



# **COSC 121**

## **Computer Programming II**

### **Data Structures**

### **Lists, Stacks, Queues, and Priority Queues**

*Part 2/2*

**Dr. Mostafa Mohamed**

# Outline

## *Previously:*

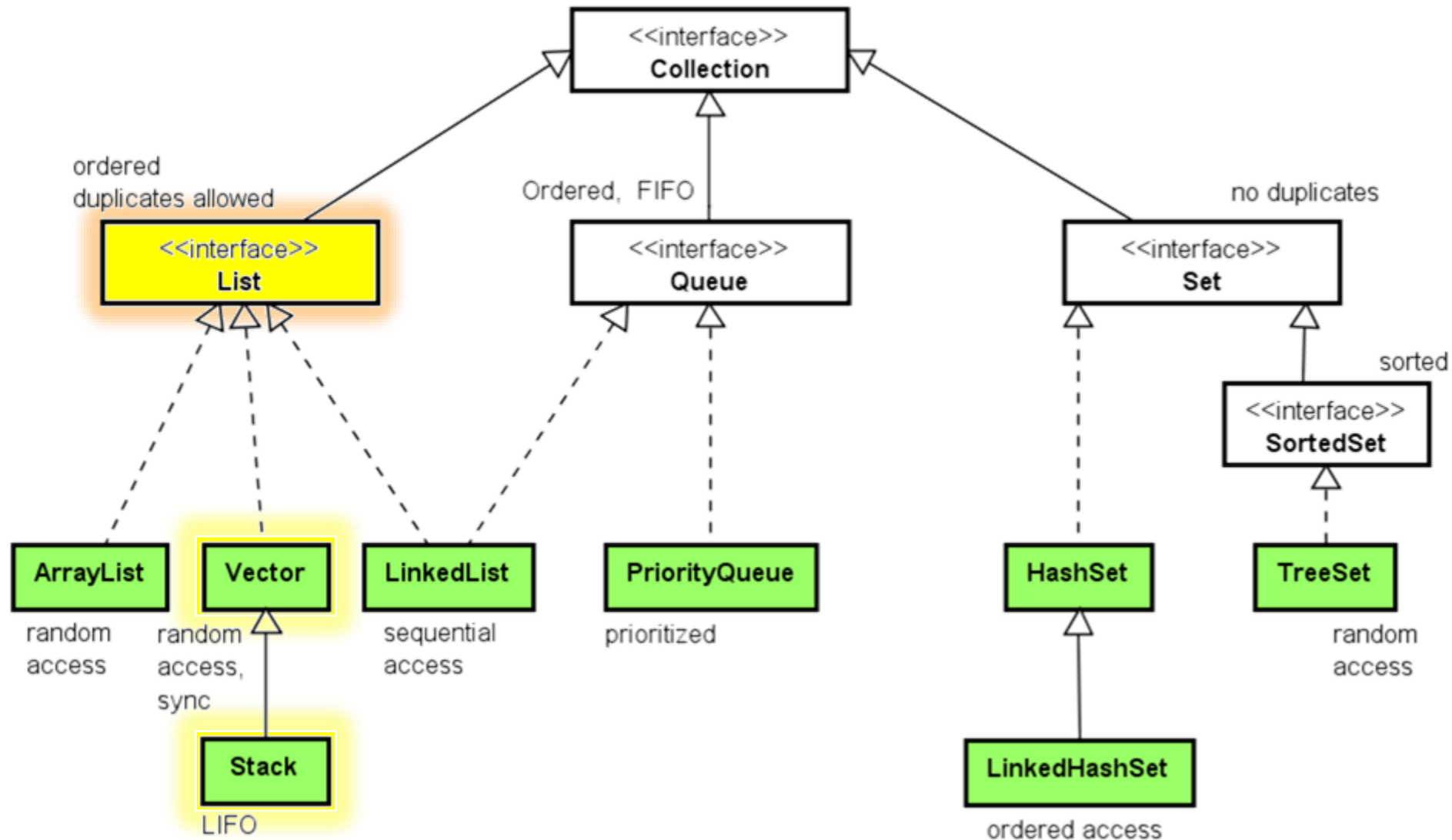
- `Collection` Interface
- `Lists`: `ArrayList` / `LinkedList`

## ***Today:***

- `More Lists`: `Vector` / `Stack`
- `Queues`: `Queue` / `PriorityQueue`
- `Interfaces`: `Comparable` / `Comparator`

# The Vector and Stack Classes

# Collections Class Hierarchy (simplified)



## *Aside:* The Vector Class

Since Java 2, `Vector` has become the same as `ArrayList`, except that `Vector` contains **synchronized methods** for accessing and modifying the vector.

- Synchronized methods are useful to control **concurrent access** of data.
  - Concurrency is outside the scope of this course.
- You can also create other synchronized collections (e.g., array lists), but this is also outside the scope of this course.

# Aside: The Vector Class, cont.

*java.util.AbstractList<E>*



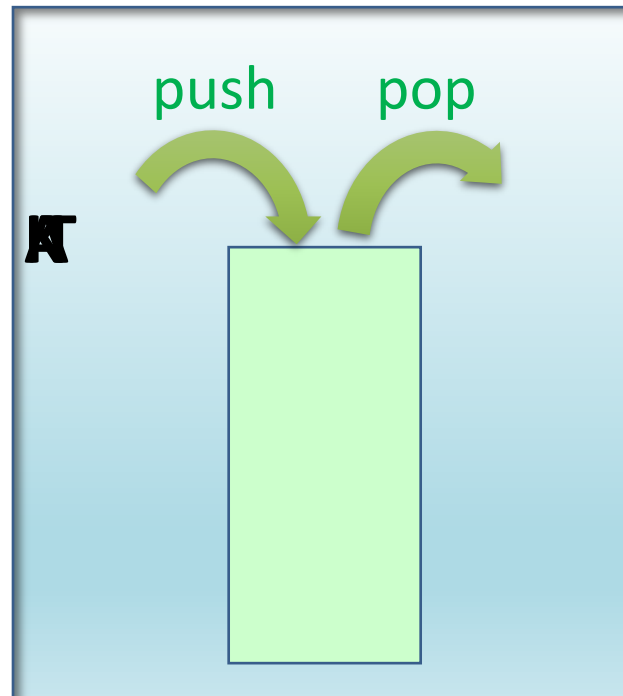
**java.util.Vector<E>**

```
+Vector()  
+Vector(c: Collection<? extends E>)  
+Vector(initialCapacity: int)  
+Vector(initCapacity: int, capacityIncr: int)  
+addElement(o: E): void  
+capacity(): int  
+copyInto(anArray: Object[]): void  
+elementAt(index: int): E  
+elements(): Enumeration<E>  
+ensureCapacity(): void  
+firstElement(): E  
+insertElementAt(o: E, index: int): void  
+lastElement(): E  
+removeAllElements(): void  
+removeElement(o: Object): boolean  
+removeElementAt(index: int): void  
+setElementAt(o: E, index: int): void  
+setSize(newSize: int): void  
+trimToSize(): void
```

Creates a default empty vector with initial capacity 10.  
Creates a vector from an existing collection.  
Creates a vector with the specified initial capacity.  
Creates a vector with the specified initial capacity and increment.  
Appends the element to the end of this vector.  
Returns the current capacity of this vector.  
Copies the elements in this vector to the array.  
Returns the object at the specified index.  
Returns an enumeration of this vector.  
Increases the capacity of this vector.  
Returns the first element in this vector.  
Inserts *o* into this vector at the specified index.  
Returns the last element in this vector.  
Removes all the elements in this vector.  
Removes the first matching element in this vector.  
Removes the element at the specified index.  
Sets a new element at the specified index.  
Sets a new size in this vector.  
Trims the capacity of this vector to its size.

# The Stack

A Stack represents a LIFO (last-in-first-out) data structure. The elements are accessed only from the top of the stack. That is, you can only retrieve, insert, or remove an element from the top of the stack.



# Some Stack Applications

Programming languages and compilers:

- method activation frames are placed onto a stack
  - *call=push, return=pop. See COSC 111 notes (Methods)*
- compilers use stacks to evaluate expressions
  - See case study in the textbook

Matching up related pairs of things:

- Getting the reverse of an array or a string
- Checking if string is palindrome
- Checking if braces { } match

Sophisticated algorithms:

- searching through a maze with "backtracking"
- "undo stack" of previous operations in a program

# The Stack Class

java.util.Vector<E>



java.util.Stack<E>

+Stack()

Creates an empty stack.

+empty(): boolean

Returns true if this stack is empty.

+peek(): E

Returns the top element in this stack.

+pop(): E

Returns and removes the top element in this stack.

+push(o: E) : E

Adds a new element to the top of this stack.

+search(o: Object) : int

Returns the position of the specified element in this stack.

# Practice

Check if a string is palindrome using a stack.

## Idea:

- Push letters of a string s1 from left to right
- Pop letters to form a new reversed string s2
- Check if s1 is equal to s2

```
public static void main(String[] args) {  
    System.out.println(isPalindrome("xyx")); //true  
    System.out.println(isPalindrome("xyz")); //false  
}  
  
public static boolean isPalindrome(String s1) {  
    Stack<Character> stack = new Stack<>();  
    String s2 = "";  
    for (int i = 0; i < s1.length(); i++)  
        stack.push(s1.charAt(i));  
    while(!stack.isEmpty())  
        s2 += stack.pop();  
    return s1.equals(s2);  
}
```

# Practice

Check if a string is palindrome using a stack.

## Idea:

- Push letters of a string s1 from left to right
- Pop letters to form a new reversed string s2
- Check if s1 is equal to s2

```
public static void main(String[] args) {
    System.out.println(isPalindrome("xyx")); //true
    System.out.println(isPalindrome("xyz")); //false
}

public static boolean isPalindrome(String s1) {
    Stack<Character> stack = new Stack<>();
    StringBuilder s2 = new StringBuilder(); //more efficient than String
    for (int i = 0; i < s1.length(); i++)
        stack.push(s1.charAt(i));
    while(!stack.isEmpty())
        s2.append(stack.pop());
    return s1.equals(s2.toString());
}
```

# Printing the Stack Elements

Let's say you want to *pop and print* all elements of a stack. What is wrong with the highlighted loop? How can you fix this?

```
public class Test {  
    public static void main(String[] args) {  
        Stack<Character> stack = new Stack<>();  
        for (int i = 0; i < "abcde".length(); i++)  
            stack.push("abcde".charAt(i));  
  
        for(int i = 0; i < stack.size(); i++)  
            System.out.println(stack.pop());  
    }  
}
```

**Answer:** The size in above loop will change in every iteration. This will result in printing **e d c** (but **not b and a**). Instead, you should use code similar to the following:

```
while(!stack.isEmpty()) //or while(stack.size() > 0)  
    System.out.print(stack.pop());
```

# Printing the Stack Elements

Let's say you want to ***ONLY print*** elements of a stack. What is wrong with the highlighted loop? How can you fix this?

```
while(!stack.isEmpty()) //or while(stack.size() > 0)
    System.out.print(stack.pop());
```

Answer: this code also destroys the stack. If you want to only print all elements you need another technique that keeps the original stack intact and yet print the elements.

- One way is to have a `get(i)` method – remember that a stack implements List interface, so it has index-based operations.
- The other way is to clone the stack first then print the cloned stack using the code above.

# Practice

The problem is that the stack will be destroyed by the method (the stack will be empty as all elements are popped during the search for the minimum).

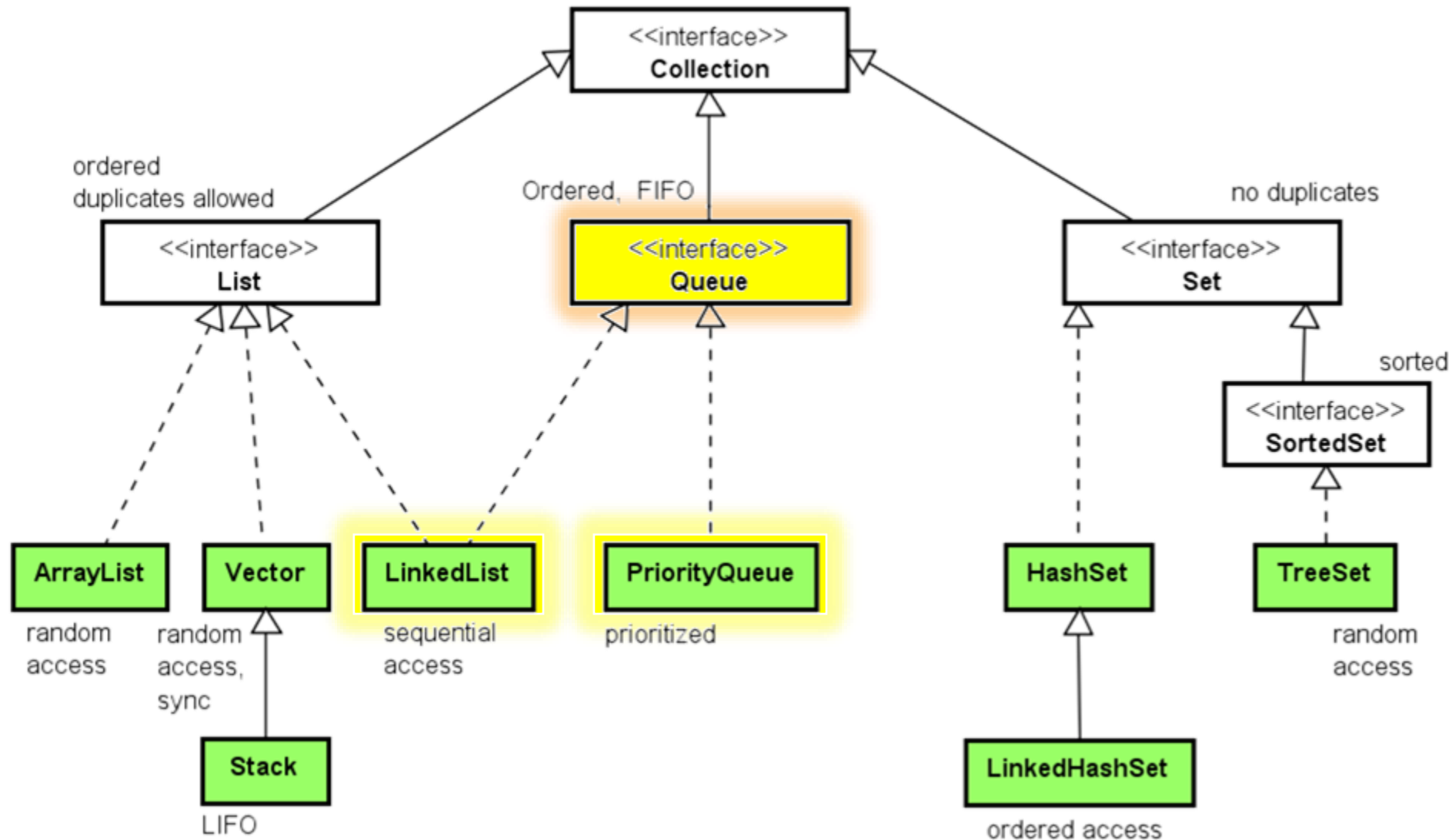
To avoid this, create a clone of your stack and work on it.

- Or use the `get(i)` method!

```
public static int stackMin2(Stack<Integer> s) {  
    Stack<Integer> s2 = (Stack<Integer>) s.clone();  
    int min = s2.pop();  
    while (!s2.isEmpty()) {  
        int element = s2.pop();  
        if (element < min)  
            min = element;  
    }  
    return min;  
}
```

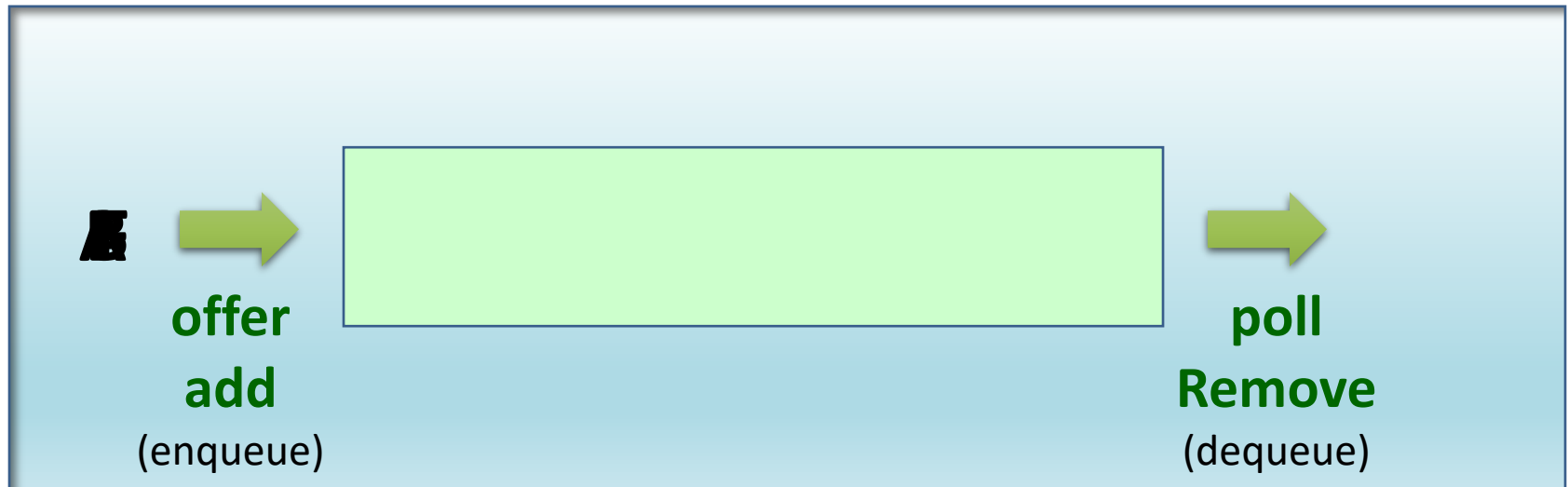
# Queues and Priority Queues

# Collections Class Hierarchy (not accurate)



# Queues

A queue is a FIFO (first-in-first-out) data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue.



# Some Queue Applications

Operating systems (assuming FIFO prioritization) :

- queue of processes to be run
- queue of print jobs to send to the printer

Programming:

- storing a queue of statements to be evaluated in order
- modeling a line of customer objects to be served in order

Real world examples:

- people waiting in a line
- cars at a gas station
- etc.

# The Queue Interface

«interface»  
*java.util.Collection<E>*



«interface»  
*java.util.Queue<E>*

*+offer(element: E): boolean*

*+poll(): E*

*+remove(): E*

*+peek(): E*

*+element(): E*

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.

# Implementation of Queue

**LinkedList** and **PriorityQueue** are two concrete implementation of **Queue**.

Example on using a **LinkedList** to implement a queue:

```
Queue<String> queue = new LinkedList<String>();
queue.offer("First");
queue.offer("Second");
System.out.print(queue.remove() + " ");
queue.offer("Third");
queue.offer("Fourth");
while (queue.size() > 0)
    System.out.print(queue.remove() + " ");
```

Output:

First Second Third Fourth

# Practice

Given the Customer class below, write a program for a restaurant that will keep a waiting list of customers to based on first-come-first-served.

```
public class Customer {  
    private int order;  
    private String name;  
  
    public Customer(int order, String name) {  
        setOrder(order);  
        setName(name);  
    }  
  
    public int getOrder() {return order;}  
    public void setOrder(int order) {this.order = order;}  
    public String getName() {return name;}  
    public void setName(String name) {this.name = name;}  
  
    public String toString(){return "order=" + order + ",name=" + name;}  
}
```

# Practice, cont.

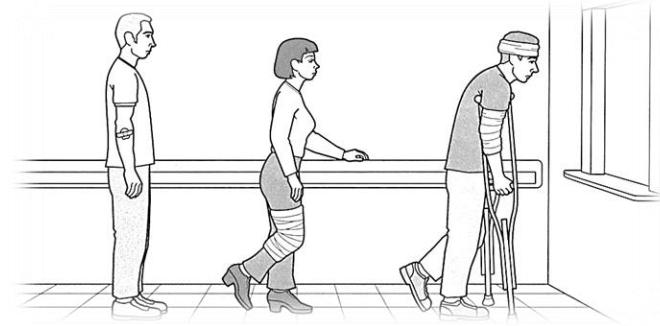
```
public class CustomerTest {  
    public static void main(String[] args) {  
        Queue<Customer> queue = new LinkedList<>();  
  
        queue.offer(new Customer(1, "Bob"));  
        queue.offer(new Customer(2, "Lily"));  
        queue.offer(new Customer(3, "Mark"));  
  
        while(!queue.isEmpty())  
            System.out.println(queue.poll().getName() + " was served.");  
    }  
}
```

## **Output**

Bob was served.  
Lily was served.  
Mark was served.

# The PriorityQueue Class

In a priority queue, elements are ordered based on some criteria.



In Java, elements are ordered based on \*:

- their **natural ordering** (imposed by **Comparable**), or
- a **Comparator** provide at queue construction time

When accessing elements, elements with the higher priority are removed first.

## Constructors:

- |  |  |
|--|--|
| <b>PrioriryQueue()</b>                     | <b>natural ordering</b> , default initial capacity   |
| <b>PrioriryQueue(int cap)</b>              | <b>natural ordering</b> , initial capacity = cap     |
| <b>PrioriryQueue(aCollection)</b>          | <b>natural ordering</b> , initialized to aCollection |
| <b>PrioriryQueue(int cap, aComparator)</b> | <b>order use aComparator</b> , capacity = cap        |

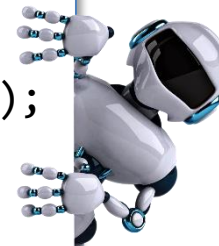
\* *Details about the `Comparable` and `Comparator` interfaces can be found at the end of this set of lecture notes.*

# Example: PriorityQueue with Comparable

```
public class Robot implements Comparable<Robot> {
    private int weight, year;
    public Robot(int weight, int year) {setWeight(weight);setYear(year);}
    public int compareTo(Robot other) {return this.weight - other.weight;}
    public int getYear() {return year;}
    public void setYear(int year) {this.year = year;}
    public int getWeight() {return weight;}
    public void setWeight(int weight) {this.weight = weight;}

    public String toString(){return "Robot [weight:"+weight+",Year:"+year+"]";}
}
```

```
public class RobotTest {
    public static void main(String[] args) {
        Queue<Robot> queue = new PriorityQueue<>();
        queue.offer(new Robot(30, 2000));
        queue.offer(new Robot(20, 2010));
        queue.offer(new Robot(50, 2005));
        while(!queue.isEmpty())
            System.out.println(queue.poll());
    }
}
```



## Output

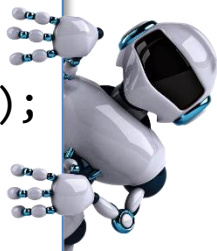
```
Robot [weight:20, Year:2010]
Robot [weight:30, Year:2000]
Robot [weight:50, Year:2005]
```

# Example: PriorityQueue with Comparator

What if we want to **add** another criteria for comparing the Robots, e.g. based on year, and use it for the priority queue?

```
import java.util.Comparator;
public class RobotComparator implements Comparator<Robot>{
    public int compare(Robot r1, Robot r2) {
        return r1.getYear() - r2.getYear();
    }
}
```

```
public class RobotTest {
    public static void main(String[] args) {
        Queue<Robot> queue = new PriorityQueue<>(new RobotComparator());
        queue.offer(new Robot(30, 2000));
        queue.offer(new Robot(20, 2010));
        queue.offer(new Robot(50, 2005));
        while(!queue.isEmpty())
            System.out.println(queue.poll());
    }
}
```



## Output

```
Robot [weight:30, Year:2000]
Robot [weight:50, Year:2005]
Robot [weight:20, Year:2010]
```

# Practice

Given the Patient class below, write a program for a hospital that will keep a waiting list of patients. Patients are usually sorted based on first-come-first-served. However, emergency cases should be treated first. Whenever you have two emergency cases, the case arrived first is treated first.

```
public class Patient {  
    //attributes  
    private int order;  
    private String name;  
    private boolean emergencyCase;  
  
    //constructor  
    public Patient(int order, String name, boolean emergencyCase) {  
  
    //getters and setters  
    public int getOrder() {  
    public void setOrder(int order) {  
    public String getName() {  
    public void setName(String name) {  
    public boolean isEmergencyCase() {  
    public void setEmergencyCase(boolean emergencyCase) {  
  
    public String toString() {  
}
```

# Practice, cont.

```
public class PatientComparator implements Comparator<Patient>{
    public int compare(Patient p1, Patient p2) {
        if(p1.isEmergencyCase() && !p2.isEmergencyCase())
            return -1; //place p1 first
        else if(!p1.isEmergencyCase() && p2.isEmergencyCase())
            return 1; //place p2 first
        else //if both are emergency or both are not emergency
            return p1.getOrder()-p2.getOrder(); //place smaller order first
    }
}
```

```
public class PatientTest {
    public static void main(String[] args) {
        PriorityQueue<Patient> waitingList =
            new PriorityQueue<>(5, new PatientComparator());

        waitingList.offer(new Patient(1, "p1", false));
        waitingList.offer(new Patient(2, "p2", false));
        waitingList.offer(new Patient(3, "p3", true));
        waitingList.offer(new Patient(4, "p4", false));
        waitingList.offer(new Patient(5, "p5", true));

        while(waitingList.size()>0)
            System.out.println(waitingList.poll());
    }
}
```

## Output

p3

p5

p1

p2

p4

# Caution!

In priority queues, an iterator or (a for-each loop) is **NOT** guaranteed to traverse the elements in any particular order. This is because it's implemented as a priority heap rather than sorted list.

The only guarantee provided by PriorityQueue is that poll(), peek() return the **least** element.

How to show all elements according to their priority without actually remove them from the priority queue?

Solutions include:

- 1) create a temporary priority queue and copy all elements from the original one to the temp one, then use poll() in a loop.
- 2) convert the priority queue to an array and use Arrays.sort

# Comparable and Comparator Interfaces

# The Comparable Interface

Any class that implements the `Comparable` interface must have a method `compareTo`.

```
public interface Comparable<T> {  
    public int compareTo(T obj);  
}
```

## ■ Example:

```
public class Robot implements Comparable<Robot> {  
    public int weight;  
    public Robot(int weight) { this.weight = weight; }  
    public String toString() { return ""+weight; }  
  
    public int compareTo(Robot r) {  
        return this.weight - r.weight;  
    }  
}
```

## ■ Sorting:

```
List<Robot> robots = new ArrayList<>();  
//add elements  
Collections.sort(robots);
```

# The Comparator Interface

The **Comparator** interface can be used to compare objects of a class, similar to the **Comparable** interface.

## How to use?

- To use **Comparator** with a class of the type **E**, you must create **ANOTHER class** that implements **Comparator** and has a method  

```
int compare(E e1, E e2)
```
- This method returns -1 if **e1** is less than **e2**, +1 if **e1** is larger than **e2**, and 0 if **e1** is equal to **e2**.

## When/why to use **Comparator**?

- If your class doesn't implement **Comparable**
  - e.g., you don't want to write a **compareTo** method in your class
  - Note that **Comparable** provides what is known as “**natural ordering**”.
- If you want to have **more than one way** of comparing your objects of a class type (in addition to the natural order).
  - e.g., based on weight only, height only, age and height only, etc.

# Practice

The following code sorts an array list with objects of the type `Robot` using `Collection.sort` method. You are required to use `Comparator` instead of `Comparable`.

```
public class Robot implements Comparable<Robot> {
    public int weight;
    public Robot(int weight) { this.weight = weight; }
    public String toString() { return ""+weight; }

    public int compareTo(Robot r) {
        return this.weight - r.weight;
    }
}
```

```
import java.util.*;
public class RobotSorting1 {
    public static void main(String[] args) {
        List<Robot> robots = new ArrayList<>();
        robots.add(new Robot(30));
        robots.add(new Robot(60));
        robots.add(new Robot(20));

        Collections.sort(robots);
        System.out.println(robots);
    }
}
```

Output

[20, 30, 60]

# Practice, cont.

```
public class Robot {  
    public int weight;  
    public Robot(int weight) { this.weight = weight; }  
    public String toString() {return "" + weight;}  
}
```

```
import java.util.Comparator;  
public class RobotComparator implements Comparator<Robot>{  
    public int compare(Robot o1, Robot o2) {  
        return o1.weight - o2.weight;  
    }  
}
```

```
import java.util.*;  
public class RobotSorting {  
    public static void main(String[] args) {  
        List<Robot> robots = new ArrayList<>();  
        robots.add(new Robot(30));  
        robots.add(new Robot(60));  
        robots.add(new Robot(20));  
  
        Collections.sort(robots, new RobotComparator());  
        System.out.println(robots);  
    }  
}
```

Output

[20, 30, 60]