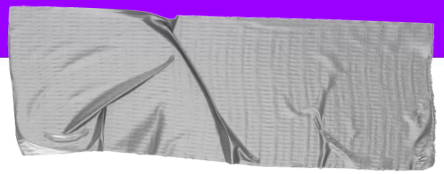

# Java Certified #11

A question lead guide to prepare Java certification



## Working with Streams and Lambda expressions

Given:

```
List<String> abc = List.of( "a", "b", "c" );
abc.stream()
   .forEach( x -> {          x = x.toUpperCase();   } );
abc.stream()
   .forEach( System.out::print );
```

What is the output?

- ➔ **abc**
- ➔ **ABC**
- ➔ **An exception is thrown.**
- ➔ **Compilation fails.**

**abc**

The output of the given code is `abc`.
Let's break down why this is the case:

Step-by-Step Explanation:
**List Initialization:** `List<String> abc = List.of("a", "b", "c");`
abc is a reference to an immutable list containing the strings "a", "b", and "c".
**First Stream with forEach:**
abc.stream()
.forEach(x -> { x = x.toUpperCase(); });
This code creates a stream from the list and iterates over each element with forEach.
Within the lambda expression `x -> { x = x.toUpperCase(); }`, x is a new local variable that holds the reference value of each element of the list.
When `x = x.toUpperCase();` is executed, it changes the local variable x to point to a new string object that is the uppercase version of the original string (e.g., "a" becomes "A").
However, this reassignment does not affect the original list because x is a local variable within the lambda.
The reference held by the list itself remains unchanged.
Here's a visualization:
For the first element `"a"`:
`x` initially holds the reference to `"a"`.
`x = x.toUpperCase();` makes x hold the reference to `"A"`.
The list still holds the reference to `"a"`.
The same process repeats for `"b"` and `"c"`.

**abc**

Second Stream with `forEach`:
abc.stream().forEach( System.out::print);
This creates another stream from the list and iterates over each element with `forEach`.
It prints each element of the list.
Since the elements of the list were not modified by the previous forEach operation *(because only the local variable x was changed, not the elements in the list*), the second forEach prints the original elements of the list: `"a", "b", "c"`.

**Key Points:**
**Pass by Value:** Java passes the value of the reference to the lambda.
This means x is a new variable within the lambda that holds the reference to the original string.
Local Variable Reassignment:
When `x = x.toUpperCase();` is executed, it only changes the local variable `x` to point to a new string.
It does not modify the original list because `x` is a local variable within the lambda expression.
**Immutable List:** The list created with `List.of(...)` is immutable, so its elements cannot be modified directly.

**Conclusion:**
Because the original list elements are not modified, the output of the code is: `abc`
This aligns with the understanding that Java uses pass by value, creating new variables in the lambda that hold the reference values, but does not alter the original references held by the list.

https://www.infoworld.com/article/2265404/does-java-pass-by-reference-or-pass-by-value.html

## Using Object-Oriented Concepts in Java

What do the following print?

```
public class Main {
    int instanceVar = staticVar;
    static int staticVar = 666;
    public static void main( String args[] ) {
        System.out.printf( "%d %d", new Main().instanceVar, staticVar );
    }
    static { staticVar = 42; }
}
```

- ➔ **666 42**
- ➔ **666 666**
- ➔ **42 42**
- ➔ **Compilation fails**

# 42 42

Explanation:

1. **Order of Declaration**: In Java, instance variables can reference static variables because static variables are initialized before any instance variables, regardless of the order of their declarations in the class.
2. **Static Variable Initialization**:
   - `staticVar` is declared as a static variable with an initial value of `666`.
   - Then, a `static` block updates the value of `staticVar` to `42`. The static block is executed before the main method when the class is loaded.
3. **Instance Variable Initialization**:
   - `instanceVar` is initialized to the current value of `staticVar`. At this point, `staticVar` already holds the value `42` due to the static block.
4. **Execution of `main` Method**:
   - A new instance of `Main` is created (`new Main()`), and the `instanceVar` of this object is accessed. Its value is `42`.
   - The value of the static variable `staticVar` is also `42`.

The program prints `42 42`.

Key Points to Remember:

- Static variables are initialized first, including updates in static blocks.
- Instance variables referencing static variables get the value at the time of their initialization.

## Using Object-Oriented Concepts in Java

What do the following print?

```
public class DefaultAndStaticMethods {
    public static void main( String[] args ) { WithStaticMethod.print(); }
}
interface WithDefaultMethod {
    default void print() { System.out.print( "default" ); }
}
interface WithStaticMethod extends WithDefaultMethod {
    static void print() { System.out.print( "static" ); }
}
```

- ➔ **default**
- ➔ **static**
- ➔ **nothing**
- ➔ **Compilation fails**

# Compilation fails

The issue is that **interface static methods cannot override instance methods, including default methods.**

Breakdown of the Code:

1. **Default Method in `WithDefaultMethod`:**
   - The `WithDefaultMethod` interface defines a default method `print()`. A default method is an **instance method** with a default implementation.}
2. **Static Method in `WithStaticMethod`:**
   - The `WithStaticMethod` interface extends `WithDefaultMethod` and declares a static method `print()`.
   - However, Java does not allow a static method in an interface to have the same name and signature as an instance method (including default methods) in a parent interface. This creates a conflict because the static method is not overriding the default method—it simply cannot coexist with it.
3. **Compilation Error:**
   - Static method 'print()' in 'WithStaticMethod' cannot override instance method 'print()' in 'WithDefaultMethod'.

Why Is This Forbidden?

- Static interface methods are associated with the **interface itself**, not its instances. They are not part of the instance method table.
- Default methods, on the other hand, are instance methods.
- Allowing both with the same name would create ambiguity in method resolution


https://bit.ly/javaOCP