Assignment 12/19/2025

Q1. 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.

- The default methods in interfaces are primarily to enable evolving existing interfaces without breaking the code of classes that already implement them.
- When adding a new method to an interface, we could provide a default method so that the existing code that implemented this interface do not need to add this method unless they want to override it.
- java.util.List uses default method for .sort().

Q2. 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?

- 1) default methods are called on the instance of a class that implements the interface. Static methods are called on the interface itself
- 2) default methods can be overridden by implementing classes. Static methods cannot be overridden
- 3) default methods are used when we want to add a new behavior to an interfect without breaking the existing implementations. Static methods are used when we need a helper method related to the interface.

Q3. What is a Functional Interface? What role does the @FunctionalInterface annotation play? Can a functional interface have multiple default methods? Explain with an example.

- A functional interface is an interface that has only one abstract method. It comes with the lambda expressions and method references from Java 8.
- The @FunctionalInterface annotation is used for sanity check only, but not required for definition.
- Yes it could have multiple default methods.

Q4. 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.

- The four major categories are: Consumer, Supplier, Predicate, and Function.
- Consumer: 1) method signature: void accept(T t). 2) It represents taking one input and return nothing. 3) We could use it when we want to do something with data, but don't need a result.
- Supplier: 1) method signature: T get(). 2) It represents taking no input but returning a value. 3) We could use it when values are generated or fetched on demand.
- Predicate: 1) method signature: boolean test(T t). 2) It represents taking one input and returning true/false. 3) We could use it to test conditions or filter collections.

- Function: 1) method signature: R apply(T t). 2) It represents taking one input and returning a value. 3) We could use it to map or convert data from one form to another.

Q5. 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?

- "effectively final" means the variable is assigned only once, and its value never changes after initialization.
- The variables in lambda expressions need to be final or effectively final because it could avoid concurrency and state confusion.
- There will be compile error if we try to modify the variable used in a lambda

Q6. 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.

- (1) ClassName::staticMethod: It refers to a static method defined in a class. The method belongs to the class, all parameters come directly from the lambda, and no object instance is involved.
- (2) object::instanceMethod: It refers to an instance method on a specific existing object. The object is already known, the lambda's parameters are passed to the method, and the object is bound at creation time.
- (3) ClassName::instanceMethod: It refers to an instance method, but without specifying which object. The first parameter of the lambda becomes the object, remaining parameters are passed to the method, and the instance is determined at call time

Q7. 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()?

- Both methods are used to create an Optional, but they differ critically in how they handle null.
- Use Optional.of() when a value must never be null and you want to fail fast, and use Optional.ofNullable() when null is a valid possibility and should result in an empty Optional.

Q8. Compare orElse() and orElseGet() methods in Optional. Which one is more efficient when the default value is expensive to compute? Explain with an example.

- Both methods are used to provide a fallback value when an Optional is empty, but they differ in when the default value is computed.
- orElseGet() is more efficient when the default value is expensive to compute, because it avoids unnecessary work.

Q9. Why is Optional.get() considered dangerous? What are the recommended alternatives? Explain the best practices for extracting values from Optional objects.

- Optional.get() is considered dangerous because it can throw an exception at runtime if the Optional is empty, and it often encourages treating Optional like a nullable value instead of handling absence explicitly.
- orElseThrow(): best when absence is an error
- orElseGet(): best for expensive defaults
- orElse(): best for cheap constants
- ifPresent() / ifPresentOrElse(): best for side effects
- map() / flatMap(): best for transformations and chaining