Java-8

1. **What are the new features released in Java 8?**

* Lambda Expressions
* Functional Interfaces
* [Stream API](https://www.journaldev.com/2774/java-8-stream)
* Date and Time API
* [Interface Default Methods and Static Methods](https://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method)
* Spliterator
* Method and Constructor References
* Collections API Enhancements
* [Concurrency](https://www.journaldev.com/1162/java-multithreading-concurrency-interview-questions-answers) 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

1. **What are the main benefits of new features introduced in Java 8?**

## Lambda Expressions

Lambda expressions gives the ability to pass a functionality as a method argument. Lambda expression help us reduce the code clutter in using a single method class. For example, when we have to associate an action button click to a functionality, then lambda expressions will make the code look neat and compact. Refer [Java Lambda expression examples](https://javapapers.com/core-java/java-lambda-expression-examples/) to learn more.

## Pipelines and Streams

Pipelines and streams enrich the Java collections framework. Sequence of aggregate operations is a pipeline. Stream is used to propagate elements from a source through a pipeline. It is a sequence of elements. Pipeline and streams will make our task easier in accessing the elements from collections and applying operations on it.

## Date and Time API

Pre Java 8, date and time related operations are little bit cumbersome. JSR 310: Date and Time API give us a brand new package [java.time package](http://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html). This is a well thought package. It contains a best list of utility methods used for regular operations. This will help in handling date and time operations in an easier way.

## Default Methods

Default methods gives the ability to add default implementation for methods in an interface. This is a rocking feature of Java 8. When we implement an interface, if we did not provide implementation for a method, then the default method will be used in that place. This will help in maintaining backward compatibility in applications, we can add a method to an interface without breaking its implementations.

## Type Annotations

Before Java 8 [Java annotations](https://javapapers.com/core-java/java-annotations/) can be applied to type declarations. From this Java 8 release onwards, annotations can be applied to type use. Annotations can be applied wherever a type is used like in new instance creates, exception throws clause etc. This will help to enforce stronger type checks and using this feature we can come up with a type checking framework itself.

## Nashorn JavaScript Engine

Nashorn is a brand new JavaScript engine provided along with the Java 8 release. Using this we can develop standalone JavaScript applications in Java. Pre Java 8, we got JDK with a JavaScript engine based on Rhino. It is a developed from scratch. It will provide better compatibility with ECMA normalized JavaScript specification and better performance than Rhino. Already we have seen a tutorial to [run Javascript in Java using the ScriptEngineManager](https://javapapers.com/core-java/run-javascript-from-java/).

## Concurrent Accumulators

java.util.concurrent.atomic package is getting additional classes as part of Java 8 release. These will help to sum values from across multiple threads.

## Parallel operations

Iterating over a collection can be done in parallel using the aggregate operations easily. Pre Java 8 Iterators are used to parse the elements of a collection one by on explicitly. Now that can be done in parallel internally with the use of streams and aggregate operations. We can create multiple substreams and those substreams can be processed internally in parallel and then the results be combined. For this to be effective, we need to have multiple cores and data volume.

## PermGen Space Removed

The PermGen space is removed from Java 8 and instead we have MetaSpace introduced. One of the most dreaded error, “java.lang.OutOfMemoryError: PermGen error” will no longer be seen from Java 8. Nice thing is that the MetaSpace default is unlimited and that the system memory itself becomes the memory.

## TLS SNI

Server Name Indentification (SNI) is an extension of the TLS protocol used for identifying the certificate to serve based on the hostname specified by the client. This enables a server to have virtual host over HTTPS. The same IP address can be used for multiple hosts and their respective different security certificates.

The above list of Java 8 features is just a highlight. I will be writing detailed tutorials about each of them in the coming days.

1. **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

1. **What is a Lambda expression in Java 8?**

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.

1. **What are the three main parts of a Lambda expression in Java?**

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](https://www.journaldev.com/2763/java-8-functional-interfaces).

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

() -> 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.

1. **What is the data type of a Lambda expression?**

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](https://www.journaldev.com/2763/java-8-functional-interfaces).

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

() -> 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.

1. **What is the meaning of following lambda expression?**
2. **Why did Oracle release a new version of Java like Java 8?**

Oracle Corporation has introduced a lot of new concepts in Java SE 8 to introduce the following benefits:

* **To Utilize Current Multi-Core CPUs Efficiently**

Recently, we can observe drastic changes in Hardware. Now-a-days, all systems are using Multi-Core CPUs (2,4,8,16-Core etc.) to deploy and run their Applications. We need new Programming Constructs in Java to utilize these Multi-Core Processors efficiently to develop Highly Concurrently and Highly Scalable applications.

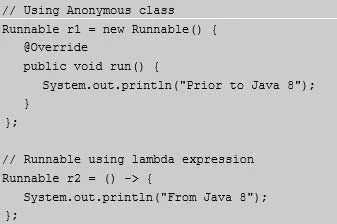
* **To Utilize FP Features**

Oracle Corporation has introduced a lot of FP(Functional Programming) concepts as part of Java SE 8 to utilize the advantages of FP.

1. **What are the advantages of a lambda expression?**

**1. A Fewer Lines of Code**

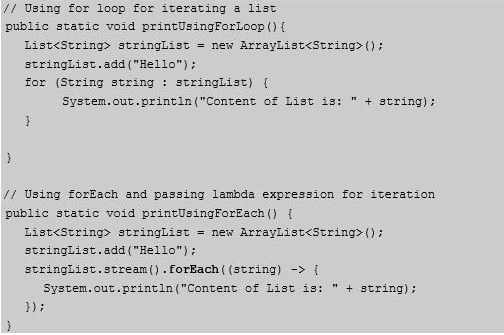
One of the benefits of using lambda expression is the reduced amount of code. See the example below.



* We know that in Java lambda can be used only with functional interfaces. In the above example, Runnable is a functional interface, so we can easily apply lambda expression here
* In this case, we are not passing any parameter in lambda expression because the run() method of the functional interface (Runnable) takes no argument.
* Also, the syntax of the lambda expression says that we can omit curly braces ({}) in case of a single statement in the method body. In case of multiple statements, we should use curly braces as done in the above example.

2. Sequential and Parallel Execution Support by passing behavior in methods

Prior to Java 8, processing the elements of any collection could be done by obtaining an iterator from the collection and then iterating over the elements and then processing each element. If the requirement is to process the elements in parallel, it would be done by the client code. With the introduction of Stream API in Java 8, functions can be passed to collection methods and now it is the responsibility of collection to process the elements either in a sequential or parallel manner.

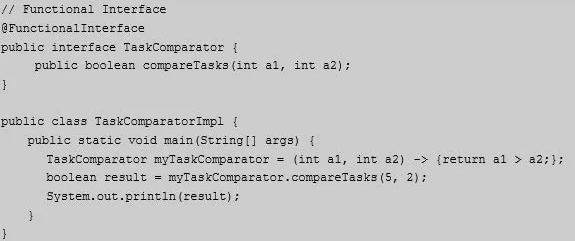


3. Higher Efficiency (Utilizing Multicore CPU’s)

Using Stream API’s and lambda expression we can achieve higher efficiency (parallel execution) in case of bulk operations on collections. Also, the lambda expressions can help in achieving internal iteration of collections rather than external iteration as shown in the above example. As nowadays we have CPUs with multicores, we can take advantage of these multicore CPU’s by parallel processing of collections using lambda.

**Lambda Expression and Objects**

In Java, any lambda expression is an object as is an instance of a functional interface. We can assign a lambda expression to any variable and pass it like any other object. See the example below on how a lambda expression is assigned to a variable, and how it is invoked.



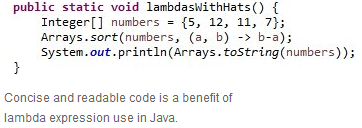
**Where you can use Lambda expressions**

Lambda expressions can be used anywhere in Java 8 where we have a target type. In Java, we have target type in the following contexts

* Variable declarations and assignments
* Return statements
* Method or constructor arguments

### Lambda expression benefits and drawbacks

With lambda expressions, the goal is to minimize as many of the drawbacks to using a single class or anonymous inner class when implementing a functional interface, while at the same time maximizing the benefits. If you look at the Comparator example implemented using a lambda expression in Java, there’s no debating the fact that the code is much more concise and compact.



Concise and readable code is a benefit of lambda expression use in Java.

Furthermore, there is no potential for variable shadowing with lambda expressions, eliminating a major downside when using inner classes. It should also be noted that even inner classes create a new .class file when the application is packaged and deployed, something that doesn’t happen with lambda expressions, providing the minor benefit that the deployment artifacts that get created are marginally more compact.

This simple example of lambda expressions is only sufficient to demonstrate a few of the benefits of lambda expressions in Java as compared to other approaches. As you explore lambda expressions at a more advanced level, you will discover how they also make iterative processing easier using the new, functional forEach method. And because lambda expressions can be assigned to a variable, they provide an opportunity for reuse that simply isn’t possible with anonymous inner classes.

It took a long time for lambda expression to finally make it into the Java language, but given the benefits of using lambda expressions in Java code, it’s no wonder that developers are excited when their target deployment platform gets upgraded to a newer version of the JDK.

1. **What is a Functional interface in Java 8?**

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.

1. **What is a Single Abstract Method (SAM) interface in Java 8?**

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

1. **How can we define a Functional interface in Java 8?**

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.

1. **Why do we need Functional interface in Java?**

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.

1. **Is it mandatory to use @FunctionalInterface annotation to define a Functional interface in Java 8?**

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.

1. **What are the main uses of Stream API in Java 8?**

When our Java project wants to perform the following operations, it’s better to use [Java 8 Stream](https://www.journaldev.com/2774/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.

1. **What are the differences between Collection and Stream API in Java 8?**

|  |  |  |
| --- | --- | --- |
|  | Collection API | Stream API |
| 1. | It’s available since Java 1.2 | It is introduced in Java SE8 |
| 2. | It is used to store Data(A set of Objects). | It is used to compute data(Computation on a set of Objects). |
| 3. | We can use both Spliterator and Iterator to iterate elements. We can use [forEach](https://www.journaldev.com/13941/java-foreach-java-8-foreach) to performs an action for each element of this stream. | We can’t use Spliterator or Iterator to iterate elements. |
| 4. | It is used to store limited number of Elements. | It is used to store either Limited or Infinite Number of Elements. |
| 5. | Typically, it uses External Iteration concept to iterate Elements such as Iterator. | Stream API uses internal iteration to iterate Elements, using forEach methods. |
| 6. | Collection Object is constructed Eagerly. | Stream Object is constructed Lazily. |
| 7. | We add elements to Collection object only after it is computed completely. | We can add elements to Stream Object without any prior computation. That means Stream objects are computed on-demand. |
| 8. | We can iterate and consume elements from a Collection Object at any number of times. | We can iterate and consume elements from a Stream Object only once. |

1. **What are the differences between Intermediate and Terminal Operations in Java 8 Streams?**

|  |  |  |
| --- | --- | --- |
| 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. |

1. **What is a Spliterator in Java 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.

1. **What are the differences between Iterator and Spliterator in Java 8?**

|  |  |  |
| --- | --- | --- |
| 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 Iterator 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() |

1. **What is Type Inference in Java 8?**

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

1. **Does Java 7 support Type Inference?**

Yes, 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:

String str[] = { "Java 7", "Java 8", "[Java 9](https://www.journaldev.com/13121/java-9-features-with-examples)" };

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.

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.

ToIntBiFunction<Integer,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 as Integers.

1. **How does Internal Iteration work in Java 8?**

Internal Iterators(or Passive Iterators) – Internal Iterators manage the iterations in the background. This leaves the programmer to just declaratively code what is meant to be done with the elements of the Collection, rather than managing the iteration and making sure that all the elements are processed one-by-one.

Lets see how simple it is to say print all elements in an ArrayList in Java 8 using an **example of internal iterator based forEach loop** –

import java.util.\*;

public class InternalIterator {

  public static void main(String args[]){

  List<String> namesList=Arrays.asList("Tom", "Dick", "Harry");

    namesList.forEach(name -> System.out.println(name));//Internal Iteration

  }

}

In the above code, we are just telling the forEach method what to do(i.e. print) with each String in the namesList list using a **lambda expression**( In case you are not familiar with lambda expressions you can read the [tutorial here](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/)).

All the work of iterating over the list of names one by one and printing them is taken care of internally by the runtime, leaving us with just declaratively defining only what is to be done i.e. print the names.

**Internal Iteration Using Streams**  
Its important to understand what role the internal iterator plays in the Java 8 scheme of things. In the previous code example for forEach method, we could have made use of the java 8 Streams API to first create a Stream of elements in the List and then pipeline the stream into a forEach method like this –

namesList.stream().forEach(name -> System.out.println(name));//Internal Iteration

The forEach method comes under the category of [terminal operations](https://www.javabrahman.com/java-8/understanding-java-8-streams-operations-intermediate-and-terminal-operations-tutorial-with-examples/) when working with streams i.e. it ends the chain of pipelined operations to produce a result.  
Note – You can also use internal iterators with parallelStream() method instead of a stream to improve on the performance.

1. **What are the main differences between Internal and External Iterator?**

|  |  |  |
| --- | --- | --- |
| 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. |

1. **What are the main advantages of Internal Iterator over External Iterator in Java 8?**

* 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.

1. **What are the applications in which we should use Internal 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.

1. **What is the main disadvantage of Internal Iteration over External Iteration?**

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

If we want to perform some check or operation for a particular index then external iterators are preferred over internal ones.

1. **Can we provide implementation of a method in a Java Interface?**

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

1. **What is a Default Method in an Interface?**

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

1. **Why do we need Default method in a Java 8 Interface?**

* 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.

1. **What is the purpose of a Static method in an Interface in Java 8?**

* 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.

1. **What are the core ideas behind the Date/Time API of Java 8?**

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.

1. **What are the advantages of new Date and Time API in Java 8 over old Date API?**

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

1. **What are the main differences between legacy Date/Time API in Java and Date/Time API of Java 8?**

|  |  |  |
| --- | --- | --- |
| 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. |

1. **How can we get duration between two dates or time in Java 8?**

**If you want logical calendar days, use** [**DAYS.between()**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoUnit.html#between-java.time.temporal.Temporal-java.time.temporal.Temporal-) **method from** [**java.time.temporal.ChronoUnit**](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoUnit.html):

LocalDate dateBefore;

LocalDate dateAfter;

long daysBetween = DAYS.between(dateBefore, dateAfter);

**If you want literal 24 hour days, (a duration), you can use the** [**Duration**](https://docs.oracle.com/javase/8/docs/api/java/time/Duration.html) **class instead:**

LocalDate today = LocalDate.now()

LocalDate yesterday = today.minusDays(1);

// Duration oneDay = Duration.between(today, yesterday); // throws an exception

Duration.between(today.atStartOfDay(), yesterday.atStartOfDay()).toDays() // another option

1. **What is the new method family introduced in Java 8 for processing of Arrays on multi core machines?**

Java 8 has enhanced the Arrays class with methods that can run efficiently on multi core machines.

These methods start with keyword parallel.  
Egg. Arrays.parallelSetAll(), Arrays.parallelSort() etc.

This parallel set of methods provides parallel processing of Arrays that can run Java code very fast on a multi core machine.

1. **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.

1. **How does Java 8 solve Diamond problem of Multiple Inheritance?**

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:

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:

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.

1. **What are the differences between Predicate, Supplier and Consumer in Java 8?**

**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.

1. **Is it possible to have default method definition in an interface without marking it with default keyword?**

No, Compiler complains that its an abstract method and hence shouldn't have the body.

1. **Can we create a class that implements two Interfaces with default methods of same name and signature?**

No, Compiler gives error saying "Duplicate Default Methods"

1. **Can we access a static method of an interface by using reference of the interface?**

No, only using Interface Name.

1. **How can you get the name of Parameter in Java by using reflection?**
2. **What is Optional in Java 8?**

Java 8 introduced a new **container class java.util.Optional<T>**. It wraps a single value, if that value is available. If the value is not available an empty optional should be returned. Thus it represents null value with absent value. This class has various utility methods like isPresent() which helps users to avoid making use of null value checks. So instead of returning the value directly, a wrapper object is returned thus users can avoid the null pointer exception.

1. **What is Optional in Java 8?**

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

Java Optional Class : Every Java Programmer is familiar with [NullPointerException](https://www.geeksforgeeks.org/null-pointer-exception-in-java/). It can crash your code. And it is very hard to avoid it without using too many null checks.  
Java 8 has introduced a new class Optional in java.util package. It can help in writing a neat code without using too many null checks. By using Optional, we can specify alternate values to return or alternate code to run. This makes the code more readable because the facts which were hidden are now visible to the developer.

// Java program without Optional Class

public class OptionalDemo{

public static void main(String[] args) {

String[] words = new String[10];

String word = words[5].toLowerCase();

System.out.print(word);

}

}

Output :

Exception in thread "main" java.lang.NullPointerException

To avoid abnormal termination, we use the Optional class. In the following example, we are using Optional. So, our program can execute without crashing.

The above program using Optional Class

// Java program with Optional Class

import java.util.Optional;

public class OptionalDemo{

public static void main(String[] args) {

String[] words = new String[10];

Optional<String> checkNull =

Optional.ofNullable(words[5]);

if (checkNull.isPresent()) {

String word = words[5].toLowerCase();

System.out.print(word);

} else

System.out.println("word is null");

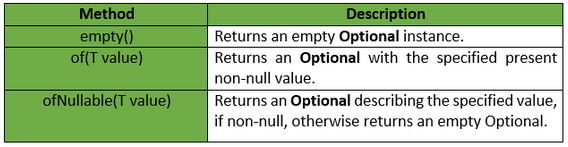
}

}

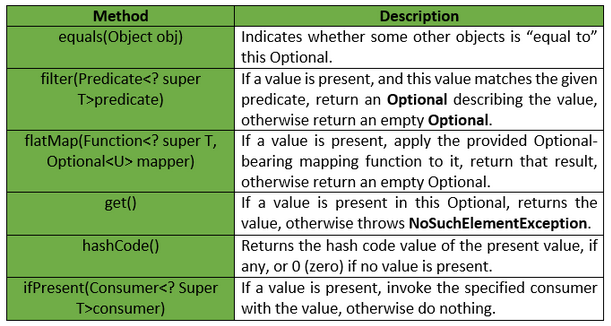
Optional is a container object which may or may not contain a non-null value. You must import ***java.util package*** to use this class. If a value is present, **isPresent()** will return true and **get()** will return the value. Additional methods that depend on the presence or absence of a contained value are provided, such as **orElse()** which returns a default value if value not present and **ifPresent()** which executes a block of code if the value is present. This is a ***value-based*** class, i.e their instances are :

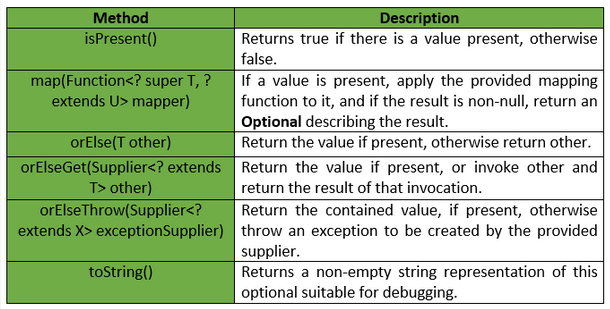
* Final and immutable (though may contain references to mutable objects).
* Considered equal solely based on equals(), not based on reference equality(==).
* Do not have accessible constructors.

Following table shows the list of Static Methods provided by Optional Class :



Following table shows the list of Instance Methods provided by Optional Class :





1. **What are the uses of Optional?**

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.

1. **Java 8 Optional using simple example**

In this post we will understand the Java 8 Optional using simple examples.  
Java 8 introduced a new **container class java.util.Optional<T>**. It wraps a single value, if that value is available. If the value is not available an empty optional should be returned. Thus it represents null value with absent value. This class has various utility methods like isPresent() which helps users to avoid making use of null value checks. So instead of returning the value directly, a wrapper object is returned thus users can avoid the null pointer exception.  
Consider the Item class-

package com.javainuse.model;

public class Item {

private String name;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

Consider the class ItemProcessor which returns an instance of Item class-

package com.javainuse;

import com.javainuse.model.Item;

public class ItemProcessor {

public Item getItem() {

// Some logic here like webservice calls or db calls

return item;

}

}

When we call the getItem(), the item instance is returned. We first have to check for null else this may cause NullPointerException.

ItemProcessor ip= new ItemProcessor();

Item item=ip.getItem();

if(item!=null)

System.out.println(item.getName());

Now lets consider the use of Optional. Instead of item instance we return optional.

package com.javainuse;

import java.util.Optional;

import com.javainuse.model.Item;

public class OptionalItemProcessor {

public **Optional<Item>** getItem() {

// Some logic here like webservice calls or db calls..

// if item is successfully processed in our logic..then create optional

// using Optional.of(Object)

return Optional.of(item);

// else if no item is returned from our logic then

return Optional.empty();

}

}

Here first check if the optional element is present and only then process it. No null checks are required.

OptionalItemProcessor ip1= new OptionalItemProcessor();

**Optional<Item>** item=ip1.getItem();

if(**!item.isPresent()**)

{

System.out.println("not present");

}else

System.out.println(item.get().getName());

1. **Which method in Optional provides the fallback mechanism in case of null value?**

In case, an Optional has null value, we can use orElseGet() method as fallback mechanism. If we implement orElseGet() method, it will be invoked when the value of Optional is null.

1. **How can we get current time by using Date/Time API of Java 8?**

// get current time without date

        LocalTime ltObj = LocalTime.now();

        System.out.println("Current time without date: "+ltObj);

1. **How can we analyze the dependencies in Java classes and packages?**
2. **What are the new JVM arguments introduced by Java 8?**

## ****2. Explicit Heap Memory – Xms and Xmx Options****

One of the most common performance-related practices is to initialize the heap memory as per the application requirements.

That's why we should specify minimal and maximal heap size. Below parameters can be used for achieving it:

|  |  |
| --- | --- |
| 1  2 | -Xms<heap size>[unit]  -Xmx<heap size>[unit] |

Here, **unit** denotes the unit in which the memory (indicated by **heap size**) is to be initialized. Units can be marked as **‘g'** for GB, ***‘m'*** for MB and ***‘k'*** for KB.

For example, if we want to assign minimum 2 GB and maximum 5 GB to JVM, we need to write:

|  |  |
| --- | --- |
| 1 | -Xms2G -Xmx5G |

Starting with Java 8, the size of [*Metaspace*](https://blogs.oracle.com/poonam/about-g1-garbage-collector,-permanent-generation-and-metaspace) is not defined. Once it reaches the global limit, JVM automatically increases it, However, to overcome any unnecessary instability, we can set Metaspace size with:

|  |  |
| --- | --- |
| 1 | -XX:MaxMetaspaceSize=<metaspace size>[unit] |

Here, ***metaspace size*** denotes the amount of memory we want to assign to Metaspace.

As per [Oracle guidelines](https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/sizing.html), after total available memory, the second most influential factor is the proportion of the heap reserved for the Young Generation. By default, the minimum size of the YG is 1310 MB, and maximum size is unlimited.

We can assign them explicitly:

|  |  |
| --- | --- |
| 1  2 | -XX:NewSize=<young size>[unit]  -XX:MaxNewSize=<young size>[unit] |

## ****3. Garbage Collection****

For better stability of the application, choosing of right [Garbage Collection](http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html) algorithm is critical.

JVM has four types of GC implementations:

* Serial Garbage Collector
* Parallel Garbage Collector
* CMS Garbage Collector
* G1 Garbage Collector

These implementations can be declared with the below parameters:

|  |  |
| --- | --- |
| 1  2  3  4 | -XX:+UseSerialGC  -XX:+UseParallelGC  -XX:+USeParNewGC  -XX:+UseG1GC |

More details on Garbage Collection implementations can be found [here](https://www.baeldung.com/jvm-garbage-collectors).

## ****4. GC Logging****

To strictly monitor the application health, we should always check the JVM's Garbage Collection performance. The easiest way to do this is to log the GC activity in human readable format.

Using the following parameters, we can log the GC activity:

|  |  |
| --- | --- |
| 1  2  3  4 | -XX:+UseGCLogFileRotation  -XX:NumberOfGCLogFiles=< number of log files >  -XX:GCLogFileSize=< file size >[ unit ]  -Xloggc:/path/to/gc.log |

**UseGCLogFileRotation** specifies the log file rolling policy, much like log4j, s4lj, etc. **NumberOfGCLogFiles** denotes the max number of log files that can be written for a single application life cycle. **GCLogFileSize** specifies the max size of the file. Finally, ***loggc*** denotes its location.

Point to note here is that, there are two more JVM parameters available (***-XX:+PrintGCTimeStamps*** and ***-XX:+PrintGCDateStamps***) which can be used to print date-wise timestamp in the GC log.

For example, if we want to assign a maximum of 100 GC log files, each having a maximum size of 50 MB and want to store them in ‘/home/user/log/' location, we can use below syntax:

|  |  |
| --- | --- |
| 1  2  3  4 | -XX:+UseGCLogFileRotation  -XX:NumberOfGCLogFiles=10  -XX:GCLogFileSize=50M  -Xloggc:/home/user/log/gc.log |

However, the problem is that one additional daemon thread is always used for monitoring system time in the background. This behavior may create some performance bottleneck; that's why it's always better not to play with this parameter in production.

## ****5. Handling Out of Memory****

It's very common for a large application to face [out of memory error](https://docs.oracle.com/javase/7/docs/api/java/lang/OutOfMemoryError.html) which, in turn, results in the application crash. It's a very critical scenario and very hard to replicate to troubleshoot the issue.

That's why JVM comes with some parameters which dump heap memory into a physical file which can be used later for finding out leaks:

|  |  |
| --- | --- |
| 1  2  3  4 | -XX:+HeapDumpOnOutOfMemoryError  -XX:HeapDumpPath=./java\_pid<pid>.hprof  -XX:OnOutOfMemoryError="< cmd args >;< cmd args >"  -XX:+UseGCOverheadLimit |

A couple of points to note here:

* ***HeapDumpOnOutOfMemoryError*** instructs the JVM to dump heap into physical file in case of OutOfMemoryError
* **HeapDumpPath** denotes the path where the file is to be written; any filename can be given; however, if JVM finds a **<pid>** tag in the name, the process id of the current process causing the out of memory error will be appended to the file name with .hprof format
* ***OnOutOfMemoryError*** is used to issue emergency commands to be executed in case of out of memory error; proper command should be used in the space of cmd args. For example, if we want to restart the server as soon as out of memory occur, we can set the parameter:

|  |  |
| --- | --- |
| 1 | -XX:OnOutOfMemoryError="shutdown -r" |

* **UseGCOverheadLimit** is a policy that limits the proportion of the VM's time that is spent in GC before an OutOfMemory error is thrown

## ****6. 32/64 Bit****

In the OS environment where both 32 and 64-bit packages are installed, the JVM automatically chooses 32-bit environmental packages.

If we want to set the environment to 64 bit manually, we can do so using below parameter:

|  |  |
| --- | --- |
| 1 | -d<OS bit> |

OS bit can be either **32** or **64**. More information about this can be found [here](http://www.oracle.com/technetwork/java/hotspotfaq-138619.html#64bit_layering).

## ****7. Misc****

* ***-server***: enables “Server Hotspot VM”; this parameter is used by default in 64 bit JVM
* ***-XX:+UseStringDeduplication***: Java 8u20 has introduced this JVM parameter for reducing the unnecessary use of memory by creating too many instances of the same String; this optimizes the heap memory by reducing duplicate String values to a single global char[] array
* ***-XX:+UseLWPSynchronization***: sets LWP (Light Weight Process) – based synchronization policy instead of thread-based synchronization
* **-XX:LargePageSizeInBytes:** sets the large page size used for the Java heap; it takes the argument in GB/MB/KB; with larger page sizes we can make better use of virtual memory hardware resources; however, this may cause larger space sizes for the PermGen, which in turn can force to reduce the size of Java heap space
* ***-XX:MaxHeapFreeRatio***: sets the maximum percentage of heap free after GC to avoid shrinking.
* ***-XX:MinHeapFreeRatio***: sets the minimum percentage of heap free after GC to avoid expansion; to monitor the heap usage you can use [VisualVM](https://visualvm.github.io/) shipped with JDK.
* ***-XX:SurvivorRatio***: Ratio of eden/survivor space size – for example, -XX:SurvivorRatio=6 sets the ratio between each survivor space and eden space to be 1:6,
* ***-XX:+UseLargePages***: use large page memory if it is supported by the system; please note that OpenJDK 7 tends to crash if using this JVM parameter
* **-XX:+UseStringCache:** enables caching of commonly allocated strings available in the String pool
* ***-XX:+UseCompressedStrings***: use a byte[] type for String objects which can be represented in pure ASCII format
* **-XX:+OptimizeStringConcat:** it optimizes String concatenation operations where possible

1. **What are the popular annotations introduced in Java 8?**

## Java Type Annotations

Java 8 has included two new features repeating and type annotations in its prior annotations topic. In early Java versions, you can apply annotations only to declarations. After releasing of Java SE 8 , annotations can be applied to any type use. It means that annotations can be used anywhere you use a type. For example, if you want to avoid NullPointerException in your code, you can declare a string variable like this:

1. @NonNull String str;

Following are the examples of type annotations:

1. @NonNull List<String>

#### Note - Java created type annotations to support improved analysis of Java programs. It supports way of ensuring stronger type checking.

## Java Repeating Annotations

In Java 8 release, Java allows you to repeating annotations in your source code. It is helpful when you want to reuse annotation for the same class. You can repeat an annotation anywhere that you would use a standard annotation.

For compatibility reasons, repeating annotations are stored in a container annotation that is automatically generated by the Java compiler. In order for the compiler to do this, two declarations are required in your code.

1. Declare a repeatable annotation type
2. Declare the containing annotation type

### 1) Declare a repeatable annotation type

Declaring of repeatable annotation type must be marked with the @Repeatable meta-annotation. In the following example, we have defined a custom @Game repeatable annotation type.

1. @Repeatable(Games.class)
2. @interfaceGame{
3. String name();
4. String day();
5. }

The value of the @Repeatable meta-annotation, in parentheses, is the type of the container annotation that the Java compiler generates to store repeating annotations. In the following example, the containing annotation type is Games. So, repeating @Game annotations is stored in an @Games annotation.

### 2) Declare the containing annotation type

Containing annotation type must have a value element with an array type. The component type of the array type must be the repeatable annotation type. In the following example, we are declaring Games containing annotation type:

1. @interfaceGames{
2. Game[] value();
3. }

#### Note - Compiler will throw a compile-time error, if you apply the same annotation to a declaration without first declaring it as repeatable.

1. **What is a StringJoiner in Java 8?**

StringJoiner is a new class added in Java 8 under java.util package.

Simply put, **it can be used for joining Strings making use of a delimiter, prefix, and suffix.**

We can add Strings using the add() method:

@Test

public void whenAddingElements\_thenJoinedElements() {

    StringJoiner joiner = new StringJoiner(",", PREFIX, SUFFIX);

    joiner.add("Red")

      .add("Green")

      .add("Blue");

    assertEquals(joiner.toString(), "[Red,Green,Blue]");

}

If we want to join all elements of a list, we'll have to loop through the list. Unfortunately, there's no easy way to do it using StringJoiner:

@Test

public void whenAddingListElements\_thenJoinedListElements() {

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

    rgbList.add("Red");

    rgbList.add("Green");

    rgbList.add("Blue");

    StringJoiner rgbJoiner = new StringJoiner(

      ",", PREFIX, SUFFIX);

    for (String color : rgbList) {

        rgbJoiner.add(color);

    }

    assertEquals(rgbJoiner.toString(), "[Red,Green,Blue]");

}

1. **What are the main differences between an interface with default method and an abstract class in Java 8?**

After introducing Default Method, it seems that interfaces and abstract classes are same. However, they are still different concept in Java 8.

Abstract class can define constructor. They are more structured and can have a state associated with them. While in contrast, default method can be implemented only in the terms of invoking other interface methods, with no reference to a particular implementation's state. Hence, both use for different purposes and choosing between two really depends on the scenario context.

Default methods in Java interface enables interface evolution. Given an existing interface, if you wish to add a method to it without breaking the binary compatibility with older versions of the interface, you have two options at hands: add a default or a static method. Indeed, any abstract method added to the interface would have to be implemented by the classes or interfaces implementing this interface.

A static method is unique to a class. A default method is unique to an instance of the class. If you add a default method to an existing interface, classes and interfaces which implement this interface do not need to implement it.

1. <https://www.javainuse.com/java/java8_intvw>
2. <https://www.wisdomjobs.com/e-university/java-8-interview-questions.html>

Investment banks

<https://www.codementor.io/savingfunda/20-java-interview-questions-and-answers-from-investment-banks-98ghl6frw>

# [Java Interview Questions Answers for 2 to 7 Year Experienced Programmers](https://javarevisited.blogspot.com/2015/10/133-java-interview-questions-answers-from-last-5-years.html)

<https://javarevisited.blogspot.com/2015/10/133-java-interview-questions-answers-from-last-5-years.html>

# [Top 50 Java Thread Interview Questions Answers for Experienced](https://javarevisited.blogspot.com/2014/07/top-50-java-multithreading-interview-questions-answers.html)

<https://javarevisited.blogspot.com/2014/07/top-50-java-multithreading-interview-questions-answers.html>

<https://dzone.com/articles/technical-job-interview>