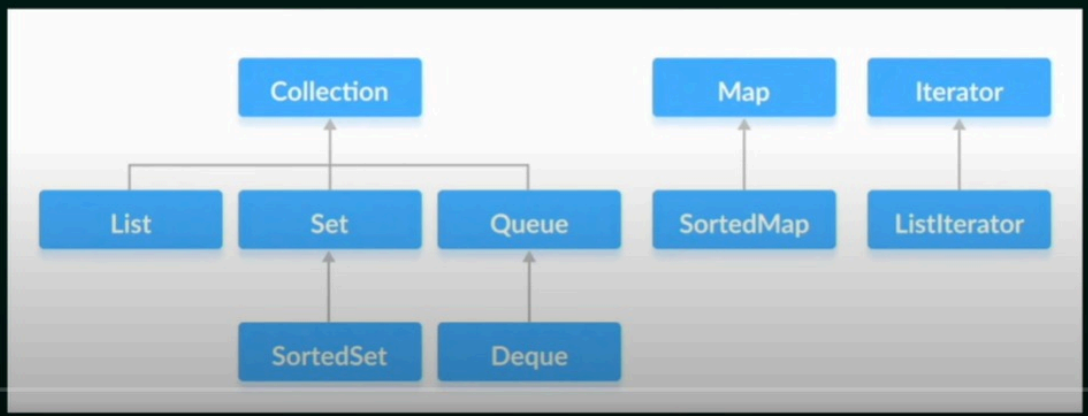
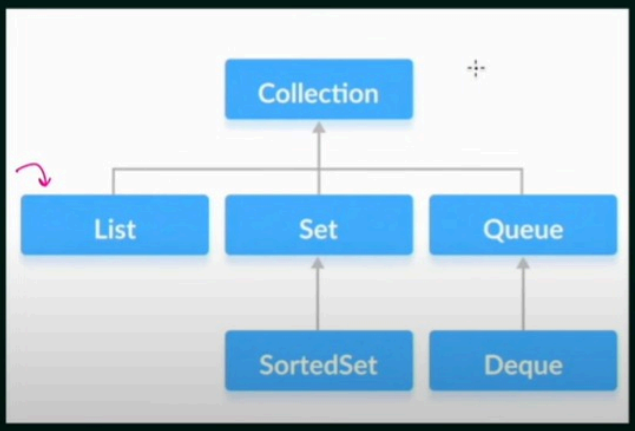
Java Collection Framework

The Java collections framework provides a set of interfaces and classes to implement various data structures and algorithms. These interfaces include several methods to perform different operations on collections.



Java Collection Interface

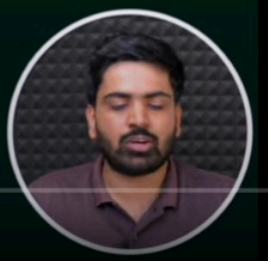
The Collection interface is the root interface of the Java collections framework.



Java Collection Interface

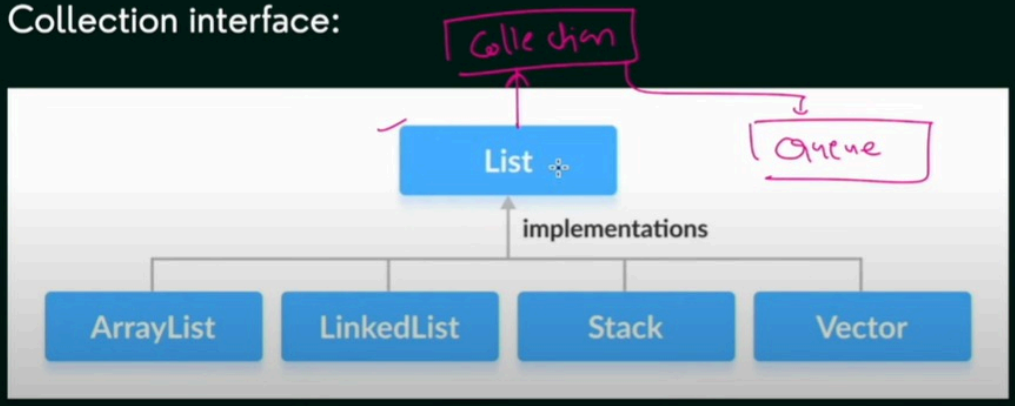
The Collection interface includes various methods that can be used to perform different operations on objects.

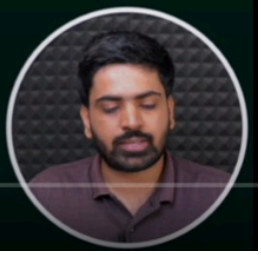
- **int size():** Returns the number of elements in the collection.
- **boolean isEmpty():** Returns **true** if the collection contains no elements.
- **boolean contains(Object o):** Returns **true** if the collection contains the specified element.
- **boolean add(E e):** Adds the specified element to the collection. Returns **true** if the collection changed as a result.
- **boolean remove(Object o):** Removes a single instance of the specified element from the collection, if it is present.
- **boolean containsAll(Collection<> c):** Returns **true** if the collection contains all elements of the specified collection.



Java List Interface

The List interface extends the Collection interface and adds methods that are specific to lists, which are ordered collections that allow duplicate elements. Here are some methods that are present in the List interface but not in the Collection interface:





Java Collection Interface

- **boolean addAll(Collection<> c):** Adds all elements from the specified collection to the collection.
- **boolean removeAll(Collection<> c):** Removes all elements in the collection that are also contained in the specified collection.
- **boolean retainAll(Collection<> c):** Removes all elements from the collection that are not present in the specified collection.
- **void clear():** Removes all elements from the collection.
- **Object[] toArray():** Returns an array containing all elements in the collection.

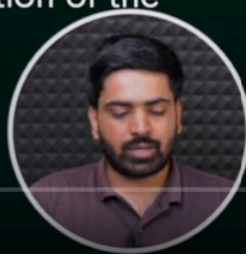
Java List Interface

The List interface extends the Collection interface and adds methods that are specific to lists, which are ordered collections that allow duplicate elements. Here are some methods that are present in the List interface but not in the Collection interface:

- **get(int index):** Retrieves the element at the specified index in the list.
- **set(int index, E element):** Replaces the element at the specified index with the given element.
- **add(int index, E element):** Inserts the specified element at the specified position in the list, shifting the current elements to the right.
- **remove(int index):** Removes the element at the specified index from the list and shifts the remaining elements to the left.

Java List Interface

- **indexOf(Object o):** Returns the index of the first occurrence of the specified element in the list, or -1 if the element is not present.
- **lastIndexOf(Object o):** Returns the index of the last occurrence of the specified element in the list, or -1 if the element is not present.
- **listIterator():** Returns a list iterator over the elements in the list.
- **listIterator(int index):** Returns a list iterator over the elements in the list, starting at the specified index.
- **subList(int fromIndex, int toIndex):** Returns a view of the portion of the list between the specified fromIndex (inclusive) and toIndex (exclusive).



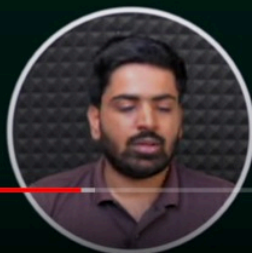
Java ArrayList

In Java, we need to declare the size of an array before we can use it. Once the size of an array is declared, it's hard to change it.
To handle this issue, we can use the ArrayList class. It allows us to create resizable arrays.
Unlike arrays, arraylists can automatically adjust their capacity when we add or remove elements from them. Hence, arraylists are also known as dynamic arrays.

Internal Working of ArrayList

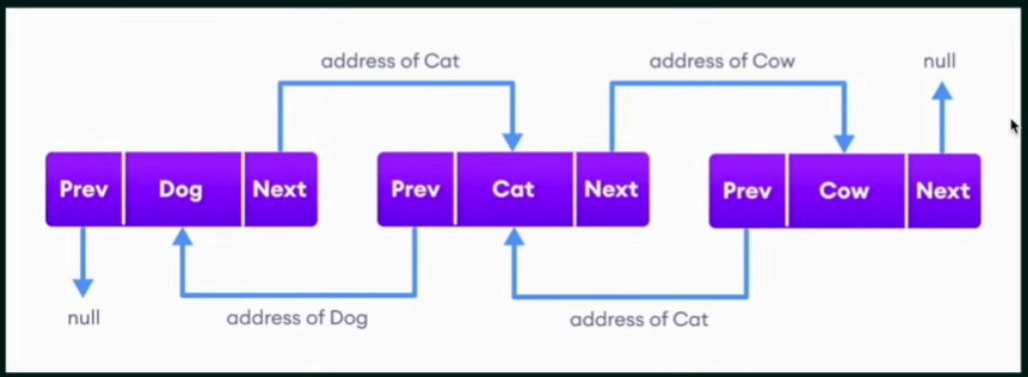
Initially, the array has a certain capacity, and as elements are added, it fills up. When the capacity is reached, the **ArrayList** creates a new larger array and copies the elements from the old array to the new one. This process of resizing and copying is transparent to the user.

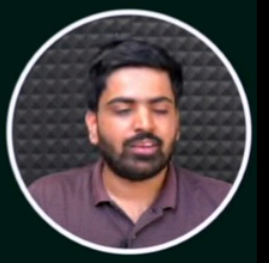
However, frequent resizing operations can lead to performance overhead, so the **ArrayList** increases its capacity by a certain factor to minimize the frequency of resizing.



Java LinkedList

The LinkedList class of the Java collections framework provides the functionality of the linked list data structure (doubly linkedlist).
Elements in linked lists are not stored in sequence. Instead, they are scattered and connected through links (Prev and Next).





Java Vector

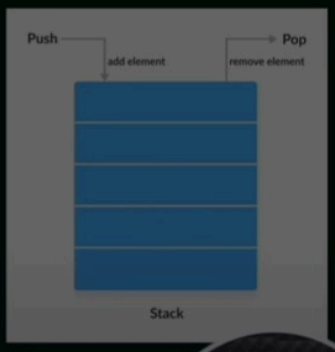
The Vector class synchronizes each individual operation. This means whenever we want to perform some operation on vectors, the Vector class automatically applies a lock to that operation.

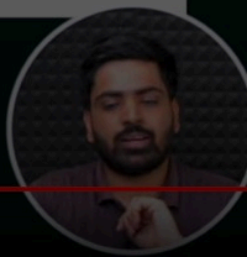
It is because when one thread is accessing a vector, and at the same time another thread tries to access it, an exception called ConcurrentModificationException is generated. Hence, this continuous use of lock for each operation makes vectors less efficient.
However, in array lists, methods are not synchronized.

Java Stack

In stack, elements are stored and accessed in Last In First Out manner. That is, elements are added to the top of the stack and removed from the top of the stack

1. **void push(E item)**: Pushes the given element onto the top of the stack.
2. **E pop()**: Removes and returns the element at the top of the stack. Throws an EmptyStackException if the stack is empty.
3. **E peek()**: Returns the element at the top of the stack without removing it. Throws an EmptyStackException if the stack is empty.
4. **boolean empty()**: Returns true if the stack is empty, false otherwise.





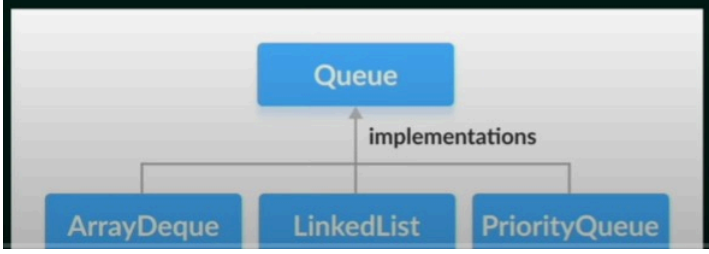
In This Lecture

1. Java Queue Interface
2. Java Queue using LinkedList
3. Java PriorityQueue
4. Java ArrayDeque
5. Java Set Interface
6. Java HashSet
7. Java Set of Custom Objects

Java Queue Interface

CODING SHUTTLE

The Queue interface of the Java collections framework provides the functionality of the queue data structure. It extends the Collection interface.



Java Queue Interface

boolean add(E e): Inserts the specified element into the queue.

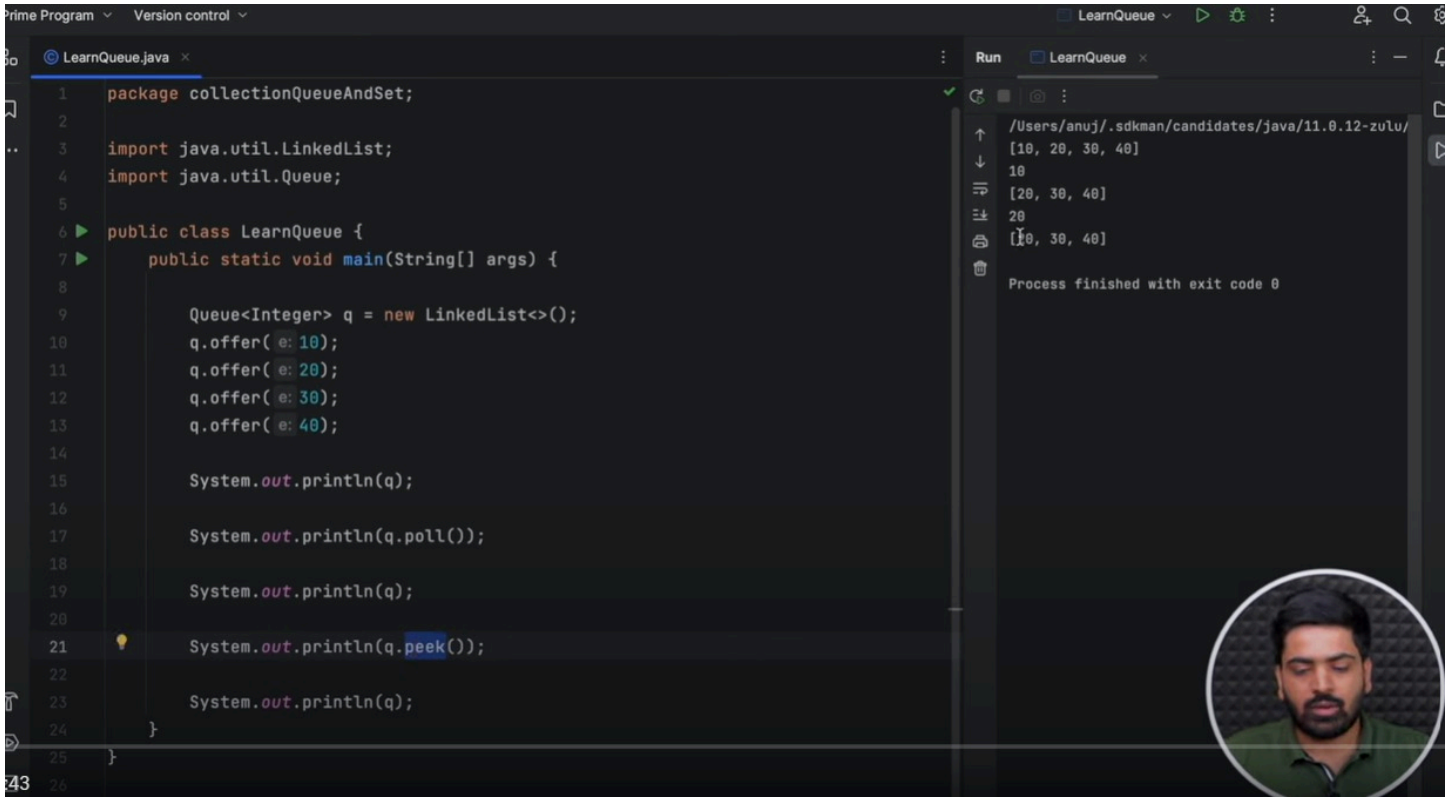
boolean offer(E e): Inserts the specified element into the queue. Returns **true** if the element was added successfully, or **false** if the queue is full.

E remove(): Removes and returns the element at the front of the queue. Throws an exception if the queue is empty.

E poll(): Removes and returns the element at the front of the queue. Returns **null** if the queue is empty.

E element(): Retrieves but does not remove the element at the front of the queue. Throws an exception if the queue is empty.

E peek(): Retrieves but does not remove the element at the front of the queue. Returns **null** if the queue is empty.




```java
package collectionQueueAndSet;

import java.util.LinkedList;
import java.util.Queue;

public class LearnQueue {
    public static void main(String[] args) {

        Queue<Integer> q = new LinkedList<>();
        q.offer( e: 10);
        q.offer( e: 20);
        q.offer( e: 30);
        q.offer( e: 40);

        System.out.println(q);

        System.out.println(q.poll());

        System.out.println(q);

        System.out.println(q.peek());

        System.out.println(q);
    }
}
```

```java
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Queue;

public class LearnDeque {
    public static void main(String[] args) {

        ArrayDeque<Integer> dq = new ArrayDeque<>();
        dq.offer( e: 10);
        dq.offerLast( e: 20);
        dq.offerFirst( e: 30);

        System.out.println(dq);

        System.out.println(dq.poll());
        System.out.println(dq.pollLast());

        System.out.println(dq);

        System.out.println(dq.peekLast());
    }
}
```

```
/Users/anuj/.sdkman/candidates/java/11.0.12-zulu/
[30, 10, 20]
30
20
[10]
10

Process finished with exit code 0
```

Stack and Queue Operations Using ArrayDeque

1. **Stack Operations:**
   - **push(E e)**: Pushes an element onto the stack represented by the deque.
   - **pop()**: Pops an element from the stack represented by the deque.
2. **Queue Operations:**
   - **add(E e)** or **offer(E e)**: Adds an element to the end of the deque, effectively maki it a queue.
   - **remove()** or **poll()**: Removes and returns the element at the front of the deque, making it a queue.

```java
public class LearnDeque {
    public static void main(String[] args) {

        ArrayDeque<Integer> stack = new ArrayDeque<>();

        stack.push( e: 10);
        stack.push( e: 20);
        stack.push( e: 30);
        System.out.println(stack);

        System.out.println(stack.pop());

        System.out.println(stack.peek());
```

```
/Users/anuj/.sdkman/candidates/ja
[30, 20, 10]
30
20

Process finished with exit code 0
```

LearnQueue.java | LearnDeque.java | LearnPriorityQueue.java

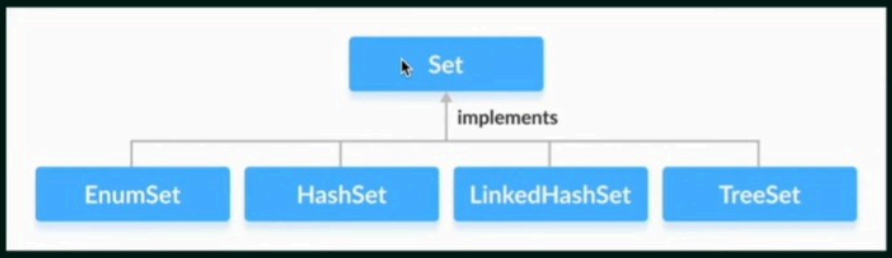
```java
package collectionQueueAndSet;

import java.util.PriorityQueue;
import java.util.Queue;

public class LearnPriorityQueue {
    public static void main(String[] args) {

        Queue<Integer> pq= new PriorityQueue<>((a, b)-> b-a);
        pq.add(30);
        pq.add(40);
        pq.add(10);
        pq.add(20);

        System.out.println(pq);
        System.out.println(pq.poll());
        System.out.println(pq);
        System.out.println(pq.poll());
        System.out.println(pq.poll());

```

Run | LearnPriorityQueue

```
/Users/anuj/.sdkman/candidates/java/11.0.12-zulu/
[40, 30, 10, 20]
40
[30, 20, 10]
30
20

Process finished with exit code 0
```

Java Set Interface

The Set interface of the Java Collections framework provides the features of the mathematical set in Java. It extends the Collection interface.
Unlike the List interface, sets cannot contain duplicate elements.



Java Set Interface

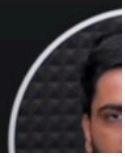
- add() - adds the specified element to the set
- addAll() - adds all the elements of the specified collection to the set
- remove() - removes the specified element from the set
- removeAll() - removes all the elements from the set that is present in another specified set
- retainAll() - retains all the elements in the set that are also present in another specified set
- clear() - removes all the elements from the set
- size() - returns the length (number of elements) of the set
- contains() - returns true if the set contains the specified element

LearnQueue.java | LearnDeque.java | LearnPriorityQueue.java | LearnSets.java

```java
package collectionQueueAndSet;

import java.util.HashSet;
import java.util.Set;

public class LearnSets {
    public static void main(String[] args) {
        Set<Integer> set = new HashSet<>();

        set.add(10);
        set.add(50);
        set.add(90);
        set.add(30);
        set.add(20);


        System.out.println(set);
    }
}
```

Run | LearnSets

```
/Users/anuj/.sdkman/candidates/java/
[50, 20, 10, 90, 30]

Process finished with exit code 0
```



```java
import java.util.Set;
import java.util.TreeSet;

public class LearnSets {
    public static void main(String[] args) {
//        Set<Integer> set = new HashSet<>(); //O(1)
//        Set<Integer> set = new LinkedHashSet<>(); //O(n)
        Set<Integer> set = new TreeSet<>(); //O(logn)

        set.add(10);
        set.add(50);
        set.add(90);
        set.add(30);
        set.add(20);
        set.add(20);
        set.add(20);

        set.remove( o: 20);
        System.out.println(set.contains(10));
```

Java HashSet

- In Java, HashSet is commonly used if we have to access elements random
  It is because elements in a hash table are accessed using hash codes.
- The hashcode of an element is a unique identity that helps to identify the element in a hash table.
- HashSet cannot contain duplicate elements. Hence, each hash set eleme
  has a unique hashcode.

Java HashSet of Custom Objects

When using Set and HashSet in Java, for the primitive types we can just use it without worry about how to implement the hashcode and the comparison logic. But when you want to use the Set with a custom class by putting custom objects into the set, that custom class has to implement the **hashCode() and equals()** methods in order for the HashSet to work.

The Map Interface

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.

Keys	Values
us	United States
br	Brazil
es	Spain