Java interview questions

Working java 8 and rest services and micro services ide eclipse

## Java 8 feature

# Lambda Expressions

Lambda Expressions advance version Anonymous function. Lambda expressions basically instances of [functional interfaces](https://www.geeksforgeeks.org/functional-interfaces-java/) (An interface with single abstract method is called functional interface). It is very useful in collection library. It helps to iterate, filter and extract data from collection.

## Functional Interface

Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface. Java provides an anotation @FunctionalInterface, which is used to declare an interface as functional interface.

Exp

1. @FunctionalInterface  //It is optional
2. **interface** Drawable{
3. **public** **void** draw();
4. }
6. **public** **class** LambdaExpressionExample2 {
7. **public** **static** **void** main(String[] args) {
8. **int** width=10;
10. //with lambda
11. Drawable d2=()->{
12. System.out.println("Drawing "+width);
13. };
14. d2.draw();
15. }
16. }
17. **interface** Addable{
18. **int** add(**int** a,**int** b);
19. }
21. **public** **class** LambdaExpressionExample5{
22. **public** **static** **void** main(String[] args) {
24. // Multiple parameters in lambda expression
25. Addable ad1=(a,b)->(a+b);
26. System.out.println(ad1.add(10,20));
28. // Multiple parameters with data type in lambda expression
29. Addable ad2=(**int** a,**int** b)->(a+b);
30. System.out.println(ad2.add(100,200));
31. }
32. }

Output:

30

300

# Java Method References

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

## Types of Method References

1. Reference to a static method. : - You can refer to static method defined in the class

.

**interface** Sayable{

**void** say();

  }

**public** **class** MethodReference {

**public** **static** **void** saySomething(){

        System.out.println("Hello, this is static method.");

    }

**public** **static** **void** main(String[] args) {

        // Referring static method

        Sayable sayable = MethodReference::saySomething;

        // Calling interface method

        sayable.say();

     }

}

Exp -2

1. **import** java.util.function.BiFunction;
2. **class** Arithmetic{
3. **public** **static** **int** add(**int** a, **int** b){
4. **return** a+b;
5. }
6. }
7. **public** **class** MethodReference3 {
8. **public** **static** **void** main(String[] args) {
9. BiFunction<Integer, Integer, Integer>adder = Arithmetic::add;
10. **int** result = adder.apply(10, 20);
11. System.out.println(result);
12. }

Ans 30

Reference to an instance method.

1. **interface** Sayable{
2. **void** say();
3. }
4. **public** **class** InstanceMethodReference {
5. **public** **void** saySomething(){
6. System.out.println("Hello, this is non-static method.");
7. }
8. **public** **static** **void** main(String[] args) {
9. InstanceMethodReference methodReference = **new** InstanceMethodReference(); // Creating object
10. // Referring non-static method using reference
11. Sayable sayable = methodReference::saySomething;
12. // Calling interface method
13. sayable.say();
14. // Referring non-static method using anonymous object
15. Sayable sayable2 = **new** InstanceMethodReference()::saySomething; // You can use anonymous object also
16. // Calling interface method
17. sayable2.say();
18. }
19. }

3.Reference to a constructor.

1. **interface** Messageable{
2. Message getMessage(String msg);
3. }
4. **class** Message{
5. Message(String msg){
6. System.out.print(msg);
7. }
8. }
9. **public** **class** ConstructorReference {
10. **public** **static** **void** main(String[] args) {
11. Messageable hello = Message::**new**;
12. hello.getMessage("Hello");
13. }
14. }

# Java Functional Interfaces

An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.

Functional Interface is also known as Single Abstract Method Interfaces or SAM Interfaces. It is a new feature in Java, which helps to achieve functional programming approach.

# Java 8 Stream

 The Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

* A stream is not a data structure . it takes input from the Collections, Arrays or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.

|  |
| --- |
| /a simple program to demonstrate the use of stream in java  import java.util.\*;  import java.util.stream.\*;  class Demo  {    public static void main(String args[])    {        // create a list of integers      List<Integer> number = Arrays.asList(2,3,4,5);        // demonstration of map method      List<Integer> square = number.stream().map(x -> x\*x).collect(Collectors.toList());      System.out.println(square);        // create a list of String      List<String> names = Arrays.asList("Reflection","Collection","Stream");        // demonstration of filter method      List<String> result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());      System.out.println(result);        // demonstration of sorted method      List<String> show = names.stream().sorted().collect(Collectors.toList());      System.out.println(show);        // create a list of integers      List<Integer> numbers = Arrays.asList(2,3,4,5,2);        // collect method returns a set      Set<Integer> squareSet =  numbers.stream().map(x->x\*x).collect(Collectors.toSet());      System.out.println(squareSet);        // demonstration of forEach method      number.stream().map(x->x\*x).forEach(y->System.out.println(y));        // demonstration of reduce method      int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);    System.out.println(even);    }  } |
|  |

Output:

[4, 9, 16, 25]

[Stream]

[Collection, Reflection, Stream]

[16, 4, 9, 25]

4

9

16

25

6

# Java Stream Filter

Java stream provides a method filter() to filter stream elements on the basis of given predicate. Suppose you want to get only even elements of your list then you can do this easily with the help of filter method.

This method takes predicate as an argument and returns a stream of consisting of resulted elements.

|  |
| --- |
| / Java code for Stream filter  // (Predicate predicate) to get a stream  // consisting of the elements of this  // stream that match the given predicate.  import java.util.\*;    class GFG {        // Driver code      public static void main(String[] args)      {            // Creating a list of Integers          List<Integer> list = Arrays.asList(3, 4, 6, 12, 20);            // Using Stream filter(Predicate predicate)          // to get a stream consisting of the          // elements that are divisible by 5          list.stream().filter(num -> num % 5 == 0).forEach(System.out::println);      }  } |

Output :

20

# Java Default Methods

Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

1. **interface** Sayable{
2. // default method
3. **default** **void** say(){
4. System.out.println("Hello, this is default method");
5. }
6. // Abstract method
7. **void** sayMore(String msg);
8. // static method
9. **static** **void** sayLouder(String msg){
10. System.out.println(msg);
11. }
12. }
13. **public** **class** DefaultMethods **implements** Sayable{
14. **public** **void** sayMore(String msg){     // implementing abstract method
15. System.out.println(msg);
16. }
17. **public** **static** **void** main(String[] args) {
18. DefaultMethods dm = **new** DefaultMethods();
19. dm.say();                       // calling default method
20. dm.sayMore("Work is worship");      // calling abstract method
21. Sayable.sayLouder("Helloooo...");   // calling static method
22. }
23. }

Output:

Hello there

Work is worship

Helloooo...

## Abstract Class vs Java 8 Interface

After having default and static methods inside the interface, we think about the need of abstract class in Java. An interface and an abstract class is almost similar except that you can create constructor in the abstract class whereas you can't do this in interface.

# Java Collectors

Collectors is a final class that extends Object class. we use widely and those functions are **filter()**, **map()**, **reduce()**, and **collect()** which belongs to the [Streams API](https://www.geeksforgeeks.org/stream-in-java/). **collect()** and **reduce()** methods are called as the terminal methods because here,

# Java StringJoiner

Java added a new final class StringJoiner in java.util package. It is used to construct a sequence of characters separated by a delimiter. Now, you can create string by passing delimiters like comma(,), hyphen(-) etc. You can also pass prefix and suffix to the char sequence.

1. **import** java.util.StringJoiner;
2. **public** **class** StringJoinerExample {
3. **public** **static** **void** main(String[] args) {
4. StringJoiner joinNames = **new** StringJoiner(","); // passing comma(,) as delimiter
6. // Adding values to StringJoiner
7. joinNames.add("Rahul");
8. joinNames.add("Raju");
9. joinNames.add("Peter");
10. joinNames.add("Raheem");
12. System.out.println(joinNames);
13. }
14. }

Output:

Rahul,Raju,Peter,Raheem

# Java Optional Class

Java introduced a new class Optional in jdk8. It is a public final class and used to deal with NullPointerException in Java application. You must import java.util package to use this class. It provides methods which are used to check the presence of value for particular variable.

### Java Optional Example: If Value is not Present

1. **import** java.util.Optional;
2. **public** **class** OptionalExample {
3. **public** **static** **void** main(String[] args) {
4. String[] str = **new** String[10];
5. Optional<String> checkNull = Optional.ofNullable(str[5]);
6. **if**(checkNull.isPresent()){  // check for value is present or not
7. String lowercaseString = str[5].toLowerCase();
8. System.out.print(lowercaseString);
9. }**else**
10. System.out.println("string value is not present");
11. }
12. }

Output:

string value is not present

### Java Optional Example: If Value is Present

1. **import** java.util.Optional;
2. **public** **class** OptionalExample {
3. **public** **static** **void** main(String[] args) {
4. String[] str = **new** String[10];
5. str[5] = "JAVA OPTIONAL CLASS EXAMPLE";// Setting value for 5th index
6. Optional<String> checkNull = Optional.ofNullable(str[5]);
7. **if**(checkNull.isPresent()){  // It Checks, value is present or not
8. String lowercaseString = str[5].toLowerCase();
9. System.out.print(lowercaseString);
10. }**else**
11. System.out.println("String value is not present");
12. }
13. }

Output:

java optional class example

# Java Nashorn

Nashorn is a JavaScript engine. It is used to execute JavaScript code dynamically at JVM (Java Virtual Machine). Java provides a command-line tool jjs which is used to execute JavaScript code.

You can execute JavaScript code by using jjs command-line tool and by embedding into Java source code

1. var hello = function(){
2. print("Hello Nashorn");
3. };
4. hello();

### Example: Executing JavaScript file in Java Code

You can execute JavaScript file directly from your Java file. In the following code, we are reading a file hello.js with the help of FileReader class.

1. **import** javax.script.\*;
2. **import** java.io.\*;
3. **public** **class** NashornExample {
4. **public** **static** **void** main(String[] args) **throws** Exception{
5. // Creating script engine
6. ScriptEngine ee = **new** ScriptEngineManager().getEngineByName("Nashorn");
7. // Reading Nashorn file
8. ee.eval(**new** FileReader("js/hello.js"));
9. }
10. }

Output:

Hello Nashorn

# Java Parallel Array Sorting

Java provides a new additional feature in Array class which is used to sort array elements parallel.New methods has added to java.util.Arrays package that use the JSR 166 Fork/Join parallelism common pool to provide sorting of arrays in parallel.The methods are called parallelSort() and are overloaded for all the primitive data types and Comparable objects.

1. **import** java.util.Arrays;
2. **public** **class** ParallelArraySorting {
3. **public** **static** **void** main(String[] args) {
4. // Creating an integer array
5. **int**[] arr = {5,8,1,0,6,9};
6. // Iterating array elements
7. **for** (**int** i : arr) {
8. System.out.print(i+" ");
9. }
10. // Sorting array elements parallel
11. Arrays.parallelSort(arr);
12. System.out.println("\nArray elements after sorting");
13. // Iterating array elements
14. **for** (**int** i : arr) {
15. System.out.print(i+" ");
16. }
17. }
18. }

Output:

5 8 1 0 6 9

Array elements after sorting

0 1 5 6 8 9

Java 9 Feature

# Java 9 Private Interface Methods

In Java 9, we can create private methods inside an interface. Interface allows us to declare private methods that help to **share** common code between **non-abstract** methods.

1. **interface** Sayable{
2. **default** **void** say() {
3. saySomething(); // Calling private method
4. sayPolitely(); //  Calling private static method
5. }
6. // Private method inside interface
7. **private** **void** saySomething() {
8. System.out.println("Hello... I'm private method");
9. }
10. // Private static method inside interface
11. **private** **static** **void** sayPolitely() {
12. System.out.println("I'm private static method");
13. }
14. }
15. **public** **class** PrivateInterface **implements** Sayable {
16. **public** **static** **void** main(String[] args) {
17. Sayable s = **new** PrivateInterface();
18. s.say();
19. }
20. }

Output:

Hello... I'm private method

I'm private static method

# Java 9 Try With Resource Enhancement

Java introduced **try-with-resource** feature in Java 7 that helps to close resource automatically after being used.

In other words, we can say that we don't need to close resources (file, connection, network etc) explicitly, try-with-resource close that automatically by using AutoClosable interface.

1. **import** java.io.FileNotFoundException;
2. **import** java.io.FileOutputStream;
3. **public** **class** FinalVariable {
4. **public** **static** **void** main(String[] args) **throws** FileNotFoundException {
5. FileOutputStream fileStream=**new** FileOutputStream("javatpoint.txt");
6. **try**(fileStream){
7. String greeting = "Welcome to javaTpoint.";
8. **byte** b[] = greeting.getBytes();
9. fileStream.write(b);
10. System.out.println("File written");
11. }**catch**(Exception e) {
12. System.out.println(e);
13. }
14. }
15. }

Output:

File written

# Java 9 Anonymous Inner Classes Improvement

Java 9 introduced a new feature that allows us to use diamond operator with anonymous classes.

Data types that can be written in Java program like int, String etc are called denotable types. Java 9 compiler is enough smart and now can infer type.

1. **abstract** **class** ABCD<T>{
2. **abstract** T show(T a, T b);
3. }
4. **public** **class** TypeInferExample {
5. **public** **static** **void** main(String[] args) {
6. ABCD<String> a = **new** ABCD<String>() { // diamond operator is not empty
7. String show(String a, String b) {
8. **return** a+b;
9. }
10. };
11. String result = a.show("Java","9");
12. System.out.println(result);
13. }
14. }

And we get the same result.

Output:

Java9

# Java I/O Tutorial

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in Java** by Java I/O API.

## Stream

A stream is a sequence of data. In Java, a stream is composed of bytes.

**1) System.out:**standard output stream

Exp System.out.println("simple message");

**2) System.in:**standard input stream

Exp **int** i=System.in.read();//returns ASCII code of 1st character

System.out.println((**char**)i);//will print the character

**3) System.err:**standard error stream

Exp System.err.println("error message");

## OutputStream vs InputStream

### OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

### InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Let's understand the working of Java OutputStream and InputStream by the figure given below.

Java IO

## OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

### Useful methods of OutputStream

|  |  |
| --- | --- |
| 1) public void write(int)throws IOException | is used to write a byte to the current output stream. |
| 2) public void write(byte[])throws IOException | is used to write an array of byte to the current output stream. |
| 3) public void flush()throws IOException | flushes the current output stream. |
| 4) public void close()throws IOException | is used to close the current output stream. |

### OutputStream Hierarchy

Java output stream hierarchy

## InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

### Useful methods of InputStream

|  |  |
| --- | --- |
| **Method** | **Description** |
| 1) public abstract int read()throws IOException | reads the next byte of data from the input stream. It returns -1 at the end of the file. |
| 2) public int available()throws IOException | returns an estimate of the number of bytes that can be read from the current input stream. |
| 3) public void close()throws IOException | is used to close the current input stream. |

### InputStream Hierarchy

Java input stream hierarchy

# Java FileOutputStream Class

Java FileOutputStream is an output stream used for writing data to a [file](https://www.javatpoint.com/java-file-class). You can write byte-oriented as well as character-oriented data through FileOutputStream class

Exp

**try**{

             FileOutputStream fout=**new** FileOutputStream("D:\\testout.txt");

             fout.write(65);

             fout.close();

             System.out.println("success...");

            }**catch**(Exception e){System.out.println(e);}

# Java FileInputStream Class

Java FileInputStream class obtains input bytes from a [file](https://www.javatpoint.com/java-file-class). It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc.

**try**{

            FileInputStream fin=**new** FileInputStream("D:\\testout.txt");

**int** i=fin.read();

            System.out.print((**char**)i);

            fin.close();

          }**catch**(Exception e){System.out.println(e);}

# Java BufferedOutputStream Class

Java BufferedOutputStream [class](https://www.javatpoint.com/object-and-class-in-java) is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

For adding the buffer in an OutputStream, use the BufferedOutputStream class. Let's see the syntax for adding the buffer in an OutputStream:

1. OutputStream os= **new** BufferedOutputStream(**new** FileOutputStream("D:\\IO Package\\testout.txt"));

EXP   FileOutputStream fout=**new** FileOutputStream("D:\\testout.txt");

     BufferedOutputStream bout=**new** BufferedOutputStream(fout);

     String s="Welcome to javaTpoint.";

**byte** b[]=s.getBytes();

     bout.write(b);

     bout.flush();

     bout.close();

     fout.close();

     System.out.println("success");

# Java BufferedInputStream Class

Java BufferedInputStream [class](https://www.javatpoint.com/object-and-class-in-java) is used to read information from [stream](https://www.javatpoint.com/java-8-stream). It internally uses buffer mechanism to make the performance fast.

**try**{

    FileInputStream fin=**new** FileInputStream("D:\\testout.txt");

    BufferedInputStream bin=**new** BufferedInputStream(fin);

**int** i;

**while**((i=bin.read())!=-1){

     System.out.print((**char**)i);

    }

    bin.close();

    fin.close();

  }**catch**(Exception e){System.out.println(e);}

# Java SequenceInputStream Class

[Java](https://www.javatpoint.com/java-tutorial) SequenceInputStream [class](https://www.javatpoint.com/object-class) is used to read data from multiple [streams](https://www.javatpoint.com/java-8-stream). It reads data sequentially (one by one).

Exp

FileInputStream input1=**new** FileInputStream("D:\\testin.txt");

   FileInputStream input2=**new** FileInputStream("D:\\testout.txt");

   SequenceInputStream inst=**new** SequenceInputStream(input1, input2);

**int** j;

**while**((j=inst.read())!=-1){

    System.out.print((**char**)j);

   }

   inst.close();

   input1.close();

   input2.close();

  }

# ava ByteArrayOutputStream Class

Java ByteArrayOutputStream class is used to **write common data** into multiple files. In this stream, the data is written into a byte [array](https://www.javatpoint.com/array-in-java) which can be written to multiple streams later.

The ByteArrayOutputStream holds a copy of data and forwards it to multiple streams.

# Java ByteArrayInputStream Class

The ByteArrayInputStream is composed of two words: ByteArray and InputStream. As the name suggests, it can be used to read byte [array](https://www.javatpoint.com/array-in-java) as input stream.

Java ByteArrayInputStream [class](https://www.javatpoint.com/object-and-class-in-java) contains an internal buffer which is used to **read byte array** as stream. In this stream, the data is read from a byte array.

**byte**[] buf = { 35, 36, 37, 38 };

    // Create the new byte array input stream

    ByteArrayInputStream byt = **new** ByteArrayInputStream(buf);

**int** k = 0;

**while** ((k = byt.read()) != -1) {

      //Conversion of a byte into character

**char** ch = (**char**) k;

      System.out.println("ASCII value of Character is:" + k + "; Special character is: " + ch);

    }

# Collections in Java And Questions

### ****1. What are the advantages of the Collection Framework in Java?****

Below table contains the major advantages of the Java Collection Framework:

|  |  |
| --- | --- |
| **Feature** | **Description** |
| Performance | The collection framework provides highly effective and efficient data structures that result in enhancing the speed and accuracy of a program. |
| Maintainability | The code developed with the collection framework is easy to maintain as it supports data consistency and interoperability within the implementation. |
| Reusability | The classes in Collection Framework can effortlessly mix with other types which results in increasing the code reusability. |
| Extensibility | The Collection Framework in Java allows the developers to customize the primitive collection types as per their requirements. |

### ****4. List down the primary interfaces provided by Java Collections Framework?****

* Collection Interface: java.util.Collection is the root of the Java Collection framework and most of the collections in Java are inherited from this [interface](https://www.edureka.co/blog/java-interface/).
* List Interface: java.util.List is an extended form of an array that contains ordered elements and may include duplicates. It supports the index-based search, but elements can be easily inserted irrespective of the position. The List interface is implemented by various classes such as ArrayList, LinkedList, Vector, etc.
* Set Interface: java.util.Set refers to a collection class that cannot contain duplicate elements. Since it doesn’t define an order for the elements, the index-based search is not supported. It is majorly used as a mathematical set abstraction model. The Set interface is implemented by various classes such as HashSet, TreeSetand LinkedHashSet.
* Queue Interface: java.util.Queue in Java follows a FIFO approach i.e. it orders the elements in First In First Out manner. Elements in Queue will be added from the rear end while removed from the front.
* Map Interface: java.util.Map is a two-dimensional data structure in Java that is used to store the data in the form of a Key-Value pair. The key here is the unique hashcode and value represent the element. Map in Java is another form of the Java Set but can’t contain duplicate elements.

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes ([ArrayList](https://www.javatpoint.com/java-arraylist), Vector, [LinkedList](https://www.javatpoint.com/java-linkedlist), [PriorityQueue](https://www.javatpoint.com/java-priorityqueue), HashSet, LinkedHashSet, TreeSet).



## Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

Iterator interface provides the facility of iterating the elements in a forward direction only.

## Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

## List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

# Java ArrayList

Java **ArrayList** class uses a dynamic [*array*](https://www.javatpoint.com/array-in-java) for storing the elements. It is like an array, but there is no size limit. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the java.util package. It is like the Vector in C++.

The ArrayList in Java can have the duplicate elements also. It implements the List interface so we can use all the methods of List interface here. The ArrayList maintains the insertion order internally.

* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non [synchronized](https://www.javatpoint.com/synchronization-in-java).
* Java ArrayList allows random access because array works at the index basis.
* In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

**import** java.util.\*;

**public** **class** ArrayListExample2{

**public** **static** **void** main(String args[]){

  ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist

  list.add("Mango");//Adding object in arraylist

  list.add("Apple");

  list.add("Banana");

  list.add("Grapes");

  //Traversing list through Iterator

  Iterator itr=list.iterator();//getting the Iterator

**while**(itr.hasNext()){//check if iterator has the elements

   System.out.println(itr.next());//printing the element and move to next

  }

 }

}

# Java LinkedList class

Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* Java LinkedList class is non synchronized.
* In Java LinkedList class, manipulation is fast because no shifting needs to occur.
* Java LinkedList class can be used as a list, stack or queue.

**import** java.util.\*;

**public** **class** LinkedList1{

**public** **static** **void** main(String args[]){

  LinkedList<String> al=**new** LinkedList<String>();

  al.add("Ravi");

  al.add("Vijay");

  al.add("Ravi");

  al.add("Ajay");

  Iterator<String> itr=al.iterator();

**while**(itr.hasNext()){

   System.out.println(itr.next());

  }

 }

}

Output: Ravi

Vijay

Ravi

Ajay

# Difference between ArrayList and LinkedList

ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.

However, there are many differences between ArrayList and LinkedList classes that are given below.

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses a **dynamic array** to store the elements. | LinkedList internally uses a **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory. |
| 3) An ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |

# Java List

**List** in Java provides the facility to maintain the ordered collection. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.

The List interface is found in the java.util package and inherits the Collection interface. It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions. The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector. The ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

1. **import** java.util.\*;
2. **public** **class** ListExample1{
3. **public** **static** **void** main(String args[]){
4. //Creating a List
5. List<String> list=**new** ArrayList<String>();
6. //Adding elements in the List
7. list.add("Mango");
8. list.add("Apple");
9. list.add("Banana");
10. list.add("Grapes");
11. //Iterating the List element using for-each loop
12. **for**(String fruit:list)
13. System.out.println(fruit);
15. }
16. }

# Java HashSet

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

* HashSet stores the elements by using a mechanism called **hashing.**
* HashSet contains unique elements only.
* HashSet allows null value.
* HashSet class is non synchronized.
* HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
* HashSet is the best approach for search operations.
* The initial default capacity of HashSet is 16, and the load factor is 0.75.

**import** java.util.\*;

**class** HashSet2{

**public** **static** **void** main(String args[]){

1. //Creating HashSet and adding elements

  HashSet<String> set=**new** HashSet<String>();

  set.add("Ravi");

  set.add("Vijay");

  set.add("Ravi");

  set.add("Ajay");

1. //Traversing elements

  Iterator<String> itr=set.iterator();

**while**(itr.hasNext()){

   System.out.println(itr.next());

  }

 }

}

Ajay

Vijay

Ravi

# Java LinkedHashSet class

Java LinkedHashSet class is a Hashtable and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

The important points about Java LinkedHashSet class are:

* Java LinkedHashSet class contains unique elements only like HashSet.
* Java LinkedHashSet class provides all optional set operation and permits null elements.
* Java LinkedHashSet class is non synchronized.
* Java LinkedHashSet class maintains insertion order.

# Java TreeSet class

ava TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

* Java TreeSet class contains unique elements only like HashSet.
* Java TreeSet class access and retrieval times are quiet fast.
* Java TreeSet class doesn't allow null element.
* Java TreeSet class is non synchronized.
* Java TreeSet class maintains ascending order.

1. **import** java.util.\*;
2. **class** TreeSet1{
3. **public** **static** **void** main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> al=**new** TreeSet<String>();
6. al.add("Ravi");
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=al.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestCollection11)

Output:

Ajay

Ravi

Vijay

# 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.

## PriorityQueue class

The PriorityQueue class provides the facility of using queue. But it does not orders the elements in FIFO manner. It inherits AbstractQueue class.

1. **import** java.util.\*;
2. **class** TestCollection12{
3. **public** **static** **void** main(String args[]){
4. PriorityQueue<String> queue=**new** PriorityQueue<String>();
5. queue.add("Amit");
6. queue.add("Vijay");
7. queue.add("Karan");
8. queue.add("Jai");
9. queue.add("Rahul");
10. System.out.println("head:"+queue.element());
11. System.out.println("head:"+queue.peek());
12. System.out.println("iterating the queue elements:");
13. Iterator itr=queue.iterator();
14. **while**(itr.hasNext()){
15. System.out.println(itr.next());
16. }
17. queue.remove();
18. queue.poll();
19. System.out.println("after removing two elements:");
20. Iterator<String> itr2=queue.iterator();
21. **while**(itr2.hasNext()){
22. System.out.println(itr2.next());
23. }
24. }
25. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestCollection12)

Output:head:Amit

head:Amit

iterating the queue elements:

Amit

Jai

Karan

Vijay

Rahul

after removing two elements:

Karan

Rahul

Vijay

# Java Deque Interface

Java Deque Interface is a linear collection that supports element insertion and removal at both ends. Deque is an acronym for **"double ended queue".**

## ArrayDeque class

The ArrayDeque class provides the facility of using deque and resizable-array. It inherits AbstractCollection class and implements the Deque interface.

The important points about ArrayDeque class are:

* Unlike Queue, we can add or remove elements from both sides.
* Null elements are not allowed in the ArrayDeque.
* ArrayDeque is not thread safe, in the absence of external synchronization.
* ArrayDeque has no capacity restrictions.
* ArrayDeque is faster than LinkedList and Stack.

1. **import** java.util.\*;
2. **public** **class** ArrayDequeExample {
3. **public** **static** **void** main(String[] args) {
4. //Creating Deque and adding elements
5. Deque<String> deque = **new** ArrayDeque<String>();
6. deque.add("Ravi");
7. deque.add("Vijay");
8. deque.add("Ajay");
9. //Traversing elements
10. **for** (String str : deque) {
11. System.out.println(str);
12. }
13. }
14. }

Output:

Ravi

Vijay

Ajay

# Multithreading in Java

**Multithreading in**[**Java**](https://www.javatpoint.com/java-tutorial) is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

You **can perform many operations together, so it saves time**.

Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread

## Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

* Process-based Multitasking (Multiprocessing)

Each process has an address in memory. In other words, each process allocates a separate memory area.

A process is heavyweight.

Cost of communication between the process is high.

Switching from one process to another requires some time for saving and loading [registers](https://www.javatpoint.com/register-memory), memory maps, updating lists, etc.

* Thread-based Multitasking (Multithreading)

Threads share the same address space.

A thread is lightweight.

Cost of communication between the thread is low.

# Life cycle of a Thread (Thread States)



### 1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

### Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

### Running

The thread is in running state if the thread scheduler has selected it.

### Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

### Terminated

A thread is in terminated or dead state when its run() method exits.

# How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

### Thread class:

|  |
| --- |
| Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface. |

1. **class** Multi **extends** Thread{
2. **public** **void** run(){
3. System.out.println("thread is running...");
4. }
5. **public** **static** **void** main(String args[]){
6. Multi t1=**new** Multi();
7. t1.start();
8. }
9. }

### Runnable interface:

|  |
| --- |
| The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run(). |

1. **class** Multi3 **implements** Runnable{
2. **public** **void** run(){
3. System.out.println("thread is running...");
4. }
6. **public** **static** **void** main(String args[]){
7. Multi3 m1=**new** Multi3();
8. Thread t1 =**new** Thread(m1);
9. t1.start();
10. }
11. }

# Thread Scheduler in Java

**Thread scheduler** in java is the part of the JVM that decides which thread should run. There is no guarantee that which runnable thread will be chosen to run by the thread scheduler. Only one thread at a time can run in a single process. The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

### Difference between preemptive scheduling and time slicing

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

3.5M

# Sleep method in java

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

1. **class** TestSleepMethod1 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<5;i++){
4. **try**{Thread.sleep(500);}**catch**(InterruptedException e){System.out.println(e);}
5. System.out.println(i);
6. }
7. }
8. **public** **static** **void** main(String args[]){
9. TestSleepMethod1 t1=**new** TestSleepMethod1();
10. TestSleepMethod1 t2=**new** TestSleepMethod1();
12. t1.start();
13. t2.start();
14. }
15. }

# Can we start a thread twice

No. After starting a thread, it can never be started again. If you does so, an IllegalThreadStateException is thrown. In such case, thread will run once but for second time, it will throw exception.

1. TestThreadTwice1 t1=**new** TestThreadTwice1();
2. t1.start();
3. t1.start();

# The join() method

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

1. **ass** TestJoinMethod1 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<=5;i++){
4. **try**{
5. Thread.sleep(500);
6. }**catch**(Exception e){System.out.println(e);}
7. System.out.println(i);
8. }
9. }
10. **public** **static** **void** main(String args[]){
11. TestJoinMethod1 t1=**new** TestJoinMethod1();
12. TestJoinMethod1 t2=**new** TestJoinMethod1();
13. TestJoinMethod1 t3=**new** TestJoinMethod1();
14. t1.start();
15. **try**{
16. t1.join();
17. }**catch**(Exception e){System.out.println(e);}
19. t2.start();
20. t3.start();
21. }
22. }

# Daemon Thread in Java

**Daemon thread in java** is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

There are many java daemon threads running automatically e.g. gc, finalizer etc.

You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

## Points to remember for Daemon Thread in Java

* It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
* Its life depends on user threads.
* It is a low priority thread.

### Why JVM terminates the daemon thread if there is no user thread?

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

1. **public** **class** TestDaemonThread1 **extends** Thread{
2. **public** **void** run(){
3. **if**(Thread.currentThread().isDaemon()){//checking for daemon thread
4. System.out.println("daemon thread work");
5. }
6. **else**{
7. System.out.println("user thread work");
8. }
9. }
10. **public** **static** **void** main(String[] args){
11. TestDaemonThread1 t1=**new** TestDaemonThread1();//creating thread
12. TestDaemonThread1 t2=**new** TestDaemonThread1();
13. TestDaemonThread1 t3=**new** TestDaemonThread1();
15. t1.setDaemon(**true**);//now t1 is daemon thread
17. t1.start();//starting threads
18. t2.start();
19. t3.start();
20. }
21. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestDaemonThread1)

#### Output

daemon thread work

user thread work

user thread work

### What’s the difference between class lock and object lock?

**Class Lock**: In java, each and every class has a unique lock usually referred to as a class level lock. These locks are achieved using the keyword ‘static synchronized’ and can be used to make static data thread-safe. It is generally used when one wants to prevent multiple threads from entering a synchronized block.   
  
Example:

**public** **class** **ClassLevelLockExample**

{

**public** **void** **classLevelLockMethod**()

{

**synchronized** (ClassLevelLockExample.class)

{

//DO your stuff here

}

}

}

**Object Lock**: In java, each and every object has a unique lock usually referred to as an object-level lock. These locks are achieved using the keyword ‘synchronized’ and can be used to protect non-static data. It is generally used when one wants to synchronize a non-static method or block so that only the thread will be able to execute the code block on a given instance of the class.    
  
Example:

**public** **class** **ObjectLevelLockExample**

{

**public** **void** **objectLevelLockMethod**()

{

**synchronized** (**this**)

{

//DO your stuff here

}

}

}

### 6. What's the difference between User thread and Daemon thread?

User and Daemon are basically two types of thread used in Java by using a ‘Thread Class’.    
  
**User Thread (Non-Daemon Thread)**: In Java, user threads have a specific life cycle and its life is independent of any other thread. JVM (Java Virtual Machine) waits for any of the user threads to complete its tasks before terminating it. When user threads are finished, JVM terminates the whole program along with associated daemon threads.   
  
**Daemon Thread**: In Java, daemon threads are basically referred to as a service provider that provides services and support to user threads. There are basically two methods available in thread class for daemon thread: setDaemon() and isDaemon().   
  
**User Thread vs Daemon Thread**

| **User Thread** | **Daemon Thread** |
| --- | --- |
| JVM waits for user threads to finish their tasks before termination. | JVM does not wait for daemon threads to finish their tasks before termination. |
| These threads are normally created by the user for executing tasks concurrently. | These threads are normally created by JVM. |
| They are used for critical tasks or core work of an application. | They are not used for any critical tasks but to do some supporting tasks. |
| These threads are referred to as high-priority tasks, therefore are required for running in the foreground. | These threads are referred to as low priority threads, therefore are especially required for supporting background tasks like garbage collection, releasing memory of unused objects, etc. |

76

### Why wait(), notify(), and notifyAll() methods are present in Object class?

We know that every object has a monitor that allows the thread to hold a lock on the object. But the thread class doesn't contain any monitors. Thread usually waits for the object’s monitor (lock) by calling the wait() method on an object, and notify other threads that are waiting for the same lock using notify() or notifyAll() method.  Therefore, these three methods are called on objects only and allow all threads to communicate with each that are created on that object.

### Explain thread pool?

A Thread pool is simply a collection of pre-initialized or worker threads at the start-up that can be used to execute tasks and put back in the pool when completed. It is referred to as pool threads in which a group of fixed-size threads is created.  By reducing the number of application threads and managing their lifecycle, one can mitigate the issue of performance using a thread pool. Using threads, performance can be enhanced and better system stability can occur. To create the thread pools, java.util.concurrent.Executors class usually provides factory methods.

### 14. What’s the purpose of the join() method?

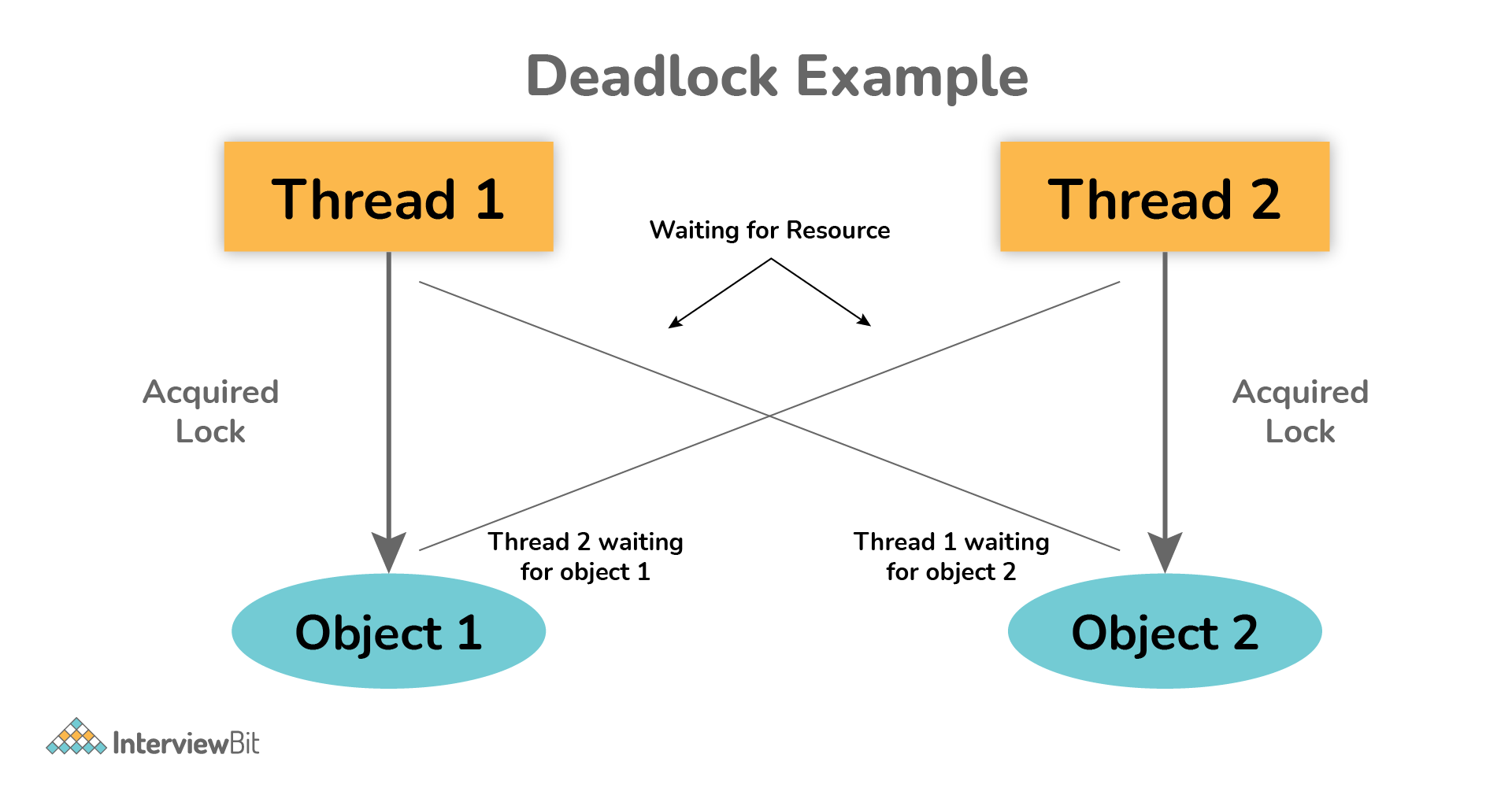
**join()** method is generally used to pause the execution of a current thread unless and until the specified thread on which join is called is dead or completed. To stop a thread from running until another thread gets ended, this method can be used. It joins the start of a thread execution to the end of another thread’s execution. It is considered the final method of a thread class.

### 15. What do you mean by garbage collection?

Garbage collection is basically a process of managing memory automatically. It uses several GC algorithms among which the popular one includes Mark and Sweep. The process includes three phases i.e., marking, deletion, and compaction/copying. In simple words, a garbage collector finds objects that are no longer required by the program and then delete or remove these unused objects to free up the memory space.

### 16. Explain the meaning of the deadlock and when it can occur?

Deadlock, as the name suggests, is a situation where multiple threads are blocked forever. It generally occurs when multiple threads hold locks on different resources and are waiting for other resources to complete their task.



The above diagram shows a deadlock situation where two threads are blocked forever.  Thread 1 is holding Object 1 but needs object 2 to complete processing whereas Thread 2 is holding Object 2 but needs object 1 first. In such conditions, both of them will hold lock forever and will never complete tasks.

### 17. Explain volatile variables in Java?

A volatile variable is basically a keyword that is used to ensure and address the visibility of changes to variables in multithreaded programming. This keyword cannot be used with classes and methods, instead can be used with variables. It is simply used to achieve thread-safety. If you mark any variable as volatile, then all the threads can read its value directly from the main memory rather than CPU cache, so that each thread can get an updated value of the variable.

### 18. How do threads communicate with each other?

Threads can communicate using three methods i.e., wait(), notify(), and notifyAll().

### 19. Can two threads execute two methods (static and non-static concurrently)?

Yes, it is possible. If both the threads acquire locks on different objects, then they can execute concurrently without any problem.

### 20. What is the purpose of the finalize() method?

Finalize() method is basically a method of Object class specially used to perform cleanup operations on unmanaged resources just before garbage collection. It is not at all intended to be called a normal method. After the complete execution of finalize() method, the object gets destroyed automatically.

### What is the synchronization process? Why use it?

Synchronization is basically a process in java that enables a simple strategy for avoiding thread interference and memory consistency errors. This process makes sure that resource will be only used one thread at a time when one thread tries to access a shared resource. It can be achieved in three different ways as given below:

* By the synchronized method
* By synchronized block
* By static synchronization

Syntax:

synchronized (object)

{

//statement to be synchronized

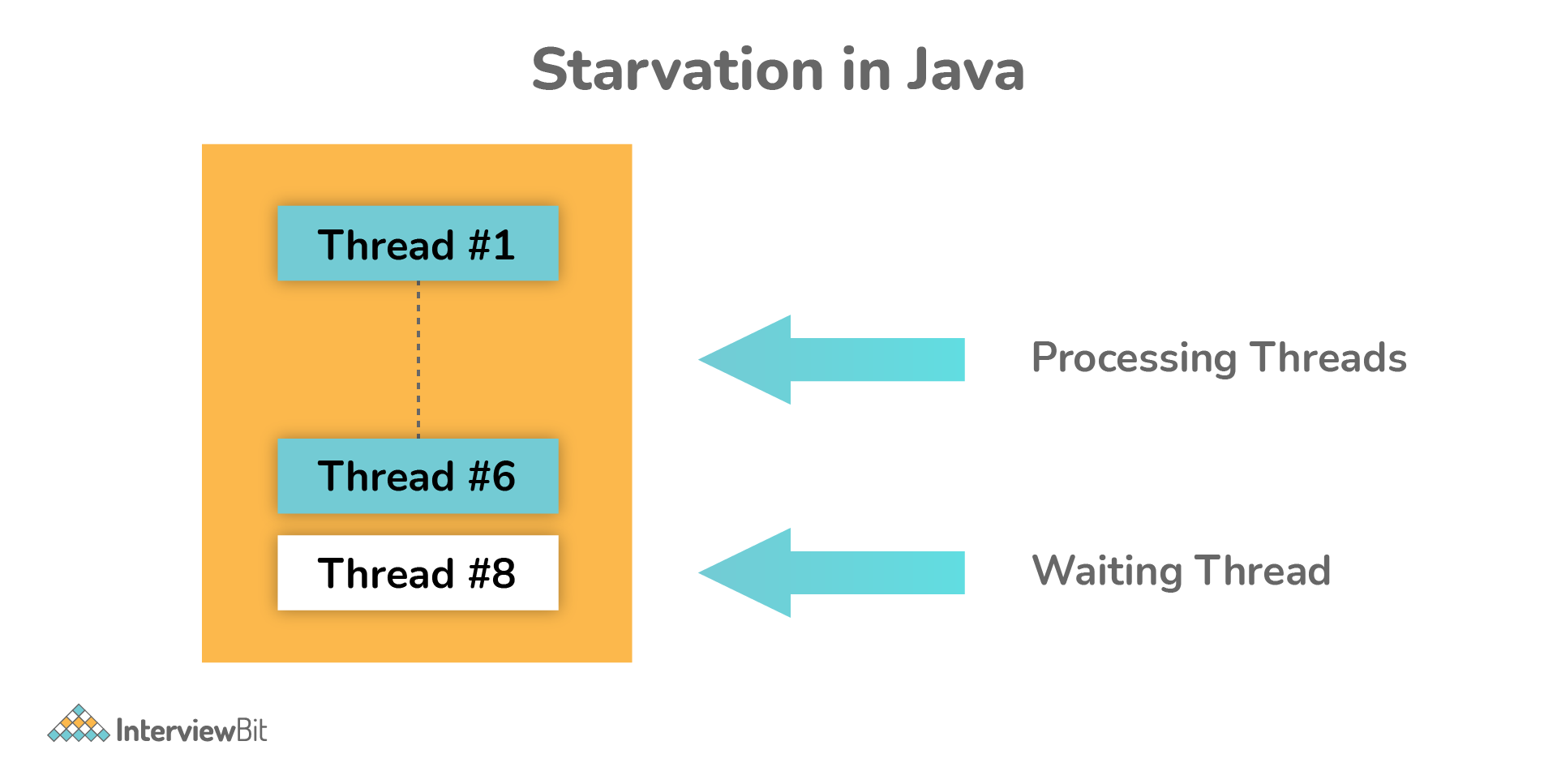
}

### 22. What is synchronized method and synchronized block? Which one should be preferred?

**Synchronized Method**: In this method, the thread acquires a lock on the object when they enter the synchronized method and releases the lock either normally or by throwing an exception when they leave the method.  No other thread can use the whole method unless and until the current thread finishes its execution and release the lock. It can be used when one wants to lock on the entire functionality of a particular method.   
  
**Synchronized Block**: In this method, the thread acquires a lock on the object between parentheses after the synchronized keyword, and releases the lock when they leave the block. No other thread can acquire a lock on the locked object unless and until the synchronized block exists. It can be used when one wants to keep other parts of the programs accessible to other threads.  
   
Synchronized blocks should be preferred more as it boosts the performance of a particular program. It only locks a certain part of the program (critical section) rather than the entire method and therefore leads to less contention.

### 23. What is thread starvation?

Thread starvation is basically a situation or condition where a thread won’t be able to have regular access to shared resources and therefore is unable to proceed or make progress. This is because other threads have high priority and occupy the resources for too long. This usually happens with low-priority threads that do not get CPU for its execution to carry on.

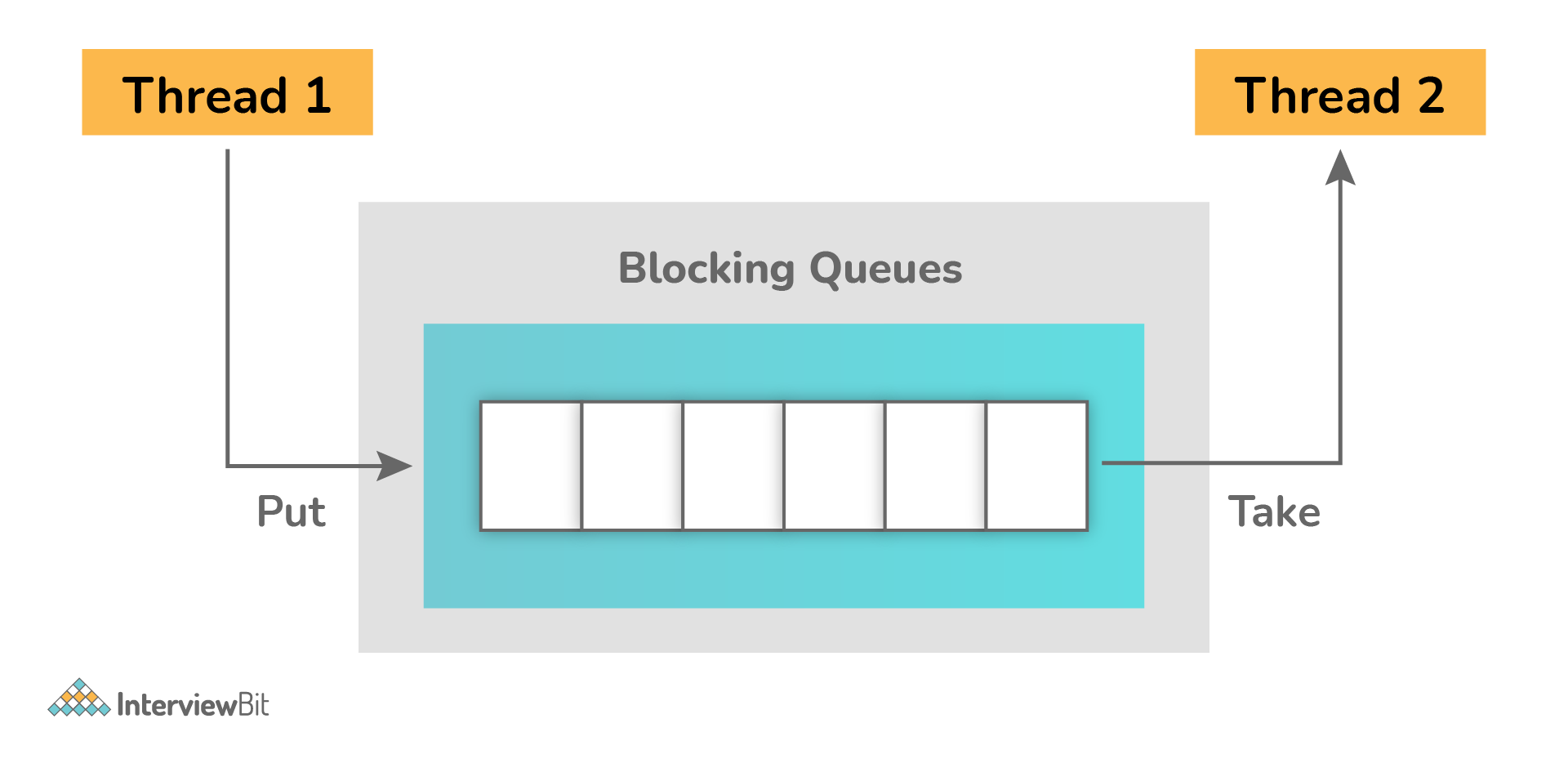


### 24. What is Livelock? What happens when it occurs?

Similar to deadlock, livelock is also another concurrency problem. In this case, the state of threads changes between one another without making any progress. Threads are not blocked but their execution is stopped due to the unavailability of resources.

### 25. What is BlockingQueue?

BlockingQueue basically represents a queue that is thread-safe. Producer thread inserts resource/element into the queue using put() method unless it gets full and consumer thread takes resources from the queue using take() method until it gets empty. But if a thread tries to dequeue from an empty queue, then a particular thread will be blocked until some other thread inserts an item into the queue, or if a thread tries to insert an item into a queue that is already full, then a particular thread will be blocked until some threads take away an item from the queue.



**Example**:

**package** org.arpit.java2blog;

**import** java.util.concurrent.ArrayBlockingQueue;

**import** java.util.concurrent.BlockingQueue;

**public** **class** **BlockingQueuePCExample** {

**public** **static** **void** **main**(String[] args) {

BlockingQueue<String> queue=**new** ArrayBlockingQueue<>(5);

Producer producer=**new** Producer(queue);

Consumer consumer=**new** Consumer(queue);

Thread producerThread = **new** Thread(producer);

Thread consumerThread = **new** Thread(consumer);

producerThread.start();

consumerThread.start();

}

**static** **class** **Producer** **implements** **Runnable** {

BlockingQueue<String> queue=**null**;

**public** **Producer**(BlockingQueue queue) {

**super**();

**this**.queue = queue;

}

@Override

**public** **void** **run**() {

**try** {

System.out.println("Producing element 1");

queue.put("Element 1");

Thread.sleep(1000);

System.out.println("Producing element 2");

queue.put("Element 2");

Thread.sleep(1000);

System.out.println("Producing element 3");

queue.put("Element 3");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**static** **class** **Consumer** **implements** **Runnable** {

BlockingQueue<String> queue=**null**;

**public** **Consumer**(BlockingQueue queue) {

**super**();

**this**.queue = queue;

}

@Override

**public** **void** **run**() {

**while**(**true**)

{

**try** {

System.out.println("Consumed "+queue.take());

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

}

**Output**:

Producing element 1

Consumed Element 1

Producing element 2

Consumed Element 2

Producing element 3

Consumed Element 3

### What is a shutdown hook?

A shutdown hook is simply a thread that is invoked implicitly before JVM shuts down. It is one of the most important features of JVM because it provides the capacity to do resource cleanup or save application state JVM shuts down.  By calling the halt(int) method of the Runtime class, the shutdown hook can be stopped. Using the following method, one can add a shutdown hook.

**public** **void** **addShutdownHook**(Thread hook){}

Runtime r=Runtime.getRuntime();

r.addShutdownHook(**new** MyThread());

### 32. What is busy spinning?

Busy Spinning, also known as Busy-waiting, is a technique in which one thread waits for some condition to happen, without calling wait or sleep methods and releasing the CPU. In this condition, one can pause a thread by making it run an empty loop for a certain time period, and it does not even give CPY control. Therefore, it is used to preserve CPU caches and avoid the cost of rebuilding cache.

# Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

### Advantage of Garbage Collection

* It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
* It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

# Java Runtime class

**Java Runtime** class is used to interact with java runtime environment. Java Runtime class provides methods to execute a process, invoke GC, get total and free memory etc. There is only one instance of java.lang.Runtime class is available for one java application.

The **Runtime.getRuntime()** method returns the singleton instance of Runtime class.

## Java Runtime exec() method

1. **public** **class** Runtime1{
2. **public** **static** **void** main(String args[])**throws** Exception{
3. Runtime.getRuntime().exec("notepad");//will open a new notepad
4. }
5. }

## How to shutdown system in Java

You can use shutdown -s command to shutdown system. For windows OS, you need to provide full path of shutdown command e.g. c:\\Windows\\System32\\shutdown.

Here you can use -s switch to shutdown system, -r switch to restart system and -t switch to specify time delay.

# Synchronization in Java

Synchronization in java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

1. Process Synchronization
2. Thread Synchronization

1.Mutual Exclusive

* 1. Synchronized method.
  2. Synchronized block.
  3. static synchronization.

2.Cooperation (Inter-thread communication in java)

History of Java

**Next**

**Stay**

### Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

# Synchronized Block in Java

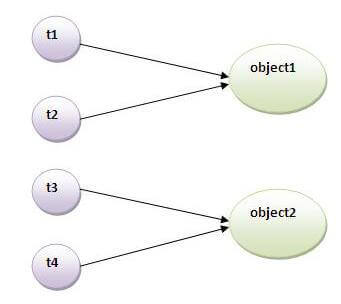
Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

# Static Synchronization

If you make any static method as synchronized, the lock will be on the class not on object.



### Problem without static synchronization

Suppose there are two objects of a shared class(e.g. Table) named object1 and object2.In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock.But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock.I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem.

### Example of static synchronization

In this example we are applying synchronized keyword on the static method to perform static synchronization.

1. **class** Table{
3. **synchronized** **static** **void** printTable(**int** n){
4. **for**(**int** i=1;i<=10;i++){
5. System.out.println(n\*i);
6. **try**{
7. Thread.sleep(400);
8. }**catch**(Exception e){}
9. }
10. }
11. }
13. **class** MyThread1 **extends** Thread{
14. **public** **void** run(){
15. Table.printTable(1);
16. }
17. }
19. **class** MyThread2 **extends** Thread{
20. **public** **void** run(){
21. Table.printTable(10);
22. }
23. }
25. **class** MyThread3 **extends** Thread{
26. **public** **void** run(){
27. Table.printTable(100);
28. }
29. }



34. **class** MyThread4 **extends** Thread{
35. **public** **void** run(){
36. Table.printTable(1000);
37. }
38. }
40. **public** **class** TestSynchronization4{
41. **public** **static** **void** main(String t[]){
42. MyThread1 t1=**new** MyThread1();
43. MyThread2 t2=**new** MyThread2();
44. MyThread3 t3=**new** MyThread3();
45. MyThread4 t4=**new** MyThread4();
46. t1.start();
47. t2.start();
48. t3.start();
49. t4.start();
50. }
51. }

# Deadlock in java

Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



# Inter-thread communication in Java

**Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of **Object class**:

* wait()
* notify()
* notifyAll()

### 1) wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

3.9M

95

OOPs Concepts in Java

**Next**

**Stay**

|  |  |
| --- | --- |
| **Method** | **Description** |
| public final void wait()throws InterruptedException | waits until object is notified. |
| public final void wait(long timeout)throws InterruptedException | waits for the specified amount of time. |

### 2) notify() method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

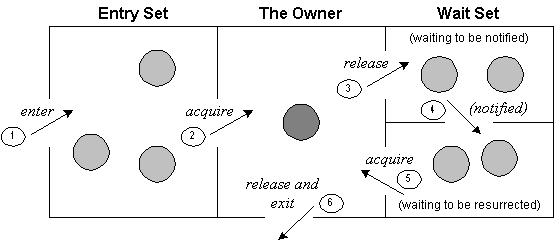
public final void notify()

### 3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor. Syntax:

public final void notifyAll()

### Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

### Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

### Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

|  |  |
| --- | --- |
| **wait()** | **sleep()** |
| wait() method releases the lock | sleep() method doesn't release the lock. |
| is the method of Object class | is the method of Thread class |
| is the non-static method | is the static method |
| is the non-static method | is the static method |
| should be notified by notify() or notifyAll() methods | after the specified amount of time, sleep is completed. |

### Example of inter thread communication in java

# interrupting a Thread:

|  |
| --- |
| If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true. Let's first see the methods provided by the Thread class for thread interruption. |

## The 3 methods provided by the Thread class for interrupting a thread

|  |
| --- |
| * **public void interrupt()** * **public static boolean interrupted()** * **public boolean isInterrupted()** |