QUEUE FAMILY

**Basic Overview:**

* **Queue** is part of the **java.util** package and extends the **Collection** interface.
* It represents a collection of elements designed for holding elements prior to processing.
* It follows the **FIFO (First-In-First-Out)** order, where elements added first are removed first.
* Commonly used in scenarios like scheduling tasks, managing requests, and handling buffers.

**Key Implementations:**

* **LinkedList**: Implements the Queue interface and also allows elements to be added or removed from both ends of the list. It is commonly used as a **queue** because it provides constant-time insertions and removals.
* **PriorityQueue**: Implements a priority queue where elements are ordered based on their priority rather than their order of insertion. It doesn't guarantee FIFO ordering but rather an order defined by their natural ordering or a **Comparator**.
* **ArrayDeque**: A resizable array implementation of a deque (double-ended queue), which can be used as a Queue for FIFO operations. It's more efficient than LinkedList for adding and removing elements.
* **BlockingQueue**: Part of **java.util.concurrent** package, it allows threads to block when trying to enqueue or dequeue elements. It includes implementations like **ArrayBlockingQueue**, **LinkedBlockingQueue**, etc.

**Queue Methods:**

* add(E e): Adds an element to the queue. Throws **IllegalStateException** if the element cannot be added (for example, if the queue is bounded and full).
* offer(E e): Adds an element to the queue. Returns **false** if the element cannot be added, as opposed to throwing an exception.
* remove(): Removes and returns the front element of the queue. Throws **NoSuchElementException** if the queue is empty.
* poll(): Removes and returns the front element of the queue. Returns **null** if the queue is empty, unlike remove().
* peek(): Returns the front element of the queue without removing it. Returns **null** if the queue is empty.
* element(): Returns the front element of the queue without removing it. Throws **NoSuchElementException** if the queue is empty.
* size(): Returns the number of elements in the queue.
* isEmpty(): Checks whether the queue is empty.
* clear(): Removes all elements from the queue.

**Queue Characteristics:**

* **FIFO Behavior**: Ensures the first element added is the first one to be removed.
* **Blocking Operations** (in Concurrent Queue variants): Allows operations like blocking on **enqueue** or **dequeue** until the queue has space or contains elements.
* **Order**: Typically, the queue's order is based on the insertion order, but for priority queues, it depends on priority, not insertion order.
* **Efficiency**: Varies by implementation. **LinkedList** allows for O(1) insertions and deletions but can be memory-intensive, whereas **ArrayDeque** is typically more efficient in terms of memory for queue operations.

**Use Cases:**

* **Scheduling**: Queues are used for task scheduling, such as in **ThreadPools** or job scheduling systems.
* **Buffering**: Used in systems where data is buffered, such as in **Producer-Consumer** models.
* **Request Handling**: In network systems, queues are used to handle incoming requests in an orderly fashion.
* **Breadth-First Search (BFS)**: Queues are commonly used in algorithms like BFS to explore nodes level by level.

**Advantages:**

* **FIFO Ordering**: Ensures elements are processed in the order they are added.
* **Concurrency**: Thread-safe versions like **BlockingQueue** allow safe concurrent operations.
* **Flexible Implementations**: Can be backed by different data structures (array, linked list, or priority heap), offering flexibility based on specific use cases.

**Disadvantages:**

* **Limited Ordering**: Some queue implementations (e.g., **PriorityQueue**) do not guarantee FIFO ordering, which may not suit all use cases.
* **Blocking Behavior**: Some implementations introduce blocking, which may require additional handling in multithreaded environments.

**Regular Queue (FIFO)**

* **Implementation:** LinkedList, ArrayDeque
* **Use Case:** Basic queuing scenarios, where elements need to be processed in the order they arrive.
* **Example Use Case:** Task scheduling, printer queues, handling incoming network requests.

A computer screen shot of a black screen

Description automatically generated

* **Explanation:** The regular FIFO queue ensures that elements are processed in the exact order they are inserted. This is the most common queue type used in scenarios like task scheduling or print job management.

**Priority Queue**

* **Implementation**: PriorityQueue<E>
* **Use Case**: When you need to prioritize some elements over others, based on a certain priority rather than the order they were inserted. Common in algorithms that require ordering elements by priority, such as **Dijkstra's shortest path algorithm**.
* **Example Use Case**: Job scheduling in systems where jobs have different priorities.

A computer screen shot of a black screen

Description automatically generated

* **Explanation**: The **PriorityQueue** doesn’t follow FIFO but instead processes elements based on their **natural ordering** or a **Comparator** that you define. It’s ideal when you need to execute tasks in order of importance or urgency.

**Deque (Double-Ended Queue)**

* **Implementation**: ArrayDeque, LinkedList
* **Use Case**: When you need to insert or remove elements from both ends of the queue, not just the front.
* **Example Use Case**: Undo functionality in applications, where you need to remove actions from both ends.

A screen shot of a computer code

Description automatically generated

* **Explanation**: A **Deque** allows insertion and removal of elements from both ends, enabling more flexible queuing mechanisms like **LIFO (Last-In-First-Out)** or **FIFO** behavior.

**Blocking Queue**

* **Implementation**: ArrayBlockingQueue, LinkedBlockingQueue, PriorityBlockingQueue
* A screenshot of a computer program

  Description automatically generated**Use Case**: Used in multi-threaded environments where one thread puts elements into the queue, and another thread takes elements from it. The queue operations may block if the queue is full or empty, allowing efficient synchronization.
* **Example Use Case**: Producer-consumer scenarios, thread pools, task processing in multithreaded environments.
* **Explanation**: **BlockingQueue** is essential in multi-threaded systems where threads must wait until an element becomes available or space becomes free in the queue. It helps prevent resource starvation and race conditions.

**Synchronous Queue**

* **Implementation**: SynchronousQueue
* **Use Case**: A special type of **BlockingQueue** where each insert operation must wait for a corresponding remove operation by another thread. It’s commonly used in situations where tasks must be handed off between threads without buffering.
* **Example Use Case**: Handing off tasks between a producer and consumer in a thread pool.

A screen shot of a computer program

Description automatically generated

**Explanation**: **SynchronousQueue** does not hold any elements; instead, each insert operation has to wait for a corresponding remove operation. It’s used in scenarios where tasks are transferred between threads without the need for buffering.

**Delay Queue**

* **Implementation**: DelayQueue<E> (Part of java.util.concurrent)
* **Use Case**: Used when elements must be processed after a certain delay. Each element in the queue must implement the **Delayed** interface, which provides a delay time.
* **Example Use Case**: Implementing time-based task scheduling, where tasks are delayed before they are executed.

A screen shot of a computer program

Description automatically generated

**Explanation**: The **DelayQueue** allows for task scheduling that ensures tasks are executed only after a specified delay, useful in scenarios like **scheduled tasks** or **delayed job execution**.

A screenshot of a computer

Description automatically generated

Note:

1. **Best Performance**: **ArrayDeque** for general-purpose FIFO queues (time and space efficient).

2. **Best for Concurrency**: **ArrayBlockingQueue** and **LinkedBlockingQueue** for multi-threaded environments with thread synchronization.

3. **Best for Prioritization**: **PriorityQueue** for tasks requiring ordering by priority rather than FIFO.

A white sheet with black text

Description automatically generated