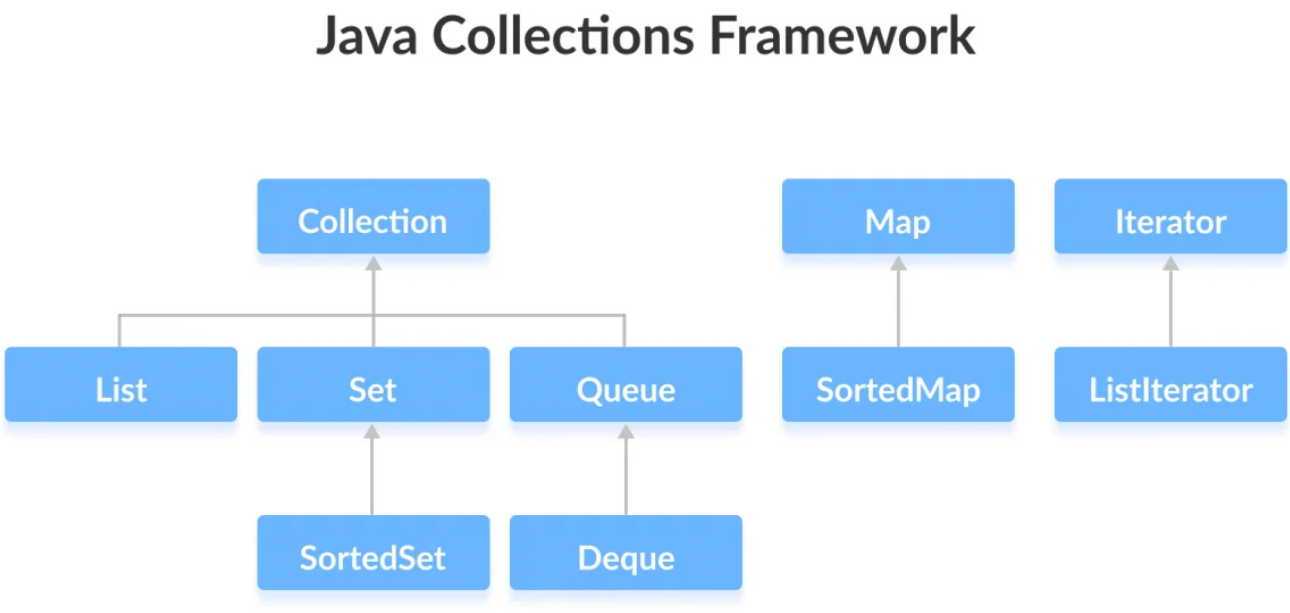
Collections

Java Collections Framework

The Java Collections Framework (JCF) is a set of classes and interfaces that implement commonly reusable collection data structures. The JCF is organized into interfaces and implementations of those interfaces. The interfaces define the functionality of the collection data structures, and the implementations provide concrete implementations of those interfaces.

The advantages of using the JCF are:

- **Consistent API** - The API has a basic set of interfaces like Collection, Set, List, or Map, all the classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have some common set of methods.
- **Reduces programming effort** - A programmer doesn't have to worry about the design of the Collection but rather he can focus on its best use in his program. Therefore, the basic concept of Object-oriented programming (i.e.) abstraction has been successfully implemented.
- **Increases program speed and quality** - Increases performance by providing high-performance implementations of useful data structures and algorithms because in this case, the programmer need not think of the best implementation of a specific data structure. He can simply use the best implementation to drastically boost the performance of his algorithm/program.



Collection Interface

The Collection interface is the root interface of the Java Collections Framework. It is the foundation upon which the collection framework is built. It declares the core methods that all collections will have. The Collection interface is a part of the java.util package.

The JDK does not provide any direct implementations of this interface: it provides implementations of more specific sub-interfaces like Set and List. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

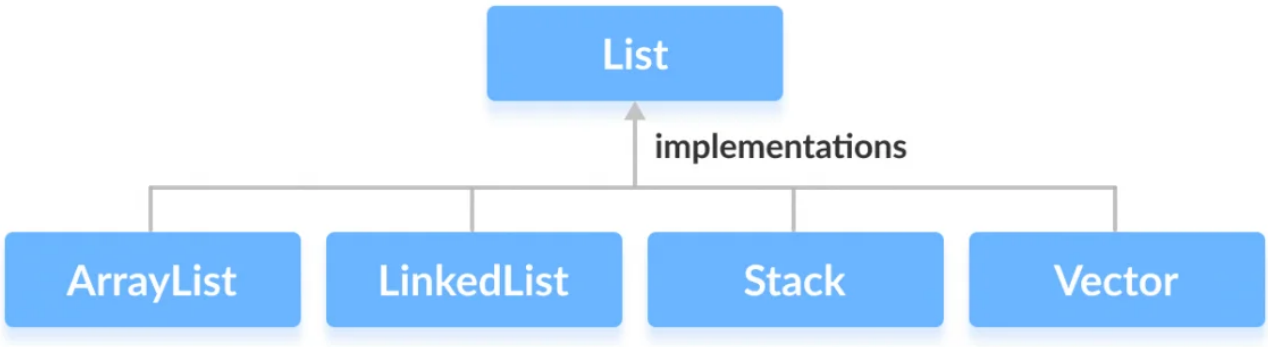
The Collection interface includes various methods that can be used to perform different operations on objects. These methods are available in all its subinterfaces.

- `add()` - inserts the specified element to the collection
- `size()` - returns the size of the collection
- `remove()` - removes the specified element from the collection
- `iterator()` - returns an iterator to access elements of the collection
- `addAll()` - adds all the elements of a specified collection to the collection
- `removeAll()` - removes all the elements of the specified collection from the collection
- `clear()` - removes all the elements of the collection

List Interface

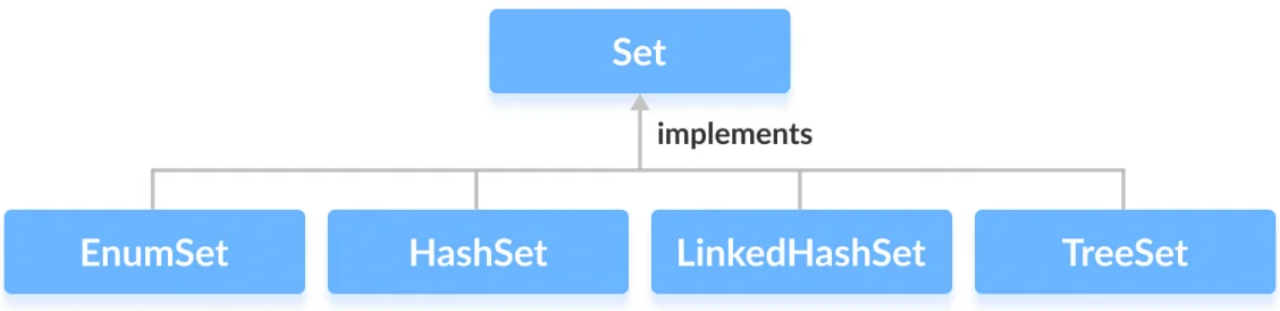
The List interface extends the Collection interface. It is an ordered collection of objects in which duplicate values can be stored. Since List preserves the insertion order, it allows positional access and insertion of elements. List Interface is implemented by:

- `ArrayList` - Resizable-array implementation of the List interface
- `LinkedList` - Doubly-linked list implementation of the List and Deque interfaces
- `Vector` - Synchronized resizable-array implementation of the List interface
- `Stack` - Subclass of Vector that implements a standard last-in, first-out stack



Set Interface

The Set interface extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set.

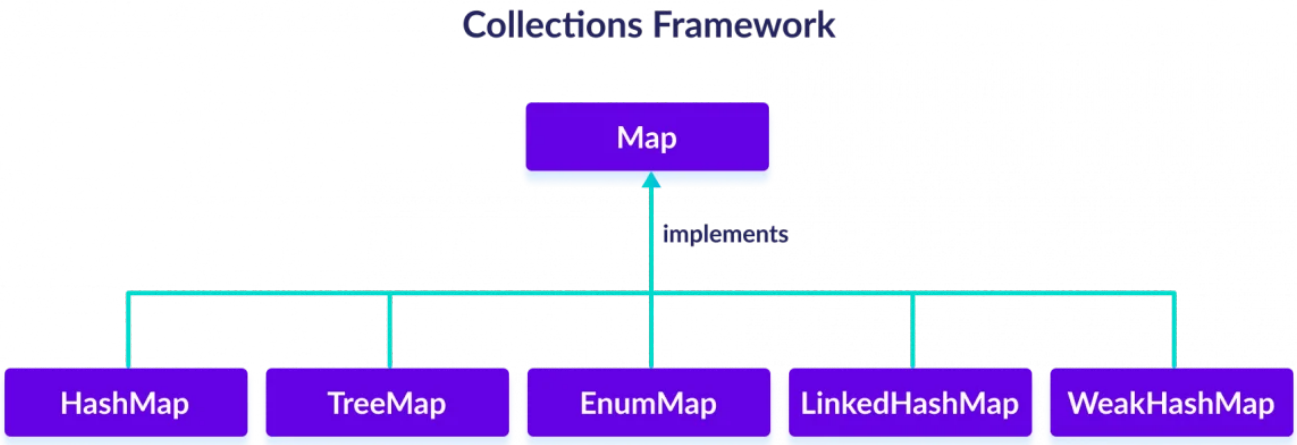


Set is implemented by:

- `HashSet` - It represents the unordered set of elements which doesn't allow us to store the duplicate items. It is the best approach for search operations.
- `LinkedHashSet` - It represents the ordered set of elements which doesn't allow us to store the duplicate items. It is the child class of HashSet.
- `TreeSet` - It contains unique elements only like HashSet. The TreeSet class implements NavigableSet interface that extends the SortedSet interface. It maintains ascending order.
- `EnumSet` - It is the specialized Set implementation for use with enum types. It inherits AbstractSet class and implements the Set interface.

Map Interface

The map is an object that stores the data in the form of key-value pairs. The Map interface is not a subtype of the Collection interface. Therefore it behaves a bit different from the rest of the collection types.



The Map interface is implemented by:

- `HashMap` - It represents the implementation of the Map interface. It represents a mapping between a key and a value. It inherits AbstractMap class and implements Map interface.
- `LinkedHashMap` - It represents the implementation of the Map interface and extends the HashMap class. Like HashMap, It also contains unique elements. It maintains the insertion order and permits null elements.
- `TreeMap` - It represents the implementation of the Map interface that uses a tree for storage. Like HashMap, TreeMap also contains unique elements. However, the access and retrieval time of TreeMap is quite fast. The elements in TreeMap stored in the sorted and ascending order. It cannot contain the null key but can have multiple null values.
- `ConcurrentHashMap` - It is a thread-safe class that represents the implementation of the Map interface. It is used to store the key-value pairs. It is introduced in Java 1.5. It does not allow null keys and null values.

Additional reading - Internals

ArrayList

An `ArrayList` in Java is implemented as a resizable array, also known as a dynamic array. It provides an interface to work with a dynamically sized list of elements, allowing for efficient insertion, deletion, and

random access. Here's how an `ArrayList` is typically implemented:

1. **Backing Array:** The core of an `ArrayList` is an underlying array that holds the elements. This array is initially created with a default size, and elements are stored sequentially in it.
2. **Resizing:** As elements are added to the `ArrayList`, the backing array may become full. When this happens, a new larger array is created, and the existing elements are copied from the old array to the new one. This process is called resizing or resizing the array.
3. **Dynamic Sizing:** The resizing operation ensures that the `ArrayList` can dynamically grow or shrink in size as needed. This dynamic sizing is a key feature that differentiates it from a regular array.
4. **Index-Based Access:** `ArrayList` allows elements to be accessed by their index. This is achieved by directly accessing the corresponding element in the backing array using the index.
5. **Insertion and Deletion:** When an element is inserted at a specific index or removed from the `ArrayList`, the other elements may need to be shifted to accommodate the change. This can involve moving multiple elements within the array.
6. **Efficiency:** `ArrayList` provides efficient constant-time (O(1)) access to elements by index. However, insertion or deletion operations at the beginning or middle of the list may require shifting elements and take linear time (O(n)), where n is the number of elements.
7. **Automatic Resizing:** Java's `ArrayList` uses automatic resizing strategies to ensure that the array is appropriately resized when needed. The exact resizing strategy can vary across different implementations and versions of Java.

Here's a simplified illustration of how an `ArrayList` might be implemented:

```
ArrayList:
[Elem1, Elem2, Elem3, ..., ElemN]

Backing Array:
[E1, E2, E3, ..., EN]

Adding an element:
1. Check if the backing array is full.
2. If full, create a new larger array (e.g., double the size) and copy
elements.
3. Add the new element to the end of the array.

Removing an element:
1. Locate the index of the element to be removed.
2. Shift all elements after that index one position to the left.
3. Update the size of the ArrayList.

Accessing an element:
1. Calculate the index based on the ArrayList index.
2. Retrieve the element directly from the backing array at the calculated
index.
```

HashSet

A `HashSet` in Java is implemented as a hash table data structure. It is part of the Java Collections Framework and is used to store a collection of distinct elements where duplicates are not allowed. The hash table is designed to provide fast average-case time complexity for basic operations like add, remove, and contains.

Here's an overview of how a `HashSet` is implemented:

1. **Hashing Function:** When an element is added to the `HashSet`, the hashing function is used to convert the element's value into an integer hash code. This hash code is used to determine the index (bucket) in the internal array where the element will be stored.
2. **Buckets and Chaining:** The internal array of the `HashSet` is divided into a fixed number of buckets. Each bucket can contain multiple elements that have the same hash code. To handle hash collisions (when different elements produce the same hash code), the `HashSet` uses a technique called chaining. In chaining, each bucket is implemented as a linked list of elements.
3. **Adding Elements:** When an element is added, its hash code is used to find the appropriate bucket. If the bucket is empty, the element is added as the first node in the linked list. If the bucket already contains elements, the new element is added to the end of the linked list.
4. **Retrieving Elements:** When searching for an element using the `contains` method, the hash code is used to locate the appropriate bucket. Then, the linked list within that bucket is traversed to find the desired element.
5. **Removing Elements:** To remove an element, the hash code is used to locate the correct bucket, and then the linked list is searched for the element. If found, the element is removed from the linked list.
6. **Resizing and Load Factor:** As elements are added to the `HashSet`, the load factor (ratio of elements to buckets) is monitored. If the load factor exceeds a certain threshold (default is 0.75), the `HashSet` is resized, which involves creating a larger array and rehashing the elements into the new buckets. This helps maintain efficient access times even as the number of elements increases.

It's important to note that while the average-case time complexity of basic operations is O(1), this is not always guaranteed. In cases of hash collisions or load factor-induced resizing, the time complexity can degrade to O(n) in the worst case.

Comparables

The `Comparable` interface in Java is used to define a natural ordering for a class. When a class implements the `Comparable` interface, it provides a way to compare instances of that class with each other. This natural ordering is primarily used for sorting elements in various collections like `TreeSet` or when using sorting algorithms like `Collections.sort()`.

The `Comparable` interface contains a single method:

```
int compareTo(T other)
```

Here, `T` represents the type of objects being compared. The `compareTo` method should return a negative integer, zero, or a positive integer based on whether the current object is less than, equal to, or greater than the object being compared (`other`).

For example, consider a `Person` class that implements the `Comparable` interface to define a natural ordering based on age:

```
import java.util.*;

class Person implements Comparable<Person> {
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age);
    }
}
```

Now, instances of the `Person` class can be easily compared and sorted using their natural ordering (in this case, based on age) without explicitly providing a separate comparator. This is particularly useful when working with collections that require sorting or maintaining a natural order, such as `TreeSet` or `Collections.sort()`.

Comparators

In Java, a `Comparator` is an interface that provides a way to define custom ordering for objects in collections, such as lists or sets. It allows you to specify how elements should be compared and sorted based on specific criteria that you define. The `Comparator` interface is particularly useful when you want to sort objects in a way that differs from their natural order or when dealing with classes that don't implement the `Comparable` interface.

Here's how the `Comparator` interface is typically used:

1. **Creating a Comparator:** You can create a class that implements the `Comparator` interface. This class should provide the logic for comparing two objects based on the desired criteria.
2. **Comparison Logic:** The `Comparator` interface requires you to implement the `compare` method, which takes two objects as parameters and returns a negative, zero, or positive integer depending on whether the first object is less than, equal to, or greater than the second object, respectively. This method allows you to define the custom ordering logic.
3. **Using the Comparator:** Once you have a `Comparator` implementation, you can use it in various ways:

- Sorting collections: You can pass the `Comparator` to sorting methods like `Collections.sort()` or `Arrays.sort()` to sort the elements in the desired order.
- Creating sorted collections: You can create collections (like `TreeSet` or `PriorityQueue`) that maintain elements in a sorted order using the provided `Comparator`.
- Custom sorting: You can use the `Comparator` to perform custom sorting tasks based on specific use cases.

Here's a simple example of how you might use a `Comparator` to sort a list of `Person` objects based on their ages:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

class Person {
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

class AgeComparator implements Comparator<Person> {
    @Override
    public int compare(Person person1, Person person2) {
        return Integer.compare(person1.age, person2.age);
    }
}

public class ComparatorExample {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 28));
        people.add(new Person("Bob", 22));
        people.add(new Person("Charlie", 25));

        // Sort the list using the AgeComparator
        Collections.sort(people, new AgeComparator());

        for (Person person : people) {
            System.out.println(person.name + " - " + person.age);
        }
    }
}
```

In this example, the `AgeComparator` class implements the `Comparator` interface to compare `Person` objects based on their ages. The `Collections.sort()` method uses the `AgeComparator` to sort the list

of `Person` objects. This allows you to customize the sorting behavior without modifying the `Person` class itself.

Using comparators, you can achieve flexible and custom sorting behavior for different scenarios without changing the original class's implementation or natural order.

TreeSet

A `TreeSet` is a type of collection in Java that implements the `Set` interface and stores its elements in a sorted tree structure. Unlike `HashSet` and `LinkedHashSet`, which use hash-based storage and maintain insertion order, respectively, `TreeSet` orders its elements according to their natural ordering or a specified comparator.

Here's an explanation of `TreeSet` and why it might be considered better than `HashSet` and `LinkedHashSet` in certain situations:

TreeSet:

1. **Ordered Storage:** `TreeSet` maintains its elements in a sorted order. This ordering provides several advantages, especially when you need to traverse the elements in a specific sequence.
2. **Natural Ordering or Custom Comparator:** Elements in a `TreeSet` are sorted either according to their natural ordering (for elements that implement the `Comparable` interface) or a custom comparator that you can provide.
3. **Balanced Tree Structure:** A self-balancing binary search tree (usually a Red-Black Tree) is used internally to store elements. This ensures efficient insertion, deletion, and retrieval operations.
4. **Efficient Range Queries:** Because elements are sorted, `TreeSet` supports efficient range queries using methods like `subSet`, `headSet`, and `tailSet`.
5. **Slower Insertions and Deletions:** The balanced tree structure of `TreeSet` ensures logarithmic time complexity for insertion, deletion, and search operations. While these operations are efficient, they are generally slower than constant-time operations of `HashSet` for non-sorted data.
6. **No Duplicates:** Like other `Set` implementations, `TreeSet` does not allow duplicate elements.

When Is TreeSet Better:

- When you need the elements to be stored in a specific order (natural or custom).
- When you require efficient range queries.
- When you want to prevent duplicates while maintaining sorted order.

Comparison with HashSet and LinkedHashSet:

- **HashSet:** Provides constant-time complexity for insertion, deletion, and search operations. Does not maintain order.
- **LinkedHashSet:** Maintains insertion order in addition to constant-time operations. Does not provide sorted order or efficient range queries like `TreeSet`.

In summary, if you need a collection that maintains a sorted order and allows efficient range queries, `TreeSet` is a good choice. However, keep in mind that the specific use case and requirements should guide

your choice of collection implementation.

1. **Create a TreeSet Instance**:

```
TreeSet<Integer> numbers = new TreeSet<>();
```

2. **Add Elements**: You can add elements to the `TreeSet` using the `add` method. The elements will be automatically sorted.

```
numbers.add(5);
numbers.add(2);
numbers.add(8);
```

3. **Retrieve Elements**: You can use the `foreach` loop or other iteration mechanisms to retrieve elements from the `TreeSet`.

```
for (Integer number : numbers) {
    System.out.println(number);
}
```

4. **Check if an Element Exists**: You can use the `contains` method to check if a specific element exists in the `TreeSet`.

```
boolean exists = numbers.contains(2); // true
```

5. **Remove Elements**: You can remove elements using the `remove` method.

```
numbers.remove(5);
```

6. **Size and Clear**: You can get the number of elements in the `TreeSet` using the `size` method, and you can clear all elements using the `clear` method.

```
int size = numbers.size();
numbers.clear();
```

The `TreeSet` maintains elements in sorted order, which makes it suitable for scenarios where you need a collection of unique elements with a specific order. Keep in mind that the elements you store in a `TreeSet` must be comparable (either naturally, through their `Comparable` implementation, or by providing a custom `Comparator`). This allows the `TreeSet` to maintain the sorting order of the elements.

```
import java.util.*;

class Person {
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

class AgeComparator implements Comparator<Person> {
    @Override
    public int compare(Person person1, Person person2) {
        return Integer.compare(person2.age, person1.age); // Compare ages
in descending order
    }
}

public class TreeSetWithCustomComparator {
    public static void main(String[] args) {
        // Create a TreeSet with the custom comparator
        TreeSet<Person> personSet = new TreeSet<>(new AgeComparator());

        // Add Person objects
        personSet.add(new Person("Alice", 30));
        personSet.add(new Person("Bob", 25));
        personSet.add(new Person("Charlie", 40));

        // Iterate and print the sorted Persons
        for (Person person : personSet) {
            System.out.println("Name: " + person.name + ", Age: " +
person.age);
        }
    }
}
```

HashMap

A `HashMap` in Java is implemented using an array of buckets, where each bucket can hold multiple key-value pairs. The primary goal of a `HashMap` is to provide efficient key-value lookups, insertions, and deletions. Here's an overview of how a `HashMap` is implemented:

1. **Hashing Function:** When you insert a key-value pair into a `HashMap`, the key is passed through a hashing function. The hashing function calculates an index (hash code) that corresponds to the location in the underlying array where the key-value pair should be stored. This index is used to determine the bucket where the pair will be placed.

2. **Buckets:** The array used by the `HashMap` is divided into buckets. Each bucket can hold multiple key-value pairs. The goal is to evenly distribute the key-value pairs among the buckets to ensure efficient

access.

3. **Collision Handling:** Since multiple keys may have the same hash code (collision), a `HashMap` employs a mechanism to handle collisions. One common approach is to use a linked list (or other data structure) within each bucket to store multiple key-value pairs that have the same hash code. This forms a chain of entries that share the same bucket.

4. **Load Factor:** The load factor of a `HashMap` determines when to resize the underlying array to maintain efficient performance. When the number of key-value pairs in the map exceeds a certain threshold (determined by the load factor), the `HashMap` is resized (rehashed) to accommodate more entries. This helps ensure that the number of elements in each bucket remains reasonable.

5. **Resizing:** When the `HashMap` is resized, a new array with a larger number of buckets is created. The existing key-value pairs are rehashed and redistributed into the new array, reducing the likelihood of collisions and ensuring that the load factor remains within an acceptable range.

6. **Hash Code and Equals:** Keys in a `HashMap` must have well-implemented `hashCode()` and `equals()` methods. The `hashCode()` method is used to calculate the initial index (hash code), and the `equals()` method is used to compare keys when searching for a specific key.

Java's `HashMap` aims to provide constant-time (O(1)) average-case performance for basic operations like `get`, `put`, and `remove` under normal circumstances. However, the performance may degrade in the presence of collisions, high load factors, or poorly distributed hash codes. To mitigate this, Java offers other map implementations like `LinkedHashMap`, `TreeMap`, and `ConcurrentHashMap` that cater to different use cases and performance requirements.

PriorityQueue

A `PriorityQueue` in Java is a specialized queue data structure that maintains its elements in a way that allows efficient retrieval of the highest-priority element. It is often used in scenarios where elements need to be processed in a specific order based on their priority.

Key Features of Priority Queue:

1. **Priority Order**: Elements in a `PriorityQueue` are ordered based on their priority, as determined by a specified comparator or the natural ordering of elements (if they implement `Comparable`).

2. **Min-Heap**: Internally, a `PriorityQueue` is implemented as a binary heap, specifically a min-heap. In a min-heap, the smallest element (based on the priority) is always at the root, making retrieval of the highest-priority element very efficient (O(1)).

3. **Addition and Removal**: Adding an element to a `PriorityQueue` (enqueue) and removing the highest-priority element (dequeue) both take O(log n) time complexity, where n is the number of elements in the queue.

4. **No Index Access**: Unlike arrays or lists, `PriorityQueue` does not provide direct index-based access to elements. Elements are removed based on their priority.

5. **Iterable**: While `PriorityQueue` does not allow random access, it is iterable and can be traversed using an iterator.

Usage Examples:

- Task Scheduling: Prioritizing tasks based on urgency or importance.
- Dijkstra's Shortest Path Algorithm: Finding the shortest path in a graph with weighted edges.
- Huffman Coding: Generating optimal prefix codes for data compression.
- A* Search Algorithm: Pathfinding in graphs or grids with heuristics.

Example Usage:

```
import java.util.PriorityQueue;

public class PriorityQueueExample {
    public static void main(String[] args) {
        // Create a min-heap priority queue of integers
        PriorityQueue<Integer> pq = new PriorityQueue<>();

        // Enqueue elements
        pq.offer(5);
        pq.offer(10);
        pq.offer(2);

        // Dequeue and print elements (ordered by priority)
        while (!pq.isEmpty()) {
            System.out.println(pq.poll());
        }
    }
}
```

In this example, elements are enqueued with different priorities, but they are dequeued in ascending order due to the min-heap property of the `PriorityQueue`.

Priority queues are a versatile data structure used to manage and process elements based on their priority, making them a valuable tool in various algorithms and applications.