1. **How to change the Java version in maven?**

**Configuring Buildpath:**

If you see JRE System Library[J2SE-1.5] then change the version by below process.

Do right-click on the project and go to Build -> Configure build path, under Libraries tab click on JRE System Library[J2SE-1.5], click on Edit button and select the appropriate jdk 1.8 from the next window. Click on Finish then Ok.

Change also the Compiler compliance level as 1.8 from Java -> Compiler.

# How to Check The Apache Tomcat Log Files?

The main Apache Tomcat configuration file is at /opt/bitnami/apache-tomcat/conf/server.xml.

Once Apache Tomcat starts, it **will create several log files in the  /opt/bitnami/apache-tomcat/logs directory. The main log file is the catalina.out file where you can find error messages.** On some platforms, you may need root account privileges to view these files.

**MAVEN Questions:**

1. **Explain what is Maven? How does it work?**

Maven is a project management tool. It provides the developer a complete build lifecycle framework. On executing Maven commands, it will look for POM file in Maven; it will run the command on the resources described in the POM.

1. **List out what are the build phases in Maven?**

Build phases in Maven are

• Validate  
• Compile  
• Test  
• Package  
• Install  
• Deploy

1. **Explain what is POM?**

In Maven, POM (Project Object Model) is the fundamental unit of work. It is an XML file which holds the information about the project and configuration details used to build a project by Maven.

1. **Explain what is Maven artifact?**

Usually an artifact is a JAR file which gets arrayed to a Maven repository. One or more artifacts a maven build produces such as compiled JAR and a sources JAR.

Each artifact includes a group ID, an artifact ID and a version string.

1. **Explain how you can exclude dependency?**

By using the exclusion element, dependency can be excluded

1. **Explain how you can produce execution debug output or error messages?**

To produce execution debug output you could call Maven with X parameter or e parameter

1. **Explain how to run test classes in Maven?**

To run test classes in Maven, you need surefire plugin, check and configure your settings in setting.xml and pom.xml for a property named “test.”

## What is Maven Plugins? What are those?

Plugin in maven is the one of the vital feature that is basically used to reuse the common build logic across different projects. Plugins are the ones through which all the tasks like compiling code, testing them with the junits, creating jar/war/ear files and documentation of the projects are carried out. Most of the work in maven is done using plugins, since the dependencies (jar files) are added only to the classpath while executing tasks.

plugins in maven :

* Creating jar/war/ear files.
* Code compilation
* Unit testing of the code.
* Project documentation

GIT:

<https://www.atlassian.com/git/tutorials/learn-git-with-bitbucket-cloud> 🡪 Learn Git with Bitbucket cloud.

<https://git-scm.com/docs/git>

<https://git-scm.com/docs>

<https://www.atlassian.com/git/tutorials> 🡪 Learn Git fully here. All topics are covered here.

<https://www.atlassian.com/git/tutorials/learn-about-code-review-in-bitbucket-cloud> --> Learn about Code review in Bitbucket cloud.

<https://www.atlassian.com/git/tutorials/learn-branching-with-bitbucket-cloud> 🡪 Learn branching with Bitbucket cloud.

<https://www.atlassian.com/git/tutorials/learn-undoing-changes-with-bitbucket> 🡪 Learn uncoding changes with Bitbucket.

<https://dzone.com/articles/top-20-git-commands-with-examples> --> with examples

Apache Spark : try to learn

JavaScript topics to become a Full Stack Developer:

JS Fundamentals,

JS Data, Decision Making, and Arrays,

JS Front End Development : Functions and Objects,

JS Front End Development : Working with JS Errors, Events, Objects, and New Features,

JS Front End Development : Front End Development Basics,

JS Front End Development : Frameworks

**Spring Boot Project to develop:**

<https://stormpath.com/blog/build-spring-boot-spring-security-app> 🡪 try to implement this application, goto this link end of the page application screenshots are there implement those to get more confidence as a full stack developer.

**Java 8:** (From Howtodoinjava.com)

**Java 8 features:**

[Lambda Expression](https://howtodoinjava.com/java-8-tutorial/#lambda)

[Functional Interface](https://howtodoinjava.com/java-8-tutorial/#functional-interface)

[Default Methods](https://howtodoinjava.com/java-8-tutorial/#default-method)

[Streams](https://howtodoinjava.com/java-8-tutorial/#stream)

[Date/Time API Changes](https://howtodoinjava.com/java-8-tutorial/#date-time)

**1. Lambda Expression:**

A Lambda expression (or function) is just an anonymous function, i.e., a function with no name and without being bounded to an identifier.

Syntax:

either

(parameters) -> expression

or

(parameters) -> { statements; }

or

() -> expression

Example:

(x, y) -> x + y //This function takes two parameters and return their sum.

#### Rules for writing lambda expressions

1. A lambda expression can have zero, one or more parameters.

Ex: () 🡪System.out.println();

(x) 🡪System.out.println(x);

(x, y) or (int x, int y) 🡪 System.out.println(x+” “+y);

1. The type of the parameters can be explicitly declared or it can be inferred from the context.

Ex:

1. Multiple parameters are enclosed in mandatory parentheses and separated by commas. Empty parentheses are used to represent an empty set of parameters.

Ex: (x, y, z)🡪 System.out.println(x+” “+y+” “+z);

() 🡪System.out.println();

1. When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses.

Ex:

a -> return a\*a.

1. The body of the lambda expressions can contain zero, one or more statements.

Ex: ((x, y) or (int x, int y) 🡪 { System.out.println(x+y);

System.out.println(x\*y);

} );

1. If body of lambda expression has single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression. When there is more than one statement in body than these must be enclosed in curly brackets.

Ex: (x, y) or (int x, int y) 🡪 System.out.println(x+y); // curly brackets are not mandatory and returns int value.

((x, y) or (int x, int y) 🡪 { System.out.println(x+y);

System.out.println(x\*y); // enclosed in curly brackets. Bcz,

} ); multiple stmt’s are written in body.

## 2. Functional Interface

Functional interfaces are also called Single Abstract Method interfaces (SAM Interfaces). As name suggest, they **permit exactly one abstract method** inside them.

Java 8 introduces an annotation i.e. **@FunctionalInterface** too, which can be used for compiler level errors when the interface you have annotated violates the contracts of exactly one abstract method.

A typical functional interface example:

|  |  |  |  |
| --- | --- | --- | --- |
| @FunctionalInterface  public interface MyFirstFunctionalInterface {      public void firstWork();  }  Let’s try to add another abstract method:   |  | | --- | | @FunctionalInterface  public interface MyFirstFunctionalInterface  {      public void firstWork();      public void doSomeMoreWork();   //error  } |   Above will result into compiler error as given below:   |  | | --- | | Error like, | | Unexpected @FunctionalInterface annotation  @FunctionalInterface ^ MyFirstFunctionalInterface is not a functional interface  multiple non-overriding abstract methods found in interface MyFirstFunctionalInterface | |

## Do’s and Don’t’s in functional interfaces

Below is list of things which are allowed and which are not in a functional interface.

* As discussed above, ***only one abstract method is allowed*** in any functional interface. Second abstract method is not permitted in a functional interface. If we remove **@FunctionInterface** annotation then we are allowed to add another abstract method, but it will make the interface non-functional interface.
* A functional interface is ***valid even if the @FunctionalInterface annotation would be omitted***. It is only for informing the compiler to enforce single [abstract method](https://howtodoinjava.com/object-oriented/exploring-interfaces-and-abstract-classes-in-java/) inside interface.
* Conceptually, a functional interface has exactly one abstract method. Since [**default methods**](https://howtodoinjava.com/java8/default-methods-in-java-8/) have an implementation, they are not abstract. Since default methods are not abstract you’re **free to add default methods to your functional interface as many as you like**.

Below is valid functional interface:

|  |
| --- |
| @FunctionalInterface  public interface MyFirstFunctionalInterface {      public void firstWork();      default void doSomeMoreWork1(){      //Method body }      default void doSomeMoreWork2() {      //Method body      }  } |

* If an interface declares an **abstract method overriding one of the public methods of java.lang.Object, that also does not count toward the interface’s abstract method count** since any implementation of the interface will have an implementation from java.lang.Object or elsewhere.
* e.g. [**Comparator**](https://howtodoinjava.com/search-sort/when-to-use-comparable-and-comparator-interfaces-in-java/) is a functional interface even though it declared two abstract methods. Why? Because one of these abstract methods “equals()” which has signature equal to public method in Object class.

e.g. Below interface is a valid functional interface.

|  |
| --- |
| @FunctionalInterface  public interface MyFirstFunctionalInterface {      public void firstWork();      @Override      public String toString();           //Overridden from Object class      @Override      public boolean equals(Object obj);   //Overridden from Object class  } |

# Method Reference:

In [Java 8](https://howtodoinjava.com/java-8-tutorial/), we can refer a method from class or object using **class::methodName** type syntax. Let’s learn about different types of available method references in java 8.

Java 8 allows four types of method references.

## 1. Method reference to static method – Class::staticMethodName

It is Used to refer static methods from a class. An example to use Math.max() which is static method.

|  |
| --- |
| List<Integer> integers = Arrays.asList(1,12,433,5);  Optional<Integer> max = integers.stream().reduce( Math::max );  max.ifPresent(value -> System.out.println(value)); |

Output:

433

## 2. Method reference to instance method from instance – ClassInstance::instanceMethodName

In above example, we use System.out.println(value) to print the max value found. We can use System.out::println to print the value.

|  |
| --- |
| List<Integer> integers = Arrays.asList(1,12,433,5);  Optional<Integer> max = integers.stream().reduce( Math::max );  max.ifPresent( System.out::println ); |

Output:

433

## 3. Method reference to instance method from class type – Class::instanceMethodName

In this example, s1.compareTo(s2) is referred as String::compareTo.

|  |
| --- |
| List<String> strings = Arrays.asList("how", "to", "do", "in", "java", "dot", "com");  List<String> sorted = strings.stream().sorted((s1, s2) -> s1.compareTo(s2))           .collect(Collectors.toList());  System.out.println(sorted);    With Mathod Reference,  List<String> sortedAlt = strings.stream()  .sorted(String::compareTo) //Method reference happen here           .collect(Collectors.toList());  System.out.println(sortedAlt); |

Output:

[com, do, dot, how, in, java, to]

[com, do, dot, how, in, java, to]

## 4. Reference to constructor – Class::new

The first method can be updated to create a list of integers from 1 to 100. Using [lambda expression](https://howtodoinjava.com/java8/complete-lambda-expressions-tutorial-in-java/) is rather easy. To create a new instance of ArrayList, we have use ArrayList::new.

|  |
| --- |
| List<Integer> integers = IntStream.range(1, 100)                   .boxed()                   .collect(Collectors.toCollection( ArrayList::new ));  Optional<Integer> max = integers.stream().reduce(Math::max);  max.ifPresent(System.out::println); |

Output:

99

# Default Methods:

Default methods in java 8 are simply default. If you do not override them, they are the methods which will be invoked by caller classes. They are defined in interfaces.

Let’s understand with an example:

|  |
| --- |
| public interface Moveable {      default void move(){          System.out.println("I am moving");      }  } |

Moveable interface defines a method move(); and provided a default implementation as well. If any class implements this interface then it need not to implement it’s own version of move() method. It can directly call instance.move();

|  |
| --- |
| public class Animal implements Moveable{      public static void main(String[] args){          Animal tiger = new Animal();          tiger.move();      }  }    Output: I am moving |

And if class willingly wants to customize the behavior then it can provide it’s own custom implementation and override the method. Now it’s own custom method will be called.

|  |  |
| --- | --- |
| public class Animal implements Moveable{        public void move(){          System.out.println("I am running");      }        public static void main(String[] args){          Animal tiger = new Animal();          tiger.move();      }  }    Output: I am running Why default methods were needed in java 8? **Sol: To enable the functionality of lambda expression in java.** Lambda expression are essentially of type of functional interface. Streams Streams provides a mechanism for processing a set of data in various ways that can include filtering, transformation, or any other way that may be useful to an application.  Ex:  items is collection of String values and you want to remove the entries that begin with some prefix text.   |  | | --- | | List<String> items;  String prefix;  List<String> filteredList = items.stream().filter(e -> (!e.startsWith(prefix))).collect(Collectors.toList()); |   Here items.stream() indicates that we wish to have the data in the items collection processed using the Streams API.  Internal Iterator vs External Iterator:  **External Iterators-** This Iterator is also known as active iterator or explicit iterator. For this type of iterator the control over iteration of elements is with the programmer. Which means that the programmer define when and how the next element of iteration is called.  Ex: List<String> items = new ArrayList<>();  items.add("test1");  items.add("test2");  items.add("test3");  items.add("test4");  items.add("test5");    **//Traditional java for-each iterator which is an External Iterator.**  for (String item : items) {  System.out.println(item);  }  **Internal Iterators-** This Iterator is also known as passive iterator, implicit iterator or callback iterator. For this type of iterator the control over the iteration of elements lies with the iterator itself. The programmer only tells the iterator "What operation is to be performed on the elements of the collection". Thus the programmer only declares what is to be done and does not manage and control how the iteration of individual elements take place.  List<String> items = new ArrayList<>();  items.add("test1");  items.add("test2");  items.add("test3");  items.add("test4");  items.add("test5");  //iterate over list items  //Java 8 forEach iterator which is an Internal Iterator.  **items.forEach(item -> System.out.println(item));** Date/Time API ChangesDates The new classes intended to replace Date class are LocalDate, LocalTime and LocalDateTime.   1. The LocalDate class represents a date. There is no representation of a time or time-zone. 2. The LocalTime class represents a time. There is no representation of a date or time-zone. 3. The LocalDateTime class represents a date-time. There is no representation of a time-zone.   If you want to use the date functionality with zone information, then Lambda provide you extra 3 classes i.e. OffsetDate, OffsetTime and OffsetDateTime. Timezone offset can be represented in “+05:30” or “Europe/Paris” formats. This is done via using another class i.e. ZoneId.  Ex:  LocalDate localDate = LocalDate.now();  LocalTime localTime = LocalTime.of(12, 20);  LocalDateTime localDateTime = LocalDateTime.now();  OffsetDateTime offsetDateTime = OffsetDateTime.now();  ZonedDateTime zonedDateTime = ZonedDateTime.now(ZoneId.of("Europe/Paris")); Timestamp and Duration For representing the specific timestamp at any moment, the class needs to be used is Instant. The Instant class represents an instant in time to an accuracy of nanoseconds. Operations on an Instant include comparison to another Instant and adding or subtracting a duration.  Ex:  Instant instant = Instant.now();  Instant instant1 = instant.plus(Duration.ofMillis(5000));  Instant instant2 = instant.minus(Duration.ofMillis(5000));  Instant instant3 = instant.minusSeconds(10);  Duration class is a whole new concept brought first time in java language. It represents the time difference between two time stamps.  Ex:  Duration duration = Duration.ofMillis(5000);  duration = Duration.ofSeconds(60);  duration = Duration.ofMinutes(10);  Duration deals with small unit of time such as milliseconds, seconds, minutes and hour. They are more suitable for interacting with application code. To interact with human, you need to get **bigger durations** which are presented with Period class.  Ex:  Period period = Period.ofDays(6);  period = Period.ofMonths(6);  period = Period.between(LocalDate.now(), LocalDate.now().plusDays(60)); |

Refer it:

<https://www.logicbig.com/tutorials/core-java-tutorial/java-util-stream/sequential-vs-parallel.html>

**Questions:**

**Lambda Expression (LE):**

**What is the Lambda Expression? Why we use?**

Sol:

**What are the three main parts of a Lambda Expression in java?**

**Sol:** **Parameter List :** A LE can have zero or more parameters. Parameter List is optional to Lambda.

**Lambda Arrow operator :** “->”is lambda arrow operator. It separates the list of parameters and body of lambda.

**Lambda Expression body :** The piece of code that we want to execute is written in LE body.

**Ex:** Arrays.asList(“a”,”b”,”c”).forEach(e ->System.out.println(e));

Here, Parameter list =e

Arrow= ->

Body= System.out.println(e).

**What is the data type of a LE?**

**Sol:** A LE fulfills the purpose of passing code as data. The data type of a LE is a Functional Interface.

**What are the advantages of LE?**

Sol:

**What is the type of a LE in java 8?**

Sol: The type of a Lambda Expression depends on the context it is being used. A Lambda is like a method reference. It does not have a type of its own. Generally, a Lambda is an instance of a functional interface.

**What is the target type of a LE?**

Sol: The target type of a lambda expression represents a type to which the expression can be converted. The target type for a lambda expression is a functional interface. The lambda expression must have same parameter type as the parameter in the function of the interface. It must also return a type compatible with the return type of function.

**Functional Interface (FI):**

**What is FI?**

Sol:

**What is Single Abstract Method (SAM) Interface?**

**Sol:** A FI is also known as SAM interface, since it has exactly one abstract method.

**How can we define FI?**

**Sol:** We can create an interface with exactly one abstract method (or) to mark an interface with annotation @FunctionalInterface. This annotation allows only one abstract method otherwise it will give the compilation error.

**Why do we need FI?**

**Sol:** FI are mainly used in LE, Method reference and constructor references. In functional programming, code can be treated as data. For this purpose LE are introduced. They can be used to pass a block of code to another method or object. FI serves as a data type for LE. Since a FI contains only one abstract method, the implementation of that method becomes the code that gets passed as an argument to another method.

**Is it mandatory to use @FunctionalInterface annotation to define FI?**

**Sol:** No

**Differences between Collection and Stream API?**

**Sol:**

|  |  |
| --- | --- |
| **Collection API** | **Stream API** |
| Collection API is use since Java 1.2 | Stream API is recent addition to java 8. |
| It is used for storing data in different kinds of data structures like set’s, map’s, Linkedlist etc. | It is used for computation of data on a large set of objects. |
| With this we can store a finite no.of elements in a data structure. | With this we can handle streams of data that can contain infinite no.of elements. |
| Most of the collection API’s support iteration and consumption of elements multiple times. | With this we can consume or iterate elements only once. |
| It constructs objects in an eager manner | It creates objects in lazy manner. |

**What are the differences between Intermediate and Terminal Operations?**

**Sol:**

|  |  |
| --- | --- |
| **Intermediate Operations** | **Terminal Operations** |
| These are not evaluated until we chain it with a terminal operation of stream. | These can be independently evaluated |
| The output of Intermediate operations is another stream. | The output of Terminal Operations is not a stream. |
| These are evaluated in lazy manner | These are evaluated in eager manner. |
| We can chain multiple Intermediate Operations in a stream. | Terminal operations can’t be chained multiple times. |
| There can be multiple Intermediate Operations in a stream operation. | There can be only one Terminal Operation in stream processing statement. |

**What is the differences b/w Sequential and parallel Streams?**

|  |  |
| --- | --- |
| **Sequential Streams** | **Parallel Streams** |
| These are performed one by one in an order. | These are processed simultaneously. |
| These are performed as imperative operations. | In Stream, these are performed as a pipeline of aggregate operations |

To work with parallel and sequential stream, we need to instantiate stream as parallel and sequential and after that both will be same in coding.

**Ex:** List<String> list = Arrays.asList("A", "B", "C");

list.stream(); //Sequential Stream

list.parallelStream(); //Parallel stream

**What is a spliterator in java 8?**

**Sol:** A spliterator is an iterator to traverse and partition elements of a source in java. A spliterator may traverse elements individually or sequentially in bulk.

**What is type inference in java 8?**

**Sol:** A java compiler can see each method’s invocation and it declaration to determine what are type arguments required for invocation. By type inference, java can determine the types of the arguments as well as the type of the result being returned.

Ex: static <T> T getNumber(T a1,T a2) {

Return a2; }

BigDecimal bigD = getNumber(“d”, BigDecimal.TEN);

**How does Internal iteration works in java 8?**

**Sol:** In case of internal iteration, the client hands over an operation to iterator and the iterator applies the operation to all the elements in aggregate. Internal iteration is easier to implement, since the iterator does not have to store the state of the collection.

**What are the differences between internal and external Iterator?**

**Sol:**

|  |  |
| --- | --- |
| **Internal Iterator** | **External Iterator** |
| It controls the iteration itself. | Collection controls the iteration. |
| It support declarative programming style that goes well with functional programming. | It follows imperative style OOPS programming. |
| It does not have to iterate elements only sequentially. | Always iterates sequentially. |
| It can iterate elements in individually as well as in bulk. | It iterates elements one by one. |

**What are the Applications In which we should use internal iteration?**

**Sol:** High performance, parallel processing, fast iteration, bulk operations support.

**Can we provide implementation of a method in java interface?**

**Sol:** Before java 8, it was not allowed to provide the implementation of a method in an interface. Java 8 has introduced the flexibility of providing implementation of a method in an interface.

**Default method:** we can give default implementation of a method.

**Static Method:** we can create a static method in an interface and provide implementation.

**Default Method:**

**What is Default Method In interface?**

Sol: In java 8, we can provide implementation of a method in an interface and mark this method with default keyword. This implementation becomes default behavior for a class implementing the interface.

**Ex:** Public interface TimeService {

Public string getTime(int hour, int minute, int second);

Static ZoneId getZone(String zoneString){

Return ZoneId.of(zoneString); }

default String getZonedTime(String zoneString) {

return getTime()+getZone(zoneString);

} }

**Why do we need Default Method in a java8 interface?**

Sol:

**What is the purpose of static method in an interface in java8?**

**Sol:** A static method in an interface is utility or helper method. This is not an object levelinstance method. These are the usecases where we use the static method in an interface,

**Single class:** There is no need to create a separateUtils class for storing utility or helper methods. We can keep these methods in same interface.

**Encapsulation:** With Static methods, complete behavior of a class is encapsulated in same class. There is no need to maintain multiple classes.

**Extension:** It is easier to extend a class/API. If we extend a collection ArrayList, we get all the methods. We need not extend collections class also.

**Differences b/w an interface with default method and an abstract class in java 8?**

Sol:

|  |  |
| --- | --- |
| Interface with default method | Abstract class |
| It can’t have instance variables. | It can have instance variables. |
| It can’t have a constructor | It can have a constructor. |
| It can’t have concrete methods other than default method. | It is allowed to define concrete methods with implementation. |
| An interface with exactly one default method can be used for lambda expression. | An abstract class can’t be used for lambda expression. |

**Date and Tme:**

**What are the core ideas behind the Date/Time API of java8?**

**Sol: Immutable-value classes:** The new API avoids thread-safety and concurrency issues by ensuring that all the core classes are immutable and represent well -defined values.

**Domain-driven design:** The new API is modeled on precise domain with clases that represent different use cases for date and time.

**Seperation of chronologies:** The new API allows people to work with different calendar systems.it supports the needs of users in different areas of the world like japan or Thailand that don’t follow ISO-8601.

**Advantages of new Date and time API in java8 over old date API?**

**Sol:**

**Concurrency:** Existing Date Time classes (such as java.util.Date and SimpleDateFormatter) are not thread-safe. This does not work well in concurrent applications. In new Date and Time API, developer does not have to deal with concurrency issues while writing date-handling code.

**Better Design:** Date/Time classes prior to java 8 have poor API design. For Example, years in java.util.Date start at 1900, months start at 0, and date start at 1. It is not intuitive.Java 8 Date Time API handles it very well.

**No need for 3rd party Libraries:** With the popularity of third-party Date/Time libraries like Joda Time, java has to make its native Date/Time API comparable. Now we can use the Java API instead of using 3rd party libraries.

**Differences between Legacy Date/Time API in java and Date/Time API of java8?**

Sol:

|  |  |
| --- | --- |
| Legacy Date/Time API (old API) | Date/Time API of Java 8 (new API) |
| It is not thread safe. | It’s thread safe. |
| It has many mutable objects. | It is based on immutable objects. |
| Performance is not good. | It gives the better performance. |
| It is less readable and maintainable. | It is very well designed and is more readable. |
| It has month values from 0 to 11. | It has month values from 1 to 12. |

**How can we get duration b/w two dates or time in java8?**

Sol: In java 8, we have a new class **Duration** that provides the utility of computing duration b/w two dates. We can call the static method **Duration.between(date1,date2)** to get the time period in hours, mins, days etc. b/w date1 and date2.

**How can we get current time by using Date/Time API of java8?**

Sol: We can use clock class to get the current time. Instead of using old method System.currentTimeMillis(), we can create a Clock object and call millis() method to get the current time in milliseconds.

We can also call instant() method on Clock object to get the current time in a readable format.

**Ex:** public class ClockTest{

Public static void main(String[] args){

Clock c=Clock.systemUTC();

System.out.println(c.millis()); } }

**Differences b/w Predicate, Supplier and Consumer in java8?**

**Sol:**

**Predicate:** It is an anonymous function that **accepts one argument** and **returns a result**. It has methods like and(), isEqual(), negate().

**Supplier:** It is an anonymous function that **accepts no argument** and **returns a result**. It has method like get().

**Consumer:** It is an anonymous function that **accepts one argument** and **returns no result**. It has methods like accept(), andThen().

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

Sol: No, we have to always mark a default method in interface with default keyword. If we create a method with implementation in an interface, but do not mark it as default, then we will get compile time error.

**Can we create a class that implements two interfaces with default methods of same name and**

**signature?**

Sol: No, it is not allowed to create a class that implements interfaces with same name default methods. It will give us compile time error for duplicate default methods.

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

Sol: No, a static method of interface has to be invoked by using the name of the interface.

**Optional:**

**What is Optional I Java8? Why we use it?**

Sol:

**Uses:**

We can use Optional to avoid NullPointer Exception in an application.

Optional performs Null check at compile time, sow edo not get run time exception for a null value.

Optional can also be used to handle default case for data when a value is null.

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

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

**Is it possible to define a static method in an interface?**

Sol: Yes, from java 8 an interface can also has a static method.

Ex: public interface StaticTest{

Public static void printName(){

System.out.println(“Hi”); }; }

Public class HelloTest implements Static Test{

Public static void main(String[] args){

StaticTest.printname(); } }

**What is a StringJoiner in java8?**

Sol: It is a new class in java 8 that can be used to create a string. It can construct a sequence of characters separated by a delimeter and optionally starting with a supplied prefix and ending with a supplied suffix. We can use this sequence to get a string.

**Syntax:**

public StringJoiner(CharSequence delimiter,

CharSequence prefix, CharSequence suffix)

**Ex:** public class Example {

public static void main(String[] args) {

/\* Passing comma(,) as delimiter and opening bracket

\* "(" as prefix and closing bracket ")" as suffix

\*/

StringJoiner mystring = new StringJoiner(",", "(", ")");

// Joining multiple strings by using add() method

mystring.add("Negan");

mystring.add("Rick");

mystring.add("Maggie");

mystring.add("Daryl");

// Displaying the output String

System.out.println(mystring);

}

}

**Output:** (Negan,Rick,Maggie,Daryl)

**What is the Method reference in java8? Why we use it?**

Sol:

**Streams API:**

**What is the Stream API? Why we use it?**

Sol: It helps in using data in a declarative way. We can make use of database functions like Max, Min etc., without running a full iteration. We can create a pipeline of data operations with java stream that can run in a sequence or in parallel. It provides support for group by, order by etc. operations. It supports writing for code in functional programming style. It provides support for parallel processing of data.

[ refer: Java67.com for below questions]

**What does map() function do? why you use it?**

**What does the filter() method do? when you use it?**

**What does flatmap() function do? why you need it?**

**What is difference between flatMap() and map() functions?**

**What is difference between findFirst() and findAny() method?**

**Can you convert an array to Stream? How?**

Sol: how to convert the stream to object array, how to convert a stream of T to an array of T using **lambda expression** and **constructor reference** in Java, and finally, stream to array conversion using Collectors and ArrayList. You can choose whichever way you like, but the constructor reference example i.e. toArray(String[]::new) is the simplest, easiest and the best way to convert a Stream to array in Java.

**Ex:**

public class Java8Demo {

public static void main(String[] args) { // stream to object array in Java

Stream<String> currencies=Stream.of("INR", "USD", "GBP", "EUR", "JPY");

Object[] objectArray = currencies.toArray();

System.out.println("Stream to object array in Java:");

System.out.println(Arrays.toString(objectArray)); // via - Stream.toArray() and lambda expression

Integer[] primes = {2, 3, 5, 7, 11};

List listOfInts = new ArrayList<>(Arrays.asList(primes));

Integer[] array = listOfInts.stream() .toArray(size -> new Integer[size]);

System.out.println("Stream to Integer array using lambda expression in Java:");

System.out.println(Arrays.toString(array)); // via - method reference

array = listOfInts.stream() .toArray(Integer[]::new);

System.out.println("Stream to Integer array using method reference in Java:");

System.out.println(Arrays.toString(array)); // via arraylist

ArrayList list = listOfInts.stream() .collect(Collectors.toCollection(ArrayList::new));

Integer[] iArray = list.toArray(new Integer[list.size()]);

 System.out.println("Stream to Integer array via ArrayList in Java:");

System.out.println(list); } } }   
  
Read more: [**https://javarevisited.blogspot.com/2017/01/3-ways-to-convert-java-8-stream-to-array.html#ixzz5v9cIRmGX**](https://javarevisited.blogspot.com/2017/01/3-ways-to-convert-java-8-stream-to-array.html#ixzz5v9cIRmGX)

**What is the parallel Stream? How can you get a parallel stream from a List?**

**Programs of java 8:**

**How many ways we convert the data from Arrays to list/map in java8?**

**Write to file**

**Join array**

**Sort the data using comparator in java8?**

**class** Book **implements** Comparable<Book> {

**private** String title;

**private** String author;

**private** **int** price;

//provide parameterized constructor, setters, getters and toString method.

and here is the list of Books which we'll sort in this article:

List<Book> listOfBooks = **new** ArrayList<>();

listOfBooks.add(**new** Book("Effective Java", "Joshua Bloch", 32));

listOfBooks.add(**new** Book("Java Puzzlers", "Joshua Bloch", 22));

listOfBooks.add(**new** Book("Java Concurrency in Practice", "Brian Goetz", 42));

listOfBooks.add(**new** Book("Java SE 8 for Really Impatient", "Cay S. Horstmann", 34));

listOfBooks.add(**new** Book("Core Java", "Cay S. Horstmann",32));

## 1. Writing Comparator using Lambda Expression

Comparator and Comparable are also SAM interfaces e.g. they contain just one abstract method like compare() and compareTo(), you can easily implement them using a lambda expression.\

If you want to write a Comparator to sort Books by their author, you can write like in the following example:

Comparator<Book> byAuthor = (b1, b2) -> b1.getAuthor().compareTo(b2.getAuthor());

diff b/w post and get

how to iterate the user defined data using list in java8?

how to convert the user defined data from list to map?

how to sort the user defined data using comparator?

how to fetch the data using post method without using parametersd like query,path parameters?

I did some changes in git, but i don't want these changed files.How can i get the before changing files?

Java 8 from Concretepage.com

[Java 8 Stream Tutorial with Example](https://www.concretepage.com/java/jdk-8/java-8-stream-tutorial-with-example)

**Ways to obtain Streams Instance?**

Collection has also introduced new methods i.e Collection.stream() and Collection.parallelStream() that is used to obtain sequential and parallel streams in our code

1. Using stream() and parallelStream() methods of Collection which is extended by List, Queue , Set etc. For example suppose we have a List, then we can use methods as follow.   
List.stream() and   
List.parallelStream()

2. In case of Map, streams are obtained as follows   
Map.entrySet().stream() and   
Map.entrySet().parallelStream()

3. Using Arrays.stream method. We can pass array of primitive data types or objects to this method such as Arrays.stream(int[] array) or Arrays.stream(Object[] array) etc.   
  
4. Using Stream.of(Object[] array). Here of() is a static method of Stream.

5. Using Stream.iterate(T seed, UnaryOperator<T> f) where **T** is the type of elements and **f** is a function that is applied to the previous element to get new element.   
  
6. Using BufferedReader.lines(). It returns the stream of string.   
  
7. Using java 8 methods of Files such as find(), lines(), walk(). These methods return stream.   
  
8. Using Random class we can obtain streams of random number for primitive data type. The methods ints(), longs() and doubles() of Random class return IntStream, LongStream and DoubleStream respectively.

**Stream tutorial with Examples:**

public class StreamConcat {

public static void main(String[] args) {

List<Integer> list1 = Arrays.asList(1,2,3);

List<Integer> list2 = Arrays.asList(4,5,6);

//**Stream.concat:**

Stream<Integer> resStream = Stream.**concat**(list1.stream(),list2.stream());

resStream.forEach(s->System.out.print(s+" "));

//**Stream.count():**

It returns the number of elements in stream.

Predicate<Integer> p = num -> num % 2 == 0;

List<Integer> list = Arrays.asList(3,4,6);

System.out.println("Count: " + list.stream()

.filter(p)

.count());

//**Stream.distinct():**

It returns stream with distinct elements.

System.out.println("Distinct Count: " + list.stream()

.distinct()

.count());

//**Stream.filter():**

It returns the stream with the elements that matches the given Predicate.

Predicate<Integer> p = num -> num % 2 == 0;

List<Integer> list = Arrays.asList(3,4,6);

list.stream().**filter**(p)

.forEach(e -> System.out.print(e+" "));

//**Stream.flatMap():**

It returns a stream of object after applying mapping function on each element and then flattens the result.

Integer[][] data = {{1,2},{3,4},{5,6}};

Arrays.stream(data).**flatMap**(row -> Arrays.stream(row))

.filter(num -> num%2 == 1)

.forEach(s -> System.out.print(s+" "));

flatMapToInt(): It is used with primitive data type int and returns IntStream.   
flatMapToLong(): It is used with primitive data type long and returns LongStream.   
flatMapToDouble(): It is used with primitive data type double and returns DoubleStream .

//**Stream.forEach() and Stream.forEachOrdered():**

forEach(): It performs an action on each element of stream.   
forEachOrdered (): It also performs an action on each element of the stream but in encountered order of the stream if defined any.

Integer[] data = {1,2,3,4,5,6,7};

Arrays.stream(data).filter(num -> num%2 == 1)

.**forEach**(s -> System.out.print(s+" "));

Arrays.stream(data).filter(num -> num%2 == 1)

.**forEachOrdered**(s -> System.out.print(s+" "));

//**Stream.generate() and Stream.limit():**

generate(): We need to pass Supplier to this method and it will return an infinite sequential unordered stream.   
limit(): We need to pass a max value and it returns the stream up to the max number of elements.

String str = "Hello World!";

Stream<String> stream = Stream.generate(str::toString)

.limit(5);

stream.forEach(s->System.out.println(s));

**output:** it generates the given String as 5 times.

//**Stream.iterate():**We need to pass seed value and UnaryOperator to this method and it will return an infinite sequential unordered stream.

Stream<Integer> stream = Stream.iterate(1, n -> n \* 2)

.limit(5);

stream.forEach(s->System.out.print(s+" ")); // 1 2 4 8 16

**//Stream.map():**It returns a stream after applying given function to each element of the stream.

List<Integer> list = Arrays.asList(1,2,3,4);

list.stream().map(i -> i\*i)

.forEach(s->System.out.print(s+" "));// 1 4 9 16

mapToInt(): It returns IntStream after applying the given function.   
mapToLong(): It returns LongStream after applying the given function.   
mapToDouble(): It returns DoubleStream after applying the given function.

//**Stream.max() and min():**

max(): It finds maximum element for the given Comparator.   
min(): It finds minimum element for the given Comparator.

List<String> list = Arrays.asList("G","B","F","E");

String max = list.stream().max(Comparator.comparing(String::valueOf))

.get();

System.out.println("Max:"+ max); //G

String min = list.stream().min(Comparator.comparing(String::valueOf))

.get();

System.out.println("Min:"+ min); //B

//**Stream.findFirst and Stream.findAny():**

findAny(): It can return any element of the stream.   
findFirst(): It returns first element of the stream and if stream has defined no encounter order, then it may return any element.

List<String> list = Arrays.asList("G","B","F","E");

String any = list.stream()

.findAny()

.get();

System.out.println("FindAny: "+ any); //G

String first = list.stream()

.findFirst()

.get();

System.out.println("FindFirst: "+ first); //G

//**Stream.peek():**

It returns a new stream which consists all the elements of stream after applying the Consumer.

List<String> list = Arrays.asList("A","B","C");

list.stream().peek(s->System.out.println(s+s))

.collect(Collectors.toList());//AA BB CC

**//Stream.reduce():**

It performs reduction on stream elements using a start value and accumulation function.

int[] array = {3,5,10,15};

int sum = Arrays.stream(array)

.reduce(0, (x,y) -> x+y);

System.out.println("Sum:"+ sum); //33

//**Stream.skip():**

It returns a stream skipping the given number of elements.

int[] array = {3,5,10,15};

Arrays.stream(array)

.skip(2)

.forEach(s -> System.out.println(s+ " ")); //10,15

//**Stream.sorted():**

It returns a stream sorted with given Comparator.

Map<Integer, String> map = new HashMap<>();

map.put(1, "BBBB");

map.put(2, "AAAA");

map.put(3, "CCCC");

System.out.println("---Sort by Map Value---");

map.entrySet().stream()

.sorted(Comparator.comparing(Map.Entry::getValue))

.forEach(e -> System.out.println("Key: "+ e.getKey() +

"Value: "+ e.getValue()));

//**Stream.toArray():**

It returns an array containing the elements of stream.

List<String> list = Arrays.asList("A", "B", "C", "D");

Object[] array = list.stream()

.toArray();

System.out.println("Length of array: "+array.length); //4

}

}

[**Stream sorted() Example**](https://www.concretepage.com/java/jdk-8/java-8-stream-sorted-example)**:**

* We can sort the stream and reverse the stream in natural ordering as well as ordering provided by **Comparator**.
* In java 8 Comparator can be instantiated using lambda expression.
* Natural ordering uses the ordering provided by Comparable which must be implemented by the class whose instances are the stream elements.

### Stream sorted() using Natural Ordering, Comparator and Reverse Ordering:

Find the syntax of sorted() method.   
  
1. sorted(): It sorts the elements of stream using natural ordering. The element class must implement Comparableinterface.   
  
2. sorted(Comparator<? super T> comparator): Here we create an instance of Comparator using lambda expression. We can sort the stream elements in ascending and descending order.   
  
The following line of code will sort the list in natural ordering.

list.stream().sorted()

* To reverse the natural ordering Comparator provides reverseOrder() method. We use it as follows.

list.stream().sorted(Comparator.reverseOrder())

* The following line of code is using Comparator to sort the list.

list.stream().sorted(Comparator.comparing(Student::getAge))

* To reverse the ordering, Comparator provides reversed() method. We use this method as follows.

list.stream().sorted(Comparator.comparing(Student::getAge).reversed())

### Stream sorted() with List:

* Here we are sorting a List of objects of Student class.
* First we will sort by natural ordering and then using Comparator. We will reverse both ordering natural ordering as well as ordering provided by Comparator .

**Ex:**

public class SortList {

public static void main(String[] args) {

List<Student> list = new ArrayList<Student>();

list.add(new Student(1, "Mahesh", 12));

list.add(new Student(2, "Suresh", 15));

list.add(new Student(3, "Nilesh", 10));

System.out.println("Natural Sorting by Name");

List<Student> slist = list.stream()

.sorted()

.collect(Collectors.toList());

slist.forEach(e -> sopln("Id:"+e.getId()+

"Name: "+e.getName()+

“Age:"+e.getAge()));

System.out.println("Natural Sorting by Name in reverse order");

slist = list.stream()

.sorted(Comparator.reverseOrder())

.collect(Collectors.toList());

slist.forEach(e -> Sopln("Id:"+ e.getId()+

"Name: "+e.getName()+

"Age:"+e.getAge()));

System.out.println("Sorting using Comparator by Age");

slist = list.stream()

.sorted(Comparator.comparing(Student::getAge))

.collect(Collectors.toList());

slist.forEach(e -> Sopln("Id:"+ e.getId()+

"Name: "+e.getName()+

"Age:"+e.getAge()));

System.out.println("Sorting using Comparator by Age with reverse order");

slist = list.stream() .sorted(Comparator.comparing(Student::getAge).reversed())

.collect(Collectors.toList());

slist.forEach(e -> sopln("Id:"+ e.getId()+

"Name: "+e.getName()+

"Age:"+e.getAge()));

}

}

**Student.java**

public class Student implements Comparable<Student> {

private int id;

private String name;

private int age;

public Student(int id, String name, int age) {

this.id = id;

this.name = name;

this.age = age;

}

//Stters and Getters

@Override

public int compareTo(Student ob) {

return name.compareTo(ob.getName());

}

@Override

public boolean equals(final Object obj) {

if (obj == null) {

return false;

}

final Student std = (Student) obj;

if (this == std) {

return true;

} else {

return (this.name.equals(std.name) && (this.age == std.age));

}

}

@Override

public int hashCode() {

int hashno = 7;

hashno = 13 \* hashno + (name == null ? 0 : name.hashCode());

return hashno;

} }

### Stream sorted() with Set

* This class must override equals() and hashCode() methods to identify unique elements.
* For natural ordering Student class needs to implement Comparable interface.

**EX:**

public class SortSet {

public static void main(String[] args) {

Set<Student> set = new HashSet<Student>();

set.add(new Student(1, "Mahesh", 12));

set.add(new Student(2, "Suresh", 15));

set.add(new Student(3, "Nilesh", 10));

System.out.println("Natural Sorting by Name");

set.stream()

**.sorted()**

.forEach(e -> Sopln("Id:"+e.getId()+

"Name: "+e.getName()+

"Age:"+e.getAge()));

System.out.println("Natural Sorting by Name in reverse order ");

set.stream()

**.sorted(Comparator.reverseOrder())**

.forEach(e -> sopln("Id:"+e.getId()+

"Name: "+e.getName()+

"Age:"+e.getAge()));

System.out.println("Sorting using Comparator by Age");

Set.stream()

**.sorted(Comparator.comparing(Student::getAge))**

.forEach(e -> sopln("Id:"+ e.getId()+

"Name: "+e.getName()+

"Age:"+e.getAge()));

System.out.println("Sorting using Comparator by Age in reverse order");

set.stream() **.sorted(Comparator.comparing(Student::getAge).reversed())**

.forEach(e -> sopln("Id:"+ e.getId()+

"Name: "+e.getName()+

"Age:"+e.getAge()));

}

}

### Stream sorted() with Map

**Ex:**

public class SortMap {

public static void main(String[] args) {

Map<Integer, String> map = new HashMap<>();

map.put(15, "Mahesh");

map.put(10, "Suresh");

map.put(30, "Nilesh");

System.out.println("Sort by Map Value");

map.entrySet().stream()

.sorted(Comparator.comparing(Map.Entry::getValue))

.forEach(e -> sopln("Key: "+ e.getKey() +

"Value: "+ e.getValue()));

System.out.println("Sort by Map Key ");

map.entrySet().stream()

.sorted(Comparator.comparing(Map.Entry::getKey))

.forEach(e -> sopln("Key: "+ e.getKey() +

"Value: "+e.getValue()));

}

}

Here we are sorting a map whose values are custom objects.

public class SortMapOfCustomObject {

public static void main(String[] args) {

Map<Integer, Student> map = new HashMap<>();

map.put(1, new Student(1, "Mahesh", 12));

map.put(2, new Student(2, "Suresh", 15));

map.put(3, new Student(3, "Nilesh", 10));

//Map Sorting by Value i.e student's natural ordering i.e by name

map.entrySet().stream()

.sorted(Comparator.comparing(Map.Entry::getValue))

.forEach(e -> {

Integer key = (Integer)e.getKey();

Student std =(Student)e.getValue();

System.out.println("Key: " + key +

"value: {"

+std.getId()+ ","

+std.getName()+ ","

+std.getAge()+

"}" );

});

}

}

[Java 8 Stream map() Example](https://www.concretepage.com/java/jdk-8/java-8-stream-map-example)

[Java 8 Stream filter() Example](https://www.concretepage.com/java/jdk-8/java-8-stream-filter-example)

[Java 8 Stream collect() Example](https://www.concretepage.com/java/jdk-8/java-8-stream-collect-example)

[Java 8 Stream reduce() Example](https://www.concretepage.com/java/jdk-8/java-8-stream-reduce-example)

[Java 8 Convert List to Map using Collectors.toMap() Example](https://www.concretepage.com/java/jdk-8/java-8-convert-list-to-map-using-collectors-tomap-example)

[Java 8 Convert Map to List using Collectors.toList() Example](https://www.concretepage.com/java/jdk-8/java-8-convert-map-to-list-using-collectors-tolist-example)

[Java 8 Sum: Array, Map and List Collection Example using reduce() and collect() Method](https://www.concretepage.com/java/jdk-8/java-8-sum-array-map-and-list-collection-example-using-reduce-and-collect-method)

[Java 8 Collectors: joining() Example](https://www.concretepage.com/java/jdk-8/java-8-collectors-joining-example)