Example: Java LinkedList as Queue

```
import java.util.LinkedList;
import java.util.Queue;
class Main {
 public static void main(String[] args) {
  Queue<String> languages = new LinkedList<>();
  // add elements
  languages.add("Python");
  languages.add("Java");
  languages.add("C");
  System.out.println("LinkedList: " + languages);
  // access the first element
  String str1 = languages.peek();
  System.out.println("Accessed Element: " + str1);
  // access and remove the first element
  String str2 = languages.poll();
  System.out.println("Removed Element: " + str2);
  System.out.println("LinkedList after poll(): " + languages);
  // add element at the end
  languages.offer("Swift");
  System.out.println("LinkedList after offer(): " + languages);
 }
}
```

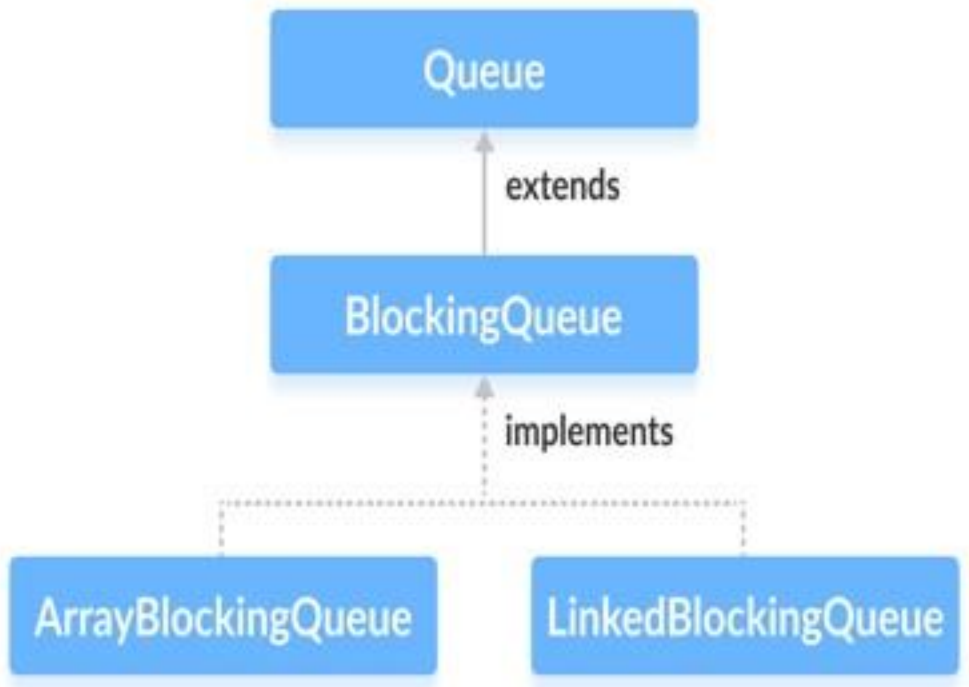
Example: LinkedList as Deque

```
import java.util.LinkedList;
import java.util.Deque;
class Main {
 public static void main(String[] args){
  Deque<String> animals = new LinkedList<>();
  // add element at the beginning
  animals.add("Cow");
  System.out.println("LinkedList: " + animals);
  animals.addFirst("Dog");
  System.out.println("LinkedList after addFirst(): " + animals);
  // add elements at the end
  animals.addLast("Zebra");
  System.out.println("LinkedList after addLast(): " + animals);
  // remove the first element
  animals.removeFirst();
  System.out.println("LinkedList after removeFirst(): " + animals);
  // remove the last element
  animals.removeLast();
  System.out.println("LinkedList after removeLast(): " + animals);
 }
}
```

Java BlockingQueue

The BlockingQueue interface of the Java Collections framework extends the Queue interface. It allows any operation to wait until it can be successfully performed.

For example, if we want to delete an element from an empty queue, then the blocking queue allows the delete operation to wait until the queue contains some elements to be deleted.



How to use blocking queues?

We must import the **java.util.concurrent.BlockingQueue** package in order to use BlockingQueue.

```
// Array implementation of BlockingQueue
BlockingQueue<String> animal1 = new ArraryBlockingQueue<>();

// LinkedList implementation of BlockingQueue
BlockingQueue<String> animal2 = new LinkedBlockingQueue<>();
```

Here, we have created objects animal1 and animal2 of classes ArrayBlockingQueue and LinkedBlockingQueue, respectively. These objects can use the functionalities of the BlockingQueue interface.

Methods of BlockingQueue

Based on whether a queue is full or empty, methods of a blocking queue can be divided into 3 categories:

Methods that throw an exception

add() - Inserts an element to the blocking queue at the end of the queue. Throws an exception if the queue is full.

element() - Returns the head of the blocking queue. Throws an exception if the queue is empty.

remove() - Removes an element from the blocking queue. Throws an exception if the queue is empty.

Methods that return some value

offer() - Inserts the specified element to the blocking queue at the end of the queue. Returns false if the queue is full.
peek() - Returns the head of the blocking queue. Returns null if the queue is empty.
poll() - Removes an element from the blocking queue. Returns null if the queue is empty.

The offer() and poll() method can be used with timeouts. That is, we can pass time units as a parameter. For example,

offer(value, 100, milliseconds)

Here,
value is the element to be inserted to the queue
And we have set a timeout of 100 milliseconds
This means the offer() method will try to insert an element to the blocking queue for 100 milliseconds. If the element cannot be inserted in 100 milliseconds, the method returns false.

Note: Instead of milliseconds, we can also use these time units: days, hours, minutes, seconds, microseconds and nanoseconds in offer() and poll() methods.

Methods that blocks the operation

The BlockingQueue also provides methods to block the operations and wait if the queue is full or empty.

put() - Inserts an element to the blocking queue. If the queue is full, it will wait until the queue has space to insert an element.

take() - Removes and returns an element from the blocking queue. If the queue is empty, it will wait until the queue has elements to be deleted.

Why BlockingQueue?

In Java, BlockingQueue is considered as the thread-safe collection. It is because it can be helpful in multi-threading operations.

Suppose one thread is inserting elements to the queue and another thread is removing elements from the queue.

Now, if the first thread runs slower, then the blocking queue can make the second thread wait until the first thread completes its operation.

put() method

To add an element to the end of an array blocking queue, we can use the put() method.

If the array blocking queue is full, it waits until there is space in the array blocking queue to add an element.
For example,

```
import java.util.concurrent.ArrayBlockingQueue;
class Main {
   public static void main(String[] args) {
      ArrayBlockingQueue<String> animals = new ArrayBlockingQueue<>(5);
     try {
      // Add elements to animals
        animals.put("Dog");
        animals.put("Cat");
        System.out.println("ArrayBlockingQueue: " + animals);
      }
      catch(Exception e) {
         System.out.println(e);
      }
   }
}
```

take() Method

To return and remove an element from the front of the array blocking queue, we can use the take() method.
If the array blocking queue is empty, it waits until there are elements in the array blocking queue to be deleted.

For example,

```
import java.util.concurrent.ArrayBlockingQueue;
class Main {
   public static void main(String[] args) {
      ArrayBlockingQueue<String> animals = new ArrayBlockingQueue<>(5);
     try {
        //Add elements to animals
        animals.put("Dog");
        animals.put("Cat");
        System.out.println("ArrayBlockingQueue: " + animals);
        // Remove an element
        String element = animals.take();
        System.out.println("Removed Element: " + element);
      }
      catch(Exception e) {
         System.out.println(e);
      }
   }
}
```

Java Map Interface

Working of Map

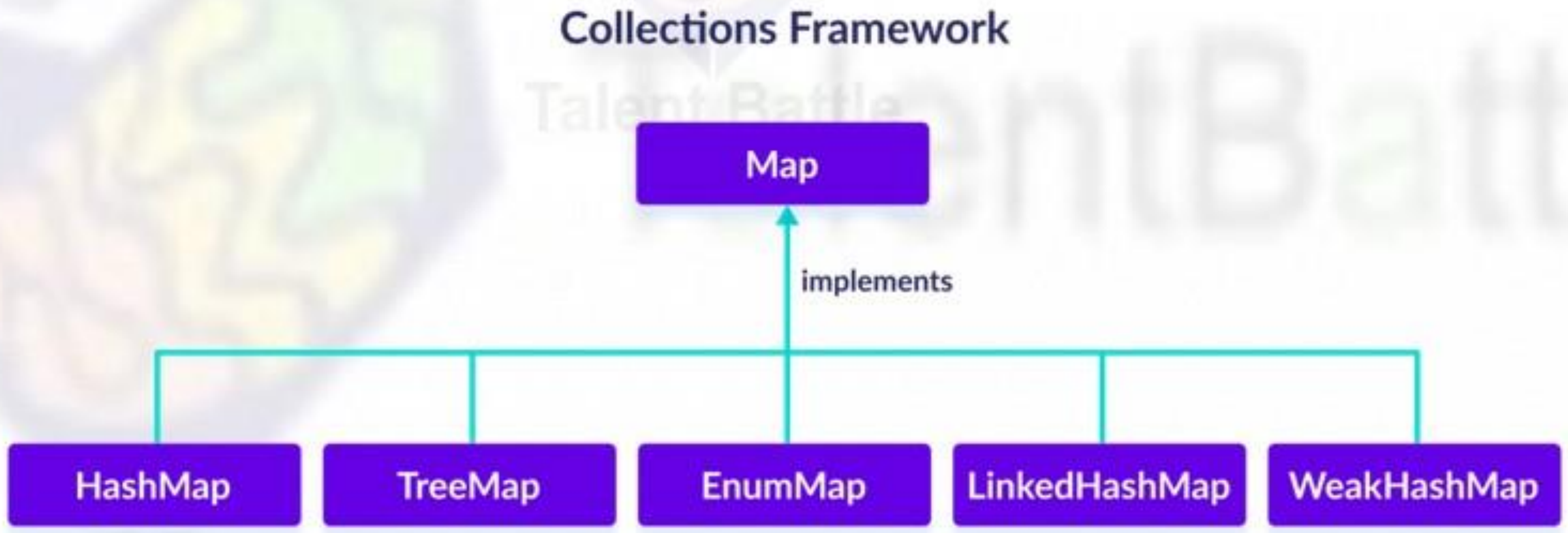
In Java, elements of Map are stored in key/value pairs. Keys are unique values associated with individual Values.

A map cannot contain duplicate keys. And, each key is associated with a single value.

Classes that implement Map

Since Map is an interface, we cannot create objects from it.

In order to use functionalities of the Map interface, we can use these classes:



How to use Map?

In Java, we must import the java.util.Map package in order to use Map. Once we import the package, here's how we can create a map.

```
// Map implementation using HashMap
Map<Key, Value> numbers = new HashMap<>();
```

In the above code, we have created a Map named numbers. We have used the HashMap class to implement the Map interface.

1. Implementing HashMap Class

```java
import java.util.Map;
import java.util.HashMap;

class Main {
  public static void main(String[] args) {
    // Creating a map using the HashMap
    Map<String, Integer> numbers = new HashMap<>();
    // Insert elements to the map
    numbers.put("One", 1);
    numbers.put("Two", 2);
    System.out.println("Map: " + numbers);
    // Access keys of the map
    System.out.println("Keys: " + numbers.keySet());
    // Access values of the map
    System.out.println("Values: " + numbers.values());
    // Access entries of the map
    System.out.println("Entries: " + numbers.entrySet());
    // Remove Elements from the map
    int value = numbers.remove("Two");
    System.out.println("Removed Value: " + value);
  }
}
```

2. Implementing TreeMap Class

```
import java.util.Map;
import java.util.TreeMap;

class Main {
  public static void main(String[] args) {
    // Creating Map using TreeMap
    Map<String, Integer> values = new TreeMap<>();

    // Insert elements to map
    values.put("Second", 2);
    values.put("First", 1);
    System.out.println("Map using TreeMap: " + values);

    // Replacing the values
    values.replace("First", 11);
    values.replace("Second", 22);
    System.out.println("New Map: " + values);

    // Remove elements from the map
    int removedValue = values.remove("First");
    System.out.println("Removed Value: " + removedValue);
  }
}
```

Iterate through a HashMap

```
import java.util.HashMap;
import java.util.Map.Entry;
class Main {
 public static void main(String[] args) {
  // create a HashMap
  HashMap<Integer, String> languages = new HashMap<>();
  languages.put(1, "Java");
  languages.put(2, "Python");
  languages.put(3, "JavaScript");
  System.out.println("HashMap: " + languages);
  // iterate through keys only
  System.out.print("Keys: ");
  for (Integer key : languages.keySet()) {
   System.out.print(key);
   System.out.print(", ");
  }
  // iterate through values only
  System.out.print("\nValues: ");
  for (String value : languages.values()) {
   System.out.print(value);
   System.out.print(", ");
  }
  // iterate through key/value entries
  System.out.print("\nEntries: ");
  for (Entry<Integer, String> entry : languages.entrySet()) {
   System.out.print(entry);
   System.out.print(", ");
  }
 }
}
```

Java LinkedHashMap

Creating a LinkedHashMap

In order to create a linked hashmap, we must import the java.util.LinkedHashMap package first. Once we import the package, here is how we can create linked hashmaps in Java.

```
// LinkedHashMap with initial capacity 8 and load factor 0.6
LinkedHashMap<Key, Value> numbers = new LinkedHashMap<>(8, 0.6f);
```

Notice the part new LinkedHashMap<>(8, 0.6). Here, the first parameter is capacity and the second parameter is loadFactor.

capacity - The capacity of this linked hashmap is 8. Meaning, it can store 8 entries.

loadFactor - The load factor of this linked hashmap is 0.6. This means, whenever our hash map is filled by 60%, the entries are moved to a new hash table of double the size of the original hash table.

Default capacity and load factor

It's possible to create a linked hashmap without defining its capacity and load factor. For example,

```
//LinkedHashMap with default capacity and load factor
LinkedHashMap<Key, Value> numbers1 = new LinkedHashMap<>();
```

By default,

the capacity of the linked hashmap will be 16

the load factor will be 0.75

LinkedHashMap Vs. HashMap

Both the LinkedHashMap and the HashMap implements the Map interface. However, there exist some differences between them.

LinkedHashMap maintains a doubly-linked list internally. Due to this, it maintains the insertion order of its elements.

The LinkedHashMap class requires more storage than HashMap. This is because LinkedHashMap maintains linked lists internally.

The performance of LinkedHashMap is slower than HashMap.

Java WeakHashMap

Note: Keys of the weak hashmap are of the **WeakReference** type.

The object of a weak reference type can be garbage collected in Java if the reference is no longer used in the program.

Create a WeakHashMap

In order to create a weak hashmap, we must import the java.util.WeakHashMap package first. Once we import the package, here is how we can create weak hashmaps in Java.

```
//WeakHashMap creation with capacity 8 and load factor 0.6
WeakHashMap<Key, Value> numbers = new WeakHashMap<>(8, 0.6);
```

In the above code, we have created a weak hashmap named numbers.

Default capacity and load factor

It is possible to create a weak hashmap without defining its capacity and load factor. For example,

```
// WeakHashMap with default capacity and load factor
WeakHashMap<Key, Value> numbers1 = new WeakHashMap<>();
```

By default,

the capacity of the map will be 16
the load factor will be 0.75

```
import java.util.WeakHashMap;

class Main {
  public static void main(String[] args) {
    // Creating WeakHashMap of numbers
    WeakHashMap<String, Integer> numbers = new WeakHashMap<>();
    String two = new String("Two");
    Integer twoValue = 2;
    String four = new String("Four");
    Integer fourValue = 4;
    // Inserting elements
    numbers.put(two, twoValue);
    numbers.put(four, fourValue);
    System.out.println("WeakHashMap: " + numbers);
    // Make the reference null
    two = null;
    // Perform garbage collection
    System.gc();

    System.out.println("WeakHashMap after garbage collection: " + numbers);
  }
}
```

Java EnumMap

Creating an EnumMap

In order to create an enum map, we must import the java.util.EnumMap package first. Once we import the package, here is how we can create enum maps in Java.

```
enum Size {
   SMALL, MEDIUM, LARGE, EXTRALARGE
}
```

```
EnumMap<Size, Integer> sizes = new EnumMap<>(Size.class);
```

In the above example, we have created an enum map named sizes.

Here,

Size - keys of the enum that map to values

Integer - values of the enum map associated with the corresponding keys

```
import java.util.EnumMap;

class Main {
  enum Size {
    SMALL, MEDIUM, LARGE, EXTRALARGE
  }
  public static void main(String[] args) {
    // Creating an EnumMap of the Size enum
    EnumMap<Size, Integer> sizes1 = new EnumMap<>(Size.class);
    // Using the put() Method
    sizes1.put(Size.SMALL, 28);
    sizes1.put(Size.MEDIUM, 32);
    System.out.println("EnumMap1: " + sizes1);
    EnumMap<Size, Integer> sizes2 = new EnumMap<>(Size.class);
    // Using the putAll() Method
    sizes2.putAll(sizes1);
    sizes2.put(Size.LARGE, 36);
    System.out.println("EnumMap2: " + sizes2);
  }
}
```

```java
import java.util.EnumMap;

class Main {
  enum Size {
    SMALL, MEDIUM, LARGE, EXTRALARGE
  }
  public static void main(String[] args) {
    // Creating an EnumMap of the Size enum
    EnumMap<Size, Integer> sizes = new EnumMap<>(Size.class);
    sizes.put(Size.SMALL, 28);
    sizes.put(Size.MEDIUM, 32);
    sizes.put(Size.LARGE, 36);
    sizes.put(Size.EXTRALARGE, 40);
    System.out.println("EnumMap: " + sizes);
    // Using the entrySet() Method
    System.out.println("Key/Value mappings: " + sizes.entrySet());
    // Using the keySet() Method
    System.out.println("Keys: " + sizes.keySet());
    // Using the values() Method
    System.out.println("Values: " + sizes.values());
  }
}
```

Java SortedMap Interface

The SortedMap interface of the Java collections framework provides sorting of keys stored in a map.

Class that implements SortedMap

Since SortedMap is an interface, we cannot create objects from it.

In order to use the functionalities of the SortedMap interface, we need to use the class TreeMap that implements it.

How to use SortedMap?

To use the SortedMap, we must import the java.util.SortedMap package first. Once we import the package, here's how we can create a sorted map.

```
// SortedMap implementation by TreeMap class
SortedMap<Key, Value> numbers = new TreeMap<>();
```

We have created a sorted map called numbers using the TreeMap class.

Methods of SortedMap

The SortedMap interface includes all the methods of the Map interface. It is because Map is a super interface of SortedMap.

Besides all those methods, here are the methods specific to the SortedMap interface.

- comparator() - returns a comparator that can be used to order keys in a map
- firstKey() - returns the first key of the sorted map
- lastKey() - returns the last key of the sorted map
- headMap(key) - returns all the entries of a map whose keys are less than the specified key
- tailMap(key) - returns all the entries of a map whose keys are greater than or equal to the specified key
- subMap(key1, key2) - returns all the entries of a map whose keys lies in between key1 and key2 including key1

```
import java.util.SortedMap;
import java.util.TreeMap;

class Main {
   public static void main(String[] args) {
      // Creating SortedMap using TreeMap
      SortedMap<String, Integer> numbers = new TreeMap<>();
      // Insert elements to map
      numbers.put("Two", 2);
      numbers.put("One", 1);
      System.out.println("SortedMap: " + numbers);
      // Access the first key of the map
      System.out.println("First Key: " + numbers.firstKey());
      // Access the last key of the map
      System.out.println("Last Key: " + numbers.lastKey());
      // Remove elements from the map
      int value = numbers.remove("One");
      System.out.println("Removed Value: " + value);
   }
}
```

Java NavigableMap Interface

The NavigableMap interface of the Java collections framework provides the features to navigate among the map entries.

It is considered as a type of SortedMap.

How to use NavigableMap?

In Java, we must import the java.util.NavigableMap package to use NavigableMap. Once we import the package, here's how we can create a navigable map.

```
// NavigableMap implementation by TreeMap class
NavigableMap<Key, Value> numbers = new TreeMap<>();
```

In the above code, we have created a navigable map named numbers of the TreeMap class.

```
import java.util.NavigableMap;
import java.util.TreeMap;

class Main {
  public static void main(String[] args) {
    // Creating NavigableMap using TreeMap
    NavigableMap<String, Integer> numbers = new TreeMap<>();
    // Insert elements to map
    numbers.put("Two", 2);
    numbers.put("One", 1);
    numbers.put("Three", 3);
    System.out.println("NavigableMap: " + numbers);
    // Access the first entry of the map
    System.out.println("First Entry: " + numbers.firstEntry());
    // Access the last entry of the map
    System.out.println("Last Entry: " + numbers.lastEntry());
    // Remove the first entry from the map
    System.out.println("Removed First Entry: " + numbers.pollFirstEntry());
    // Remove the last entry from the map
    System.out.println("Removed Last Entry: " + numbers.pollLastEntry());
  }
}
```

Java ConcurrentMap Interface

The ConcurrentMap interface of the Java collections framework provides a thread-safe map. That is, multiple threads can access the map at once without affecting the consistency of entries in a map.

ConcurrentMap is known as a synchronized map.

How to use ConcurrentMap?

To use the ConcurrentMap, we must import the java.util.concurrent.ConcurrentMap package first. Once we import the package, here's how we can create a concurrent map.

```
// ConcurrentMap implementation by ConcurrentHashMap
CocurrentMap<Key, Value> numbers = new ConcurrentHashMap<>();
```

```
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.ConcurrentHashMap;

class Main {

  public static void main(String[] args) {
    // Creating ConcurrentMap using ConcurrentHashMap
    ConcurrentMap<String, Integer> numbers = new ConcurrentHashMap<>();

    // Insert elements to map
    numbers.put("Two", 2);
    numbers.put("One", 1);
    numbers.put("Three", 3);
    System.out.println("ConcurrentMap: " + numbers);
    // Access the value of specified key
    int value = numbers.get("One");
    System.out.println("Accessed Value: " + value);

    // Remove the value of specified key
    int removedValue = numbers.remove("Two");
    System.out.println("Removed Value: " + removedValue);
  }
}
```