Java SE 8 Topic-Wise Preparation Strategy

Java 8 MCQs – Topic: Advanced Class Design

Subtopics:

- Inheritance and overriding rules
- Abstract classes vs interfaces
- Static and default methods in interfaces
- Nested classes
- Access modifiers and method resolution

Question 1

Which of the following statements about interfaces in Java 8 is correct?

a) Interfaces can have private abstract methods.
b) Interfaces can have static methods with implementations.
c) Interfaces cannot have default methods.
d) Interfaces can implement other interfaces using `extends`.

Answer: b) Interfaces can have static methods with implementations.
Reasoning: Java 8 introduced static and default methods in interfaces. Static methods must have implementations.

Question 2

What will be the output of the following code?

```
java
```

```
CopyEdit
interface A {
    default void hello() {
        System.out.println("Hello from A");
    }
}

interface B {
    default void hello() {
        System.out.println("Hello from B");
    }
}

class C implements A, B {
    public void hello() {
        A.super.hello();
    }
}

public class Test {
    public static void main(String[] args) {
        new C().hello();
    }
}
```

a) Hello from A
b) Hello from B
c) Compilation error
d) Runtime Exception

Answer: a) Hello from A
Reasoning: The class `C` resolves the ambiguity between `A` and `B` by explicitly calling `A.super.hello()`.

Question 3

Which of the following nested classes has access to all members (including private) of its outer class?

a) Static nested class
b) Anonymous class
c) Top-level class in same file
d) Inner (non-static) class

Answer: d) Inner (non-static) class
Reasoning: Inner classes (non-static nested classes) can access private members of the outer class directly.

Question 4

Choose the correct statement regarding abstract classes.

a) Abstract classes must have at least one abstract method.
b) Abstract classes can be instantiated using new.
c) Abstract classes can have constructors.
d) Abstract methods can be private.

Answer: c) Abstract classes can have constructors.
Reasoning: Abstract classes can have constructors, which are called during instantiation of subclasses.

Question 5

Which keyword is used to resolve method ambiguity in multiple inherited interfaces?

a) this
b) super
c) interface
d) interfaceName.super

Answer: d) interfaceName.super
Reasoning: Java 8 allows `InterfaceName.super.methodName()` to resolve default method ambiguity between multiple interfaces.

Question 6

Which of these declarations will **fail to compile**?

```
java
CopyEdit
interface X {
    default void print() { }
}

class Y {
    public void print() { }
}

class Z extends Y implements X { }
```

a) Compiles without error
b) Compilation error due to conflict in method print()
c) Runtime error
d) Cannot implement interface with default method

Answer: a) Compiles without error
Reasoning: Class method overrides default method in interface. No conflict as class methods take precedence over default methods.

Question 7

Which of the following is true about static methods in interfaces?

a) They can be overridden in implementing classes.
b) They belong to the instance of the class.
c) They can only be called using the interface name.
d) They are inherited like default methods.

Answer: c) They can only be called using the interface name.
Reasoning: Static methods in interfaces are **not inherited** and must be called using `InterfaceName.method()`.

Question 8

What is the access level of members of an interface by default?

a) private
b) protected
c) package-private
d) public

Answer: d) public
Reasoning: All methods in an interface are implicitly `public abstract` unless marked `default` or `static`.

Question 9

Which of the following combinations is valid for an interface?

a) `public abstract void m();`
b) `abstract default void m();`
c) `final default void m();`
d) `private abstract void m();`

Answer: a) `public abstract void m();`
Reasoning: This is the valid form. `abstract default`, `final default`, or `private abstract` are illegal combinations.

Question 10

What is the output of this nested class usage?

```
java
CopyEdit
class Outer {
    private int value = 42;

    class Inner {
        int get() {
            return value;
        }
    }
}

public class Test {
    public static void main(String[] args) {
        Outer.Inner in = new Outer().new Inner();
        System.out.println(in.get());
    }
}
```

a) 0
b) 42
c) Compilation error
d) NullPointerException

Answer: b) 42
Reasoning: Inner class can directly access private members of the outer class.

Question 11

Which of the following best describes method overriding?

a) Method in subclass must have a different name than superclass
b) Method in subclass must declare `throws` clause
c) Method in subclass must have same signature and return type (or covariant)
d) Method in subclass must be static

Answer: c) Method in subclass must have same signature and return type (or covariant)
Reasoning: Overriding requires identical method name, parameters, and compatible (covariant) return type.

Question 12

Given:

```
java
CopyEdit
abstract class A {
    abstract void test();
}

class B extends A {
    void test() { System.out.println("B"); }
}

class C extends B {
    void test() { System.out.println("C"); }
}
```

What is the result of `new C().test();`?

a) A
b) B
c) C
d) Compilation error

Answer: c) C
Reasoning: `test()` is overridden successively down the inheritance chain. `C`'s method is invoked.

Question 13

Can an abstract class implement an interface without providing implementation?

a) No, it must implement all methods
b) Yes, abstract class can defer implementation
c) Only if the interface has default methods
d) Only if the class is final

Answer: b) Yes, abstract class can defer implementation
Reasoning: Abstract classes may implement interfaces and leave the implementation to subclasses.

Question 14

Which of the following will result in a **compilation error**?

```
java
CopyEdit
interface I {
    default void go() { }
}

class A {
    public void go() { }
}

class B extends A implements I { }
```

a) Compiles successfully
b) Error due to `go()` conflict
c) Error: class can't implement interface with default method
d) Error: `go()` must be marked `@Override`

Answer: a) Compiles successfully
Reasoning: Class method overrides the interface's default method silently.

Question 15

Which two modifiers are allowed for methods in an interface? (Choose two)

a) protected
b) abstract
c) static
d) final
e) private

Answer: b) abstract, c) static
Reasoning: Interface methods can be `abstract`, `static`, or `default`. `final` and `protected` are illegal.

Question 16

What happens if a class implements two interfaces that define the same default method?

a) Compilation succeeds; method is inherited
b) Compilation fails unless overridden in the class
c) JVM picks one arbitrarily
d) Runtime exception is thrown

Answer: b) Compilation fails unless overridden in the class
Reasoning: Ambiguity must be resolved by overriding in the implementing class.

Question 17

Which concept allows different objects to be treated as instances of the same type?

a) Inheritance
b) Encapsulation
c) Polymorphism
d) Abstraction

Answer: c) Polymorphism
Reasoning: Polymorphism lets you treat a subclass object as an instance of its superclass or interface.

Question 18

Can an interface extend a class?

a) Yes
b) No
c) Only if the class is abstract
d) Only if the class is final

Answer: b) No
Reasoning: Interfaces cannot extend classes. They can only extend other interfaces.

Question 19

Which of the following statements is true?

a) You can instantiate an abstract class directly.
b) A class may extend only one class but implement multiple interfaces.

c) An interface can be declared final.
d) A method marked `default` must be overridden.

Answer: b) A class may extend only one class but implement multiple interfaces.
Reasoning: Java supports single inheritance for classes and multiple inheritance via interfaces.

Question 20

Which output is expected from this code?

```
java
CopyEdit
interface Animal {
    default void sound() {
        System.out.println("Animal");
    }
}

class Dog implements Animal {
    public void sound() {
        System.out.println("Dog");
    }
}

public class Test {
    public static void main(String[] args) {
        new Dog().sound();
    }
}
```

a) Animal
b) Dog
c) Compilation error
d) NullPointerException

Answer: b) Dog
Reasoning: Class method overrides the default method in the interface.

Question 21

What does this code print?

```
java
CopyEdit
interface A {
    default void print() {
        System.out.println("A");
    }
}

interface B extends A {
    default void print() {
        System.out.println("B");
    }
}
```

```
class C implements B {
}

new C().print();
```

a) A
b) B
c) Compilation error
d) Runtime exception

Answer: b) B
Reasoning: Interface B overrides A's default method. C inherits B's version.

Question 22

Which class definition will **fail to compile**?

```
java
CopyEdit
abstract class Shape {
    abstract void draw();
}

class Circle extends Shape {
    void draw() {
        System.out.println("Draw Circle");
    }
}

class Rectangle extends Shape {
}
```

a) All classes compile
b) Rectangle needs to implement `draw()`
c) Shape cannot be abstract
d) Circle must be abstract

Answer: b) Rectangle needs to implement `draw()`
Reasoning: Since Rectangle is not abstract, it must implement the abstract method `draw()`.

Question 23

Which is a valid abstract class?

a)

```
java
CopyEdit
abstract class Test {
    private abstract void run();
}
```

b)

```
java
```

```
CopyEdit
abstract class Test {
    abstract void run();
}
```

c)

```
java
CopyEdit
abstract class Test {
    final abstract void run();
}
```

d)

```
java
CopyEdit
abstract class Test {
    static abstract void run();
}
```

Answer: b)
Reasoning: Abstract methods cannot be `private`, `final`, or `static`.

Question 24

Which of the following statements about `instanceof` is true?

a) `instanceof` can check for interface types
b) It works only on primitive types
c) It throws an exception if the types are unrelated
d) It is evaluated at compile time only

Answer: a) `instanceof` can check for interface types
Reasoning: `instanceof` checks whether an object is an instance of a specific class or interface.

Question 25

Which of the following **modifiers** can you use with **interface methods**?

a) protected
b) default
c) native
d) volatile

Answer: b) default
Reasoning: Java 8 allows `default` methods in interfaces, not `protected`, `native`, or `volatile`.

Java 8 MCQs – Topic: Design Patterns & Principles

Subtopics:

- SOLID principles
- Functional patterns with lambdas
- Strategy pattern, Factory pattern
- DRY/KISS/YAGNI
- Refactoring best practices
- High-cohesion, low-coupling design

Question 1

Which of the following principles is violated when a class has multiple responsibilities?

a) DRY
b) KISS
c) SRP
d) LSP

Answer: c) SRP (Single Responsibility Principle)
Reasoning: SRP requires that a class should have only one reason to change. Multiple responsibilities violate this principle.

Question 2

Which design pattern best aligns with the behavior of a lambda expression?

a) Singleton
b) Strategy
c) Factory
d) Template Method

Answer: b) Strategy
Reasoning: Lambdas can be used to encapsulate interchangeable behaviors, making them ideal for implementing the Strategy pattern.

Question 3

What does the Open/Closed Principle state?

a) A class should be open for extension but closed for modification
b) A class should be open for modification at all times
c) Code should always be rewritten when requirements change
d) Class hierarchy should be deep

Answer: a) A class should be open for extension but closed for modification
Reasoning: OCP means you should be able to add new behavior without changing existing code.

Question 4

Which of the following is a **benefit** of using the Factory Pattern?

a) Reduces inheritance
b) Avoids tight coupling to object creation
c) Enables multiple inheritance
d) Promotes static design

Answer: b) Avoids tight coupling to object creation
Reasoning: Factory pattern encapsulates object creation logic, making code loosely coupled.

Question 5

In functional programming style, which is preferred?

a) Mutable shared state
b) Method overriding
c) Side-effect-free functions
d) Anonymous inner classes

Answer: c) Side-effect-free functions
Reasoning: Functional style encourages immutability and no side effects for predictability and testability.

Question 6

Which SOLID principle is directly supported by interfaces and dependency injection?

a) SRP
b) OCP
c) DIP
d) ISP

Answer: c) DIP (Dependency Inversion Principle)
Reasoning: DIP promotes coding to abstractions (interfaces), not concrete classes.

Question 7

What is the result of violating the Liskov Substitution Principle?

a) More cohesive code
b) Broken polymorphism

c) Better performance
d) Looser coupling

Answer: b) Broken polymorphism
Reasoning: LSP ensures that derived classes can stand in for base classes without breaking functionality.

Question 8

Which design principle advises **not to add functionality until it is needed**?

a) DRY
b) SRP
c) YAGNI
d) DIP

Answer: c) YAGNI (You Aren't Gonna Need It)
Reasoning: YAGNI warns against premature design complexity.

Question 9

Which of the following is a good use case for a functional interface in Java 8?

a) Representing configuration files
b) Representing a task to be deferred or executed
c) Persisting objects in database
d) Representing a thread class

Answer: b) Representing a task to be deferred or executed
Reasoning: Functional interfaces are ideal for defining logic to execute later, like in lambda-based callbacks.

Question 10

Choose the correct lambda-compatible interface for a method that takes no parameters and returns nothing.

a) `Supplier<Void>`
b) `Runnable`
c) `Consumer<Void>`
d) `Callable<Void>`

Answer: b) `Runnable`
Reasoning: Runnable has a single `run()` method with no parameters and no return value.

Question 11

Which of the following can help apply **Strategy Pattern** using Java 8?

a) Method overloading
b) Class inheritance
c) `Function<T, R>`
d) Static utility methods

Answer: c) `Function<T, R>`
Reasoning: The functional interface `Function<T, R>` can be passed to switch strategies dynamically.

Question 12

Which design principle is violated when an interface contains unrelated methods?

a) ISP
b) OCP
c) LSP
d) DIP

Answer: a) ISP (Interface Segregation Principle)
Reasoning: ISP requires that interfaces should have only the methods that are meaningful to the implementer.

Question 13

What does the term **high cohesion** mean in OOP?

a) A class knows about many others
b) A class has many responsibilities
c) A class is focused on a single task
d) Classes are loosely related

Answer: c) A class is focused on a single task
Reasoning: High cohesion indicates that a class is tightly focused and easier to maintain.

Question 14

Which Java 8 feature enables **loose coupling** between components?

a) Method overloading
b) Lambda expressions
c) Static binding
d) Object serialization

Answer: b) Lambda expressions
Reasoning: Lambdas allow injecting behavior, which decouples the implementation logic.

Question 15

Which best applies the DRY (Don’t Repeat Yourself) principle?

a) Copying code for faster prototyping
b) Writing reusable methods
c) Declaring variables inside loops
d) Avoiding abstraction

Answer: b) Writing reusable methods
Reasoning: DRY encourages reducing repetition by reusing functions and abstractions.

Java 8 MCQs – Topic: Design Patterns & Principles (Set 2 of 25–25)

Question 16

What is the **primary benefit** of the Builder Pattern?

a) Encapsulates complex factory logic
b) Eliminates need for constructors
c) Allows creation of immutable objects with optional parameters
d) Replaces the Singleton Pattern

Answer: c) Allows creation of immutable objects with optional parameters
Reasoning: The Builder pattern helps create complex objects in a readable and flexible way, especially useful with many optional fields.

Question 17

Which principle does the following violate?

```
java
CopyEdit
class ReportGenerator {
    public void generate() { /* logic */ }
    public void saveToDatabase() { /* DB logic */ }
    public void print() { /* print logic */ }
}
```

a) DRY
b) SRP
c) ISP
d) DIP

Answer: b) SRP
Reasoning: The class has multiple responsibilities (generation, persistence, printing), violating SRP.

Question 18

What design pattern does the following snippet implement?

```
java
CopyEdit
public class Logger {
    private static Logger instance = new Logger();
    private Logger() {}
    public static Logger getInstance() {
        return instance;
    }
}
```

a) Factory
b) Builder
c) Prototype
d) Singleton

Answer: d) Singleton
Reasoning: This is a classic implementation of the Singleton pattern.

Question 19

Which functional interface from Java 8 fits a **command** or **task** abstraction?

a) `Predicate<T>`
b) `Consumer<T>`
c) `Runnable`
d) `Function<T, R>`

Answer: c) `Runnable`
Reasoning: `Runnable` represents a command or task that takes no arguments and returns no result.

Question 20

Which pattern is most useful when behavior changes based on type at runtime without using `if-else`?

a) Observer
b) Decorator
c) Strategy
d) Singleton

Answer: c) Strategy
Reasoning: Strategy pattern allows switching behavior at runtime via interchangeable strategy objects.

Question 21

What is the consequence of violating the **Dependency Inversion Principle**?

a) Code is tightly coupled to concrete classes
b) You must use abstract classes
c) Objects can no longer be serialized
d) It disables static imports

Answer: a) Code is tightly coupled to concrete classes
Reasoning: DIP advocates for depending on abstractions rather than concrete implementations.

Question 22

Which of these best represents **loose coupling** in Java 8?

a) Using reflection to discover class behavior
b) Using `new` to instantiate concrete implementations
c) Injecting a `Predicate<T>` as filter criteria
d) Making all fields public

Answer: c) Injecting a `Predicate<T>` as filter criteria
Reasoning: Passing behavior via functional interfaces like `Predicate` decouples decision logic from object creation.

Question 23

What does the **KISS** principle advocate?

a) Keep Interfaces Small & Secure
b) Keep Inheritance Strategy Specific
c) Keep It Simple, Stupid
d) Keep Inner Static Singletons

Answer: c) Keep It Simple, Stupid
Reasoning: KISS promotes simplicity in code design and avoids overengineering.

Question 24

Which of the following violates the **Interface Segregation Principle**?

```
java
CopyEdit
```

```
interface Machine {
    void print();
    void fax();
    void scan();
}
```

a) Class implementing only `print()`
b) Interface with multiple unrelated methods
c) Interface extending another interface
d) None of the above

Answer: b) Interface with multiple unrelated methods
Reasoning: ISP encourages creating focused interfaces. Unrelated operations should be separated.

Question 25

Which is the best reason to use **lambdas** over anonymous inner classes in design?

a) Improved runtime performance
b) More flexible type-checking
c) Cleaner, more concise syntax
d) Better exception handling

Answer: c) Cleaner, more concise syntax
Reasoning: Lambdas improve readability and are much more concise than verbose anonymous inner classes.

Java 8 MCQs – Topic: Generics & Collections

Subtopics:

- Raw vs parameterized types
- Wildcards: `?`, `? extends T`, `? super T`
- Type inference and diamond operator <>
- Generic methods
- Collections API integration
- Comparable vs Comparator

Question 1

What is the output of this code?

```
java
CopyEdit
List<String> list = new ArrayList<>();
```

```
list.add("A");
list.add("B");
list.add("C");

for (String s : list) {
    System.out.print(s + " ");
}
```

a) A B C
b) Compilation error
c) NullPointerException
d) RuntimeException

Answer: a) A B C
Reasoning: Iterating a properly typed list works as expected.

Question 2

Which wildcard allows reading elements **but not adding**, except `null`?

a) `List<Object>`
b) `List<? super Number>`
c) `List<? extends Number>`
d) `List<?>`

Answer: c) `List<? extends Number>`
Reasoning: `? extends T` is a **producer** (read-only); adding is not allowed (except `null`) because the actual type is unknown.

Question 3

Which of the following declarations is **invalid**?

a) `List<?> list = new ArrayList<String>();`
b) `List<? super Integer> list = new ArrayList<Number>();`
c) `List<String> list = new ArrayList<? extends String>();`
d) `List<? extends Number> list = new ArrayList<Integer>();`

Answer: c)
Reasoning: You cannot use wildcards (`<? extends String>`) in **instantiation** on the right-hand side like that.

Question 4

Choose the correct usage of the **diamond operator** <>.

a) `Map<String, Integer> map = new HashMap<String, Integer>();`
b) `List<> list = new ArrayList<String>();`

c) `Map<String, Integer> map = new HashMap<>();`
d) `List list = new ArrayList<>();`

Answer: c)
Reasoning: Java 7+ allows `<>` on the right to infer type parameters. Option `b` is invalid (no type in diamond).

Question 5

What is the purpose of `Comparator.comparing()` in Java 8?

a) It sorts using natural order
b) It compares two maps
c) It builds a comparator based on a key extractor function
d) It is used for filtering

Answer: c)
Reasoning: `Comparator.comparing()` is a functional-style way to create a comparator from a lambda key extractor.

Question 6

What does `Collections.unmodifiableList(list)` return?

a) A deep clone of `list`
b) A read-only view of `list`
c) A shallow copy of `list`
d) A new modifiable list

Answer: b)
Reasoning: The returned list is read-only; any attempt to modify it throws `UnsupportedOperationException`.

Question 7

Which of these allows adding elements?

a) `List<? super Number>`
b) `List<? extends Number>`
c) `List<?>`
d) `List<? extends Object>`

Answer: a)
Reasoning: `? super Number` allows adding Number or its subclasses safely.

Question 8

What will this generic method return?

```
java
CopyEdit
public static <T> T identity(T value) {
    return value;
}
```

a) Compilation error
b) It returns a deep copy of `value`
c) It returns the exact same object
d) It returns null

Answer: c)
Reasoning: The method simply returns what it was passed.

Question 9

Which functional interface can be used for sorting a list?

a) `Predicate<T>`
b) `Function<T, R>`
c) `Consumer<T>`
d) `Comparator<T>`

Answer: d)
Reasoning: `Comparator<T>` has `compare(T o1, T o2)` used for sorting logic.

Question 10

Which is true about raw types?

a) They provide compile-time safety
b) They are used to preserve backward compatibility
c) They allow adding primitives
d) They are preferred in Java 8

Answer: b)
Reasoning: Raw types exist to maintain compatibility with pre-generics code.

Question 11

Which code correctly defines a bounded type parameter?

a) `<T super Number>`
b) `<T> T extends Number`
c) `<T extends Number>`
d) `<T implements Number>`

Answer: c) `<T extends Number>`
Reasoning: Bounded type parameters use `extends` for classes and interfaces (yes, also for interfaces).

Question 12

Which of these can be used with `forEach` in a Java 8 stream?

a) `Runnable`
b) `Function<T, R>`
c) `Predicate<T>`
d) `Consumer<T>`

Answer: d) `Consumer<T>`
Reasoning: `forEach` consumes each element without returning anything—perfect use case for `Consumer<T>`.

Question 13

Which of the following will compile?

a) `List<int> list = new ArrayList<>();`
b) `List<?> list = new ArrayList<String>();`
c) `List<? extends Object> list = new ArrayList<int>();`
d) `List<T> list = new ArrayList<T>();`

Answer: b)
Reasoning: Primitive types like `int` can't be used as type parameters. `<?>` accepts any object type safely.

Question 14

How can you sort a list of Strings in reverse order using Java 8 streams?

a) `list.sort((a, b) -> a.compareTo(b));`
b) `Collections.sort(list);`
c) `list.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toList());`
d) `list.stream().collect(Collectors.reverseOrder());`

Answer: c)
Reasoning: The sorted stream followed by `Comparator.reverseOrder()` returns elements in descending order.

Question 15

What is the correct return type of `stream().filter(...)` on a `List<String>`?

a) `String`
b) `List<String>`
c) `Stream<String>`
d) `Optional<String>`

Answer: c) `Stream<String>`
Reasoning: `filter()` operates on a stream and returns a new stream with filtered elements.

Question 16

What happens if we try to add a String to `List<? extends Number>`?

a) Compilation error
b) Runtime exception
c) It adds successfully
d) String is auto-boxed

Answer: a) Compilation error
Reasoning: `? extends Number` is a **producer** type. We cannot safely add any object except `null`.

Question 17

Given `List<? super Integer>`, which element can be added?

a) `"hello"`
b) `1.5`
c) `new Object()`
d) `new Integer(5)`

Answer: d)
Reasoning: Only `Integer` or its subtypes are allowed when `? super Integer` is the declaration.

Question 18

Choose the correct statement:

a) `List<Object>` can be assigned to `List<String>`
b) `List<? extends Object>` can be written to
c) `List<? super String>` can accept a String
d) `List<String>` and `List<Integer>` are interchangeable with casting

Answer: c)
Reasoning: `? super String` ensures type-safety for writing String values into the list.

Question 19

What does `peek()` do in Java 8 Stream?

a) Filters elements
b) Removes duplicates
c) Transforms data
d) Performs a side-effect without modifying the stream

Answer: d)
Reasoning: `peek()` is used for debugging/logging—side-effects during intermediate operations.

Question 20

Which interface is **not** a functional interface?

a) `Runnable`
b) `Callable<T>`
c) `Comparator<T>`
d) `List<T>`

Answer: d) `List<T>`
Reasoning: Functional interfaces must have exactly one abstract method. `List<T>` has many.

Question 21

What is the **main advantage** of using generics?

a) Slower compilation
b) Ability to mix object types
c) Type-safety at runtime
d) Compile-time type-safety

Answer: d)
Reasoning: Generics prevent ClassCastException by enforcing type checks during compilation.

Question 22

Which functional interface does `Comparator<T>` implement?

a) `BiFunction<T,T,Integer>`
b) `Predicate<T>`

c) `BinaryOperator<T>`
d) None – it is standalone

Answer: d)
Reasoning: `Comparator<T>` is a functional interface itself, not derived from another one.

Question 23

Which method is used to convert a stream into a list?

a) `toArray()`
b) `toList()`
c) `collect(Collectors.toList())`
d) `join()`

Answer: c)
Reasoning: The collector is required to aggregate a stream into a collection like `List`.

Question 24

Which of the following best describes `Optional<T>`?

a) It is a wrapper to avoid `try-catch`
b) It is a thread-safe singleton
c) It is a container to avoid `null`
d) It is an annotation processor

Answer: c)
Reasoning: `Optional<T>` is a container to express that a value **may or may not** be present.

Question 25

Which declaration allows a method to accept a list of any numeric type?

a) `void test(List<? extends Number> list)`
b) `void test(List<Number> list)`
c) `void test(List<Integer> list)`
d) `void test(List<Object> list)`

Answer: a)
Reasoning: `? extends Number` allows you to pass in `List<Integer>`, `List<Double>`, etc.

Subtopics:

- Functional interface types (`Predicate`, `Function`, `Consumer`, `Supplier`)
- Lambda syntax and variable capture
- Method references
- Anonymous class vs lambda
- Effectively final variables
- Exception handling in lambdas

Question 1

Which of the following is a valid functional interface?

a) An interface with two abstract methods
b) An interface with no methods
c) An interface with one abstract method and multiple default/static methods
d) An interface with a single default method

Answer: c)
Reasoning: A functional interface must have only **one abstract method**, but may have any number of `default` or `static` methods.

Question 2

What is the return type of `Predicate<T>`?

a) void
b) T
c) boolean
d) Optional<T>

Answer: c) boolean
Reasoning: `Predicate<T>` is used to evaluate a condition and returns a boolean.

Question 3

Which functional interface is used to perform an action without returning a result?

a) `Function<T, R>`
b) `Consumer<T>`
c) `Supplier<T>`
d) `Predicate<T>`

Answer: b) `Consumer<T>`
Reasoning: `Consumer<T>` accepts a value and performs an operation but returns nothing (`void`).

Question 4

What does the following lambda expression do?

```
java
CopyEdit
x -> x + 10
```

a) Adds 10 to x and returns it
b) Prints x
c) Multiplies x by 10
d) Compiles with error

Answer: a)
Reasoning: It's a simple lambda that takes one argument `x` and returns `x + 10`.

Question 5

Which of the following best describes a **method reference**?

a) A way to call a method dynamically
b) A compact lambda expression referring to a method
c) A method that implements a functional interface
d) An anonymous class

Answer: b)
Reasoning: Method references (`Class::method`) are shorthand for lambdas that just call an existing method.

Question 6

Which of the following is the correct syntax for a method reference?

a) `Object => method`
b) `::method()`
c) `ClassName::methodName`
d) `methodName::Class`

Answer: c)
Reasoning: Method references follow the syntax `ClassName::methodName`.

Question 7

What does `Supplier<T>` represent?

a) A function that consumes a value and returns nothing
b) A function that returns a value and takes no input
c) A function that filters a collection
d) A function that compares two values

Answer: b)
Reasoning: `Supplier<T>` is used to supply values on demand; it takes no input and returns a value.

Question 8

Which lambda is equivalent to `Function<String, Integer> f = s -> s.length();`?

a) `s -> { return s.length(); }`
b) `(String s) -> s.length()`
c) `String::length`
d) All of the above

Answer: d)
Reasoning: All three expressions are valid and do the same thing: map a `String` to its length.

Question 9

Which of the following **will not compile**?

a) `Predicate<String> p = s -> s.isEmpty();`
b) `Supplier<String> s = () -> "hello";`
c) `Function<String> f = s -> s.toUpperCase();`
d) `Consumer<String> c = System.out::println;`

Answer: c)
Reasoning: `Function` requires two type parameters: input and output. Correct usage: `Function<String, String>`.

Question 10

Which of these variables can be used in a lambda?

```
java
CopyEdit
String prefix = "Hello ";
Consumer<String> c = s -> System.out.println(prefix + s);
```

a) `prefix` must be final
b) `prefix` must be static
c) `prefix` must be effectively final
d) `prefix` must be public

Answer: c)
Reasoning: Variables used inside lambdas must be effectively final — they must not be modified after initialization.

Question 11

What happens if a lambda expression throws a checked exception?

a) It is always allowed
b) The compiler infers the exception
c) Compilation fails unless the interface declares the exception
d) The lambda silently swallows the exception

Answer: c)
Reasoning: If the functional interface method does not declare a checked exception, the lambda cannot throw it.

Question 12

Which of the following is **not** a built-in functional interface in Java 8?

a) `BiConsumer<T, U>`
b) `UnaryOperator<T>`
c) `Comparator<T>`
d) `Iterable<T>`

Answer: d)
Reasoning: `Iterable<T>` is not a functional interface. It has multiple abstract methods.

Question 13

Choose the correct behavior of this lambda:

```
java
CopyEdit
(IntPredicate p) -> p.test(5)
```

a) Returns `true` if 5 passes the test
b) Always returns false
c) Always throws an exception
d) Does not compile

Answer: a)
Reasoning: `IntPredicate` is a primitive specialization that tests an `int` for a boolean condition.

Question 14

Which interface is best suited for converting one value to another?

a) `Supplier<T>`
b) `Consumer<T>`
c) `Predicate<T>`
d) `Function<T, R>`

Answer: d)
Reasoning: `Function<T, R>` maps a value of type T to another value of type R.

Question 15

Which lambda is valid for a `BiFunction<Integer, Integer, Integer>`?

a) `(a, b) -> a + b`
b) `a, b -> a + b`
c) `a -> a + b`
d) `(a, b) -> return a + b`

Answer: a)
Reasoning: `(a, b) -> a + b` is valid shorthand when returning a value without braces or `return`.

Question 16

What does `Predicate<T>.negate()` return?

a) A reversed predicate
b) A null predicate
c) A compiled lambda
d) A function of type `Function<T, Boolean>`

Answer: a)
Reasoning: `negate()` returns a new `Predicate<T>` that is the logical negation of the current predicate.

Question 17

Which is the correct use of a `BiConsumer<String, Integer>`?

a) `System.out::println`
b) `(s, i) -> System.out.println(s + i)`
c) `s -> System.out.println(s)`
d) `() -> System.out.println("Hello")`

Answer: b)
Reasoning: `BiConsumer<T, U>` takes two arguments and performs a side-effect.

Question 18

Which feature of lambda enables **lazy execution**?

a) Runtime compilation
b) Stream operations
c) Method overloading
d) Boxing

Answer: b)
Reasoning: Streams in Java 8 are evaluated **lazily**, and lambdas delay execution until a terminal operation is called.

Question 19

Given:

```
java
CopyEdit
List<String> list = Arrays.asList("a", "b", "c");
list.forEach(System.out::println);
```

What is the functional interface in use?

a) `Predicate<String>`
b) `Supplier<String>`
c) `Consumer<String>`
d) `Function<String, Void>`

Answer: c)
Reasoning: `forEach` uses a `Consumer<T>` to perform actions with no return value.

Question 20

Which of the following statements is false?

a) Lambdas can access static variables
b) Lambdas can access effectively final variables
c) Lambdas can modify local variables
d) Lambdas can be assigned to functional interfaces

Answer: c)
Reasoning: Local variables referenced from a lambda must be **effectively final** and cannot be modified.

Question 21

How many abstract methods can a functional interface have?

a) One or more
b) Exactly one
c) Zero
d) Only static and default methods

Answer: b)
Reasoning: A functional interface must have exactly one abstract method to support lambda expressions.

Question 22

Which lambda can be used with `BinaryOperator<T>`?

a) `(a, b) -> a + b`
b) `(x) -> x.toUpperCase()`
c) `() -> "Hello"`
d) `x -> x > 10`

Answer: a)
Reasoning: `BinaryOperator<T>` takes two arguments of the same type and returns the same type.

Question 23

Which best describes a `UnaryOperator<T>`?

a) `Function<T, T>`
b) `Predicate<T>`
c) `Consumer<T>`
d) `Supplier<T>`

Answer: a)
Reasoning: A `UnaryOperator<T>` is a `Function<T, T>` — input and output types are the same.

Question 24

What does the following code do?

```
java
CopyEdit
Predicate<String> p = s -> s != null;
System.out.println(p.test("abc"));
```

a) Prints `false`
b) Prints `true`
c) Compilation error
d) Runtime exception

Answer: b)
Reasoning: `"abc"` is not null, so `test` returns `true`.

Question 25

Which of these allows **capturing external variables**?

a) Static method reference
b) Lambda expression
c) Anonymous inner class
d) Both b and c

Answer: d)
Reasoning: Both lambdas and anonymous inner classes can capture effectively final variables.

Java 8 MCQs – Topic: Streams & Collectors

Subtopics:

- Stream pipeline structure
- Intermediate vs terminal operations
- Common stream methods (`map`, `filter`, `flatMap`, `collect`, `reduce`)
- Collector API (`groupingBy`, `partitioningBy`, `joining`)
- Order of execution and laziness

Question 1

What is the output of the following?

```
java
CopyEdit
Stream.of("a", "b", "c").map(String::toUpperCase).forEach(System.out::print);
```

a) abc
b) ABC

c) Compilation error
d) Runtime exception

Answer: b) ABC
Reasoning: Stream transforms all strings to uppercase and prints them in the original order.

Question 2

Which of the following is **not** a terminal operation?

a) `forEach()`
b) `collect()`
c) `filter()`
d) `reduce()`

Answer: c) `filter()`
Reasoning: `filter()` is an intermediate operation; it builds a new stream for further processing.

Question 3

What does `flatMap()` do?

a) Maps values to keys
b) Flattens nested streams
c) Filters duplicate values
d) Concatenates two streams

Answer: b)
Reasoning: `flatMap()` transforms each element into a stream and flattens them into a single stream.

Question 4

Which method is used to convert a stream into a `List`?

a) `toArray()`
b) `collect(Collectors.toList())`
c) `flatMap()`
d) `listify()`

Answer: b)
Reasoning: The `collect()` method with `Collectors.toList()` is used to gather results into a list.

Question 5

Which statement about stream operations is true?

a) Stream operations modify the original collection
b) Intermediate operations are executed immediately
c) Streams are reusable
d) Stream operations are lazy until a terminal operation is invoked

Answer: d)
Reasoning: Streams are lazily evaluated and only execute when a terminal operation is present.

Question 6

What is the result of this code?

```
java
CopyEdit
Stream.of(1, 2, 3, 4).filter(i -> i % 2 == 0).findFirst().get();
```

a) 2
b) 1
c) 4
d) Runtime error

Answer: a)
Reasoning: `filter()` keeps even numbers, `findFirst()` returns the first one: 2.

Question 7

Which collector can be used to convert a list of strings into a single string?

a) `Collectors.partitioningBy()`
b) `Collectors.joining()`
c) `Collectors.toMap()`
d) `Collectors.groupingBy()`

Answer: b)
Reasoning: `Collectors.joining()` concatenates strings in the stream into one final string.

Question 8

What is the output?

```
java
CopyEdit
Stream.of("Java", "Spring", "Hibernate")
      .filter(s -> s.length() > 5)
      .count();
```

a) 3
b) 0
c) 2
d) 1

Answer: c) 2
Reasoning: Only "Spring" and "Hibernate" have length > 5.

Question 9

Which collector creates a map grouping values by a classifier function?

a) `Collectors.partitioningBy()`
b) `Collectors.joining()`
c) `Collectors.groupingBy()`
d) `Collectors.toMap()`

Answer: c)
Reasoning: `groupingBy()` classifies elements and maps them into groups based on that classifier.

Question 10

Which is true about `reduce()`?

a) Always returns a `List`
b) Takes a BinaryOperator to combine elements
c) Used only for strings
d) Cannot be used on an empty stream

Answer: b)
Reasoning: `reduce()` accumulates stream elements using a provided BinaryOperator.

Question 11

What is the output of the following?

```
java
CopyEdit
Stream<String> stream = Stream.of("a", "b", "c");
stream.filter(s -> s.equals("b"));
stream.forEach(System.out::print);
```

a) abc
b) b
c) Compilation error
d) Runtime exception

Answer: a) abc
Reasoning: `filter()` is lazy and unused. It returns a new stream that isn't used, so `forEach` prints the original stream.

Question 12

Which operation is **short-circuiting**?

a) `map()`
b) `sorted()`
c) `limit()`
d) `peek()`

Answer: c) `limit()`
Reasoning: `limit()` can end the pipeline early, making it short-circuiting.

Question 13

Choose the correct difference between `map()` and `flatMap()`:

a) `map()` removes duplicates, `flatMap()` doesn't
b) `map()` returns a stream, `flatMap()` returns a list
c) `map()` transforms values; `flatMap()` transforms and flattens
d) They are the same

Answer: c)
Reasoning: `map()` transforms elements one-to-one; `flatMap()` maps elements to streams and flattens the result.

Question 14

What does the following code do?

```
java
CopyEdit
Stream.of("A", "B", "C").collect(Collectors.toSet());
```

a) Returns a list of strings
b) Returns a set of strings
c) Modifies the stream source
d) Throws exception

Answer: b)
Reasoning: `Collectors.toSet()` gathers elements into a `Set`.

Question 15

What is `Collectors.partitioningBy()` used for?

a) Sorting elements into a list
b) Grouping by key
c) Splitting elements into true/false groups
d) Counting elements

Answer: c)
Reasoning: `partitioningBy()` splits data into two groups based on a boolean predicate.

Question 16

How many times does `peek()` run in this code?

```
java
CopyEdit
Stream.of("x", "y", "z")
      .peek(System.out::println)
      .count();
```

a) 0
b) 1
c) 3
d) Depends on stream size

Answer: c)
Reasoning: `peek()` runs once per element if there is a terminal operation like `count()`.

Question 17

What does the stream pipeline return?

```
java
CopyEdit
List<String> result = Stream.of("Java", "JEE", "Spring")
    .filter(s -> s.startsWith("J"))
    .collect(Collectors.toList());
```

a) Compilation error
b) ["Java", "JEE"]
c) ["Java", "JEE", "Spring"]
d) []

Answer: b)
Reasoning: Only strings starting with "J" are included: "Java" and "JEE".

Question 18

Which stream operation will cause execution?

a) `map()`
b) `filter()`
c) `forEach()`
d) `peek()`

Answer: c)
Reasoning: `forEach()` is a terminal operation; it triggers execution of the pipeline.

Question 19

What happens if a stream is consumed twice?

```
java
CopyEdit
Stream<String> s = Stream.of("a", "b");
s.forEach(System.out::print);
s.forEach(System.out::print);
```

a) Prints "abab"
b) Prints nothing
c) Compiles and runs fine
d) Throws `IllegalStateException`

Answer: d)
Reasoning: A stream can only be consumed once. Reuse throws `IllegalStateException`.

Question 20

Which of these is **not true** about streams?

a) They can be parallel
b) They are lazily evaluated
c) They can mutate the source list
d) They support infinite sequences

Answer: c)
Reasoning: Streams are not supposed to mutate their source. They operate on a pipeline.

Question 21

What is the output?

```
java
CopyEdit
Stream.of(1, 2, 3)
```

```
    .map(x -> x * x)
    .reduce((a, b) -> a + b)
    .get();
```

a) 6
b) 14
c) 36
d) Compilation error

Answer: b) 14
Reasoning: Squares are [1, 4, 9]; sum is 14.

Question 22

Which is **true** about parallel streams?

a) Order of results is guaranteed
b) Only supported in Java 11+
c) Can use multiple CPU cores
d) More memory efficient

Answer: c)
Reasoning: Parallel streams enable concurrent processing across CPU cores.

Question 23

Which method returns an `Optional<T>`?

a) `findFirst()`
b) `filter()`
c) `map()`
d) `forEach()`

Answer: a)
Reasoning: `findFirst()` is a terminal operation that returns an `Optional<T>` with the first matching element.

Question 24

Which stream operation is used to compute a **summary statistic**?

a) `reduce()`
b) `summaryStatistics()`
c) `collect(Collectors.summarizingInt(...))`
d) `groupingBy()`

Answer: c)
Reasoning: `Collectors.summarizingInt()` collects min, max, average, sum, count.

Question 25

Choose the correct result type for:

```
java
CopyEdit
Stream<String> stream = Stream.of("a", "b", "c");
Map<Integer, List<String>> result =
stream.collect(Collectors.groupingBy(String::length));
```

a) Map<String, List<String>>
b) Map<Integer, Set<String>>
c) Map<Integer, List<String>>
d) Compilation error

Answer: c)
Reasoning: `groupingBy()` collects into a map with key as length (Integer), value as list of strings.

Java 8 MCQs – Topic: Date and Time API (java.time)

Subtopics:

- `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`
- `Period`, `Duration`, `Instant`
- Parsing and formatting dates
- Date arithmetic and immutability
- Time zone handling

Question 1

What does the following print?

```
java
CopyEdit
System.out.println(LocalDate.of(2020, Month.JANUARY, 1).plusDays(30));
```

a) 2020-01-30
b) 2020-02-01
c) 2020-01-31
d) Compilation error

Answer: b) 2020-02-01
Reasoning: Adding 30 days to Jan 1, 2020, lands on Feb 1.

Question 2

Which class represents a point in time with nanosecond precision?

a) `LocalDateTime`
b) `ZonedDateTime`
c) `Instant`
d) `Period`

Answer: c) `Instant`
Reasoning: `Instant` is a machine timestamp from the epoch with nanosecond resolution.

Question 3

What is true about `LocalDate`?

a) It stores time and zone
b) It is mutable
c) It is immutable and represents a date without time
d) It includes milliseconds

Answer: c)
Reasoning: `LocalDate` is immutable and stores date only (year, month, day).

Question 4

What does `Duration.between()` work with?

a) `LocalDate` only
b) `LocalDateTime` and `Instant`
c) `Period`
d) Any object

Answer: b)
Reasoning: `Duration` measures time between **two temporal objects**, like `Instant` or `LocalDateTime`.

Question 5

What will the following return?

```
java
CopyEdit
Period.between(LocalDate.of(2022, 1, 1), LocalDate.of(2022, 2, 15));
```

a) P1M14D
b) P45D
c) P2M15D
d) P15D

Answer: a)
Reasoning: `Period` breaks down duration in terms of years, months, and days—not total days.

Question 6

Which of these is **not** part of the java.time package?

a) `ZonedDateTime`
b) `DateTimeFormatter`
c) `GregorianCalendar`
d) `Instant`

Answer: c)
Reasoning: `GregorianCalendar` is from the older `java.util` time API.

Question 7

What does this code do?

```
java
CopyEdit
LocalDateTime dt = LocalDateTime.now();
dt.plusDays(5);
System.out.println(dt);
```

a) Adds 5 days to the date
b) Throws an exception
c) Returns the new date with 5 days added
d) Prints current date-time (unchanged)

Answer: d)
Reasoning: `LocalDateTime` is immutable. `plusDays()` returns a new object which is ignored here.

Question 8

Which formatter is used to format `LocalDate` in a custom way?

a) `SimpleDateFormat`
b) `DateFormat`
c) `DateTimeFormatter`
d) `PatternFormatter`

Answer: c)
Reasoning: `DateTimeFormatter` is the new formatting class in `java.time`.

Question 9

How to parse a `LocalDate` from a string?

```
java
CopyEdit
LocalDate.parse("2023-06-01")
```

a) Not allowed
b) Requires a formatter
c) Uses ISO_LOCAL_DATE format
d) Requires `SimpleDateFormat`

Answer: c)
Reasoning: `LocalDate.parse(...)` uses the default `ISO_LOCAL_DATE` format unless a formatter is provided.

Question 10

What will be the result of:

```
java
CopyEdit
ZonedDateTime.now(ZoneId.of("UTC"));
```

a) Compilation error
b) Current time in default timezone
c) Current time in UTC
d) Epoch timestamp

Answer: c)
Reasoning: It gives the current time in the **UTC** timezone.

Question 11

What does the following code output?

```
java
CopyEdit
LocalDate d1 = LocalDate.of(2022, 5, 10);
LocalDate d2 = d1.minusDays(5);
System.out.println(d2);
```

a) 2022-05-05
b) 2022-05-15
c) 2022-05-04
d) Compilation error

Answer: a)
Reasoning: Subtracting 5 days from May 10 results in May 5.

Question 12

Which method will throw a `DateTimeParseException`?

a) `LocalTime.parse("10:15")`
b) `LocalDate.parse("2022-13-01")`
c) `LocalDateTime.parse("2022-05-01T10:15:30")`
d) `ZonedDateTime.parse("2022-05-01T10:15:30+01:00[Europe/Paris]")`

Answer: b)
Reasoning: Month 13 is invalid. This will throw a `DateTimeParseException`.

Question 13

Which class is best for measuring the duration between two timestamps?

a) `Period`
b) `Instant`
c) `Duration`
d) `ZoneId`

Answer: c)
Reasoning: `Duration` measures time-based values (hours, minutes, seconds) between `Instant`s or `LocalDateTime`s.

Question 14

What does the following print?

```
java
CopyEdit
LocalTime t = LocalTime.of(23, 59, 59);
System.out.println(t.plusSeconds(1));
```

a) 00:00:00
b) 23:59:60
c) 00:00:01
d) 24:00:00

Answer: a)
Reasoning: One second after 23:59:59 is 00:00:00 (start of next day).

Question 15

What is true about `Period`?

a) It can include hours and minutes
b) It is used with `LocalDateTime`
c) It is used to represent a date-based amount of time
d) It is mutable

Answer: c)
Reasoning: `Period` represents a quantity of time in days/months/years—not time-of-day.

Question 16

Which method creates a `Period` of 2 years and 5 months?

a) `Period.of(2, 5, 0)`
b) `Period.between(2, 5, 0)`
c) `Duration.of(2, ChronoUnit.YEARS).plusMonths(5)`
d) `LocalDate.of(2, 5, 0)`

Answer: a)
Reasoning: `Period.of(years, months, days)` is the correct factory method.

Question 17

Which of these classes has a `from()` method to convert from another temporal object?

a) `LocalDate`
b) `ZonedDateTime`
c) `Instant`
d) All of the above

Answer: d)
Reasoning: Most classes in `java.time` support conversion from other temporal types using `from()`.

Question 18

Which `ZoneId` string is valid?

a) `"UTC+5"`
b) `"America/Los_Angeles"`
c) `"Europe/London/GMT"`
d) `"Asia-India"`

Answer: b)
Reasoning: Zone IDs follow a fixed structure like `Continent/City`, such as `America/Los_Angeles`.

Question 19

Which of the following best describes `Instant.now()`?

a) Returns current date
b) Returns current local time
c) Returns machine-readable UTC timestamp
d) Returns time in system zone

Answer: c)
Reasoning: `Instant.now()` gives a UTC-based timestamp useful for time-stamping logs, etc.

Question 20

Which formatter pattern will correctly format a `LocalDateTime` as "2025-06-12 14:30"?

a) `"yyyy/MM/dd HH:mm"`
b) `"dd-MM-yyyy hh:mm"`
c) `"yyyy-MM-dd HH:mm"`
d) `"MM-dd-yyyy hh:mm:ss"`

Answer: c)
Reasoning: Correct Java time format pattern for the required output is `"yyyy-MM-dd HH:mm"`.

Question 21

What will the following output?

```
java
CopyEdit
LocalDateTime dt = LocalDateTime.of(2022, 12, 31, 23, 59);
dt = dt.plusMinutes(2);
System.out.println(dt);
```

a) 2023-01-01T00:01
b) 2022-12-31T00:01
c) 2022-12-31T23:01
d) 2023-01-01T01:01

Answer: a)
Reasoning: Adding 2 minutes to 23:59 on Dec 31 rolls over to Jan 1 at 00:01.

Question 22

What does `ChronoUnit.DAYS.between(d1, d2)` return?

a) A `Period`
b) A `Duration`
c) A `long`
d) A `String`

Answer: c)
Reasoning: `ChronoUnit.DAYS.between(...)` returns the difference in days as a long.

Question 23

Which of the following is true?

a) `Period` can be used with `LocalTime`
b) `Duration` can measure weeks
c) `Instant` can be converted to `ZonedDateTime`
d) `LocalDateTime` includes time zone

Answer: c)
Reasoning: You can convert `Instant` to `ZonedDateTime` using a time zone.

Question 24

Which method adjusts a date to the **last day of the month**?

a) `withLastDayOfMonth()`
b) `adjustToLastDay()`
c) `with(TemporalAdjusters.lastDayOfMonth())`
d) `lastDayOfMonth()`

Answer: c)
Reasoning: Use `TemporalAdjusters.lastDayOfMonth()` to shift to the end of the month.

Question 25

What is the result of this code?

```
java
CopyEdit
LocalDate.now().plusYears(1).minusMonths(2).getDayOfWeek();
```

a) Returns current day
b) Returns the day of week one year ahead and 2 months back

c) Throws exception
d) Always returns MONDAY

Answer: b)
Reasoning: It calculates a new date and returns the `DayOfWeek` for it.

Java 8 MCQs – Topic: Concurrency & ForkJoin Framework

Subtopics:

- `java.util.concurrent` interfaces
- `Runnable`, `Callable`, `Future`
- Thread-safety and synchronization
- ForkJoinPool and `RecursiveTask`
- Parallel streams

Question 1

Which of the following can return a result or throw an exception?

a) `Runnable`
b) `Thread`
c) `Callable<V>`
d) `FutureTask`

Answer: c)
Reasoning: `Callable` is designed to return a value and throw checked exceptions.

Question 2

Which class is used to schedule tasks to run after a delay or periodically?

a) `ThreadPoolExecutor`
b) `ScheduledExecutorService`
c) `ForkJoinPool`
d) `Timer`

Answer: b)
Reasoning: `ScheduledExecutorService` supports delay-based and periodic task scheduling.

Question 3

Which interface represents the result of an asynchronous computation?

a) `Runnable`
b) `Callable`
c) `Future`
d) `Thread`

Answer: c)
Reasoning: `Future<V>` represents the result of a computation that may complete later.

Question 4

Which method blocks until the result is available?

a) `get()` on `Future`
b) `run()` on `Runnable`
c) `invoke()` on `ExecutorService`
d) `execute()` on `ForkJoinTask`

Answer: a)
Reasoning: `future.get()` blocks until the result is available or an exception occurs.

Question 5

What is the default parallelism level of a common `ForkJoinPool`?

a) Number of processors × 2
b) Number of threads in the JVM
c) Number of available processors
d) 1

Answer: c)
Reasoning: `ForkJoinPool.commonPool()` uses `Runtime.getRuntime().availableProcessors()`.

Question 6

Which method in `RecursiveTask` must be overridden?

a) `execute()`
b) `compute()`
c) `run()`
d) `invoke()`

Answer: b)
Reasoning: `compute()` is the core method to define a task in Fork/Join.

Question 7

What is the role of `join()` in Fork/Join?

a) Starts a thread
b) Waits for a thread to finish
c) Blocks until the subtask completes and returns result
d) Suspends a thread indefinitely

Answer: c)
Reasoning: `join()` is used in `ForkJoinTask` to block until result is ready.

Question 8

Which is a valid way to submit a task to `ExecutorService`?

a) `submit(new Thread())`
b) `execute(new Callable())`
c) `submit(new Runnable())`
d) `invokeAll(new Future())`

Answer: c)
Reasoning: `submit()` accepts `Runnable` or `Callable`.

Question 9

How do you create a thread-safe map?

a) `new TreeMap()`
b) `Collections.synchronizedMap(new HashMap<>())`
c) `HashMap.putSync()`
d) `ConcurrentHashSet`

Answer: b)
Reasoning: Wrapping `HashMap` with `Collections.synchronizedMap` provides thread safety.

Question 10

Which stream runs in parallel?

```
java
CopyEdit
list.stream()
list.parallelStream()
```

a) Both
b) Only `parallelStream()`

c) Only `stream()`
d) Neither

Answer: b)
Reasoning: `parallelStream()` creates a stream that can process in parallel using `ForkJoinPool`.

Question 11

What does `invokeAll()` method of `ExecutorService` return?

a) A list of threads
b) A list of callables
c) A list of `Future` objects
d) A list of results

Answer: c)
Reasoning: `invokeAll()` takes a collection of `Callable` tasks and returns a `List<Future<T>>`.

Question 12

Which method is used to submit a `Callable` to `ExecutorService`?

a) `invoke()`
b) `run()`
c) `submit()`
d) `execute()`

Answer: c)
Reasoning: `submit()` allows submission of a `Callable` and returns a `Future`.

Question 13

Which of the following causes a thread to wait for another thread to finish?

a) `Thread.sleep()`
b) `Thread.run()`
c) `Thread.join()`
d) `Thread.interrupt()`

Answer: c)
Reasoning: `join()` blocks the current thread until the target thread completes.

Question 14

What happens if `ForkJoinTask.compute()` doesn't invoke `fork()`?

a) The task executes asynchronously
b) The task is skipped
c) The task runs on the same thread
d) Compilation error

Answer: c)
Reasoning: If `fork()` is not used, no new task is created. It behaves like a regular method call.

Question 15

Which of the following is **not** thread-safe?

a) `ConcurrentHashMap`
b) `StringBuffer`
c) `ArrayList`
d) `Vector`

Answer: c)
Reasoning: `ArrayList` is not synchronized and is not safe in multithreaded environments.

Question 16

What does `invoke()` do in ForkJoinPool?

a) Blocks until the task is complete
b) Starts a new thread
c) Submits a task asynchronously
d) None of the above

Answer: a)
Reasoning: `invoke()` blocks until the task completes and returns the result.

Question 17

Which of the following classes implement `Executor` interface?

a) `ForkJoinPool`
b) `Thread`
c) `Timer`
d) `Callable`

Answer: a)
Reasoning: `ForkJoinPool` is a subclass of `AbstractExecutorService` which implements `Executor`.

Question 18

What is a common use of `volatile` keyword?

a) Prevent thread switching
b) Enable synchronization
c) Prevent instruction reordering
d) Make method atomic

Answer: c)
Reasoning: `volatile` ensures visibility and prevents instruction reordering for that variable.

Question 19

Which method will shut down an `ExecutorService` gracefully?

a) `terminate()`
b) `stop()`
c) `shutdown()`
d) `close()`

Answer: c)
Reasoning: `shutdown()` initiates an orderly shutdown by rejecting new tasks but processing existing ones.

Question 20

Which concurrency utility is used for **phased** thread synchronization?

a) `CountDownLatch`
b) `Semaphore`
c) `CyclicBarrier`
d) `Phaser`

Answer: d)
Reasoning: `Phaser` allows flexible phase-based coordination between threads.

Question 21

When using `parallelStream()`, which framework does it use underneath?

a) Thread class
b) ScheduledExecutorService
c) ForkJoinPool.commonPool()
d) java.util.Timer

Answer: c)
Reasoning: `parallelStream()` uses `ForkJoinPool.commonPool()` internally for task execution.

Question 22

Which of the following is **not** a feature of `ConcurrentHashMap`?

a) Segment-based locking
b) Allows null keys
c) Thread-safe updates
d) Better performance than `Hashtable`

Answer: b)
Reasoning: `ConcurrentHashMap` does **not allow null keys or values**.

Question 23

Which class is used to coordinate a one-time event across threads?

a) `Phaser`
b) `CyclicBarrier`
c) `CountDownLatch`
d) `ReentrantLock`

Answer: c)
Reasoning: `CountDownLatch` is used for a one-shot signaling event among threads.

Question 24

In a `ForkJoinTask`, calling `fork()` does what?

a) Executes immediately
b) Waits for result
c) Queues the task in ForkJoinPool
d) Forks thread in OS

Answer: c)
Reasoning: `fork()` submits the task to the work queue in the pool for asynchronous execution.

Question 25

What is the purpose of the `compute()` method in `RecursiveTask`?

a) Fork a new thread
b) Override to define the actual task
c) Blocks the thread
d) Returns a Future

Answer: b)
Reasoning: `compute()` must be overridden to define the logic of the recursive task.

Java 8 MCQs – Topic: File I/O and NIO.2

Subtopics:

- `java.nio.file.Path` and `Paths`
- `Files` operations (`exists()`, `copy()`, `walk()`, `newBufferedReader()`, etc.)
- `DirectoryStream`, `BufferedReader/Writer`
- Symbolic links, attributes
- File traversal, I/O exceptions

Question 1

What is the correct way to obtain a `Path` object?

a) `new Path("file.txt")`
b) `Path.get("file.txt")`
c) `Paths.get("file.txt")`
d) `FileSystems.path("file.txt")`

Answer: c)
Reasoning: Use `Paths.get(...)` to get a `Path` instance. `Path` has no public constructor.

Question 2

What does the `Files.exists(path)` method return?

a) `true` if path exists and is readable
b) `true` if file exists and is a directory
c) `true` if the file or directory exists
d) Throws IOException

Answer: c)
Reasoning: `Files.exists()` returns true if the file/directory exists at the path.

Question 3

Which method reads all lines from a file into a `List<String>`?

a) `Files.readAll(path)`
b) `Files.readLines(path)`
c) `Files.readAllLines(path)`
d) `Files.read(path).toList()`

Answer: c)
Reasoning: `Files.readAllLines(Path)` returns a list of all lines in the file.

Question 4

What does `Files.copy()` return?

a) Number of bytes copied
b) New Path
c) void
d) Boolean

Answer: b)
Reasoning: The method `Files.copy(Path, Path)` returns the path to the target file.

Question 5

Which method is used to delete a file if it exists?

a) `Files.deleteIfExists(path)`
b) `Files.delete(path)`
c) `Files.remove(path)`
d) `Files.removeIfExists(path)`

Answer: a)
Reasoning: `Files.deleteIfExists(Path)` deletes the file or directory and returns `true` if it existed.

Question 6

Which method can create a new file only if it does not exist?

a) `Files.createFile(path)`
b) `Files.touch(path)`
c) `Files.newBufferedWriter(path)`
d) `Files.createOrUpdate(path)`

Answer: a)
Reasoning: `Files.createFile()` throws `FileAlreadyExistsException` if the file exists.

Question 7

Which of the following is `true` about `Path.resolve()`?

a) It creates a symbolic link
b) It converts path to absolute
c) It appends one path to another
d) It returns a URI

Answer: c)
Reasoning: `resolve()` appends the given path to the current path unless it's absolute.

Question 8

What does `Files.isDirectory(path)` do?

a) Checks if the path points to a file
b) Checks if the path is a symbolic link
c) Returns true if path is a directory
d) Converts file to directory

Answer: c)
Reasoning: It checks if the file at path is a directory.

Question 9

How do you walk through a directory recursively?

a) `DirectoryStream`
b) `FileStream.walk()`
c) `Files.walk(path)`
d) `FileVisitor.walk()`

Answer: c)
Reasoning: `Files.walk(Path)` returns a `Stream<Path>` of files/subdirectories recursively.

Question 10

Which exception is thrown by most `Files` methods?

a) `FileNotFoundException`
b) `IOException`
c) `RuntimeException`
d) `NullPointerException`

Answer: b)
Reasoning: All I/O operations in NIO.2 throw `IOException` on error.

Question 11

Which method writes a list of strings to a file?

a) `Files.writeString()`
b) `Files.write(Path, List<String>)`
c) `Files.output(Path)`
d) `Files.appendLines(Path, List<String>)`

Answer: b)
Reasoning: `Files.write(Path, Iterable<? extends CharSequence>)` writes lines to a file.

Question 12

Which of the following creates a buffered writer to a file?

a) `Files.newWriter()`
b) `BufferedWriter.write()`
c) `Files.newBufferedWriter(path)`
d) `Files.openBufferedWriter(path)`

Answer: c)
Reasoning: `Files.newBufferedWriter(Path)` provides efficient character stream writing.

Question 13

What will this code do?

```
java
CopyEdit
Path p = Paths.get("test.txt");
Files.createFile(p);
Files.createFile(p);
```

a) Creates two files
b) Overwrites the file
c) Throws `FileAlreadyExistsException`
d) Appends data to file

Answer: c)
Reasoning: `Files.createFile()` throws exception if the file already exists.

Question 14

How can you get file attributes like creation or modified time?

a) `Files.getMetadata()`
b) `Files.readAttributes()`
c) `Path.getAttributes()`
d) `Files.attributesOf()`

Answer: b)
Reasoning: `Files.readAttributes(Path, BasicFileAttributes.class)` provides file metadata.

Question 15

Which of the following statements is true about `Files.copy()`?

a) It always overwrites
b) It throws exception if target exists, unless options specify overwrite
c) It deletes the source file
d) It requires both files to exist

Answer: b)
Reasoning: To overwrite, pass `StandardCopyOption.REPLACE_EXISTING`.

Question 16

Which is used to read a large text file line-by-line efficiently?

a) `FileInputStream.read()`
b) `Scanner.nextLine()`
c) `BufferedReader.readLine()`
d) `Files.readAllLines()`

Answer: c)
Reasoning: `BufferedReader.readLine()` is memory-efficient for large files.

Question 17

Which API allows you to iterate through a directory's contents without recursion?

a) `Files.list()`
b) `DirectoryStream`

c) `Files.walk()`
d) `Stream<Path>`

Answer: b)
Reasoning: `DirectoryStream<Path>` is used for non-recursive directory listing.

Question 18

What happens when you call `Files.move(source, target)` if target exists?

a) Overwrites silently
b) Throws exception unless `REPLACE_EXISTING` is used
c) Always throws exception
d) Merges contents

Answer: b)
Reasoning: Use `StandardCopyOption.REPLACE_EXISTING` to allow overwrite.

Question 19

What does this code return?

```
java
CopyEdit
Files.isSymbolicLink(Paths.get("test.lnk"));
```

a) true if it's a soft link
b) true if file exists
c) false always
d) Compilation error

Answer: a)
Reasoning: This method checks if the path is a symbolic (soft) link.

Question 20

Which class is used to handle exceptions during file walking?

a) `IOException`
b) `DirectoryWalker`
c) `FileVisitor`
d) `StreamExceptionHandler`

Answer: c)
Reasoning: `FileVisitor` interface lets you define logic on visiting files and handling exceptions.

Question 21

Which method converts a `Path` to a URI?

a) `path.uri()`
b) `path.toURL()`
c) `path.toURI()`
d) `path.asURI()`

Answer: c)
Reasoning: `toURI()` is the standard method to convert `Path` to `URI`.

Question 22

Which of these operations is most efficient for walking file trees?

a) `Files.walk(path)`
b) `Files.list(path)`
c) `Files.newDirectoryStream(path)`
d) `Files.readAllLines(path)`

Answer: a)
Reasoning: `Files.walk()` supports recursive traversal and streaming of paths.

Question 23

Which method should you use to create a directory?

a) `new File("dir").mkdir()`
b) `Files.createDirectory(Path)`
c) `File.mkdirs()`
d) `Path.create()`

Answer: b)
Reasoning: `Files.createDirectory()` is the NIO.2 way to create a new directory.

Question 24

Which copy option would be required to copy file attributes?

a) `REPLACE_EXISTING`
b) `NOFOLLOW_LINKS`
c) `COPY_ATTRIBUTES`
d) `COPY_FILE_ONLY`

Answer: c)
Reasoning: `StandardCopyOption.COPY_ATTRIBUTES` copies file attributes like timestamps and permissions.

Question 25

Which of the following causes `Files.walk()` to throw an exception?

a) File not found
b) Path is a symbolic link
c) Folder has no children
d) Path is empty

Answer: a)
Reasoning: If the root path does not exist, `Files.walk()` throws `IOException`.

Java 8 MCQs – Topic: JDBC & Transactions

Subtopics:

- `Connection`, `Statement`, `PreparedStatement`, `ResultSet`
- SQL execution: `execute()`, `executeQuery()`, `executeUpdate()`
- Auto-commit, manual transactions
- Try-with-resources in JDBC
- Batch updates, rollback

Question 1

What does `Connection.prepareStatement(String sql)` return?

a) `ResultSet`
b) `Statement`
c) `PreparedStatement`
d) `QueryExecutor`

Answer: c)
Reasoning: It returns a `PreparedStatement` that can be used to execute parameterized SQL queries.

Question 2

Which method is used to execute an SQL `SELECT` query?

a) `executeQuery()`
b) `executeUpdate()`
c) `executeSelect()`
d) `runQuery()`

Answer: a)
Reasoning: `executeQuery()` returns a `ResultSet` from a `SELECT` statement.

Question 3

Which interface is used to retrieve query results?

a) `Statement`
b) `ResultSet`
c) `PreparedStatement`
d) `QueryOutput`

Answer: b)
Reasoning: `ResultSet` is used to navigate and read query results row by row.

Question 4

Which JDBC object is used to run parameterized SQL queries?

a) `Statement`
b) `ResultSet`
c) `CallableStatement`
d) `PreparedStatement`

Answer: d)
Reasoning: `PreparedStatement` lets you bind parameters using `?` placeholders.

Question 5

Which method can be used to commit a transaction?

a) `commit()`
b) `save()`
c) `executeCommit()`
d) `commitTransaction()`

Answer: a)
Reasoning: `Connection.commit()` is used to commit the current transaction.

Question 6

What is the default behavior of a new `Connection` regarding transactions?

a) Transactions must be started explicitly
b) `autoCommit = false`
c) Each SQL statement is committed automatically
d) Transactions are unsupported by default

Answer: c)
Reasoning: JDBC connections start with `autoCommit = true`.

Question 7

Which method disables auto-commit?

a) `disableAutoCommit()`
b) `setAutoCommit(false)`
c) `autoCommit(false)`
d) `beginTransaction()`

Answer: b)
Reasoning: `Connection.setAutoCommit(false)` disables automatic commit.

Question 8

Which JDBC interface supports stored procedure execution?

a) `PreparedStatement`
b) `Statement`
c) `CallableStatement`
d) `ProcedureExecutor`

Answer: c)
Reasoning: `CallableStatement` is used for calling database stored procedures.

Question 9

How can resources be closed automatically in JDBC?

a) Use `finally` block
b) Use `System.gc()`
c) Use try-with-resources
d) Use `Statement.destroy()`

Answer: c)
Reasoning: Try-with-resources ensures automatic resource closing (`Connection`, `Statement`, `ResultSet`).

Question 10

What is returned by `executeUpdate("INSERT INTO ...")`?

a) A `ResultSet`
b) A `boolean`
c) Number of rows affected
d) Always 1

Answer: c)
Reasoning: `executeUpdate()` returns the count of affected rows.

Question 11

What is the correct order for JDBC operations?

a) Connect → Create Statement → Execute → Close
b) Connect → Execute → Create Statement → Close
c) Create Statement → Connect → Execute → Close
d) Execute → Connect → Create Statement → Close

Answer: a)
Reasoning: First connect to DB, create statement, execute query, and finally close resources.

Question 12

What happens if you don't close a `ResultSet`?

a) It is garbage collected immediately
b) It causes memory leaks or DB connection exhaustion
c) It automatically resets
d) Nothing

Answer: b)
Reasoning: Unclosed `ResultSet` can hold DB cursors and resources, leading to performance issues.

Question 13

Which of these allows positional parameters using `?`?

a) `Statement`
b) `PreparedStatement`
c) `CallableStatement`
d) Both b and c

Answer: d)
Reasoning: Both `PreparedStatement` and `CallableStatement` support positional parameters.

Question 14

Which method is used to check for more rows in `ResultSet`?

a) `ResultSet.hasNext()`
b) `ResultSet.next()`
c) `ResultSet.more()`
d) `ResultSet.read()`

Answer: b)
Reasoning: `ResultSet.next()` moves cursor forward and returns `true` if another row exists.

Question 15

Which method retrieves a string from the second column?

```
java
CopyEdit
ResultSet rs = stmt.executeQuery("SELECT name, age FROM users");
```

a) `rs.get(2)`
b) `rs.getString("age")`
c) `rs.getString(2)`
d) `rs.getInt(2)`

Answer: c)
Reasoning: Columns can be retrieved by index starting at 1. `getString(2)` returns the value as a string.

Question 16

What will happen if `commit()` is called while `autoCommit` is `true`?

a) Commits the transaction
b) Throws exception
c) Does nothing
d) Commits twice

Answer: c)
Reasoning: If `autoCommit = true`, every SQL is committed automatically, and `commit()` does nothing.

Question 17

Which interface is returned by `DriverManager.getConnection()`?

a) `Driver`
b) `DBManager`
c) `Connection`
d) `Statement`

Answer: c)
Reasoning: `DriverManager.getConnection(...)` establishes and returns a `Connection`.

Question 18

Which JDBC method supports execution of **any** SQL statement?

a) `executeQuery()`
b) `executeUpdate()`
c) `execute()`
d) `runSQL()`

Answer: c)
Reasoning: `execute()` handles DDL, DML, or DQL (returns true if `ResultSet` is returned).

Question 19

Which method allows a batch of SQL updates?

a) `addBatch()`
b) `executeBatch()`
c) Both a and b
d) `prepareBatch()`

Answer: c)
Reasoning: Add multiple SQLs using `addBatch()` and execute them with `executeBatch()`.

Question 20

If `rollback()` is called, what happens?

a) Previous commits are reversed
b) Statements since last commit are undone
c) All statements are undone
d) All data is deleted

Answer: b)
Reasoning: `rollback()` undoes changes since the last successful commit point.

Question 21

How do you ensure proper resource cleanup in JDBC?

a) Use `finally` block
b) Use `try-with-resources`
c) Use `System.exit(0)`
d) Use `catch` block

Answer: b)
Reasoning: `try-with-resources` is preferred as it ensures proper `AutoCloseable` cleanup.

Question 22

Which of the following **is not** a valid JDBC type?

a) `DOUBLE`
b) `TEXT`
c) `VARCHAR`
d) `BOOLEAN`

Answer: b)
Reasoning: `TEXT` is not a JDBC standard type; databases like SQLite use it internally.

Question 23

If `Connection.close()` is called, what happens to active statements?

a) They remain active
b) They are closed automatically
c) They throw a warning
d) They block indefinitely

Answer: b)
Reasoning: Closing a `Connection` automatically closes associated statements and result sets.

Question 24

What is the benefit of using `PreparedStatement`?

a) Code readability
b) Query caching and prevention of SQL injection

c) Supports ORM
d) Runs in a separate thread

Answer: b)
Reasoning: `PreparedStatement` precompiles and helps prevent SQL injection via bound parameters.

Question 25

Which method sets a `String` parameter on a `PreparedStatement`?

```
java
CopyEdit
PreparedStatement ps = conn.prepareStatement("INSERT INTO users(name) VALUES
(?)");
```

a) `ps.putString(1, "Alice")`
b) `ps.setText(1, "Alice")`
c) `ps.setString(1, "Alice")`
d) `ps.writeString(1, "Alice")`

Answer: c)
Reasoning: `setString(index, value)` is the correct method for setting a string parameter.

Java 8 MCQs – Topic: Localization

Subtopics:

- `Locale`, `ResourceBundle`, `PropertyResourceBundle`
- `Locale.getDefault()`, `Locale.Builder`
- Localization file naming (`_en_US`, etc.)
- Message formatting and fallbacks
- ResourceBundle loading behavior

Question 1

Which class is used to represent a specific geographical, political, or cultural region?

a) `Region`
b) `Culture`
c) `Locale`
d) `Locality`

Answer: c)
Reasoning: `java.util.Locale` represents a specific locale for formatting or resource lookup.

Question 2

Which of the following creates a US English locale?

a) `new Locale("US", "EN")`
b) `new Locale("en", "US")`
c) `new Locale("English", "UnitedStates")`
d) `Locale.create("en_US")`

Answer: b)
Reasoning: The constructor uses language as first param and country as second: `("en", "US")`.

Question 3

Which method is used to get the default locale?

a) `Locale.get()`
b) `Locale.getSystem()`
c) `Locale.getDefault()`
d) `Locale.systemLocale()`

Answer: c)
Reasoning: `Locale.getDefault()` returns the JVM's current default locale.

Question 4

Which class is used to manage localized resources?

a) `ResourceHandler`
b) `LocaleManager`
c) `ResourceBundle`
d) `PropertiesManager`

Answer: c)
Reasoning: `ResourceBundle` provides locale-specific objects like messages or labels.

Question 5

What is the correct base name for a resource bundle file?

a) `MessagesBundle.locale`
b) `Messages.en_US.properties`
c) `MessagesBundle_en_US.properties`
d) `Messages_Bundle.properties`

Answer: c)
Reasoning: Bundle file follows `BaseName_language_COUNTRY.properties`.

Question 6

Which method retrieves a localized string from a resource bundle?

a) `bundle.read()`
b) `bundle.getValue()`
c) `bundle.getString("key")`
d) `bundle.load("key")`

Answer: c)
Reasoning: `getString()` is used to fetch the value for a key from a `ResourceBundle`.

Question 7

Which method loads the correct resource bundle for a given locale?

a) `ResourceBundle.load()`
b) `ResourceBundle.getBundle()`
c) `Locale.getBundle()`
d) `Locale.loadBundle()`

Answer: b)
Reasoning: `getBundle()` finds the correct `ResourceBundle` for a `Locale`.

Question 8

Which is true about fallback behavior of `ResourceBundle`?

a) It throws an error if locale-specific file is not found
b) It falls back to the default locale
c) It searches for the most specific match and falls back to base
d) It skips all unknown locales

Answer: c)
Reasoning: It attempts `Base_lang_COUNTRY`, `Base_lang`, then `Base`.

Question 9

What is the type of a `.properties` file-based bundle?

a) `PropertyFileBundle`
b) `Properties`
c) `PropertyResourceBundle`
d) `Bundle`

Answer: c)
Reasoning: `PropertyResourceBundle` is used when backing `.properties` file.

Question 10

What happens if a key is missing from the bundle?

a) It returns null
b) It throws a `MissingResourceException`
c) It uses default value
d) It skips the key

Answer: b)
Reasoning: If a key does not exist, `MissingResourceException` is thrown.

Question 11

What would `new Locale.Builder().setLanguage("en").setRegion("GB").build()` produce?

a) Invalid locale
b) Locale for US English
c) Locale for Great Britain English
d) Default JVM locale

Answer: c)
Reasoning: Builder creates `Locale` for `"en-GB"` (English, Great Britain).

Question 12

If you have the following bundle files, which one is chosen for locale `fr_CA`?

- `Messages.properties`
- `Messages_fr.properties`
- `Messages_fr_CA.properties`

a) `Messages.properties`
b) `Messages_fr.properties`

c) `Messages_fr_CA.properties`
d) All three at once

> **Answer:** c)
> **Reasoning:** Java first tries most specific, then falls back.

Question 13

Which locale constant represents US English?

a) `Locale.US`
b) `Locale.ENGLISH_US`
c) `Locale.UK`
d) `Locale.US_EN`

> **Answer:** a)
> **Reasoning:** `Locale.US` is predefined as English (United States).

Question 14

What is the behavior of `Locale.getISOCountries()`?

a) Returns all supported locales
b) Returns two-letter country codes
c) Returns country names
d) Returns language scripts

> **Answer:** b)
> **Reasoning:** It returns all two-letter ISO 3166 country codes.

Question 15

How do you specify a variant in `Locale`?

a) With `new Locale("en", "US", "variant")`
b) Using `Locale.setVariant()`
c) Through `Locale.withVariant()`
d) You cannot specify variants

> **Answer:** a)
> **Reasoning:** Locale has a 3-argument constructor: language, country, variant.

Question 16

What is the result of:

```
java
```

```
CopyEdit
Locale locale = new Locale("fr", "CA");
System.out.println(locale.getDisplayCountry());
```

a) `CA`
b) `Canada`
c) `fr_CA`
d) `French`

Answer: b)
Reasoning: `getDisplayCountry()` gives human-readable name like "Canada".

Question 17

What type of resource bundle file should you use for localized text?

a) XML
b) `.bundle`
c) `.properties`
d) `.loc`

Answer: c)
Reasoning: `.properties` is the standard format for `ResourceBundle`.

Question 18

Which method gets the language code from a `Locale` object?

a) `getLang()`
b) `getLanguage()`
c) `getCode()`
d) `getLocaleLanguage()`

Answer: b)
Reasoning: `getLanguage()` returns ISO 639 language code like `en`, `fr`.

Question 19

What does this return?

```
java
CopyEdit
Locale loc = new Locale("de", "DE");
System.out.println(loc.toString());
```

a) `de`
b) `de_DE`

c) `DE_de`
d) `deDE`

Answer: b)
Reasoning: `Locale.toString()` returns `language_COUNTRY`.

Question 20

What happens if a resource key is missing and no fallback exists?

a) Returns default key
b) Throws `NullPointerException`
c) Throws `MissingResourceException`
d) Skips the key

Answer: c)
Reasoning: If key is missing and not overridden, exception is thrown.

Question 21

How are `.properties` files loaded?

a) As serialized objects
b) As XML parsers
c) As key-value pairs using ISO-8859-1 encoding
d) Using JSON format

Answer: c)
Reasoning: `.properties` files use `ISO-8859-1`, keys/values are string literals.

Question 22

Which class supports locale-sensitive message formatting?

a) `MessageBuilder`
b) `LocaleFormatter`
c) `MessageFormat`
d) `StringFormatter`

Answer: c)
Reasoning: `java.text.MessageFormat` formats messages with localization support.

Question 23

Which method retrieves all keys in a `ResourceBundle`?

a) `bundle.getAllKeys()`
b) `bundle.keys()`
c) `bundle.keySet()`
d) `bundle.getKeys()`

Answer: d)
Reasoning: `getKeys()` returns `Enumeration<String>` of all keys.

Question 24

If a bundle file is missing but base file exists, what happens?

a) Exception is thrown
b) Default values are used
c) Base bundle is loaded
d) Fallback is skipped

Answer: c)
Reasoning: Java uses base bundle when specific localization is unavailable.

Question 25

Which of the following is **true** about `Locale`?

a) Locale affects JVM memory allocation
b) Locale must match system timezone
c) Locale influences language/region-sensitive APIs
d) Locale affects garbage collection

Answer: c)
Reasoning: Locale affects date, number, and message formatting APIs.

Java 8 MCQs – Topic: Annotations & Reflection

Subtopics:

- `@Override`, `@Deprecated`, `@FunctionalInterface`
- Meta-annotations: `@Target`, `@Retention`, `@Inherited`
- `RetentionPolicy`, `ElementType`
- Reflection API (`Class`, `Method`, `Field`, `getAnnotations`)
- `AnnotatedElement`, runtime type inspection

Question 1

What does the `@Override` annotation indicate?

a) The method hides a superclass method
b) The method overrides an interface method
c) The method overrides a superclass method
d) The method is overloaded

Answer: c)
Reasoning: `@Override` confirms that the method overrides a superclass method.

Question 2

Which meta-annotation defines when an annotation is available?

a) `@Target`
b) `@Documented`
c) `@Retention`
d) `@Inherited`

Answer: c)
Reasoning: `@Retention` defines how long annotations are retained (SOURCE, CLASS, RUNTIME).

Question 3

What does `@FunctionalInterface` enforce?

a) The class has one method
b) The interface contains only abstract methods
c) Only one abstract method is allowed
d) It can’t contain any default methods

Answer: c)
Reasoning: It enforces that exactly one abstract method is declared.

Question 4

Which retention policy makes annotations available at runtime?

a) `RetentionPolicy.SOURCE`
b) `RetentionPolicy.CLASS`
c) `RetentionPolicy.RUNTIME`
d) `RetentionPolicy.DEFAULT`

Answer: c)
Reasoning: Only `RUNTIME` retention allows reflection access.

Question 5

What does `@Target(ElementType.METHOD)` mean?

a) Annotation can be used in all methods
b) Annotation can only be applied to methods
c) Annotation can be used on all elements
d) Annotation is required on methods

Answer: b)
Reasoning: It restricts usage to method declarations only.

Question 6

Which reflection class provides access to annotations?

a) `AnnotationHandler`
b) `ClassInspector`
c) `AnnotatedElement`
d) `AnnotationFactory`

Answer: c)
Reasoning: `Class`, `Method`, `Field` all implement `AnnotatedElement`.

Question 7

What method retrieves a single annotation instance?

a) `getAnnotation()`
b) `getDeclaredAnnotation()`
c) `readAnnotation()`
d) `getAnnotationInstance()`

Answer: a)
Reasoning: `getAnnotation(Class<T>)` returns annotation if present, else null.

Question 8

What is the default retention policy if not specified?

a) `SOURCE`
b) `CLASS`
c) `RUNTIME`
d) `NONE`

Answer: b)
Reasoning: If `@Retention` is not specified, default is `CLASS`.

Question 9

Which of these is not a valid `ElementType`?

a) `TYPE`
b) `FIELD`
c) `LOOP`
d) `METHOD`

Answer: c)
Reasoning: `LOOP` is not a valid element target in annotations.

Question 10

Which reflection method retrieves all declared methods of a class?

a) `getAllMethods()`
b) `getDeclaredMethods()`
c) `getMethods()`
d) `getAllClassMethods()`

Answer: b)
Reasoning: `getDeclaredMethods()` includes private and inherited methods.

Question 11

What does `clazz.getMethods()` return?

a) Only private methods
b) All declared methods, including inherited public ones
c) Only final methods
d) All methods including annotations

Answer: b)
Reasoning: `getMethods()` returns public methods declared in the class and its superclasses/interfaces.

Question 12

What does the `@Inherited` annotation do?

a) Makes annotations runtime accessible
b) Allows annotations to be inherited by subclasses

c) Applies annotation to all classes
d) Prevents annotation from being used in interfaces

Answer: b)
Reasoning: `@Inherited` makes an annotation automatically inherited by subclasses.

Question 13

Which annotation is used to indicate a method is obsolete?

a) `@Ignore`
b) `@Deprecated`
c) `@Removed`
d) `@Obsolete`

Answer: b)
Reasoning: `@Deprecated` marks methods as discouraged or obsolete.

Question 14

What does this code output?

```
java
CopyEdit
Class<?> clazz = MyClass.class;
Annotation[] anns = clazz.getAnnotations();
System.out.println(anns.length);
```

Assuming no annotations are present.

a) 0
b) 1
c) Compilation error
d) NullPointerException

Answer: a)
Reasoning: `getAnnotations()` returns an empty array if none are found.

Question 15

Which method retrieves annotations declared directly in the class?

a) `getAnnotation()`
b) `getDeclaredAnnotation()`
c) `getDeclaredAnnotations()`
d) Both b and c

Answer: d)
Reasoning: Both `getDeclaredAnnotation()` and `getDeclaredAnnotations()` access annotations present directly on the class.

Question 16

How do you check if a class has a specific annotation?

a) `isAnnotationPresent()`
b) `hasAnnotation()`
c) `annotationExists()`
d) `existsAnnotation()`

Answer: a)
Reasoning: `isAnnotationPresent(Class<? extends Annotation>)` returns true if annotation is present.

Question 17

Which annotation is used to mark an annotation as applicable to types and methods?

a) `@Target(TYPE, METHOD)`
b) `@ElementType(TYPE, METHOD)`
c) `@Target({ElementType.TYPE, ElementType.METHOD})`
d) `@Retention(TYPE, METHOD)`

Answer: c)
Reasoning: `@Target` accepts an array of `ElementType` values.

Question 18

Which method retrieves a specific method reflectively?

a) `getMethod(name)`
b) `getDeclaredMethod(name)`
c) `getDeclaredMethod(name, paramTypes...)`
d) `fetchMethod(name)`

Answer: c)
Reasoning: `getDeclaredMethod(String name, Class<?>... parameterTypes)` is the correct API.

Question 19

Which exception is thrown when accessing a private method reflectively?

a) `NoSuchMethodException`
b) `IllegalAccessException`
c) `IllegalArgumentException`
d) `AnnotationException`

Answer: b)
Reasoning: Accessing private members without setting accessibility causes `IllegalAccessException`.

Question 20

What is the return type of `getAnnotations()`?

a) `List<Annotation>`
b) `Annotation[]`
c) `AnnotationCollection`
d) `Set<Annotation>`

Answer: b)
Reasoning: It returns an array of all annotations present on the element.

Question 21

Which method allows you to invoke a method via reflection?

a) `run()`
b) `call()`
c) `execute()`
d) `invoke(Object obj, Object... args)`

Answer: d)
Reasoning: `Method.invoke()` is used to dynamically call a method.

Question 22

Which annotation type can accept multiple values?

a) Only `@Retention`
b) Only `@Target`
c) Any annotation with array-based attribute
d) None

Answer: c)
Reasoning: Annotations can define array-type attributes to allow multiple values.

Question 23

Which reflection class helps get field-level annotation?

a) `Class`
b) `Field`
c) `Method`
d) `Parameter`

Answer: b)
Reasoning: Use `Class.getDeclaredField()` and then call `getAnnotation()` on it.

Question 24

What is required to access private members using reflection?

a) Make class public
b) Compile with `-Xreflection`
c) Use `setAccessible(true)`
d) Use `allowPrivateAccess()`

Answer: c)
Reasoning: `AccessibleObject.setAccessible(true)` allows access to private fields/methods.

Question 25

If a class has an annotation with `RetentionPolicy.RUNTIME`, what will happen?

a) Annotation ignored at runtime
b) Annotation available for reflection
c) Annotation discarded after compilation
d) Annotation only used by compiler

Answer: b)
Reasoning: `RUNTIME` retention allows annotations to be visible via reflection.

Java 8 MCQs – Topic: Concurrency & Parallelism

Subtopics:

- `Thread`, `Runnable`, `ExecutorService`
- `synchronized`, `volatile`, `AtomicInteger`
- `ForkJoinPool`, `parallelStream()`
- Thread lifecycle, race conditions, lock mechanisms
- `Callable`, `Future`, `CompletionService`

Question 1

Which method starts a thread in Java?

a) `run()`
b) `start()`
c) `execute()`
d) `launch()`

Answer: b)
Reasoning: `start()` begins a new thread; `run()` would execute in current thread.

Question 2

What interface allows for a thread to return a value?

a) `Runnable`
b) `Executor`
c) `Callable`
d) `Future`

Answer: c)
Reasoning: `Callable` has `call()` which returns a value and can throw exceptions.

Question 3

Which class represents a future result of an asynchronous computation?

a) `Promise`
b) `Future`
c) `Completable`
d) `ThreadResult`

Answer: b)
Reasoning: `Future<V>` represents result of a task that may complete in future.

Question 4

How is mutual exclusion achieved in Java?

a) Using `volatile`
b) Using `synchronized`
c) Using `static`
d) Using `final`

Answer: b)
Reasoning: `synchronized` ensures that only one thread accesses a block at a time.

Question 5

Which is **true** about `volatile` keyword?

a) Ensures atomicity
b) Ensures visibility between threads
c) Is the same as `synchronized`
d) Prevents thread from switching

Answer: b)
Reasoning: `volatile` ensures updates to a variable are visible across threads.

Question 6

Which executor service allows for a pool of reusable threads?

a) `Executors.newThreadExecutor()`
b) `Executors.newCachedThreadPool()`
c) `Executors.newSingleThreadExecutor()`
d) `Executors.newFixedThreadPool()`

Answer: d)
Reasoning: `newFixedThreadPool(n)` reuses up to `n` threads.

Question 7

Which class is used for fork/join parallelism?

a) `ExecutorService`
b) `ForkJoinTask`
c) `ForkJoinPool`
d) `FutureTask`

Answer: c)
Reasoning: `ForkJoinPool` is designed for work-stealing and divide-and-conquer parallelism.

Question 8

Which method submits a task and returns a `Future`?

a) `execute(Runnable)`
b) `submit(Runnable)`

c) `submit(Callable)`
d) Both b and c

Answer: d)
Reasoning: `submit()` supports both `Runnable` and `Callable`.

Question 9

Which exception is thrown if a `Future.get()` times out?

a) `TimeoutError`
b) `TimeoutException`
c) `InterruptedException`
d) `ExecutionException`

Answer: b)
Reasoning: `get(timeout, unit)` throws `TimeoutException` if result not ready.

Question 10

What is the correct way to shutdown an `ExecutorService`?

a) `shutdown()`
b) `terminate()`
c) `exit()`
d) `close()`

Answer: a)
Reasoning: `shutdown()` initiates an orderly shutdown in which tasks already submitted are executed, but no new tasks are accepted.

Question 11

What does `Thread.sleep(1000)` do?

a) Pauses thread permanently
b) Waits for 1000 seconds
c) Pauses the current thread for ~1 second
d) Terminates thread after 1 second

Answer: c)
Reasoning: `sleep(ms)` pauses the **current** thread for given milliseconds.

Question 12

Which condition can occur when multiple threads access shared data unsafely?

a) Deadlock
b) Starvation
c) Race condition
d) Thread leak

Answer: c)
Reasoning: A race condition happens when outcome depends on timing of thread interleaving.

Question 13

How many threads are in a `newSingleThreadExecutor()`?

a) Unlimited
b) 0
c) 1
d) Fixed to 10

Answer: c)
Reasoning: It uses exactly **one** thread to execute submitted tasks sequentially.

Question 14

Which of the following is **thread-safe** for counters?

a) `int`
b) `volatile int`
c) `AtomicInteger`
d) `Long`

Answer: c)
Reasoning: `AtomicInteger` provides lock-free, thread-safe operations.

Question 15

Which method forces a thread to give up CPU?

a) `Thread.stop()`
b) `Thread.yield()`
c) `Thread.pause()`
d) `Thread.freeze()`

Answer: b)
Reasoning: `yield()` hints that the thread is willing to yield execution.

Question 16

Which component is part of the Fork/Join framework?

a) `ForkPoolManager`
b) `RecursiveAction`
c) `BatchExecutor`
d) `ThreadJoiner`

Answer: b)
Reasoning: `RecursiveAction` (no return) and `RecursiveTask` (returns value) are core Fork/Join components.

Question 17

What will happen if `shutdownNow()` is called on ExecutorService?

a) Gracefully finishes tasks
b) Cancels currently executing tasks
c) Waits for all tasks to complete
d) Blocks forever

Answer: b)
Reasoning: `shutdownNow()` attempts to **stop all actively executing tasks** immediately.

Question 18

How can we ensure a block is accessed by only one thread?

a) Use `final`
b) Use `synchronized`
c) Use `volatile`
d) Use `static`

Answer: b)
Reasoning: `synchronized` provides mutual exclusion.

Question 19

Which method allows a thread to wait for another to complete?

a) `join()`
b) `wait()`
c) `block()`
d) `finish()`

Answer: a)
Reasoning: `join()` blocks the current thread until the target thread finishes.

Question 20

How do you create a thread-safe map?

a) `HashMap`
b) `TreeMap`
c) `ConcurrentHashMap`
d) `LinkedHashMap`

Answer: c)
Reasoning: `ConcurrentHashMap` is designed for safe concurrent access.

Question 21

Which executor service scales dynamically based on demand?

a) `newFixedThreadPool()`
b) `newSingleThreadExecutor()`
c) `newCachedThreadPool()`
d) `newScheduledThreadPool()`

Answer: c)
Reasoning: `newCachedThreadPool()` creates threads as needed and reuses idle ones.

Question 22

Which class handles multiple tasks with different completion times and lets you retrieve them in the order they finish?

a) `FutureQueue`
b) `CompletionService`
c) `CallableManager`
d) `TaskBatcher`

Answer: b)
Reasoning: `ExecutorCompletionService` decouples submission from result collection.

Question 23

What does `invokeAll()` return?

a) List of completed results
b) List of exceptions
c) List of `Future` objects
d) List of `Runnable` objects

Answer: c)
Reasoning: `invokeAll()` takes a collection of `Callable` and returns `List<Future<T>>`.

Question 24

Which Java 8 stream operation supports concurrency?

a) `stream()`
b) `sequentialStream()`
c) `parallelStream()`
d) `multiStream()`

Answer: c)
Reasoning: `parallelStream()` splits tasks across multiple threads for performance.

Question 25

What will happen if you submit a long-running task to a fixed thread pool with only 1 thread?

a) Executes all tasks simultaneously
b) All tasks are rejected
c) Tasks are queued and executed one after another
d) Causes deadlock

Answer: c)
Reasoning: Tasks are queued and executed in order based on thread availability.

Java 8 MCQs – Topic: File I/O (NIO.2, Path, Files, Streams)

Subtopics:

- `Path`, `Paths`, `Files`, `StandardOpenOption`
- `BufferedReader`, `BufferedWriter`
- `Files.walk`, `walkFileTree`, `DirectoryStream`
- Reading/writing lines, file attributes
- `Path.resolve()`, `Path.relativize()`, `Path.normalize()`

Question 1

Which package contains the `Path` and `Files` classes?

a) `java.io`
b) `java.nio`
c) `java.nio.file`
d) `java.file.io`

Answer: c)
Reasoning: Both `Path` and `Files` belong to `java.nio.file`.

Question 2

What does `Files.exists(path)` return?

a) Always true
b) Always false
c) True if the file/directory exists
d) Throws exception if path is invalid

Answer: c)
Reasoning: It checks physical existence on the filesystem.

Question 3

How do you get a `Path` object for a file?

a) `new Path("file.txt")`
b) `Files.path("file.txt")`
c) `Paths.get("file.txt")`
d) `File.getPath("file.txt")`

Answer: c)
Reasoning: `Paths.get()` is the standard method to create a `Path`.

Question 4

What does `path.normalize()` do?

a) Converts relative path to absolute
b) Cleans redundant path elements (e.g., `.` or `..`)
c) Deletes the file
d) Checks file size

Answer: b)
Reasoning: It simplifies path elements (e.g., `/a/b/../c` → `/a/c`).

Question 5

Which method reads all lines of a file as a `List<String>`?

a) `Files.readText()`
b) `Files.getLines()`
c) `Files.readAllLines(path)`
d) `BufferedReader.readLines()`

Answer: c)
Reasoning: `Files.readAllLines()` loads entire file content into memory.

Question 6

Which method is used to write lines to a file in one shot?

a) `Files.appendLines()`
b) `Files.printLines()`
c) `Files.write(path, lines)`
d) `Files.addAll(path, lines)`

Answer: c)
Reasoning: `Files.write()` takes a `Path` and `Iterable<String>`.

Question 7

Which interface is used for filtering directory entries?

a) `DirectoryScanner`
b) `FilterStream`
c) `DirectoryFilter`
d) `DirectoryStream.Filter`

Answer: d)
Reasoning: Used with `DirectoryStream` to filter files/directories during iteration.

Question 8

Which method is used to traverse directory trees?

a) `Files.loop()`
b) `Files.walk()`
c) `Files.trace()`
d) `Path.walkTree()`

Answer: b)
Reasoning: `Files.walk()` uses a depth-first approach to walk file tree.

Question 9

What happens if the file already exists during `Files.createFile(path)`?

a) It replaces the file
b) It deletes the existing file
c) It throws `FileAlreadyExistsException`
d) It appends to the file

Answer: c)
Reasoning: `createFile` expects the file to **not exist**.

Question 10

Which method writes a file with `APPEND` option?

a) `Files.append(path, lines)`
b) `Files.write(path, lines, APPEND)`
c) `Files.write(path, lines, StandardOpenOption.APPEND)`
d) `Files.output(path, lines, AppendMode)`

Answer: c)
Reasoning: To write in append mode, you pass `StandardOpenOption.APPEND`.

Question 11

What does `Files.isDirectory(path)` check?

a) If path exists
b) If path points to a directory
c) If file is readable
d) If file is a symbolic link

Answer: b)
Reasoning: Returns true if the file exists and is a directory.

Question 12

What does `Files.copy(src, dest)` do by default?

a) Appends source content to destination
b) Overwrites destination file
c) Throws exception if dest exists
d) Creates a symbolic link

Answer: c)
Reasoning: Without extra options, it fails if destination exists.

Question 13

Which exception is thrown if a file doesn't exist during read?

a) `NoFileException`
b) `FileNotFoundException`
c) `NoSuchFileException`
d) `IOException`

Answer: c)
Reasoning: `NoSuchFileException` is a subclass of `IOException`.

Question 14

What does `Files.deleteIfExists(path)` do?

a) Always deletes the file
b) Deletes if file exists; otherwise does nothing
c) Deletes and returns deleted content
d) Throws exception if file is missing

Answer: b)
Reasoning: Prevents unnecessary exceptions when the file may not exist.

Question 15

How do you read a file line-by-line with minimal memory?

a) `Files.readAllLines()`
b) `BufferedReader` via `Files.newBufferedReader()`
c) `Files.getLines()`
d) `Files.stream()`

Answer: b)
Reasoning: `BufferedReader` reads efficiently with small memory usage.

Question 16

Which method creates a directory (but not its parents)?

a) `Files.createDirectory()`
b) `Files.makeDir()`
c) `Files.mkdir()`
d) `Paths.createDir()`

Answer: a)
Reasoning: Only `Files.createDirectory()` creates a single directory.

Question 17

What does `path.resolve("data.txt")` return?

a) Absolute path
b) Relative path
c) New `Path` with “data.txt” joined
d) Nothing

Answer: c)
Reasoning: It appends the argument to the current path.

Question 18

Which method retrieves the file name from a `Path`?

a) `path.getName()`
b) `path.fileName()`
c) `path.getFileName()`
d) `path.file()`

Answer: c)
Reasoning: Returns the final name in the path, like `log.txt`.

Question 19

Which method creates a temporary file?

a) `File.createTempFile()`
b) `Files.tempFile()`
c) `Files.createTempFile()`
d) `Paths.createTempFile()`

Answer: c)
Reasoning: `Files.createTempFile()` creates a unique temp file.

Question 20

What is the result of:

```
java
CopyEdit
Path p1 = Paths.get("/home/user");
Path p2 = Paths.get("docs/readme.txt");
Path result = p1.resolve(p2);
```

a) `/home/user/docs/readme.txt`
b) `/docs/readme.txt`
c) `docs/readme.txt`
d) `/home/user`

Answer: a)
Reasoning: `resolve()` appends p2 to p1 unless p2 is absolute.

Question 21

Which method walks the file tree recursively?

a) `Files.list()`
b) `Files.walkFileTree()`
c) `Files.search()`
d) `Path.walk()`

Answer: b)
Reasoning: Walks directory tree using a `FileVisitor`.

Question 22

Which interface must you implement for `walkFileTree()`?

a) `PathScanner`
b) `FileVisitor`
c) `PathVisitor`
d) `DirectoryReader`

Answer: b)
Reasoning: Implement `FileVisitor<Path>` for file traversal logic.

Question 23

Which `StandardOpenOption` creates file if it doesn’t exist?

a) `CREATE_NEW`
b) `APPEND`
c) `CREATE`
d) `WRITE`

Answer: c)
Reasoning: `CREATE` opens file if exists, or creates it if not.

Question 24

Which statement about `Files.lines(path)` is true?

a) Loads entire file into memory
b) Returns `List<String>`
c) Returns `Stream<String>`
d) Cannot be used with large files

Answer: c)
Reasoning: Streams lines lazily for efficiency.

Question 25

How can you detect if a file is a symbolic link?

a) `Files.isSymbolicLink(path)`
b) `Files.isSoftLink(path)`
c) `path.isSymbolic()`
d) `Files.linkType(path)`

Answer: a)
Reasoning: Only `Files.isSymbolicLink()` determines symlink.

Java 8 MCQs – Topic: Lambda Expressions & Functional Interfaces

Subtopics:

- `Lambda` syntax and scope rules
- `Predicate`, `Consumer`, `Function`, `Supplier`
- `BiFunction`, `UnaryOperator`, `BinaryOperator`
- Method references
- Variable capture and `effectively final`

Question 1

Which is the correct lambda syntax for a no-arg function returning 5?

a) `()-> return 5;`
b) `( ) => 5;`
c) `() -> 5`
d) `-> 5`

Answer: c)
Reasoning: `() -> 5` is valid syntax for no-arg lambdas returning a value.

Question 2

Which functional interface takes no arguments and returns a value?

a) `Predicate<T>`
b) `Function<T, R>`
c) `Supplier<T>`
d) `Consumer<T>`

Answer: c)
Reasoning: `Supplier<T>` has `T get()` method with no arguments.

Question 3

Which interface is used when accepting and returning same type?

a) `UnaryOperator<T>`
b) `Function<T, R>`
c) `Supplier<T>`
d) `Predicate<T>`

Answer: a)
Reasoning: `UnaryOperator<T>` extends `Function<T, T>`.

Question 4

Which lambda matches `Predicate<String>`?

a) `(String s) -> s.length()`
b) `s -> s.equals("")`
c) `() -> true`
d) `s -> System.out.println(s)`

Answer: b)
Reasoning: `Predicate<T>` has method `boolean test(T t)`, matching `s -> s.equals("")`.

Question 5

Which method reference is equivalent to `x -> System.out.println(x)`?

a) `System::println(x)`
b) `System.out::println`
c) `println::System.out`
d) `::System.out.println`

Answer: b)
Reasoning: Instance method reference syntax is `objectRef::method`.

Question 6

Which functional interface consumes and returns nothing?

a) `Runnable`
b) `Consumer<T>`
c) `Function<T, R>`
d) `Supplier<T>`

Answer: b)
Reasoning: `Consumer<T>` has method `void accept(T t)`.

Question 7

What does this lambda do: `x -> x + 10`?

a) It's invalid
b) Implements `Predicate`
c) Implements `Function<Integer, Integer>`
d) Implements `Supplier`

Answer: c)
Reasoning: One argument → one return implies `Function<T, R>`.

Question 8

Which method does `Runnable` contain?

a) `void test()`
b) `boolean run()`
c) `void run()`
d) `T execute()`

Answer: c)
Reasoning: `Runnable` only contains `void run()` with no parameters.

Question 9

When is a variable "effectively final"?

a) After it’s marked with `final`
b) When it’s declared as `static`

c) If it’s never modified after initialization
d) When modified inside a lambda

Answer: c)
Reasoning: Variables used inside lambdas must be **effectively final**, i.e., not reassigned.

Question 10

Which of the following is not a functional interface?

a) `Predicate`
b) `Runnable`
c) `Comparator`
d) `List`

Answer: d)
Reasoning: `List` is not a functional interface. It has many abstract methods.

Question 11

Which lambda expression is valid for a `BinaryOperator<Integer>`?

a) `(a, b) -> a * b`
b) `(a) -> a * a`
c) `() -> 5`
d) `(x, y) -> System.out.println(x + y)`

Answer: a)
Reasoning: `BinaryOperator<T>` requires `(T, T) -> T`.

Question 12

What does `Function<String, Integer>` represent?

a) Function that takes a `String` and returns an `Integer`
b) Function that prints a string
c) Function that takes an `Integer` and returns a `String`
d) A supplier of string functions

Answer: a)
Reasoning: The first type is input, second is output.

Question 13

What is the purpose of `Predicate<T>`?

a) To consume a value
b) To produce a value

c) To return a boolean based on a test
d) To run a thread

Answer: c)
Reasoning: `Predicate<T>` is used to test a condition on input and return a boolean.

Question 14

What is the return type of `Predicate<T>.test(T t)`?

a) `T`
b) `void`
c) `boolean`
d) `Object`

Answer: c)
Reasoning: `.test()` returns a `boolean`.

Question 15

Which of the following is **not** a valid method reference?

a) `String::toUpperCase`
b) `System.out::println`
c) `Math::max`
d) `int::parseInt`

Answer: d)
Reasoning: `int` is a primitive, so it cannot have method references.

Question 16

Choose the correct signature for a `Supplier<String>`.

a) `String apply()`
b) `void accept(String s)`
c) `String get()`
d) `boolean test(String s)`

Answer: c)
Reasoning: `Supplier<T>` has `T get()` method.

Question 17

What does `list.removeIf(e -> e.isEmpty())` do?

a) Removes empty elements from the list
b) Removes all elements
c) Adds empty elements
d) Filters non-empty elements

Answer: a)
Reasoning: `removeIf(Predicate)` removes items that match the predicate.

Question 18

Which lambda is valid for a `Consumer<String>`?

a) `s -> s.length()`
b) `s -> s.toUpperCase()`
c) `s -> System.out.println(s)`
d) `s -> return s;`

Answer: c)
Reasoning: `Consumer<T>` uses `void accept(T t)`; printing fits this pattern.

Question 19

What is the correct return type of a lambda implementing `Callable<T>`?

a) `void`
b) `int`
c) `T`
d) `boolean`

Answer: c)
Reasoning: `Callable<T>` requires the lambda to return a value of type `T`.

Question 20

Which lambda is invalid due to variable scoping?

```
java
CopyEdit
int val = 10;
Runnable r = () -> {
  int val = 15;
  System.out.println(val);
};
```

a) Compiles and runs
b) Compiles but throws at runtime
c) Compilation error
d) Syntax error

Answer: c)
Reasoning: Cannot redefine `val` inside the lambda block; causes a scope conflict.

Question 21

Can a lambda access instance variables?

a) No
b) Yes, if marked `final`
c) Yes, always
d) Only from static context

Answer: c)
Reasoning: Lambdas can freely access instance variables.

Question 22

Which interface takes two arguments and returns a result?

a) `BiConsumer`
b) `BiFunction`
c) `Function`
d) `BinaryPredicate`

Answer: b)
Reasoning: `BiFunction<T, U, R>` takes two inputs and returns a value.

Question 23

Which of the following is NOT a characteristic of lambdas?

a) Can capture effectively final variables
b) Can throw checked exceptions
c) Can override multiple abstract methods
d) Can be passed as functional interface implementations

Answer: c)
Reasoning: Lambdas must implement a **single** abstract method (SAM interface).

Question 24

Which lambda is valid for filtering strings starting with "A"?

a) `s -> s.startsWith("A")`
b) `s -> System.out.println(s)`

c) `() -> "A"`
d) `s -> return s.contains("A")`

Answer: a)
Reasoning: This matches `Predicate<String>` which returns a boolean.

Question 25

What happens if you modify a local variable inside a lambda?

a) Compilation error
b) Value updated
c) NullPointerException
d) Lambda returns null

Answer: a)
Reasoning: Only **effectively final** variables can be used in lambdas.

Java 8 MCQs – Topic: Streams API

Subtopics:

- Stream creation and pipeline
- Intermediate vs terminal operations
- Filtering, mapping, sorting, collecting
- `reduce`, `collect`, `count`, `forEach`
- Stream vs parallelStream behavior

Question 1

Which is a **terminal** operation in streams?

a) `filter()`
b) `map()`
c) `sorted()`
d) `collect()`

Answer: d)
Reasoning: Terminal operations trigger processing; `collect()` ends the stream pipeline.

Question 2

Which intermediate operation changes stream elements?

a) `filter()`
b) `map()`
c) `count()`
d) `forEach()`

Answer: b)
Reasoning: `map()` transforms each element into another form.

Question 3

What does `stream.filter(x -> x > 5)` return?

a) The same stream
b) A new stream with matching elements
c) A list of all elements
d) None of the above

Answer: b)
Reasoning: `filter()` returns a new stream with elements that match the predicate.

Question 4

Which method collects stream elements into a list?

a) `collect(Collectors.toList())`
b) `stream.toList()`
c) `stream.list()`
d) `stream.asList()`

Answer: a)
Reasoning: `Collectors.toList()` collects elements into a `List`.

Question 5

Which of the following creates a **finite** stream?

a) `Stream.generate(Math::random)`
b) `Stream.iterate(0, n -> n + 1)`
c) `Arrays.stream(new int[]{1,2,3})`
d) `Stream.empty().limit(5)`

Answer: c)
Reasoning: Stream from array is naturally finite.

Question 6

Which operation terminates a stream?

a) `filter()`
b) `map()`
c) `limit()`
d) `forEach()`

Answer: d)
Reasoning: `forEach()` consumes the stream and ends its lifecycle.

Question 7

What does `distinct()` do?

a) Sorts the stream
b) Filters nulls
c) Removes duplicates
d) Maps elements

Answer: c)
Reasoning: Removes duplicate elements using `equals()`.

Question 8

What does `peek()` do?

a) Collects data
b) Changes values
c) Performs side-effects (for debugging)
d) Terminates the stream

Answer: c)
Reasoning: `peek()` is useful for **debugging**, not transformation or collection.

Question 9

What does `reduce()` do?

a) Combines elements into one result
b) Splits elements
c) Terminates a stream early
d) Returns a new stream

Answer: a)
Reasoning: `reduce()` combines elements using an accumulator and identity.

Question 10

Which stream method counts elements?

a) `stream.size()`
b) `stream.count()`
c) `stream.length()`
d) `Collectors.counting()`

Answer: b)
Reasoning: `count()` is a terminal operation that returns `long`.

Question 11

Which of the following is a valid use of `reduce()`?

```
java
CopyEdit
List<Integer> nums = Arrays.asList(1, 2, 3);
```

a) `nums.stream().reduce((a, b) -> a + b)`
b) `nums.stream().reduce((a, b) -> a * b)`
c) `nums.stream().reduce(1, (a, b) -> a * b)`
d) All of the above

Answer: d)
Reasoning: `reduce()` can use no identity (returns `Optional`), or with identity (returns result).

Question 12

Which of the following returns `true` if any element matches a predicate?

a) `allMatch()`
b) `noneMatch()`
c) `anyMatch()`
d) `filter().isPresent()`

Answer: c)
Reasoning: `anyMatch()` checks if **at least one** element satisfies the condition.

Question 13

Which stream operation short-circuits?

a) `map()`
b) `filter()`
c) `limit()`
d) `forEach()`

Answer: c)
Reasoning: `limit()` ends processing early after N elements.

Question 14

What does this return?

```
java
CopyEdit
Stream.of("a", "bb", "ccc").mapToInt(String::length).sum();
```

a) 6
b) 5
c) 3
d) Compilation error

Answer: a)
Reasoning: `"a"=1`, `"bb"=2`, `"ccc"=3` → 1+2+3 = 6.

Question 15

Which collector returns a `Map` from stream elements?

a) `toMap()`
b) `groupingBy()`
c) `partitioningBy()`
d) All of the above

Answer: d)
Reasoning: All these collectors can return `Map` depending on key/value logic.

Question 16

What does `Collectors.groupingBy(String::length)` do?

a) Groups strings by their character count
b) Sorts strings
c) Filters strings by length
d) Partitions strings by even/odd length

Answer: a)
Reasoning: Groups by key: string length.

Question 17

What is the result of:

```
java
```

```
CopyEdit
Stream.of("java", "code").collect(Collectors.joining("-"));
```

a) `java code`
b) `java-code`
c) `[java, code]`
d) Compilation error

> **Answer:** b)
> **Reasoning:** `joining()` with `"-"` uses it as delimiter.

Question 18

Which stream method is best for printing values during processing?

a) `map()`
b) `filter()`
c) `forEach()`
d) `peek()`

> **Answer:** d)
> **Reasoning:** `peek()` is non-terminal and designed for side-effects like logging.

Question 19

Which is **true** about `parallelStream()`?

a) Always faster than `stream()`
b) Suitable for all tasks
c) May give better performance for large independent tasks
d) Guarantees order

> **Answer:** c)
> **Reasoning:** `parallelStream()` can improve performance but must be used with care.

Question 20

What does `findFirst()` return?

a) First element as `Optional<T>`
b) First element directly
c) Throws exception if not found
d) Index of the first element

> **Answer:** a)
> **Reasoning:** `findFirst()` returns `Optional<T>` in case stream is empty.

Question 21

What happens if two keys collide in `Collectors.toMap()`?

a) Runtime exception
b) Returns first key
c) Returns last key
d) Compilation error

Answer: a)
Reasoning: By default, `toMap()` throws `IllegalStateException` on duplicate keys unless merge function is supplied.

Question 22

What's the correct way to count unique elements?

a) `stream.count()`
b) `stream.distinct().count()`
c) `stream.unique().count()`
d) `stream.filter().count()`

Answer: b)
Reasoning: `distinct()` ensures uniqueness, then `count()` returns the total.

Question 23

Which collector splits stream into two groups based on predicate?

a) `groupingBy()`
b) `partitioningBy()`
c) `splittingBy()`
d) `dividingBy()`

Answer: b)
Reasoning: `partitioningBy()` splits into true/false groups.

Question 24

Which of the following is a **lazy** operation?

a) `forEach()`
b) `collect()`
c) `filter()`
d) `count()`

Answer: c)
Reasoning: Intermediate ops like `filter()` are lazy — they don't execute until a terminal operation triggers.

Question 25

How to create a stream from a collection?

a) `collection.toStream()`
b) `collection.stream()`
c) `Stream.of(collection)`
d) `collection.newStream()`

Answer: b)
Reasoning: `stream()` is the standard method on Java collections.

Java 8 MCQs – Topic: `Optional<T>` API

Subtopics:

- `Optional.of()`, `Optional.ofNullable()`, `Optional.empty()`
- `isPresent()`, `ifPresent()`, `orElse()`, `orElseGet()`, `orElseThrow()`
- `map()`, `flatMap()`, `filter()`
- Avoiding `NullPointerException` with `Optional`

Question 1

What does `Optional.of(null)` do?

a) Returns an empty optional
b) Returns null
c) Throws `NullPointerException`
d) Compiles but fails at runtime

Answer: c)
Reasoning: `Optional.of()` requires a non-null value; null causes immediate exception.

Question 2

What does `Optional.ofNullable(null)` return?

a) Throws NPE
b) Optional.empty

c) null
d) Optional with null value

Answer: b)
Reasoning: `ofNullable()` safely wraps null into an empty optional.

Question 3

Which method is used to retrieve the value inside `Optional<T>`?

a) `getValue()`
b) `unwrap()`
c) `get()`
d) `fetch()`

Answer: c)
Reasoning: `get()` returns the value or throws if empty.

Question 4

When should `get()` be avoided?

a) When `Optional` is empty
b) When value is present
c) Always
d) In loops

Answer: a)
Reasoning: `get()` on an empty optional throws `NoSuchElementException`.

Question 5

Which is safer than `get()`?

a) `isPresent()`
b) `orElse()`
c) `ifPresent()`
d) All of the above

Answer: d)
Reasoning: These methods prevent exception-prone access.

Question 6

What does `optional.orElse("default")` do?

a) Returns "default" always
b) Returns "default" if empty
c) Throws exception
d) Ignores the value inside

Answer: b)
Reasoning: Supplies a fallback value if Optional is empty.

Question 7

What is the key difference between `orElse()` and `orElseGet()`?

a) `orElse()` is lazy, `orElseGet()` is eager
b) `orElse()` takes a supplier, `orElseGet()` takes a value
c) `orElse()` always evaluates the default value
d) `orElseGet()` cannot be used

Answer: c)
Reasoning: `orElse()` always evaluates its argument, `orElseGet()` evaluates only if needed.

Question 8

What does `ifPresent(System.out::println)` do?

a) Prints only if value is present
b) Always prints
c) Never prints
d) Throws if empty

Answer: a)
Reasoning: It's a safe, conditional execution if value exists.

Question 9

What is the result of:

```
java
CopyEdit
Optional.empty().orElse("fallback");
```

a) null
b) "fallback"
c) Optional["fallback"]
d) throws exception

Answer: b)
Reasoning: Returns the fallback value since optional is empty.

Question 10

Which method transforms an `Optional<T>`?

a) `map()`
b) `get()`
c) `orElse()`
d) `isPresent()`

Answer: a)
Reasoning: `map()` applies a function and wraps result into another optional.

Question 11

What does `Optional.of("java").map(String::toUpperCase)` return?

a) `"JAVA"`
b) `Optional["JAVA"]`
c) `Optional[Optional["JAVA"]]`
d) `"Optional[JAVA]"`

Answer: b)
Reasoning: `map()` transforms the value inside and returns an `Optional` wrapping the result.

Question 12

Which method is preferred when the mapping function returns an `Optional`?

a) `map()`
b) `flatMap()`
c) `get()`
d) `orElseThrow()`

Answer: b)
Reasoning: `flatMap()` avoids nested optionals by flattening the result.

Question 13

What is the output?

```
java
CopyEdit
Optional.of("abc").filter(s -> s.length() > 3)
```

a) `Optional["abc"]`
b) `"abc"`

c) `Optional.empty`
d) Throws Exception

Answer: c)
Reasoning: "abc".length = 3 → filter fails → result is empty.

Question 14

What happens if you call `get()` on an empty `Optional`?

a) Returns null
b) Throws `IllegalStateException`
c) Throws `NoSuchElementException`
d) Compiles but prints null

Answer: c)
Reasoning: `Optional.get()` on empty throws `NoSuchElementException`.

Question 15

What is the type of return from `Optional.map(f)`?

a) `T`
b) `Optional<T>`
c) `Optional<R>`
d) `Function<T, R>`

Answer: c)
Reasoning: `map(Function<T, R>)` returns `Optional<R>`.

Question 16

How would you handle the absence of a value using a lambda?

a) `ifPresentOrElse()`
b) `map()`
c) `filter()`
d) `Optional.of(null)`

Answer: a)
Reasoning: Available from Java 9, `ifPresentOrElse()` gives a fallback lambda to run.

Question 17

Which method throws a custom exception if value is absent?

a) `orElse()`
b) `orElseThrow(Supplier)`
c) `get()`
d) `orElseGet()`

Answer: b)
Reasoning: `orElseThrow()` allows defining an exception supplier.

Question 18

Which of the following is **true**?

a) `Optional<String>` can be null
b) `Optional.of(null)` is valid
c) `Optional` replaces all use of null
d) `Optional` is a container for possibly-null values

Answer: d)
Reasoning: It's a wrapper — doesn't replace null but offers safer access.

Question 19

What does this return?

```
java
CopyEdit
Optional.ofNullable(null).isPresent()
```

a) `true`
b) `false`
c) `Optional.empty`
d) `null`

Answer: b)
Reasoning: `Optional.empty().isPresent()` is `false`.

Question 20

Which of the following best avoids a `NullPointerException`?

a) `value.get()`
b) `Optional.of(value).get()`
c) `Optional.ofNullable(value).orElse("default")`
d) `value.toString()`

Answer: c)
Reasoning: `ofNullable()` + `orElse()` guards against null.

Question 21

When is `flatMap()` required?

a) When nested optionals need to be flattened
b) When mapping to `Stream<T>`
c) When reducing to one result
d) When supplying a default

Answer: a)
Reasoning: `flatMap()` avoids `Optional<Optional<T>>` nesting.

Question 22

Which returns a value or computes it if absent?

a) `orElse()`
b) `orElseGet()`
c) `map()`
d) `filter()`

Answer: b)
Reasoning: `orElseGet(Supplier)` is lazy — only called if value is absent.

Question 23

Which is a valid usage?

a) `Optional.get().orElse("default")`
b) `Optional.of("val").orElseGet(() -> "default")`
c) `Optional.of("val").getOrElse("default")`
d) `Optional("val")`

Answer: b)
Reasoning: Valid use of `orElseGet()` on non-empty Optional.

Question 24

Choose the correct transformation chain:

```
java
CopyEdit
Optional<String> val = Optional.of("java");
int len = val.map(String::length).orElse(0);
```

a) Compiles and returns 4
b) Compilation error

c) Returns Optional[4]
d) Throws exception

Answer: a)
Reasoning: `"java".length()` → 4, orElse not triggered.

Question 25

Which one **avoids** evaluating the fallback if present?

a) `orElse()`
b) `orElse(null)`
c) `orElseGet()`
d) `orElseThrow()`

Answer: c)
Reasoning: `orElseGet(Supplier)` defers evaluation until needed.

Java 8 MCQs – Topic: Default & Static Methods in Interfaces

Subtopics:

- Default methods in interfaces
- Static methods in interfaces
- Interface method conflict resolution
- Inheritance rules with interfaces and classes
- Diamond problem handling in Java 8

Question 1

Which of the following is **valid** in a Java 8 interface?

a) `private void log()`
b) `default void run() {}`
c) `static default void help() {}`
d) `protected void start()`

Answer: b)
Reasoning: Java 8 allows `default` methods with implementation in interfaces.

Question 2

What is the correct way to declare a static method in an interface?

a) `public static void help();`
b) `default static void help() {}`
c) `static void help() {}`
d) `void help() static {}`

Answer: c)
Reasoning: Static methods in interfaces must have a body.

Question 3

Can default methods be overridden in implementing classes?

a) No
b) Yes, optionally
c) Only if abstract
d) Only if private

Answer: b)
Reasoning: Default methods can be overridden or inherited as-is.

Question 4

What happens if a class implements two interfaces with the same default method?

a) Compile error
b) Runtime error
c) Inherits first one
d) Must override the method

Answer: d)
Reasoning: Java requires conflict resolution via explicit override.

Question 5

Can interface default methods access class instance variables?

a) Yes
b) No
c) Only private ones
d) Only via static context

Answer: b)
Reasoning: Interfaces cannot access class instance state directly.

Question 6

What does this code output?

```
java
CopyEdit
interface A {
    default String getName() { return "A"; }
}
interface B {
    default String getName() { return "B"; }
}
class C implements A, B {
    public String getName() { return A.super.getName(); }
}
```

a) A
b) B
c) Compile error
d) Runtime error

Answer: a)
Reasoning: Conflict resolved explicitly with `A.super.getName()`.

Question 7

Where can static methods of an interface be called from?

a) Instance of implementing class
b) Directly via interface name
c) Via `super`
d) Cannot be called

Answer: b)
Reasoning: Interface static methods are called as `InterfaceName.method()`.

Question 8

What is true about default methods?

a) They make interfaces abstract
b) They allow adding new behavior without breaking implementations
c) They can be private
d) They must be final

Answer: b)
Reasoning: Java 8 added default methods to evolve interfaces safely.

Question 9

Which is NOT allowed in an interface in Java 8?

a) Abstract methods
b) Static methods

c) Default methods
d) Constructors

Answer: d)
Reasoning: Interfaces still cannot have constructors.

Question 10

What happens if a class implements an interface with only default methods?

a) Must override all methods
b) Compiles without override
c) Throws runtime exception
d) Interface becomes abstract

Answer: b)
Reasoning: Default methods have implementation — override is optional.

Question 11

Can an abstract class override a default method from an interface?

a) No
b) Yes, and provide its own implementation
c) Only if the method is final
d) Only static methods can be overridden

Answer: b)
Reasoning: Abstract classes can override default methods and leave them abstract or provide a body.

Question 12

Which of the following allows you to call a default method from inside the implementing class?

a) `this.getDefault()`
b) `super.getDefault()`
c) `InterfaceName.super.method()`
d) `default.method()`

Answer: c)
Reasoning: `InterfaceName.super.method()` is the syntax to call interface default methods.

Question 13

Which modifier is **not** valid for interface default methods?

a) `public`
b) `private`
c) `protected`
d) `final`

Answer: c)
Reasoning: Default methods are implicitly public; `protected` is not allowed in interfaces.

Question 14

What happens if a class inherits a method from a superclass and an interface with a default method?

a) Compiler chooses the interface method
b) Compiler chooses the superclass method
c) It results in ambiguity
d) Runtime error

Answer: b)
Reasoning: Class wins over interface — class method is chosen over interface default.

Question 15

Which of the following is **true** about static methods in interfaces?

a) They can be inherited
b) They cannot be overridden
c) They can only be private
d) They must be abstract

Answer: b)
Reasoning: Static methods in interfaces are **not inherited** and **cannot be overridden**.

Question 16

Choose valid Java 8 interface structure:

```
java
CopyEdit
interface Converter {
    static void log(String msg) {
        System.out.println(msg);
    }
    default String convert(String s) {
        return s.toUpperCase();
    }
}
```

a) Valid
b) Invalid — static method not allowed

c) Invalid — default method needs abstract keyword
d) Invalid — return type must be void

Answer: a)
Reasoning: Correct usage of static and default methods in an interface.

Question 17

Which resolves the "diamond problem" with interfaces?

a) Interfaces can't extend each other
b) Abstract class is needed
c) Java forces explicit method override in case of conflict
d) Use `final` keyword

Answer: c)
Reasoning: Java requires the class to override conflicting default methods to avoid ambiguity.

Question 18

What does this code print?

```
java
CopyEdit
interface A {
    default String who() { return "A"; }
}
interface B extends A {
    default String who() { return "B"; }
}
class C implements B {}
System.out.println(new C().who());
```

a) A
b) B
c) Compile error
d) Runtime error

Answer: b)
Reasoning: B overrides A, and C implements B — so B's method is invoked.

Question 19

Can a class implement multiple interfaces with **non-conflicting** default methods?

a) No
b) Only one
c) Yes
d) Only static methods are allowed

Answer: c)
Reasoning: No conflict → compiler allows multiple default methods from different interfaces.

Question 20

What is the return type of a default method?

a) Must be `void`
b) Must match Object class methods
c) Can be anything
d) Must match static method return type

Answer: c)
Reasoning: Default methods can return any type, like regular instance methods.

Question 21

Can you call a static method of an interface using an instance?

a) Yes
b) No
c) Only inside the interface
d) Only inside default methods

Answer: b)
Reasoning: Static methods must be called with interface name, not instance.

Question 22

Default methods cannot override which methods?

a) Other default methods
b) Methods from `Object` class
c) Methods with the same signature
d) Private methods

Answer: b)
Reasoning: You cannot override `Object` methods like `toString()`, `equals()` in interface default.

Question 23

What happens if two interfaces provide identical default methods and no override is given?

a) First interface is chosen
b) Compile error

c) Runtime error
d) Method is ignored

Answer: b)
Reasoning: Java requires you to resolve ambiguity explicitly by overriding the method.

Question 24

Which is a correct use of default method?

a) Used to extend an interface without breaking old implementations
b) Used as a constructor
c) Used as a private helper
d) Must be abstract

Answer: a)
Reasoning: Primary reason Java added default methods — backward compatibility.

Question 25

Which of the following is a valid call?

```
java
CopyEdit
interface Helper {
    static String get() { return "value"; }
}
```

a) `Helper.get()`
b) `new Helper().get()`
c) `Helper::get()`
d) `Helper.get(this)`

Answer: a)
Reasoning: Static methods must be called on interface name directly.