**Java SE 8 New Features?**

* Lambda Expressions
* Functional Interfaces
* Stream API
* Date and Time API
* Interface Default Methods and Static Methods
* Spliterator
* Method and Constructor References
* Collections API Enhancements
* Concurrency Utils Enhancements
* Fork/Join Framework Enhancements
* Internal Iteration
* Parallel Array and Parallel Collection Operations
* Optional
* Type Annotations and Repeatable Annotations
* Method Parameter Reflection
* Base64 Encoding and Decoding
* IO and NIO2 Enhancements
* Nashorn JavaScript Engine
* javac Enhancements
* JVM Changes
* Java 8 Compact Profiles: compact1,compact2,compact3
* JDBC 4.2
* JAXP 1.6
* Java DB 10.10
* Networking
* Security Changes

**Advantages of Java SE 8 New Features?**

We can get the following benefits from Java SE 8 New Features:

* More Concise and Readable code
* More Reusable code
* More Testable and Maintainable Code
* Highly Concurrent and Highly Scalable Code
* Write Parallel Code
* Write Database Like Operations
* Better Performance Applications
* More Productive code

### What is Lambda Expression?

Lambda Expression is an anonymous function which accepts a set of input parameters and returns results.

Lambda Expression is a block of code without any name, with or without parameters and with or without results. This block of code is executed on demand.

**What are the three parts of a Lambda Expression? What is the type of Lambda Expression?**

A Lambda Expression contains 3 parts:

* Parameter List

A Lambda Expression can contain zero or one or more parameters. It is optional.

* Lambda Arrow Operator

“->” is known as Lambda Arrow operator. It separates parameters list and body.

* Lambda Expression Body

The type of “Journal Dev” is java.lang.String. The type of “true” is Boolean. In the same way, what is the type of a Lambda Expression?  
The Type of a Lambda Expression is a Functional Interface.

Example:- What is the type of the following Lambda Expression?

|  |  |
| --- | --- |
| 1 | () -> System.out.println("Hello World"); |

This Lambda Expression does not have parameters and does return any results. So it’s type is “java.lang.Runnable” Functional Interface.

### What is a Functional Interface? What is SAM Interface?

A Functional Interface is an interface, which contains one and only one abstract method. Functional Interface is also know as SAM Interface because it contains only one abstract method.

SAM Interface stands for Single Abstract Method Interface. Java SE 8 API has defined many Functional Interfaces.

**Is is possible to define our own Functional Interface? What is @FunctionalInterface? What are the rules to define a Functional Interface?**

Yes, it is possible to define our own Functional Interfaces. We use Java SE 8’s @FunctionalInterface annotation to mark an interface as Functional Interface.

We need to follow these rules to define a Functional Interface:

* Define an interface with one and only one abstract method.
* We cannot define more than one abstract method.
* Use @FunctionalInterface annotation in interface definition.
* We can define any number of other methods like Default methods, Static methods.
* If we override java.lang.Object class’s method as an abstract method, which does not count as an abstract method.

### Is @FunctionalInterface annotation mandatory to define a Functional Interface? What is the use of @FunctionalInterface annotation? Why do we need Functional Interfaces in Java?

It is not mandatory to define a Functional Interface with @FunctionalInterface annotation. If we don’t want, We can omit this annotation. However, if we use it in Functional Interface definition, Java Compiler forces to use one and only one abstract method inside that interface.

Why do we need Functional Interfaces? The type of a Java SE 8’s Lambda Expression is a Functional Interface. Whereever we use Lambda Expressions that means we are using Functional Interfaces.

**When do we go for Java 8 Stream API? Why do we need to use Java 8 Stream API in our projects?**

When our Java project wants to perform the following operations, it’s better to use Java 8 Stream API to get lot of benefits:

* When we want perform Database like Operations. For instance, we want perform groupby operation, orderby operation etc.
* When want to Perform operations Lazily.
* When we want to write Functional Style programming.
* When we want to perform Parallel Operations.
* When want to use Internal Iteration
* When we want to perform Pipelining operations.
* When we want to achieve better performance.

### What is Spliterator in Java SE 8?Differences between Iterator and Spliterator in Java SE 8?

Spliterator stands for Splitable Iterator. It is newly introduced by Oracle Corporation as part Java SE 8.  
Like Iterator and ListIterator, It is also one of the Iterator interface.

|  |  |  |
| --- | --- | --- |
| S.NO. | SPLITERATOR | ITERATOR |
| 1. | It is introduced in Java SE 8. | It is available since Java 1.2. |
| 2. | Splitable Iterator | Non-Splitable Iterator |
| 3. | It is used in Stream API. | It is used for Collection API. |
| 4. | It uses Internal Iteration concept to iterate Streams. | It uses External Iteration concept to iterate Collections. |
| 5. | We can use Spliterator to iterate Streams in Parallel and Sequential order. | We can use Spliterator to iterate Collections only in Sequential order. |
| 6. | We can get Spliterator by calling spliterator() method on Stream Object. | We can get Iterator by calling iterator() method on Collection Object. |
| 7. | Important Method: tryAdvance() | Important Methods: next(), hasNext( |

**What is Optional in Java 8? What is the use of Optional?Advantages of Java 8 Optional?**

**Optional:**  
Optional is a final Class introduced as part of Java SE 8. It is defined in java.util package.

It is used to represent optional values that is either exist or not exist. It can contain either one value or zero value. If it contains a value, we can get it. Otherwise, we get nothing.

It is a bounded collection that is it contains at most one element only. It is an alternative to “null” value.

**Main Advantage of Optional is:**

* It is used to avoid null checks.
* It is used to avoid “NullPointerException”.

### What is Type Inference? Is Type Inference available in older versions like Java 7 and Before 7 or it is available only in Java SE 8?

Type Inference means determining the Type by compiler at compile-time.

It is not new feature in Java SE 8. It is available in Java 7 and before Java 7 too.

**Before Java 7:-**  
Let us explore Java arrays. Define a String of Array with values as shown below:

|  |  |
| --- | --- |
| 1 | String str[] = { "Java 7", "Java 8", "Java 9" }; |

Here we have assigned some String values at right side, but not defined it’s type. Java Compiler automatically infers it’s type and creates a String of Array.

**Java 7:-**  
Oracle Corporation has introduced “Diamond Operator” new feature in Java SE 7 to avoid unnecessary Type definition in Generics.

|  |  |
| --- | --- |
| 1 | Map<String,List<Customer>> customerInfoByCity = new HashMap<>(); |

Here we have not defined Type information at right side, simply defined Java SE 7’s Diamond Operator “”.

**Java SE 8:-**  
Oracle Corporation has enhanced this Type Inference concept a lot in Java SE 8. We use this concept to define Lambda Expressions, Functions, Method References etc.

|  |  |
| --- | --- |
| 1 | Integer add = (a,b) -> a + b; |

Here Java Compiler observes the type definition available at left-side and determines the type of Lambda Expression parameters a and b is Integer.

**Difference between Predicate, Supplier and Consumer ?**  
  
Ans. Predicate represents an anonymous function that accepts one argument and produces a result.  
  
Supplier represents an anonymous function that accepts no argument and produces a result.  
  
Consumer represents an anonymous function that accepts an argument and produces no result.

**What does the following lambda expression means ?**  
  
helloJava8 ( x-> x%2 )  
  
  
Ans. helloJava8 receives an Integer as argument and then returns the modulus of that Integer.  
  
**Q5.  What are Default Methods ?**  
  
Ans. With Java 8, We can provide method definitions in the Interfaces that gets carried down the classes implementing that interface in case they are not overridden by the Class. Keyword "default" is used to mark the default method.

**Can we have a default method definition in the interface without specifying the keyword "default" ?**  
  
Ans. No. Compiler complains that its an abstract method and hence shouldn't have the body.

**Can a class implement two Interfaces having default method with same name and signature ?**  
  
public interface DefaultMethodInterface {  
    default public void defaultMethod(){  
       System.out.println("DefaultMethodInterface");          
    }  
}  
  
public interface DefaultMethodInterface2 {            
        default public void defaultMethod(){  
               System.out.println("DefaultMethodInterface2");          
        }  
}  
  
public class HelloJava8 implements DefaultMethodInterface,DefaultMethodInterface2 {  
   public static void main(String[] args){     
           DefaultMethodInterface defMethIn = new HelloJava8();  
           defMethIn.defaultMethod();  
    }  
}  
  
Ans. No. Compiler gives error saying "Duplicate Default Methods"

**What If we make the method as abstract in another Interface ?**  
  
public interface DefaultMethodInterface {  
    default public void defaultMethod(){  
       System.out.println("DefaultMethodInterface");          
    }  
}  
  
public interface DefaultMethodInterface2 {            
        public void defaultMethod(){  
               System.out.println("DefaultMethodInterface2");          
        }  
}  
  
public class HelloJava8 implements DefaultMethodInterface,DefaultMethodInterface2 {  
   public static void main(String[] args){     
           DefaultMethodInterface defMethIn = new HelloJava8();  
           defMethIn.defaultMethod();  
    }  
}  
  
Ans. Even then the Compiler will give error saying that there is a conflict.

**What if we override the conflicting method in the Class ?**  
  
public interface DefaultMethodInterface {  
    default public void defaultMethod(){  
       System.out.println("DefaultMethodInterface");          
    }  
}  
  
public interface DefaultMethodInterface2 {            
        default public void defaultMethod(){  
               System.out.println("DefaultMethodInterface2");          
        }  
}  
  
public class HelloJava8 implements DefaultMethodInterface,DefaultMethodInterface2 {  
   public static void main(String[] args){     
           DefaultMethodInterface defMethIn = new HelloJava8();  
           defMethIn.defaultMethod();  
    }  
  
   public void defaultMethod(){  
       System.out.println("HelloJava8");   
   }  
}  
  
Ans. There won't be any error and upon execution the overriding class method will be executed.

**What will happen if there is a default method conflict as mentioned above and we have specified the same signature method in the base class instead of overriding in the existing class ?**  
  
Ans. There won't be any problem as the Base class method will have precedence over the Interface Default methods.

**If there is a conflict between Base Class Method definition and Interface Default method definition, Which definition is Picked ?**  
  
Ans. Base Class Definition.

**If a method definition has been specified in Class , its Base Class , and the interface which the class is implementing, Which definition will be picked if we try to access it using Interface Reference  and Class object ?**  
  
Ans. Class method definition is overriding both the definitions and hence will be picked.

**If a method definition has been specified in the Base Class and the interface which the class is implementing, Which definition will be picked if we try to access it using Interface Reference and Class object ?**  
  
Ans. Base Class Definition will have precedence over the Interface Default method definition.

**Can we use static method definitions in Interfaces ?**  
  
Ans. Yes, Effective Java 8.  
  
**Q15.  Can we access Interface static method using Interface references ?**  
  
Ans. No, only using Interface Name.

**Can we have default method with same name and signature in the derived Interface as the static method in base Interface and vice versa ?**  
  
Ans. Yes , we can do that as static methods are not accessible using references and hence cannot lead to conflict. We cannot do inverse as Default methods cannot be overridden with the static methods in derived interface.

**Name few Java 8 annotations ?**  
  
Ans.   
  
@FunctionalInterface annotation, introduced in Java SE 8, indicates that the type declaration is intended to be a functional interface, as defined by the Java Language Specification.  
  
@Repeatable annotation, introduced in Java SE 8, indicates that the marked annotation can be applied more than once to the same declaration or type use. For more information, see Repeating Annotations.

**Difference between final and effectively final ? Why is effectively final even required ?**  
  
Ans. Final variable means a variable that has been declared final and hence cannot be changed after initialization.  
  
Effective final means a variable that has not been declared final but haven't been reassigned the value after initialization.  
  
First is the regulation that restricts the reassignment and will raise a compilation error if we try to do so. Second is the outcome without the restriction.  
  
Effective Final is the eventual treatment of the variable that is required for many features. For eq - Java 8 requires that  
  
local variables referenced from a lambda expression must be final or effectively final  
  
It means all local referenced from lambda expressions must be such that their value shouldn't be changed after initialization whether declared final or not.

**Difference between DoubleSummaryStatistics , IntSummaryStatistics and LongSummaryStatistics ?**  
  
Ans. They all does the same task i.e to compute statistical information on the stream of data. They differ by the way they store the statistical information as they expect a different data type of the values being used.   
  
IntSummaryStatistics and LongSummaryStatistics expect non floating point values and hence stores the statistical information like min,max and sum as non floating values ( int or long ) whereas DoubleSummaryStatistics stores these information as floating value.

**What is StringJoiner ?**  
  
Ans. StringJoiner is a util method to construct a string with desired delimiter. This has been introduced with wef from Java 8.  
  
Sample Code  
  
StringJoiner strJoiner = new StringJoiner(".");  
strJoiner.add("Buggy").add("Bread");  
System.out.println(strJoiner); // prints Buggy.Bread

**What is the use of Optional ?**  
  
Ans. Optional is a good way to protect application from runtime nullPointerException in case the the absent value has been represented as null. So basically Optional class provides the type checking during compile time and hence will never result in NPE.  
  
For ex -  
  
List<Optional<Employee>> intList = new ArrayList<Optional<Employee>>();  
intList.add(Optional.empty());  
intList.add(Optional.of(new Employee("abc")));  
intList.add(Optional.of(new Employee("xyz")));  
intList.add(Optional.of(new Employee("123")));  
System.out.println(intList.get(0).getName());  
  
So Now , even when the first list element is empty, this code will never throw an NullPointerException.

**What is Internal Iteration in Java SE 8?**

Before Java 8, We don’t Internal Iteration concept. Java 8 has introduced a new feature known as “Internal Iteration”. Before Java 8, Java Language has only External Iteration to iterate elements of an Aggregated Object like Collections, Arrays etc.

Internal Iteration means “Iterating an Aggregated Object elements one by one internally by Java API”. Instead of Java Application do iteration externally, We ask Java API to do this job internally.

**Differences between External Iteration and Internal Iteration?**

|  |  |  |
| --- | --- | --- |
| S.NO. | EXTERNAL ITERATION | INTERNAL ITERATION |
| 1. | Available before Java 8 too. | It is introduced in Java SE 8 |
| 2. | Iterating an Aggregated Object elements externally. | Iterating an Aggregated Object elements internally (background). |
| 3. | Iterate elements by using for-each loop and Iterators like Enumeration, Iterator, ListIterator. | Iterate elements by using Java API like “forEach” method. |
| 4. | Iterating elements in Sequential and In-Order only. | Not required to iterate elements in Sequential order. |
| 5. | It follows OOP approach that is Imperative Style. | It follows Functional Programming approach that is Declarative Style. |
| 6. | It does NOT separate responsibilities properly that is, it defines both “What is to be done” and “How it is to be done”. | It defines only “What is to be done”. No need to worry about “How it is to be done”. Java API takes care about “How to do”. |
| 7. | Less Readable Code. | More Readable code. |

**What are the major drawbacks of External Iteration?**

External Iteration has the following drawbacks:

* We need to write code in Imperative Style.
* There is no clear separation of Responsibilities. Tightly-Coupling between “What is to be done” and “How it is to be done” code.
* Less Readable Code.
* More Verbose and Boilerplate code.
* We have to iterate elements in Sequential order only.
* It does not support Concurrency and Parallelism properly.

**What are the major advantages of Internal Iteration over External Iteration?**

Compare to External Iteration, Internal Iteration has the following advantages:

* As it follows Functional Programming style, we can write Declarative Code.
* More Readable and concise code.
* Avoids writing Verbose and Boilerplate code
* No need to iterate elements in Sequential order.
* It supports Concurrency and Parallelism properly.
* We can write Parallel code to improve application performance.
* Clear separation of Responsibilities. Loosely-Coupling between “What is to be done” and “How it is to be done” code.
* We need to write code only about “What is to be done” and Java API takes care about “How it is to be done” code.

**What is the major drawback of Internal Iteration over External Iteration?**

Compare to External Iteration, Internal Iteration has one major drawback:

* In Internal Iteration, as Java API takes care about Iterating elements internally, we do NOT have control over Iteration.

**What is the major advantage of External Iteration over Internal Iteration?**

Compare to Internal Iteration, External Iteration has one major advantage:

* In External Iteration, as Java API does NOT take care about Iterating elements, we have much control over Iteration.

**When do we need to use Internal Iteration? When do we need to use External Iteration?**

We need to understand the situations to use either Internal Iteration or External Iteration.

* When we need more control over Iteration, we can use External Iteration.
* When we do NOT need more control over Iteration, we can use Internal Iteration.
* When we need to develop Highly Concurrency and Parallel applications and we , we should use Internal Iteration.

**Differences between Intermediate Operations and Terminal Operations of Java 8’s Stream API?**

|  |  |  |
| --- | --- | --- |
| S.NO. | STREAM INTERMEDIATE OPERATIONS | STREAM TERMINAL OPERATIONS |
| 1. | Stream Intermediate operations are not evaluated until we chain it with Stream Terminal Operation. | Stream Terminal Operations are evaluated on it’s own. No need other operations help. |
| 2. | The output of Intermediate Operations is another Stream. | The output of Intermediate Operations is Not a Stream. Something else other than a Stream. |
| 3. | Intermediate Operations are evaluated Lazily. | Terminal Operations are evaluated Eagerly. |
| 4. | We can chain any number of Stream Intermediate Operations. | We can NOT chain Stream Terminal Operations. |
| 5. | We can use any number of Stream Intermediate Operations per Statement. | We can use only one Stream Terminal Operation per Statement. |

**Is it possible to provide method implementations in Java Interfaces? If possible, how do we provide them?**

In Java 7 or earlier, It is not possible to provide method implementations in Interfaces. Java 8 on-wards, it is possible.

In Java SE 8, We can provide method implementations in Interfaces by using the following two new concepts:

* Default Methods
* Static Methods

**What is a Default Method? Why do we need Default methods in Java 8 Interfaces?**

A Default Method is a method which is implemented in an interface with “default” keyword. It’s new featured introduced in Java SE 8.

**We need Default Methods because of the following reasons:**

* It allow us to provide method’s implementation in Interfaces.
* To add new Functionality to Interface without breaking the Classes which implement that Interface.
* To provide elegant Backwards Compatibility Feature.
* To ease of extend the existing Functionality.
* To ease of Maintain the existing Functionality.

**What is a Static Method? Why do we need Static methods in Java 8 Interfaces?**

A Static Method is an Utility method or Helper method, which is associated to a class (or interface). It is not associated to any object.

**We need Static Methods because of the following reasons:**

* We can keep Helper or Utility methods specific to an interface in the same interface rather than in a separate Utility class.
* We do not need separate Utility Classes like Collections, Arrays etc to keep Utility methods.
* Clear separation of Responsibilities. That is we do not need one Utility class to keep all Utility methods of Collection API like Collections etc.
* Easy to extend the API.
* Easy to Maintain the API.

**Differences between Functional Programming and Object-Oriented Programming?**

|  |  |
| --- | --- |
| FUNCTIONAL PROGRAMMING | OOP |
| Does not exist State | Exists State |
| Uses Immutable data | Uses Mutable data |
| It follows Declarative Programming Model | It follows Imperative Programming Model |
| Stateless Programming Model | Stateful Programming Model |
| Main Fcous on: “What you are doing” | Main focus on “How you are doing” |
| Good for Parallel (Concurrency) Programming | Poor for Parallel (Concurrency) Programming |
| Good for BigData processing and analysis | NOT Good for BigData processing and analysis |
| Supports pure Encapsulation | It breaks Encapsulation concept |
| Functions with No-Side Effects | Methods with Side Effects |
| Functions are first-class citizens | Objects are first-class citizens |
| Primary Manipulation Unit is “Function” | Primary Manipulation Unit is Objects(Instances of Classes) |
| Flow Controls: Function calls, Function Calls with Recursion | Flow Controls: Loops, Conditional Statements |
| It uses “Recursion” concept to iterate Collection Data. | It uses “Loop” concept to iterate Collection Data. For example:-For-each loop in Java |
| Order of execution is less importance. | Order of execution is must and very important. |
| Supports both “Abstraction over Data” and “Abstraction over Behavior”. | Supports only “Abstraction over Data”. |
| We use FP when we have few Things with more operations. | We use OOP when we have few Operations with more Things. For example: Things are classes and Operations are Methods in Java. |

NOTE:- For more information about FP, IP and OOP comparisons, Please go through my previous post at: “[Compare FP, OOP(IP)](http://www.journaldev.com/8693/functional-imperative-object-oriented-programming-comparison)”

**Explain issues of Old Java Date API? What are the advantages of Java 8’s Date and Time API over Old Date API and Joda Time API?**

Java’s OLD Java Date API means Date API available before Java SE 8 that is Date, Calendar, SimpleDateFormat etc.

Java’s Old Date API has the following Issues or Drawbacks compare to Java 8’s Date and Time API and Joda Time API.

* Most of the API is deprecated.
* Less Readability.
* java.util.Date is Mutable and not Thread-Safe.
* java.text.SimpleDateFormat is not Thread-Safe.
* Less Performance.

Java SE 8’s Date and Time API has the following Advantages compare to Java’s OLD Date API.

* Very simple to use.
* Human Readable Syntax that is More Readability.
* All API is Thread-Safe.
* Better Performance.

**Why do we need new Date and Time API in Java SE 8?Explain how Java SE 8 Data and Time API solves issues of Old Java Date API?**

We need Java 8’s Date and Time API to develop Highly Performance, Thread-Safe and Highly Scalable Java Applications.

Java 8’s Date and Time API solves all Java’s Old Date API issues by following Immutability and Thread-Safety principles.

**What are the Differences between Java’s OLD Java Date API and Java 8’s Date and Time API?**

**Differences between Java’s OLD Java Date API and Java 8’s Date and Time API:**

|  |  |  |
| --- | --- | --- |
| S.NO. | JAVA’S OLD JAVA DATE API | JAVA 8’S DATE AND TIME API |
| 1. | Available before Java 8 too. | It is introduced in Java SE 8 |
| 2. | Not Thread Safe. | Thread Safe. |
| 3. | Mutable API. | Immutable API. |
| 4. | Less Performance. | Better Performance. |
| 5. | Less Readability. | More Readability. |
| 6. | It’s not recommended to use as its deprecated. | It’s always recommended to use. |
| 7. | Not Extendable. | Easy to Extend. |
| 8. | It defines months values from 0 to 11, that is January = 0. | It defines months values from 1 to 12, that is January = 1. |
| 9. | It’s an old API. | It’s a new API. |

**What is Multiple Inheritance? How Java 8 supports Multiple Inheritance?**

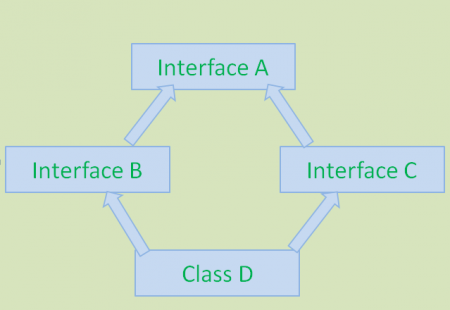
Multiple Inheritance means a class can inherit or extend characteristics and features from more than one parent class.

In Java 7 or Earlier, Multiple Inheritance is not possible because Java follows “A class should extend one and only one class or abstract class” Rule. However, it’s possible to provide Multiple Implementation Inheritance using Interface because Java follows “A class can extend any number of Interfaces” Rule.

However, Java 8 supports “Implementing Methods in Interfaces” by introducing new features: Default methods in Interface. Because of this feature, Java 8 supports Multiple Inheritance with some limitations.

**What is Diamond Problem in Inheritance? How Java 8 Solves this problem?**

A Diamond Problem is a Multiple Inheritance problem. In Java, It occurs when a Class extends more than one Interface which have same method implementation (Default method).



This above diagram shows Diamond Problem. To avoid this problem, Java 7 and Earlier versions does not support methods implementation in interface and also doesn’t support Multiple Inheritance. Java 8 has introduced new feature: Default methods to support Multiple Inheritance with some limitations.

Sample Java SE 8 Code to show this Diamond Problem:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public interface A{    default void display() { //code goes here }  }  public interface B extends A{ }  public interface C extends A{ }  public class D implements B,C{ } |

In the above code snippet, class D gives compiltime errors because Java Compiler will get bit confusion about which display() has to provide in class D. Class D inherits display() method from both interfaces B and C. To solve this problem, Java SE 8 has given the following remedy:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public interface A{    default void display() { //code goes here }  }  public interface B extends A{ }  public interface C extends A{ }  public class D implements B,C{    void display() {      B.super.display();    }  } |

This **B.super.display();** will solve this Diamond Problem.

## Java Heap Space

Java Heap space is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create any object, it’s always created in the Heap space. Garbage Collection runs on the heap memory to free the memory used by objects that doesn’t have any reference. Any object created in the heap space has global access and can be referenced from anywhere of the application.

### Java Stack Memory

Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. Stack memory is always referenced in LIFO (Last-In-First-Out) order. Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method. As soon as method ends, the block becomes unused and become available for next method.  
Stack memory size is very less compared to Heap memory.

Let’s understand the Heap and Stack memory usage with a simple program.

package com.journaldev.test;

public class Memory {

public static void main(String[] args) { // Line 1

int i=1; // Line 2

Object obj = new Object(); // Line 3

Memory mem = new Memory(); // Line 4

mem.foo(obj); // Line 5

} // Line 9

private void foo(Object param) { // Line 6

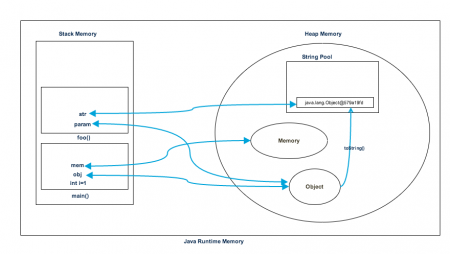
String str = param.toString(); //// Line 7

System.out.println(str);

} // Line 8

}

Below image shows the Stack and Heap memory with reference to above program and how they are being used to store primitive, Objects and reference variables.

[](http://cdn.journaldev.com/wp-content/uploads/2014/08/Java-Heap-Stack-Memory.png)

Let’s go through the steps of execution of the program.

* As soon as we run the program, it loads all the Runtime classes into the Heap space. When main() method is found at line 1, Java Runtime creates stack memory to be used by main() method thread.
* We are creating primitive local variable at line 2, so it’s created and stored in the stack memory of main() method.
* Since we are creating an Object in line 3, it’s created in Heap memory and stack memory contains the reference for it. Similar process occurs when we create Memory object in line 4.
* Now when we call foo() method in line 5, a block in the top of the stack is created to be used by foo() method. Since Java is pass by value, a new reference to Object is created in the foo() stack block in line 6.
* A string is created in line 7, it goes in the [String Pool](http://www.journaldev.com/797/what-is-java-string-pool) in the heap space and a reference is created in the foo() stack space for it.
* foo() method is terminated in line 8, at this time memory block allocated for foo() in stack becomes free.
* In line 9, main() method terminates and the stack memory created for main() method is destroyed. Also the program ends at this line, hence Java Runtime frees all the memory and end the execution of the program.

## Difference between Java Heap Space and Stack Memory

Based on the above explanations, we can easily conclude following differences between Heap and Stack memory.

1. Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.
2. Whenever an object is created, it’s always stored in the Heap space and stack memory contains the reference to it. Stack memory only contains local primitive variables and reference variables to objects in heap space.
3. Objects stored in the heap are globally accessible whereas stack memory can’t be accessed by other threads.
4. Memory management in stack is done in LIFO manner whereas it’s more complex in Heap memory because it’s used globally. Heap memory is divided into Young-Generation, Old-Generation etc, more details at [Java Garbage Collection](http://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java).
5. Stack memory is short-lived whereas heap memory lives from the start till the end of application execution.
6. We can use **-Xms** and **-Xmx** JVM option to define the startup size and maximum size of heap memory. We can use **-Xss** to define the stack memory size.
7. When stack memory is full, Java runtime throws java.lang.StackOverFlowError whereas if heap memory is full, it throws java.lang.OutOfMemoryError: Java Heap Space error.
8. Stack memory size is very less when compared to Heap memory. Because of simplicity in memory allocation (LIFO), stack memory is very fast when compared to heap memory.

That’s all for **Java Heap Space vs Stack Memory** in terms of java application, I hope it will clear your doubts regarding memory allocation when any java program is executed.

1. **What are Collection related features in Java 8?**

Java 8 has brought major changes in the Collection API. Some of the changes are:

* 1. [Java Stream API](http://www.journaldev.com/2774/java-8-stream) for collection classes for supporting sequential as well as parallel processing
  2. [Iterable interface is extended with forEach()](http://www.journaldev.com/2389/java-8-features-with-examples#iterable-forEach) default method that we can use to iterate over a collection. It is very helpful when used with [lambda expressions](http://www.journaldev.com/2763/java-8-functional-interfaces) because it’s argument Consumer is a [function interface](http://www.journaldev.com/2763/java-8-functional-interfaces).
  3. Miscellaneous Collection API improvements such as forEachRemaining(Consumer action)method in Iterator interface, Map replaceAll(), compute(), merge() methods.

1. **What is Java Collections Framework? List out some benefits of Collections framework?**

Collections are used in every programming language and initial java release contained few classes for collections: **Vector**, **Stack**, **Hashtable**, **Array**. But looking at the larger scope and usage, Java 1.2 came up with Collections Framework that group all the collections interfaces, implementations and algorithms.  
Java Collections have come through a long way with usage of Generics and Concurrent Collection classes for thread-safe operations. It also includes blocking interfaces and their implementations in java concurrent package.  
Some of the benefits of collections framework are;

* 1. Reduced development effort by using core collection classes rather than implementing our own collection classes.
  2. Code quality is enhanced with the use of well tested collections framework classes.
  3. Reduced effort for code maintenance by using collection classes shipped with JDK.
  4. Reusability and Interoperability

1. **What is the benefit of Generics in Collections Framework?**

Java 1.5 came with Generics and all collection interfaces and implementations use it heavily. Generics allow us to provide the type of Object that a collection can contain, so if you try to add any element of other type it throws compile time error.  
This avoids ClassCastException at Runtime because you will get the error at compilation. Also Generics make code clean since we don’t need to use casting and *instanceof* operator. I would highly recommend to go through [**Java Generic Tutorial**](http://www.journaldev.com/1663/java-generics-example-method-class-interface) to understand generics in a better way.

1. **What are the basic interfaces of Java Collections Framework?**

[Collection](http://www.journaldev.com/1260/collections-in-java-tutorial#collection-interface) is the root of the collection hierarchy. A collection represents a group of objects known as its elements. The Java platform doesn’t provide any direct implementations of this interface.

[Set](http://www.journaldev.com/1260/collections-in-java-tutorial#set-interface) is a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the deck of cards.

[List](http://www.journaldev.com/1260/collections-in-java-tutorial#list-interface) is an ordered collection and can contain duplicate elements. You can access any element from it’s index. List is more like array with dynamic length.

A [Map](http://www.journaldev.com/1260/collections-in-java-tutorial#map-interface) is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value.

Some other interfaces are [Queue](http://www.journaldev.com/1260/collections-in-java-tutorial#queue-interface), [Dequeue](http://www.journaldev.com/1260/collections-in-java-tutorial#dequeue-interface), [Iterator](http://www.journaldev.com/1260/collections-in-java-tutorial#iterator-interface), [SortedSet](http://www.journaldev.com/1260/collections-in-java-tutorial#sortedset-interface), [SortedMap](http://www.journaldev.com/1260/collections-in-java-tutorial#sortedmap-interface) and [ListIterator](http://www.journaldev.com/1260/collections-in-java-tutorial#listiterator-interface).

1. **Why Collection doesn’t extend Cloneable and Serializable interfaces?**

Collection interface specifies group of Objects known as elements. How the elements are maintained is left up to the concrete implementations of Collection. For example, some Collection implementations like List allow duplicate elements whereas other implementations like Set don’t.  
A lot of the Collection implementations have a public clone method. However, it does’t really make sense to include it in all implementations of Collection. This is because Collection is an abstract representation. What matters is the implementation.  
The semantics and the implications of either cloning or serializing come into play when dealing with the actual implementation; so concrete implementation should decide how it should be cloned or serialized, or even if it can be cloned or serialized.  
So mandating cloning and serialization in all implementations is actually less flexible and more restrictive. The specific implementation should make the decision as to whether it can be cloned or serialized.

1. **Why Map interface doesn’t extend Collection interface?**

Although Map interface and it’s implementations are part of Collections Framework, Map are not collections and collections are not Map. Hence it doesn’t make sense for Map to extend Collection or vice versa.  
If Map extends Collection interface, then where are the elements? Map contains key-value pairs and it provides methods to retrieve list of Keys or values as Collection but it doesn’t fit into the “group of elements” paradigm.

1. **What is an Iterator?**

Iterator interface provides methods to iterate over any Collection. We can get iterator instance from a Collection using *iterator()* method. Iterator takes the place of Enumeration in the Java Collections Framework. Iterators allow the caller to remove elements from the underlying collection during the iteration. Java Collection iterator provides a generic way for traversal through the elements of a collection and implements [**Iterator Design Pattern**](http://www.journaldev.com/1716/iterator-design-pattern-java).

1. **What is difference between Enumeration and Iterator interface?**

Enumeration is twice as fast as Iterator and uses very less memory. Enumeration is very basic and fits to basic needs. But Iterator is much safer as compared to Enumeration because it always denies other threads to modify the collection object which is being iterated by it.  
Iterator takes the place of Enumeration in the Java Collections Framework. Iterators allow the caller to remove elements from the underlying collection that is not possible with Enumeration. Iterator method names have been improved to make it’s functionality clear.

1. **Why there is not method like Iterator.add() to add elements to the collection?**

The semantics are unclear, given that the contract for Iterator makes no guarantees about the order of iteration. Note, however, that ListIterator does provide an add operation, as it does guarantee the order of the iteration.

1. **Why Iterator don’t have a method to get next element directly without moving the cursor?**

It can be implemented on top of current Iterator interface but since it’s use will be rare, it doesn’t make sense to include it in the interface that everyone has to implement.

1. **What is different between Iterator and ListIterator?**
   1. We can use Iterator to traverse Set and List collections whereas ListIterator can be used with Lists only.
   2. Iterator can traverse in forward direction only whereas ListIterator can be used to traverse in both the directions.
   3. ListIterator inherits from Iterator interface and comes with extra functionalities like adding an element, replacing an element, getting index position for previous and next elements.
2. **What are different ways to iterate over a list?**

We can iterate over a list in two different ways – using iterator and using for-each loop.

List<String> strList = new ArrayList<>();

//using for-each loop

for(String obj : strList){

System.out.println(obj);

}

//using iterator

Iterator<String> it = strList.iterator();

while(it.hasNext()){

String obj = it.next();

System.out.println(obj);

}

Using iterator is more thread-safe because it makes sure that if underlying list elements are modified, it will throw ConcurrentModificationException.

1. **What do you understand by iterator fail-fast property?**

Iterator fail-fast property checks for any modification in the structure of the underlying collection everytime we try to get the next element. If there are any modifications found, it throws ConcurrentModificationException. All the implementations of Iterator in Collection classes are fail-fast by design except the concurrent collection classes like ConcurrentHashMap and CopyOnWriteArrayList.

1. **What is difference between fail-fast and fail-safe?**

Iterator fail-safe property work with the clone of underlying collection, hence it’s not affected by any modification in the collection. By design, all the collection classes in java.util package are fail-fast whereas collection classes in java.util.concurrent are fail-safe.  
Fail-fast iterators throw ConcurrentModificationException whereas fail-safe iterator never throws ConcurrentModificationException.  
Check this post for [CopyOnWriteArrayList Example](http://www.journaldev.com/1289/copyonwritearraylist-java).

1. **How to avoid ConcurrentModificationException while iterating a collection?**

We can use concurrent collection classes to avoid ConcurrentModificationException while iterating over a collection, for example CopyOnWriteArrayList instead of ArrayList.  
Check this post for [ConcurrentHashMap Example](http://www.journaldev.com/122/java-concurrenthashmap-example-iterator).

1. **Why there are no concrete implementations of Iterator interface?**

Iterator interface declare methods for iterating a collection but it’s implementation is responsibility of the Collection implementation classes. Every collection class that returns an iterator for traversing has it’s own Iterator implementation nested class.  
This allows collection classes to chose whether iterator is fail-fast or fail-safe. For example ArrayList iterator is fail-fast whereas CopyOnWriteArrayList iterator is fail-safe.

1. **What is UnsupportedOperationException?**

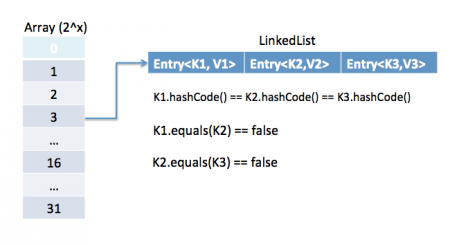
UnsupportedOperationException is the exception used to indicate that the operation is not supported. It’s used extensively in [JDK](http://www.journaldev.com/546/difference-jdk-vs-jre-vs-jvm) classes, in collections framework java.util.Collections.UnmodifiableCollection throws this exception for all add and removeoperations.

1. **How HashMap works in Java?**

HashMap stores key-value pair in Map.Entry static nested class implementation. HashMap works on hashing algorithm and uses hashCode() and equals() method in put and get methods.

When we call put method by passing key-value pair, HashMap uses Key hashCode() with hashing to find out the index to store the key-value pair. The Entry is stored in the LinkedList, so if there are already existing entry, it uses equals() method to check if the passed key already exists, if yes it overwrites the value else it creates a new entry and store this key-value Entry.

When we call get method by passing Key, again it uses the hashCode() to find the index in the array and then use equals() method to find the correct Entry and return it’s value. Below image will explain these detail clearly.

[](http://cdn.journaldev.com/wp-content/uploads/2013/01/java-hashmap-entry-impl.png)

The other important things to know about HashMap are capacity, load factor, threshold resizing. HashMap initial default capacity is **16** and load factor is 0.75. Threshold is capacity multiplied by load factor and whenever we try to add an entry, if map size is greater than threshold, HashMap rehashes the contents of map into a new array with a larger capacity. The capacity is always power of 2, so if you know that you need to store a large number of key-value pairs, for example in caching data from database, it’s good idea to initialize the HashMap with correct capacity and load factor.

1. **What is the importance of hashCode() and equals() methods?**

HashMap uses Key object hashCode() and equals() method to determine the index to put the key-value pair. These methods are also used when we try to get value from HashMap. If these methods are not implemented correctly, two different Key’s might produce same hashCode() and equals() output and in that case rather than storing it at different location, HashMap will consider them same and overwrite them.

Similarly all the collection classes that doesn’t store duplicate data use hashCode() and equals() to find duplicates, so it’s very important to implement them correctly. The implementation of equals() and hashCode() should follow these rules.

* 1. If o1.equals(o2), then o1.hashCode() == o2.hashCode()should always be true.
  2. If o1.hashCode() == o2.hashCode is true, it doesn’t mean that o1.equals(o2) will be true.

1. **Can we use any class as Map key?**

We can use any class as Map Key, however following points should be considered before using them.

* 1. If the class overrides equals() method, it should also override hashCode() method.
  2. The class should follow the rules associated with equals() and hashCode() for all instances. Please refer earlier question for these rules.
  3. If a class field is not used in equals(), you should not use it in hashCode() method.
  4. Best practice for user defined key class is to make it immutable, so that hashCode() value can be cached for fast performance. Also immutable classes make sure that hashCode() and equals() will not change in future that will solve any issue with mutability.  
     For example, let’s say I have a class MyKey that I am using for HashMap key.
  5. //MyKey name argument passed is used for equals() and hashCode()
  6. MyKey key = new MyKey("Pankaj"); //assume hashCode=1234
  7. myHashMap.put(key, "Value");
  8. // Below code will change the key hashCode() and equals()
  9. // but it's location is not changed.
  10. key.setName("Amit"); //assume new hashCode=7890
  11. //below will return null, because HashMap will try to look for key
  12. //in the same index as it was stored but since key is mutated,
  13. //there will be no match and it will return null.

myHashMap.get(new MyKey("Pankaj"));

This is the reason why String and Integer are mostly used as HashMap keys.

1. **What are different Collection views provided by Map interface?**

Map interface provides three collection views:

* 1. **Set keySet()**: Returns a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator’s own remove operation), the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll, and clear operations. It does not support the add or addAll operations.
  2. **Collection values()**: Returns a Collection view of the values contained in this map. The collection is backed by the map, so changes to the map are reflected in the collection, and vice-versa. If the map is modified while an iteration over the collection is in progress (except through the iterator’s own remove operation), the results of the iteration are undefined. The collection supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Collection.remove, removeAll, retainAll and clear operations. It does not support the add or addAll operations.
  3. **Set<Map.Entry<K, V>> entrySet()**: Returns a Set view of the mappings contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator’s own remove operation, or through the setValue operation on a map entry returned by the iterator) the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll and clear operations. It does not support the add or addAll operations.

1. **What is difference between HashMap and Hashtable?**

HashMap and Hashtable both implements Map interface and looks similar, however there are following difference between HashMap and Hashtable.

* 1. HashMap allows null key and values whereas Hashtable doesn’t allow null key and values.
  2. Hashtable is synchronized but HashMap is not synchronized. So HashMap is better for single threaded environment, Hashtable is suitable for multi-threaded environment.
  3. LinkedHashMap was introduced in Java 1.4 as a subclass of HashMap, so incase you want iteration order, you can easily switch from HashMap to LinkedHashMap but that is not the case with Hashtable whose iteration order is unpredictable.
  4. HashMap provides Set of keys to iterate and hence it’s fail-fast but Hashtable provides Enumeration of keys that doesn’t support this feature.
  5. Hashtable is considered to be legacy class and if you are looking for modifications of Map while iterating, you should use ConcurrentHashMap.

1. **How to decide between HashMap and TreeMap?**

For inserting, deleting, and locating elements in a Map, the HashMap offers the best alternative. If, however, you need to traverse the keys in a sorted order, then TreeMap is your better alternative. Depending upon the size of your collection, it may be faster to add elements to a HashMap, then convert the map to a TreeMap for sorted key traversal.

1. **What are similarities and difference between ArrayList and Vector?**

ArrayList and Vector are similar classes in many ways.

* 1. Both are index based and backed up by an array internally.
  2. Both maintains the order of insertion and we can get the elements in the order of insertion.
  3. The iterator implementations of ArrayList and Vector both are fail-fast by design.
  4. ArrayList and Vector both allows null values and random access to element using index number.

These are the differences between ArrayList and Vector.

* 1. Vector is synchronized whereas ArrayList is not synchronized. However if you are looking for modification of list while iterating, you should use CopyOnWriteArrayList.
  2. ArrayList is faster than Vector because it doesn’t have any overhead because of synchronization.
  3. ArrayList is more versatile because we can get synchronized list or read-only list from it easily using Collections utility class.

1. **What is difference between Array and ArrayList? When will you use Array over ArrayList?**

Arrays can contain primitive or Objects whereas ArrayList can contain only Objects.  
Arrays are fixed size whereas ArrayList size is dynamic.  
Arrays doesn’t provide a lot of features like ArrayList, such as addAll, removeAll, iterator etc.

Although ArrayList is the obvious choice when we work on list, there are few times when array are good to use.

* 1. If the size of list is fixed and mostly used to store and traverse them.
  2. For list of primitive data types, although Collections use autoboxing to reduce the coding effort but still it makes them slow when working on fixed size primitive data types.
  3. If you are working on fixed multi-dimensional situation, using [][] is far more easier than List<List<>>

1. **What is difference between ArrayList and LinkedList?**

ArrayList and LinkedList both implement List interface but there are some differences between them.

* 1. ArrayList is an index based data structure backed by Array, so it provides random access to it’s elements with performance as O(1) but LinkedList stores data as list of nodes where every node is linked to it’s previous and next node. So even though there is a method to get the element using index, internally it traverse from start to reach at the index node and then return the element, so performance is O(n) that is slower than ArrayList.
  2. Insertion, addition or removal of an element is faster in LinkedList compared to ArrayList because there is no concept of resizing array or updating index when element is added in middle.
  3. LinkedList consumes more memory than ArrayList because every node in LinkedList stores reference of previous and next elements.

1. **Which collection classes provide random access of it’s elements?**

ArrayList, HashMap, TreeMap, Hashtable classes provide random access to it’s elements. Download [java collections pdf](http://cdn.journaldev.com/wp-content/uploads/2013/01/java-collections-framework.pdf) for more information.

1. **What is EnumSet?**

java.util.EnumSet is Set implementation to use with enum types. All of the elements in an enum set must come from a single enum type that is specified, explicitly or implicitly, when the set is created. EnumSet is not synchronized and null elements are not allowed. It also provides some useful methods like copyOf(Collection c), of(E first, E… rest) and complementOf(EnumSet s).

Check this post for [java enum tutorial](http://www.journaldev.com/716/java-enum).

1. **Which collection classes are thread-safe?**

Vector, Hashtable, Properties and Stack are synchronized classes, so they are thread-safe and can be used in multi-threaded environment. Java 1.5 Concurrent API included some collection classes that allows modification of collection while iteration because they work on the clone of the collection, so they are safe to use in multi-threaded environment.

1. **What are concurrent Collection Classes?**

Java 1.5 Concurrent package (java.util.concurrent) contains thread-safe collection classes that allow collections to be modified while iterating. By design Iterator implementation in java.utilpackages are fail-fast and throws ConcurrentModificationException. But Iterator implementation in java.util.concurrent packages are fail-safe and we can modify the collection while iterating. Some of these classes are CopyOnWriteArrayList, ConcurrentHashMap, CopyOnWriteArraySet.

Read these posts to learn about them in more detail.

* 1. [Avoid ConcurrentModificationException](http://www.journaldev.com/378/java-util-concurrentmodificationexception)
  2. [CopyOnWriteArrayList Example](http://www.journaldev.com/1289/copyonwritearraylist-java)
  3. [HashMap vs ConcurrentHashMap](http://www.journaldev.com/122/java-concurrenthashmap-example-iterator)

1. **What is BlockingQueue?**

java.util.concurrent.BlockingQueue is a Queue that supports operations that wait for the queue to become non-empty when retrieving and removing an element, and wait for space to become available in the queue when adding an element.

BlockingQueue interface is part of java collections framework and it’s primarily used for implementing producer consumer problem. We don’t need to worry about waiting for the space to be available for producer or object to be available for consumer in BlockingQueue as it’s handled by implementation classes of BlockingQueue.

Java provides several BlockingQueue implementations such as ArrayBlockingQueue, LinkedBlockingQueue, PriorityBlockingQueue, SynchronousQueue etc.  
Check this post for use of BlockingQueue for [producer-consumer problem](http://www.journaldev.com/1034/java-blockingqueue-example).

1. **What is Queue and Stack, list their differences?**

Both Queue and Stack are used to store data before processing them. java.util.Queue is an interface whose implementation classes are present in java concurrent package. Queue allows retrieval of element in First-In-First-Out (FIFO) order but it’s not always the case. There is also Deque interface that allows elements to be retrieved from both end of the queue.  
Stack is similar to queue except that it allows elements to be retrieved in Last-In-First-Out (LIFO) order.  
Stack is a class that extends Vector whereas Queue is an interface.

1. **What is Collections Class?**

java.util.Collections is a utility class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, “wrappers”, which return a new collection backed by a specified collection, and a few other odds and ends.

This class contains methods for collection framework algorithms, such as binary search, sorting, shuffling, reverse etc.

1. **What is Comparable and Comparator interface?**

Java provides Comparable interface which should be implemented by any custom class if we want to use Arrays or Collections sorting methods. Comparable interface has compareTo(T obj) method which is used by sorting methods. We should override this method in such a way that it returns a negative integer, zero, or a positive integer if “this” object is less than, equal to, or greater than the object passed as argument.

But, in most real life scenarios, we want sorting based on different parameters. For example, as a CEO, I would like to sort the employees based on Salary, an HR would like to sort them based on the age. This is the situation where we need to use Comparator interface because Comparable.compareTo(Object o) method implementation can sort based on one field only and we can’t chose the field on which we want to sort the Object.

Comparator interface compare(Object o1, Object o2) method need to be implemented that takes two Object argument, it should be implemented in such a way that it returns negative int if first argument is less than the second one and returns zero if they are equal and positive int if first argument is greater than second one.

Check this post for use of Comparable and Comparator interface to [sort objects](http://www.journaldev.com/780/comparable-and-comparator-in-java-example).

1. **What is difference between Comparable and Comparator interface?**

Comparable and Comparator interfaces are used to sort collection or array of objects.

Comparable interface is used to provide the natural sorting of objects and we can use it to provide sorting based on single logic.  
Comparator interface is used to provide different algorithms for sorting and we can chose the comparator we want to use to sort the given collection of objects.

1. **How can we sort a list of Objects?**

If we need to sort an array of Objects, we can use Arrays.sort(). If we need to sort a list of objects, we can use Collections.sort(). Both these classes have overloaded sort() methods for natural sorting (using Comparable) or sorting based on criteria (using Comparator).  
Collections internally uses Arrays sorting method, so both of them have same performance except that Collections take sometime to convert list to array.

1. **While passing a Collection as argument to a function, how can we make sure the function will not be able to modify it?**

We can create a read-only collection using Collections.unmodifiableCollection(Collection c)method before passing it as argument, this will make sure that any operation to change the collection will throw UnsupportedOperationException.

1. **How can we create a synchronized collection from given collection?**

We can use Collections.synchronizedCollection(Collection c) to get a synchronized (thread-safe) collection backed by the specified collection.

1. **What are common algorithms implemented in Collections Framework?**

Java Collections Framework provides algorithm implementations that are commonly used such as sorting and searching. Collections class contain these method implementations. Most of these algorithms work on List but some of them are applicable for all kinds of collections.  
Some of them are sorting, searching, shuffling, min-max values.

1. **What is Big-O notation? Give some examples?**

The Big-O notation describes the performance of an algorithm in terms of number of elements in a data structure. Since Collection classes are actually data structures, we usually tend to use Big-O notation to chose the collection implementation to use based on time, memory and performance.

Example 1: ArrayList get(index i) is a constant-time operation and doesn’t depend on the number of elements in the list. So it’s performance in Big-O notation is O(1).  
Example 2: A linear search on array or list performance is O(n) because we need to search through entire list of elements to find the element.

1. **What are best practices related to Java Collections Framework?**
   1. Chosing the right type of collection based on the need, for example if size is fixed, we might want to use Array over ArrayList. If we have to iterate over the Map in order of insertion, we need to use TreeMap. If we don’t want duplicates, we should use Set.
   2. Some collection classes allows to specify the initial capacity, so if we have an estimate of number of elements we will store, we can use it to avoid rehashing or resizing.
   3. Write program in terms of interfaces not implementations, it allows us to change the implementation easily at later point of time.
   4. Always use Generics for type-safety and avoid ClassCastException at runtime.
   5. Use immutable classes provided by JDK as key in Map to avoid implementation of hashCode() and equals() for our custom class.
   6. Use Collections utility class as much as possible for algorithms or to get read-only, synchronized or empty collections rather than writing own implementation. It will enhance code-reuse with greater stability and low maintainability.
2. **What is Java Priority Queue?**

PriorityQueue is an unbounded queue based on a priority heap and the elements are ordered in their natural order or we can provide [Comparator](http://www.journaldev.com/780/comparable-and-comparator-in-java-example) for ordering at the time of creation. PriorityQueue doesn’t allow null values and we can’t add any object that doesn’t provide natural ordering or we don’t have any comparator for them for ordering. Java PriorityQueue is not [thread-safe](http://www.journaldev.com/1061/thread-safety-in-java) and provided O(log(n)) time for enqueing and dequeing operations. Check this post for [java priority queue example](http://www.journaldev.com/1642/java-priority-queue-priorityqueue-example).

1. **Why can’t we write code as List<Number> numbers = new ArrayList<Integer>();?**

Generics doesn’t support sub-typing because it will cause issues in achieving type safety. That’s why List<T> is not considered as a subtype of List<S> where S is the super-type of T. To understanding why it’s not allowed, let’s see what could have happened if it has been supported.

List<Long> listLong = new ArrayList<Long>();

listLong.add(Long.valueOf(10));

List<Number> listNumbers = listLong; // compiler error

listNumbers.add(Double.valueOf(1.23));

As you can see from above code that IF generics would have been supporting sub-typing, we could have easily add a Double to the list of Long that would have caused ClassCastException at runtime while traversing the list of Long.

1. **Why can’t we create generic array? or write code as List<Integer>[] array = new ArrayList<Integer>[10];**

We are not allowed to create generic arrays because array carry type information of it’s elements at runtime. This information is used at runtime to throw ArrayStoreException if elements type doesn’t match to the defined type. Since generics type information gets erased at runtime by Type Erasure, the array store check would have been passed where it should have failed. Let’s understand this with a simple example code.

List<Integer>[] intList = new List<Integer>[5]; // compile error

Object[] objArray = intList;

List<Double> doubleList = new ArrayList<Double>();

doubleList.add(Double.valueOf(1.23));

objArray[0] = doubleList; // this should fail but it would pass because at runtime intList and doubleList both are just List

Arrays are covariant by nature i.e S[] is a subtype of T[] whenever S is a subtype of T but generics doesn’t support covariance or sub-typing as we saw in last question. So if we would have been allowed to create generic arrays, because of type erasure we would not get array store exception even though both types are not related.

### What is the difference between Process and Thread?

A process is a self contained execution environment and it can be seen as a program or application whereas Thread is a single task of execution within the process. Java runtime environment runs as a single process which contains different classes and programs as processes. Thread can be called lightweight process. Thread requires less resources to create and exists in the process, thread shares the process resources.

### What are the benefits of multi-threaded programming?

In Multi-Threaded programming, multiple threads are executing concurrently that improves the performance because CPU is not idle incase some thread is waiting to get some resources. Multiple threads share the heap memory, so it’s good to create multiple threads to execute some task rather than creating multiple processes. For example, Servlets are better in performance than CGI because Servlet support multi-threading but CGI doesn’t.

### What is difference between user Thread and daemon Thread?

When we create a Thread in java program, it’s known as user thread. A daemon thread runs in background and doesn’t prevent JVM from terminating. When there are no user threads running, JVM shutdown the program and quits. A child thread created from daemon thread is also a daemon thread.

### How can we create a Thread in Java?

There are two ways to create Thread in Java – first by implementing Runnable interface and then creating a Thread object from it and second is to extend the Thread Class. Read this post to learn more about [creating threads in java](http://www.journaldev.com/1016/java-thread-example).

### What are different states in lifecycle of Thread?

When we create a Thread in java program, its state is New. Then we start the thread that change it’s state to Runnable. Thread Scheduler is responsible to allocate CPU to threads in Runnable thread pool and change their state to Running. Other Thread states are Waiting, Blocked and Dead. Read this post to learn more about [life cycle of thread](http://www.journaldev.com/1044/thread-life-cycle-in-java-thread-states-in-java).

[sociallocker id=”2713″]

### Can we call run() method of a Thread class?

Yes, we can call run() method of a Thread class but then it will behave like a normal method. To actually execute it in a Thread, we need to start it using **Thread.start()** method.

### How can we pause the execution of a Thread for specific time?

We can use Thread class sleep() method to pause the execution of Thread for certain time. Note that this will not stop the processing of thread for specific time, once the thread awake from sleep, it’s state gets changed to runnable and based on thread scheduling, it gets executed.

### What do you understand about Thread Priority?

Every thread has a priority, usually higher priority thread gets precedence in execution but it depends on Thread Scheduler implementation that is OS dependent. We can specify the priority of thread but it doesn’t guarantee that higher priority thread will get executed before lower priority thread. Thread priority is an int whose value varies from 1 to 10 where 1 is the lowest priority thread and 10 is the highest priority thread.

### What is Thread Scheduler and Time Slicing?

Thread Scheduler is the Operating System service that allocates the CPU time to the available runnable threads. Once we create and start a thread, it’s execution depends on the implementation of Thread Scheduler. Time Slicing is the process to divide the available CPU time to the available runnable threads. Allocation of CPU time to threads can be based on thread priority or the thread waiting for longer time will get more priority in getting CPU time. Thread scheduling can’t be controlled by java, so it’s always better to control it from application itself.

### What is context-switching in multi-threading?

Context Switching is the process of storing and restoring of CPU state so that Thread execution can be resumed from the same point at a later point of time. Context Switching is the essential feature for multitasking operating system and support for multi-threaded environment.

### How can we make sure main() is the last thread to finish in Java Program?

We can use Thread join() method to make sure all the threads created by the program is dead before finishing the main function. Here is an article about [Thread join method](http://www.journaldev.com/1024/java-thread-join-example).

### How does thread communicate with each other?

When threads share resources, communication between Threads is important to coordinate their efforts. Object class wait(), notify() and notifyAll() methods allows threads to communicate about the lock status of a resource. Check this post to learn more about [thread wait, notify and notifyAll](http://www.journaldev.com/1037/java-thread-wait-notify-and-notifyall-example).

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

In Java every Object has a monitor and wait, notify methods are used to wait for the Object monitor or to notify other threads that Object monitor is free now. There is no monitor on threads in java and synchronization can be used with any Object, that’s why it’s part of Object class so that every class in java has these essential methods for inter thread communication.

### Why wait(), notify() and notifyAll() methods have to be called from synchronized method or block?

When a Thread calls wait() on any Object, it must have the monitor on the Object that it will leave and goes in wait state until any other thread call notify() on this Object. Similarly when a thread calls notify() on any Object, it leaves the monitor on the Object and other waiting threads can get the monitor on the Object. Since all these methods require Thread to have the Object monitor, that can be achieved only by synchronization, they need to be called from synchronized method or block.

### Why Thread sleep() and yield() methods are static?

Thread sleep() and yield() methods work on the currently executing thread. So there is no point in invoking these methods on some other threads that are in wait state. That’s why these methods are made static so that when this method is called statically, it works on the current executing thread and avoid confusion to the programmers who might think that they can invoke these methods on some non-running threads.

### How can we achieve thread safety in Java?

There are several ways to achieve thread safety in java – synchronization, atomic concurrent classes, implementing concurrent Lock interface, using volatile keyword, using immutable classes and Thread safe classes. Learn more at [thread safety tutorial](http://www.journaldev.com/1061/thread-safety-in-java).

### What is volatile keyword in Java

When we use volatile keyword with a variable, all the threads read it’s value directly from the memory and don’t cache it. This makes sure that the value read is the same as in the memory.

### Which is more preferred – Synchronized method or Synchronized block?

Synchronized block is more preferred way because it doesn’t lock the Object, synchronized methods lock the Object and if there are multiple synchronization blocks in the class, even though they are not related, it will stop them from execution and put them in wait state to get the lock on Object.

### How to create daemon thread in Java?

Thread class setDaemon(true) can be used to create daemon thread in java. We need to call this method before calling start() method else it will throw IllegalThreadStateException.

### What is ThreadLocal?

Java ThreadLocal is used to create thread-local variables. We know that all threads of an Object share it’s variables, so if the variable is not thread safe, we can use synchronization but if we want to avoid synchronization, we can use ThreadLocal variables.  
Every thread has it’s own ThreadLocal variable and they can use it’s get() and set() methods to get the default value or change it’s value local to Thread. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread. Check this post for small example program showing [ThreadLocal Example](http://www.journaldev.com/1076/java-threadlocal-example).

### What is Thread Group? Why it’s advised not to use it?

ThreadGroup is a class which was intended to provide information about a thread group. ThreadGroup API is weak and it doesn’t have any functionality that is not provided by Thread. Two of the major feature it had are to get the list of active threads in a thread group and to set the uncaught exception handler for the thread. But Java 1.5 has added setUncaughtExceptionHandler(UncaughtExceptionHandler eh) method using which we can add uncaught exception handler to the thread. So ThreadGroup is obsolete and hence not advised to use anymore.

t1.setUncaughtExceptionHandler(new UncaughtExceptionHandler(){

@Override

public void uncaughtException(Thread t, Throwable e) {

System.out.println("exception occured:"+e.getMessage());

}

});

### What is Java Thread Dump, How can we get Java Thread dump of a Program?

Thread dump is list of all the threads active in the JVM, thread dumps are very helpful in analyzing bottlenecks in the application and analyzing deadlock situations. There are many ways using which we can generate Thread dump – Using Profiler, Kill -3 command, jstack tool etc. I prefer jstack tool to generate thread dump of a program because it’s easy to use and comes with JDK installation. Since it’s a terminal based tool, we can create script to generate thread dump at regular intervals to analyze it later on. Read this post to know more about [generating thread dump in java](http://www.journaldev.com/1053/java-thread-dump-visualvm-jstack-kill-3-jcmd).

### What is Deadlock? How to analyze and avoid deadlock situation?

Deadlock is a programming situation where two or more threads are blocked forever, this situation arises with at least two threads and two or more resources.

To analyze a deadlock, we need to look at the java thread dump of the application, we need to look out for the threads with state as BLOCKED and then the resources it’s waiting to lock, every resource has a unique ID using which we can find which thread is already holding the lock on the object.

Avoid Nested Locks, Lock Only What is Required and Avoid waiting indefinitely are common ways to avoid deadlock situation, read this post to learn how to [analyze deadlock in java](http://www.journaldev.com/1058/deadlock-in-java-example) with sample program.

### What is Java Timer Class? How to schedule a task to run after specific interval?

java.util.Timer is a utility class that can be used to schedule a thread to be executed at certain time in future. Java Timer class can be used to schedule a task to be run one-time or to be run at regular intervals.

java.util.TimerTask is an [**abstract class**](http://www.journaldev.com/1582/abstract-class-in-java) that implements Runnable interface and we need to extend this class to create our own TimerTask that can be scheduled using java Timer class.

Check this post for [java Timer example](http://www.journaldev.com/1050/java-timer-timertask-example).

### What is Thread Pool? How can we create Thread Pool in Java?

A thread pool manages the pool of worker threads, it contains a queue that keeps tasks waiting to get executed.

A thread pool manages the collection of Runnable threads and worker threads execute Runnable from the queue.

java.util.concurrent.Executors provide implementation of java.util.concurrent.Executor interface to create the thread pool in java. [Thread Pool Example](http://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice) program shows how to create and use Thread Pool in java. Or read [ScheduledThreadPoolExecutor Example](http://www.journaldev.com/2340/java-scheduler-scheduledexecutorservice-scheduledthreadpoolexecutor-example) to know how to schedule tasks after certain delay.

### What will happen if we don’t override Thread class run() method?

Thread class run() method code is as shown below.

public void run() {

if (target != null) {

target.run();

}

}

Above target set in the init() method of Thread class and if we create an instance of Thread class as new TestThread(), it’s set to null. So nothing will happen if we don’t override the run() method. Below is a simple example demonstrating this.

public class TestThread extends Thread {

//not overriding Thread.run() method

//main method, can be in other class too

public static void main(String args[]){

Thread t = new TestThread();

System.out.println("Before starting thread");

t.start();

System.out.println("After starting thread");

}

}

It will print only below output and terminate.

Before starting thread

After starting thread

## Java Concurrency Interview Questions and Answers

### What is atomic operation? What are atomic classes in Java Concurrency API?

Atomic operations are performed in a single unit of task without interference from other operations. Atomic operations are necessity in multi-threaded environment to avoid data inconsistency.

int++ is not an atomic operation. So by the time one threads read it’s value and increment it by one, other thread has read the older value leading to wrong result.

To solve this issue, we will have to make sure that increment operation on count is atomic, we can do that using Synchronization but Java 5 java.util.concurrent.atomic provides wrapper classes for int and long that can be used to achieve this atomically without usage of Synchronization. Go to this article to learn more about [atomic concurrent classes](http://www.journaldev.com/1095/atomicinteger-java).

### What is Lock interface in Java Concurrency API? What are it’s benefits over synchronization?

Lock interface provide more extensive locking operations than can be obtained using synchronized methods and statements. They allow more flexible structuring, may have quite different properties, and may support multiple associated Condition objects.  
The advantages of a lock are

* + it’s possible to make them fair
  + it’s possible to make a thread responsive to interruption while waiting on a Lock object.
  + it’s possible to try to acquire the lock, but return immediately or after a timeout if the lock can’t be acquired
  + it’s possible to acquire and release locks in different scopes, and in different orders

Read more at [**Java Lock Example**](http://www.journaldev.com/2377/java-lock-example-reentrantlock).

### What is Executors Framework?

In Java 5, Executor framework was introduced with the java.util.concurrent.Executor interface.

The Executor framework is a framework for standardizing invocation, scheduling, execution, and control of asynchronous tasks according to a set of execution policies.

Creating a lot many threads with no bounds to the maximum threshold can cause application to run out of heap memory. So, creating a ThreadPool is a better solution as a finite number of threads can be pooled and reused. Executors framework facilitate process of creating Thread pools in java. Check out this post to learn with example code to [create thread pool using Executors framework](http://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice).

### What is BlockingQueue? How can we implement Producer-Consumer problem using Blocking Queue?

java.util.concurrent.BlockingQueue is a Queue that supports operations that wait for the queue to become non-empty when retrieving and removing an element, and wait for space to become available in the queue when adding an element.

BlockingQueue doesn’t accept null values and throw NullPointerException if you try to store null value in the queue.

BlockingQueue implementations are thread-safe. All queuing methods are atomic in nature and use internal locks or other forms of concurrency control.

BlockingQueue interface is part of java collections framework and it’s primarily used for implementing producer consumer problem.  
Check this post for [producer-consumer problem implementation using BlockingQueue](http://www.journaldev.com/1034/java-blockingqueue-example).

### What is Callable and Future?

Java 5 introduced java.util.concurrent.Callable interface in concurrency package that is similar to Runnable interface but it can return any Object and able to throw Exception.

Callable interface use Generic to define the return type of Object. Executors class provide useful methods to execute Callable in a thread pool. Since callable tasks run in parallel, we have to wait for the returned Object. Callable tasks return java.util.concurrent.Future object. Using Future we can find out the status of the Callable task and get the returned Object. It provides get() method that can wait for the Callable to finish and then return the result.  
Check this post for [Callable Future Example](http://www.journaldev.com/1090/java-callable-future-example).

### What is FutureTask Class?

FutureTask is the base implementation class of Future interface and we can use it with Executors for asynchronous processing. Most of the time we don’t need to use FutureTask class but it comes real handy if we want to override some of the methods of Future interface and want to keep most of the base implementation. We can just extend this class and override the methods according to our requirements. Check out [**Java FutureTask Example**](http://www.journaldev.com/1650/java-futuretask-example-program) post to learn how to use it and what are different methods it has.

### What are Concurrent Collection Classes?

Java Collection classes are fail-fast which means that if the Collection will be changed while some thread is traversing over it using iterator, the iterator.next() will throw ConcurrentModificationException.

Concurrent Collection classes support full concurrency of retrievals and adjustable expected concurrency for updates.  
Major classes are ConcurrentHashMap, CopyOnWriteArrayList and CopyOnWriteArraySet, check this post to learn [how to avoid ConcurrentModificationException when using iterator](http://www.journaldev.com/378/java-util-concurrentmodificationexception).

### What is Executors Class?

Executors class provide utility methods for Executor, ExecutorService, ScheduledExecutorService, ThreadFactory, and Callable classes.

Executors class can be used to easily create Thread Pool in java, also this is the only class supporting execution of Callable implementations.

### What are some of the improvements in Concurrency API in Java 8?

Some important concurrent API enhancements are:

* + ConcurrentHashMap compute(), forEach(), forEachEntry(), forEachKey(), forEachValue(), merge(), reduce() and search() methods.
  + CompletableFuture that may be explicitly completed (setting its value and status).
  + Executors newWorkStealingPool() method to create a work-stealing thread pool using all available processors as its target parallelism level.

1. What is the purpose of BiConsumer<T,U> functional interface?
2. It represents an operation that accepts two input arguments, and returns no result.
3. What is the purpose of BiFunction<T,U,R> functional interface?
4. It represents a function that accepts two arguments and produces a result.
5. What is the purpose of BinaryOperator<T> functional interface?
6. It represents an operation upon two operands of the same type, producing a result of the same type as the operands.
7. What is the purpose of BiPredicate<T,U> functional interface?
8. It represents a predicate (Boolean-valued function) of two arguments.
9. What is the purpose of BooleanSupplier functional interface?
10. It represents a supplier of Boolean-valued results.
11. What is the purpose of Consumer<T> functional interface?
12. It represents an operation that accepts a single input argument and returns no result.
13. What is the purpose of DoubleBinaryOperator functional interface?
14. It represents an operation upon two double-valued operands and producing a double-valued result.
15. What is the purpose of DoubleConsumer functional interface?
16. It represents an operation that accepts a single double-valued argument and returns no result.
17. What is the purpose of DoubleFunction<R> functional interface?
18. It represents a function that accepts a double-valued argument and produces a result.
19. What is the purpose of DoublePredicate functional interface?
20. It represents a predicate (Boolean-valued function) of one double-valued argument.
21. What is the purpose of DoubleSupplier functional interface?
22. It represents a supplier of double-valued results.
23. What is the purpose of DoubleToIntFunction functional interface?
24. It represents a function that accepts a double-valued argument and produces an int-valued result.
25. What is the purpose of DoubleToLongFunction functional interface?
26. It represents a function that accepts a double-valued argument and produces a long-valued result.
27. What is the purpose of DoubleUnaryOperator functional interface?
28. It represents an operation on a single double-valued operand that produces a double-valued result.
29. What is the purpose of Function<T,R> functional interface?
30. It represents a function that accepts one argument and produces a result.
31. What is the purpose of IntBinaryOperator functional interface?
32. It represents an operation upon two int-valued operands and produces an int-valued result.
33. What is the purpose of IntConsumer functional interface?
34. It represents an operation that accepts a single int-valued argument and returns no result.
35. What is the purpose of IntFunction<R> functional interface?
36. It represents a function that accepts an int-valued argument and produces a result.
37. What is the purpose of IntPredicate functional interface?
38. It represents a predicate (Boolean-valued function) of one int-valued argument.
39. What is the purpose of IntSupplier functional interface?
40. It represents a supplier of int-valued results.
41. What is the purpose of IntToDoubleFunction functional interface?
42. It represents a function that accepts an int-valued argument and produces a double-valued result.
43. What is the purpose of IntToLongFunction functional interface?
44. It represents a function that accepts an int-valued argument and produces a long-valued result.
45. What is the purpose of IntUnaryOperator functional interface?
46. It represents an operation on a single int-valued operand that produces an int-valued result.
47. What is the purpose of LongBinaryOperator functional interface?
48. It represents an operation upon two long-valued operands and produces a long-valued result.
49. What is the purpose of LongConsumer functional interface?
50. It represents an operation that accepts a single long-valued argument and returns no result.
51. What is the purpose of LongFunction<R> functional interface?
52. It represents a function that accepts a long-valued argument and produces a result.
53. What is the purpose of LongPredicate functional interface?
54. It represents a predicate (Boolean-valued function) of one long-valued argument.
55. What is the purpose of LongSupplier functional interface?
56. It represents a supplier of long-valued results.
57. What is the purpose of LongToDoubleFunction functional interface?
58. It represents a function that accepts a long-valued argument and produces a double-valued result.
59. What is the purpose of LongToIntFunction functional interface?
60. It represents a function that accepts a long-valued argument and produces an int-valued result.
61. What is the purpose of LongUnaryOperator functional interface?
62. It represents an operation on a single long-valued operand that produces a long-valued result.
63. What is the purpose of ObjDoubleConsumer<T> functional interface?
64. It represents an operation that accepts an object-valued and a double-valued argument, and returns no result.
65. What is the purpose of ObjIntConsumer<T> functional interface?
66. It represents an operation that accepts an object-valued and an int-valued argument, and returns no result.
67. What is the purpose of ObjLongConsumer<T> functional interface?
68. It represents an operation that accepts an object-valued and a long-valued argument, and returns no result.
69. What is the purpose of Predicate<T> functional interface?
70. It represents a predicate (Boolean-valued function) of one argument.
71. What is the purpose of Supplier<T> functional interface?
72. It represents a supplier of results.
73. What is the purpose of ToDoubleBiFunction<T,U> functional interface?
74. It represents a function that accepts two arguments and produces a double-valued result.
75. What is the purpose of ToDoubleFunction<T> functional interface?
76. It represents a function that produces a double-valued result.
77. What is the purpose of ToIntBiFunction<T,U> functional interface?
78. It represents a function that accepts two arguments and produces an int-valued result.
79. What is the purpose of ToIntFunction<T> functional interface?
80. It represents a function that produces an int-valued result.
81. What is the purpose of ToLongBiFunction<T,U> functional interface?
82. It represents a function that accepts two arguments and produces a long-valued result.
83. What is the purpose of ToLongFunction<T> functional interface?
84. It represents a function that produces a long-valued result.
85. What is the purpose of UnaryOperator<T> functional interface?
86. It represents an operation on a single operand that produces a result of the same type as its operand.
87. What are default methods?
88. With java 8, an interface can have default implementation of a function in interfaces.
89. What are static default methods?
90. An interface can also have static helper methods from Java 8 onwards.
91. public interface vehicle {
92. default void print(){
93. System.out.println("I am a vehicle!");
94. }
96. static void blowHorn(){
97. System.out.println("Blowing horn!!!");
98. }
99. }
100. How will you call a default method of an interface in a class?
101. Using super keyword along with interface name.
102. interface Vehicle {
103. default void print(){
104. System.out.println("I am a vehicle!");
105. }
106. }
107. class Car implements Vehicle {
108. public void print(){
109. Vehicle.super.print();
110. }
111. }
112. How will you call a static method of an interface in a class?
113. Using name of the interface.
114. interface Vehicle {
115. static void blowHorn(){
116. System.out.println("Blowing horn!!!");
117. }
118. }
119. class Car implements Vehicle {
120. public void print(){
121. Vehicle.blowHorn();
122. }
123. }
124. What is streams in Java 8?
125. Stream represents a sequence of objects from a source, which supports aggregate operations.
126. What is stream pipelining in Java 8?
127. Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.
128. What is the difference between Collections and Stream in Java8 ?
129. Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.
130. What is the purpose of forEach method of stream in java 8?
131. Stream has provided a new method 'forEach' to iterate each element of the stream.
132. How will you print 10 random numbers using forEach of java 8?
133. The following code segment shows how to print 10 random numbers using forEach.
134. Random random = new Random();
135. random.ints().limit(10).forEach(System.out::println);
136. What is the purpose of map method of stream in java 8?
137. The 'map' method is used to map each element to its corresponding result.
138. How will you print unique squares of numbers in java 8?
139. The following code segment prints unique squares of numbers using map.
140. List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
141. //get list of unique squares
142. List<Integer> squaresList = numbers.stream().map( i -> i\*i).distinct().collect(Collectors.toList());
143. What is the purpose of filter method of stream in java 8?
144. The 'filter' method is used to eliminate elements based on a criteria.
145. How will you print count of empty strings in java 8?
146. The following code segment prints a count of empty strings using filter.
147. List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
148. //get count of empty string
149. int count = strings.stream().filter(string −> string.isEmpty()).count();
150. What is the purpose of limit method of stream in java 8?
151. The 'limit' method is used to reduce the size of the stream.
152. How will you print 10 random numbers in java 8?
153. The following code segment shows how to print 10 random numbers.
154. Random random = new Random();
155. random.ints().limit(10).forEach(System.out::println);
156. What is the purpose of sorted method of stream in java 8?
157. The 'sorted' method is used to sort the stream.
158. How will you print 10 random numbers in a sorted order in java 8?
159. The following code segment shows how to print 10 random numbers in a sorted order.
160. Random random = new Random();
161. random.ints().limit(10).sorted().forEach(System.out::println);
162. What is Parallel Processing in Java 8?
163. parallelStream is the alternative of stream for parallel processing. Take a look at the following code segment that prints a count of empty strings using parallelStream.
164. List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
165. //get count of empty string
166. int count = strings.parallelStream().filter(string −> string.isEmpty()).count();
167. //It is very easy to switch between sequential and parallel streams.
168. What are collectors in Java 8?
169. Collectors are used to combine the result of processing on the elements of a stream. Collectors can be used to return a list or a string.
170. List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
171. List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());
172. System.out.println("Filtered List: " + filtered);
173. String mergedString = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.joining(", "));
174. System.out.println("Merged String: " + mergedString);
175. What are Statistics collectors in Java 8?
176. With Java 8, statistics collectors are introduced to calculate all statistics when stream processing is being done.
177. How will you get the highest number present in a list using Java 8?
178. Following code will print the highest number present in a list.
179. List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
180. IntSummaryStatistics stats = integers.stream().mapToInt((x) −> x).summaryStatistics();
181. System.out.println("Highest number in List : " + stats.getMax());
182. How will you get the lowest number present in a list using Java 8?
183. Following code will print the highest number present in a list.
184. List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
185. IntSummaryStatistics stats = integers.stream().mapToInt((x) −> x).summaryStatistics();
186. System.out.println("Lowest number in List : " + stats.getMin());
187. How will you get the sum of all numbers present in a list using Java 8?
188. Following code will print the sum of all numbers present in a list.
189. List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
190. IntSummaryStatistics stats = integers.stream().mapToInt((x) −> x).summaryStatistics();
191. System.out.println("Sum of all numbers : " + stats.getSum());
192. How will you get the average of all numbers present in a list using Java 8?
193. Following code will print the average of all numbers present in a list.
194. List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
195. IntSummaryStatistics stats = integers.stream().mapToInt((x) −> x).summaryStatistics();
196. System.out.println("Average of all numbers : " + stats.getAverage());
197. What is Optional in Java8?
198. Optional is a container object which is used to contain not-null objects. Optional object is used to represent null with absent value. This class has various utility methods to facilitate code to handle values as 'available' or 'not available' instead of checking null values. It is introduced in Java 8 and is similar to what Optional is in Guava.
199. What is Nashorn in Java8?
200. With Java 8, Nashorn, a much improved javascript engine is introduced, to replace the existing Rhino. Nashorn provides 2 to 10 times better performance, as it directly compiles the code in memory and passes the bytecode to JVM. Nashorn uses invokedynamics feature, introduced in Java 7 to improve performance.
201. What is jjs in JAVA8?
202. For Nashorn engine, JAVA 8 introduces a new command line tool, jjs, to execute javascript codes at console.
203. Can you execute javascript code from java 8 code base?
204. Yes! Using ScriptEngineManager, JavaScript code can be called and interpreted in Java.
205. What is local datetime API in JAVA8?
206. Local − Simplified date-time API with no complexity of timezone handling.
207. What is zoned datetime API in JAVA8?
208. Zoned − Specialized date-time API to deal with various timezones.
209. What is chromounits in java8?How will you get the current date using local datetime api of java8?How will you add 1 week to current date using local datetime api of java8?
210. Following code adds 1 week to current date using local datetime api −
211. //add 1 week to the current date
212. LocalDate today = LocalDate.now();
213. LocalDate nextWeek = today.plus(1, ChronoUnit.WEEKS);
214. System.out.println("Next week: " + nextWeek);
215. How will you add 1 month to current date using local datetime api of java8?
216. Following code adds 1 month to current date using local datetime api:
217. //add 1 month to the current date
218. LocalDate today = LocalDate.now();
219. LocalDate nextMonth = today.plus(1, ChronoUnit.MONTHS);
220. System.out.println("Next month: " + nextMonth);
221. How will you add 1 year to current date using local datetime api of java8?
222. Following code adds 1 year to current date using local datetime api −
223. //add 1 year to the current date
224. LocalDate today = LocalDate.now();
225. LocalDate nextYear = today.plus(1, ChronoUnit.YEARS);
226. System.out.println("Next year: " + nextYear);
227. How will you add 10 years to current date using local datetime api of java8?
228. Following code adds 10 years to current date using local datetime api −
229. //add 10 years to the current date
230. LocalDate today = LocalDate.now();
231. LocalDate nextDecade = today.plus(1, ChronoUnit.DECADES);
232. System.out.println("Date after ten year: " + nextDecade);
233. How will you get next tuesday using java8?
234. Following code gets next tuesday using java8 −
235. //get the next tuesday
236. LocalDate today = LocalDate.now();
237. LocalDate nextTuesday = today.with(TemporalAdjusters.next(DayOfWeek.TUESDAY));
238. System.out.println("Next Tuesday on : " + nextTuesday);
239. How will you get second saturday of next month using java8?
240. Following code gets second saturday of next month using java8 −
241. //get the second saturday of next month
242. LocalDate firstInYear = LocalDate.of(date1.getYear(),date1.getMonth(), 1);
243. LocalDate secondSaturday = firstInYear.with(TemporalAdjusters.nextOrSame(DayOfWeek.SATURDAY)).with(TemporalAdjusters.next(DayOfWeek.SATURDAY));
244. System.out.println("Second Saturday on : " + secondSaturday);
245. How will you get the instant of current date in terms of milliseconds using java8?
246. Following code gets the instant of current date in terms of milliseconds −
247. //Get the instant of current date in terms of milliseconds
248. Instant now = currentDate.toInstant();
249. How will you get the instant of local date time using time in of milliseconds using java8?
250. Following code gets the instant of local date time using time in of milliseconds −
251. Instant now = currentDate.toInstant();
252. ZoneId currentZone = ZoneId.systemDefault();
253. LocalDateTime localDateTime = LocalDateTime.ofInstant(now, currentZone);
254. System.out.println("Local date: " + localDateTime);
255. How will you get the instant of zoned date time using time in of milliseconds using java8?
256. Following code gets the instant of zoned date time using time in of milliseconds −
257. Instant now = currentDate.toInstant();
258. ZoneId currentZone = ZoneId.systemDefault();
259. ZonedDateTime zonedDateTime = ZonedDateTime.ofInstant(now, currentZone);
260. System.out.println("Zoned date: " + zonedDateTime);
261. Which class implements a decoder for decoding byte data using the Base64 encoding scheme in Java8?
262. static class Base64.Decoder − This class implements a decoder for decoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045.
263. Which class implements an encoder for encoding byte data using the Base64 encoding scheme in Java8?
264. static class Base64.Encoder − This class implements an encoder for encoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045.
265. How will you create a Base64 decoder?
266. getDecoder() method of Base64 class returns a Base64.Decoder that decodes using the Basic type base64 encoding scheme.
267. How will you create a Base64 encoder?
268. getEncoder() method of Base64 class returns a Base64.Encoder that encodes using the Basic type base64 encoding scheme.
269. How will you create a Base64 decoder that decodes using the MIME type base64 encoding scheme?
270. getMimeDecoder() method of Base64 class returns a Base64.Decoder that decodes using the MIME type base64 decoding scheme.
271. How will you create a Base64 encoder that encodes using the MIME type base64 encoding scheme?
272. getMimeEncoder() method of Base64 class returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme.
273. How will you create a Base64 decoder that decodes using the URL and Filename safe type base64 encoding scheme?
274. getUrlDecoder() method of Base64 class returns a Base64.Decoder that decodes using the URL and Filename safe type base64 encoding scheme.
275. How will you create a Base64 encoder that encodes using the URL and Filename safe type base64 encoding scheme?
276. getUrlEncoder() method of Base64 class returns a Base64.Encoder that encodes using the URL and Filename safe type base64 encoding scheme.

**Should we create system software ( e.g Operating system ) in Java ?**

Ans. No, Java runs on a virtual machine called JVM and hence doesn't embed well with the underlying hardware. Though we can create a platform independent system software but that would be really slow and that's what we would never need.

**Q2.  What are the different types of memory used by JVM ?**

Ans. Class , Heap , Stack , Register , Native Method Stack.

**Q3.  What are the benefits of using Spring Framework ?**

Ans. Spring enables developers to develop enterprise-class applications using POJOs. The benefit of using only POJOs is that you do not need an EJB container product.

Spring is organized in a modular fashion. Even though the number of packages and classes are substantial, you have to worry only about ones you need and ignore the rest.

Spring does not reinvent the wheel instead, it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, other view technologies.

Testing an application written with Spring is simple because environment-dependent code is moved into this framework. Furthermore, by using JavaBean-style POJOs, it becomes easier to use dependency injection for injecting test data.

Spring’s web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over engineered or less popular web frameworks.

Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.

Lightweight IoC containers tend to be lightweight, especially when compared to EJB containers, for example. This is beneficial for developing and deploying applications on computers with limited memory and CPU resources.

Spring provides a consistent transaction management interface that can scale down to a local transaction

**Q4.  What are various types of Class loaders used by JVM ?**

Ans. Bootstrap - Loads JDK internal classes, java.\* packages.

Extensions - Loads jar files from JDK extensions directory - usually lib/ext directory of the JRE

System  - Loads classes from system classpath.

**Q5.  What is PermGen or Permanent Generation ?**

Ans. The memory pool containing all the reflective data of the java virtual machine itself, such as class and method objects. With Java VMs that use class data sharing, this generation is divided into read-only and read-write areas. The Permanent generation contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.

**Q6.  What is metaspace ?**

Ans. The Permanent Generation (PermGen) space has completely been removed and is kind of replaced by a new space called Metaspace. The consequences of the PermGen removal is that obviously the PermSize and MaxPermSize JVM arguments are ignored and you will never get a java.lang.OutOfMemoryError: PermGen error.

**Q7.  How does volatile affect code optimization by compiler?**

Ans. Volatile is an instruction that the variables can be accessed by multiple threads and hence shouldn't be cached. As volatile variables are never cached and hence their retrieval cannot be optimized.

**Q8.  What things should be kept in mind while creating your own exceptions in Java?**

Ans. All exceptions must be a child of Throwable.

If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.

You want to write a runtime exception, you need to extend the RuntimeException class.

**Q9.  What is the best practice configuration usage for files - pom.xml or settings.xml ?**

Ans. The best practice guideline between settings.xml and pom.xml is that configurations in settings.xml must be specific to the current user and that pom.xml configurations are specific to the project.

**Q10.  Can you provide some implementation of a Dictionary having large number of words ?**

Ans. Simplest implementation we can have is a List wherein we can place ordered words and hence can perform Binary Search.

Other implementation with better search performance is to use HashMap with key as first character of the word and value as a LinkedList.

Further level up, we can have linked Hashmaps like ,

hashmap {

a ( key ) -> hashmap (key-aa , value (hashmap(key-aaa,value)

b ( key ) -> hashmap (key-ba , value (hashmap(key-baa,value)

....................................................................................

z( key ) -> hashmap (key-za , value (hashmap(key-zaa,value)

}

upto n levels ( where n is the average size of the word in dictionary.

**Q11.  What is database deadlock ? How can we avoid them?**

Ans. When multiple external resources are trying to access the DB locks and runs into cyclic wait, it may makes the DB unresponsive.

Deadlock can be avoided using variety of measures, Few listed below -

Can make a queue wherein we can verify and order the request to DB.

Less use of cursors as they lock the tables for long time.

Keeping the transaction smaller.

**Q12.  Why Web services use HTTP as the communication protocol ?**

Ans. With the advent of Internet, HTTP is the most preferred way of communication. Most of the clients ( web thin client , web thick clients , mobile apps )  are designed to communicate using http only. Web Services using http makes them accessible from vast variety of client applications.

**Q13.  Why using cookie to store session info is a better idea than just using session info in the request ?**

Ans. Session info in the request can be intercepted and hence a vulnerability. Cookie can be read and write  by respective domain only and make sure that right session information is being passed by the client.

**Q14.  Difference between first level and second level cache in hibernate ?**

Ans. 1. First level cache is enabled by default whereas Second level cache needs to be enabled explicitly.

2. First level Cache came with Hibernate 1.0 whereas Second level cache came with Hibernate 3.0.

3. First level Cache is Session specific whereas Second level cache is shared by sessions that is why First level cache is considered local and second level cache is considered global.

**Q15.  What are the ways to avoid LazyInitializationException ?**

Ans. 1. Set lazy=false in the hibernate config file.

2. Set @Basic(fetch=FetchType.EAGER) at the mapping.

3. Make sure that we are accessing the dependent objects before closing the session.

4. Using Fetch Join in HQL.

**Q16.  What are new features introduced with Java 8 ?**

Ans. Lambda Expressions , Interface Default and Static Methods , Method Reference , Parameters Name , Optional , Streams, Concurrency.

**Q17.  What things you would care about to improve the performance of Application if its identified that its DB communication that needs to be improved ?**

Ans. 1. Query Optimization ( Query Rewriting , Prepared Statements )

2. Restructuring Indexes.

3. DB Caching Tuning ( if using ORM )

4. Identifying the problems ( if any ) with the ORM Strategy ( If using ORM )

**Q18.  If you are given a choice to implement the code to either Insert a Record or Update if already exist, Which approach will you follow ?**

1. Insert into the DB Table. If exception occurs, update the existing record.

2. Check if the record exists and update it if it exists, If not insert a new record.

Ans. In first case, there would be 2 DB calls in worst case and 1 in best case. In 2nd approach there will be always 2 DB calls.

Decision on the approach should depend on the following considerations -

1. How costly is the call to DB ? Are we using indices , hibernate etc

If calls to DB are costly , 1st approach should be the choice.

2. Exception Book keeping load upon exception.

The benefit of saving 1st call in approach 1 should be bigger than the Book keeping for the exception.

3. Probability of the exception in first apparoach.

If the DB Table is almost empty, it makes sense to follow Approach 1 as majority of the 1st calls will pass through without exception.

**Q19.  What would you do if you have to add a jar to the project using Maven ?**

Ans. If its already there in Maven local repository, We can add that as a dependency in the project pom file with its Group Id, Artifact Id and version.

We can provide additional attribute SystemPath if its unable to locate the jar in the local repository.

If its not there in the local repository, we can install it first in the local repository and then can add it as dependency.

**Q20.  Should we create system software ( e.g Operating system ) in Java ?**

Ans. No, Java runs on a virtual machine called JVM and hence doesn't embed well with the underlying hardware. Though we can create a platform independent system software but that would be really slow and that's what we would never need.

**Q21.  Which UML diagrams you usually use for design ?**

Ans. Use Case Diagram, Component Diagram for High level Design and Class Diagram , Sequence Diagram for low level design.

**Q22.  How do you coordinate and communicate with the team developers ?**

Ans. We as a team of developers , testers , analyst , lead and architect sit close to each other. Most of the time I would just jump to their seat and talk to them ( if required ). We have daily stand up where we discuss things that needs team attention.

**Q23.  What kind of software architecture your organization follow ?**

Ans. We have multi tier architecture with multiple layers , We have series of web servers and applications in application tier, infrastructure libraries at middle tier and Database servers at the lower tier. We are using Oracle as Database, ESB ( Enterprise service Bus ) for asynchronous communication and Rest Web Services.

**Q24.  Difference between Proxy and Adapter Deisgn Patterns ?**

Ans. Adapter object has a different input than the real subject whereas Proxy object has the same input as the real subject. Proxy object is such that it should be placed as it is in place of the real subject.

**Q25.  Difference between Adapter and Facade ?**

Ans. The Difference between these patterns in only the intent. Adapter is used because the objects in current form cannot communicate where as in Facade , though the objects can communicate , A Facade object is placed between the client and subject to simplify the interface.

**Q26.  Difference between Builder and Composite ?**

Ans. Builder is a creational Design Pattern whereas Composite is a structural design pattern. Composite creates Parent - Child relations between your objects while Builder is used to create group of objects of predefined types.

**Q27.  Difference between Factory and Strategy Design Pattern ?**

Ans. Factory is a creational design pattern whereas Strategy is behavioral design pattern. Factory revolves around the creation of object at runtime whereas Strategy or Policy revolves around the decision at runtime.

**Q28.  Shall we use abstract classes or Interfaces in Policy / Strategy Design Pattern ?**

Ans. Strategy deals only with decision making at runtime so Interfaces should be used.

Q-3 : What is a functional interface ?

Ans:

A functional interface is an interface with only one abstract method.

The main advantage of functional interface is that it can be used to refer to a lambda expression or as a method interface.

These are needed to execute a lambda expression or method.  
   
[Read more on Functional Interface in Java 8](http://www.topjavatutorial.com/java-8/functional-interface-in-java8/)

Q-4 : What is the type of a lambda expression ?

Ans:

The type of a lambda expression depends on the context it is being used.

(int x,int y) -> {return x+y;}

Often, the type is inferred by the Java compiler.

Q-5 : What is the target type of a lambda expression ?

Ans:

The target type of a lambda expression represents a type to which the expression can be converted.

All lambda expressions are converted to a functional interface type. So, the target type is functional interface.

Q-6: What is the difference between an interface with default method and an abstract class ?

Ans:

When we add default method to an interface, it looks like an abstract class, but they are not the same.

An abstract class can have constructors, instance variables, concrete methods, but we can’t have instance variables or constructors in the interface.

An interface with single default method can be used to refer to a lambda expression, but an abstract class cannot be used to refer to lambda expressions.

Q-7: Are the abstract classes still useful?

Abstract classes can still do more in comparison to Java 8 interfaces:

1. Abstract class can have a constructor. The interface has no constructors to be invoked by the descendants

1. Abstract classes are more structured and can hold a state.  
   In comparison, Interface methods are all public, field members are all constants (final & public). You may want to restrict access privileges of methods and/or make them operate on non-constant state.

1. A child class can call upon the abstract class method(s) by super, while it can not do so on default interface methods.

1. Type clarity:

You can only extend one class. This makes it clearer what your object is an how to use it.

1. The diamond problem:

The problem occurs when you inherit from two places that declare the same method, and you have to pick one when resolving a function call.

You have to implement all of the default methods in an interface to help avoid the diamond problem. If the interface is Collections, this can be a huge pain since there are a billion of them. With an abstract class, you only overwrite what you need to.

Q-8 : What is the difference between a predicate and a function ?

Ans :

A predicate takes one argument and returns a boolean value.

A function takes one argument and returns an object.

Both are useful for evaluating lambda expressions.

Q-9 : What is the difference between a collection and a stream ?

Ans :

A collection contains group of objects in memory.

A stream is specific to lambda expressions. Stream contains objects taken from

collection that can be easily acted on using lambda expressions.

Q-10 : What is the difference between intermediate and terminal operations in streams ?

Ans:

Intermediate operations return a stream as the result of the operation.

Terminal operations return a value as result.