June 01, 2016

Java 8 Interview Questions and Answers..

What are new features introduced with Java 8 ?

Lambda Expressions, Interface Default and Static Methods, Method Reference, Parameters Name, Optional, Streams, Concurrency.

What is a Lambda Expression? What's its use?

Lambda expressions are introduced in Java 8 and are touted to be the biggest feature of Java 8. Its an anonymous method without any declaration. Lambda Expression are useful to write shorthand code and hence saves the effort of writing lengthy Code. It promotes developer productivity, better readable and reliable code.

A lambda expression is characterized by the following syntax :
parameter -> expression body

Important characteristics of a lambda expression :

- A lambda expression can have zero, one or more parameters.
- The type of the parameters can be explicitly declared or it can be inferred from the context. e.g. (int a) is same as just (a)
- Parameters are enclosed in parentheses and separated by commas. e.g. (a, b) or (int a, int b) or (String a, int b, float c)
- Empty parentheses are used to represent an empty set of parameters. e.g. () -> 42
- When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses. e.g. a -> return a*a
- The body of the lambda expressions can contain zero, one or more statements.
- 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 (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.

Lambda expression eliminates the need of anonymous class and gives a very simple yet powerful functional programming capability to Java.

Lambda expressions are used primarily to define inline implementation of a functional interface, i.e., an interface with a single method only.

Following are some examples of Lambda expressions:

- (int a, int b) -> { return a + b; }
- () -> System.out.println("Hello World");
- (String s) -> { System.out.println(s); }
- () -> 42
- () -> { return 3.1415 };

In the below example, we've used various types of lambda expressions to define the operation method of DoMathOperation interface.

Example:
```
package com.test;
public class LambdaTest {
    interface DoMathOperation {
        int doOperation(int a, int b);
    }
    private int callDoOperation(int a, int b, DoMathOperation doMathOperation) {
        return doMathOperation.doOperation(a, b);
    }

    public static void main(String args[]){
        LambdaTest lamdaTest=new LambdaTest();

        // with type declaration
        DoMathOperation addition = (int a, int b) -> a + b;
        System.out.println("addition: "+ lamdaTest.callDoOperation(10, 5, addition));

        // with out type declaration
        DoMathOperation subtraction = (a, b) -> a - b;
        System.out.println("subtraction: "+ lamdaTest.callDoOperation(10, 5, subtraction));

        // with return statement along with curly braces
        DoMathOperation multiplication = (int a, int b) -> {return a * b;};
```

K Himaanshu Shuklaa
Follow 0
View my complete profile

Blog Archive

- ▼ 2016 (241)
  - ► December (6)
  - ► November (37)
  - ► October (24)
  - ► September (28)
  - ► August (25)
  - ► July (21)
  - ▼ June (22)
    - BARC Ratings (Impressions)- Week 25, 2016
    - Namish Taneja in Comedy Nights Bachao!!
    - I'm a fan of Bruce Lee, Jackie Chan: Remo D'Souza
    - Mohit Abrol steps in instead of Cezzane Khan in Ga...
    - Ratan Rajput's item number in Santoshi Maa..
    - #Awwdorable: First pic of Tusshar Kapoor's son Lak...
    - New book on RD Burman unravels music maestro's esc...
    - #SPOTTED : Divyanka Tripathi selecting the perfect...
    - Zee Anmol Presents 'Afsar Bitiya'..
    - BARC Ratings (Impressions)- Week 24, 2016
    - Will Amba's lie be exposed in &TV's Waaris?
    - SAB TV opens its 'Khidki' on 28th June!
    - Shiv Pandit to be seen in a cameo in SAB TVs Dr Bh...
    - Aishwarya Sakhuja to play a boxer in the first sto...
    - Anita Bhabi to turn into a Khooni Dulhan in Bhabi ...
    - BARC Ratings (Impressions)- Week 23, 2016 : Star P...
    - BARC Ratings (Impressions)- Week 22, 2016 : Star P...
    - Zindagi presents 'Paani Jaisa Pyaar'..
    - IN PICS: Pakistani Actress Sadia Khan
    - Dancing with Madhuri Dixit is like a dream come tr...
    - BARC Ratings (Impressions)- Week 21, 2016
    - Java 8 Interview Questions and Answers..
  - ► May (11)
  - ► April (15)
  - ► March (19)
  - ► February (15)

Featured Post

BARC Ratings (Impressions)- Week 47, 2016

Join us on Facebook

B A R C
BROADCAST AUDIENCE RESEARCH COUNCIL INDIA
ZEE TV
StarPlus
SONY SAB
Life OK
colors
SONY ENTERTAINMENT TELEVISION

Enter keywords here...

Most Popular..

- BARC Ratings (Impressions)- Week 47, 2016
- Sahil Anand and Bani J get nostalgic..
- Pranav Sachdeva gives Shah Rukh Khan's Darr character an EVIL TWIST

► January (18)
► 2015 (395)
► 2014 (699)
► 2013 (522)
► 2012 (514)
► 2011 (262)
► 2010 (136)
► 2009 (59)

```
        System.out.println("multiplication: "+ lamdaTest.callDoOperation(10, 5, multiplication));

        // without return statement and without curly braces
        DoMathOperation division = (int a, int b) -> a / b;
        System.out.println("division: "+ lamdaTest.callDoOperation(10, 5, division));
    }
}
```

Output:

```
addition: 15
subtraction: 5
multiplication: 50
division: 2
```

Let us take a simple example to sort a list of strings. In prior versions of Java, sorting is done:

```
List names = Arrays.asList("thomas", "andreas", "michael", "mattrias");
Collections.sort(names, new Comparator() {
    @Override
    public int compare(String a, String b) {
        return b.compareTo(a);
    }
});
```

In above example, the static utility method Collections.sort accepts a list and a comparator object in order to sort the elements of the given list. We usually create anonymous comparators and pass them to the sort() method, like we did in above example.

In Java 8 we can do the similar thing with a much shorter syntax by using lambda expressions.

```
Collections.sort(names, (String a, String b) -> {
    return b.compareTo(a);
});
```

If you want, you you skip both the braces {} and the return keyword from the above code. Here is even more shorter way:

```
Collections.sort(names, (String a, String b) -> b.compareTo(a));
```

We can also skip the parameter types, as the java compiler is aware of the parameter types. Another way to write the above code:

```
Collections.sort(names, (a, b) -> b.compareTo(a));
```

Difference between Lambda Expression and Anonymous class?
Lambda expressions are very powerful and can replace many usage of anonymous class but not all. An anonymous class can be used to create a subclass of an abstract class, a concrete class and can be used to implement an interface in Java. It can even add state fields.

An instance of anonymous class can also be refereed by using 'this' keyword inside its methods, which means you can call other methods and state can be changed.

Since the most common use of Anonymous class is to provide throwaway, stateless implementation of abstract class and interface with single function, those can be replaced by lambda expressions, but when you have a state field or implementing more than one interface, you cannot use lambdas to replace anonymous class.

For anonymous class 'this' keyword resolves to anonymous class, whereas for lambda expression 'this' keyword resolves to enclosing class where lambda is written.

Another difference between lambda expression and anonymous class is in the way these two are compiled. Java compiler compiles lambda expressions and convert them into private method of the class. It uses invokedynamic instruction that was added in Java 7 to bind this method dynamically.

Difference between Predicate, Supplier and Consumer?

- **Predicate** represents an anonymous function that accepts one argument and produces a result.
- **Supplier** represents an anonymous function that accepts no argument and produces a result.
- **Consumer** represents an anonymous function that accepts an argument and produces no result.

How you can Default Methods for Interfaces?

Java 8 enables us to add non-abstract method implementations to interfaces by utilizing the default keyword. This feature is also known as Extension Methods. For example:

```
package com.defaultmethods;
public interface Formula {
    double doubleIt(int a);

    default double sqrt(int a) {
        return Math.sqrt(a);
    }
}
```

In the above example, apart from an abstract method doubleIt(), one more method sqrt() is also defined. sqrt() is the default method, which can be used out of the box.

When I class implements Formula interface, that class need to only implement the abstract method doubleIt().

```
package com.defaultmethods;
public class TestFormula {
    public static void main(String[] args) {
        //anonymous inner class
        Formula formula=new Formula() {
            @Override
            public double doubleIt(int a) {
                return (double)a;
            }
        };
        System.out.println(formula.doubleIt(10));
        System.out.println(formula.sqrt(16));
    }
}
```

Output is:
```
10.0
4.0
```

What are Functional interfaces in Java8?

All the Java developer's must have used at least one of the following interfaces, while developing Java based projects: java.lang.Runnable, java.awt.event.ActionListener, java.util.Comparator or java.util.concurrent.Callable.

All these interfaces have one thing in common, they all have only one method declared in their interface definition. There are lot more such interfaces in JDK and also lot more created by Java developers. These interfaces are also called Single Abstract Method interfaces (SAM Interfaces).

To use such interfaces, the developer's create Anonymous Inner classes, e.g :

```
public class TestAnonymousInnerClass {
  public static void main(String[] args) {
    new Thread(new Runnable() {
      @Override
      public void run() {
        System.out.println("Inside run method..");
      }
    }).start();
  }
}
```

With Java 8, we can recreate the same concept of SAM interfaces by using @FunctionalInterface annotation. These can be represented using Lambda expressions, Method reference and constructor references.

```
@FunctionalInterface
public interface TestFunctionalInterface {
  public void testIt();
}
```

If required, you can also declare the abstract methods from the java.lang.Object class in Functional Interface, e.g:

```
@FunctionalInterface
public interface TestFunctionalInterface {
  public void testIt();
  public String toString();
  public boolean equals(Object o);
}
```

If you try to add a second abstract method declaration in the FunctionalInterface, the compiler will throw an error. You can declare any number of default methods in the FunctionalInterface.

You can use the lambda expression as against anonymous inner class for implementing functional interfaces:

```
public class FunctionalInterfaceExample {
    public static void doTesting(TestFunctionalInterface tfi) {
        tfi.testIt();
    }

    public static void main(String[] args) {
        doTesting(new TestFunctionalInterface() {
            @Override
            public void testIt() {
                System.out.println("testIt() with anonymous inner class");

            }
        });
        doTesting(() -> System.out.println("testIt() with lambda expression"));
    }
}
```

Output of above program is :
```
testIt() with anonymous inner class
testIt() with lambda expression
```

Another Example:
```
@FunctionalInterface
interface Converter {
    T convert(F from);
}

/*in another class*/
Converter converter = (from) -> Integer.valueOf(from);
Integer converted = converter.convert("619");
System.out.println(converted);
```

What is Method references in Java8?

We can use lambda expressions to create anonymous methods. Sometimes, however, a lambda expression does nothing but call an existing method. In those cases, it's often clearer to refer to the existing method by name. Method references enable you to do this; they are compact, easy-to-read lambda expressions for methods that already have a name.

It allows us to reference constructors or methods without executing them. Method references and Lambda are similar in that they both require a target type that consist of a compatible functional interface (similar to lambda expressions, a method reference when evaluated creates an instance of the functional interface)

A method reference is described using :: (double colon) symbol. A method reference can be used to point the following types of methods :

- Static methods
- Instance methods
- Constructors using new operator (TreeSet::new)

In the first two types, the method reference is equivalent to a lambda expression on which parameters of a method will be supplied. In the 3rd type, the 1st parameter of a method becomes the target of the method and second parameter becomes argument to the method just like s1.equals(s2); .

The above example from 'Functional interfaces' to convert string to Integer can be further simplified by utilizing static method references:

```
Converter converter = Integer::valueOf;
Integer converted = converter.convert("619");
System.out.println(converted);
```

Example:

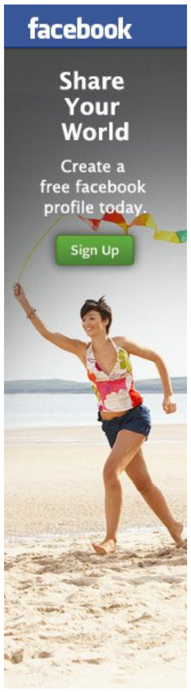
```
package com.test;
import java.util.ArrayList;
import java.util.List;

public class MethodReferenceExample {
    public static void main(String[] args) {
        List employeeNames = new ArrayList();
        employeeNames.add("Martin");
        employeeNames.add("Thomas");
        employeeNames.add("Ulrich");
        employeeNames.add("Stephan");
        employeeNames.add("Andreas");
        employeeNames.forEach(System.out::println);
    }
}
```

Output of the above program:

```
Martin
Thomas
Ulrich
Stephan
Andreas
```

Java 8 has defined a lot of functional interfaces to be used extensively in lambda expressions. Here is the list of functional interfaces defined in java.util.Function package of Java8:
1. BiConsumer : Represents an operation that accepts two input arguments, and returns no result.
2. BiFunction : Represents a function that accepts two arguments and produces a result.
3. BinaryOperator : Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
4. BiPredicate : Represents a predicate (Boolean-valued function) of two arguments.
5. BooleanSupplier : Represents a supplier of Boolean-valued results.
6. Consumer : Represents an operation that accepts a single input argument and returns no result.
7. DoubleBinaryOperator : Represents an operation upon two double-valued operands and producing a double-valued result.
8. DoubleConsumer : Represents an operation that accepts a single double-valued argument and returns no result.
9. DoubleFunction : Represents a function that accepts a double-valued argument and produces a result.
10. DoublePredicate : Represents a predicate (Boolean-valued function) of one double-valued argument.
11. DoubleSupplier : Represents a supplier of double-valued results.
12. DoubleToIntFunction : Represents a function that accepts a double-valued argument and produces an int-valued result.
13. DoubleToLongFunction : Represents a function that accepts a double-valued argument and produces a long-valued result.
14. DoubleUnaryOperator : Represents an operation on a single double-valued operand that produces a double-valued result.
15. Function : Represents a function that accepts one argument and produces a result.
16. IntBinaryOperator : Represents an operation upon two int-valued operands and produces an int-valued result.
17. IntConsumer : Represents an operation that accepts a single int-valued argument and returns no result.
18. IntFunction : Represents a function that accepts an int-valued argument and produces a result.
19. IntPredicate : Represents a predicate (Boolean-valued function) of one int-valued argument.
20. IntSupplier : Represents a supplier of int-valued results.
21. IntToDoubleFunction : Represents a function that accepts an int-valued argument and produces a double-valued result.
22. IntToLongFunction : Represents a function that accepts an int-valued argument and produces a long-valued result.
23. IntUnaryOperator : Represents an operation on a single int-valued operand that produces an int-valued result.
24. LongBinaryOperator : Represents an operation upon two long-valued operands and produces a long-valued result.
25. LongConsumer : Represents an operation that accepts a single long-valued argument and returns no result.
26. LongFunction : Represents a function that accepts a long-valued argument and produces a result.
27. LongPredicate : Represents a predicate (Boolean-valued function) of one long-valued argument.
28. LongSupplier : Represents a supplier of long-valued results.
29. LongToDoubleFunction : Represents a function that accepts a long-valued argument and produces a double-valued result.
30. LongToIntFunction : Represents a function that accepts a long-valued argument and produces an int-valued result.
31. LongUnaryOperator : Represents an operation on a single long-valued operand that produces a long-valued result.
32. ObjDoubleConsumer : Represents an operation that accepts an object-valued and a double-valued argument, and returns no result.
33. ObjIntConsumer : Represents an operation that accepts an object-valued and an int-valued argument, and returns no result.



and returns no result.
34. ObjLongConsumer : Represents an operation that accepts an object-valued and a long-valued argument, and returns no result.
35. Predicate : Represents a predicate (Boolean-valued function) of one argument.
36. Supplier : Represents a supplier of results.
37. ToDoubleBiFunction : Represents a function that accepts two arguments and produces a double-valued result.
38. ToDoubleFunction : Represents a function that produces a double-valued result.
39. ToIntBiFunction : Represents a function that accepts two arguments and produces an int-valued result.
40. ToIntFunction : Represents a function that produces an int-valued result.
41. ToLongBiFunction : Represents a function that accepts two arguments and produces a long-valued result.
42. ToLongFunction : Represents a function that produces a long-valued result.
43. UnaryOperator : Represents an operation on a single operand that produces a result of the same type as its operand. : : :

Posted by K Himaanshu Shuklaa at 6/01/2016 10:43:00 AM

G+1 Recommend this on Google

Labels: Java, Java 8, Java 8 Interview Questions and Answers, Java Interview Questions, Lambda Expression

No comments:

Post a Comment

Enter your comment...

Comment as: Select profile...

Publish Preview

Links to this post

Create a Link

BACK NEXT

Followers (185) Next

Follow