Chuwa - Java Backend 1219 Assignment

Delin "Angelina" Liang

29 Dec 2025

Question 1. What is the primary reason Java 8 introduced default methods in interfaces? Explain how default methods help maintain backward compatibility when adding new methods to existing interfaces. Give an example of a JDK interface that uses default methods.

ANSWER

Reasoning:
The primary reason Java 8 introduced default methods in interfaces was to maintain backward compatibility while allowing interfaces to evolve

Before Java 8: Adding a new method to an existing interface would break all implementing classes, because they would be required to implement the new method. Default methods solve this problem by providing a default implementation in the interface itself, so existing classes do not need to be modified

Example:
In the `List` interface.
Methods like `forEach()` were added as default method in Java 8, allowing existing implementations such as `ArrayList` to continue working without changes

Question 2. Compare default methods and static methods in interfaces. What are the key differences in (1) how they are called, (2) whether they can be overridden, and (3) their typical use cases?

ANSWER

How they are called

- **Default methods** are called on an instance of a class that implements the interface
- **Static methods** are called directly on the interface, not on instances

Whether they can be overridden

- **Default methods** CAN be overridden by implementing classes
- **Static methods** CANNOT be overridden, because they belong to the interface itself

Typical use cases

- **Default methods** are used to add new behavior to interfaces while maintaining backward compatibility and allowing implementing classes to customize the behavior if needed
- **Static methods** are used as utility or helper methods that are logically related to the interface but do not depend on instance state

Question 3. What is a Functional Interface? What role does the `@FunctionalInterface` annotation play? Can a functional interface have multiple default methods? Explain with an example

ANSWER

A **Functional Interface** is an interface that contains exactly one abstract method. It can have any number of default methods and static methods, but **only one** abstract method, which enables it to be implemented using a **lambda expression**

The `@FunctionalInterface` annotation is an optional annotation that serves as a compile-time sanity check. It tells the compiler to verify that the interface has **exactly one abstract method**, and let the compiler report an error if the check failed

A functional interface **CAN have multiple default methods**, as default methods do not count as abstract methods

Example:

```
@FunctionalInterface
public interface StringProcessor {
    String process(String input);
    default String toUpper(String input) {
        return input.toUpperCase();
    }
    default String addPrefix(String input) {
        return "Result: " + input;
    }
}
```

Question 4. Describe the four major categories of functional interfaces in java.util.function package. For each category (Consumer, Supplier, Predicate, Function), explain (1) its method signature, (2) what it represents, and (3) give one practical use case.

ANSWER

Consumer

- Method signature:
  `void accept(T t)`
- What it represents
  An operation that **takes an input and returns no result**
- Practical use case:
  Printing elements of a collection or logging information

Supplier

- Method signature:
  `T get()`
- What it represents
  A **provider** of values that takes no input and returns a result
- Practical use case:
  Lazily creating objects or providing default values

Predicate

- Method signature:
  `boolean test(T t)`
- What it represents
  A boolean-valued condition used for **testing or filtering**
- Practical use case:
  Filtering elements in a stream

Function

- Method signature:
  `R apply(T t)`
- What it represents
  A **transformation** from one value to another
- Practical use case:
  Mapping data from one form to another

Question 5. What is "effectively final"? Why does Java require variables used in lambda expressions to be final or effectively final? What happens if you try to modify a variable before or after it's used in a lambda?

ANSWER

A local variable is **effectively final** if its value is **assigned once and never changed**, even if it is not explicitly declared with the final keyword

"Why?"

- Lambdas may execute after the surrounding method returns
- Java captures variable values, not references
- Enforcing immutability avoids concurrency, lifetime, and consistency issues

If a local variable is **modified**, it is no longer effectively final, and the code will **fail to compile**

Question 6. Explain the difference between the three types of method references: (1) ClassName::staticMethod , (2) object::instanceMethod , and (3) ClassName::instanceMethod . For type 3, explain why the first parameter becomes the calling object

ANSWER

- `ClassName::staticMethod`
  Refers to a static method of a class
- `object::instanceMethod`
  Refers to an instance method of a specific object
- `ClassName::instanceMethod`
  Refers to an instance method of a class, but without specifying a particular object
  - **Why does the first parameter become the calling object?** Because an instance method requires a receiver object to be invoked on, Java uses the **first parameter of the lambda** as that receiver

Question 7. What is the difference between Optional.of() and Optional.ofNullable() ? When should you use each one? What happens if you pass null to Optional.of() ?

ANSWER

Difference

- `Optional.of()`
  Requires a non-null value. It is used when you are certain the value is not null
- `Optional.ofNullable()`
  Accepts either a non-null value or null. If the value is null, it returns an empty `Optional`

When to use?

- `Optional.of()`
  When a null value indicates a programming error and should fail fast
- `Optional.ofNullable()`
  When the value may legitimately be null, and you want to handle the absence safely

Passing a `null` to `Optional.of()`
causes a `NullPointerException` at runtime

Question 8. Compare orElse() and orElseGet() methods in Optional. Which one is more efficient when the default value is expensive to compute? Explain with an example.

ANSWER `orElse()`

- The default value is eagerly evaluated, meaning it is computed regardless of whether the Optional contains a value

`orElseGet()`

- The default value is lazily evaluated, meaning it is computed only if the Optional is empty

Efficiency:
`orElseGet()` **is more efficient** when the default value is expensive to compute, because the computation is performed only when needed.
See this example:

```
Optional<String> opt = Optional.of("value");

String result1 = opt.orElse(expensiveComputation());
String result2 = opt.orElseGet(() -> expensiveComputation());
```

In this example, `expensiveComputation()` is always executed with `orElse()`, but executed only when the Optional is empty with `orElseGet()`

Question 9. Why is Optional.get() considered dangerous? What are the recommended alternatives? Explain the best practices for extracting values from Optional objects.

ANSWER

`Optional.get()` is considered dangerous because it throws `NoSuchElementException` if the `Optional` is empty. Calling `get()` without first checking `isPresent()` defeats the purpose of using `Optional` and can lead to runtime exceptions

Recommended Alternatives
`orElse()`
`orElseGet()`
`orElseThrow()`
`ifPresent()` / `ifPresentOrElse()`
`map()` / `flatMap()`

Best Practices

- Avoid using `get()` directly
- Prefer explicit handling of the empty case using `orElse()`, `orElseGet()`, or `orElseThrow()`
- Use functional-style methods (`map`, `ifPresent`) to operate on the value without unwrapping it
- Treat Optional as a tool to model absence explicitly, not as a container to bypass null checks