**Java 8 Interview Questions**

1. Why do we need change to Java again?
2. Java SE 8 New Features?
3. Advantages of Java SE 8 New Features?
4. What is Lambda Expression?
5. What are the three parts of a Lambda Expression? What is the type of Lambda Expression?
6. What is a Functional Interface? What is SAM Interface?
7. Is is possible to define our own Functional Interface? What is @FunctionalInterface? What are the rules to define a Functional Interface?
8. Is @FunctionalInterface annotation mandatory to define a Functional Interface? What is the use of @FunctionalInterface annotation? Why do we need Functional Interfaces in Java?
9. When do we go for Java 8 Stream API? Why do we need to use Java 8 Stream API in our projects?
10. Explain Differences between Collection API and Stream API?
11. What is Spliterator in Java SE 8?Differences between Iterator and Spliterator in Java SE 8?
12. What is Optional in Java 8? What is the use of Optional?Advantages of Java 8 Optional?
13. 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?

### 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 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.

Example of SAM:

**Java Lambdas and the Single Method Interface**

Functional programming is very often used to implement event listeners. Event listeners in Java are often defined as Java interfaces with a single method. Here is a fictive single method interface example:

public interface StateChangeListener {

public void onStateChange(State oldState, State newState);

}

In Java 7 you would have to implement this interface in order to listen for state changes. Imagine you have a class called StateOwner which can register state event listeners. Here is an example:

public class StateOwner {

public void addStateListener(StateChangeListener listener) { ... }

}

In Java 7 you could add an event listener using an anonymous interface implementation, like this:

StateOwner stateOwner = new StateOwner();

stateOwner.addStateListener(new StateChangeListener() {

public void onStateChange(State oldState, State newState) {

// do something with the old and new state.

}

});

First a StateOwner instance is created. Then an anonymous implementation of the StateChangeListenerinterface is added as listener on the StateOwner instance.

In Java 8 you can add an event listener using a Java lambda expression, like this:

StateOwner stateOwner = new StateOwner();

stateOwner.addStateListener(

**(oldState, newState) -> System.out.println("State changed")**

);

The lambda expressions is this part:

(oldState, newState) -> System.out.println("State changed")

### 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.

### 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 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.

### 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 Internal Iteration concept to iterate Elements. | It (Stream API) uses External Iteration to iterate Elements. |
| 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 (Spliterator) 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" };

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.

That’s it about Java 8 Interview Questions.

# =============Java Lambda Expressions===================

* [Java Lambdas and the Single Method Interface](http://tutorials.jenkov.com/java/lambda-expressions.html#single-method-interface)
  + [Matching Lambdas to Interfaces](http://tutorials.jenkov.com/java/lambda-expressions.html#matching-lambdas-to-interfaces)
* [Lambda Type Inference](http://tutorials.jenkov.com/java/lambda-expressions.html#type-inference)
* [Lambda Parameters](http://tutorials.jenkov.com/java/lambda-expressions.html#lambda-parameters)
  + [Zero Parameters](http://tutorials.jenkov.com/java/lambda-expressions.html#zero-parameter)
  + [One Parameter](http://tutorials.jenkov.com/java/lambda-expressions.html#one-parameter)
  + [Multiple Parameters](http://tutorials.jenkov.com/java/lambda-expressions.html#multiple-parameters)
  + [Parameter Types](http://tutorials.jenkov.com/java/lambda-expressions.html#parameter-types)
* [Lambda Function Body](http://tutorials.jenkov.com/java/lambda-expressions.html#lambda-body)
* [Returning a Value From a Lambda Expression](http://tutorials.jenkov.com/java/lambda-expressions.html#returning values-from-lambda-expression)
* [Lambdas as Objects](http://tutorials.jenkov.com/java/lambda-expressions.html#lambdas-as-objects)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2015-03-10 |

Java lambda expressions are new in Java 8. Java lambda expressions are Java's first step into functional programming. A Java lambda expression is thus a function which can be created without belonging to any class. A lambda expression can be passed around as if it was an object and executed on demand.

## Java Lambdas and the Single Method Interface

Functional programming is very often used to implement event listeners. Event listeners in Java are often defined as Java interfaces with a single method. Here is a fictive single method interface example:

public interface StateChangeListener {

public void onStateChange(State oldState, State newState);

}

This Java interface defines a single method which is called whenever the state changes (in whatever is being observed).

In Java 7 you would have to implement this interface in order to listen for state changes. Imagine you have a class called StateOwner which can register state event listeners. Here is an example:

public class StateOwner {

public void addStateListener(StateChangeListener listener) { ... }

}

In Java 7 you could add an event listener using an anonymous interface implementation, like this:

StateOwner stateOwner = new StateOwner();

stateOwner.addStateListener(new StateChangeListener() {

public void onStateChange(State oldState, State newState) {

// do something with the old and new state.

}

});

First a StateOwner instance is created. Then an anonymous implementation of the StateChangeListenerinterface is added as listener on the StateOwner instance.

In Java 8 you can add an event listener using a Java lambda expression, like this:

StateOwner stateOwner = new StateOwner();

stateOwner.addStateListener(

**(oldState, newState) -> System.out.println("State changed")**

);

The lambda expressions is this part:

(oldState, newState) -> System.out.println("State changed")

The lambda expression is matched against the parameter type of the addStateListener() method's parameter. If the lambda expression matches the parameter type (in this case the StateChangeListenerinterface) , then the lambda expression is turned into a function that implements the same interface as that parameter.

Java lambda expressions can only be used where the type they are matched against is a single method interface. In the example above, a lambda expression is used as parameter where the parameter type was the StateChangeListener interface. This interface only has a single method. Thus, the lambda expression is matched successfully against that interface.

### Matching Lambdas to Interfaces

A single method interface is also sometimes referred to as a *functional interface*. Matching a Java lambda expression against a functional interface is divided into these steps:

* Does the interface have only one method?
* Does the parameters of the lambda expression match the parameters of the single method?
* Does the return type of the lambda expression match the return type of the single method?

If the answer is yes to these three questions, then the given lambda expression is matched successfully against the interface.

## Lambda Type Inference

Before Java 8 you would have to specify what interface to implement, when making anonymous interface implementations. Here is the anonymous interface implementation example from the beginning of this text:

stateOwner.addStateListener(new StateChangeListener() {

public void onStateChange(State oldState, State newState) {

// do something with the old and new state.

}

});

With lambda expressions the type can often be *inferred* from the surrounding code. For instance, the interface type of the parameter can be inferred from the method declaration of the addStateListener()method (the single method on the StateChangeListener interface). This is called *type inference*. The compiler infers the type of a parameter by looking elsewhere for the type - in this case the method definition. Here is the example from the beginning of this text, showing that the StateChangeListenerinterface is not mentioned in the lambda expression:

stateOwner.addStateListener(

(oldState, newState) -> System.out.println("State changed")

);

In the lambda expression the parameter types can often be inferred too. In the example above, the compiler can infer their type from the onStateChange() method declaration. Thus, the type of the parameters oldState and newState are inferred from the method declaration of the onStateChange()method.

## Lambda Parameters

Since Java lambda expressions are effectively just methods, lambda expressions can take parameters just like methods. The (oldState, newState) part of the lambda expression shown earlier specifies the parameters the lambda expression takes. These parameters have to match the parameters of the method on the single method interface. In this case, these parameters have to match the parameters of the onStateChange() method of the StateChangeListener interface:

public void onStateChange(State oldState, State newState);

As a minimum the number of parameters in the lambda expression and the method must match.

Second, if you have specified any parameter types in the lambda expression, these types must match too. I haven't shown you how to put types on lambda expression parameters yet (it is shown later in this text), but in many cases you don't need them.

### Zero Parameters

If the method you are matching your lambda expression against takes no parameters, then you can write your lambda expression like this:

() -> System.out.println("Zero parameter lambda");

Notice how the parentheses have no content in between. That is to signal that the lambda takes no parameters.

### One Parameter

If the method you are matching your Java lambda expression against takes one parameter, you can write the lambda expression like this:

(param) -> System.out.println("One parameter: " + param);

Notice the parameter is listed inside the parentheses.

When a lambda expression takes a single parameter, you can also omit the parentheses, like this:

param -> System.out.println("One parameter: " + param);

### Multiple Parameters

If the method you match your Java lambda expression against takes multiple parameters, the parameters need to be listed inside parentheses. Here is how that looks in Java code:

(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);

Only when the method takes a single parameter can the parentheses be omitted.

### Parameter Types

Specifying parameter types for a lambda expression may sometimes be necessary if the compiler cannot infer the parameter types from the functional interface method the lambda is matching. Don't worry, the compiler will tell you when that is the case. Here is a Java lambda parameter type example:

(Car car) -> System.out.println("The car is: " + car.getName());

As you can see, the type (Car) of the car parameter is written in front of the parameter name itself, just like you would when declaring a parameter in a method elsewhere, or when making an anonymous implementation of an interface.

## Lambda Function Body

The body of a lambda expression, and thus the body of the function / method it represents, is specified to the right of the -> in the lambda declaration: Here is an example:

(oldState, newState) -> **System.out.println("State changed")**

If your lambda expression needs to consist of multiple lines, you can enclose the lambda function body inside the { } bracket which Java also requires when declaring methods elsewhere. Here is an example:

(oldState, newState) -> {

System.out.println("Old state: " + oldState);

System.out.println("New state: " + newState);

}

## Returning a Value From a Lambda Expression

You can return values from Java lambda expressions, just like you can from a method. You just add a return statement to the lambda function body, like this:

(param) -> {

System.out.println("param: " + param);

return "return value";

}

In case all your lambda expression is doing is to calculate a return value and return it, you can specify the return value in a shorter way. Instead of this:

(a1, a2) -> { return a1 > a2; }

You can write:

(a1, a2) -> a1 > a2;

The compiler then figures out that the expression a1 > a2 is the return value of the lambda expression (hence the name lambda *expressions* - as expressions return a value of some kind).

## Lambdas as Objects

A Java lambda expression is essentially an object. You can assign a lambda expression to a variable and pass it around, like you do with any other object. Here is an example:

public interface MyComparator {

public boolean compare(int a1, int a2);

}

MyComparator myComparator = (a1, a2) -> return a1 > a2;

boolean result = myComparator.compare(2, 5);

The first code block shows the interface which the lambda expression implements. The second code block shows the definition of the lambda expression, how the lambda expression is assigned to variable, and finally how the lambda expression is invoked by invoking the interface method it implements.

### Java Stream

Before we look into Java Stream API Examples, let’s see why it was required. Suppose we want to iterate over a list of integers and find out sum of all the integers greater than 10.

Prior to Java 8, the approach to do it would be:

private static int sumIterator(List<Integer> list) {

Iterator<Integer> it = list.iterator();

int sum = 0;

while (it.hasNext()) {

int num = it.next();

if (num > 10) {

sum += num;

}

}

return sum;

}

There are three major problems with the above approach:

1. We just want to know the sum of integers but we would also have to provide how the iteration will take place, this is also called **external iteration** because client program is handling the algorithm to iterate over the list.
2. The program is sequential in nature, there is no way we can do this in parallel easily.
3. There is a lot of code to do even a simple task.

To overcome all the above shortcomings, Java 8 Stream API was introduced. We can use Java Stream API to implement **internal iteration**, that is better because java framework is in control of the iteration.

**Internal iteration** provides several features such as sequential and parallel execution, filtering based on the given criteria, mapping etc.

Most of the Java 8 Stream API method arguments are functional interfaces, so lambda expressions work very well with them. Let’s see how can we write above logic in a single line statement using Java Streams.

private static int sumStream(List<Integer> list) {

return list.stream().filter(i -> i > 10).mapToInt(i -> i).sum();

}

Notice that above program utilizes java framework iteration strategy, filtering and mapping methods and would increase efficiency.

First of all we will look into the core concepts of Java 8 Stream API and then we will go through some examples for understanding most commonly used methods.

# ======Java.util Interface Spliterator in Java8===

**Prerequisite :**[Iterators in java](http://www.geeksforgeeks.org/iterators-in-java/)

Spliterators, like other Iterators, are for traversing the elements of a source. A source can be a [Collection](http://www.geeksforgeeks.org/collections-in-java-2/), an [IO channel](https://docs.oracle.com/javase/7/docs/api/java/nio/channels/Channels.html) or a [generator function](https://en.wikipedia.org/wiki/Generator_(computer_programming)).

* It is included in JDK 8 for support of efficient parallel traversal(parallel programming) in addition to sequential traversal.
* However, you can use Spliterator even if you won’t be using parallel execution. One reason you might want to do so is because it combines the hasNext and next operations into one method.

For collections, Spliterator object can be created by calling spliterator() method present in Collection interface.

**// Here "c" is any Collection object. splitr is of**

**// type Spliterator interface and refers to "c"**

**Spliterator splitr = c.spliterator();**

**Spliterator interface defines 8 methods:**

1. **int characteristics()** : Returns a set of characteristics of this Spliterator and its elements. The result is from ORDERED(0x00000010), DISTINCT(0x00000001), SORTED(0x00000004), SIZED(0x00000040), NONNULL(0x00000100), IMMUTABLE(0x00000400), CONCURRENT(0x00001000), SUBSIZED(0x00004000).
2. **Syntax :**
3. int characteristics()
4. **Parameters :**
5. NA
6. **Returns :**
7. Returns the characteristics of the invoking spliterator,
8. encoded into an integer.
9. **long estimateSize( )** : It returns an estimate the number of elements left to iterate or returns Long.MAX\_VALUE if infinite, unknown, or too expensive to compute.
10. **Syntax :**
11. long estimateSize( )
12. **Parameters :**
13. NA
14. **Returns :**
15. Estimates the number of elements left to iterate and
16. returns the result. Returns Long.MAX\_VALUE if the
17. count cannot be obtained for any reason.
18. **default long getExactSizeIfKnown( )** : Convenience method that returns estimateSize() if this Spliterator is SIZED, else -1.
19. **Syntax :**
20. default long getExactSizeIfKnown( )
21. **Parameters :**
22. NA
23. **Returns :**
24. If the invoking spliterator is SIZED, returns the number of
25. elements left to iterate. Returns –1 otherwise.
26. **default Comparator<? super T> getComparator( )**: If this Spliterator’s source is SORTED by a Comparator, returns that Comparator. If the source is SORTED in natural order, returns null. Otherwise, if the source is not SORTED, throws IllegalStateException.
27. **Syntax :**
28. default Comparator<? super T> getComparator( )
29. **Parameters :**
30. NA
31. **Returns :**
32. Returns the comparator used by the invoking spliterator
33. or null if natural ordering is used.
34. **Throws:**
35. IllegalStateException - If the sequence is unordered,
36. IllegalStateException is thrown.
37. **default boolean hasCharacteristics(int val)**: Returns true if this Spliterator’s characteristics() contain all of the given characteristics.
38. **Syntax :**
39. default boolean hasCharacteristics(int val)
40. **Parameters :**
41. characteristics - the characteristics to check for
42. **Returns :**
43. Returns true if the invoking spliterator has the
44. characteristics passed in val. Returns false otherwise.
45. **boolean tryAdvance(Consumer<? super T> action)** : If a remaining element exists, performs the given action on it, returning true; else returns false. If this Spliterator is ORDERED the action is performed on the next element in encounter order. Exceptions thrown by the action are relayed to the caller.
46. **Syntax :**
47. boolean tryAdvance(Consumer<? super T> action)
48. **Parameters :**
49. action - The action
50. **Returns :**
51. Returns true if there is a next element. Returns false if no
52. elements remain.
53. **Throws :**
54. NullPointerException - if the specified action is null
55. **default void forEachRemaining(Consumer<? super T>action)** : Performs the given action for each remaining element, sequentially in the current thread, until all elements have been processed or the action throws an exception. If this Spliterator is ORDERED, actions are performed in encounter order. Exceptions thrown by the action are relayed to the caller.
56. **Syntax :**
57. default void forEachRemaining(Consumer<? super T>action)
58. **Parameters :**
59. action - The action
60. **Returns :**
61. NA
62. **Throws :**
63. NullPointerException - if the specified action is null
64. **Spliterator<T> trySplit( )** : If possible, splits the invoking spliterator, returning a reference to a new spliterator for the partition. Otherwise, returns null. Thus, if successful, the original spliterator iterates over one portion of the sequence and the returned spliterator iterates over the other portion.
65. **Syntax :**
66. Spliterator<T> trySplit( )
67. **Parameters :**
68. NA
69. **Returns :**
70. a Spliterator covering some portion of the elements,
71. or null if this spliterator cannot be split

The below example demonstrate methods of Spliterator.

|  |
| --- |
| // Java program to demonstrate  // methods of Spliterator    import java.util.ArrayList;  import java.util.Spliterator;  import java.util.stream.Stream;    public class SpliteratorDemo  {      public static void main(String[] args)      {          // Create an array list for doubles.          ArrayList<Integer> al = new ArrayList<>();            // Add values to the array list.          al.add(1);          al.add(2);          al.add(-3);          al.add(-4);          al.add(5);            // Obtain a Stream to the array list.          Stream<Integer> str = al.stream();            // getting Spliterator object on al          Spliterator<Integer> splitr1 = str.spliterator();            // estimateSize method          System.out.println("estimate size : " + splitr1.estimateSize());            // getExactSizeIfKnown method          System.out.println("exact size : " + splitr1.getExactSizeIfKnown());            // hasCharacteristics and characteristics method          System.out.println(splitr1.hasCharacteristics(splitr1.characteristics()));            System.out.println("Content of arraylist :");          // forEachRemaining method          splitr1.forEachRemaining((n) -> System.out.println(n));            // Obtaining another  Stream to the array list.          Stream<Integer> str1 = al.stream();          splitr1 = str1.spliterator();            // trySplit() method          Spliterator<Integer> splitr2 = splitr1.trySplit();            // If splitr1 could be split, use splitr2 first.          if(splitr2 != null) {          System.out.println("Output from splitr2: ");          splitr2.forEachRemaining((n) -> System.out.println(n));          }            // Now, use the splitr          System.out.println("\nOutput from splitr1: ");          splitr1.forEachRemaining((n) -> System.out.println(n));        }  } |

Run on IDE

Output:

estimate size : 5

exact size : 5

true

Content of arraylist :

1

2

-3

-4

5

Output from splitr2:

1

2

Output from splitr1:

-3

-4

5

**Java program for tryadvance method**

Have a look at tryAdvance() method.It performs an action on the next element and then advances the iterator. It is shown here:

boolean tryAdvance(Consumer<? super T> action)

Here, action specifies the action that is executed on the next element in the iteration and Consumer is a [functional interface](http://www.geeksforgeeks.org/functional-interfaces-java/)that applies an action to an object. It is a generic functional interface declared in java.util.function. It has only one abstract method, accept( ), which is  
shown here:

void accept(T objRef)

here T is type of object reference.

For implementing our action, we must implement accept method.To implement accept method, here we use [lambda expression](http://www.geeksforgeeks.org/lambda-expressions-java-8/) .This will be more clear from below example.

**How to use Spliterator with Collections**: Using Spliterator for basic iteration tasks is quite easy, simply call tryAdvance( ) until it returns false.

|  |
| --- |
| // Java program to demonstrate simple Spliterator  // using tryAdvance method    import java.util.ArrayList;  import java.util.Spliterator;    public class SpliteratorDemo  {      public static void main(String[] args)      {          // Create an array list for doubles.          ArrayList<Integer> al1 = new ArrayList<>();            // Add values to the array list.          al1.add(1);          al1.add(2);          al1.add(-3);          al1.add(-4);          al1.add(5);              // Use tryAdvance() to display(action) contents of arraylist.            System.out.print("Contents of arraylist:\n");            // getting Spliterator object on al1          Spliterator<Integer> splitr = al1.spliterator();            // Use tryAdvance() to display(action) contents of arraylist.          // Notice how lambda expression is used to implement accept method          // of Consumer interface            while(splitr.tryAdvance((n) -> System.out.println(n)));            // Use tryAdvance() for getting absolute values(action) of contents of arraylist.              // Create new list that contains absolute values.          ArrayList<Integer> al2 = new ArrayList<>();                splitr = al1.spliterator();            // Here our action is to get absolute values          // Notice how lambda expression is used to implement accept method          // of Consumer interface          while(splitr.tryAdvance((n) -> al2.add(Math.abs(n))));            System.out.print("Absolute values of contents of arraylist:\n");            // getting Spliterator object on al2          splitr = al2.spliterator();              while(splitr.tryAdvance((n) -> System.out.println(n)));        }  } |

Run on IDE

Output:

Contents of arraylist:

1

2

-3

-4

5

Absolute values of contents of arraylist:

1

2

3

4

5

Notice how tryAdvance( ) consolidates the purposes of hasNext( ) and next( ) provided by [Iterator](http://www.geeksforgeeks.org/iterators-in-java/) into a single method in above example. This improves the efficiency of the iteration process.

In some cases, you might want to perform some action on each element collectively, rather than one at a time. To handle this type of situation, Spliterator provides the forEachRemaining( ) method, it is generally used in cases involving [streams](http://www.geeksforgeeks.org/stream-in-java/). This method applies action to each unprocessed element and then returns.

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [contribute.geeksforgeeks.org](http://contribute.geeksforgeeks.org/) or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

# [GATE CS Corner](http://quiz.geeksforgeeks.org/gate-corner-2/)[Company Wise Coding Practice](http://practice.geeksforgeeks.org/company-tags)

[Java](http://www.geeksforgeeks.org/category/programming-language/java/)

[Java - util package](http://www.geeksforgeeks.org/tag/java-util-package/)

==================LAMDA EXP EXAMPLE===========

TriFunction<**Sum**, String, String, Integer> anon =

new TriFunction<**Sum**, String, String, Integer>() {

@Override

**public** Integer apply(**Sum** s, String arg1, String arg2) {

**return** s.doSum(arg1, arg1);

}

};

System.out.println(anon.apply(new **Sum**(), "1", "4"));

Or by using a lambda expression:

TriFunction<**Sum**, String, String, Integer> lambda =

(**Sum** s, String arg1, String arg2) -> s.doSum(arg1, arg1);

System.out.println(lambda.apply(new **Sum**(), "1", "4"));

### 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.

### 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?

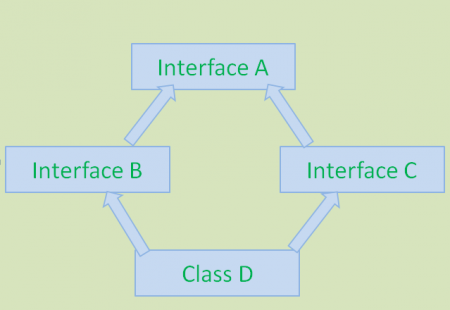
Multiple Inheritance means a class can inherit or extend characteristics and features from more than one parent class.

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:

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.