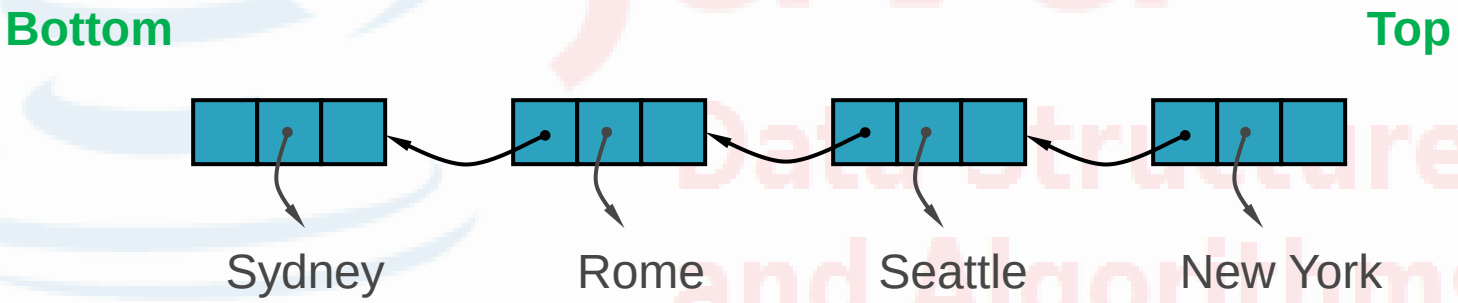
- Analyzing the Array-Based Stack Implementation

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

LinkedList-Based

Singly Linked List-based Stack

- The top of the stack is the head of the linked list.
- Push: create a new node and add it at the top of the stack.
- Pop: delete the node at the top of the stack.



Singly Linked List-based Stack

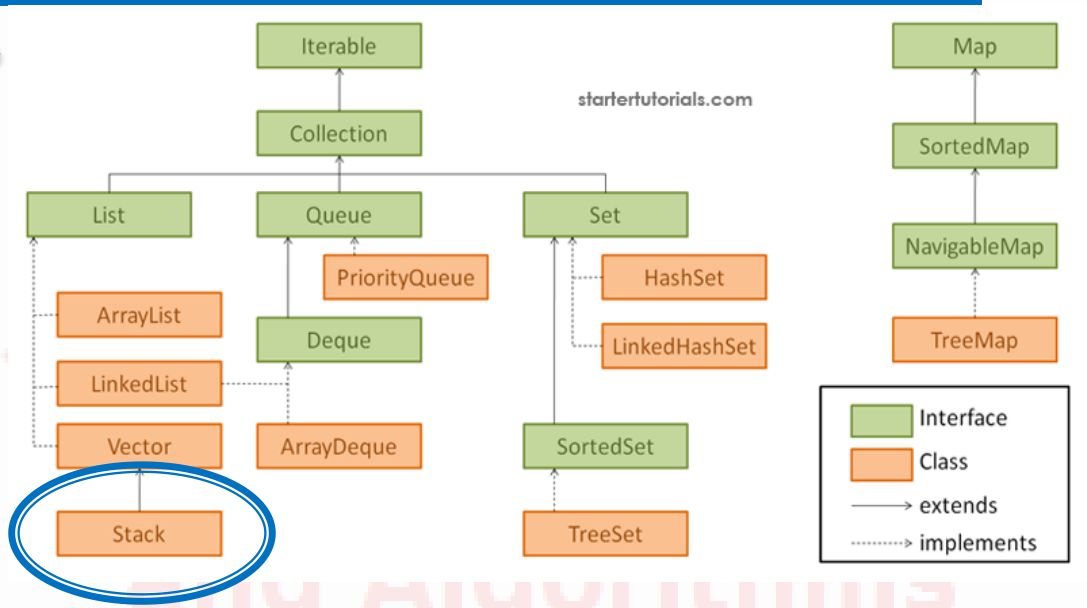
- Methods:

Stack Method	*Singly Linked List Method*
size( )	list.size( )
isEmpty( )	list.isEmpty( )
push(*e*)	list.addFirst(*e*)
pop( )	list.removeFirst( )
top( )	list.first( )

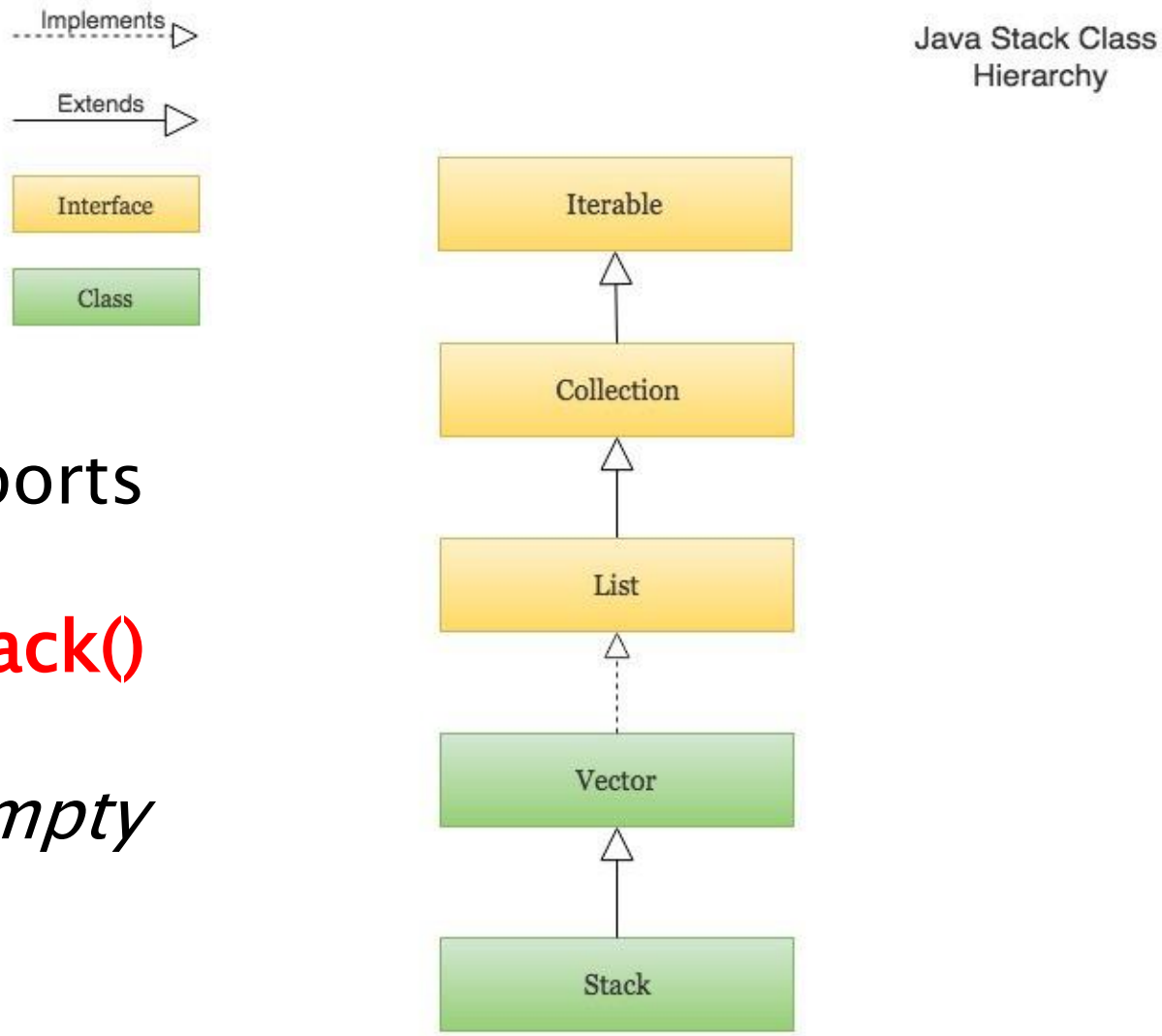
Singly Linked List–based Stack

```
1  public class LinkedStack<E> implements Stack<E> {
2    private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty list
3    public LinkedStack() { }                  // new stack relies on the initially empty list
4    public int size() { return list.size(); }
5    public boolean isEmpty() { return list.isEmpty(); }
6    public void push(E element) { list.addFirst(element); }
7    public E top() { return list.first(); }
8    public E pop() { return list.removeFirst(); }
9  }
```

Stack in Java Collection Framework

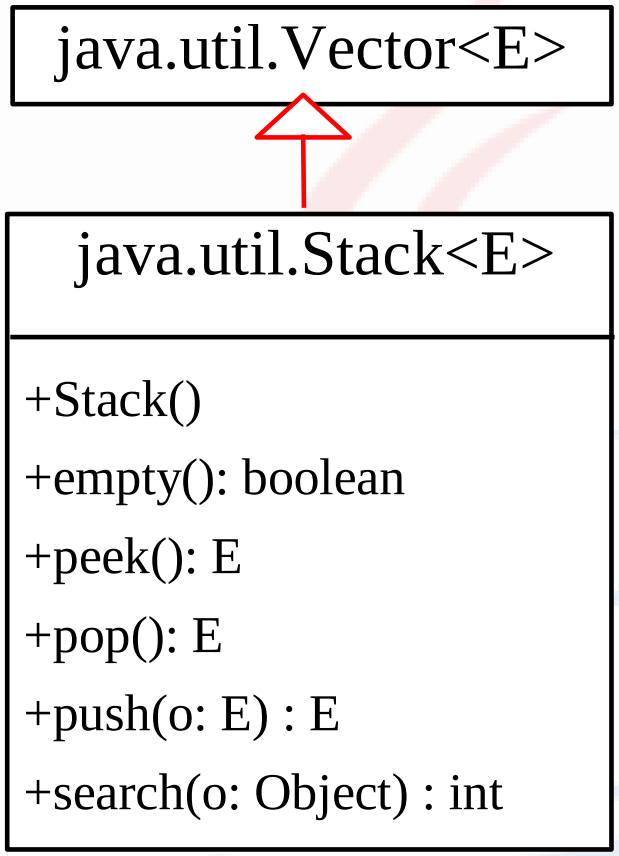


Stack in Java Collection Framework



- The class supports one *default constructor* **Stack()** which is used to *create an empty stack*.

Stack in Java … (cont.)



Creates an empty stack.

Returns true if this stack is empty.

Returns the top element in this stack.

Returns and removes the top element in this stack.

Adds a new element to the top of this stack.

Returns the position of the specified element in this stack.

Stack in Java Collection Framework:
methods

- **E push(*E element*) :**
  - Pushes an element on the top of the stack.
- **E pop():**
  - Removes and returns the top element of the stack.
  - An 'EmptyStackException' exception is thrown if we call pop() when the invoking stack is empty.
- **E peek():**
  - Returns the element on the top of the stack, but does not remove it.
- **boolean empty():**
  - It returns true if nothing is on the top of the stack.
  - Else, returns false.

Stack operation (cont.)

- **int search(*element*)**: It determines whether an object exists in the stack.
  - If the element is found, it returns the position of the element from the top of the stack.
  - Else, it returns -1.

Stack usage

```
// Creating a Stack
Stack<String> stackOfCards = new Stack<String>();

// Pushing new items to the Stack
stackOfCards.push("Jack");
stackOfCards.push("Queen");
stackOfCards.push("King");
stackOfCards.push("Ace");

System.out.println("Stack => " + stackOfCards);
System.out.println();
```

 # Output

Stack => [Jack, Queen, King, Ace]

Stack usage (cont.)

```
// Popping items from the Stack
// Throws EmptyStackException if the stack is empty
String cardAtTop = stackOfCards.pop();
System.out.println("Stack.pop() => " + cardAtTop);
System.out.println("Current Stack => " + stackOfCards);
System.out.println();

// Get the item at the top of the stack without removing it
cardAtTop = stackOfCards.peek();
System.out.println("Stack.peek() => " + cardAtTop);
System.out.println("Current Stack => " + stackOfCards);
```

 # Output

Stack.pop() => Ace
Current Stack => [Jack, Queen, King]

Stack.peek() => King
Current Stack => [Jack, Queen, King]

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 int 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;
}
```

Exercise 1

- How to reverse an array using a stack?

```
1  /** Tester routine for reversing arrays */
2  public static void main(String args[ ]) {
3    Integer[ ] a = {4, 8, 15, 16, 23, 42};      // autoboxing allows this
4    String[ ] s = {"Jack", "Kate", "Hurley", "Jin", "Michael"};
5    System.out.println("a = " + Arrays.toString(a));
6    System.out.println("s = " + Arrays.toString(s));
7    System.out.println("Reversing...");
8    reverse(a);
9    reverse(s);
10   System.out.println("a = " + Arrays.toString(a));
11   System.out.println("s = " + Arrays.toString(s));
12 }
```

The output from this method is the following:

```
a = [4, 8, 15, 16, 23, 42]
s = [Jack, Kate, Hurley, Jin, Michael]
Reversing...
a = [42, 23, 16, 15, 8, 4]
s = [Michael, Jin, Hurley, Kate, Jack]
```

Reversing an Array using Stack

```
1  /** A generic method for reversing an array. */
2  public static <E> void reverse(E[ ] a) {
3    Stack<E> buffer = new ArrayStack<>(a.length);
4    for (int i=0; i < a.length; i++)
5      buffer.push(a[i]);
6    for (int i=0; i < a.length; i++)
7      a[i] = buffer.pop( );
8  }
```

Exercise 2

- Matching Parentheses:
  - Parentheses: “(” and “)”
  - Braces: “{” and “}”
  - Brackets: “[” and “]”

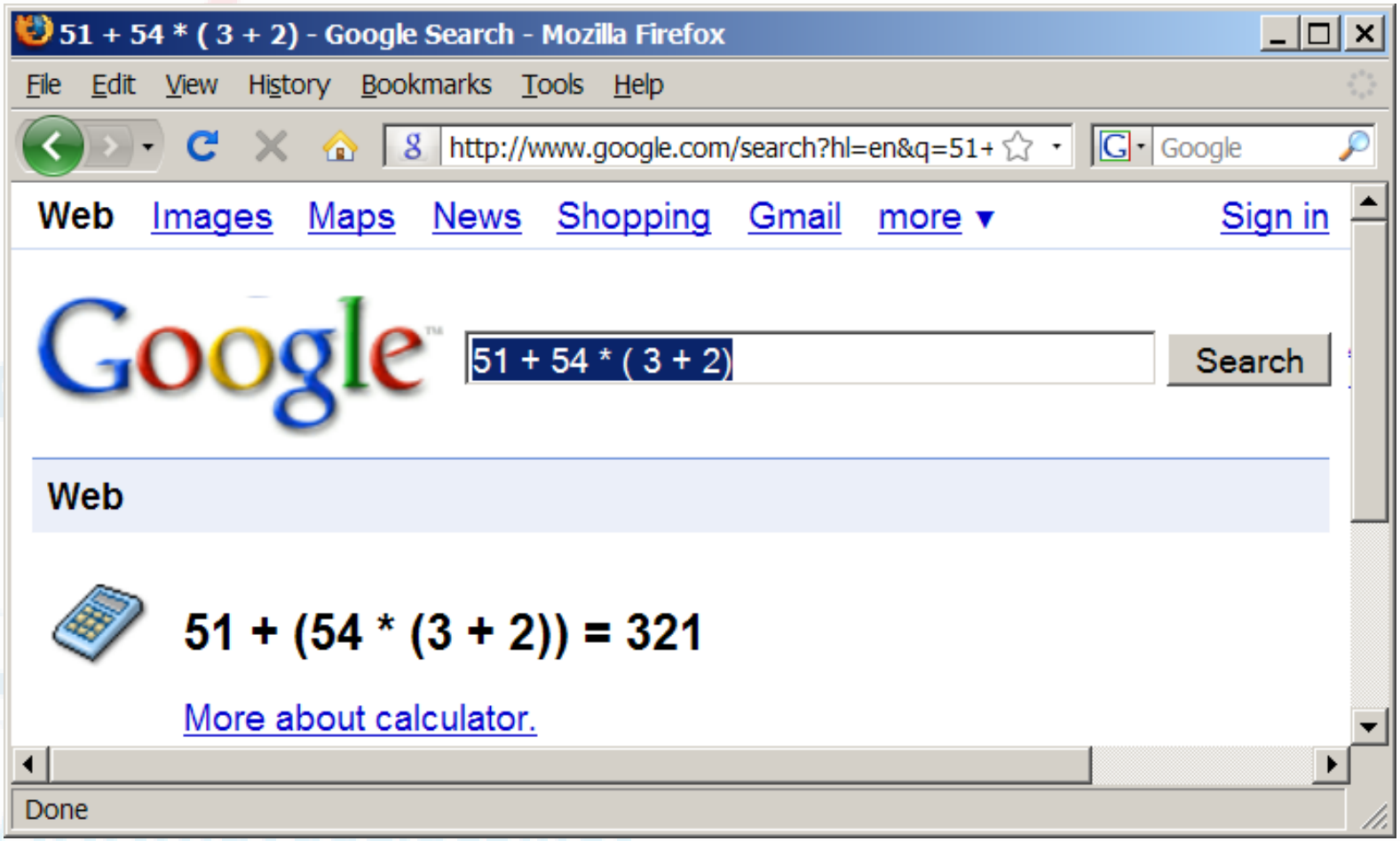
- Correct: ( )(( )){ ([( )]) }
- Correct: (( ) ( )){ ([( )]) })
- Incorrect:) (( )){ ([( )]) }
- Incorrect: ({ [ ]) }
- Incorrect: (

- Each opening symbol must match its corresponding closing symbol.
- For example, a left bracket, “[” must match a corresponding right bracket, “]”

$$[(5+x)-(y+z)].$$

Exercise 3

- Evaluating Expressions



Algorithm

Phase 1: Scanning the expression

The program scans the expression from left to right to extract operands, operators, and the parentheses.

1.1. If the extracted item is an operand, push it to **operandStack**.

1.2. If the extracted item is a + or – operator, process all the operators at the top of **operatorStack** and push the extracted operator to **operatorStack**.

1.3. If the extracted item is a * or / operator, process the * or / operators at the top of **operatorStack** and push the extracted operator to **operatorStack**.

1.4. If the extracted item is a (symbol, push it to **operatorStack**.

1.5. If the extracted item is a) symbol, repeatedly process the operators from the top of **operatorStack** until seeing the (symbol on the stack.

Phase 2: Clearing the stack

Repeatedly process the operators from the top of **operatorStack** until **operatorStack** is empty.

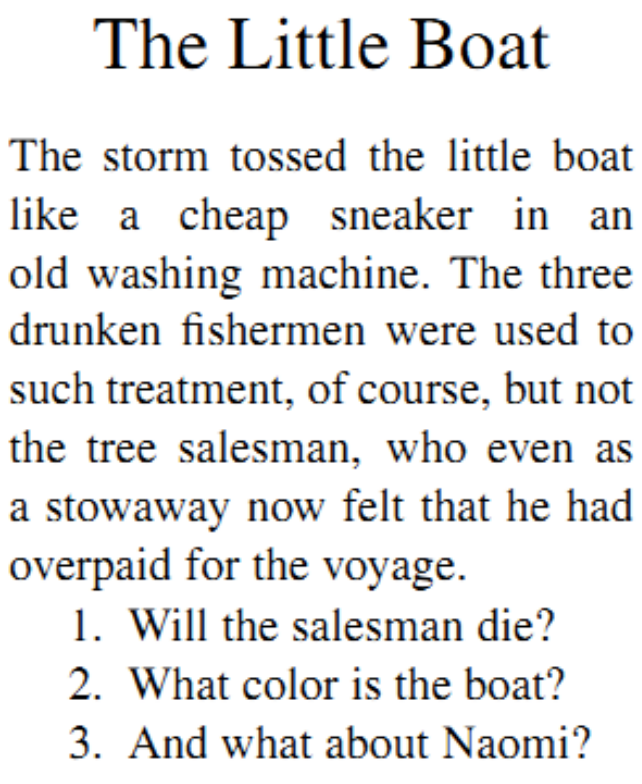
Example

Expression	*Scan*	*Action*	operandStack	operatorStack
(1 + 2)*4 − 3 ↑	(	Phase 1.4		(
(1 + 2)*4 − 3 ↑	1	Phase 1.1	1	(
(1 + 2)*4 − 3 ↑	+	Phase 1.2	1	+ (
(1 + 2)*4 − 3 ↑	2	Phase 1.1	2 1	(
(1 + 2)*4 − 3 ↑	)	Phase 1.5	3	
(1 + 2)*4 − 3 ↑	*	Phase 1.3	3	*
(1 + 2)*4 − 3 ↑	4	Phase 1.1	4 3	*
(1 + 2)*4 − 3 ↑	−	Phase 1.2	12	−
(1 + 2)*4 − 3 ↑	3	Phase 1.1	3 12	−
(1 + 2)*4 − 3 ↑	none	Phase 2	9	

Exercise 4.

- Matching Tags in a Markup Language (Illustrating (a) an HTML document and (b) its rendering)

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine.  The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```



Exercise 4 (cont.)

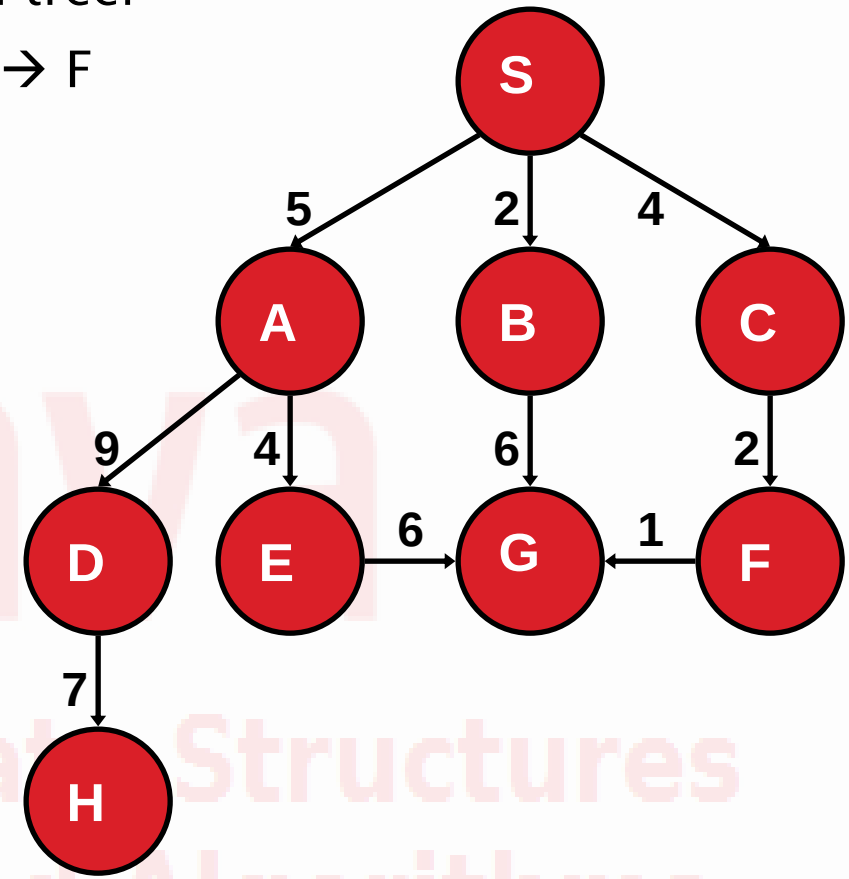
- <body>: document body
- <h1>: section header
- : center justify
- <p>: paragraph
- : numbered (ordered) list
- : list item

Other application

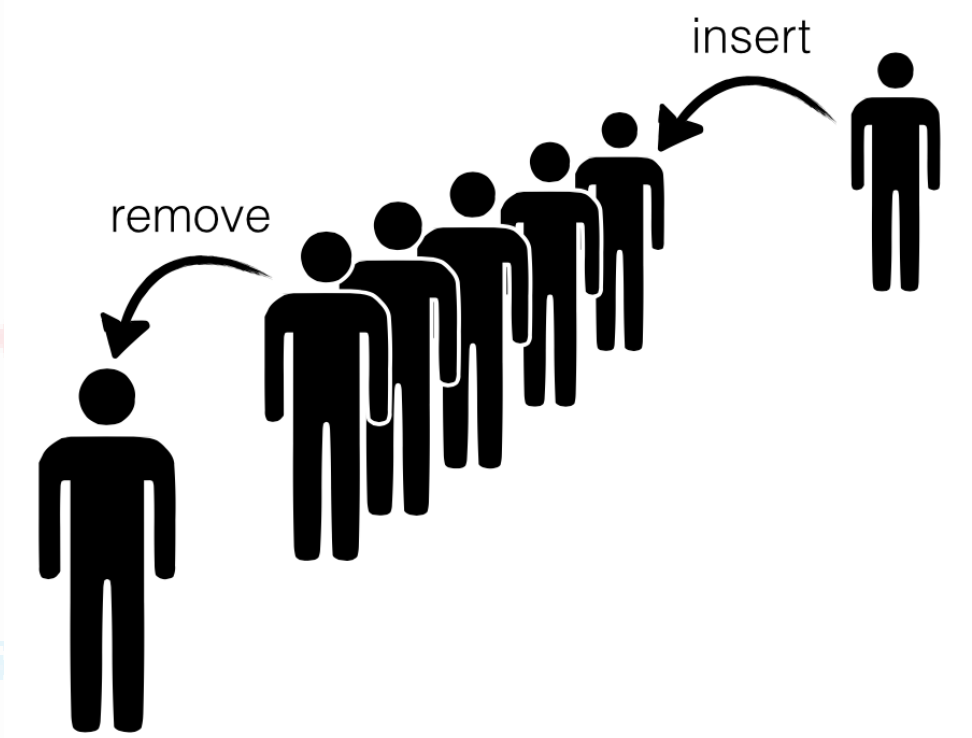
- "**undo**" mechanism in text editors;
  - this operation is accomplished by keeping all text changes in a stack.

Deapth-First Search with a Stack

- Deapth-First Traversal of the given tree:
- S → A → D → H → E → G → B → C → F

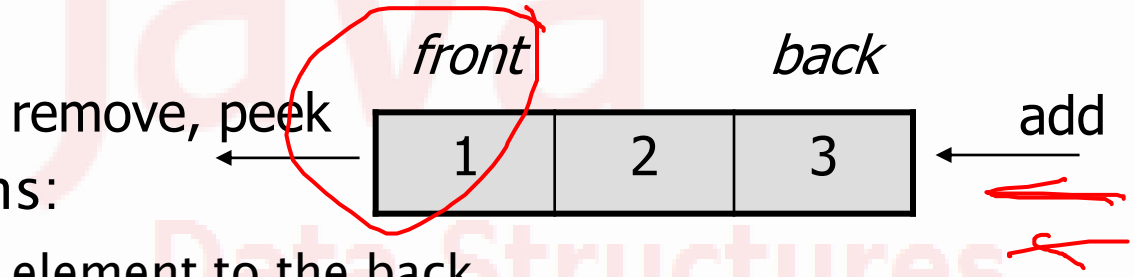


Queue



Queues

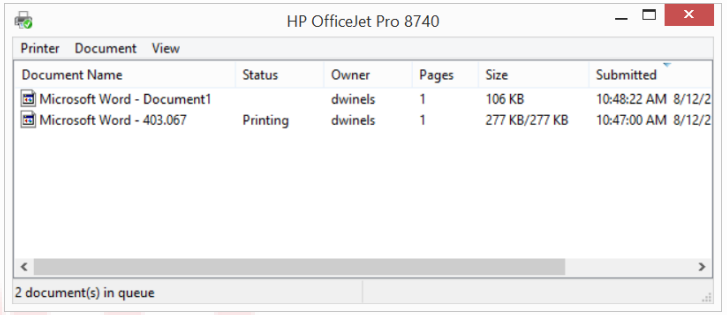
- **Queue**: Retrieves elements in the order they were added.
  - First-In, First-Out ("FIFO")
  - Elements are stored in order of insertion but don't have indexes.
  - Client can only add to the end of the queue, and can only examine/remove the front of the queue.



- Basic queue operations:
  - **add** (enqueue): Add an element to the back.
  - **remove** (dequeue): Remove the front element.
  - **peek**: Examine the front element.

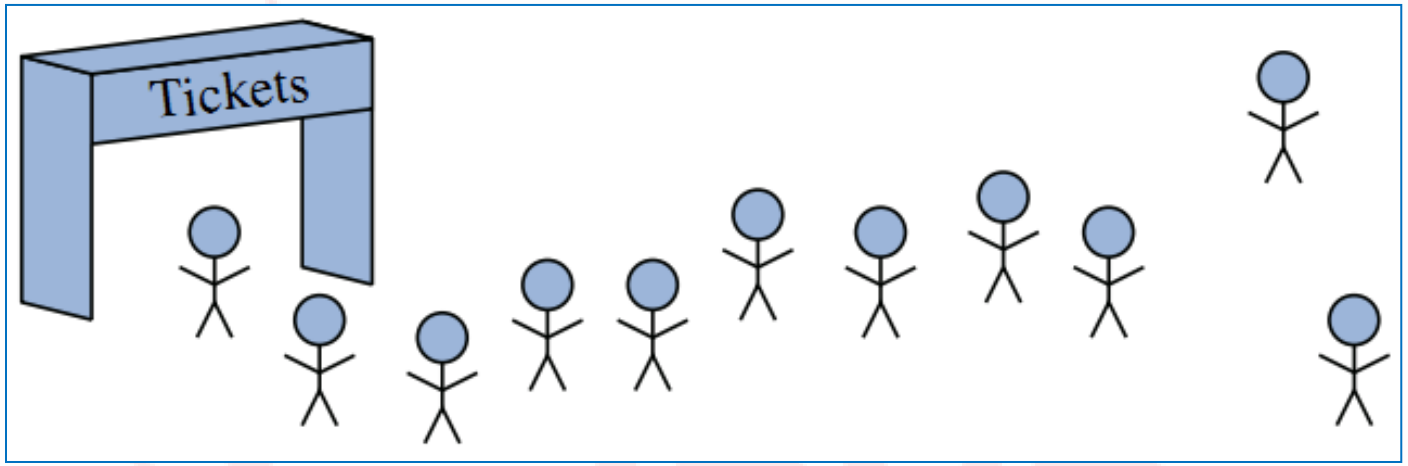
Applications of Queues

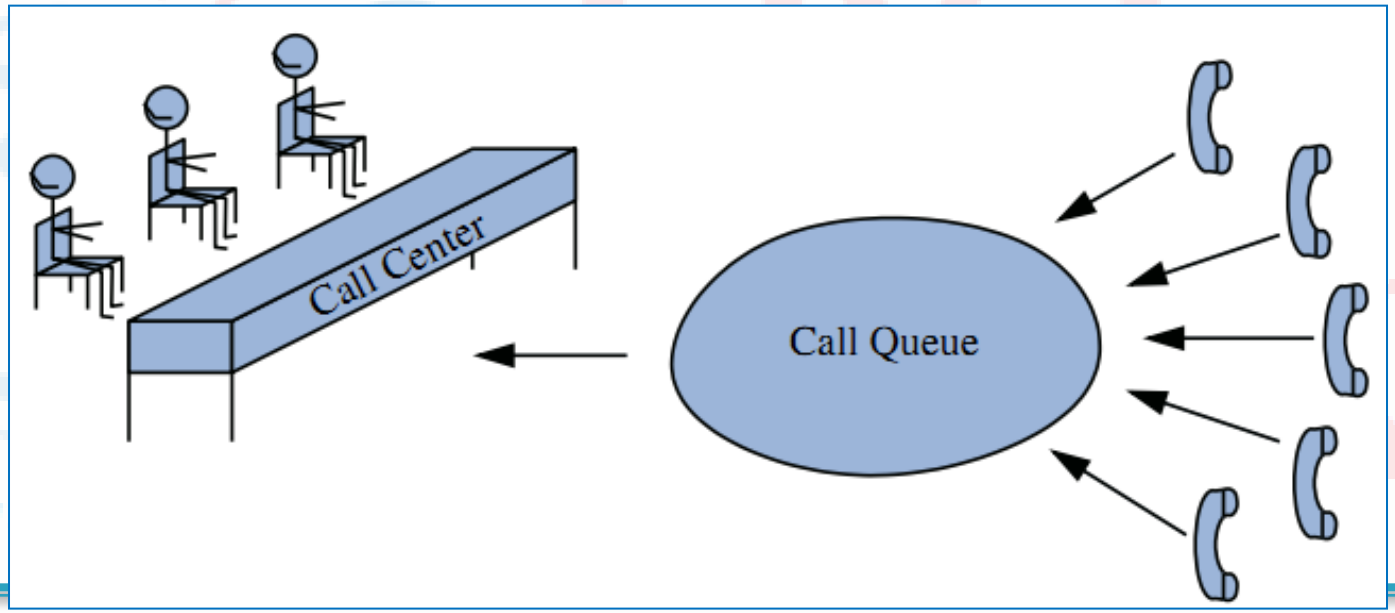
- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming



- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Queue in practice





Queue ADT methods

enqueue(e):	Adds element e to the back of queue.
dequeue( ):	Removes and returns the first element from the queue (or null if the queue is empty).
first( ):	Returns the first element of the queue, without removing it (or null if the queue is empty).
size( ):	Returns the number of elements in the queue.
isEmpty( ):	Returns a boolean indicating whether the queue is empty.

Queue Implementation

Queue Implementation

```
1   public interface Queue<E> {
2     /** Returns the number of elements in the queue. */
3     int size( );
4     /** Tests whether the queue is empty. */
5     boolean isEmpty( );
6     /** Inserts an element at the rear of the queue. */
7     void enqueue(E e);
8     /** Returns, but does not remove, the first element of the queue (null if empty). */
9     E first( );
10    /** Removes and returns the first element of the queue (null if empty). */
11    E dequeue( );
12  }
```

Queue operations Example

Method	Return Value	first $\leftarrow Q \leftarrow$ last
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
size( )	2	(5, 3)
dequeue( )	5	(3)
isEmpty( )	false	(3)
dequeue( )	3	( )
isEmpty( )	true	( )
dequeue( )	null	( )
enqueue(7)	–	(7)
enqueue(9)	–	(7, 9)
first( )	7	(7, 9)
enqueue(4)	–	(7, 9, 4)

Queue Implementation?

Which data structures can be used to implement Queue?

Array-Based

Array-Based Queue Implementation

- Elements are stored in an array with capacity N for some fixed N.

- The first element is at index 0, the second element at index 1, and so on

Array-Based Queue Implementation

```
1  /** Implementation of the queue ADT using a fixed-length array. */
2  public class ArrayQueue<E> implements Queue<E> {
3    // instance variables
4    private E[ ] data;                          // generic array used for storage
5    private int f = 0;                          // index of the front element
6    private int sz = 0;                         // current number of elements
7
8    // constructors
9    public ArrayQueue( ) {this(CAPACITY);} // constructs queue with default capacity
10   public ArrayQueue(int capacity) {          // constructs queue with given capacity
11     data = (E[ ]) new Object[capacity];      // safe cast; compiler may give warning
12   }
13
14   // methods
15   /** Returns the number of elements in the queue. */
16   public int size( ) { return sz; }
17
18   /** Tests whether the queue is empty. */
19   public boolean isEmpty( ) { return (sz == 0); }
20
```

Array-Based Queue Implementation (cont.)

```
20
21    /** Inserts an element at the rear of the queue. */
22    public void enqueue(E e) throws IllegalStateException {
23      if (sz == data.length) throw new IllegalStateException("Queue is full");
24      int avail = (f + sz) % data.length;       // use modular arithmetic
25      data[avail] = e;
26      sz++;
27    }
28
29    /** Returns, but does not remove, the first element of the queue (null if empty). */
30    public E first( ) {
31      if (isEmpty( )) return null;
32      return data[f];
33    }
34
35    /** Removes and returns the first element of the queue (null if empty). */
36    public E dequeue( ) {
37      if (isEmpty( )) return null;
38      E answer = data[f];
39      data[f] = null;                           // dereference to help garbage collection
40      f = (f + 1) % data.length;
41      sz--;
42      return answer;
43    }
```

Array-Based Queue Implementation (cont.)

- Analyzing the Efficiency of an Array-Based Queue

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
first	$O(1)$
enqueue	$O(1)$
dequeue	$O(1)$

LinkedList-Based

Queue Implementation:
With a Singly Linked List

```
1  /** Realization of a FIFO queue as an adaptation of a SinglyLinkedList. */
2  public class LinkedQueue<E> implements Queue<E> {
3    private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty list
4    public LinkedQueue() { }                    // new queue relies on the initially empty list
5    public int size() { return list.size(); }
6    public boolean isEmpty() { return list.isEmpty(); }
7    public void enqueue(E element) { list.addLast(element); }
8    public E first() { return list.first(); }
9    public E dequeue() { return list.removeFirst(); }
10 }
```

Analyzing the Efficiency of a Linked Queue

- Each method of LinkedQueue adaptation also runs in O(1) worst-case time.

Circular Queue Implementation

- Circularly linked list class that supports all behaviors of a singly linked list, and an additional rotate( ) method that efficiently moves the first element to the end of the list

```
1  public interface CircularQueue<E> extends Queue<E> {
2    /**
3     * Rotates the front element of the queue to the back of the queue.
4     * This does nothing if the queue is empty.
5     */
6    void rotate( );
7  }
```

Joseplus Problem

- A group of children sit in a circle passing an object, called “potato”, around the circle.
- The potato begins with a starting child in the circle, and the children continue passing the potato until a leader rings a bell, at which point the child holding the potato must leave the game after handing the potato to the next child in the circle.
- After the selected child leaves, the other children close up the circle.

Joseplus Problem (cont.)

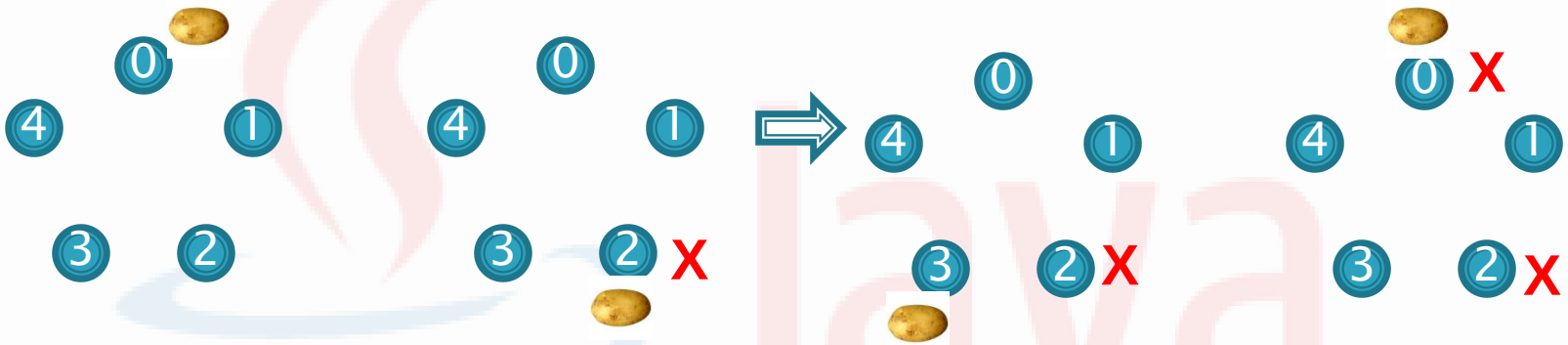
- This process then continues until there is only child remaining, who is declared the winner.

- If the leader always uses the strategy of ringing the bell after the potato has been passed k times, for some fixed ***k***, determining the winner for a given list of children is known as the josephus problem

Josephus Problem – Example

- M=2, N=5

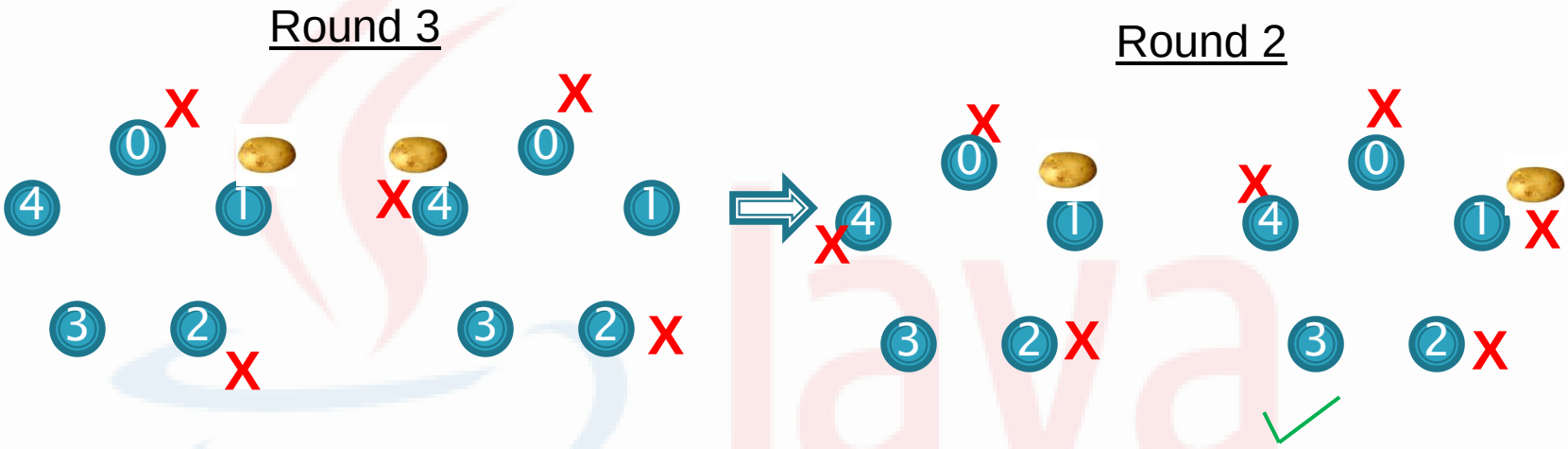
Initial state: Round 1 Round 2



Person removed so far: 2, 0,

Problem – Example

- M=2, N=5



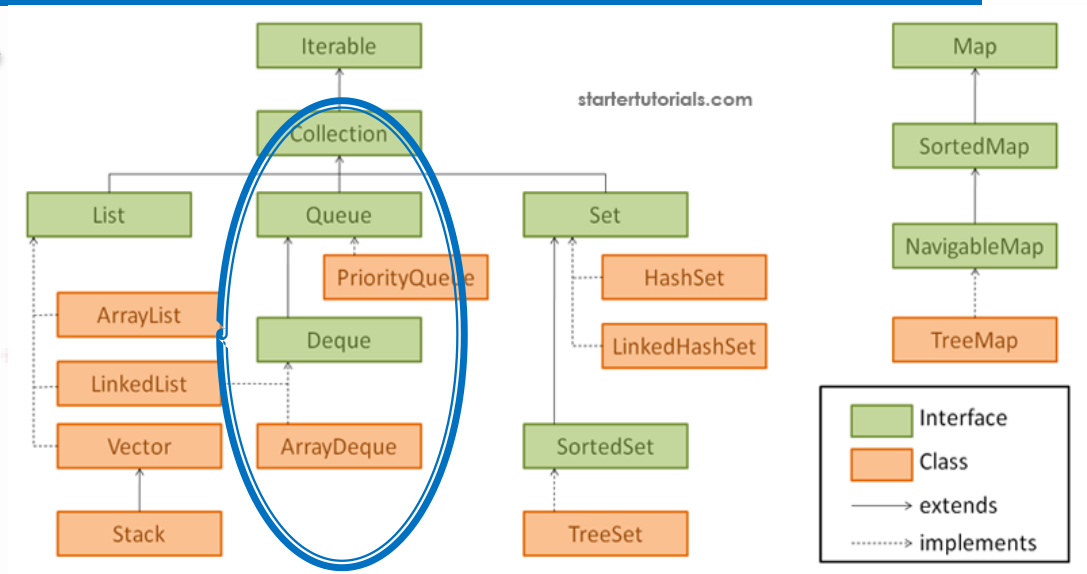
Person removed so far: 2, 0, 4, 1

Winner is 3

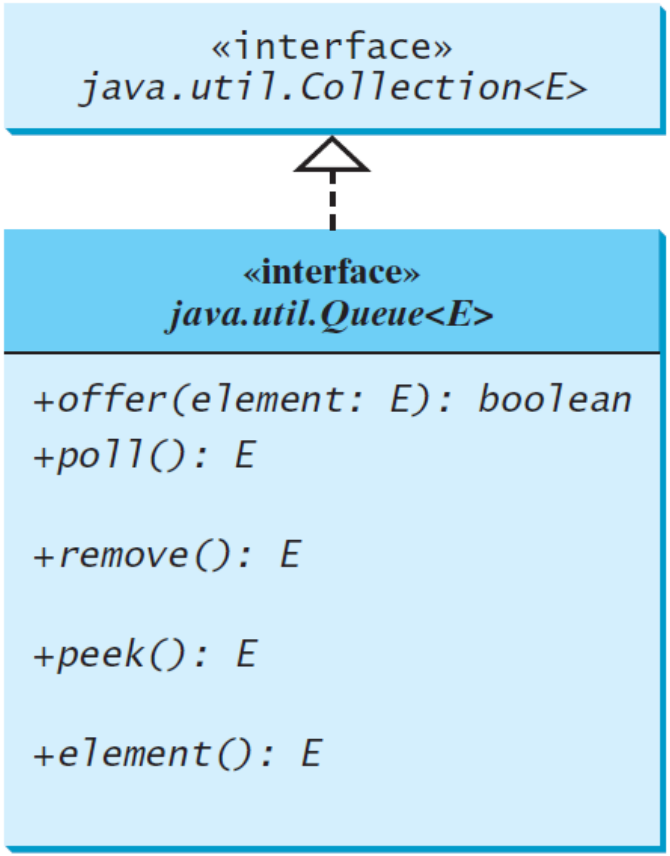
Joseplus Problem

```
1  public class Josephus {
2    /** Computes the winner of the Josephus problem using a circular queue. */
3    public static <E> E Josephus(CircularQueue<E> queue, int k) {
4      if (queue.isEmpty( )) return null;
5      while (queue.size( ) > 1) {
6        for (int i=0; i < k−1; i++)    // skip past k-1 elements
7          queue.rotate( );
8        E e = queue.dequeue( );        // remove the front element from the collection
9        System.out.println("       " + e + " is out");
10       }
11     return queue.dequeue( );         // the winner
12   }
13
14   /** Builds a circular queue from an array of objects. */
15   public static <E> CircularQueue<E> buildQueue(E a[ ]) {
16     CircularQueue<E> queue = new LinkedCircularQueue<>( );
17     for (int i=0; i<a.length; i++)
18       queue.enqueue(a[i]);
19     return queue;
20   }
```

Queue in Java Collection Framework



Queue implementation in Java



Inserts an element into the queue.	
Retrieves and removes the head of this queue, or `null` if this queue is empty.	
Retrieves and removes the head of this queue and throws an exception if this queue is empty.	
Retrieves, but does not remove, the head of this queue, returning `null` if this queue is empty.	
Retrieves, but does not remove, the head of this queue, throws an exception if this queue is empty.	

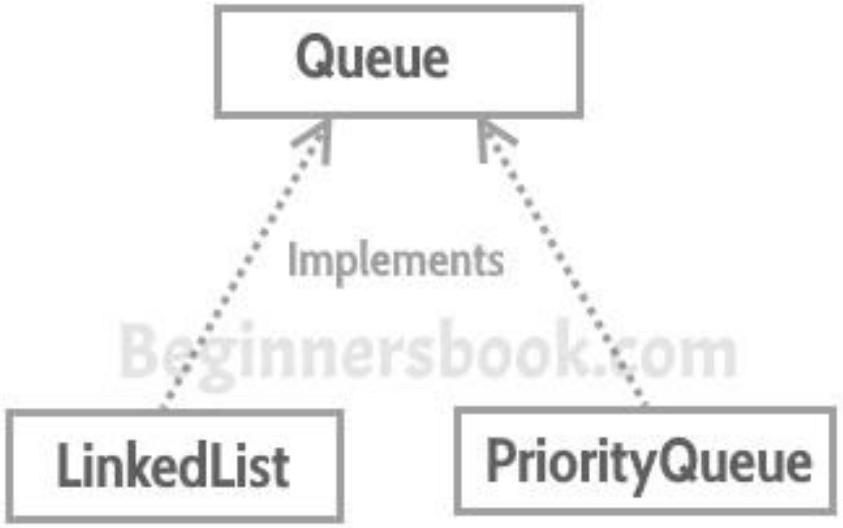
Queue ADT vs java.util.Queue

Our Queue ADT	Interface java.util.Queue	
	throws exceptions	returns special value
enqueue(e)	add(e)	offer(e)
dequeue( )	remove( )	poll( )
first( )	element( )	peek( )
size( )	size( )	
isEmpty( )	isEmpty( )	

- If a queue is empty:
  - remove( ) and element( ) methods throw a NoSuchElementException,
  - poll( ) and peek( ) return null

Queue implementation in Java (cont.)

- Queue interface in Java collections has two implementation:



LinkedList & PriorityQueue Classes implements Queue Interface.

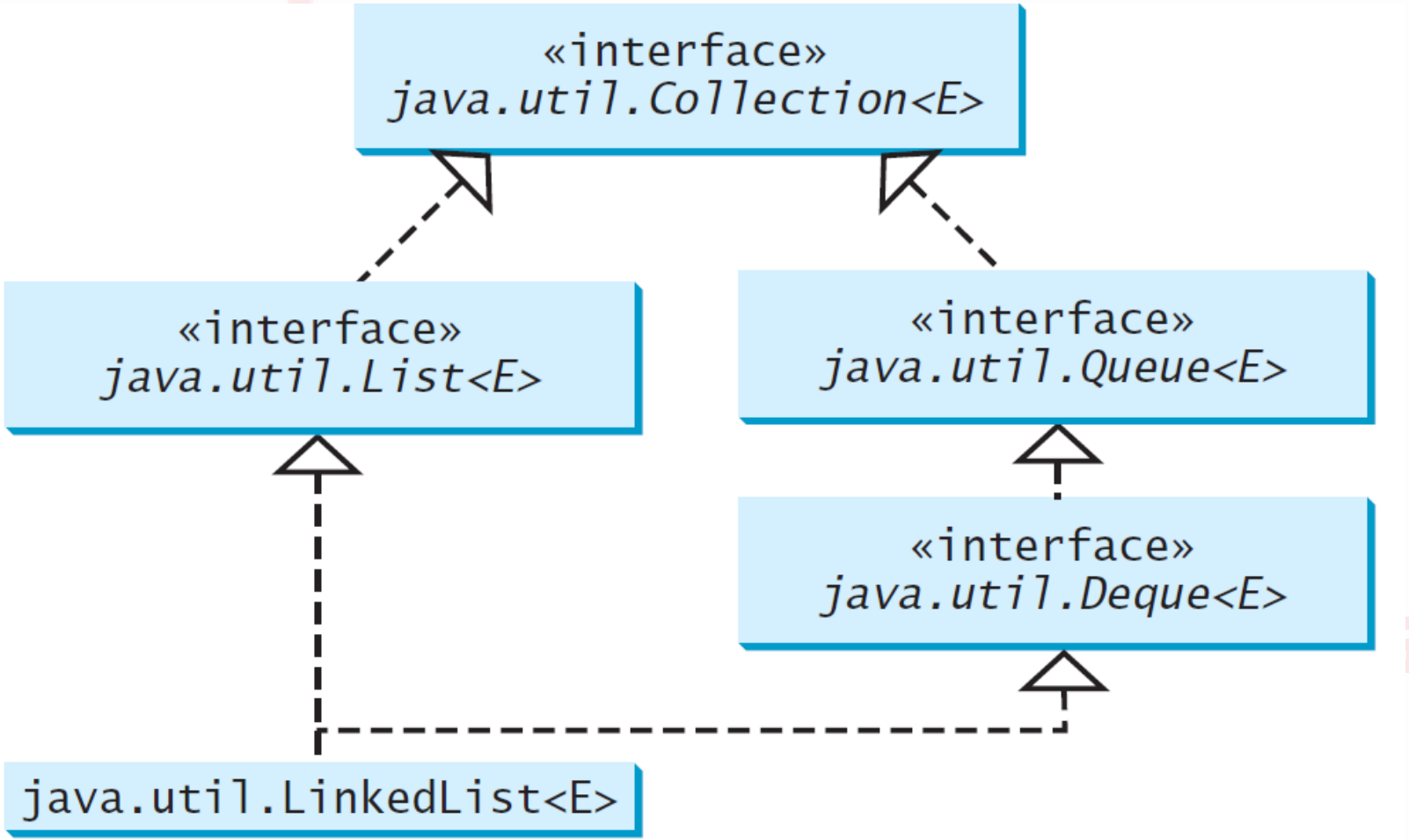
Methods

- **boolean add(E e)**:
  - adds the specified element at the end of Queue.
  - Returns true if the the element is added successfully;
  - Or false if the element is not added that basically happens when the Queue is at its max capacity and cannot take any more elements.
- **E element()**:
  - returns the head (the first element) of the Queue.
- **boolean offer(object)**:
  - same as add() method.

Methods (cont.)

- **E remove()**:
  - removes the head(first element) of the Queue and returns its value.
- **E poll()**:
  - almost same as remove() method. The only difference between poll() and remove() is that poll() method returns null if the Queue is empty.
- **E peek()**:
  - almost same as element() method.
  - The only difference between peek() and element() is that peek() method returns null if the Queue is empty.

Using LinkedList for Queue



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]
```

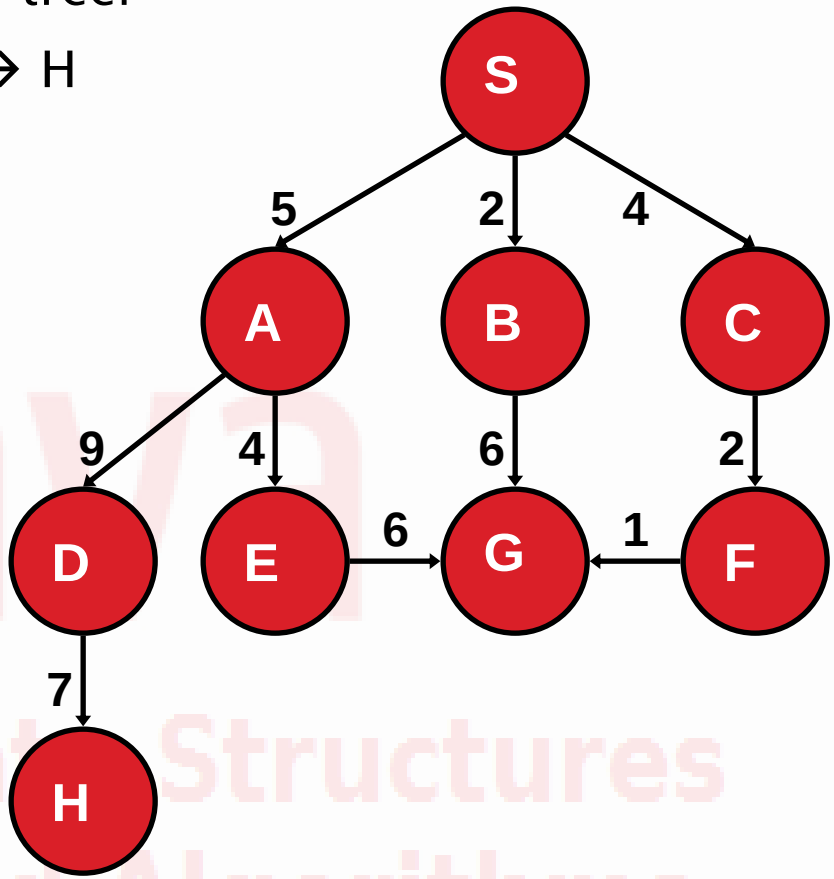
Exercises

- 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`

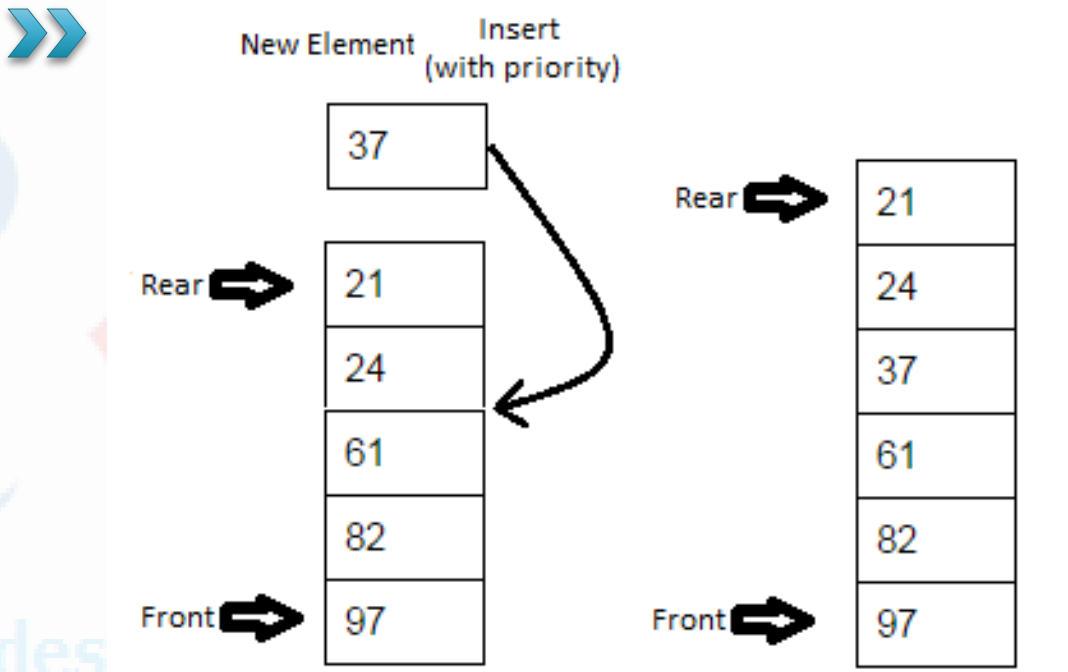
- Write a method **`mirror`** that accepts a queue of strings as a parameter and appends the queue's contents to itself in reverse order.
  - `front [a, b, c] back`
    becomes
    `front [a, b, c, c, b, a] back`

Breadth-First Search with a Queue

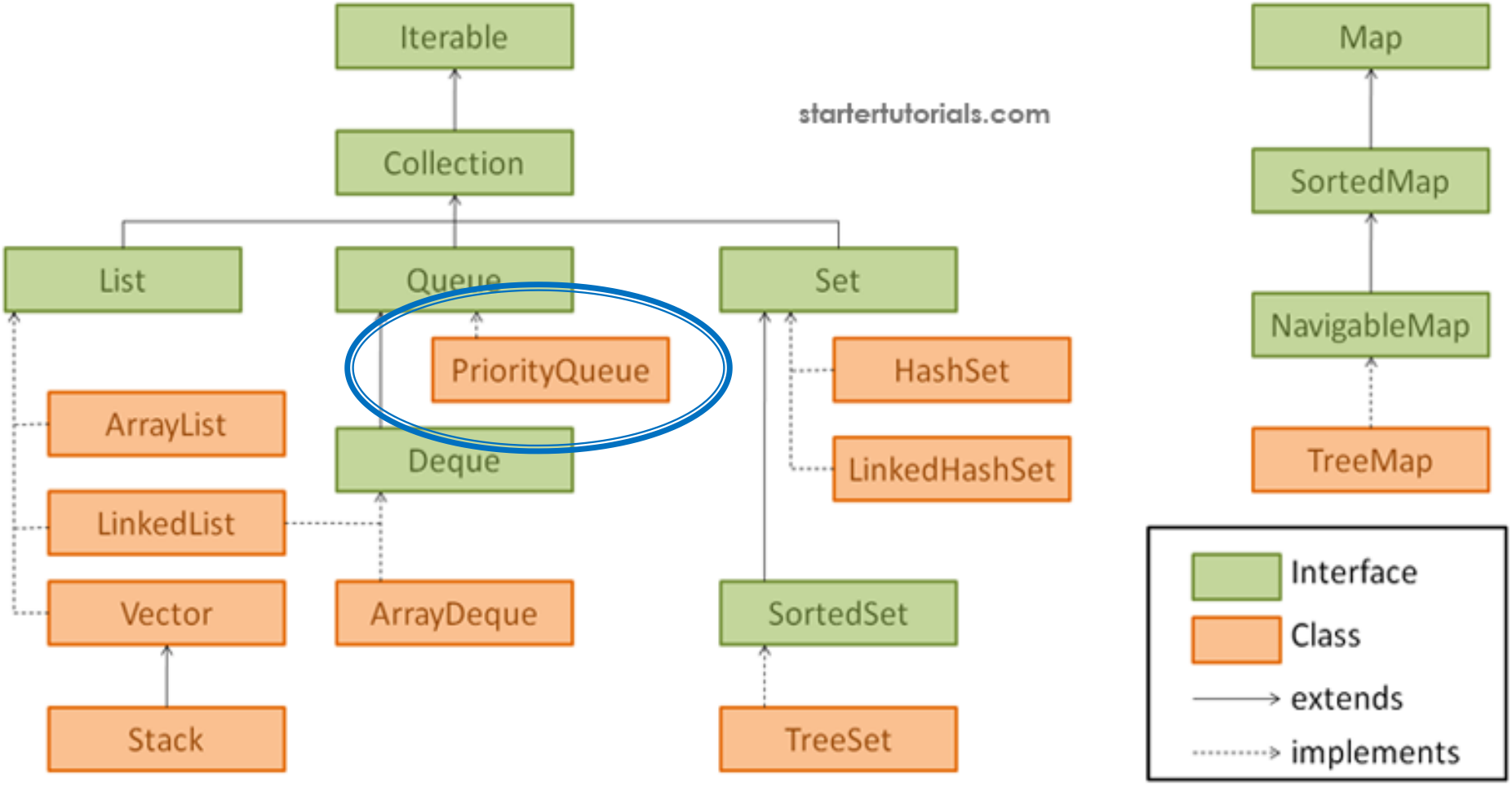
- Breadth-First Traversal of the given tree:
- S → A → B → C → D → E → G → F → H



Priority Queue



Java Collection framework



Definition: Priority Queue ADT

- Recall that a FIFO queue removes elements in the order in which they were added.
- A PriorityQueue removes items in sorted order from lowest to highest value, independent of the order in which they were added
- Priority queues can be implemented by using arrays, linked list and heap.
- A priority queue stores a collection of entries. Each entry is a pair (key, value)

Priority Queue ADT operations

insert(k, v):	Creates an entry with key k and value v in the priority queue.
min( ):	Returns (but does not remove) a priority queue entry (k,v) having minimal key; returns null if the priority queue is empty.
removeMin( ):	Removes and returns an entry (k,v) having minimal key from the priority queue; returns null if the priority queue is empty.
size( ):	Returns the number of entries in the priority queue.
isEmpty( ):	Returns a boolean indicating whether the priority queue is empty.

Example

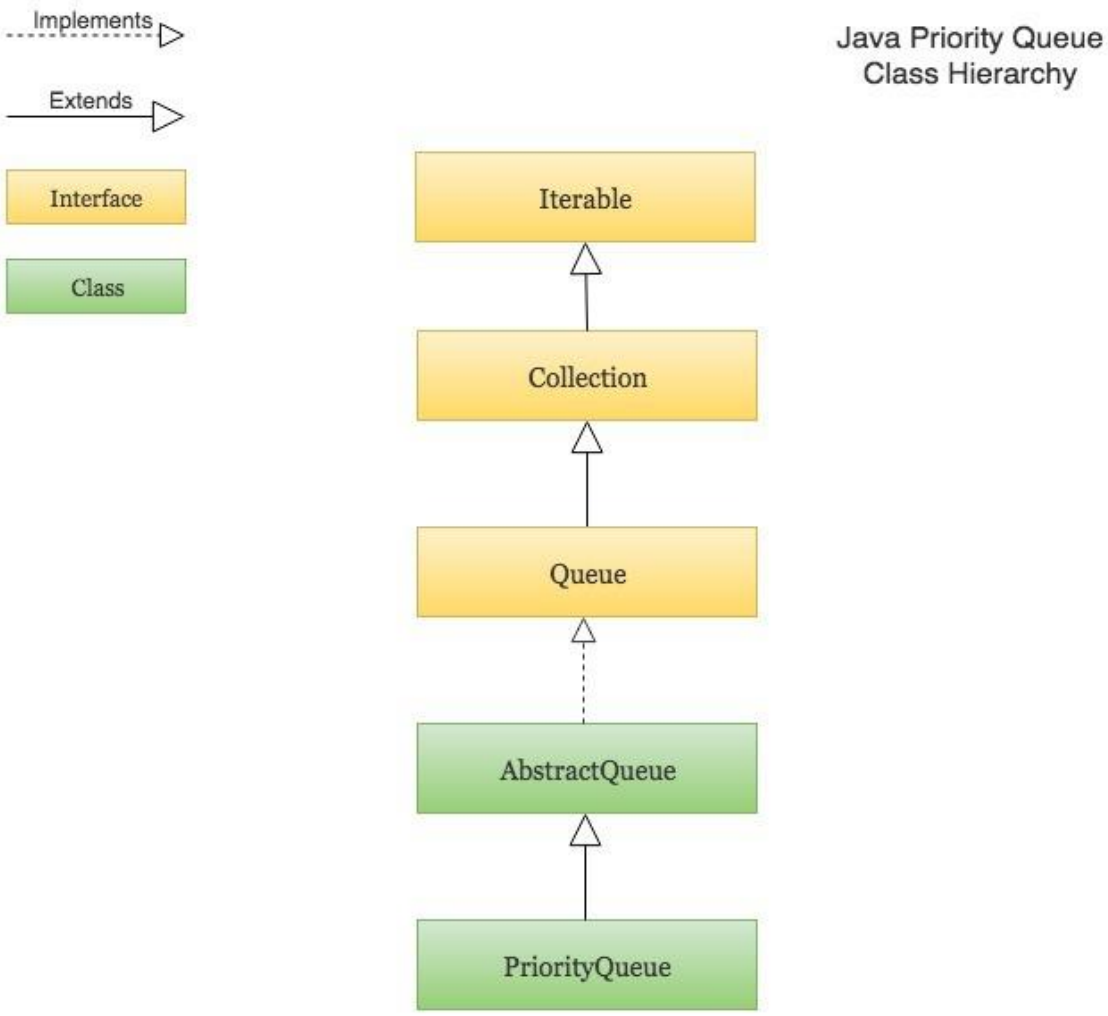
- A sequence of priority queue methods: The numbers are keys (priorities), letters are values (data).

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }
removeMin( )	null	{ }
isEmpty( )	true	{ }

Implementing a Priority Queue

- See Chapter 9: Data Structures and Algorithms in Java 6th Edition

Priority Queue in Java Collection Framework



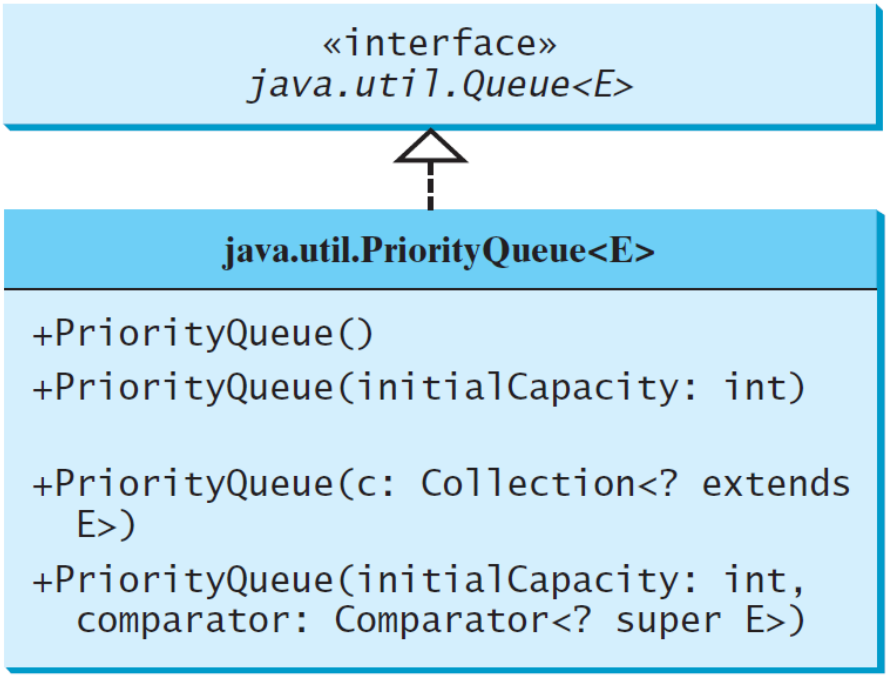
Priority Queue in Java Collection Framework

- Priority queue:
  - retrieves elements in sorted order after they were inserted in arbitrary order.
  - does not sort all its elements.
- A priority queue can either hold elements of a class that implements the Comparable interface or a Comparator object supplying in the constructor (like TreeSet)
- A typical use for a priority queue is job scheduling

Queue ADT vs java.util.PriorityQueue

Our Priority Queue ADT	java.util.PriorityQueue Class
insert(k, v)	add(**new** SimpleEntry(k, v))
min( )	peek( )
removeMin( )	remove( )
size( )	size( )
isEmpty( )	isEmpty( )

The PriorityQueue Class



Creates a default priority queue with initial capacity 11.	
Creates a default priority queue with the specified initial capacity.	
Creates a priority queue with the specified collection.	
Creates a priority queue with the specified initial capacity and the comparator.	

PriorityQueue:
Constructor

- **PriorityQueue()**: Creates a PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering.
- **PriorityQueue(Collection<E> c)**: Creates a PriorityQueue containing the elements in the specified collection.
- **PriorityQueue(int initialCapacity)**: Creates a PriorityQueue with the specified initial capacity that orders its elements according to their natural ordering.
- **PriorityQueue(int initialCapacity, Comparator<E> comparator)**: Creates a PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.
- **PriorityQueue(PriorityQueue<E> c)**: Creates a PriorityQueue containing the elements in the specified priority queue.
- **PriorityQueue(SortedSet<E> c)**: Creates a PriorityQueue containing the elements in the specified sorted set.

PriorityQueue:
Methods

Method	Behavior
`boolean offer(E item)`	Inserts an item into the queue. Returns **`true`** if successful; returns **`false`** if the item could not be inserted.
`E remove()`	Removes the smallest entry and returns it if the queue is not empty. If the queue is empty, throws a `NoSuchElementException`.
`E poll()`	Removes the smallest entry and returns it. If the queue is empty, returns **`null`**.
`E peek()`	Returns the smallest entry without removing it. If the queue is empty, returns **`null`**.
`E element()`	Returns the smallest entry without removing it. If the queue is empty, throws a `NoSuchElementException`.

PriorityQueue usage

```
public static void main(String[] args) {
    PriorityQueue<Integer> pq1 = new PriorityQueue<>();
    PriorityQueue<Integer> pq2 = new PriorityQueue<>();
    // add values
    pq1.add(1);
    pq1.add(5);
    pq1.add(3);
    pq1.offer(2);
    pq1.offer(6);
    pq1.offer(4);

    System.out.println("All values: " + pq1);

    pq2.addAll(pq1);
}
```

PriorityQueue usage (cont.)

```
Object o1 = pq1.peek();
System.out.println("Head of queue: " + o1);
System.out.println("Size after after peek: " + pq1.size());
System.out.println("All values after peek: " + pq1);

int i = pq2.peek();
System.out.println("Head of queue: " + i);
System.out.println("Size after after peek: " + pq2.size());
System.out.println("All values after peek: " + pq2);

// poll will return the value from the head of the queue and will remove the
// element
```

PriorityQueue usage (cont.)

```
// poll will return the value from the head of the queue and will remove the
// element

int x = pq2.poll();
System.out.println("Head of queue: " + x);
System.out.println("Size after after poll: " + pq2.size());
System.out.println("All values after poll: " + pq2);

// element() returns the element at the head of the queue. The element is not
// removed.

Object o2 = pq1.element();
System.out.println(o2);
System.out.println("Size after after element: " + pq2.size());

// remove() removes the element at the head of the queue, returning the element
// in the process.

Object o3 = pq1.remove();
System.out.println(o3);
System.out.println("Size after after remove: " + pq2.size());
```

Notes on PriorityQueue

- An unbounded priority queue based on a priority heap.
- PriorityQueue doesn't permit null.
- We can't create PriorityQueue of Objects that are non-comparable
- The head of the queue is the least element with respect to the specified ordering.
  - If multiple elements are tied for least value, the head is one of those elements — ties are broken arbitrarily.
- The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.
- It inherits methods from AbstractQueue, AbstractCollection, Collection and Object class.

What is Heap?

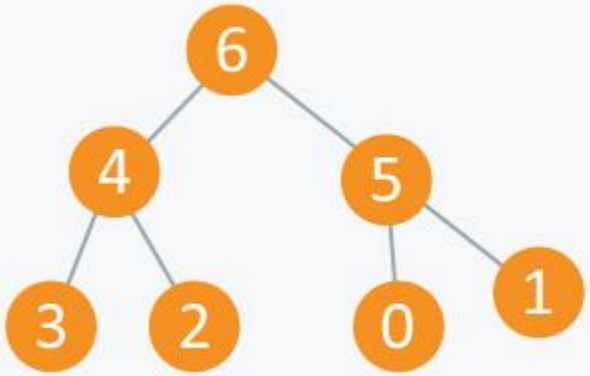
- Heaps: a specific tree based data structure in which all the nodes of tree are in a specific order.

- Let X be a parent node of Y, then the value of X follows some specific order with respect to value of Y and the same order will be followed across the tree.

- The maximum number of children of a node in the heap depends on the type of heap

Binary Heap

- Heap has at most 2 children of a node:



- Each node has greater value than any of its children.
- Suppose there are N Jobs in a queue to be done, and each job has its own priority.
- The job with maximum priority will get completed first than others.

FACULTY OF INFORMATION TECHNOLOGY

Java
Data Structures