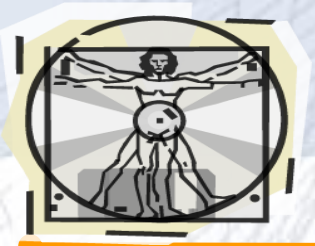
Programming Languages - II
Stacks and Queues

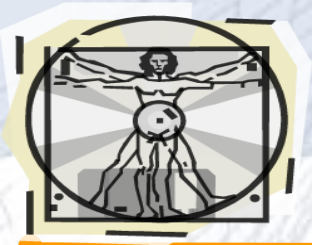
Özgür Koray ŞAHİNGÖZ
Prof.Dr.

Biruni University
Computer Engineering Department



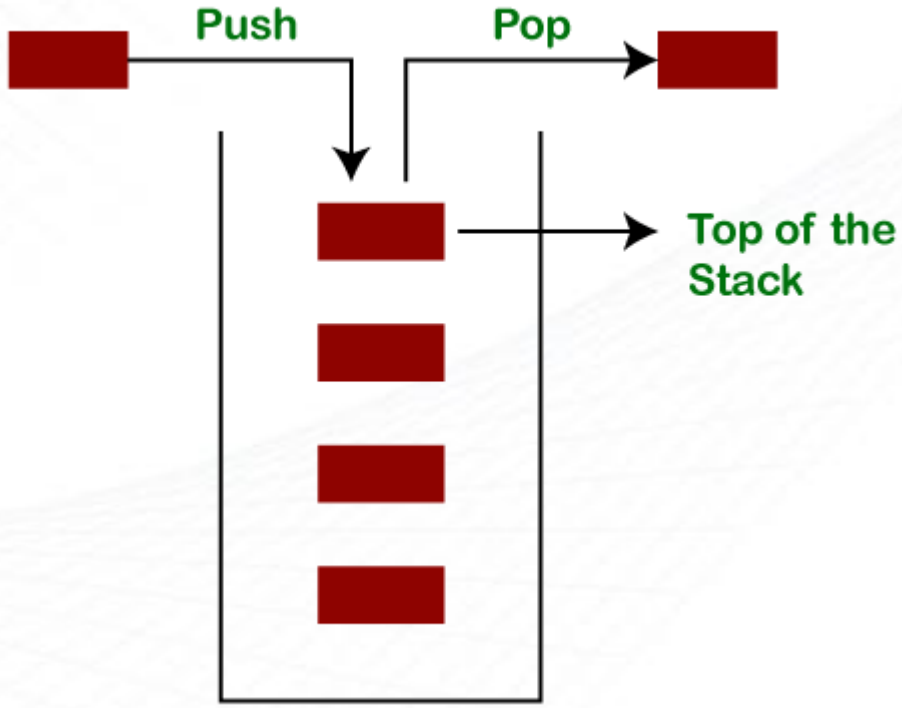
Java Stack

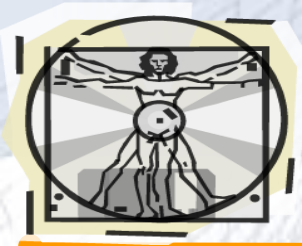
- The **stack** is a linear data structure that is used to store the collection of objects. It is based on **Last-In-First-Out** (LIFO). Java collection framework provides many interfaces and classes to store the collection of objects. One of them is the **Stack class** that provides different operations such as push, pop, search, etc.
- In this section, we will discuss the **Java Stack class**, its **methods,** and **implement** the stack data structure in a Java program. But before moving to the Java Stack class have a quick view of how the stack works.



Stack Operations

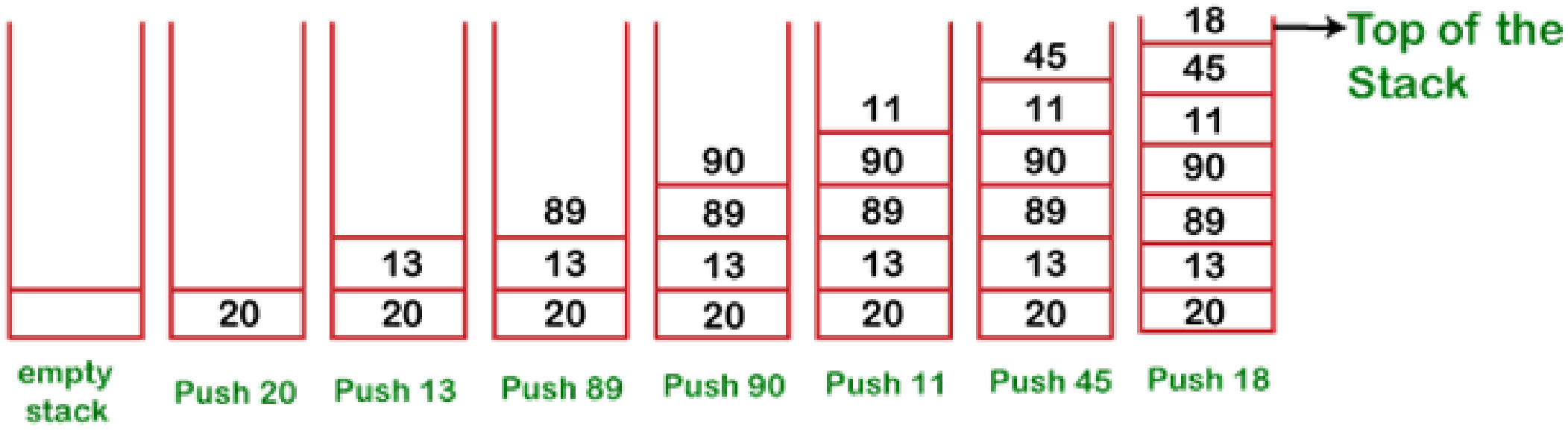
- The stack data structure has the two most important operations that are **push** and **pop**. The push operation inserts an element into the stack and pop operation removes an element from the top of the stack. Let's see how they work on stack.



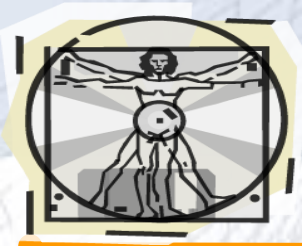


Push Items

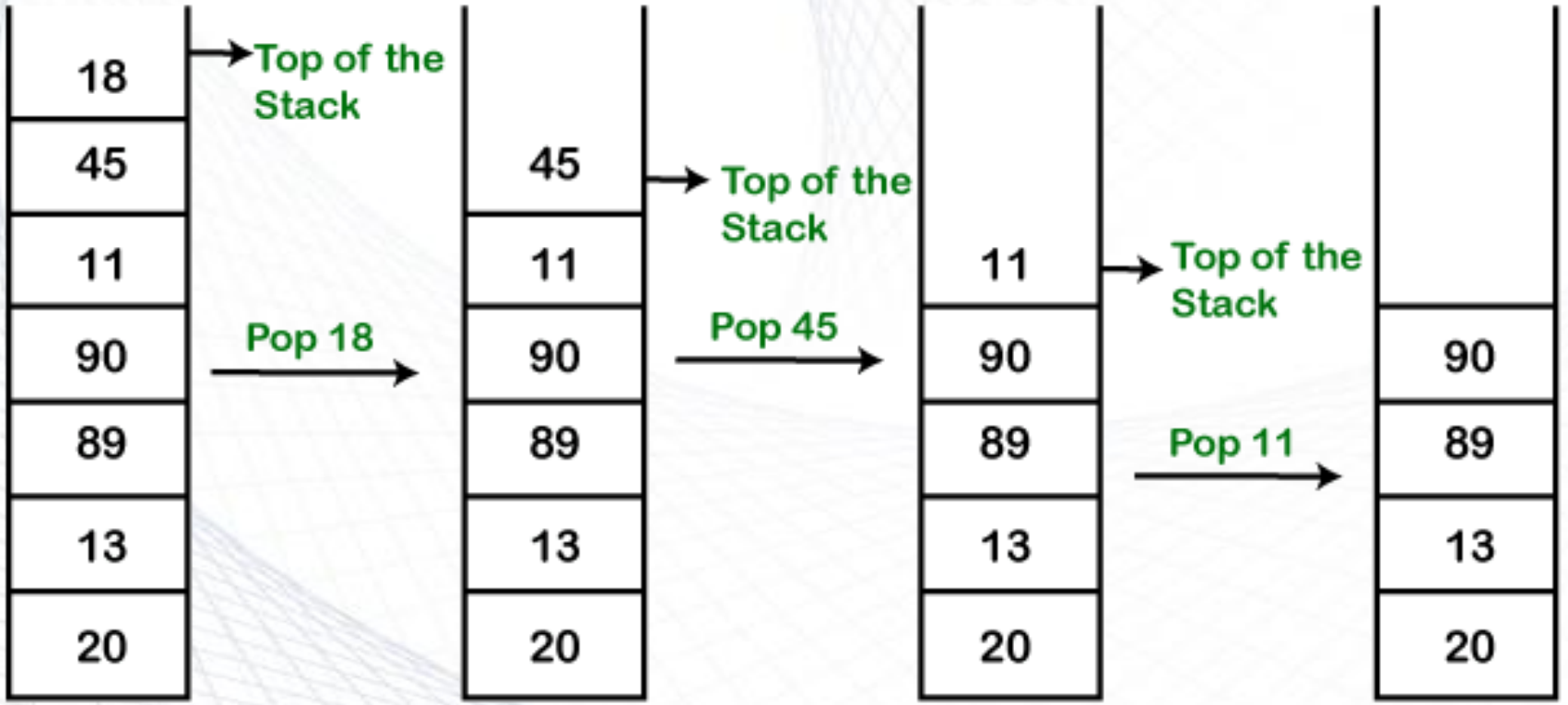
- Let's push 20, 13, 89, 90, 11, 45, 18, respectively into the stack.



Push operation

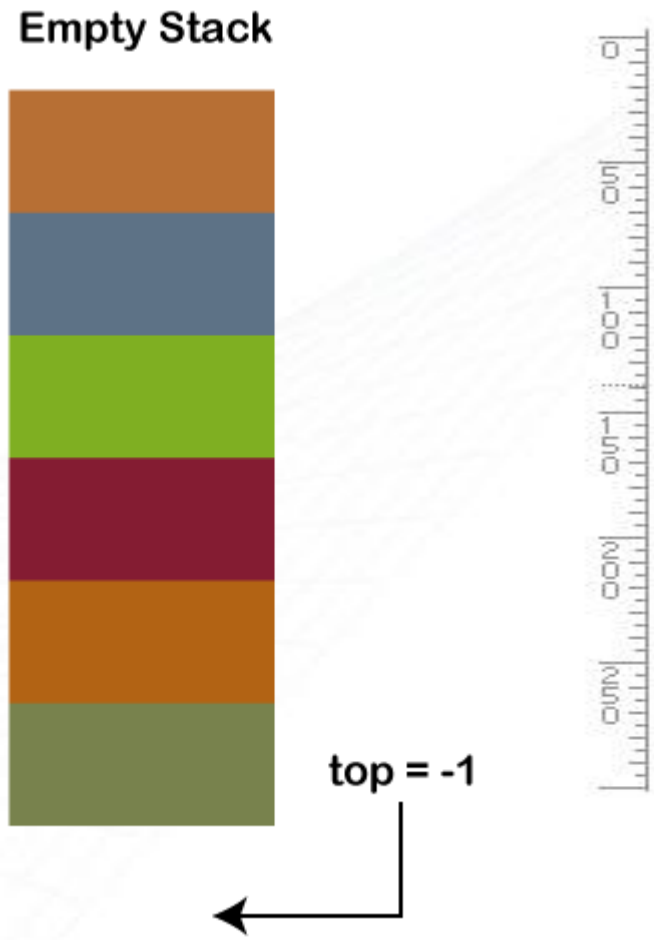


Pop Items

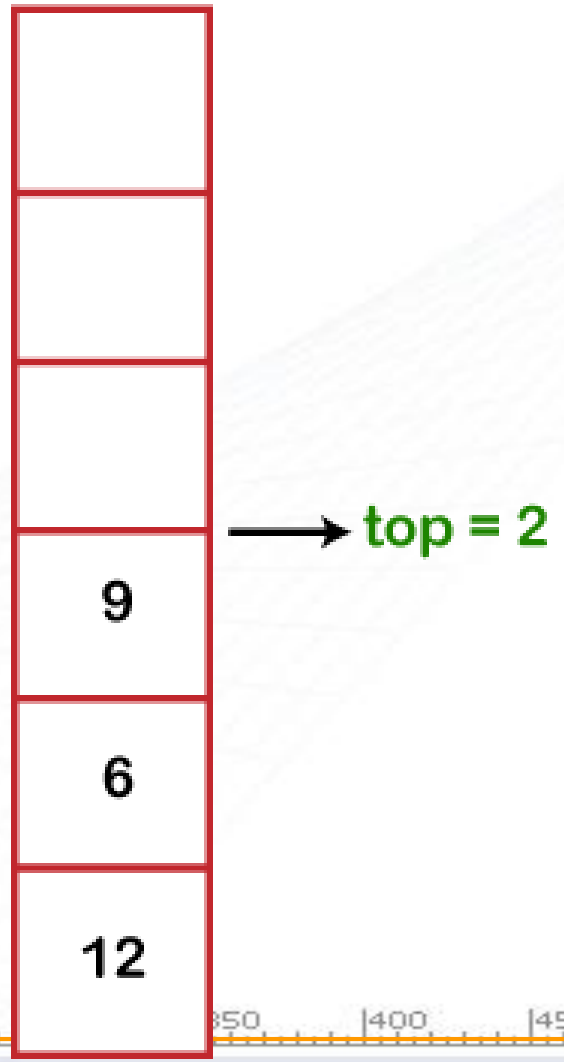


Pop operation

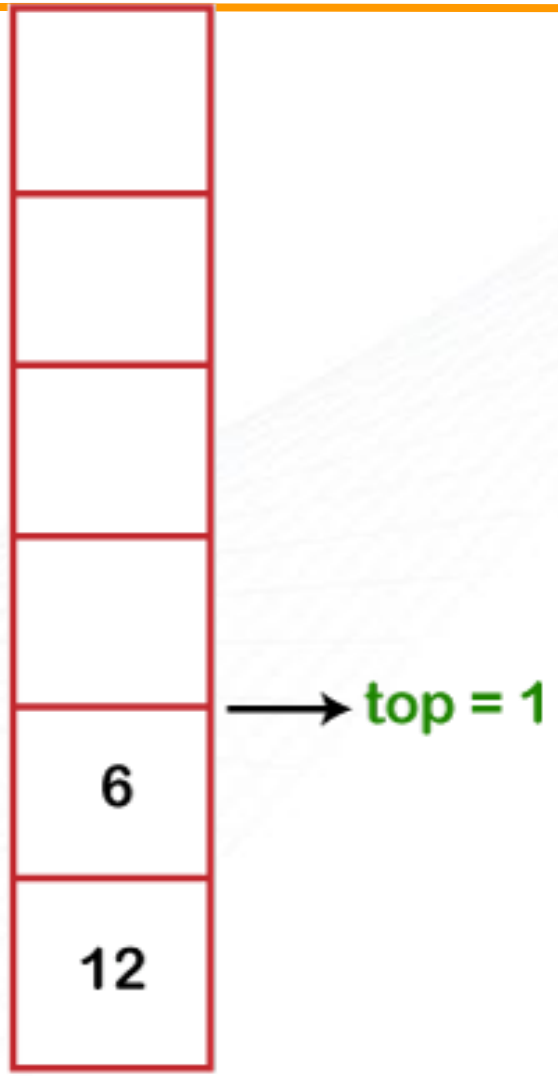
- Empty Stack: If the stack has no element is known as an empty stack. When the stack is empty the value of the top variable is -1.



- When we push an element into the stack the top is **increased by 1**. In the following figure,
- Push 12, top=0
- Push 6, top=1
- Push 9, top=2

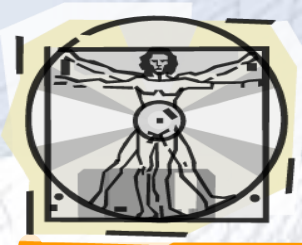


- When we pop an element from the stack the value of top is decreased by 1. In the following figure, we have popped 9.



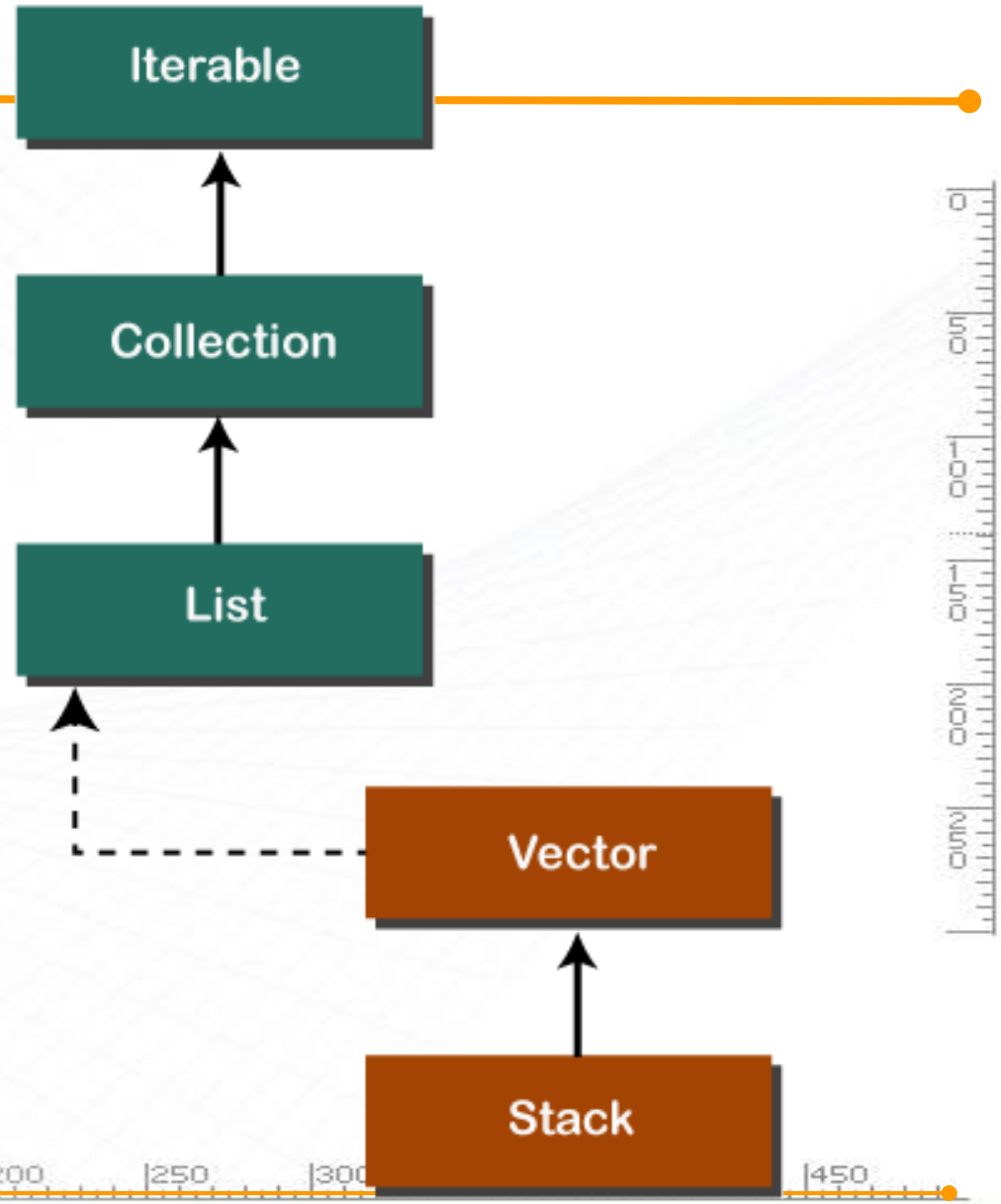
- The following table shows the different values of the top.

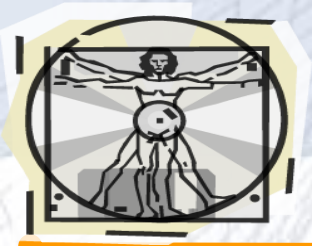
Top value	Meaning
-1	It shows the stack is empty.
0	The stack has only an element.
N-1	The stack is full.
N	The stack is overflow.



Java Stack Class

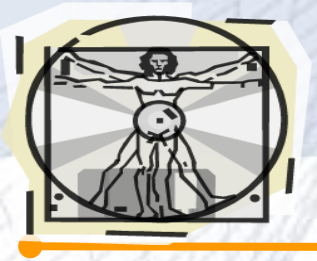
- In Java, **Stack** is a class that falls under the Collection framework that extends the **Vector** class. It also implements interfaces **List, Collection, Iterable, Cloneable, Serializable.** It represents the LIFO stack of objects. Before using the Stack class, we must import the java.util package. The stack class arranged in the Collections framework hierarchy, as shown below.





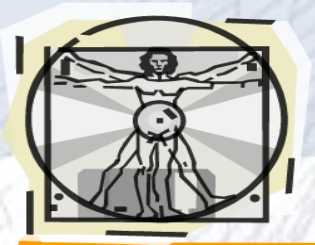
Creating a Stack

- If we want to create a stack, first, import the java.util package and create an object of the Stack class.
  - Stack stk = **new** Stack();
- Or
  - Stack<type> stk = **new** Stack<>();
- Where type denotes the type of stack like Integer, String, etc.



Methods of the Stack Class

Method	Modifier and Type	Method Description
empty()	boolean	The method checks the stack is empty or not.
push(E item)	E	The method pushes (insert) an element onto the top of the stack.
pop()	E	The method removes an element from the top of the stack and returns the same element as the value of that function.
peek()	E	The method looks at the top element of the stack without removing it.
search(Object o)	int	The method searches the specified object and returns the position of the object.

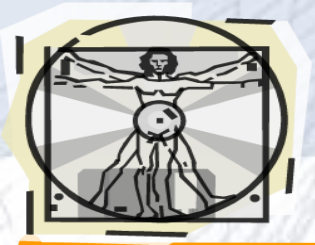


Example

```
Stack<Integer> stk= new Stack<>();    // checking stack is empty or not
boolean result = stk.empty();
System.out.println("Is the stack empty? " + result);
stk.push(78);                         // pushing elements into stack
stk.push(113);
stk.push(90);
stk.push(120);
System.out.println("Elements in Stack: " + stk);          //prints elements of the stack
result = stk.empty();
System.out.println("Is the stack empty? " + result);
```

Output:

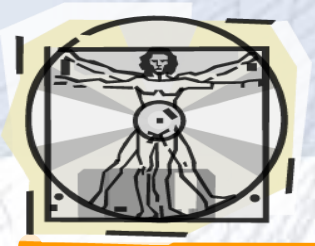
```
Is the stack empty? true
Elements in Stack: [78, 113, 90, 120]
Is the stack empty? false
```



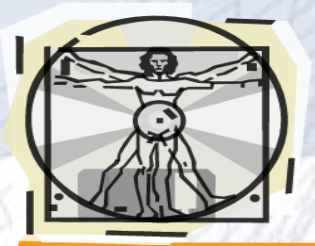
```
Stack<String> stk= new Stack<>();          // pushing elements into Stack
stk.push("Apple");
stk.push("Grapes");
stk.push("Mango");
stk.push("Orange");
System.out.println("Stack: " + stk);          // Access element from the top of the stack
String fruits = stk.peek();
System.out.println("Element at top: " + fruits);    //prints stack
```

Output:

```
Stack: [Apple, Grapes, Mango, Orange]
Element at the top of the stack: Orange
```



```
Stack<String> stk= new Stack<>();          //pushing elements into Stack
stk.push("Mac Book");
stk.push("HP");
stk.push("DELL");
stk.push("Asus");
System.out.println("Stack: " + stk);                  // Search an element
int location = stk.search("HP");
System.out.println("Location of Dell: " + location);
```



Iterator

```
Stack stk = new Stack();     //pushing elements into stack
stk.push("BMW");                  stk.push("Audi");
stk.push("Ferrari");              stk.push("Bugatti");
stk.push("Jaguar");    //iteration over the stack
Iterator iterator = stk.iterator();
while(iterator.hasNext())  {
        Object values = iterator.next();
        System.out.println(values);
}
```

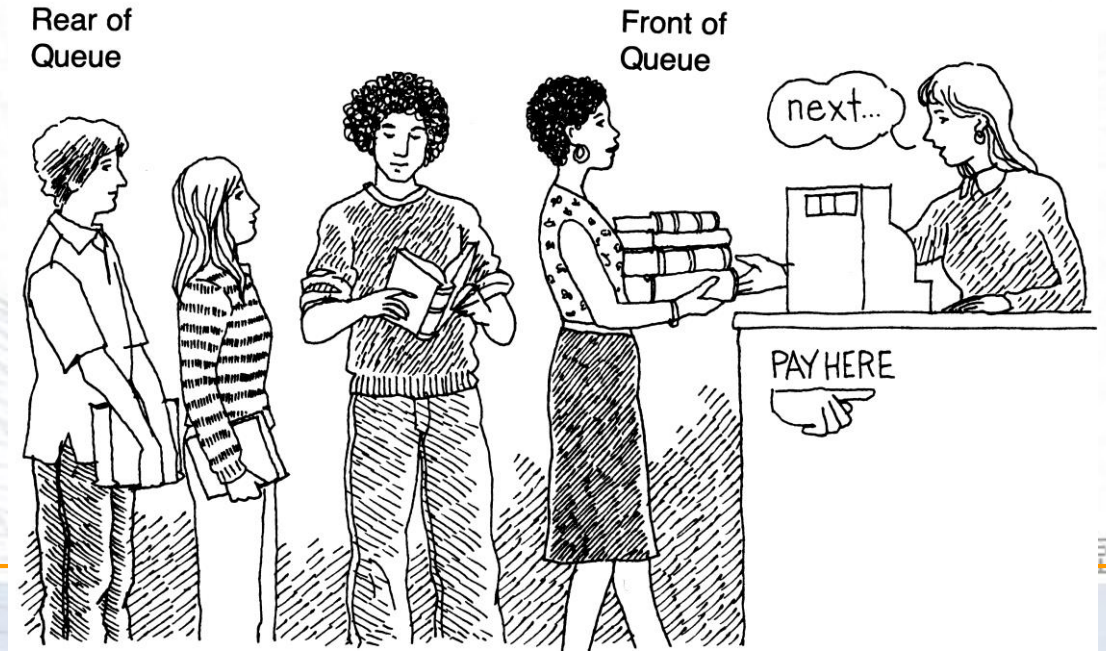
Output:

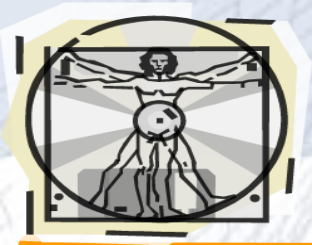
```
BMW
Audi
Ferrari
Bugatti
Jaguar
```

Queues

What is a queue?

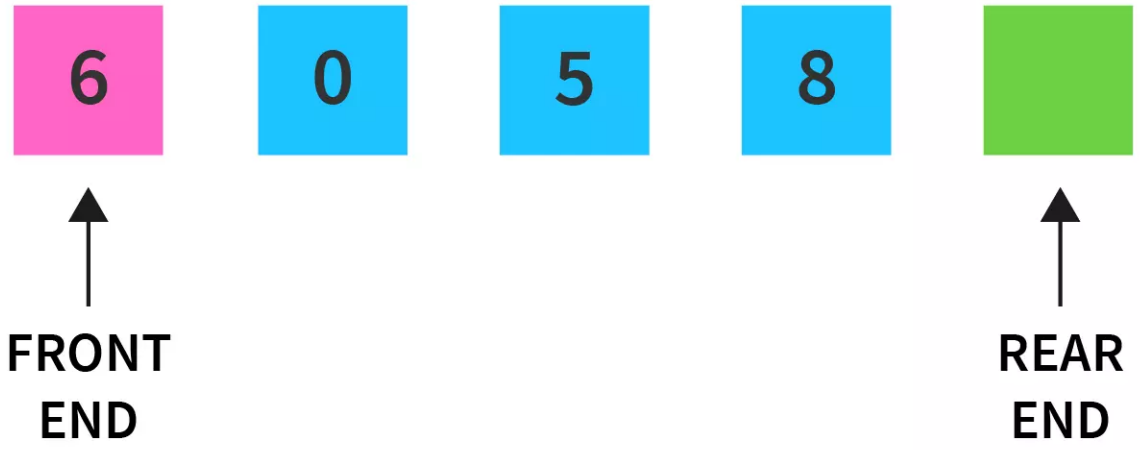
- It is an ordered group of homogeneous items of elements.
- Queues have two ends:
  - Elements are added at one end.
  - Elements are removed from the other end.
- The element added first is also removed first (**FIFO**: First In, First Out).

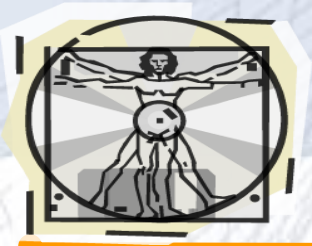




Java Queue Interface

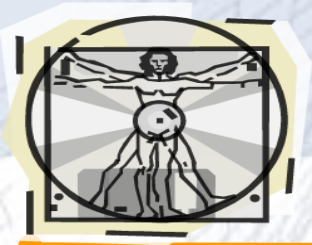
- Java Queue interface orders the element in FIFO(First In First Out) manner. In FIFO, first element is removed first and last element is removed at last.
- A queue is a collection of objects that are inserted at the rear end of the list and are deleted from the front end of the list. We can also perform various other operations on Queue in Java. We can see the element at the front end, know the size of the queue and check if the queue is empty.





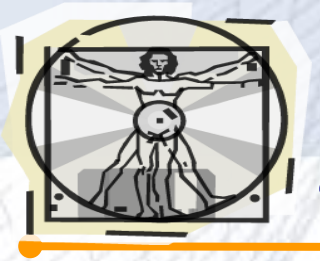
Queues

- You must have stood in a queue while submitting your assignments to your teacher or in a voting line. How does it work? The person standing first submits the assignment first and comes out of the queue.
- Our Queue data structure in java also works in the same way. The element first entered in the queue is removed first as well. No element can be removed from the rear end or from the middle of the queue. Also, no element can be added at the front end or any other position, it has to be added only from the rear end. This is also known as the **first-in, first-out (FIFO)** principle.



Methods

Method	Description
boolean add(object)	It is used to insert the specified element into this queue and return true upon success.
boolean offer(object)	It is used to insert the specified element into this queue.
Object remove()	It is used to retrieves and removes the head of this queue.
Object poll()	It is used to retrieves and removes the head of this queue, or returns null if this queue is empty.
Object element()	It is used to retrieves, but does not remove, the head of this queue.
Object peek()	It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

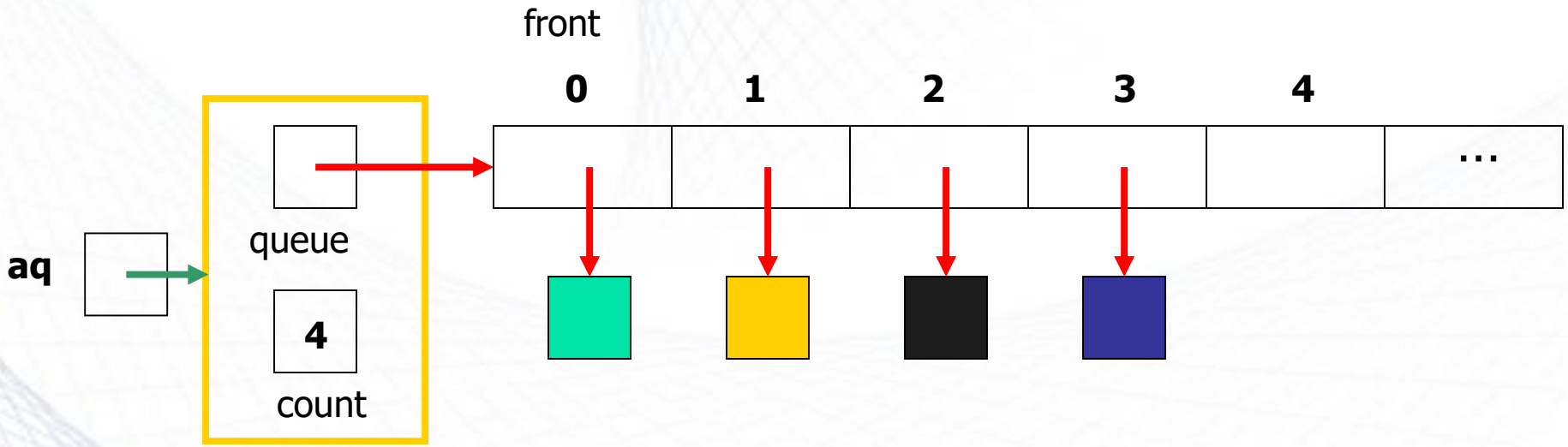


Array Implementation of a Queue

- ***First Approach***:
  - Use an array in which index 0 represents one end of the queue (the *front*)
  - Integer value ***count*** represents the number of elements in the array (so the element at the rear of the queue is in position count - 1)
- ***Discussion***: What is the challenge with this approach?

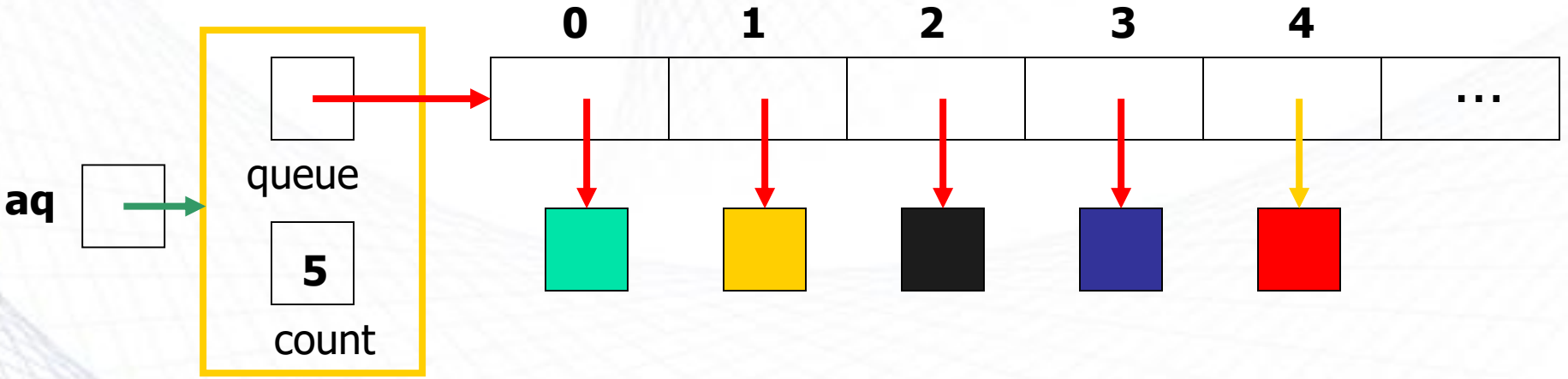
An Array Implementation of a Queue

A queue aq containing four elements

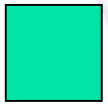


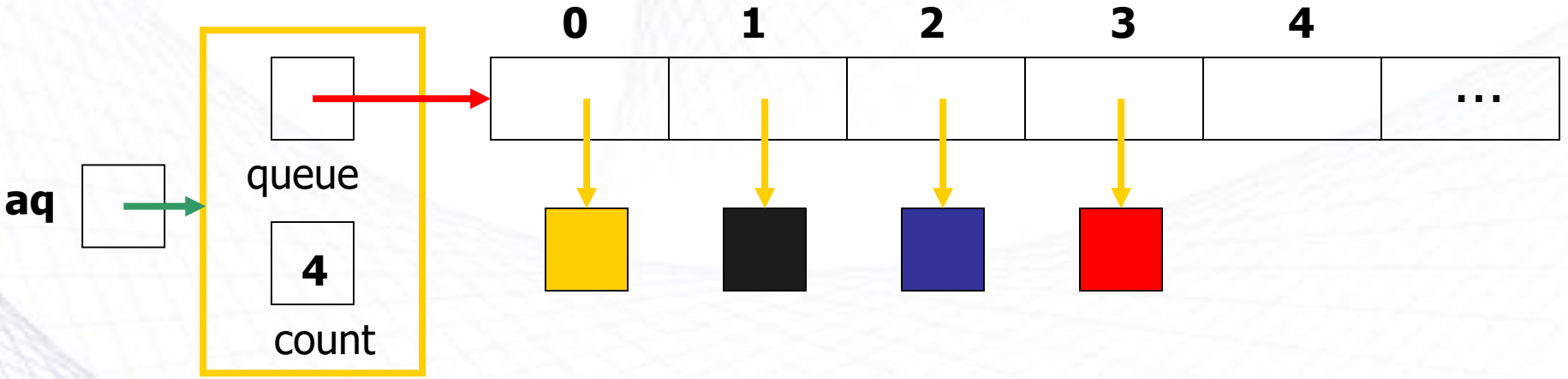
Queue After Adding an Element

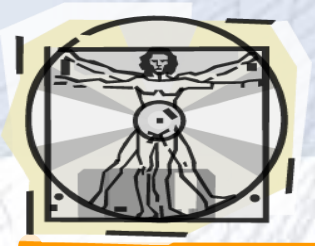
The element is added at the array location given by the value of count and then count is increased by 1.



Queue After Removing an Element

Element  **is removed from array location 0, remaining elements are shifted forward one position in the array, and then count is decremented.**

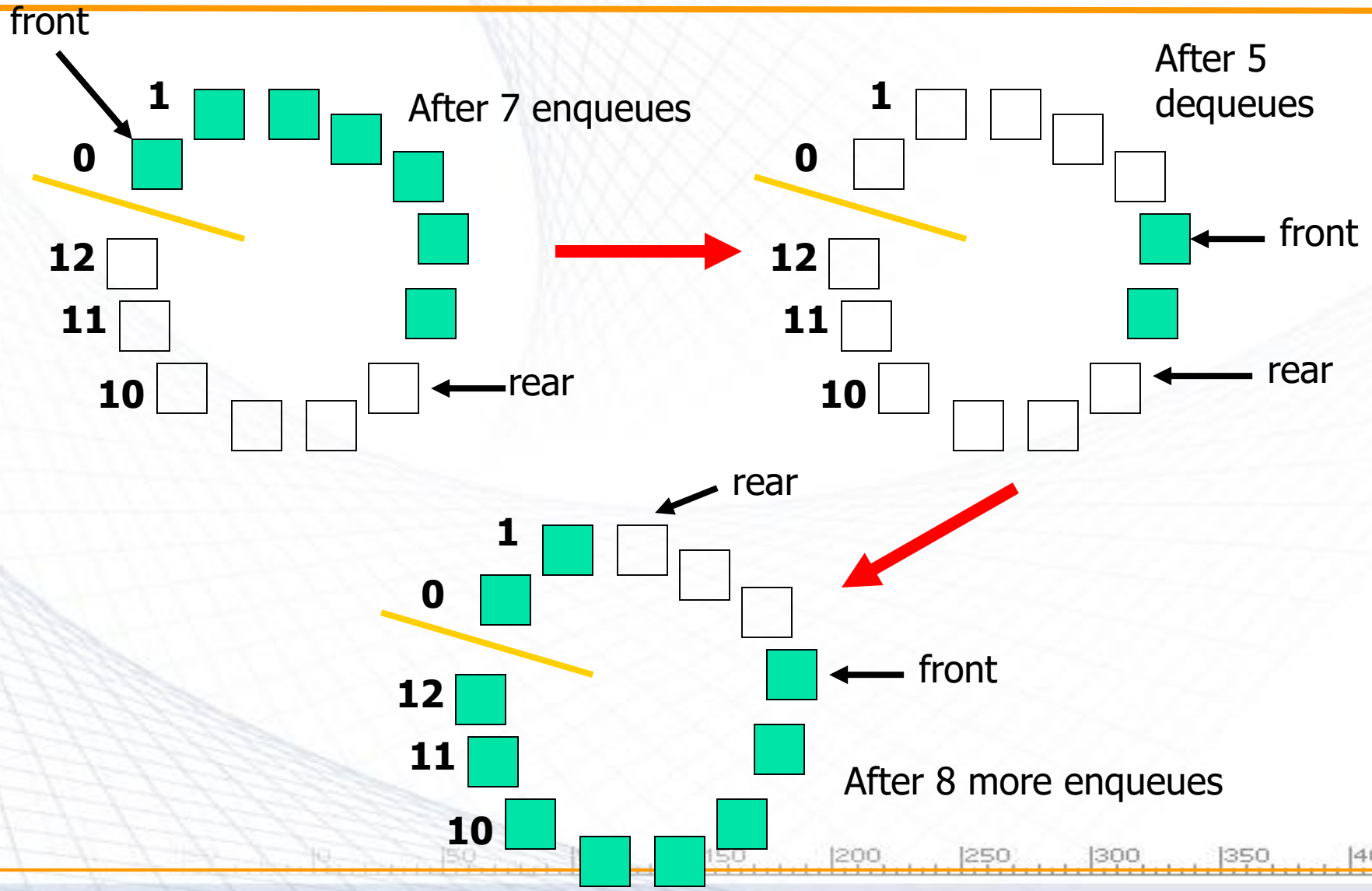




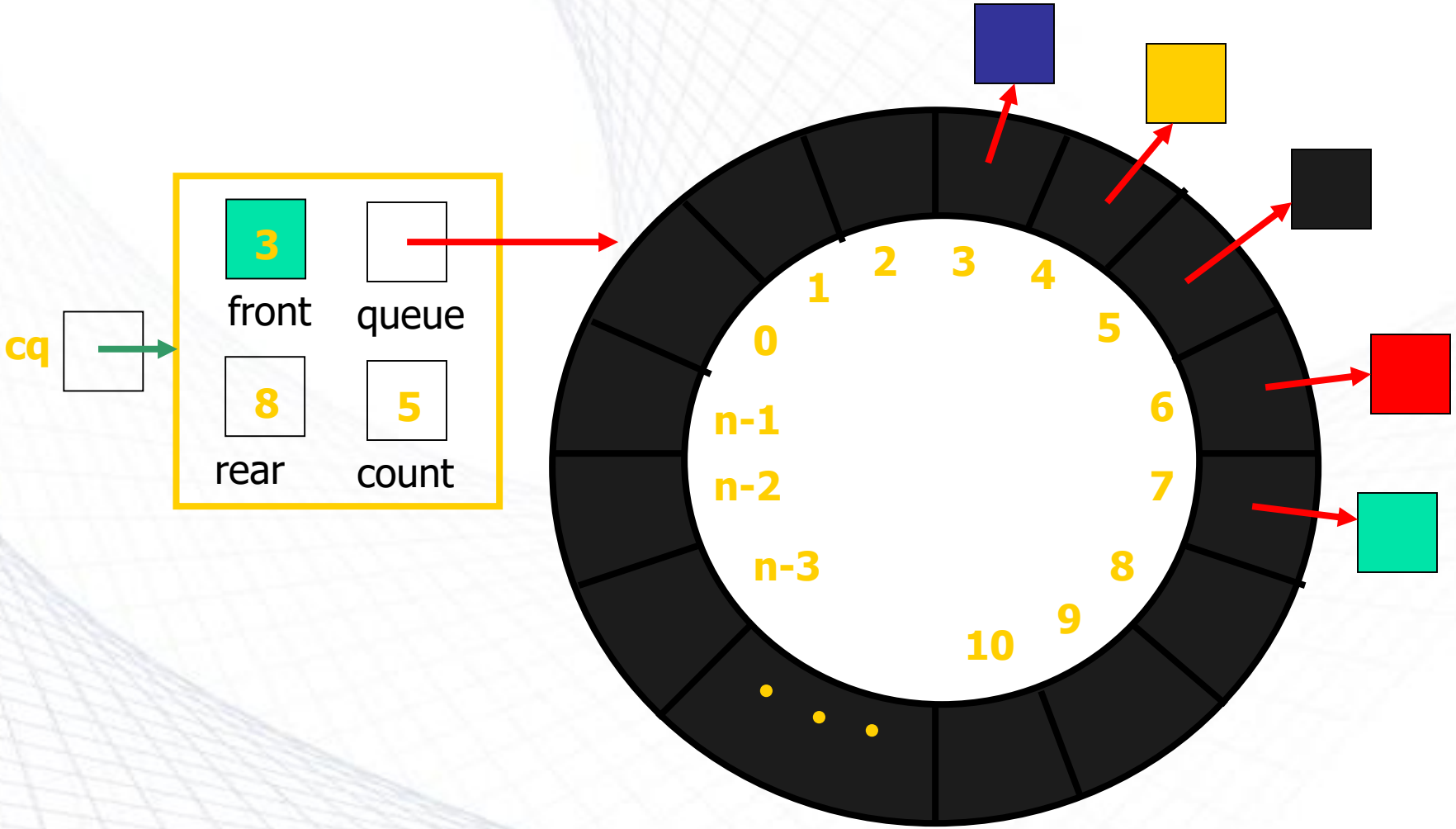
Second Approach: Queue as a Circular Array

- If we don't fix one end of the queue at index 0, we won't have to shift elements
- ***Circular array*** is an array that conceptually loops around on itself
  - The last index is thought to "***precede***" index 0
  - In an array whose last index is **n**, the location "***before***" index **0** is index **n**; the location "***after***" index **n** is index **0**
- Need to keep track of where the ***front*** as well as the ***rear*** of the queue are at any given time

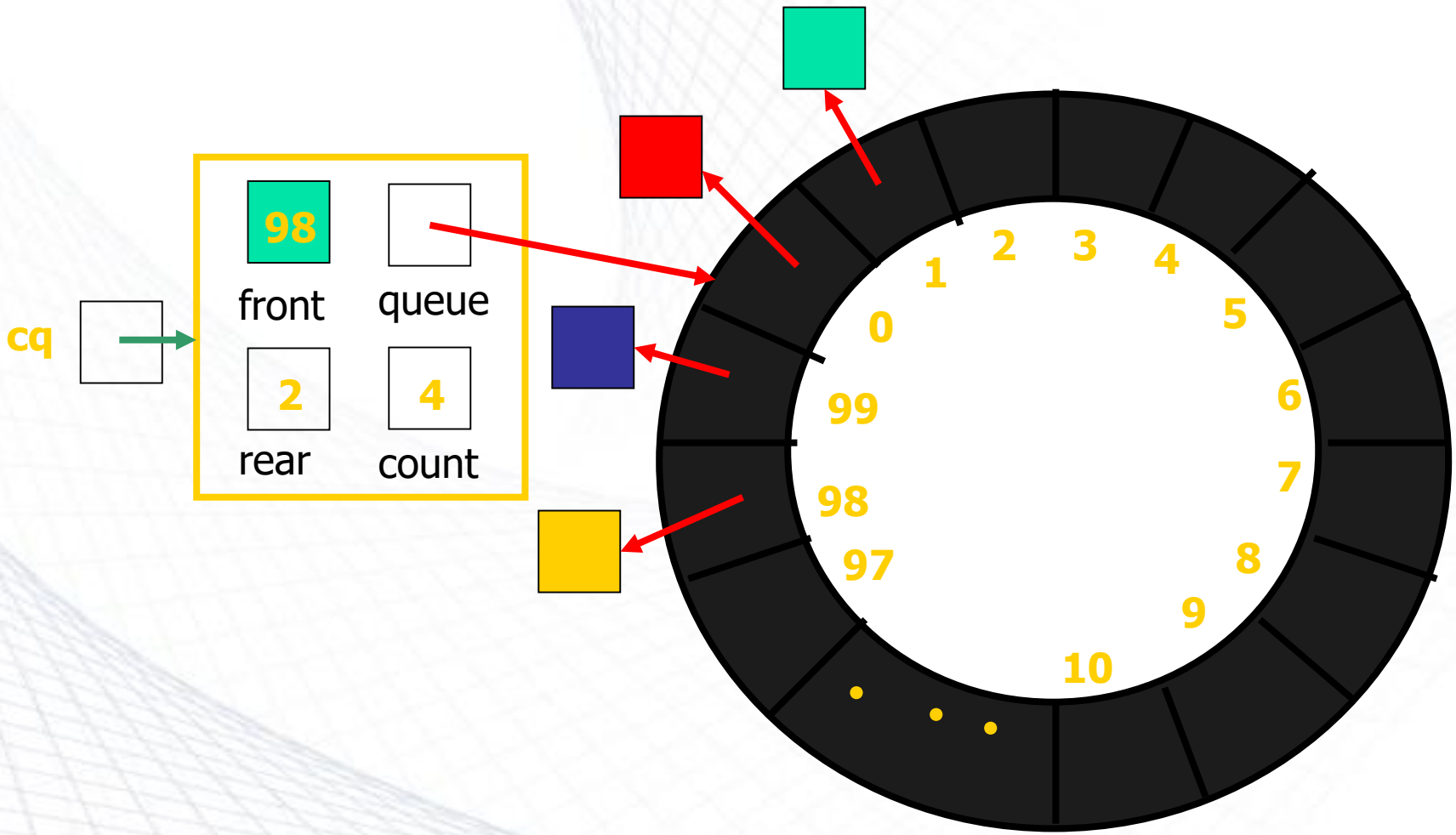
Conceptual Example of a Circular Queue

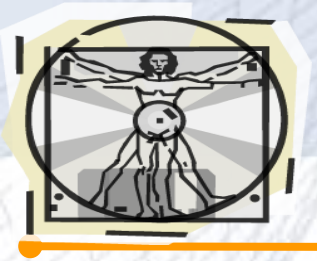


Circular Array Implementation of a Queue

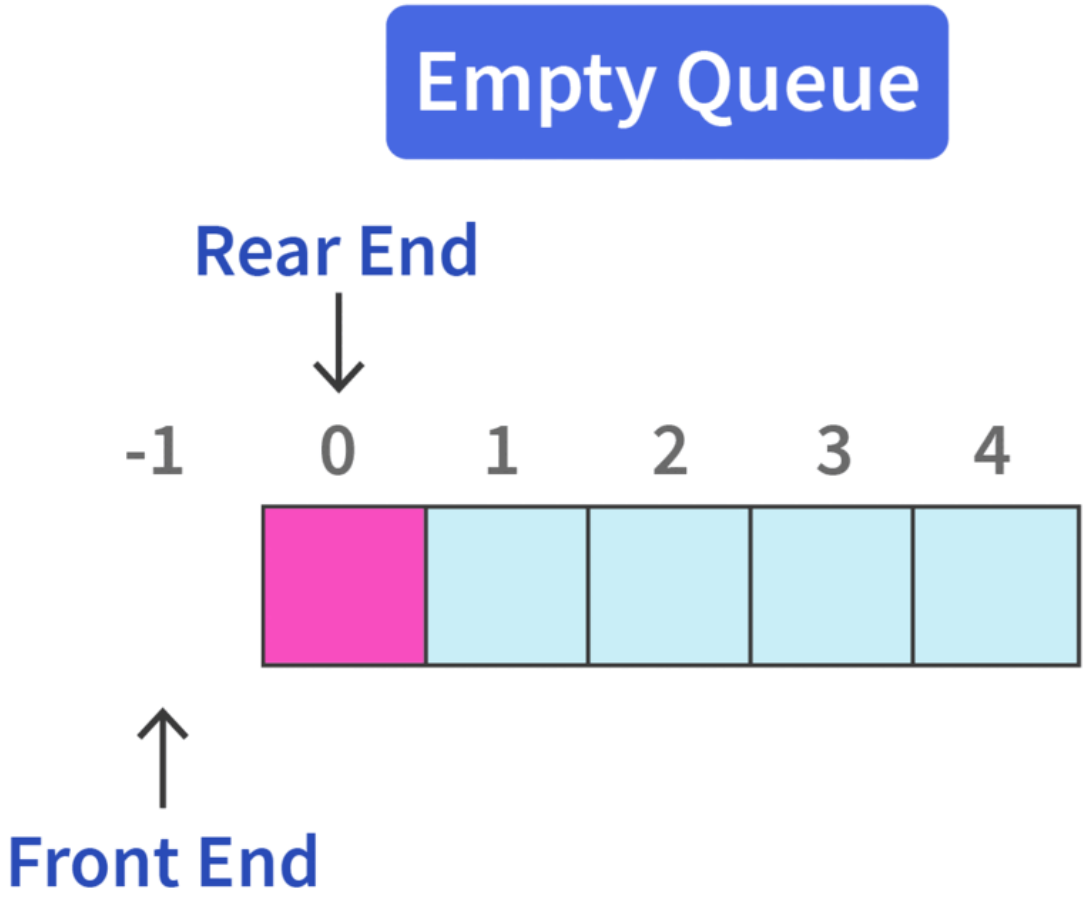


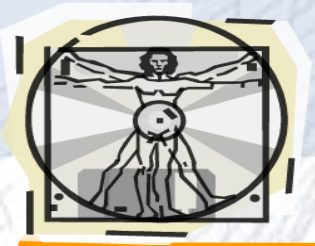
A Queue Straddling the End of a Circular Array





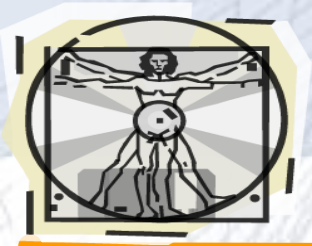
Animation





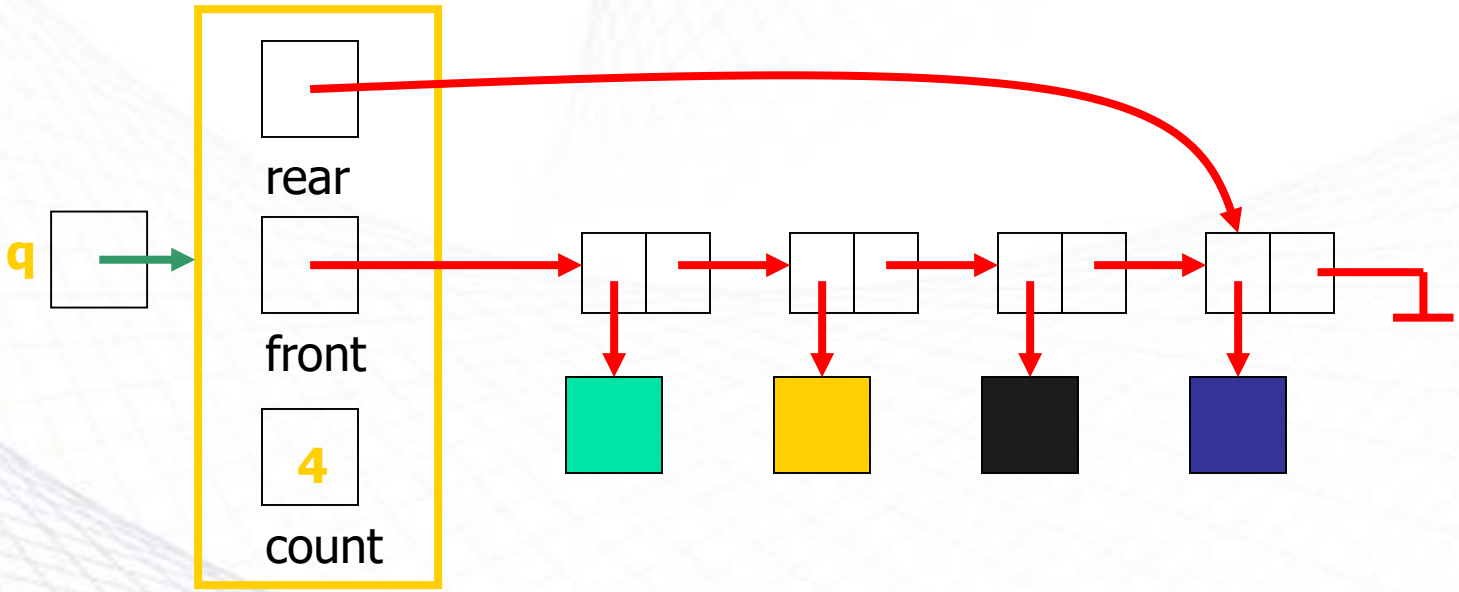
Queue Impl. using a Linked List

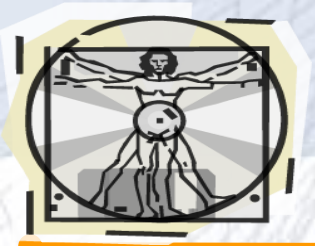
- Internally, the queue is represented as a ***linked list of nodes***, with each node containing a data element
- We need *two* pointers for the linked list
  - A pointer to the beginning of the linked list (***front*** of queue)
  - A pointer to the end of the linked list (***rear*** of queue)
- We will also have a ***count*** of the number of items in the queue



Linked Implementation of a Queue

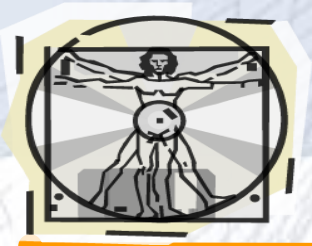
A queue q containing four elements





Discussion

- What if the queue is empty?
- What if there is only 1 element?



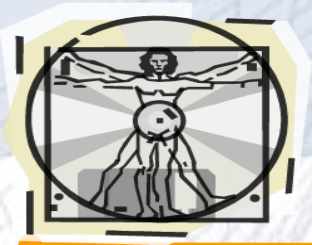
Generic Queue

By default you can put any Object into a Queue, but from Java 5, Java Generics makes it possible to limit the types of object you can insert into a Queue. Here is an example:

- Queue<MyObject> queue = new LinkedList<MyObject>();

This Queue can now only have MyObject instances inserted into it. You can then access and iterate its elements without casting them. Here is how it looks:

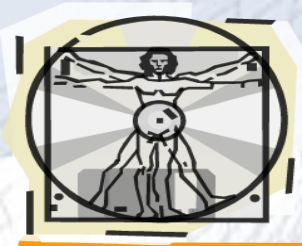
- Queue<MyObject> queue = new LinkedList<MyObject>();



Add Element to Queue

The Java Queue interface contains two methods you can use to add elements to a Queue. These methods are the add() method and the offer() method. These two methods add an element to the end of the Queue. The add() and offer() methods differ in how the behave if the Queue is full, so no more elements can be added. The add() method throws an exception in that case, whereas the offer() method just returns false. Here are two examples of adding elements to a Java Queue via its add() and offer() methods:

- Queue<String> queue = new LinkedList<>();
- queue.add("element 1");
- queue.offer("element 2");

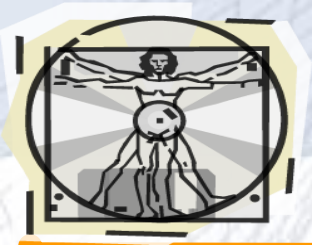


Getting an Element

- MyObject myObject = queue.remove();

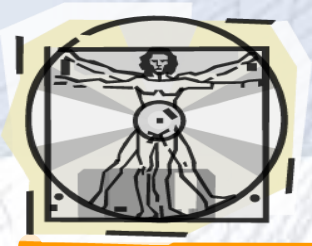
```
for(MyObject anObject : queue){
  //do someting to anObject...
}
```

Notice how no cast is needed on the first line, and how the for-each loop can cast each element of the Queue directly to a MyObject instance. This is possible because the Queue was created with MyObject as generic type.



Peek at the Queue

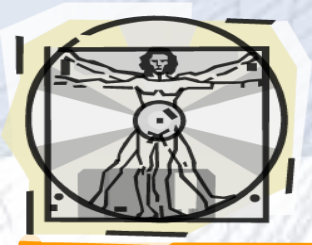
- You can peek at the element at the head of a Queue without taking the element out of the Queue. This is done via the Queue element() or peek() methods.
- The element() method returns the first element in the Queue. If the Queue is empty, the element() method throws a NoSuchElementException. Here is an example of peeking at the first element of a Java Queue using the element() /peek() method:
  - Queue<String> queue = new LinkedList<>();
  - queue.add("element 1");
  - queue.add("element 2");
  - queue.add("element 3");
  - String firstElement = queue.peek();



Get Queue Size

- You can read the number of elements stored in a Java Queue via its size() method. Here is an example of obtaining the size of a Java Queue via its size() method:

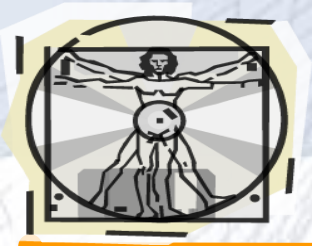
  - Queue<String> queue = new LinkedList<>();
  - queue.add("element 1");
  - queue.add("element 2");
  - queue.add("element 3");
  - int size = queue.size();



Check if Queue Contains Element

- You can check if a Java Queue contains a certain element via its contains() method. The contains() method will return true if the Queue contains the given element, and false if not. The contains() method is actually inherited from the Collection interface, but in practice that doesn't matter. Here are two examples of checking if a Java Queue contains a given element:

  - Queue<String> queue = new LinkedList<>();
  - queue.add("Mazda");
  - boolean containsMazda = queue.contains("Mazda");
  - boolean containsHonda = queue.contains("Honda");

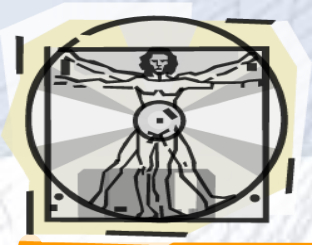


Iterate All Elements in Queue

You can also iterate all elements of a Java Queue, instead of just processing one at a time. Here is an example of iterating all elements in a Java Queue:

```
//access via Iterator
    Iterator<String> iterator = queue.iterator();
    while(iterator.hasNext(){
      String element = iterator.next();
    }

//access via new for-loop
    for(String element : queue) {
        //do something with each element
    }
```

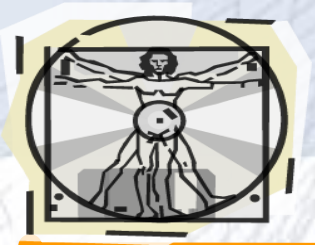


Example

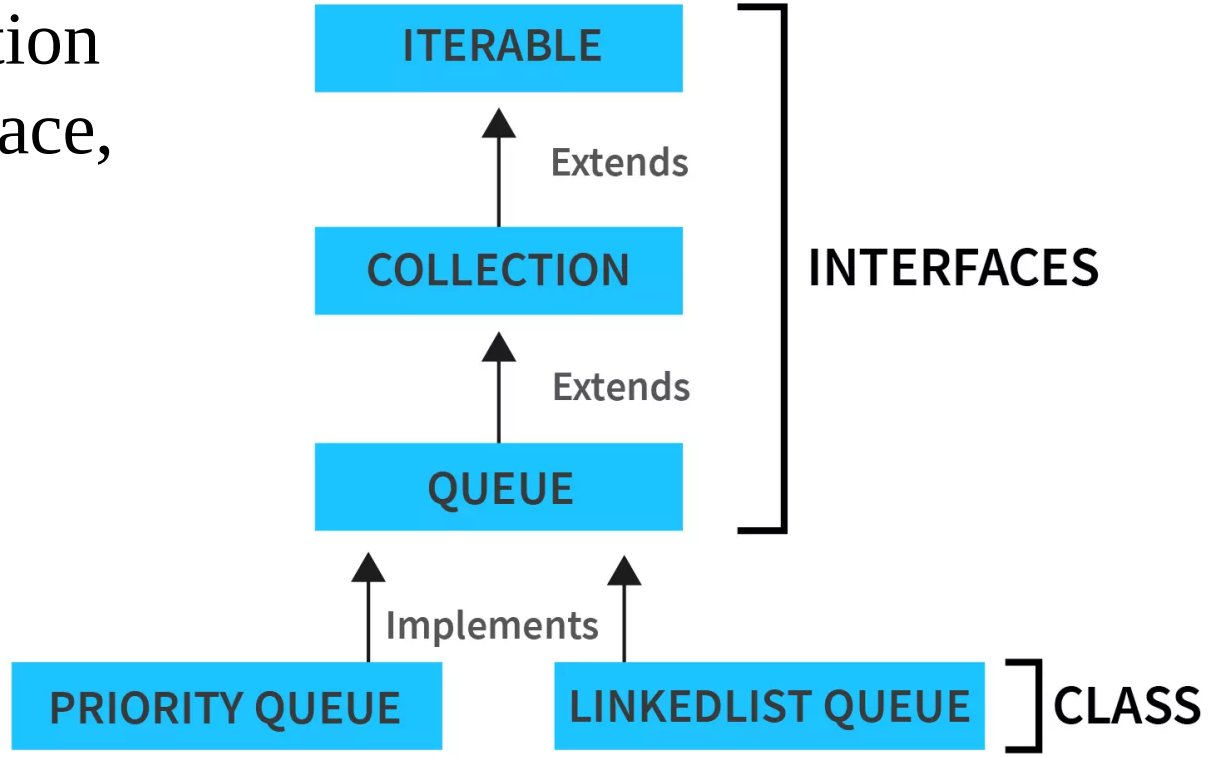
```
public static void main(String[] args) {
    Queue<Integer> q = new LinkedList<>();
    q.add(6);        q.add(1);                q.add(8);
    q.add(4);        q.add(7);
    System.out.println("The queue is: " + q);
    int num1 = q.remove();
    System.out.println("The element deleted from the head is: " + num1);
    System.out.println("The queue after deletion is: " + q);
    int head = q.peek();
    System.out.println("The head of the queue is: " + head);
    int size = q.size();
    System.out.println("The size of the queue is: " + size);
```

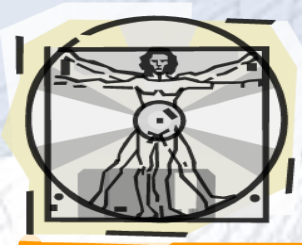
Output

```
The queue is: [6, 1, 8, 4, 7]
The element deleted from the head is: 6
The queue after deletion is: [1, 8, 4, 7]
The head of the queue is: 1
The size of the queue is: 4
```



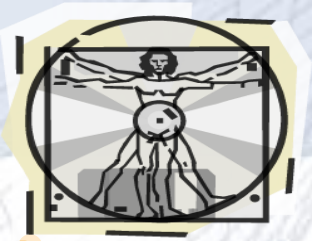
- In Java, the Queue interface is present in java.util package and extends Collection Framework. Since Queue is an Interface, queue needs a concrete class for the declaration such as LinkedList, PriorityQueue, etc.





Methods

- Queue Interface in java.util.Queue has additional methods for enqueue(), dequeue() and first(). The methods in the second column throws Exceptions. add(e) throws exception if the Queue size if full and remove() and element() throws Exception if Queue is empty.
- These additional methods also include methods that don't throw any exception in any case and just return null. These are offer(e), poll(), peek().
- size() and empty() methods are same for all.



Stack vs Queue

STACK	QUEUE
The stack is a linear data structure in which insertion and deletion of elements are done by only one end.	The Queue is a linear data structure in which insertion and deletion are done from two different ends i.e., rear and front ends respectively.
Stack has only one end name as **TOP**	The queue has two ends name as **REAR** and **FRONT**.
Stack follows **LIFO** principal	Queue follows **FIFO** principal
Stack is used for expression conversion and evaluation, handling recursive function calls, and checking the well-formedness of parenthesis.	The queue is used for prioritizing jobs in operating systems, categorizing data, and for simulations.