Java Advanced Interface, Inheritance and Exception Handling

---

1. Multiple Interface Default Methods: Conflict Resolution

If a class implements two interfaces with the same default method, you must override the method and explicitly choose which default to call.

```
interface A {
    default void ping() { System.out.println("A"); }
}
interface B {
    default void ping() { System.out.println("B"); }
}
class C implements A, B {
    public void ping() {
        A.super.ping(); // Call A's ping
        B.super.ping(); // Call B's ping
    }
}
public class Main {
    public static void main(String[] args) {
        new C().ping();
    }
}
```

Error if not overridden: > class C inherits unrelated defaults for ping() from types A and B

Why: - Java requires you to resolve ambiguity when multiple interfaces provide the same default method.

---

2. Static Methods in Interfaces

- Static methods in interfaces are not inherited by implementing classes.
- You must call them using the interface name.

```
interface Util {
    static String id() { return "Util ID"; }
}
class X implements Util {
    static String id() { return "X ID"; }
}
public class Main {
```

```
    public static void main(String[] args) {
        System.out.println(X.id());      // X ID
        System.out.println(Util.id());   // Util ID
        System.out.println(u.id());   // Error: Cannot call static method on instance
    }
}
```

---

3. Private Methods in Interfaces

- Private methods in interfaces are only accessible within the interface itself.
- They cannot be called from implementing classes.

```
interface Parser {
    default String sanitize(String s) { return trimAndLower(s); }
    private String trimAndLower(String s) { return s.trim().toLowerCase(); }
}
class Impl implements Parser {
    public String demo(String s) {
        // return Parser.trimAndLower(s); // Error: Not accessible
        return sanitize(s);
    }
}
```

---

4. Method Overloading vs Overriding with Interfaces

- Overloading: Same method name, different parameter types.
- Overriding: Same method name and parameter types.

```
interface I { void m(Number n); }
class A { public void m(Integer i) { System.out.println("A"); } }
class B extends A implements I {
    @Override
    public void m(Number n) { System.out.println("B"); }
}
public class Main {
    public static void main(String[] args) {
        new B().m(10);          // A (overloading)
        ((I) new B()).m(10);    // B (overriding via interface reference)
    }
}
```

Why: - Reference type determines which method signatures are visible at compile time.

---

5. Covariant Return Types in Interfaces

- You can override a method and return a subtype.

```
class Shape {}
class Circle extends Shape {}
interface Factory { Shape create(); }
class CircleFactory implements Factory {
    @Override
    public Circle create() { return new Circle(); }
}
```

---

6. Exception Compatibility in Overriding

- When a class implements an interface and extends a class, the overriding method cannot throw broader checked exceptions than those declared in both.

```
import java.io.IOException;
import java.sql.SQLException;
abstract class A { abstract void run() throws IOException; }
interface I { void run() throws SQLException; }
class C extends A implements I {
    @Override
    public void run() { System.out.println("C"); } // No checked exceptions allowed
}
```

Why: - Only unchecked exceptions are allowed if the checked exceptions in the parent and interface do not overlap.

---

7. Sealed and Non-Sealed Interfaces

- `sealed` restricts which classes can implement the interface.
- `non-sealed` allows further extension.

```
sealed interface Animal permits Dog, Cat {}
non-sealed interface Dog extends Animal {}
interface Cat extends Animal {}
class Persian implements Dog {}
```

---

8. Static Methods Cannot Be Overridden

- Static methods belong to the class, not the instance.

```java
class P { static void hello() { System.out.println("Hello from P"); } }
class Q { static void hello() { System.out.println("Hello from Q"); } }
P.hello(); // Hello from P
Q.hello(); // Hello from Q
```

---

9. Generics and Type Erasure

- Raw types cause warnings; use parameterized types for safety.

```java
interface Box<T> { T get(); }
class StringBox implements Box<String> {
    @Override
    public String get() { return "Hello"; }
}
Box b = new StringBox(); // Warning: unchecked
Box<String> b2 = new StringBox(); // Safe
```

---

10. Overloading: Most Specific Method Chosen

```java
class Overload {
    void test(String s) { System.out.println("String"); }
    void test(Object o) { System.out.println("Object"); }
}
new Overload().test(null); // String
```

Why: - The most specific applicable method is chosen.

---

11. Overloading vs Overriding: Parameter Types

```java
class Parent { void show(Number n) { System.out.println("Parent"); } }
class Child extends Parent { void show(Integer n) { System.out.println("Child"); } }
Parent p = new Child();
p.show(10); // Parent
```

Why: - Overloading is resolved at compile time based on reference type and parameter types.

---

12. Final Methods Cannot Be Overridden

```java
class Parent { public final void ping() { System.out.println("Parent"); } }
class Child extends Parent { /* public void ping() { ... } // Error */ }
```

---

13. Class vs Interface Default Method Precedence

- If a class (or superclass) provides a method, it overrides any interface default method.

```
interface I { default void ping() { System.out.println("I"); } }
class Super { public void ping() { System.out.println("Super"); } }
class Sub extends Super implements I {}
new Sub().ping(); // Super
```

Why: - Class method always takes precedence over interface default method.

---

14. Method Overloading: Widening vs Boxing

```
class Demo {
    void a(long i) { System.out.println("long"); }
    void a(Integer i) { System.out.println("Integer"); }
}
public class Main {
    public static void main(String[] args) {
        Demo demo = new Demo();
        demo.a(5); // long
    }
}
```

Explanation: - `5` is an `int`. There is no `a(int)` method. - Java prefers primitive widening (`int` → `long`) over boxing (`int` → `Integer`). - So, `a(long)` is called and prints `long`.

Error to watch for: - If both `a(long)` and `a(Integer)` are present, passing an `int` will always call `a(long)` due to widening preference.

---

15. Overloading: Widening vs Boxing with Floating Point

```
class Demo {
    void a(Long i) { System.out.println("Long"); }
    void a(double i) { System.out.println("double"); }
}
public class Main {
    public static void main(String[] args) {
        Demo demo = new Demo();
        demo.a(5L); // double
```

```
    }
}
```

Explanation: - `5L` is a `long`. There is no `a(long)` method. - Java can box `long` to `Long` or widen `long` to `double`. - **Widening to `double` is preferred over boxing to `Long`**, so `a(double)` is called.

Why: - Widening beats boxing in overload resolution.

---

16. Interface Default Method Ambiguity

```
interface T1 { default void m() { System.out.println("T1"); } }
interface T2 { void m(); }
class C implements T1, T2 {} // Error: class C inherits unrelated defaults for m() from type
```

Error: - "class C inherits unrelated defaults for m() from types T1 and T2"

Why: - If an interface provides a default method and another interface declares the same method (without default), the implementing class must provide its own implementation to resolve ambiguity.

How to fix:

```
class C implements T1, T2 {
    @Override
    public void m() { T1.super.m(); }
}
```

---

17. Abstract Class vs Interface Default Method

```
interface I { default void m() { System.out.println("I"); } }
abstract class A { abstract void m(); }
class D extends A implements I {} // Error: D is not abstract and does not override abstract
```

Error: - "D is not abstract and does not override abstract method m() in A"

Why: - Abstract class's abstract method takes precedence over interface default method. The subclass must implement the method.

How to fix:

```
class D extends A implements I {
    @Override
    public void m() { I.super.m(); }
}
```

---

18. Overloading with Null: Ambiguity

```
public class Main {
    void g(Integer x) { System.out.println("Integer"); }
    void g(Long x) { System.out.println("Long"); }
    public static void main(String[] args) {
        new Main().g(null); // Error: reference to g is ambiguous
    }
}
```

Why: - `null` matches both `Integer` and `Long` equally well, so the compiler cannot decide which method to call.

---

19. Interface Fields and Hiding

```
interface K { int X = 1; }
class Imp1 implements K { public int X = 2; }
public class Main {
    public static void main(String[] args) {
        K k = new Imp1();
        Imp1 imp1 = new Imp1();
        System.out.println(k.X + " " + imp1.X); // 1 2
    }
}
```

Explanation: - Interface fields are `public static final` by default. - The field in `Imp1` hides the interface field, but does not override it.

20-22. Overloading with Varargs and Widening

```
public class Main {
    static void f(long s) { System.out.println("long"); }
    static void f(int... s) { System.out.println("varargs"); }
    static void f(Integer... s) { System.out.println("varargs Integer"); }
    public static void main(String[] args) {
        byte n = 5;
        f(n); // long
        f((short)n); // long
    }
}
```

Explanation: - Widening is preferred over varargs. - `byte` and `short` are widened to `long`.

---

23. Method References and Overloading

```
import java.util.function.Function;
class S {
    String m(Object o) { return "O"; }
    String m(String s) { return "S"; }
}
public class Main {
    public static void main(String[] args) {
        Function<String, String> f = new S()::m;
        Function<Object, String> f2 = new S()::m;
        System.out.println(f.apply(null)); // S
        System.out.println(f2.apply(null)); // O
    }
}
```

Why: - The most specific method is chosen for the method reference type.

---

24. Sealed and Non-Sealed Interfaces: Ambiguity

```
sealed interface A permits B, C {}
non-sealed interface B extends A { default void hello() { System.out.println("Hello from B")
non-sealed interface C extends A { default void hello() { System.out.println("Hello from C")
class D implements B, C {}
public class Main {
    public static void main(String[] args) {
        new D().hello(); // Error: ambiguous
    }
}
```

Error: - "class D inherits unrelated defaults for hello() from types B and C"

How to fix: - Override `hello()` in `D` and specify which default to use.

---

25. Overriding and Narrowing Visibility

```
interface Box<T> { T get(); }
class Sbox implements Box<String> {
    protected String get() { return "X"; } // Error: Cannot reduce visibility
}
```

Error: - "attempting to assign weaker access privileges; was public"

Why: - Overriding methods cannot have more restrictive access than the interface method.

---

26. Static Initialization Order

```
class J {
    static { System.out.println("Init"); }
    static final int A = 1;
    static final Integer B = 2;
}
public class Main {
    public static void main(String[] args) {
        System.out.println(J.A); // 1
        System.out.println(J.B); // 2
    }
}
```

Order: - Static primitives are initialized first, then static blocks, then static reference variables.

---

27. Exception Overriding: Subclass Exception

```
import java.io.FileNotFoundException;
import java.io.IOException;
class A { void read() throws IOException { System.out.println("A"); } }
class B extends A { void read() throws FileNotFoundException { System.out.println("B"); } }
```

Why: - Overriding method can throw a subclass of the exception thrown by the base method.

28-29. Multi-Catch Variable is Final

```
public class Main {
    public static void main(String[] args) {
        try {
            throw new IllegalArgumentException();
        } catch (IllegalArgumentException | NullPointerException e) {
            // e = null; // Error: Variable in multi-catch cannot be assigned
            System.out.println("Caught: ");
        }
    }
}
```

Why: - Multi-catch variables are implicitly final to prevent confusion about which exception is being handled.

30-32. AutoCloseable, try-with-resources, and Suppressed Exceptions

```
class R implements AutoCloseable {
    @Override
    public void close() { throw new RuntimeException("close"); }
}
public class Main {
    public static void main(String[] args) {
        try (R r = new R()) {
            throw new RuntimeException("body");
        }
    }
}
```

Explanation: - `AutoCloseable` allows objects to be used in try-with-resources. - If both the try block and `close()` throw exceptions, the one from `close()` is suppressed.

How to see suppressed exceptions:

```
try (R r = new R()) {
    throw new RuntimeException("body");
} catch (RuntimeException e) {
    System.out.println(e.getMessage()); // body
    for (Throwable t : e.getSuppressed()) {
        System.out.println("Suppressed: " + t.getMessage()); // close
    }
}
```

---

33. Static Initializer Exception

```
public class Main {
    static { if (true) throw new RuntimeException("init"); }
    public static void main(String[] args) { System.out.println("main"); }
}
```

Output: - Exception in thread “main”: init - The main method is never reached if static initialization fails.

---

34. Lambda and Checked Exceptions

```java
import java.io.IOException;
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("a", "b");
        list.forEach(s -> {
            try {
                throw new IOException();
            } catch (IOException e) {
                System.out.println(s);
            }
        });
    }
}
```

Explanation: - Lambdas cannot throw checked exceptions unless handled inside the lambda. - Here, `IOException` is caught inside the lambda, so the code compiles and runs, printing each string.

Java Exception Handling MCQs

MCQ 1

```java
import java.io.*;

class A {
    void read() throws IOException { System.out.println("A"); }
}
class B extends A {
    @Override
    void read() throws FileNotFoundException { System.out.println("B"); }
}
public class TestE1 {
    public static void main(String[] args) throws IOException {
        A a = new B();
        a.read();
    }
}
```

Question: What happens?

MCQ 2

```
public class TestE2 {
    public static void main(String[] args) {
        try {
            if (true) throw new IllegalArgumentException();
        } catch (IllegalArgumentException | NullPointerException ex) {
            ex = null; // line X
            System.out.println("Caught");
        }
    }
}
```

Question: What happens?

MCQ 3

```
class R implements AutoCloseable {
    @Override
    public void close() {
        throw new RuntimeException("close");
    }
}

public class TestE3 {
    public static void main(String[] args) {
        try (R r = new R()) {
            throw new RuntimeException("body");
        }
    }
}
```

Question: What happens?

MCQ 4

```
public class TestE4 {
    static {
        if (true) throw new RuntimeException("init");
    }
    public static void main(String[] args) {
        System.out.println("Main");
```

```
    }
}
```

Question: What happens?

MCQ 5

```java
import java.util.*;

public class TestE5 {
    public static void main(String[] args) {
        List<String> list = List.of("a","b");
        list.forEach(s -> {
            try {
                throw new java.io.IOException();
            } catch (java.io.IOException e) {
                System.out.println(s);
            }
        });
    }
}
```

Question: What happens?

Java Interfaces MCQs

Question 1

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

class C implements A, B { }

public class Test1 {
  public static void main(String[] args) {
    new C().ping();
  }
}
```

What happens?

Question 2

```
interface Util {
  static String id() { return "I"; }
}

class X implements Util {
  static String id() { return "X"; }
}

public class Test2 {
  public static void main(String[] args) {
    Util u = new X();
    System.out.println(Util.id()); // line 1
    System.out.println(X.id());    // line 2
    System.out.println(u.id());    // line 3
  }
}
```

What happens at runtime/compile-time?

Question 3

```
interface Parser {
  default String sanitize(String s) { return trimAndLower(s); }
  private static String trimAndLower(String s) { return s.trim().toLowerCase(); }
}

class Impl implements Parser {
  String demo(String s) {
    return Parser.trimAndLower(s); // line X
  }
}

public class Test3 {
  public static void main(String[] args) {
    System.out.println(new Impl().sanitize("  HeLLo "));
  }
}
```

What happens?

Question 4

```
interface I { int X = 5; }

class T implements I {
  int X = 6;
  void show() { System.out.println(X + " " + I.X); }
}

public class Test4 {
  public static void main(String[] args) {
    new T().show();
  }
}
```

Output?

Question 5

```
interface I { void m(Number n); }

class A {
  public void m(Integer i) { System.out.println("A"); }
}

class B extends A implements I {
  @Override public void m(Number n) { System.out.println("B"); }
}

public class Test5 {
  public static void main(String[] args) {
    new B().m(10);          // prints B
    ((I) new B()).m(10);    // prints A
  }
}
```

Output?

Question 6

```
class Shape { }
class Circle extends Shape { }
interface Factory {
    Shape create();
```

```
}
class CircleFactory implements Factory {
    @Override
    public Circle create() {
        return new Circle();
    }
}
public class Test6 {
    public static void main(String[] args) {
        Factory f = new CircleFactory();
        Shape s = f.create();
        System.out.println(s.getClass().getSimpleName());
    }
}
```

What will be printed?

Question 7

```
import java.io.IOException;
import java.sql.SQLException;

abstract class A {
    abstract void run() throws IOException;
}
interface I {
    void run() throws SQLException;
}
class C extends A implements I {
    @Override
    public void run() { System.out.println("Run"); }
}
public class Test7 {
    public static void main(String[] args) {
        new C().run();
    }
}
```

What happens?

Question 8

```
sealed interface Animal permits Dog, Cat { }
```

```java
final class Dog implements Animal { }

non-sealed class Cat implements Animal { }

class Persian extends Cat { }

public class Test8 {
    public static void main(String[] args) {
        Animal a = new Persian();
        System.out.println(a.getClass().getSimpleName());
    }
}
```

What will be printed?

Question 9

```java
class P {
    static void hello() { System.out.println("P"); }
}

class Q extends P {
    static void hello() { System.out.println("Q"); }
}

public class Test9 {
    public static void main(String[] args) {
        P p = new Q();
        p.hello();
        Q q = new Q();
        q.hello();
    }
}
```

What will be printed?

Question 10

```java
interface Box<T> {
    T get();
}

class StringBox implements Box<String> {
    @Override
```

```
    public String get() { return "Hello"; }
}

public class Test10 {
    public static void main(String[] args) {
        Box b = new StringBox();
        System.out.println(b.get());
    }
}
```

What will be printed?

Question 11

```
class Overload {
    void test(String s) { System.out.println("String"); }
    void test(Object o) { System.out.println("Object"); }
}

public class Test11 {
    public static void main(String[] args) {
        new Overload().test(null);
    }
}
```

What will be printed?

Question 12

```
class Parent {
    void show(Number n) { System.out.println("Parent"); }
}

class Child extends Parent {
    void show(Integer i) { System.out.println("Child"); }
}

public class Test12 {
    public static void main(String[] args) {
        Parent p = new Child();
        p.show(10);
    }
}
```

What will be printed?

Question 13

```
class Base {
    public final void ping() { System.out.println("Base"); }
}

class Derived extends Base {
    public void ping() { System.out.println("Derived"); }
}

public class Test13 {
    public static void main(String[] args) {
        new Derived().ping();
    }
}
```

What happens?

Question 14

```
interface I {
    default void hello() { System.out.println("I"); }
}

class Super {
    public void hello() { System.out.println("Super"); }
}

class Sub extends Super implements I { }

public class Test14 {
    public static void main(String[] args) {
        new Sub().hello();
    }
}
```

What will be printed?

Question 15

```
class Demo {
    void run(long x) { System.out.println("long"); }
    void run(Integer x) { System.out.println("Integer"); }
}

public class Test15 {
    public static void main(String[] args) {
        int val = 5;
        new Demo().run(val);
    }
}
```

What will be printed?

Question 16

```
class Demo {
    static void f(Long x) { System.out.println("Long"); }
    static void f(double x) { System.out.println("double"); }
    public static void main(String[] args) {
        long v = 5L;
        f(v);
    }
}
```

What prints?

Question 17

```
interface I1 {
    default void m() { System.out.println("I1"); }
}
interface I2 {
    void m();
}
class C implements I1, I2 { }

public class Test17 {
    public static void main(String[] args) {
        new C().m();
    }
}
```

What happens?

Question 18

```
abstract class A {
    abstract void m();
}
interface I {
    default void m() { System.out.println("I"); }
}
class D extends A implements I { }

public class Test18 {
    public static void main(String[] args) {
        new D().m();
    }
}
```

What happens?

Question 19

```
class Over {
    void g(Integer x) { System.out.println("Integer"); }
    void g(Long x)    { System.out.println("Long"); }
    public static void main(String[] args) {
        new Over().g(null);
    }
}
```

What happens?

Question 20

```
interface K { int X = 1; }
class Impl implements K { public int X = 2; }

public class Test20 {
    public static void main(String[] args) {
        K k = new Impl();
        Impl i = new Impl();
        System.out.println(k.X + " " + i.X);
```

```
    }
}
```

What prints?

Question 21

```
class V {
    static void f(long x)        { System.out.println("long"); }
    static void f(int... xs)     { System.out.println("int..."); }
    static void f(Integer... xs) { System.out.println("Integer..."); }

    public static void main(String[] args) {
        int n = 5;
        f(n);
    }
}
```

What prints?

Question 22

```
import java.util.function.Function;

class S {
    String m(Object o) { return "O"; }
    String m(String s) { return "S"; }
}

public class Test22 {
    public static void main(String[] args) {
        Function<String, String> f = new S()::m;   // pick?
        Function<Object, String> g = new S()::m;   // pick?

        System.out.println(f.apply(null) + g.apply(null));
    }
}
```

Output?

Question 23

```
sealed interface A permits B, C {}

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

interface C extends A {
    default void hello() { System.out.println("C"); }
}

class D implements B, C { }  // no override

public class Test23 {
    public static void main(String[] args) {
        new D().hello();
    }
}
```

What happens?

Question 24

```
interface Box<T> {
    T get();
}

class SBox implements Box<String> {
    protected String get() { return "x"; } // line X
}

public class Test24 {
    public static void main(String[] args) {
        Box<String> b = new SBox();
        System.out.println(b.get());
    }
}
```

What happens?

Question 25

```
class J {
    static final int A = 1;
    static final Integer B = 1;
    static {
        System.out.println("init");
    }
}

public class Test25 {
    public static void main(String[] args) {
        System.out.println(J.A);
        System.out.println(J.B);
    }
}
```

What prints?

“‘