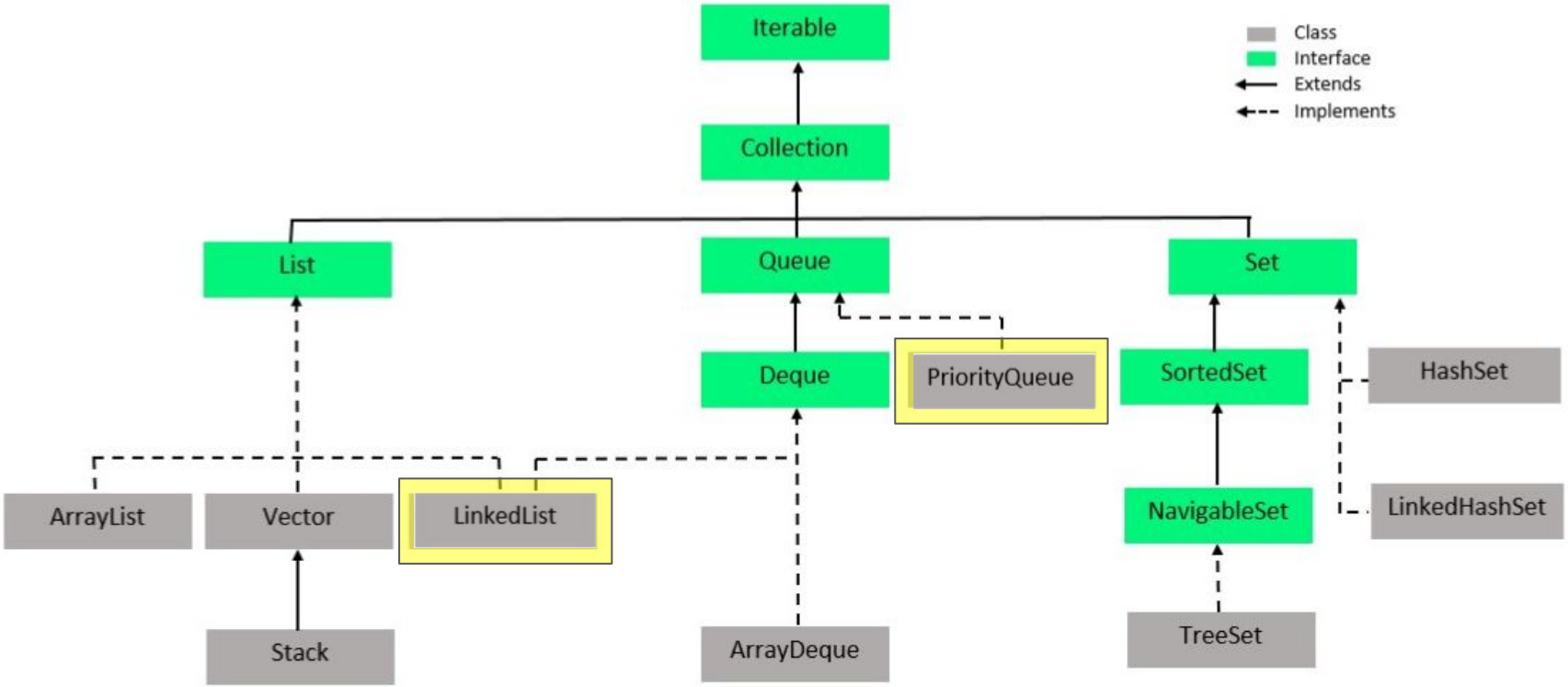
COSC 121: Computer Programming II



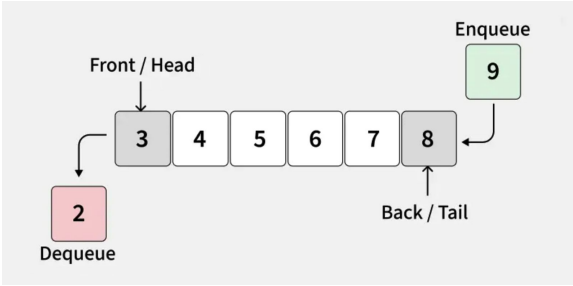
Today's Key Concepts

- Continue with the Java Collection Framework
  - Queue interface

    FIFO: First-in-first-out order
  - Concrete Classes: LinkedList, PriorityQueue
- Defining specialized queue ordering
- Comparing objects for ordering:
  - Comparable interface
  - Comparator interface

The Queue Data Structure

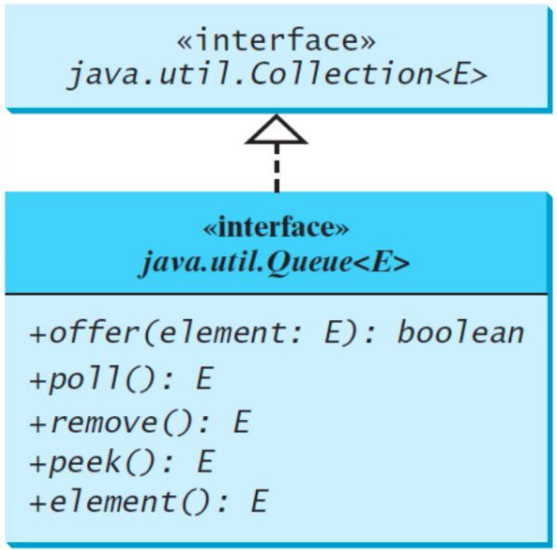
- A queue implements a FIFO (first-in-first-out) ordered data structure
  - Elements are added to the end and removed from the front

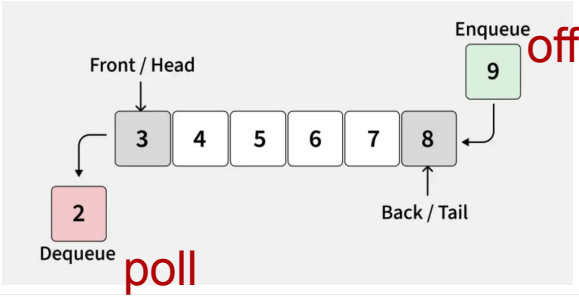


What real world applications naturally make use of a queue?

The Queue Data Structure

- A queue implements a FIFO (first-in-first-out) ordered data structure
  - Elements are added to the end and removed from the front





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.	

What real world applications naturally make use of a queue?

Implementing a Queue

- Two concrete classes of Queue: LinkedList and PriorityQueue
- Example of implementing a Queue using a LinkedList:

```
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() + " ");
```

why does this work?

- Output:
  - First Second Third Fourth

Restaurant Example

- Write the Restaurant class that keeps a waiting list of customers and serves them in first-come-first-served order
- Use the following Customer class:

```
public class Customer {
    //attributes
    private int order;
    private String name;

    //constructor
    public Customer(int order, String name) {

    //getters and setters
    public int getOrder() {
    public String getName() {
    public void setOrder(int order) {
    public void setName(String name) {
}
```

Restaurant class:
- make a waiting list
- poll from the list

iClicker Question

What is the output of the following code if you have a queue where the elements are added in this order: 60, 10, 50, 30, 40, 20?

```
while (!queue.isEmpty())
    System.out.print(queue.poll() + " ");
```

A. 10 20 30 40 50 60
B. 60 10 50 30 40 20
C. 60 50 40 30 20 10
D. 20 40 30 50 10 60
E. There is no guaranteed order

The PriorityQueue Class

- A PriorityQueue is a class that implements the Queue interface, but it orders elements by priority instead of insertion order
  - Elements are removed based on priority
  - By default, it uses natural ordering (imposed by Comparable)

```
Insert:  30 → 10 → 20
Remove:  10 → 20 → 30
```

The PriorityQueue Class

- A PriorityQueue is a class that implements the Queue interface, but it orders elements by priority instead of insertion order
  - Elements are removed based on priority
  - By default, it uses natural ordering (imposed by Comparable)

```
Insert:  30 → 10 → 20
Remove:  10 → 20 → 30
```

```java
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.offer(30);
pq.offer(10);
pq.offer(20);
System.out.println(pq.poll()); // 10
System.out.println(pq.poll()); // 20
```

The PriorityQueue Class

- A PriorityQueue is a class that implements the Queue interface, but it orders elements by priority instead of insertion order
  - Elements are removed based on priority
  - By default, it uses natural ordering (imposed by Comparable)

```
Insert:  30 → 10 → 20
Remove:  10 → 20 → 30
```

```java
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.offer(30);
pq.offer(10);
pq.offer(20);
System.out.println(pq.poll()); // 10
System.out.println(pq.poll()); // 20
```

  - Or, use a Comparator provided at queue construction time

Priority Queue Constructors

- Priority queues (as well as most other Queues in Java) grow dynamically
  - You can optionally specify an initial capacity

Constructors:

PrioirtyQueue()	natural ordering	default initial capacity
PrioirtyQueue(int cap)	natural ordering	initial capacity = cap
PrioirtyQueue(aCollection)	natural ordering	initialized to aCollection
PrioirtyQueue(aComparator)	order use aComparator	default initial capacity
PrioirtyQueue(int cap, aComparator)	order use aComparator	initial capacity = cap

Recall: 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);
```

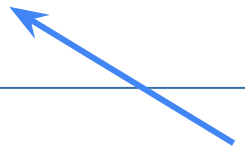
Robot Example with Comparable

- Recall the Comparable interface requires implementation of compareTo()

```
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;
    }
}
```

- How to add Robots to a priority queue?



lighter robots
should have priority

Robot Example with Comparable

- Given Robot class with implementation of compareTo()
- Adding Robots to a priority queue:

```
import java.util.*;
public class TestPriorityQueue2 {
    public static void main(String[] args) {

        // initialize
        // add Robots
        // poll queue

    }
}
```

Robot Example with Comparable

- Given Robot class with implementation of compareTo()
- Adding Robots to a priority queue:

```
import java.util.*;
public class TestPriorityQueue2 {
    public static void main(String[] args) {
        Queue<Robot> queue = new PriorityQueue<>();

        // add Robots
        // poll queue

    }
}
```

Robot Example with Comparable

- Given Robot class with implementation of compareTo()
- Adding Robots to a priority queue:

```
import java.util.*;
public class TestPriorityQueue2 {
    public static void main(String[] args) {
        Queue<Robot> queue = new PriorityQueue<>();
        queue.offer(new Robot(30));
        queue.offer(new Robot(60));
        queue.offer(new Robot(20));
        // poll queue

    }
}
```

Robot Example with Comparable

- Given Robot class with implementation of compareTo()
- Adding Robots to a priority queue:

```
import java.util.*;
public class TestPriorityQueue2 {
    public static void main(String[] args) {
        Queue<Robot> queue = new PriorityQueue<>();
        queue.offer(new Robot(30));
        queue.offer(new Robot(60));
        queue.offer(new Robot(20));
        while (queue.size() > 0)
            System.out.print(queue.poll() + " ");
    }
}
```

Output?

Robot Example with Comparable

- Given Robot class with implementation of compareTo()
- Adding Robots to a priority queue:

```
import java.util.*;
public class TestPriorityQueue2 {
    public static void main(String[] args) {
        Queue<Robot> queue = new PriorityQueue<>();
        queue.offer(new Robot(30));
        queue.offer(new Robot(60));
        queue.offer(new Robot(20));
        while (queue.size() > 0)
            System.out.print(queue.poll() + " ");
    }
}
```

Output:

```
[20, 30, 60]
```

New: The Comparator Interface

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

New: The Comparator Interface

- The Comparator interface can be used to compare objects of a class, similar to the Comparable interface
- How to implement it?
  - 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 e1is less than e2,
    +1 if e1is larger than e2,
    0 if e1is equal to e2

New: The Comparator Interface

- The Comparator interface can be used to compare objects of a class, similar to the Comparable interface
- How to implement it?
  - 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 e1is less than e2,
    +1 if e1is larger than e2,
    0 if e1is equal to e2
- When/why to use Comparator instead of Comparable?
  - Comparable: for defining a single, natural sort order within a class
  - Comparator: for defining multiple, custom sort orders externally

Robot Example with a Comparator Object

- Changes to the Robot class

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

- Creating a Comparator
- Changes to queue creation

Robot Example with a Comparator Object

- Changes to the Robot class
- Creating a Comparator

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

- Changes to queue creation

Robot Example with a Comparator Object

- Changes to the Robot class
- Creating a Comparator
- Changes to queue creation

```
import java.util.*;
public class TestPriorityQueue2 {
    public static void main(String[] args) {
        Queue<Robot> queue = new PriorityQueue<>(3, new RobotComparator());
        queue.offer(new Robot(30));
        queue.offer(new Robot(60));
        queue.offer(new Robot(20));
        while (queue.size() > 0)
            System.out.print(queue.poll() + " ");
    }
}
```

Output

```
[20, 30, 60]
```

iClicker Question

What is the output of the following code:

```
PriorityQueue<Integer> queue= new PriorityQueue<Integer>(
                        Arrays.asList(60,10,50,30,40,20) );

while (!queue.isEmpty())
     System.out.print(queue.poll() + " ");
```

A. 10 20 30 40 50 60
B. 60 10 50 30 40 20
C. 60 50 40 30 20 10
D. 20 40 30 50 10 60
E. There is no guaranteed order

Hospital Example

- Given the following Patient class (next slide), 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.

Regular queue or priority queue?

Default or specialized comparator?

Patient Class

```
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() {
}
```

Comparator class:
- define compare(Patient, Patient)

Hospital class:
- make a waiting list
- add a mix of patients
- poll from the list

compare using getOrder() unless isEmergencyCase() is true

Caution on Iterators

- In priority queues, an iterator or a for-each loop is NOT guaranteed to traverse the elements in any particular order
  - Due to implementation detail (a priority heap, not a sorted list)

- The only guarantee provided by PriorityQueue is that poll() and peek() return the **smallest** element

Stepping Back …

- Which data structure would you use to tackle the following problem and why?
- Problem:

  You are building a tic-tac-toe AI by considering all possible moves the AI can take, then checking all possible moves an opponent can take, and repeating that pattern, until it finds a board where the AI wins. If the board ends up in the AI losing, then it changes a previous move and try another option, and repeat.

- Options:
  - Recursion
  - ArrayList
  - LinkedList
  - Queue
  - PriorityQueue

Stepping Back …

- Which data structure would you use to tackle the following problem and why?
- Problem:

  You are writing the software for managing jobs submitted to a school printer. Documents sent to the printer should print in the same order they were submitted.

- Options:
  - Recursion
  - ArrayList
  - LinkedList
  - Queue
  - PriorityQueue

Stepping Back …

- Which data structure would you use to tackle the following problem and why?
- Problem:

  You are writing the software for making a music playlist. When making a playlist, users are likely to add songs, remove songs, insert songs – anywhere in the playlist, until they are happy with the sequence.

- Options:
  - Recursion
  - ArrayList
  - LinkedList
  - Queue
  - PriorityQueue

Stepping Back …

- Which data structure would you use to tackle the following problem and why?
- Problem:

  Suppose you are building software to benchmark experiments. Specifically, multiple objects will emit one observation at midnight each day for 200 days. After that, the software needs to computer averages, max, min, standard deviations per object.

- Options:
  - Recursion
  - ArrayList
  - LinkedList
  - Queue
  - PriorityQueue

Stepping Back …

- Which data structure would you use to tackle the following problem and why?
- Problem:

  You are writing the software for handling customer phone complaints. Customers call a help center and must be served in the order they arrived.

- Options:
  - Recursion
  - ArrayList
  - LinkedList
  - Queue
  - PriorityQueue

Stepping Back …

- Which data structure would you use to tackle the following problem and why?
- Problem:

  You are an automatic maze solver. The solver can take these actions: go straight, turn right, or turn left. It repeatedly selects a new action until it reaches the end or is stuck. If stuck, it can backtrack and try a new action.

- Options:
  - Recursion
  - ArrayList
  - LinkedList
  - Queue
  - PriorityQueue

Stepping Back …

- Which data structure would you use to tackle the following problem and why?
- Problem:

  As part of a web browser, you are writing the code to manage browser tabs, where users frequently insert a tab between others, close tabs, and move to the next or previous tab.

- Options:
  - Recursion
  - ArrayList
  - LinkedList
  - Queue
  - PriorityQueue

Stepping Back …

- Which data structure would you use to tackle the following problem and why?
- Problem:

  You are writing the software for an elevator. The software needs to take passengers' button presses (for each floor) and decide the order of floors to stop at.

- Options:
  - Recursion
  - ArrayList
  - LinkedList
  - Queue
  - PriorityQueue

Stepping Back …

- Which data structure would you use to tackle the following problem and why?
- Problem:

  You are designing gradebook software for teachers. Teachers need to input grades for Assignments and Exams. At the end of the course, they need to calculate totals, averages, and weigh the scores based on course criteria.

- Options:
  - Recursion
  - ArrayList
  - LinkedList
  - Queue
  - PriorityQueue

Stepping Back …

- Which data structure would you use to tackle the following problem and why?
- Problem:

  You are programming a web crawler that follows page links given a starting URL. The program follows every link to a new page and follow those links etc. until it reaches a page without links.

- Options:
  - Recursion
  - ArrayList
  - LinkedList
  - Queue
  - PriorityQueue

Quick Review

Data Structure	Typical Scenario
ArrayList	Random access collections
LinkedList	Frequent insert/remove
Queue	First-come-first-served processing
PriorityQueue	Process highest priority first
Recursion	Tree / nested / divide problems