DOCUMENTATION

**Group Name: Trojan Horses**

Haritha Thaipparambil Hari (245205)

Aysha Cherupilly Rasheed (245254)

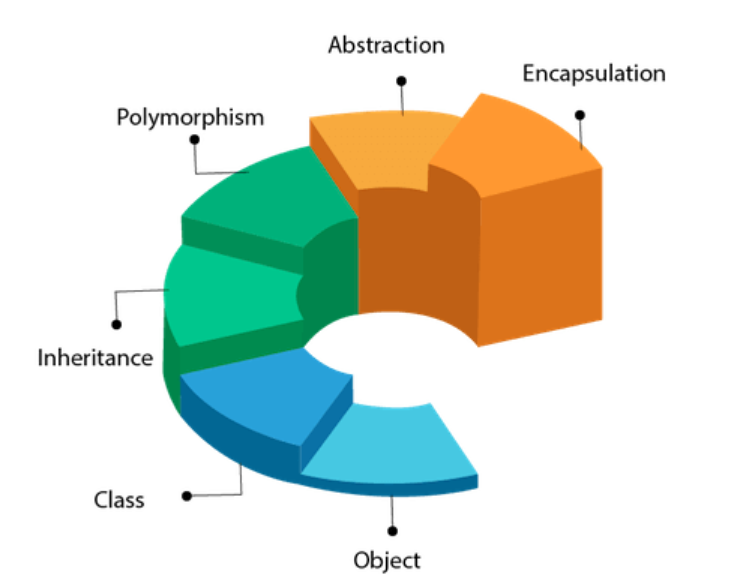
Ameena Kallepuram Abubacker (245255)

Aiswarya Kinattukara Kalam (245182)

**OOPS CONCEPTS IN JAVA**

**Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

* [Object](https://www.javatpoint.com/object-and-class-in-java)
* Class
* [Inheritance](https://www.javatpoint.com/inheritance-in-java)
* [Polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java)
* [Abstraction](https://www.javatpoint.com/abstract-class-in-java)
* [Encapsulation](https://www.javatpoint.com/encapsulation)



**Object**

Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

**Example:** A dog is an object because it has states like color, name, breed, etc. as well as behaviours like wagging the tail, barking, eating, etc.



## Class

Collection of objects is called class. It is a logical entity.

A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

**Inheritance in Java:**

Inheritance in Java is a mechanism in which one object acquires   
all the properties and behaviours of a parent object. Inheritance   
represents the IS-A relationship which is also known as a   
parent-child relationship. We use inheritance for Method Overriding.  
So, runtime polymorphism can be achieved and also for Code Reusability.

**Types of inheritance :**

There are five types of inheritance.

1. Single Inheritance - A single subclass extends from a single superclass.

2. Multilevel Inheritance - A subclass extends from a   
superclass and then the same subclass acts as a superclass   
for another class.

3. Hierarchical Inheritance - multiple subclasses extend from a single superclass.

4. Multiple Inheritance - A single subclass extends from multiple superclasses.

5. Hybrid Inheritance - Hybrid inheritance is a combination of two or more types  
of inheritance.

**The syntax of Java Inheritance:**

class Subclass-name extends Superclass-name   
{   
//methods and fields   
}

The extends keyword indicates that you are making a new class that   
derives from an existing class. In the terminology of Java, a class which   
is inherited is called a parent or superclass, and the new class is called   
child or subclass.

**Example of Inheritance**

class Animal {

// field and method of the parent class  
String name;  
public void eat() {  
System.out.println("I can eat");  
}  
}

// inherit from Animal  
class Dog extends Animal {

// new method in subclass  
public void display() {  
System.out.println("My name is " + name);  
}  
}

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

// create an object of the subclass  
Dog labrador = new Dog();

// access field of superclass  
labrador.name = "Rohu";  
labrador.display();

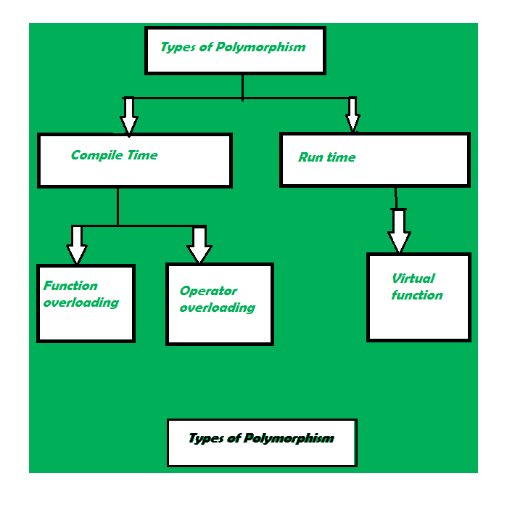
// call method of superclass  
// using object of subclass  
labrador.eat();

}  
}

Output is :  
My name is Rohu  
I can eat

Here we have derived a subclass Dog from superclass Animal.  
labrador is an object of Dog. However, name and eat() are   
the members of the Animal class. Since Dog inherits the field   
and method from Animal, we are able to access the field and   
method using the object of the Dog.

**Polymorphism in Java**

  
  
The word “poly” means many and “morphs” means forms, So it means many forms.  
In simple words, we can define polymorphism as the ability of a message to be   
displayed in more than one form. Polymorphism is the ability of an object to   
take on many forms. The most common use of polymorphism in OOP occurs when a   
parent class reference is used to refer to a child class object.  
Any Java object that can pass more than one IS-A test is considered to be   
polymorphic. In Java, all Java objects are polymorphic since any object   
will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is   
through a reference variable.

**Example**  
Polymorphism can be demonstrated by various examples, such as a polygon class   
that can render different shapes, an animal class that can have different eating  
behaviors, or a bank account class that can have different interest rates.

class Polygon {

// method to render a shape  
public void render() {  
System.out.println("Rendering Polygon...");  
}  
}

class Square extends Polygon {

// renders Square  
public void render() {  
System.out.println("Rendering Square...");  
}  
}

class Circle extends Polygon {

// renders circle  
public void render() {  
System.out.println("Rendering Circle...");  
}  
}

class Main {  
public static void main(String[] args) {  
  
// create an object of Square  
Square s1 = new Square();  
s1.render();

// create an object of Circle  
Circle c1 = new Circle();  
c1.render();  
}  
}

In Java polymorphism is mainly divided into two types:  
compile-time polymorphism and runtime polymorphism,   
which differ in when and how they are resolved.

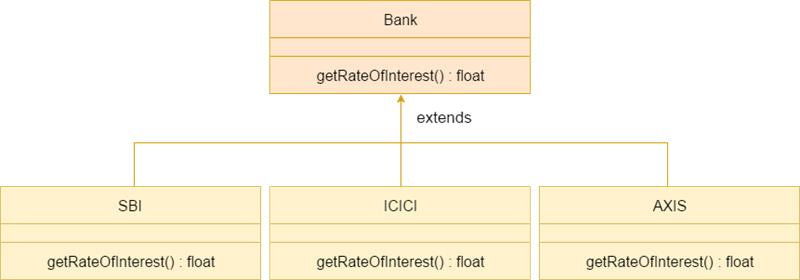
Compile-time polymorphism, also known as static polymorphism, occurs when the   
compiler determines the method to be invoked based on the reference type and   
arguments. It is achieved by method overloading and operator overloading.

Runtime polymorphism, also known as dynamic polymorphism, occurs when the   
method to be invoked is determined by the actual object type at runtime.   
It is achieved by method overriding and inheritance.

RUN TIME POLYMORPHISM

## Run time polymorphism is implemented through **Method overriding.**Whereas, Compile Time polymorphism is implemented through **Method overloading** and **Operator overloading**. Java Runtime Polymorphism Example: Bank

Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.

[](https://as-prod.asyncgw.teams.microsoft.com/v1/objects/0-sa-d8-6128c7eb21204155e894e72f7a44cfef/views/imgo)

#### Note: This example is also given in method overriding but there was no upcasting.

class Bank{

float getRateOfInterest(){return 0;}

}

class SBI extends Bank{

float getRateOfInterest(){return 8.4f;}

}

class ICICI extends Bank{

float getRateOfInterest(){return 7.3f;}

}

class AXIS extends Bank{

float getRateOfInterest(){return 9.7f;}

}

class TestPolymorphism{

public static void main(String args[]){

Bank b;

b=new SBI();

System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());

b=new ICICI();

System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());

b=new AXIS();

System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());

}

}

Output:

SBI Rate of Interest: 8.4

ICICI Rate of Interest: 7.3

AXIS Rate of Interest: 9.7

## **COMPILE-TIME POLYMORPHISM**

Compile-time polymorphism is also known as static polymorphism or early binding. Compile-time polymorphism is a polymorphism that is resolved during the compilation process. Overloading of methods is called through the reference variable of a class. Compile-time polymorphism is achieved by **method overloading** and **operator overloading.**

### **1. Method overloading**

We can have one or more methods with the same name that are solely distinguishable by argument numbers, type, or order.

Method Overloading occurs when a class has many methods with the same name but different parameters. Two or more methods may have the same name if they have other numbers of parameters, different data types, or different numbers of parameters and different data types.

void gfg() { ... }

void gfg(int num1 ) { ... }

void gfg(float num1) { ... }

void gfg(int num1 , float num2 ) { ... }

#### **a). Method overloading by changing the number of parameters**

In this type, Method overloading is done by overloading methods in the function call with a varied number of parameters

**Example:**

show( char a )

show( char a ,char b )

In the given example, the first show method has one parameter, and the second show method has two methods. When a function is called, the compiler looks at the number of parameters and decides how to resolve the method call.

Java

|  |
| --- |
| // Java program to demonstrate the working of method// overloading by changing the number of parameters **public** **class** MethodOverloading  {  // 1 parameter **void** show(**int** num1)  {  System.out.println("number 1 : " + num1);  } // 2 parameter **void** show(**int** num1, **int** num2)  {  System.out.println("number 1 : " + num1 + " number 2 : " + num2);  }  **public** **static** **void** main(String[] args) {  MethodOverloading obj = **new** MethodOverloading();  // 1st show function obj.show(3);  // 2nd show function obj.show(4, 5); }} |

**Output**

number 1 : 3

number 1 : 4 number 2 : 5

In the above example, we implement method overloading by changing several parameters. We have created two methods, show(int num1 ) and show(int num1, int num2 ). In the show(int num1) method display, one number and the void show(int num1, int num2 ) display two numbers

#### **(b). Method overloading by changing Datatype of parameter**

In this type, Method overloading is done by overloading methods in the function call with different types of parameters

**Example:**

show( float a float b)

show( int a, int b )

In the above example, the first show method has two float parameters, and the second show method has two int parameters. When a function is called, the compiler looks at the data type of input parameters and decides how to resolve the method call.

**Program:**

// Java program to demonstrate the working of method// overloading by changing the Datatype of parameter

public class MethodOverloading {

// arguments of this function are of integer type

static void show(int a,int b)

{

System.out.println("This is integer function ");

}

// argument of this function are of float type

static void show(double a,double b)

{

System.out.println("This is double function ");

}

public static void main(String[] args) {

// 1st show function show(1,2);

// 2nd show function show(1.2,2.4); }

}  
Output  
This is integer function   
This is double function

In the above example, we changed the data type of the parameters of both functions. In the first show() function datatype of the parameter is int. After giving integer type input, the output will be ‘ This is integer function.’ In the second show() function datatype of a parameter is double. After giving double type input, the output would be ‘This is double function.’

(c). By changing the sequence of parameters   
In this type, overloading is dependent on the sequence of the parameters

Example:

show(int a, float b )   
show( float a, int b )  
Here in this example, The parameters int and float are used in the first declaration. The parameters are int and float in the second declaration, but their order in the parameter list is different.

// Java program to demonstrate the working of method// overloading by changing the sequence of parameters

public class MethodOverloading

{ // arguments of this function are of int and char type static

void show(int a,char ch)

{

System.out.println("integer : " + a +" and character : " + ch);

}

// argument of this function are of char and int type static

void show(char ch,int a)

{

System.out.println("character : " + ch +" and integer : " + a);

}

public static void main(String[] args) {

// 1st show function show(6,'G');

// 2nd show function show('G',7); }}

Output

integer : 6 and character : G  
character : G and integer : 7

In the above example, in the first show, function parameters are int and char, and in the second shoe, function parameters are char, and int. changed the sequence of data type.

Invalid cases of method overloading

Method overloading does not allow changing the return type of method( function ); it occurs ambiguity.

Examples

int sum(int, int);  
String sum(int, int);  
Because the arguments are matching, the code above will not compile. Both methods have the same amount of data types and the same sequence of data types in the parameters.

Operator Overloading   
An operator is said to be overloaded if it can be used to perform more than one function. Operator overloading is an overloading method in which an existing operator is given a new meaning. In Java, the + operator is overloaded. Java, on the other hand, does not allow for user-defined operator overloading. To add integers, the + operator can be employed as an arithmetic addition operator. It can also be used to join strings together.

// Java program to demonstrate the// working of operator overloading

public class GFG { // function for adding two integers

void add(int a,int b)

{

int sum = a + b;

System.out.println(" Addition of two integer :" + sum);

}

// function for concatenating two strings

void add(String s1, String s2)

{

String con\_str = s1 + s2; System.out.println("Concatenated strings :" + con\_str);

}

public static void main(String args[]) {

GFG obj =new GFG();

// addition of two numbers obj.add(10,10);

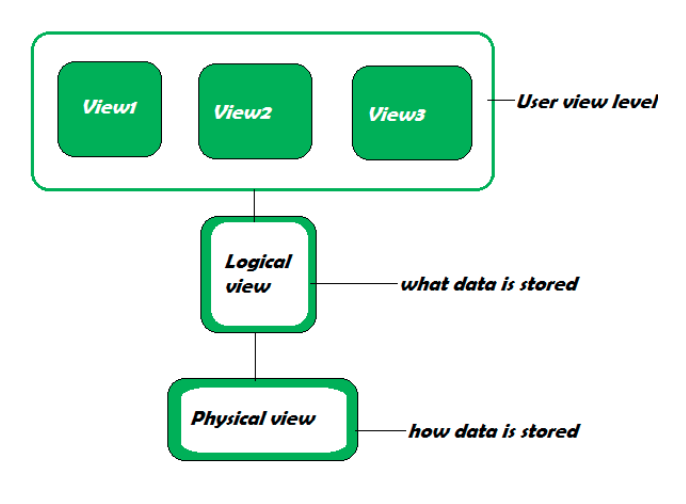
// concatenation of two string obj.add("Operator "," overloading ");

}}

Output  
Addition of two integer :20  
Concatenated strings :Operator overloading   
In the above example, The ‘+’ operator has been overloaded. When we send two numbers to the overloaded method, we get a sum of two integers, and when we pass two Strings, we get the concatenated text.

Advantages of Compile-time Polymorphism:  
It improves code clarity and allows for the use of a single name for similar procedures.  
It has a faster execution time since it is discovered early in the compilation process.  
The only disadvantage of compile-time polymorphism is that it doesn’t include inheritance.

**ABSTRACTION IN JAVA**



Abstraction is a feature of OOPs. The feature allows us to hide the implementation detail from the user and shows only the functionality of the programming to the user. Because the user is not interested to know the implementation. The best example of abstraction is a car. When we derive a car, we do not know how is the car moving or how internal components are working? But we know how to derive a car. It means it is not necessary to know how the car is working, but it is important how to derive a car. The same is an abstraction.

The same principle (as we have explained in the above example) also applied in Java programming and any OOPs. In the language of programming, the code implementation is hidden from the user and only the necessary functionality is shown or provided to the user. We can achieve abstraction in two ways:

1.Using Abstract Class

Abstract classes are the same as normal Java classes the difference is only that an abstract class uses abstract keyword while the normal Java class does not use. We use the abstract keyword before the class name to declare the class as abstract. Using an abstract class, we can achieve 0-100% abstraction. An abstract class contains abstract methods as well as concrete methods. If we want to use an abstract class, we have to inherit it from the base class. If the class does not have the implementation of all the methods of the interface, we should declare the class as abstract. It provides complete abstraction. It means that fields are public static and final by default and methods are empty. Example of an Abstract Class: Main.java   
//abstract class   
abstract class Demo   
{   
//abstract method   
abstract void display();   
}   
//extends the abstract class   
public class MainClass extends Demo   
{   
//defining the body of the method of the abstract class   
void display()   
{   
System.out.println("Abstract method called.");   
}   
public static void main(String[] args)   
{   
MainClass obj = new MainClass ();   
//invoking abstract method

2.Using Interface

In Java, an interface is similar to Java classes. The difference is only that an interface contains empty methods (methods that do not have method implementation) and variables. In other words, it is a collection of abstract methods (the method that does not have a method body) and static constants. The important point about an interface is that each method is public and abstract and does not contain any constructor. Along with the abstraction, it also helps to achieve multiple inheritance. The implementation of these methods provided by the clients when they implement the interface. Using interface, we can achieve 100% abstraction. Separating interface from implementation is one way to achieve abstraction. The Collection framework is an excellent example of it.

Features of Interface:We can achieve total abstraction.We can use multiple interfaces in a class that leads to multiple inheritance.It also helps to achieve loose coupling.To use an interface in a class, Java provides a keyword called implements. We provide the necessary implementation of the method that we have declared in the interface.

Let's see an example of an interface.

Car.java

interface CarStart   
{   
void start();   
}   
interface CarStop   
{   
void stop();   
}   
public class Car implements CarStart, CarStop   
{   
public void start()   
{   
System.out.println("The car engine has been started.");   
}   
public void stop()   
{   
System.out.println("The car engine has been stopped.");   
}   
public static void main(String args[])   
{   
Car c = new Car();   
c.start();   
c.stop();   
}   
}   
Output: The car engine has been started. The car engine has been stopped.

**ENCAPSULATION**

It is defined as the wrapping up of data under a single unit. It is the mechanism that binds together the code and the data it manipulates. Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

* Technically, in encapsulation, the variables or the data in a class is hidden from any other class and can be accessed only through any member function of the class in which they are declared.
* In encapsulation, the data in a class is hidden from other classes, which is similar to what **data-hiding** does. So, the terms “encapsulation” and “data-hiding” are used interchangeably.
* Encapsulation can be achieved by declaring all the variables in a class as private and writing public methods in the class to set and get the values of the variables.

Example:

class Area {

// fields to calculate area

int length;

int breadth;

// constructor to initialize values

Area(int length, int breadth) {

this.length = length;

this.breadth = breadth;

}

// method to calculate area

public void getArea() {

int area = length \* breadth;

System.out.println("Area: " + area);

}

}

class Main {

public static void main(String[] args) {

// create object of Area

// pass value of length and breadth

Area rectangle = new Area(5, 6);

rectangle.getArea();

}

}

Output:

Area: 30

In the above example, we have created a class named Area. The main purpose of this class is to calculate the area.

To calculate an area, we need two variables: length and breadth and a method: getArea(). Hence, we bundled these fields and methods inside a single class.

It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.

The encapsulate class is **easy to test**. So, it is better for unit testing.

The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

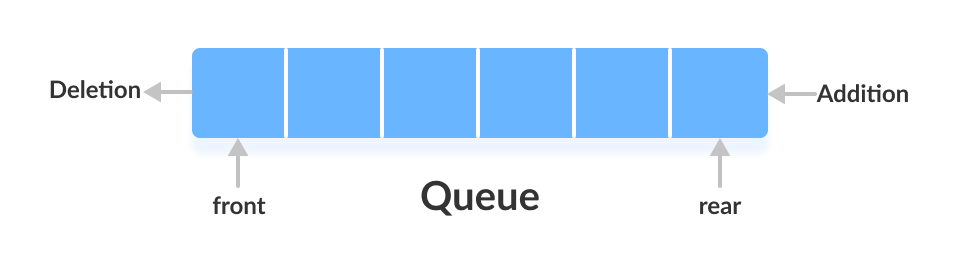
**QUEUE**

A queue is another kind of linear data structure that is used to store elements just like any other data structure but in a particular manner. In simple words, we can say that the queue is a type of data structure in the Java programming language that stores elements of the same kind. The components in a queue are stored in a FIFO (First In, First Out) behavior. There are two ends in the queue collection, i.e., front & rear. Queue has two ends that is front and rear.

The Queue is an interface in the [Java](https://www.javatpoint.com/java-tutorial) that belongs to [Java.util package](https://www.javatpoint.com/java-util-package). It also extends the [Collection interface](https://www.javatpoint.com/java-collection).

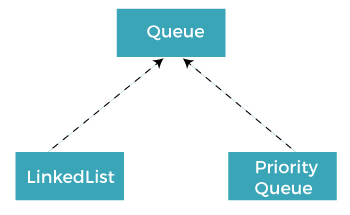
The generic representation of the Java Queue interface is shown below:

1. **public** **interface** Queue<T> **extends** Collection<T>



In Java programming language, there are two different classes which are used to implement the Queue interface. These classes are:

* [LinkedList](https://www.javatpoint.com/java-linkedlist)
* [PriorityQueue](https://www.javatpoint.com/java-priorityqueue)



## Characteristics of the Java Queue

The Java Queue can be considered as one of the most important data structures in the programming world. Java Queue is attractive because of its properties. The significant properties of the Java Queue data structure are given as follows:

* Java Queue obeys the FIFO (First In, First Out) manner. It indicates that elements are entered in the queue at the end and eliminated from the front.
* The Java Queue interface gives all the rules and processes of the Collection interface like inclusion, deletion, etc.
* There are two different classes that are used to implement the Queue interface. These classes are LinkedList and PriorityQueue.
* Other than these two, there is a class that is, Array Blocking Queue that is used to implement the Queue interface.
* There are two types of queues, Unbounded queues and Bounded queues. The Queues that are a part of the java.util package are known as the Unbounded queues and bounded queues are the queues that are present in java.util.concurrent package.
* The Deque or (double-ended queue) is also a type of queue that carries the inclusion and deletion of elements from both ends.
* The deque is also considered thread-safe.
* Blocking Queues are also one of the types of queues that are also thread-safe. The Blocking Queues are used to implement the producer-consumer queries.
* Blocking Queues do not support null elements. In Blocking queues, if any work similar to null values is tried, then the NullPointerException is also thrown.

## Implementation of Queue

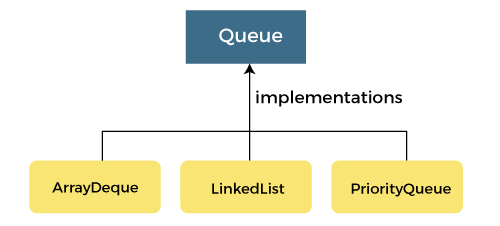
### **Classes used in implementation of Queue**

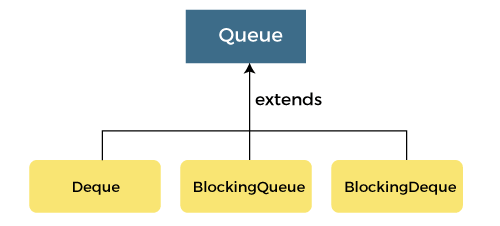
The classes that are used to implement the functionalities of the queue are given as follows:

* [ArrayDeque](https://www.programiz.com/java-programming/arraydeque)
* [LinkedList](https://www.programiz.com/java-programming/linkedlist)
* [PriorityQueue](https://www.programiz.com/java-programming/priorityqueue)

### **Interfaces used in implementation of Queue**

The Java interfaces are also used in the implementation of the Java queue. The interfaces that are used to implement the functionalities of the queue are given as follows:



* Deque
* Blocking Queue
* BlockingDeque

**Methods of queue**

add() - Inserts the specified element into the queue. If the task is successful, add() returns true, if not it throws an exception.

offer() - Inserts the specified element into the queue. If the task is successful, offer() returns true, if not it returns false.

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

peek() - Returns the head of the queue. Returns null if the queue is empty.

remove() - Returns and removes the head of the queue. Throws an exception if the queue is empty.

poll() - Returns and removes the head of the queue. Returns null if the queue is empty.

**Implementation of the Queue Interface**

1. Implementing the LinkedList Class

import java.util.Queue;

import java.util.LinkedList;

class Main {

public static void main(String[] args) {

// Creating Queue using the LinkedList class

Queue<Integer> numbers = new LinkedList<>();

// offer elements to the Queue

numbers.offer(1);

numbers.offer(2);

numbers.offer(3);

System.out.println("Queue: " + numbers);

// Access elements of the Queue

int accessedNumber = numbers.peek();

System.out.println("Accessed Element: " + accessedNumber);

// Remove elements from the Queue

int removedNumber = numbers.poll();

System.out.println("Removed Element: " + removedNumber);

System.out.println("Updated Queue: " + numbers);

}

}

Output:

Queue: [1, 2, 3]

Accessed Element: 1

Removed Element: 1

Updated Queue: [2, 3]

2. Implementing the PriorityQueue Class

import java.util.Queue;

import java.util.PriorityQueue;

class Main {

public static void main(String[] args) {

// Creating Queue using the PriorityQueue class

Queue<Integer> numbers = new PriorityQueue<>();

// offer elements to the Queue

numbers.offer(5);

numbers.offer(1);

numbers.offer(2);

System.out.println("Queue: " + numbers);

// Access elements of the Queue

int accessedNumber = numbers.peek();

System.out.println("Accessed Element: " + accessedNumber);

// Remove elements from the Queue

int removedNumber = numbers.poll();

System.out.println("Removed Element: " + removedNumber);

System.out.println("Updated Queue: " + numbers);

}

}

Output:

Queue: [1, 5, 2]

Accessed Element: 1

Removed Element: 1

Updated Queue: [2, 5]

**Applications of Queue:**

* **Multi programming:**Multi programming means when multiple programs are running in the main memory. It is essential to organize these multiple programs and these multiple programs are organized as queues.
* **Network:**In a network, a queue is used in devices such as a router or a switch. another application of a queue is a mail queue which is a directory that stores data and controls files for mail messages.
* **Job Scheduling:**The computer has a task to execute a particular number of jobs that are scheduled to be executed one after another. These jobs are assigned to the processor one by one which is organized using a queue.
* **Shared resources:**Queues are used as waiting lists for a single shared resource.

**Advantages of Queue:**

* A large amount of data can be managed efficiently with ease.
* Operations such as insertion and deletion can be performed with ease as it follows the first in first out rule.
* Queues are useful when a particular service is used by multiple consumers.
* Queues are fast in speed for data inter-process communication.
* Queues can be used in the implementation of other data structures.

**Disadvantages of Queue:**

* The operations such as insertion and deletion of elements from the middle are time consuming.
* Limited Space.
* In a classical queue, a new element can only be inserted when the existing elements are deleted from the queue.
* Searching an element takes O(N) time.
* Maximum size of a queue must be defined prior

Java Deque

A deque is a linear collection that supports insertion and deletion of elements

from both the ends. The name 'deque' is an abbreviation for double-ended queue.

A deque is a linear collection that supports insertion and deletion of elements from both the ends.

The name 'deque' is an abbreviation for double-ended queue.

There are no fixed limits on the deque for the number of elements they may contain. However, this interface

supports capacity restricted deques as well as the deques with no fixed size limits. There are various methods

which are provided to insert, remove and examine the elements.

This differs from the queue abstract data type or first in first out list (FIFO), where elements can only be added to

one end and removed from the other. This general data class has some possible sub-types:

An input-restricted deque is one where deletion can be made from both ends, but insertion can be made at one end only.

An output-restricted deque is one where insertion can be made at both ends, but deletion can be made from one end only.

Both the basic and most common list types in computing, queues and stacks can be considered specializations of deques,

and can be implemented using deques.

These methods typically exist in two forms: the one that throws an exception when the particular operation fails

and the other returns a special value which can be either null or false depending upon the operations.

The dynamic array approach uses a variant of a dynamic array that can grow from both ends, sometimes called array deques. These array deques have all the properties of a dynamic array, such as constant-time random access, good locality of reference, and inefficient insertion/removal in the middle, with the addition of amortized constant-time insertion/removal at both ends, instead of just one end. Three common implementations include:

Storing deque contents in a circular buffer, and only resizing when the buffer becomes full. This decreases the frequency of resizings.

Allocating deque contents from the center of the underlying array, and resizing the underlying array when either end is reached. This approach may require more frequent resizings and waste more space, particularly when elements are only inserted at one end.

Storing contents in multiple smaller arrays, allocating additional arrays at the beginning or end as needed. Indexing is implemented by keeping a dynamic array containing pointers to each of the smaller arrays.

There are no fixed limits on the deque for the number of elements they may

contain. However, this interface supports capacity restricted deques as well

as the deques with no fixed size limits. There are various methods which are

provided to insert, remove and examine the elements.

These methods typically exist in two forms: the one that throws an exception

when the particular operation fails and the other returns a special value

which can be either null or false depending upon the operations.

Syntax: The deque interface is declared as:

public interface Deque extends Queue

import java.util.ArrayDeque;

import java.util.Deque;

public class Example {

public static void main(String[] args) {

Deque<Integer> deque = new ArrayDeque<>();

deque.addFirst(1);

deque.addLast(2);

int first = deque.removeFirst();

int last = deque.removeLast();

System.out.println("First: " + first + ", Last: " + last);

}

}

Working of Deque

In a regular queue, elements are added from the rear and removed from the front. However, in a deque, we can insert and remove elements

from both front and rear.

Working of deque (double-ended queue) data structure

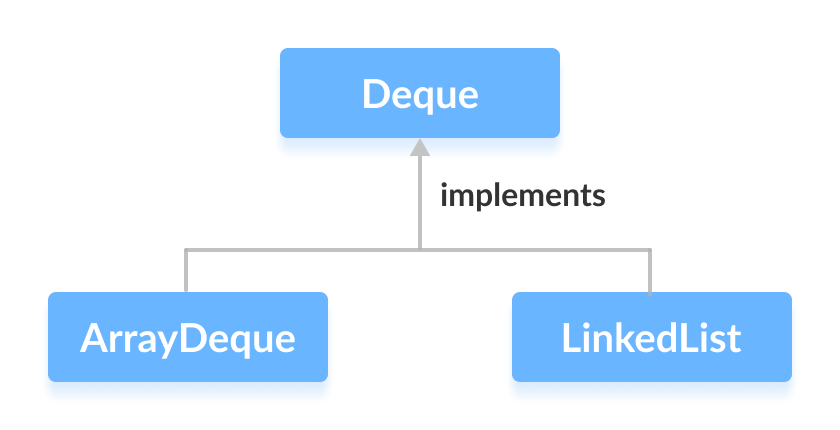


Classes that implement Deque

In order to use the functionalities of the Deque interface, we need to use classes that implement it:

ArrayDeque

LinkedList



How to use Deque?

In Java, we must import the java.util.Deque package to use Deque.

// Array implementation of Deque

Deque<String> animal1 = new ArrayDeque<>();

Deque as Stack Data Structure

The Stack class of the Java Collections framework provides the implementation of the stack.

However, it is recommended to use Deque as a stack instead of the Stack class. It is because methods of Stack are synchronized.

Here are the methods the Deque interface provides to implement stack:

push() - adds an element at the beginning of deque

pop() - removes an element from the beginning of deque

peek() - returns an element from the beginning of deque

// LinkedList implementation of Deque

Deque<String> animal2 = new LinkedList<>();

Here, we have created objects animal1 and animal2 of classes ArrayDeque and LinkedList, respectively. These objects can use the functionalities of the Deque

interface.

Applications of Deque: Since Deque supports both stack and queue operations, it can be used as both. The Deque data structure supports

clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications. Also, the problems where elements need to be

removed and or added to both ends can be efficiently solved using Deque. For example see the Maximum of all subarrays of size k problem.,

0-1 BFS, and Find the first circular tour that visits all petrol pumps. See the wiki page for another example of the A-Steal job scheduling

algorithm where Deque is used as deletions operation is required at both ends.

Methods

The java.util.Deque interface provides methods to perform the operations of double-ended queue in Java. Its implementation class is java.util.ArrayDeque.

**Methods**

add(E e) - This method is used to insert a specified element into the queue represented by the deque

addAll(Collection<? Extends E>c) - Adds all the elements in the specified collection at the end of the deque.

addFirst(E e) - Inserts the specified element at the front of the deque.

addLast(E e) - Inserts the specified element at the end of the deque.

contains(object o) - Returns true if the deque contains the specified element.

descendingIterator() - Returns an iterator over the elements in reverse sequential order.

element() - Retrieves the head of the queue represented by the deque.

getFirst() - Retrieves but does not remove the first element of the deque.

getLast() - Retrieves but does not remove the last element of the deque.

iterator() - Returns an iterator over the element in the deque in a proper sequence.

offer(E e) - Inserts the specified element into the deque, returning true upon success and false if no space is available.

offerFirst() - Inserts the specified element at the front of the deque unless it violates the capacity restriction.

offerLast() - Inserts the specified element at the end of the deque unless it violates the capacity restriction.

peek() - Retrieves but does not move the head of the queue represented by the deque or may return null if the deque is empty.

peekFirst() - Retrieves but does not move the first element of the deque or may return null if the deque is empty.

peekLast() - Retrieves but does not move the last element of the deque or may return null if the deque is empty.

poll() - Retrieves and remove the head of the queue represented by the deque or may return null if the deque is empty.

pollFirst() - Retrieves and remove the first element of the deque or may return null if the deque is empty.

pollLast() - Retrieves and remove the last element of the deque or may return null if the deque is empty.

pop() - Pops an element from the stack represented by the deque.

push() - Pushes an element onto the stack represented by the deque.

remove() - Retrieves and remove the head of the queue represented by the deque.

remove(Object o) - Removes the first occurrence of the specified element from the deque.

removeFirst() - Retrieves and remove the first element from the deque.

removeFirstOccurrence(Object o) -Remove the first occurrence of the element from the deque.

removeLast() - Retrieve and remove the last element from the deque.

removeLastOccurrence(Object o) - Remove the last occurrence of the element from the deque.

size() - Returns the total number of elements in the deque.

Java Deque add() Method :

The add() method of Deque interface is used to insert the specified element into the specified queue represented by the deque and returns true upon success and throws an IllegalStateException if there is no current space available.

Specified by:

add in interface Collection<E>

add in interface Queue<E>

Syntax:

boolean add(E e)

Parameter:

The above method consists of only one parameter:

The element 'e' that needs to be added.

Return:

The above method returns a Boolean value, i.e., true.

Java Deque addAll() Method

The addAll() method of Java Deque Interface is used to add all the elements in a specified collection at the end of the deque.

Specified by:

addAll in interface Collection<E>

Syntax:

boolean addAll(Collection< ? Extends E>c)

Parameter:

The above method consists of only one parameter:

The element 'c' that needs to be inserted.

Return:

The above method returns true if the given deque is changed.

Java Deque addFirst() Method

The addFirst() method of Java Deque Interface is used to insert the specified element at the front of the deque .

Syntax:

void addFirst(E e)

Parameter:

The above method consists of only one parameter:

The element 'e' that needs to be added.

Return:

NA

Throw:

IllegalStateException- If the elements cannot be added at the time due to capacity restriction.

ClassCastException- If the class of the specified element avoids it from being added to the deque.

NullPointerException- If the specified element is a null and the deque does not allow the null elements.

IllegalArgumentException- If some of the property of the given element avoids it from being added to the deque.

Java Deque addLast() Method

The addLast() method of Java Deque Interface is used to insert the specified element at the end of the deque.

Syntax:

void addLast(E e)

Parameter:

The above method consists of only one parameter:

The element 'e' that needs to be inserted.

Return:

NA

Throw:

IllegalStateException- If the elements cannot be added at the time due to capacity restriction.

ClassCastException- If the class of the specified element avoids it from being added to the deque.

NullPointerException- If the specified element is a null and the deque does not allow the null elements.

IllegalArgumentException- If some of the property of the given element avoids it from being added to the deque.

Java Deque contains() Method

The contains() method of Java Deque Interface returns true if the deque contains the specified element.

Specified by:

contains in interface Collection<E>

Syntax:

boolean contains(Object o)

Parameter:

The above method consists of only one parameter:

a.) The element 'o' which is already present in the deque and is tested.

Return:

The above method returns a Boolean value, i.e., true if the specified element is already present in the deque .

Throw:

ClassCastException- If the class of the given element is incompatible with the deque.

NullPointerException- If the specified element is a null and the deque does not allow the null elements.

Java Deque descendingIterator() Method

The descendingIterator() method of Java Deque Interface returns an iterator for the elements in the

specified deque in a reverse sequence. The elements will return in a sequential order from first(head) to the last(tail).

Syntax:

Iterator<E> descendingIterator()

Parameter:

NA

Return:

The above method returns an iterator for the elements in the specified deque in a reverse sequence.

Java Deque iterator() Method

The iterator() method of Java Deque Interface is used to return an iterator for the elements in the deque

in a proper sequence. The elements are returned in a order from first(head) to last(tail).

Specified by:

iterator in interface Collection<E>

iterator in interface Iterable<E>

Syntax:

Iterator <E> iterator()

Parameter:

NA

Return:

The above method returns an iterator for the elements in the given deque in a proper sequence.

Java Deque poll() Method

The poll() method of Java Deque Interface is used to retrieve and remove the head of the deque. On the other hand,

if the deque is empty, the method may return null. The above method is equivalent to pollFirst().

Specified by:

poll in interface Queue<E>

Syntax:

public E poll()

Parameter:

NA

Return:

The above method is used to return the first element of the given deque. Otherwise, the method may return a null

value if the deque is empty.

Some Practical Applications of Deque:

Applied as both stack and queue, as it supports both operations.

Storing a web browser’s history.

Storing a software Application's list of undo operation

Job Scheduling Algorithms.

Monotonic Deque :

It is deque which stores elements in strictly increasing order or in strictly decreasing order

To maintain monotonicity, we need to delete elements

For example – Consider monotonic(decreasing) deque dq = {5, 4, 2, 1}

Insert 3 into dq

So we need to delete elements till dq.back() < 3 to insert 3 into dq (2,1 are the deleted elements)

Resulting dq = {5, 4, 3}

Applications of monotonic deque :

It can be used to get next maximum in a subarray (sliding-window-maximum-of-all-subarrays-of-size-k) by using monotonically decreasing deque

Like this it can be used to get previous maximum also in a subarray

It is frequently used in sliding window problems (hard)

Other Applications:

Deques have several other applications, some of which include:

Palindrome checking: Deques can be used to check if a word or phrase is a palindrome. By inserting each character of the word or phrase into a deque,

it is possible to check if the word or phrase is a palindrome by comparing the first and last characters, the second and second-to-last characters,

and so on.

Graph traversal: Deques can be used to implement Breadth-First Search (BFS) on a graph. BFS uses a queue to keep track of the vertices to be visited

next, and a deque can be used as an alternative to a queue in this case.

Task scheduler: Deques can be used to implement a task scheduler that keeps track of tasks to be executed. Tasks can be added to the back of the

deque, and the scheduler can remove tasks from the front of the deque and execute them.

A double-ended queue can be used to store the browsing history: new websites are added to the end of the queue, while the oldest entries will be deleted

when the history is too large. When a user asks to clear the browsing history for the past hour, the most recently added entries are removed.

One example where a deque can be used is the work stealing algorithm.[6] This algorithm implements task scheduling for several processors.

A separate deque with threads to be executed is maintained for each processor. To execute the next thread, the processor gets the first

element from the deque (using the "remove first element" deque operation). If the current thread forks, it is put back to the front of the

deque ("insert element at front") and a new thread is executed. When one of the processors finishes execution of its own threads (i.e. its deque is empty),

it can "steal" a thread from another processor: it gets the last element from the deque of another processor ("remove last element") and executes it.

The work stealing algorithm is used by Intel's Threading Building Blocks (TBB) library for parallel programming.

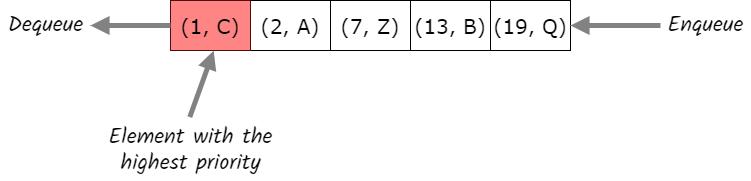
Multi-level undo/redo functionality: Deques can be used to implement undo and redo functionality in applications. Each time a user performs an

action, the current state of the application is pushed onto the deque. When the user undoes an action, the front of the deque is popped, and the

previous state is restored. When the user redoes an action, the next state is popped from the deque.

In computer science, deque can be used in many algorithms like LRU Cache, Round Robin Scheduling, Expression Evaluation.

**PRIORITY QUEUE**

A priority queue is a special type of queue. **Each queue’s item has an additional piece of information, namely priority**. Unlike a regular queue, the values in the priority queue are removed based on priority instead of the [first-in-first-out (FIFO)](https://www.baeldung.com/cs/fifo-page-replacement) rule. The following example illustrates a priority queue with an ordering imposed on the values from least to the greatest:**[](https://as-prod.asyncgw.teams.microsoft.com/v1/objects/0-sa-d3-f12697d9389e1dbc0b710f3a600026fd/views/imgo)**Here, , etc. denotes the value of items while , etc. denotes the priority of items. So the item with the highest priority in this example is (with the priority of 1) that is removed first. And the lowest priority item, (with the priority of 19), will be removed at the end of the process. **In this tutorial, from now on, we’ll use priority as the value of items since other information can be easily attached to the queue’s elements.** The main operations on a priority queue include:

* add: adds an item to the queue
* peek: returns the item in the queue with the highest priority without deleting the node
* remove: removes and returns the item in the queue with the highest priority

### **2.1. Characteristics of a Priority Queue**

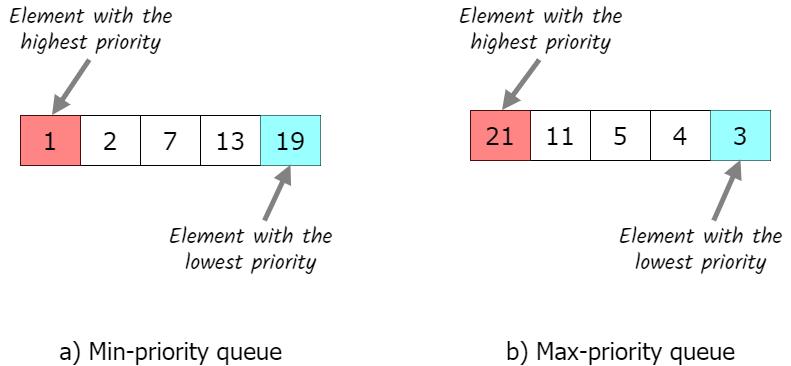
A priority queue is an extension of a queue that contains the following characteristics:

* Every element in a priority queue has a priority value associated with it
* The element with the higher priority will be moved to the top and removed first
* If two elements in a priority queue have the same priority value, they’ll be arranged using the FIFO principle

### **2.2. Types of Priority Queue**

There are two types of priority queues:

* Min-priority queue: in a min-priority queue, a lower priority number is given as a higher priority
* Max-priority queue: in a max-priority queue, a higher priority number is given as a higher priority

**[](https://as-prod.asyncgw.teams.microsoft.com/v1/objects/0-sa-d3-7bdda458d99bc0166e547b041e4d06f7/views/imgo)**In both types, **the priority queue stores a collection of elements and is always able to provide the most “extreme” element, which is the only way to extract the value from the priority queue**. For the remainder of this tutorial, we’ll discuss max-priority queues. Min-priority queues are analogous.

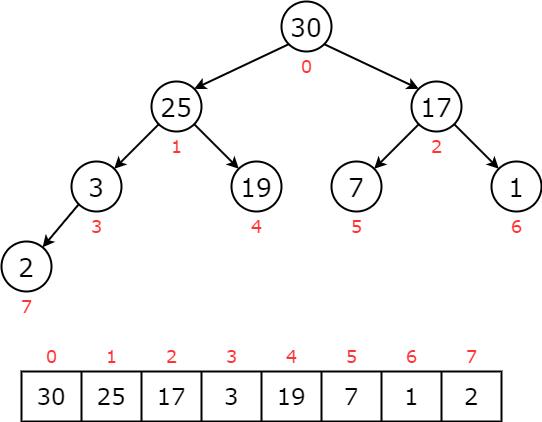
## 3. Implementation

There are different ways to implement a priority queue. The main ways include array, linked list, [binary search tree (BST)](https://www.baeldung.com/cs/binary-search-trees), and binary heap tree. **The heap data structure is the most efficient way to implement a priority queue**. Before diving deeper into the heap data structure, let’s briefly discuss the efficiency of each implementation approach on basic operations of queues:

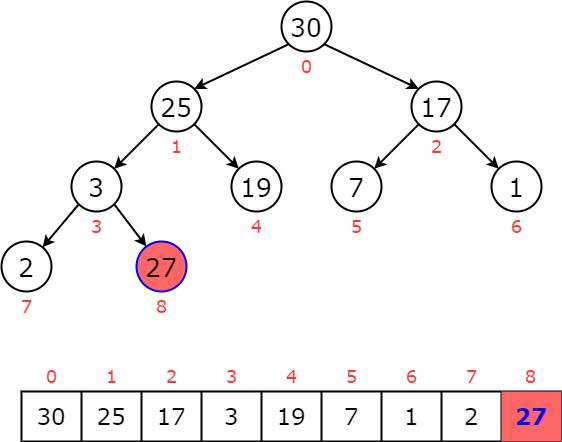
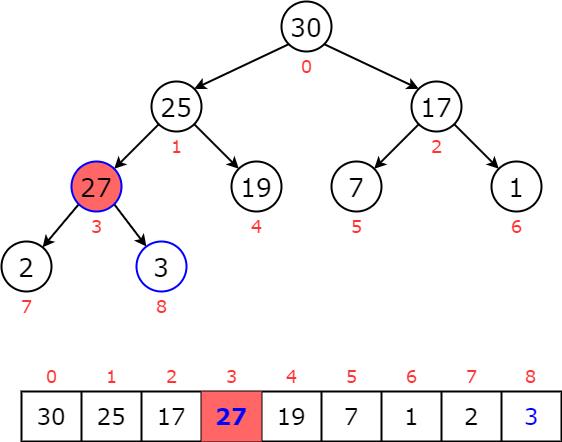
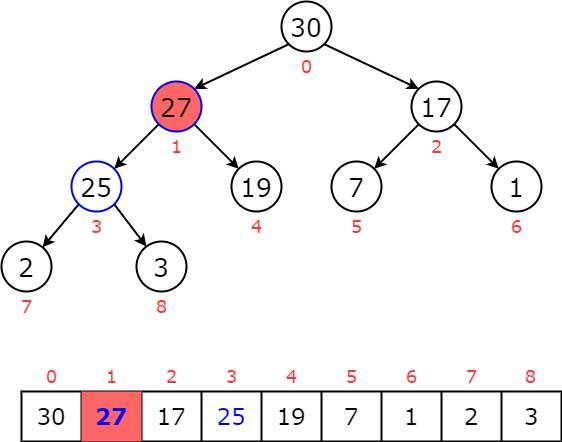
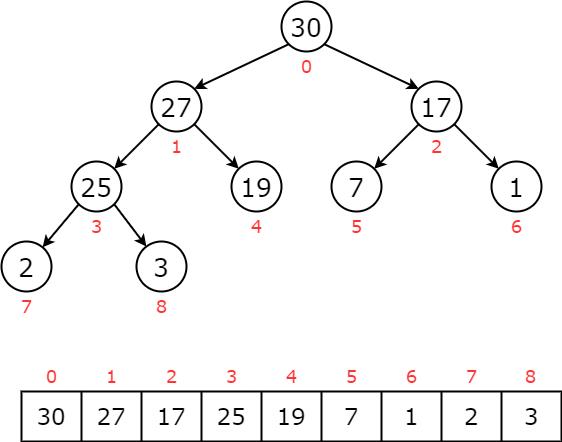
### **3.1. Heap Data Structure**

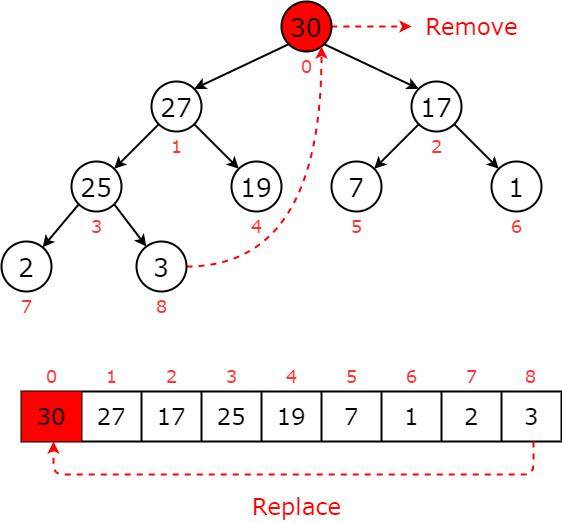
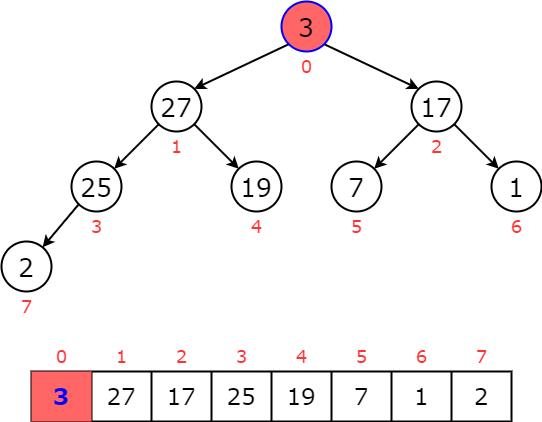
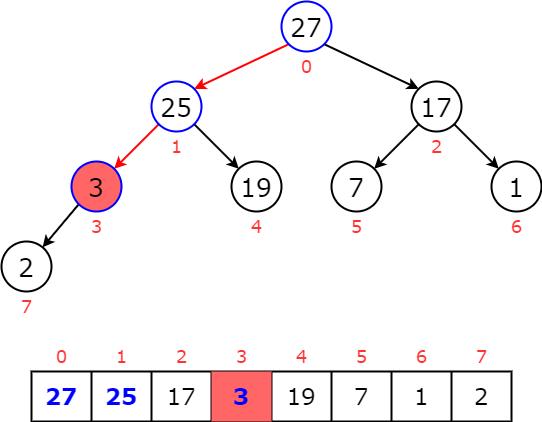
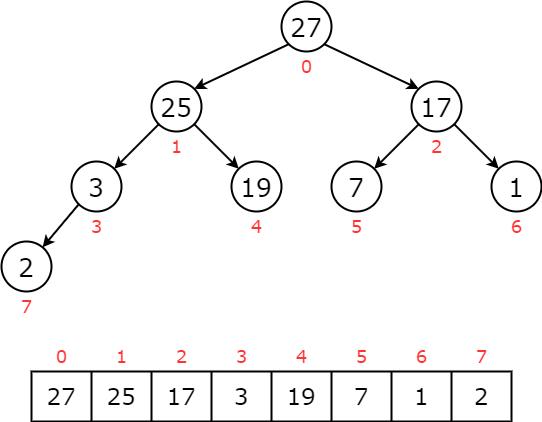
A [binary heap](https://www.baeldung.com/cs/heap-vs-binary-search-tree) is a type of binary tree (but not a binary search tree) that has the following properties:

* **Every node’s value must be greater (or smaller) than all values stored in its children**
* It’s a [complete tree](https://www.baeldung.com/cs/complete-vs-almost-complete-binary-tree), which means it has the smallest possible height

Throughout this tutorial, we’ll be using examples of a [max heap](https://www.baeldung.com/cs/binary-tree-max-heapify). Let’s examine the following example to understand the heap structure better:**[](https://as-prod.asyncgw.teams.microsoft.com/v1/objects/0-sa-d6-d3e9fccb376f4c419b772167b4b051b6/views/imgo)**For each node , the left child will be at , and the right child will be at . The parent will be at (except for the root node at ). For a more in-depth overview of the heap data structure, we can read the article which explains [Heap Sort in Java](https://www.baeldung.com/java-heap-sort).

### **3.2. Priority Queue Operations**

The common operations that we can perform on a priority queue include insertion, deletion, and peek. We’ll use a binary heap to maintain a max-priority queue. The item at the root of the heap has the highest priority among all elements. **If we want to add a new node to a binary heap, we need to ensure that our two properties of the heap are maintained after the new node is added.** At first, we insert the new item at the end of the priority queue. If it is found greater than its parent node, elements are swapped. This process continues until the new item is placed in the correct position. Let’s go through an example to understand the insertion process. If we add to the priority queue, we end up with:**[](https://as-prod.asyncgw.teams.microsoft.com/v1/objects/0-sa-d7-1b42623579c57d8fcd3f829004fffc1c/views/imgo)**We notice that is greater than its parent , so we swap them:**[](https://as-prod.asyncgw.teams.microsoft.com/v1/objects/0-sa-d3-cb44b9e0777efcb51f3e16a5b097323f/views/imgo)**Then, is still greater than its new parent , so we swap them:**[](https://as-prod.asyncgw.teams.microsoft.com/v1/objects/0-sa-d2-e7209e9d550ea7628acfd6b8003e6bc0/views/imgo)**Now, we notice that is smaller than its parent, so we stop and reach our priority queue:**[](https://as-prod.asyncgw.teams.microsoft.com/v1/objects/0-sa-d7-975c5675d24b3fcca6fd2d351be4a6ed/views/imgo)**The following code shows the details of the insertion function:

**Next, we can remove the maximum element from the priority queue.** As we know, the root node is the item with the highest priority in a max heap. When removing the root node, we replace it with the last item of the priority queue. Then, this item is compared with the child nodes and swap with the greater one. It keeps moving down the tree until the heap property is restored. Now, let’s apply to an example to understand how this process works. First, we replace the root node with the last item :**[](https://as-prod.asyncgw.teams.microsoft.com/v1/objects/0-sa-d3-ffbe901de67d44cddb45e22b944fc7e6/views/imgo)**The new priority queue is now violent the max heap’s property where the root node is smaller than its children:**[](https://as-prod.asyncgw.teams.microsoft.com/v1/objects/0-sa-d8-5c7b6940758a51fbd39ab9b59d5e4878/views/imgo)**Then, we swap node with the greater child until it reaches the leaf or greater than both children:**[](https://as-prod.asyncgw.teams.microsoft.com/v1/objects/0-sa-d5-5f5402357b76cc29705943017afd2e27/views/imgo)**Now, is greater than its child of , we stop and reach our priority queue:**[](https://as-prod.asyncgw.teams.microsoft.com/v1/objects/0-sa-d5-8e6a68ba36d22a3540221efcc8711616/views/imgo)**The following code shows the details of the deletion function:

**If we want to peek and see the largest node in a heap quickly, that is easy. Just return a pointer to the root:**

### **3.4. Time Complexity Analysis**

For insertion, we may have to heapify the entire heap data structure. So, while the insertion process only takes O(1) time, **the heapify process will take** . The same goes for deletion. We know where the max priority value is, but **remaking the heap still takes time**. **These are guaranteed worst-case efficiency since a binary heap always guarantees a complete tree.** The time complexity to extract the value from a priority queue is since we only need to peek at the root node of the heap.

**4. Constructors:**

**1.PriorityQueue():**Creates a PriorityQueue with the default initial capacity that orders its elements according to their natural ordering.

**2**.**PriorityQueue(Collection<E> c):**Creates a PriorityQueue containing the elements in the specified collection

**3.PriorityQueue(int initialCapacity)**: Creates a PriorityQueue with the specified initial capacity that orders its elements according to their natural ordering.

**4. PriorityQueue(int initialCapacity, Comparator<E> comparator):** Creates a PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.

**5. PriorityQueue(PriorityQueue<E> c)**: Creates a PriorityQueue containing the elements in the specified priority queue.

**6. PriorityQueue(SortedSet<E> c)**: Creates a PriorityQueue containing the elements in the specified sorted set.

## 5. Applications of Priority Queues

Priority queues are widely applied on other algorithms as well as in real-world systems. The main applications include:

* **Algorithms:** Certain foundational algorithms rely on priority queues, such as Dijkstra’s shortest path algorithm, prim’s algorithm, and heap sort algorithm, etc.
* **Data compression:** It is used in data compression techniques like Huffman code
* **Operating Systems:** Priority queues are used to select the next process to run, ensuring high-priority tasks run before low-priority ones**.** It is also applied for load balancing, and interrupt handling
* **Bandwidth Management:** Priority queues are utilized to prioritize the important data packet, so the network can make sure that those packets reach the destination as quickly as possible.

**ARRAYDEQUE IN JAVA**

The ArrayDeque in Java provides a way to apply resizable-array in addition to the implementation of the Deque interface. It is also known as Array Double Ended Queue or Array Deck. This is a special kind of array that grows and allows users to add or remove an element from both sides of the queue.

The ArrayDeque class in Java is an implementation of the Deque interface that uses a resizable array to store its elements. This class provides a more efficient alternative to the traditional Stack class, which was previously used for double-ended operations. The ArrayDeque class provides constant-time performance for inserting and removing elements from both ends of the queue, making it a good choice for scenarios where you need to perform many add and remove operations.

**Here’s an example of how you might use an ArrayDeque in Java:**

import java.util.ArrayDeque; import java.util.Deque; public class Example { public static void main(String Deque<Integer> deque = new ArrayDeque<>(); deque.addFirst(1); deque.addLast(2); int first = deque.removeFirst(); int last = deque.removeLast(); System.out.println("First: " + first + ", Last: " + last); } }

**Output:** First: 1, Last: 2

**Advantages of using ArrayDeque:**

1.Efficient: The ArrayDeque class provides constant-time performance for inserting and removing elements from both ends of the queue, making it a good choice for scenarios where you need to perform many add and remove operations.

2.Resizable: The ArrayDeque class uses a resizable array to store its elements, which means that it can grow and shrink dynamically to accommodate thenumber of elements in the queue. 3.Lightweight: The ArrayDeque class is a lightweight data structure that does not require additional overhead, such as linked list nodes, making it a good choice for scenarios where memory is limited. 4.Thread-safe: The ArrayDeque class is not thread-safe, but you can use the Collections.synchronizedDeque method to create a thread-safe version of the ArrayDeque class.

**Disadvantages of using ArrayDeque:**

1.Not synchronized: By default, the ArrayDeque class is not synchronized, which means that multiple threads can access it simultaneously, leading to potential data corruption. 2.Limited capacity: Although the ArrayDeque class uses a resizable array to store its elements, it still has a limited capacity, which means that you may need to create a new ArrayDeque when the old one reaches its maximum size.

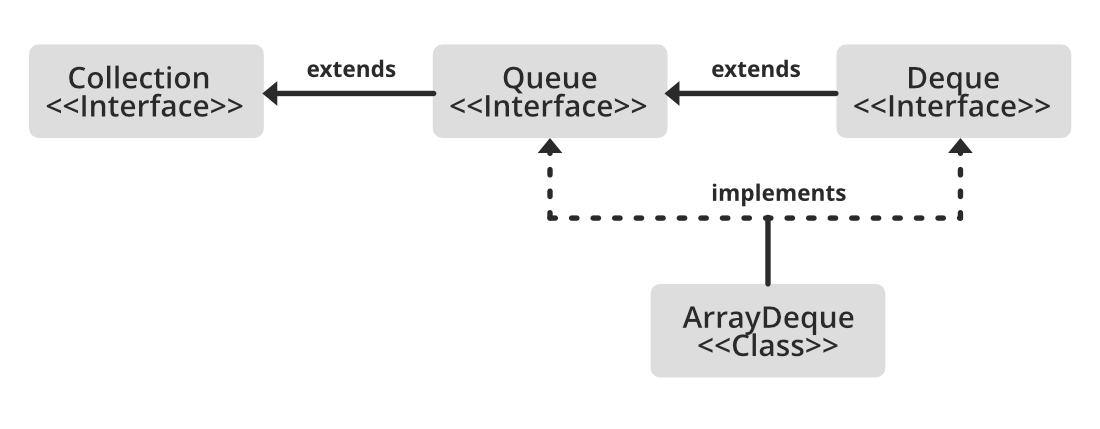
**Few important features of ArrayDeque are as follows:**

1.Array deques have no capacity restrictions and they grow as necessary to support usage. 2.They are not thread-safe which means that in the absence of external synchronization, ArrayDeque does not support concurrent access by multiple threads. 3.Null elements are prohibited in the ArrayDeque. 4.ArrayDeque class is likely to be faster than Stack when used as a stack. 5.ArrayDeque class is likely to be faster than LinkedList when used as a queue.

**Interfaces implemented by ArrayDeque:**

The ArrayDeque class implements these two interfaces:

1.Queue Interface: It is an Interface that is a FirstIn – FirstOut Data Structure where the elements are added from the back.2.Deque Interface: It is a Doubly Ended Queue in which you can insert the elements from both sides. It is an interface that implements the Queue.ArrayDeque implements both Queue and Deque. It is dynamically resizable from both sides. All implemented interfaces of ArrayDeque in the hierarchy are Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, Queue<E>.



**Syntax: Declaration-->**

public class ArrayDeque<E> extends AbstractCollection<E> implements Deque<E>, Cloneable, Serializable

Here, E refers to the element which can refer to any class, such as Integer or String class.

**Constructors of Array Deque**

1.ArrayDeque(): This constructor is used to create an empty ArrayDeque and by default holds an initial capacity to hold 16 elements.

*ArrayDeque<E> dq = new ArrayDeque<E>();*

2.ArrayDeque(Collection<? extends E> c): This constructor is used to create an ArrayDeque containing all the elements the same as that of the specified collection. *ArrayDeque<E> dq = new ArrayDeque<E>(Collection col);*

3.ArrayDeque(int numofElements): This constructor is used to create an empty ArrayDeque and holds the capacity to contain a specified number of elements.

*ArrayDeque<E> dq = new ArrayDeque<E>(int numofElements);*

**Methods of ArrayDeque**

1.add(Element e)-The method inserts a particular element at the end of the deque. 2.clear()-The method removes all deque elements. 3.clone()-The method copies the deque. 4.contains(Obj)-The method checks whether a deque contains the element or not. 5.element()-The method returns element at the head of the deque. 6.getFirst()-The method returns first element of the deque. 7.getLast()-The method returns last element of the deque. 8.isEmpty()-The method checks whether the deque is empty or not. 9.iterator()-Returns an iterator over the elements in this deque. 10.offer(Elemente)-The method inserts element at the end of deque. 11.peek()-The method returns head element without removing it. 12.poll()-The method returns head element and also removes it. 13.pop()-The method pops out an element for stack represented by deque. 14.push(Element e)-The method pushes an element onto stack represented by deque. 15.remove()-The method returns head element and also removes it. 16.descendingIterator()-Returns an iterator over the elements in this deque in reverse sequential order.

**Example:**

// Java program to Implement ArrayDeque in Java

//

// Importing utility classes

import java.util.\*;

// ArrayDequeDemo

public class GGFG {

public static void main(String[] args)

{

// Creating and initializing deque

// Declaring object of integer type

Deque<Integer> de\_que = new ArrayDeque<Integer>(10);

// Operations 1

// add() method

// Adding custom elements

// using add() method to insert

de\_que.add(10);

de\_que.add(20);

de\_que.add(30);

de\_que.add(40);

de\_que.add(50);

// Iterating using for each loop

for (Integer element : de\_que) {

// Print the corresponding element

System.out.println("Element : " + element);

}

// Operation 2

// clear() method

System.out.println("Using clear() ");

// Clearing all elements using clear() method

de\_que.clear();

// Operations 3

// addFirst() method

// Inserting at the start

de\_que.addFirst(564);

de\_que.addFirst(291);

// Operation 4

// addLast() method

// Inserting at end

de\_que.addLast(24);

de\_que.addLast(14);

// Display message

System.out.println(

"Above elements are removed now");

// Iterators

// Display message

System.out.println(

"Elements of deque using Iterator :");

for (Iterator itr = de\_que.iterator();

itr.hasNext();) {

System.out.println(itr.next());

}

// descendingIterator()

// To reverse the deque order

System.out.println(

"Elements of deque in reverse order :");

for (Iterator dItr = de\_que.descendingIterator();

dItr.hasNext();) {

System.out.println(dItr.next());

}

// Operation 5

// element() method : to get Head element

System.out.println(

"\nHead Element using element(): "

+ de\_que.element());

// Operation 6

// getFirst() method : to get Head element

System.out.println("Head Element using getFirst(): "

+ de\_que.getFirst());

// Operation 7

// getLast() method : to get last element

System.out.println("Last Element using getLast(): "

+ de\_que.getLast());

// Operation 8

// toArray() method :

Object[] arr = de\_que.toArray();

System.out.println("\nArray Size : " + arr.length);

System.out.print("Array elements : ");

for (int i = 0; i < arr.length; i++)

System.out.print(" " + arr[i]);

// Operation 9

// peek() method : to get head

System.out.println("\nHead element : "

+ de\_que.peek());

// Operation 10

// poll() method : to get head

System.out.println("Head element poll : "

+ de\_que.poll());

// Operation 11

// push() method

de\_que.push(265);

de\_que.push(984);

de\_que.push(2365);

// Operation 12

// remove() method : to get head

System.out.println("Head element remove : "

+ de\_que.remove());

System.out.println("The final array is: " + de\_que);

}

}

**Output:**

Element : 10

Element : 20

Element : 30

Element : 40

Element : 50

Using clear()

Above elements are removed now

Elements of deque using Iterator :

291

564

24

14

Elements of deque in reverse order :

14

24

564

291

Head Element using element(): 291

Head Element using getFirst(): 291

Last Element using getLast(): 14

Array Size : 4

Array elements : 291 564 24 14

Head element : 291

Head element poll : 291

Head element remove : 2365

The final array is: [984, 265, 564, 24, 14]

**Operations in ArrayDeque**

1.Adding operation 2.Accessing operation 3.Removing operations 4.Iterating through the Deque

**Operation 1: Adding Elements**

In order to add an element to the ArrayDeque, we can use the methods add(), addFirst(), addLast(), offer(), offerFirst(), offerLast() methods.

add() addFirst() addLast()offer() offerFirst() offerLast()

**Example**

// Java program to Illustrate Addition of elements

// in ArrayDeque

// Importing required classes

import java.io.\*;

import java.util.\*;

// Main class

// AddingElementsToArrayDeque

public class GFG {

// Main driver method

public static void main(String[] args)

{

// Initializing a deque

// since deque is an interface

// it is assigned the

// ArrayDeque class

Deque<String> dq = new ArrayDeque<String>();

// add() method to insert

dq.add("The");

dq.addFirst("To");

dq.addLast("Geeks");

// offer() method to insert

dq.offer("For");

dq.offerFirst("Welcome");

dq.offerLast("Geeks");

// Printing Elements of ArrayDeque to the console

System.out.println("ArrayDeque : " + dq);

}

}

**Output:**

ArrayDeque : [Welcome, To, The, Geeks, For, Geeks]

**Operation 2: Accessing the Elements**

After adding the elements, if we wish to access the elements, we can use inbuilt methods like getFirst(), getLast(), etc.

getFirst() getLast() peek() peekFirst() peekLast()

**Example**

// Main driver method

public static void main(String args[])

{

// Creating an empty ArrayDeque

ArrayDeque<String> de\_que

= new ArrayDeque<String>();

// Using add() method to add elements into the Deque

// Custom input elements

de\_que.add("Welcome");

de\_que.add("To");

de\_que.add("Geeks");

de\_que.add("4");

de\_que.add("Geeks");

// Displaying the ArrayDeque

System.out.println("ArrayDeque: " + de\_que);

// Displaying the First element

System.out.println("The first element is: "

+ de\_que.getFirst());

// Displaying the Last element

System.out.println("The last element is: "

+ de\_que.getLast());

}

}

**Output:**

ArrayDeque: [Welcome, To, Geeks, 4, Geeks]

The first element is: Welcome

The last element is: Geeks

**Operation 3. Removing Elements**

In order to remove an element from a deque, there are various methods available. Since we can also remove from both the ends, the deque interface provides us with removeFirst(), removeLast() methods. Apart from that, this interface also provides us with the poll(), pop(), pollFirst(), pollLast() methods where pop() is used to remove and return the head of the deque. However, poll() is used because this offers the same functionality as pop() and doesn’t return an exception when the deque is empty. These sets of operations are as listed below as follows:

remove() removeFirst() removeLast() poll() pollFirst() pollLast() pop()

**Example**

// Java program to Illustrate Removal Elements in Deque

// Importing all utility classes

import java.util.\*;

// RemoveElementsOfArrayDeque

public class GFG {

// Main driver method

public static void main(String[] args)

{

// Initializing a deque

Deque<String> dq = new ArrayDeque<String>();

// add() method to insert

dq.add("One");

// addFirst inserts at the front

dq.addFirst("Two");

// addLast inserts at the back

dq.addLast("Three");

// print elements to the console

System.out.println("ArrayDeque : " + dq);

// remove element as a stack from top/front

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

// remove element as a queue from front

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

// remove element from front

System.out.println(dq.pollFirst());

// remove element from back

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

}

}

**Output:**

ArrayDeque : [Two, One, Three]

Two

One

Three

null

**Operation 4: Iterating through the Deque**

Since a deque can be iterated from both directions, the iterator method of the deque interface provides us two ways to iterate. One from the first and the other from the back. These sets of operations are listed below as follows:

remove() iterator() descendingIterator()

**Example**

// Java program to Illustrate Iteration of Elements

// in Deque

// Importing all utility classes

import java.util.\*;

// Main class

// IterateArrayDeque

public class GFG {

// Main driver method

public static void main(String[] args)

{

// Declaring and initializing an deque

Deque<String> dq = new ArrayDeque<String>();

// Adding elements at the back

// using add() method

dq.add("For");

// Adding element at the front

// using addFirst() method

dq.addFirst("Geeks");

// add element at the last

// using addLast() method

dq.addLast("Geeks");

dq.add("is so good");

// Iterate using Iterator interface

// from the front of the queue

for (Iterator itr = dq.iterator(); itr.hasNext();) {

// Print the elements

System.out.print(itr.next() + " ");

}

// New line

System.out.println();

// Iterate in reverse sequence in a queue

for (Iterator itr = dq.descendingIterator();

itr.hasNext();) {

System.out.print(itr.next() + " ");

}

}

}

**Output:**

Geeks For Geeks is so good

is so good Geeks For Geeks

**ArrayDeque as Stack**

1.To implement LIFO stacks in JAVA. 2.Faster 3.Methods for implementing stack. -->push()-adds an element to the top of the stack. -->Peek()-returns an element from the top of the stack. -->Pop()-returns and removes an element from the top of the stack.