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 primary reason Java 8 introduced default methods in interfaces was to allow interfaces to evolve without breaking existing implementations. By providing a default implementation, new methods can be added to an interface without forcing all implementing classes to implement them. This preserves backward compatibility. A typical example is the `Collection` interface, which introduced default methods such as `stream()`.

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?

Default methods are called on an instance of an implementing class, while static methods are called on the interface itself. Default methods can be overridden by implementing classes, whereas static methods cannot be overridden. Default methods are typically used to provide shared behavior across implementations, while static methods are used as utility or helper methods 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 contains exactly one abstract method and is intended to be implemented by a lambda expression. The `@FunctionalInterface` annotation enforces this constraint at compile time and improves readability, but it is not mandatory. A functional interface can have multiple default methods because they do not count as abstract methods, such as `Comparator`, which has one abstract method and several 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 `Consumer<T>` interface defines the method signature `void accept(T t)` and represents an operation that takes an input and returns no result, commonly used for processing elements such as logging. The `Supplier<T>` interface defines `T get()` and represents a provider of values, often used for lazy value creation. The `Predicate<T>` interface defines `boolean test(T t)` and represents a boolean condition, commonly used for filtering. The `Function<T, R>` interface defines `R apply(T t)` and represents a transformation from one value to another, such as mapping objects to DTOs.

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?

A variable is effectively final if its value is assigned once and never changed afterward. Java requires variables used in lambda expressions to be final or effectively final to ensure thread safety and consistent behavior, since lambdas may be executed later or in a different context. If a variable is modified either before or after it is captured by a lambda, the code will fail to compile.

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.

`ClassName::staticMethod` refers to a static method and maps directly to a functional interface method with matching parameters. `object::instanceMethod` refers to an instance method on a specific object and binds that object as the method receiver. `ClassName::instanceMethod` refers to an instance method where the first parameter of the functional interface becomes the target object, because the method is invoked on that parameter at runtime.

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()?

`Optional.of()` should be used when the value is guaranteed to be non-null, while `Optional.ofNullable()` should be used when the value may be null. Passing null to `Optional.of()` throws a `NullPointerException`, whereas `Optional.ofNullable()` safely returns 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.

`orElse()` always evaluates its argument, while `orElseGet()` evaluates its supplier lazily. When the default value is expensive to compute, `orElseGet()` is more efficient because the computation only occurs if the `Optional` is empty. For example, `orElse(expensiveCall())` always runs the call, while `orElseGet(() -> expensiveCall())` runs it only when needed.

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 throws `NoSuchElementException` if the `Optional` is empty. Recommended alternatives include `orElse`, `orElseGet`, `orElseThrow`, and `ifPresent`, which explicitly handle the absence of a value. Best practice is to extract values in a way that makes null-handling explicit and avoids unchecked assumptions.