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://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.

**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>