**Java SE 8 New Features?**

* 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

**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](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.

**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](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.

**Explain Differences between Collection API and Stream API?**

|  |  |  |
| --- | --- | --- |
| S.No. | 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. |

**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 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() |

**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:

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.

### Java SE 8 Interview Questions

1. What is Internal Iteration in Java SE 8?
2. Differences between External Iteration and Internal Iteration?
3. What are the major drawbacks of External Iteration?
4. What are the major advantages of Internal Iteration over External Iteration?
5. What is the major drawback of Internal Iteration over External Iteration?
6. What is the major advantage of External Iteration over Internal Iteration?
7. When do we need to use Internal Iteration? When do we need to use External Iteration?
8. Differences between Intermediate Operations and Terminal Operations of [Java 8](https://www.journaldev.com/2389/java-8-features-with-examples)’s Stream API?
9. Is it possible to provide method implementations in Java Interfaces? If possible, how do we provide them?
10. What is a Default Method? Why do we need Default methods in Java 8 Interfaces?
11. What is a Static Method? Why do we need Static methods in Java 8 Interfaces?
12. Differences between Functional Programming and Object-Oriented Programming?
13. 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?
14. 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?
15. What are the Differences between Java’s OLD Java Date API and Java 8’s Date and Time API?
16. What is Multiple Inheritance? How Java 8 supports Multiple Inheritance?
17. What is Diamond Inheritance Problem? How Java 8 Solves this problem?

### Java SE 8 Interview Questions and Answers

### 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](https://www.journaldev.com/1162/java-multithreading-concurrency-interview-questions-answers) 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](https://www.journaldev.com/129/how-to-create-immutable-class-in-java) 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)](https://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?

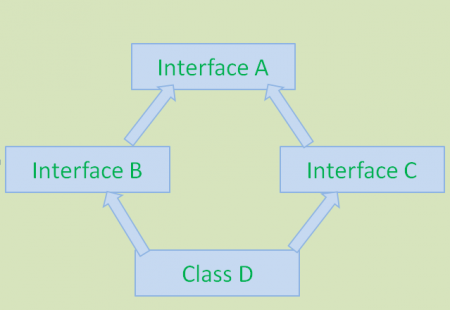
Multiple Inheritance means a class can inherit or extend characteristics and features from more than one parent class.

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:

**Copy**

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:

**Copy**

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.