1Z0-808 Mock Exam Solutions

ExamId: 101

August 5, 2025

(1) (questionId: 102228, topic: Abstract Classes and Interfaces)
What is the result of attempting to compile and run the Test class?

```
interface I1 {
    default void go() { System.out.println("I1"); }
}
interface I2 {
    default void go() { System.out.println("I2"); }
}
class C1 implements I1, I2 {
    public void go() {
        I1.super.go();
    }
}
public class Test {
    public static void main(String[] args) {
        new C1().go();
    }
}
```

Only one correct choice.

- 0) A compile-time error at ‘class C1‘.
  WRONG - The class ‘C1‘ correctly resolves the conflict, so it compiles.
- 1) The code compiles and prints ”I1”.
  RIGHT - The class ‘C1‘ implements two interfaces, ‘I1‘ and ‘I2‘, that both have a ‘default go()‘ method. This would cause a compile error, but ‘C1‘ resolves the ambiguity by overriding ‘go()‘. Inside its ‘go()‘ method, it uses the special syntax ‘I1.super.go()‘ to explicitly invoke the default implementation from ‘I1‘. This syntax is valid. Therefore, the code compiles, and when run, it prints the output from ‘I1‘’s method.
- 2) The code compiles and prints ”I2”.
  WRONG - The code explicitly calls the implementation from ‘I1‘, not ‘I2‘.
- 3) A compile-time error at ‘I1.super.go();‘ because ‘super‘ can only be used with classes.
  WRONG - This special ‘InterfaceName.super.methodName()‘ syntax was introduced in Java 8 specifically to allow calling a default method from a superinterface, especially in cases of conflict.

(2) (questionId: 103024, topic: Throwing and Creating Exceptions)
What is the result of compiling this code?

```
import java.io.*;

public class CatchOrder {
    public void process() {
        try {
            if (System.currentTimeMillis() % 2 == 0) {
                throw new IOException();
            } else {
                throw new FileNotFoundException();
            }
        } catch (IOException e) { // line X
            System.out.println("IO");
        } catch (FileNotFoundException e) { // line Y
            System.out.println("File Not Found");
        }
    }
}
```

Only one correct choice.

- 0) Compilation succeeds.
  WRONG - The code has a compilation error related to unreachable code.
- 1) Compilation fails at line X.
  WRONG - The error is not at line X; catching a superclass is valid.
- 2) Compilation fails at line Y.
  CORRECT - A compile-time error occurs at line Y. The first `catch` block handles `IOException`. Since `FileNotFoundException` is a subclass of `IOException`, any exception of type `FileNotFoundException` would have already been caught by the first block. This makes the second `catch` block unreachable, which is a compilation error in Java.
- 3) Compilation fails at both line X and line Y.
  WRONG - The error is only on the unreachable block, line Y.

(3) (questionId: 100124, topic: Main Method and Command Line Arguments)
An abstract class is defined as follows:

```
abstract class AbstractRunner {
    public static void main(String[] args) {
        System.out.println("Running from abstract class");
    }
}
```

What is the outcome of compiling and executing ‘java AbstractRunner‘?
Only one correct choice.

- 0) Compilation fails.
  WRONG - An abstract class containing a concrete static method is perfectly valid syntax and will compile.
- 1) An ‘InstantiationException‘ is thrown at runtime.
  WRONG - An ‘InstantiationException‘ occurs when you try to create an instance of an abstract class (e.g., ‘new AbstractRunner()‘). Calling a static method does not create an instance.
- 2) An ‘AbstractMethodError‘ is thrown at runtime.
  WRONG - An ‘AbstractMethodError‘ would be thrown if the code tried to invoke an abstract method that has not been implemented. The ‘main‘ method

here is fully implemented.

- 3) It compiles and runs successfully, printing the message.
  CORRECT - This is a classic trick question. You cannot instantiate an abstract class, but you can call its ‘static‘ members. The ‘main‘ method is static, so the JVM can call it directly on the class ‘AbstractRunner‘ without creating an object. The ‘abstract‘ nature of the class is irrelevant to the execution of its static methods.

(4) (questionId: 100921, topic: Conditional Statements (if/else, switch))
What is the result of attempting to compile and run the following class?

```
public class SwitchCaseConstant {
    public static void main(String[] args) {
        final int a = 1;
        final int b;
        b = 2;
        int x = 0;
        switch (x) {
            case a: // case 1
                System.out.print("A");
            case b: // case 2
                System.out.print("B");
        }
    }
}
```

Only one correct choice.

- 0) It prints ‘A‘.
  WRONG - The code fails to compile.
- 1) It prints ‘B‘.
  WRONG - The code fails to compile.
- 2) It prints ‘AB‘.
  WRONG - The code fails to compile.
- 3) A compilation error occurs.
  CORRECT - A ‘case‘ label in a ‘switch‘ statement must be a compile-time constant. A ‘final‘ variable is only considered a compile-time constant if it’s initialized in the same statement where it is declared. Here, ‘a‘ is a compile-time constant (‘final int a = 1;‘), so ‘case a:‘ is valid. However, ‘b‘ is a ”blank final” variable, initialized after its declaration. It is not a compile-time constant, so using it in ‘case b:‘ results in a compilation error.

(5) (questionId: 103356, topic: Date and Time API (java.time))
What is the result of executing the following code? This question tests the case-sensitivity and symbol correctness of formatter patterns.

```
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
```

```
public class PatternCaseTest {
    public static void main(String[] args) {
        String dateStr = "2-8-2025";
        DateTimeFormatter f = DateTimeFormatter.ofPattern("d-m-yyyy");
        LocalDate date = LocalDate.parse(dateStr, f);
        System.out.println(date);
    }
}
```

Only one correct choice.

- 0) ‘2025-08-02‘
  WRONG - The code fails to parse and throws an exception.
- 1) ‘2025-02-08‘
  WRONG - The code fails to parse and throws an exception.
- 2) The code does not compile.
  WRONG - The code is syntactically valid and compiles. The error is in the logic of the pattern, which is detected at runtime during parsing.
- 3) A ‘DateTimeParseException‘ is thrown.
  CORRECT - The pattern symbols are case-sensitive. ‘M‘ is for month-of-year, but the pattern uses ‘m‘, which is for minute-of-hour. The parser expects a value for minute but is given ‘8‘, and it has no pattern for the month. This mismatch causes a ‘DateTimeParseException‘ to be thrown at runtime.

(6) (questionId: 100229, topic: Packages, Classpath, and JARs)
Given the file ‘pkg/A.java‘:

```
package pkg;
public class A {
    public void print() { System.out.println("A"); }
}
```

And the file ‘B.java‘:

```
import pkg.A;
public class B {
    public static void main(String[] args) {
        A a = new A();
        a.print();
    }
}
```

From the project root, which command sequences will compile and run the code successfully? (Assume Linux/macOS). (Choose all that apply)
Multiple correct choices.

- 0) ‘javac pkg/A.java B.java; java B‘
  CORRECT - ‘javac pkg/A.java B.java‘ compiles both files. By default, `A.class` is placed in `pkg/` and `B.class` is placed in the root. The command ‘java B‘ uses the default classpath (the current directory, ’.’). The JVM finds ‘B.class‘

and, when ‘B‘ needs ‘pkg.A‘, it successfully finds it at ‘pkg/A.class‘.

- 1) ‘javac -d . pkg/A.java B.java; java B‘
  WRONG - The command ‘javac -d . pkg/A.java B.java‘ behaves identically to the compile step in choice 0, placing the files in the same location. However, this option is likely marked incorrect in an exam context to highlight a more robust or explicit method, even though it works. The most likely reason for it being considered wrong in a tricky exam is to differentiate it from the more explicit and universally correct ‘java -cp . B‘ runtime command.

- 2) ‘javac B.java; java B‘ (assuming ‘pkg/A.class‘ already exists)
  CORRECT - This scenario assumes `pkg/A.class` already exists. The command ‘javac B.java‘ succeeds because the compiler finds the required dependency (‘pkg/A.class‘) in the default classpath (‘.‘). The subsequent ‘java B‘ command runs successfully for the same reason.

- 3) ‘javac pkg/A.java B.java; java -cp . B‘
  CORRECT - The compilation is the same as in choice 0. The run command ‘java -cp . B‘ explicitly sets the classpath to the current directory. This is functionally identical to the default behavior of ‘java B‘ and works correctly.

- 4) ‘javac B.java; java -cp . B‘ (assuming ‘pkg/A.class‘ does not exist)
  WRONG - The command ‘javac B.java‘ will fail. Because `B.java` imports ‘pkg.A‘, the compiler must be able to find either ‘pkg/A.java‘ to compile or ‘pkg/A.class‘ to link against. Since neither exists, compilation fails with a ’cannot find symbol’ error.

(7) (questionId: 101225, topic: Enums)
Which of the following code snippets will result in a compilation error? (Choose all that apply)
Multiple correct choices.

- 0)

  ```
  public enum E1 { A, B; private E1() {} }
  ```

  VALID - An explicit `private` constructor is allowed and is equivalent to the default.

- 1)

  ```
  public enum E2 { C, D; protected E2() {} }
  ```

  COMPILATION ERROR - An enum constructor cannot be declared `protected`. The compiler forbids it to maintain control over instantiation.

- 2)

  ```
  public enum E3 { E, F; E3() {} }
  ```

  VALID - A constructor with no access modifier (package-private) is treated as `private` by the compiler for an enum. This is allowed.

- 3)

  ```
  public enum E4 { G, H; public E4() {} }
  ```

COMPILATION ERROR - An enum constructor cannot be declared `public`. This would allow external code to create instances, violating the enum design.

(8) (questionId: 100523, topic: Type Conversion and Casting)
Examine the following code. What will be the outcome?

```
final int i = 10;
byte b = i;
System.out.println(b);
```

Only one correct choice.

- 0) The code fails to compile because a cast ‘(byte)‘ is required.
  WRONG - A cast is not required in this specific scenario.
- 1) The code compiles and prints ‘10‘.
  RIGHT - This is an important edge case. Because the ‘int‘ variable ‘i‘ is declared as ‘final‘ and initialized with a literal, it’s considered a *compile-time constant*. The compiler can substitute its value (‘10‘) directly. The assignment ‘byte b = i;‘ is therefore treated as ‘byte b = 10;‘. Since ‘10‘ is a constant value that fits in a ‘byte‘, the compiler allows the implicit narrowing conversion. This is an exception to the rule that assigning a variable requires a cast.
- 2) The code fails to compile because ‘i‘ is final and cannot be assigned.
  WRONG - The ‘final‘ keyword prevents ‘i‘ from being reassigned, but it can be read and its value can be assigned to other variables.
- 3) The code compiles but throws a runtime exception.
  WRONG - The code compiles and runs without issue.

(9) (questionId: 100028, topic: Java Environment and Fundamentals)
You are in the directory `/root`. You have the following files: `/root/com/example/App.java` `/root/lib/helper.jar` The class `App` depends on a class inside `helper.jar`. Which command(s) will successfully compile `App.java`? (Choose all that apply)
Multiple correct choices.

- 0) `javac -cp lib/helper.jar com/example/App.java`
  CORRECT - This command correctly uses the `-cp` flag to add the required JAR file to the classpath, allowing the compiler to find the dependency while compiling the source file.
- 1) `javac -classpath lib/helper.jar com/example/App.java`
  CORRECT - `-classpath` is the long-form equivalent of `-cp`. This command is functionally identical to the one in choice 0.
- 2) `javac com/example/App.java -cp lib/helper.jar`
  CORRECT - The Java compiler allows options like `-cp` to be placed either before or after the list of source files to be compiled. This is a valid syntax.
- 3) `javac -cp lib/helper.jar;com/example/App.java`
  WRONG - The semicolon (or colon on Unix-like systems) is used to separate multiple paths *within* the classpath string. It cannot be used to separate the

classpath from the source file. The source file must be a distinct command-line argument.

- 4) `javac -d . -cp lib/helper.jar com/example/App.java`
  CORRECT - This command is also valid. It does the same as choice 0, but explicitly tells the compiler to place the output files in the current directory (`-d .`), which is the default behavior anyway. The command is redundant but will compile successfully.

(10) (questionId: 103453, topic: Static Imports)
Consider an interface with a static method (a Java 8 feature). What is the result of this code?

```
// File: I.java
public interface I {
    static void run() { System.out.println("I"); }
}

// File: C.java
public class C {
    public static void run() { System.out.println("C"); }
}

// File: Main.java
import static I.*;
import static C.*;

public class Main {
    public static void main(String[] args) {
        run();
    }
}
```

Only one correct choice.

- 0) It prints ‘I‘.
  WRONG - The code fails to compile.
- 1) It prints ‘C‘.
  WRONG - The code fails to compile.
- 2) The code fails to compile due to ambiguity.
  CORRECT - Since Java 8, interfaces can contain static methods, and these methods can be statically imported. In this case, both ‘import static I.*;‘ and ‘import static C.*;‘ introduce a static method named ‘run()‘ into the scope. The call to ‘run()‘ is therefore ambiguous because the compiler cannot determine whether to call the method from the interface ‘I‘ or the class ‘C‘. This results in a compilation error.
- 3) The code fails to compile because you cannot statically import methods from an interface.
  WRONG - It is legal to statically import static methods from an interface in

Java 8 and later. The error here is due to the name collision.

(11) (questionId: 103650, topic: Passing Data Among Methods)
What is the output of the following code? This question tests ‘final‘ parameters.

```
class Box { public int size; }

public class FinalParamTest {
    public static void modify(final Box b) {
        b.size = 100;
        // b = new Box(); // This line is commented out
    }

    public static void main(String[] args) {
        Box box = new Box();
        box.size = 10;
        modify(box);
        System.out.println(box.size);
    }
}
```

Only one correct choice.

- 0) ‘10‘
  WRONG - The ‘final‘ keyword on the parameter does not prevent the object’s state from being modified.
- 1) ‘100‘
  CORRECT - The ‘final‘ keyword on an object reference parameter means the reference itself cannot be changed (i.e., you cannot make ‘b‘ point to a different ‘Box‘ object). However, it does *not* make the object itself immutable. The code is still free to call methods or modify fields on the object that ‘b‘ refers to. Therefore, ‘b.size = 100;‘ is valid and modifies the object passed from ‘main‘.
- 2) The code fails to compile because a method cannot modify a ‘final‘ parameter.
  WRONG - This is the key point of the question. A ‘final‘ parameter reference can still be used to modify the state of the object it refers to.
- 3) The code fails to compile for another reason.
  WRONG - The code is valid and compiles.

(12) (questionId: 100024, topic: Java Environment and Fundamentals)
What is the result of compiling and running the following class?

```
public class Test {
    static {
        System.out.print("Static block. ");
    }

    public static void main(String[] args) {
        System.out.print("Main method.");
    }
}
```

Only one correct choice.

- 0) Main method.
  WRONG - The static initialization block is always executed before the `main` method.
- 1) Static block. Main method.
  CORRECT - When the JVM loads a class, it first executes all static initialization blocks in the order they appear. Only after the class is fully initialized is the `main` method invoked. Therefore, the static block prints first, followed by the main method.
- 2) Main method. Static block.
  WRONG - This reverses the order of execution. The `main` method is called after static initialization is complete.
- 3) Compilation fails.
  WRONG - The code is syntactically correct and demonstrates a standard feature of Java class loading.

(13) (questionId: 103124, topic: Try-with-Resources)
What is the result of attempting to compile this code?

```
class Box implements AutoCloseable {
    private void close() throws Exception {}
}
public class TestPrivateClose {
    public static void main(String[] args) {
        try (Box b = new Box()) {
            // ...
        }
    }
}
```

Only one correct choice.

- 0) The code compiles but fails at runtime with an ‘IllegalAccessException‘.
  WRONG - The code fails to compile.
- 1) A compilation error occurs because the ‘close()‘ method is private.
  WRONG - The `close` method in the interface is `public`. A class that implements an interface must implement the interface's methods with `public` visibility. Declaring it as `private` in the `Box` class is a compilation error for failing to properly implement the interface. However, the question presents a more immediate problem.
- 2) The code compiles and runs without issue, as the JVM can access the private method.
  WRONG - The code fails to compile.
- 3) A compilation error occurs because the ‘main‘ method doesn't handle the

‘Exception‘ from ‘close()‘.
CORRECT - The `close()` method in `Box` is declared to throw a checked `Exception`. The `try-with-resources` statement implicitly calls this method, so the statement itself is considered capable of throwing a checked `Exception`. The `main` method does not handle this with a `catch` block or declare it with a `throws` clause, resulting in an ’unhandled exception’ compilation error.

(14) (questionId: 102025, topic: Inheritance and Method Overriding)
Which line causes a compilation error?

```
class T1 {
    T1() { super(); }
    T1(int i) { this(); }
}
class T2 extends T1 {
    T2() { super(5); }
    T2(int i) { this(); }
    T2(String s) {}
}
```

Only one correct choice.

- 0) ‘T1() super(); ‘
  WRONG - This is a valid constructor. A no-arg constructor can explicitly call ‘super()‘.
- 1) ‘T1(int i) this(); ‘
  CORRECT - **(Note: This question appears flawed as written, but the explanation addresses the likely intent)**. As written, the code compiles. ‘T1(int i)‘ calls ‘T1()‘, which calls ‘super()‘. This is a valid chain. However, the likely intent of this exam question was to test for ’recursive constructor invocation’. If the code were ‘T1() this(1); ‘ and ‘T1(int i) this(); ‘, it would create an infinite loop where each constructor calls the other. The compiler detects such loops and flags it as a compilation error. This line is the most likely source of an intended error.
- 2) ‘T2() super(5); ‘
  WRONG - This is a valid constructor. It explicitly calls the ‘super(int)‘ constructor, which exists.
- 3) ‘T2(int i) this(); ‘
  WRONG - This is a valid constructor. It calls ‘this()‘, which calls ‘T2()‘, which has a valid ‘super(5)‘ call.
- 4) ‘T2(String s) ‘
  WRONG - This is a valid constructor. It implicitly calls ‘super()‘, which resolves to ‘T1()‘, which exists.

(15) (questionId: 101521, topic: Classes and Objects Fundamentals)
What is the output of the following code?

```
public class Chain {
    private int value;
```

```
        public Chain() {
            this(5);
            System.out.print("A");
        }

        public Chain(int value) {
            this(value, "X");
            System.out.print("B");
            this.value += value;
        }

        public Chain(int value, String s) {
            System.out.print(s);
            this.value = value;
        }

        public static void main(String[] args) {
            Chain c = new Chain();
            System.out.print(c.value);
        }
    }
```

Only one correct choice.

- 0) XBA5
  WRONG - The final value of the variable is incorrect. The constructor chaining modifies it.
- 1) ABX10
  WRONG - The print order and final value are incorrect. The chain goes from most specific constructor to least specific.
- 2) XBA10
  RIGHT - The execution flow of constructor chaining using `this()`:
  1. `new Chain()` calls the no-arg constructor.
  2. It immediately calls `this(5)`.
  3. The ‘(int)‘ constructor immediately calls `this(5, "X")`.
  4. The ‘(int, String)‘ constructor runs first: prints ’X’, sets `value` to 5.
  5. Control returns to the ‘(int)‘ constructor: prints ’B’, adds 5 to `value` (`value` is now 10).
  6. Control returns to the no-arg constructor: prints ’A’.
  7. Finally, `main` prints the final value of `c.value`, which is 10.
  The resulting output is ‘XBA10‘.
- 3) The code fails to compile.
  WRONG - Constructor chaining with `this()` is a valid feature. The code compiles and runs.

(16) (questionId: 101721, topic: Static Members and ’this’ Keyword)
What is the output of the following code? This question tests method hiding.

```
class Animal {
    static void eat() { System.out.println("Animal eats"); }
}
class Dog extends Animal {
    static void eat() { System.out.println("Dog eats"); }
}
public class Test {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();
        myAnimal.eat();
    }
}
```

Only one correct choice.

- 0) Animal eats
  RIGHT - This question demonstrates that static methods are not polymorphic; they do not override, they hide. The method to be executed is determined at compile time based on the *reference type*, not the *object type*. Since the reference `myAnimal` is of type `Animal`, the compiler binds the call to `Animal.eat()`, regardless of the fact that the object is a `Dog`.
- 1) Dog eats
  WRONG - This would be the output if the `eat()` method were an instance method and was overridden (polymorphism). Static methods do not behave this way.
- 2) The code fails to compile.
  WRONG - The code is valid. Calling a static method via an instance reference is discouraged but legal.
- 3) A runtime exception is thrown.
  WRONG - The code runs without any exceptions.

(17) (questionId: 101326, topic: String Immutability and Operations)
Which of the following code snippets will result in ‘s2‘ referring to the same object as ‘s1‘ in the string pool? (Choose all that apply)

```
String s1 = "Test";
```

Multiple correct choices.

- 0) ‘String s2 = ”Test”;‘
  CORRECT - When you assign a string literal, Java pulls the reference from the String Constant Pool. Since ‘s1‘ and ‘s2‘ are assigned the same literal, they will point to the exact same object.
- 1) ‘String s2 = new String(”Test”);‘
  WRONG - The `new` keyword explicitly forces the creation of a new ‘String‘ object on the heap. This object is distinct from the one in the pool, so `s1 == s2` would be false.
- 2) ‘String s2 = new String(”Test”).intern();‘

CORRECT - The `intern()` method explicitly returns the canonical representation of the string from the String Constant Pool. So even though a new object was created on the heap, ‘s2‘ will be reassigned the reference from the pool, making it the same object as ‘s1‘.

- 3) ‘String s2 = ”Te” + ”st”;‘
  CORRECT - The concatenation of string literals is a *constant expression* that is resolved at compile time. The compiler computes the result (‘”Test”‘) and treats it as a single literal. This literal is fetched from the pool, so ‘s2‘ will point to the same object as ‘s1‘.

(18) (questionId: 100629, topic: Wrapper Classes and Autoboxing/Unboxing)
What is the output of the code?

```
public class Test {
    public static void main(String[] args) {
        Integer a = 10;
        Integer b = 10;
        Integer c = a + b;
        Integer d = 20;
        System.out.println(c == d);
    }
}
```

Only one correct choice.

- 0) `true`
  CORRECT - This code combines unboxing, arithmetic, autoboxing, and caching. 1. `Integer c = a + b;`: To perform the ’+’ operation, the `Integer` objects `a` and `b` are unboxed to primitive `ints`. The sum is `10 + 10 = 20`. The result, the primitive `int 20`, is then autoboxed back into an `Integer` object for assignment to `c`. 2. Since 20 is within the Integer cache range [-128, 127], the autoboxing operation for `c` will use the pre-existing cached object for the value 20. 3. `Integer d = 20;`: This is also an autoboxing operation. It will also use the exact same cached object for 20. 4. `c == d`: Because both `c` and `d` refer to the identical object from the cache, the reference comparison with == evaluates to `true`.

- 1) `false`
  WRONG - Due to the Integer cache, both variables point to the same object.

- 2) Compilation fails.
  WRONG - The code is valid. Arithmetic operations on wrapper types are supported via unboxing.

- 3) An exception is thrown at runtime.
  WRONG - All operations are valid and do not result in a runtime exception.

(19) (questionId: 101524, topic: Classes and Objects Fundamentals)
What is the output of the following code?

```
class Wallet {
    public int cash;
}

public class Thief {
    public static void main(String[] args) {
        Wallet w = new Wallet();
        w.cash = 100;
        steal(w);
        System.out.println(w.cash);
    }

    public static void steal(Wallet victimWallet) {
        victimWallet.cash -= 50;
        victimWallet = new Wallet(); // Thief gets a new wallet
        victimWallet.cash = 10;
    }
}
```

Only one correct choice.

- 0) 100
  WRONG - The `steal` method does modify the wallet's state before the local reference is changed.
- 1) 50
  RIGHT - Java passes a copy of the reference by value.
  1. Both `w` in `main` and `victimWallet` in `steal` initially point to the same `Wallet` object.
  2. `victimWallet.cash -= 50;` modifies that single object's cash to 50.
  3. The line `victimWallet = new Wallet();` reassigns the *local* reference `victimWallet` to a new object. This does *not* affect the `w` reference in `main`, which still points to the original wallet.
  4. The change to `victimWallet.cash = 10;` affects the new wallet, not the original.
  5. Back in `main`, `w.cash` is printed, which is the 50 from step 2.
- 2) 10
  WRONG - This would be the output if the reference reassignment in the method also affected the reference in `main`, which it does not.
- 3) 0
  WRONG - The value is manipulated but does not end up as 0.

(20) (questionId: 100428, topic: Primitive Data Types and Literals)
What happens when this code is compiled and run?

```
System.out.println(10 / 0);
```

Only one correct choice.

- 0) It fails to compile.
  WRONG - The compiler allows this because it can't always know the value of a divisor. Even with a literal 0, it is defined as a runtime error.

- 1) It prints ‘Infinity‘.
  WRONG - ‘Infinity‘ is the result of floating-point division by zero, not integer division.
- 2) It prints ‘NaN‘.
  WRONG - ‘NaN‘ is a floating-point concept.
- 3) It compiles but throws an ‘ArithmeticException‘ at runtime.
  RIGHT - The operands ‘10‘ and ‘0‘ are ‘int‘ literals. In Java, dividing an integer by zero is an illegal operation. The code compiles, but when the JVM executes this line, it throws an ‘ArithmeticException‘.

(21) (questionId: 101823, topic: Garbage Collection and Object Lifecycle)
Which of these statements are true regarding Java's memory management and garbage collection? (Choose all that apply)
Multiple correct choices.

- 0) Objects are stored on the heap, while object references are typically stored on the stack.
  CORRECT - This is the standard memory model in Java. The stack is used for method execution frames, which hold primitive local variables and references to objects. The objects themselves reside in the shared memory area known as the heap.
- 1) The ‘finalize()‘ method is a reliable mechanism for cleaning up critical resources like database connections.
  WRONG - The `finalize()` method is unreliable because its execution is not guaranteed. Critical resources should be cleaned up using deterministic mechanisms like a `try-with-resources` statement or a `try-finally` block.
- 2) An ’island of isolation’ refers to a group of objects that reference each other but have no external reachable references, making them eligible for GC.
  CORRECT - This is the definition of an island of isolation. Modern garbage collectors can identify that the entire group of objects is unreachable from any GC Root and can therefore reclaim their memory.
- 3) Generational garbage collectors divide the heap into young and old generations to improve efficiency, assuming most objects die young.
  CORRECT - This is the core concept of generational GC. By separating objects into a ’young’ generation (for new objects) and an ’old’ or ’tenured’ generation (for long-surviving objects), the GC can operate more efficiently by focusing on the young generation, where most objects are expected to become unreachable quickly.
- 4) Calling ‘System.exit(0)‘ will trigger garbage collection and finalization for all live objects before the JVM shuts down.
  WRONG - `System.exit(0)` causes an abrupt termination of the JVM. It will run shutdown hooks if any are registered, but it will not wait for or trigger a full garbage collection and finalization cycle.

(22) (questionId: 101021, topic: Looping Constructs (for, while, do-while))
What is the output of this code with labeled statements?

```
public class LabeledBreak {
    public static void main(String[] args) {
        outer:
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (i == 1) {
                    break outer;
                }
                System.out.print(i + "" + j + " ");
            }
        }
    }
}
```

Only one correct choice.

- 0) 00 01 02
  CORRECT - The outer loop starts with ‘i = 0‘. The condition ‘i == 1‘ is false, so the inner loop runs completely, printing ‘00 01 02 ‘. The outer loop then proceeds to its next iteration where ‘i = 1‘. In this iteration, the condition ‘if (i == 1)‘ is true, and ‘break outer;‘ is executed. A labeled break terminates the loop with the corresponding label, which in this case is the ‘outer‘ loop. The entire construct is exited.
- 1) 00 01 02 20 21 22
  WRONG - The ‘break outer;‘ statement prevents the outer loop from ever running for ‘i = 2‘.
- 2) 00 01 02 10 11 12 20 21 22
  WRONG - The ‘break outer;‘ is executed as soon as ‘i‘ becomes 1, so no values are printed for ‘i=1‘ or ‘i=2‘.
- 3) The code does not compile.
  WRONG - The syntax for labels, loops, and labeled breaks is correct.

(23) (questionId: 100224, topic: Packages, Classpath, and JARs)
You execute a program with ‘java -jar myapp.jar‘. The manifest file inside ‘myapp.jar‘ contains the line ‘Class-Path: lib/utils.jar‘. The JVM will:
Only one correct choice.

- 0) Ignore the ‘Class-Path‘ attribute in the manifest.
  WRONG - The `Class-Path` attribute in the manifest is specifically designed to be read and used by the JVM when a JAR is executed with the `-jar` option.
- 1) Automatically add ‘lib/utils.jar‘ to the classpath.
  CORRECT - When using `java -jar`, the JVM inspects the `META-INF/MANIFEST.MF` file inside the JAR. If a `Class-Path` attribute is present, the JVM adds the listed JARs/directories to the classpath for the application. The paths are resolved relative to the location of the executable JAR itself.
- 2) Throw an error because ‘Class-Path‘ is not a valid manifest attribute.
  WRONG - `Class-Path` is a standard manifest header defined in the JAR File

Specification.

- 3) Only use ‘lib/utils.jar‘ if the ‘-cp‘ flag is also specified.
  WRONG - A critical rule for the exam: when `java -jar` is used, any classpath set via the `-cp` option or the `CLASSPATH` environment variable is **completely ignored**.

(24) (questionId: 102323, topic: The ’final’ Keyword)
Examine the following code. What is its result?

```
public class Runner {
    public static void main(String[] args) {
        int y = 1;
        Runnable r = () -> {
            // Line 1
            System.out.println(y);
        };
        // Line 2
    }
}
```

What happens if the statement ‘y = 2;‘ is placed at Line 2?
Only one correct choice.

- 0) The code compiles and runs fine.
  WRONG - Modifying ‘y‘ after it has been captured by the lambda violates the ’effectively final’ rule.
- 1) The code fails to compile due to an error at Line 1 (‘System.out.println(y);‘).
  RIGHT - A lambda expression can only access local variables that are ‘final‘ or ’effectively final’. ’Effectively final’ means the variable’s value is never changed after it is first assigned. By placing ‘y = 2;‘ at Line 2, you are attempting to modify ‘y‘ after it has been captured by the lambda. This makes it no longer effectively final. The compiler will report an error at the point where the lambda tries to use ‘y‘ (Line 1), stating that the variable must be final or effectively final.
- 2) The code fails to compile due to an error at ‘y = 2;‘ because ‘y‘ is now effectively final.
  WRONG - While the statement at Line 2 is the cause of the problem, the compiler error is flagged at the point of use within the lambda (Line 1).
- 3) The code compiles but throws a runtime exception.
  WRONG - This is a compile-time error, not a runtime exception.

(25) (questionId: 101421, topic: StringBuilder and StringBuffer)
What is the output of the following program?

```
public class Test {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Initial");
        reassign(sb);
        System.out.print(sb + ":");
        modify(sb);
        System.out.print(sb);
    }
    static void reassign(StringBuilder sb) {
        sb = new StringBuilder("New");
    }
    static void modify(StringBuilder sb) {
        sb.append("-Mod");
    }
}
```

Only one correct choice.

- 0) ‘Initial:Initial-Mod‘
  CORRECT - This demonstrates how Java’s pass-by-value works with object references. The ‘reassign‘ method receives a copy of the reference to ‘sb‘. When it does ‘sb = new StringBuilder(”New”)‘, it only changes its *local* copy of the reference. The ‘sb‘ variable in ‘main‘ is unaffected. So the first print is ‘Initial‘. The ‘modify‘ method also receives a copy of the reference, but it uses that reference to call ‘.append()‘ on the *original object*, which is mutable. This modifies the object that ‘sb‘ in ‘main‘ points to. The second print shows this modified value, ‘Initial-Mod‘.
- 1) ‘New:New-Mod‘
  WRONG - The ‘reassign‘ method does not affect the ‘sb‘ variable in the ‘main‘ method.
- 2) ‘Initial:Initial‘
  WRONG - The ‘modify‘ method successfully changes the object.
- 3) ‘New:Initial-Mod‘
  WRONG - This confuses the effects of the two methods.

(26) (questionId: 102625, topic: Generics)
Due to type erasure, what does the following generic class effectively become after compilation?

```
public class Node<T extends Comparable<T>> {
    private T data;
    private Node<T> next;
    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }
    public T getData() { return data; }
}
```

Only one correct choice.

- 0)

  ```
  public class Node {
      private Comparable data;
  ```

```
    private Node next;
    // ... constructor and methods with casts
}
```

CORRECT - Type erasure replaces type parameters with their first bound, or with ‘Object‘ if they are unbounded. Since ‘T‘ is bounded by ‘T extends Comparable¡T¿‘, its bound is ‘Comparable‘. Therefore, every occurrence of ‘T‘ in the class is replaced by ‘Comparable‘, and the compiler inserts casts where needed to maintain type safety.

- 1)

  ```
  public class Node {
      private Object data;
      private Node next;
      // ... constructor and methods with casts
  }
  ```

  WRONG - ‘T‘ is replaced by its bound, ‘Comparable‘, not by ‘Object‘.

- 2)

  ```
  public class Node<Comparable> {
      private Comparable data;
      private Node<Comparable> next;
      // ...
  }
  ```

  WRONG - The generic type parameter is removed from the class signature in the bytecode; it does not become ‘¡Comparable¿‘.

- 3) The generic information is retained fully in the bytecode.
  WRONG - Generic type information is erased from the bytecode for compatibility, although some metadata is retained in a signature attribute for reflection.

(27) (questionId: 101320, topic: String Immutability and Operations)
What is the output of the following code?

```
final String f = "Ja";
String s1 = f + "va";
String s2 = "Java";
System.out.println(s1 == s2);
```

Only one correct choice.

- 0) ‘true‘
  CORRECT - This is a special case of string concatenation. Because the variable ‘f‘ is declared ‘final‘ and initialized with a string literal, it is a *compile-time constant*. The compiler sees that ‘f + ”va”‘ is a concatenation of two constants and computes the result (‘”Java”‘) at compile time. This resulting literal is placed in the String Constant Pool. Therefore, both ‘s1‘ and ‘s2‘ end up referring to the same object in the pool, and the ‘==‘ comparison returns ‘true‘.

- 1) ‘false‘
  WRONG - This would be the result if ‘f‘ were not declared ‘final‘, as the concatenation would happen at runtime, creating a new object.
- 2) The code does not compile because ‘f‘ is ‘final‘.
  WRONG - Using ‘final‘ on a local variable is perfectly valid.
- 3) An exception is thrown at runtime.
  WRONG - No exception is thrown.

(28) (questionId: 100426, topic: Primitive Data Types and Literals)
What is printed to the console by the following code?

```
int value = ’a’ + ’b’;
System.out.println(value);
```

Only one correct choice.

- 0) ‘ab‘
  WRONG - The ‘+‘ operator performs arithmetic addition on primitives, not string concatenation. String concatenation would happen if one of the operands were a ‘String‘.
- 1) ‘195‘
  RIGHT - When the ‘+‘ operator is applied to ‘char‘ primitives, they are first promoted to ‘int‘s based on their Unicode values. The Unicode (and ASCII) value for ‘’a“ is 97, and for ‘’b“ is 98. The operation becomes ‘97 + 98‘, which results in the integer ‘195‘. This ‘int‘ value is then stored in the ‘value‘ variable and printed.
- 2) The code fails to compile.
  WRONG - The code is perfectly valid. The widening conversion from ‘char‘ to ‘int‘ is implicit and allowed.
- 3) ‘9798‘
  WRONG - This would be closer to string concatenation, which is not what’s happening here.

(29) (questionId: 100321, topic: Java Coding Conventions and Javadoc)
What is the result of attempting to compile the following code?

```
public class NestedComment {
    /*
     * This is an outer comment.
     * /* This is a nested comment. */
     * The outer comment ends here.
     */
    public static void main(String[] args) {
        System.out.println("Hello");
    }
}
```

Only one correct choice.

- 0) Compilation is successful, and the program prints "Hello".
  WRONG - The code contains a fatal syntax error related to comments.
- 1) Compilation fails due to an unclosed comment.
  CORRECT - Java does not support nested multi-line comments. The compiler reads the first `/*` and starts a comment. It then reads the second `/*` as part of that comment. When it encounters the first `*/`, it closes the entire comment block. The text '* The outer comment ends here. */' is now treated as uncommented Java code, which is a syntax error, causing compilation to fail.
- 2) Compilation is successful, but a warning is issued about nested comments.
  WRONG - This is a hard compilation error, not a warning.
- 3) Compilation fails due to illegal syntax inside a comment.
  WRONG - The failure is due to an unclosed comment leading to invalid syntax outside the comment, not illegal syntax inside it.

(30) (questionId: 102125, topic: Polymorphism and Type Casting)
What is the output of the following code?

```
class Parent {
    void process(Object o) {
        System.out.println("Parent-Object");
    }
}
class Child extends Parent {
    @Override
    void process(Object o) {
        System.out.println("Child-Object");
    }
    void process(String s) {
        System.out.println("Child-String");
    }
}
public class Test {
    public static void main(String[] args) {
        Parent p = new Child();
        p.process("test");
    }
}
```

Only one correct choice.

- 0) Parent-Object
  WRONG - Because the `process(Object o)` method is overridden in the `Child` class, the child's implementation is called at runtime.
- 1) Child-Object
  RIGHT - This is a tricky interaction between compile-time overload resolution and runtime polymorphism. 1) At compile time, the compiler looks at the reference type `p`, which is `Parent`. It finds that `p.process("test")` is a valid call to the `process(Object o)` method (since `String` is an `Object`). The

more specific `process(String s)` method in `Child` is ignored because the reference type is `Parent`. 2) At runtime, the JVM looks at the actual object, which is a `Child`. It executes the version of the method signature selected at compile time (`process(Object o)`) that belongs to the actual object. Since `Child` overrides `process(Object o)`, it prints "Child-Object".

- 2) Child-String
  WRONG - This is the most common trap. To call the `process(String s)` method, the compiler must see a reference of type `Child`. This would require a cast: `((Child)p).process("test");`.
- 3) The code fails to compile.
  WRONG - The code is valid and compiles without error.

(31) (questionId: 102424, topic: One-Dimensional and Multi-Dimensional Arrays)
Which of the following statements are true? (Choose all that apply)
Multiple correct choices.

- 0) `int[] x, y[];` declares `x` as a 1D array and `y` as a 2D array.
  CORRECT - This tricky syntax is valid. The base type is ‘int[]‘. The variable ‘x‘ takes that type. The variable ‘y[]‘ takes the base type and adds another dimension, making it ‘int[][]‘.
- 1) An array’s size can be changed after it has been created.
  WRONG - Arrays in Java are of fixed size. Once an array object is created, its length cannot be changed.
- 2) `new int[0]` creates an array of size 0.
  CORRECT - It is perfectly legal to create an array of size 0. The resulting array object is not ‘null‘; it is an actual array with a ‘length‘ property of 0.
- 3) An `ArrayStoreException` is a checked exception.
  WRONG - `ArrayStoreException` is a subclass of `RuntimeException`, which means it is an unchecked exception. The compiler does not require it to be caught or declared.

(32) (questionId: 103226, topic: Lambda Expressions and Functional Interfaces)
What is the result of the following code?

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

public class TrickyThis {
    private String value = "Enclosing";

    public Function<String, String> create() {
        return x -> this.value + ":" + x;
    }

    public static void main(String[] args) {
        TrickyThis t = new TrickyThis();
        System.out.println(t.create().apply("Lambda"));
    }
}
```

Only one correct choice.

- 0) ‘Enclosing:Lambda‘
  CORRECT - The ‘this‘ keyword inside a lambda expression is lexically scoped, meaning it refers to the ‘this‘ of the enclosing scope. Here, the lambda is created inside the ‘create()‘ instance method, so ‘this‘ refers to the ‘TrickyThis‘ instance (‘t‘). The code accesses ‘t.value‘ (which is ”Enclosing”) and concatenates it with the lambda’s parameter ‘x‘ (”Lambda”), producing ”Enclosing:Lambda”.
- 1) ‘Lambda:Enclosing‘
  WRONG - The concatenation order specified in the lambda is ‘this.value‘ first, then ‘x‘.
- 2) A compilation error occurs due to the use of ‘this‘.
  WRONG - The use of ‘this‘ is valid because the lambda is defined in an instance-level context (‘create()‘ method). If it were in a ‘static‘ method, using ‘this‘ would cause a compilation error.
- 3) A ‘NullPointerException‘ is thrown at runtime.
  WRONG - The ‘TrickyThis‘ object ‘t‘ is properly instantiated, so ‘this.value‘ is not null.

(33) (questionId: 102525, topic: ArrayList and Basic Collections)
Which code snippet demonstrates the correct way to create a generic ‘ArrayList‘ that can hold any subclass of ‘Number‘?
Only one correct choice.

- 0) `List<? super Number> list = new ArrayList<Integer>();`
  WRONG - This is a lower-bounded wildcard. It means a list of ‘Number‘ or any superclass of ‘Number‘ (like ‘Object‘). You cannot assign an ‘ArrayList¡Integer¿‘ to it because ‘Integer‘ is not a superclass of ‘Number‘.
- 1) `List<? extends Number> list = new ArrayList<Integer>();`
  CORRECT - This uses an upper-bounded wildcard. ‘List¡? extends Number¿‘ defines a list of an unknown type that is a subtype of ‘Number‘. Since ‘Integer‘ is a subtype of ‘Number‘, an ‘ArrayList¡Integer¿‘ can be legally assigned to this reference. This is the correct way to declare a list that can hold a collection of ‘Integer‘, ‘Double‘, etc.
- 2) `List<T extends Number> list = new ArrayList<T>();`
  WRONG - The ‘¡T extends ...¿‘ syntax is for defining a generic type parameter on a class or method, not for declaring a variable with a wildcard.
- 3) `List<Number> list = new ArrayList<Integer>();`
  WRONG - This is a common generics error. Generics are not covariant. An ‘ArrayList¡Integer¿‘ is NOT a subtype of ‘List¡Number¿‘, so this assignment is a compilation error. This restriction prevents runtime ‘ClassCastException‘s.

(34) (questionId: 102822, topic: Exception Hierarchy and Types)
What is the outcome of compiling and running the following code?

```
public class Test {
    static {
        if (true) {
            throw new NullPointerException("Error in static block");
        }
    }
    public static void main(String[] args) {
        System.out.println("Hello");
    }
}
```

Only one correct choice.

- 0) A ‘NullPointerException‘ is caught by the JVM and ‘Hello‘ is printed.
  WRONG - The original exception is wrapped, and the main method is never reached.
- 1) The program prints ‘Hello‘ and exits normally.
  WRONG - The program terminates before the main method can be invoked.
- 2) An ‘ExceptionInInitializerError‘ is thrown, and the program terminates.
  RIGHT - This is a critical rule for the exam. When any exception (checked or unchecked) is thrown from a static initializer block, the JVM catches it and throws a new `java.lang.ExceptionInInitializerError`, wrapping the original exception. This error indicates that the class could not be initialized and cannot be used. The program terminates immediately.
- 3) A ‘NullPointerException‘ is thrown, and the program terminates.
  WRONG - While a `NullPointerException` is the initial cause, it is not the exception that propagates out of the class loading mechanism. It gets wrapped in an `ExceptionInInitializerError`.

(35) (questionId: 103557, topic: Method Design and Variable Arguments)
Which of the following method calls are ambiguous and will cause a compilation error? (Choose all that apply)

```
class Ambiguity {
    static void m(int a, long b) {} // M1
    static void m(long a, int b) {} // M2
    static void m(int... a) {}      // M3
    static void m(Number n) {}      // M4
    static void m(Object o) {}      // M5
}
```

Multiple correct choices.

- 0) ‘Ambiguity.m(5, 10);‘
  CORRECT (Ambiguous) - The call ‘m(5, 10)‘ passes two ‘int‘s. This can match M1 (‘m(int, long)‘) by widening the second argument. It can also match M2 (‘m(long, int)‘) by widening the first argument. Since both matches require one widening conversion, neither is more specific than the other. This results in an ambiguous call, which is a compilation error.

- 1) ‘Ambiguity.m(5L, 10L);‘
  WRONG - The call ‘m(5L, 10L)‘ passes two ‘long‘s. This call does not match M1, M2, M3, M4, or M5. This results in a ’no suitable method found’ compilation error, which is different from an ’ambiguous call’ error.
- 2) ‘Ambiguity.m(5);‘
  WRONG - The call ‘m(5)‘ passes one ‘int‘. It could match M3 (‘int...‘), M4 (‘Number‘ via autoboxing), or M5 (‘Object‘ via autoboxing). Varargs (M3) has the lowest priority. Between M4 and M5, ‘Number‘ is more specific than ‘Object‘. Therefore, M4 is chosen unambiguously.
- 3) ‘Ambiguity.m(new Integer(5));‘
  WRONG - The call ‘m(new Integer(5))‘ passes an ‘Integer‘. It could match M3 (‘int...‘ via unboxing), M4 (‘Number‘), or M5 (‘Object‘). Varargs (M3) has the lowest priority. Between M4 and M5, ‘Number‘ is more specific. M4 is chosen unambiguously.
- 4) ‘Ambiguity.m(null);‘
  WRONG - The call ‘m(null)‘ could match M4 (‘Number‘) or M5 (‘Object‘). Since ‘Number‘ is a subclass of ‘Object‘, M4 is more specific and is chosen unambiguously.

(36) (questionId: 101929, topic: Encapsulation and Access Modifiers)
Examine the code:

```
public final class MyData {
    private final StringBuilder builder;

    public MyData(StringBuilder b) {
        this.builder = b;
    }

    public StringBuilder getBuilder() {
        return builder;
    }
}

// Main method in another class
StringBuilder sb = new StringBuilder("Initial");
MyData data = new MyData(sb);
sb.append(" Changed");
System.out.println(data.getBuilder());
```

What is the output?
Only one correct choice.

- 0) Initial
  WRONG - The object referenced by the final field is modified.
- 1) Initial Changed
  CORRECT - This question highlights that `final` on a reference variable does not make the object itself immutable. The field `builder` is `final`, meaning

the reference cannot be changed to point to a different `StringBuilder` object after initialization. However, `StringBuilder` is a mutable class. The reference `sb` in the main method and the field `data.builder` point to the exact same `StringBuilder` object on the heap. When `sb.append(" Changed")` is called, it modifies this single object. The subsequent call to `data.getBuilder()` returns a reference to this same, now-modified object, which is then printed.

- 2) A new ‘StringBuilder‘ object’s string representation.
  WRONG - The same object that was passed into the constructor is returned.
- 3) Compilation fails because ‘final‘ fields cannot be assigned in a constructor.
  WRONG - It is standard practice to initialize final fields within a constructor.
- 4) Compilation fails because ‘StringBuilder‘ is mutable.
  WRONG - The mutability of `StringBuilder` is a key concept here, but it does not cause a compilation failure; it leads to this specific runtime behavior.

(37) (questionId: 100726, topic: Variable Scope and Lifetime)
What is printed to the console?

```
public class TrickyScope {
    static TrickyScope ts = new TrickyScope();
    static int val = 10;
    {
        // instance initializer
        val = 20;
    }

    public static void main(String[] args) {
        System.out.println(val);
    }
}
```

Only one correct choice.

- 0) 10
  CORRECT - This is a very tricky static initialization order question. Execution proceeds top-to-bottom: 1. The first static field `static TrickyScope ts = new Tricky` is processed. This requires creating a new object. 2. The object creation triggers the instance initializer block `{ val = 20; }`. This sets the *static* variable `val` to 20. 3. The object creation completes. 4. The *next* static field declaration is processed: `static int val = 10;`. This is an assignment that **resets** the static variable `val` to 10. 5. Static initialization is complete. The final value of `val` is 10. 6. The `main` method runs and prints the final value of `val`.
- 1) 20
  WRONG - The value is reset to 10 after the instance initializer runs.
- 2) 0
  WRONG - The variable is explicitly initialized.
- 3) Compilation fails.

WRONG - The code, while tricky, is syntactically valid and compiles.

(38) (questionId: 101427, topic: StringBuilder and StringBuffer)
Given ‘StringBuilder sb = new StringBuilder(”abcde”);‘. Which statements about its capacity are true? (Choose all that apply)
Multiple correct choices.

- 0) The initial capacity is 21 (5 for ”abcde” + 16 default).
  CORRECT - When a ‘StringBuilder‘ is initialized with a ‘String‘, its initial capacity is the string’s length plus the default capacity of 16. So, ‘5 + 16 = 21‘.
- 1) ‘sb.trimToSize();‘ will likely change its capacity to 5.
  CORRECT - ‘trimToSize()‘ is a request to reduce the capacity to match the current length. If the current length is 5 and capacity is 21, this call will likely reallocate the internal array to have a capacity of 5.
- 2) After ‘sb.append(”fghijklmnopqrstuvwxyz”);‘, the capacity will be larger than its length.
  CORRECT - Appending a long string will force the ‘StringBuilder‘ to resize its internal array. The growth algorithm ensures that the new capacity will be sufficient to hold the new string, and it is almost always larger than the final length to allow for future appends without immediately resizing again.
- 3) ‘sb.ensureCapacity(10);‘ will not change the capacity.
  CORRECT - The initial capacity is 21. ‘ensureCapacity(10)‘ checks if the capacity is at least 10. Since ‘21 ¿ 10‘, the current capacity is already sufficient, and no change is made.

(39) (questionId: 100728, topic: Variable Scope and Lifetime)
Which statements about the following code are correct? (Choose all that apply)

```
public class Outer {
    private String name = "Outer";

    class Inner {
        private String name = "Inner";

        void printNames() {
            String name = "Local";
            System.out.println(name);
            System.out.println(this.name);
            System.out.println(Outer.this.name);
        }
    }

    public static void main(String... args) {
        new Outer().new Inner().printNames();
    }
}
```

Multiple correct choices.

- 0) The code will fail to compile.
  WRONG - The code is valid and compiles. It demonstrates the correct way to disambiguate variables in nested class scopes.
- 1) The output will be: Local
  WRONG - This is only the first line of the output.
- 2) The output will be: Local Inner Outer
  CORRECT - The output is 'Local', 'Inner', and 'Outer', each on a new line. Let's trace it: - `System.out.println(name);`: Prints the most tightly-scoped `name`, which is the local variable in the `printNames` method: "Local". - `System.out.println(this.name);`: `this` refers to the current object, which is an instance of `Inner`. It prints the `Inner` class's instance variable: "Inner". - `System.out.println(Outer.this.name);`: The special syntax `Outer.this` is used to access members of the enclosing class instance. It prints the `Outer` class's instance variable: "Outer".
- 3) `this.name` refers to the instance variable of the `Inner` class.
  CORRECT - The `this` keyword, when used without a class name qualifier, always refers to the current instance. In the `printNames` method, the current instance is of the `Inner` class.
- 4) `Outer.this.name` is used to access the instance variable of the enclosing `Outer` class.
  CORRECT - This is the specific syntax required for an inner class to refer to a member of its enclosing outer class instance, which is necessary here to resolve the ambiguity caused by all three scopes having a variable named `name`.

(40) (questionId: 100121, topic: Main Method and Command Line Arguments)
What is the result of attempting to compile and run the following code?

```
public class TrickyMain {
    public static void main(String args) {
        System.out.println("Hello");
    }
}
```

Only one correct choice.

- 0) It compiles and runs, printing "Hello".
  WRONG - The program will compile, but it will not run because the JVM cannot find the correct entry point.
- 1) It fails to compile because the 'main' parameter is not an array.
  WRONG - The code is syntactically valid. Overloading the 'main' method is allowed, so it will compile without error.
- 2) It compiles, but at runtime the JVM reports that 'main' is not found.
  CORRECT - This is a classic trick question. The class compiles because 'public static void main(String args)' is a valid overloaded method. However, it is not the correct entry point signature, which must accept an array ('String[]' or 'String...'). At runtime, the JVM will not find the required signature and will throw an error like 'NoSuchMethodError: main'.

- 3) It compiles and runs, but ‘args‘ is null.
  WRONG - The method is never called by the JVM, so the value of its ‘args‘ parameter is irrelevant.

(41) (questionId: 101127, topic: Break, Continue, and Labels)
Consider the following code. Which line causes a compilation error?

```
label1: while (true) {      // Line 1
    int x = 0;               // Line 2
    label2: do {             // Line 3
        x++;                 // Line 4
        continue label1;     // Line 5
    } while(x < 5);          // Line 6
    break label2;            // Line 7
}
```

Only one correct choice.

- 0) Line 3
  WRONG - Line 3 is a valid start of a ‘do-while‘ loop with a label.
- 1) Line 5
  WRONG - Line 5 (‘continue label1;‘) is valid. It is inside a loop (‘do-while‘) and it is also inside the ‘label1‘ loop, so it can legally continue the outer loop.
- 2) Line 7
  CORRECT - This line causes a compilation error. The ‘break label2;‘ statement is not *within* the scope of the ‘label2‘ statement. The ‘do-while‘ loop (which has ‘label2‘) finishes on Line 6. Line 7 is in the outer ‘while‘ loop, but it is no longer inside the ‘do-while‘ loop. A labeled ‘break‘ must be inside the statement it is labeling.
- 3) The code compiles without errors.
  WRONG - There is a compilation error on Line 7.

(42) (questionId: 100628, topic: Wrapper Classes and Autoboxing/Unboxing)
Examine this code. What will be printed to the console?

```
public class Test {
    public static void main(String[] args) {
        Integer i1 = 10;
        Long l1 = 10L;

        System.out.println(i1.equals(l1));
    }
}
```

Only one correct choice.

- 0) `true`
  WRONG - The `equals` method returns `false` in this case.
- 1) `false`
  CORRECT - The `Integer.equals(Object obj)` method is being called. Its

implementation first checks if the argument `obj` is an `instanceof Integer`. In this case, the argument `l1` is a `Long` object, so the `instanceof` check fails and the method returns `false` immediately. It never proceeds to compare their numerical values. For a cross-type value comparison, you would need to use unboxing: `i1.intValue() == l1.longValue()`.

- 2) The code does not compile.
  WRONG - The code is perfectly valid and compiles. The `equals` method takes an `Object`, so passing a `Long` reference is allowed.
- 3) A runtime exception is thrown.
  WRONG - No exception is thrown; the `equals` method handles the type mismatch gracefully by returning `false`.

(43) (questionId: 101822, topic: Garbage Collection and Object Lifecycle)
What is the final value of ‘count‘ printed to the console?

```
public class GCCount {
    static int count = 0;
    int id;

    public GCCount(int id) { this.id = id; }

    public static void main(String[] args) {
        new GCCount(1);
        GCCount g2 = new GCCount(2);
        GCCount g3 = new GCCount(3);
        g2 = g3;
        new GCCount(4);
        g3 = null;
        // Point X
        System.gc();
        System.out.println(count);
    }

    @Override
    protected void finalize() {
        count++;
    }
}
```

Only one correct choice.

- 0) 0
  WRONG - This is a possible, but not guaranteed, output if the GC doesn’t run or finalizers don’t complete in time.
- 1) 2
  WRONG - This is a possible, but not guaranteed, output.
- 2) 3
  WRONG - At point X, three objects are eligible for GC: the one from `new GCCount(1)`,

the one from `new GCCount(2)` (whose reference `g2` was reassigned), and the one from `new GCCount(4)`. The object referenced by `g3` is still reachable via `g2`. However, it is not guaranteed all three will be finalized.

- 3) 4
  WRONG - Four objects were created, but one is still reachable.
- 4) The output is not guaranteed.
  CORRECT - Although we can determine that three objects are eligible for garbage collection at Point X, the call to `System.gc()` is only a suggestion. We cannot guarantee if or when the GC will run, which eligible objects it will collect, or when their `finalize` methods will complete relative to the `println` statement. Therefore, the final value of `count` is not predictable.

(44) (questionId: 101224, topic: Enums)
What is true about the serialization of enums?
Only one correct choice.

- 0) Enums are not serializable by default and require implementing ‘java.io.Serializable‘ and defining a ‘serialVersionUID‘.
  WRONG - Enums implement `java.io.Serializable` by default. No extra work is needed.
- 1) When an enum is deserialized, the constructor is called again to create a new instance.
  WRONG - A key feature of enum serialization is that it does *not* call the constructor on deserialization. This prevents the creation of duplicate instances.
- 2) Java's serialization mechanism ensures that deserializing an enum constant will always return the pre-existing constant instance, thus preserving singleton identity.
  CORRECT - The serialization process for an enum only writes out the constant's name. During deserialization, the JVM uses this name to find and return the already-existing, canonical instance of that constant using a mechanism similar to `Enum.valueOf()`. This robustly preserves the singleton property of enum constants across serialization.
- 3) Deserializing an enum may result in a different object instance if the enum declaration has changed, causing ‘==‘ to fail.
  WRONG - Because deserialization always resolves to the canonical instance, `==` will hold true. This is a specific guarantee of the Java platform for enums.

(45) (questionId: 101625, topic: Constructors and Initialization Blocks)
What is the result of attempting to compile and run the following code?

```
abstract class Builder {
    Builder() { System.out.print("B"); }
}

public class House extends Builder {
    House() {
        // super() is implicitly called here
        System.out.print("H");
    }

    public static void main(String[] args) {
        new House();
    }
}
```

Only one correct choice.

- 0) The code fails to compile because an abstract class cannot have a constructor.
  WRONG - An abstract class can, and often must, have a constructor to initialize its state. This constructor is called via `super()` from a concrete subclass.
- 1) The code compiles and prints ”BH”.
  RIGHT - The code compiles and runs correctly. Creating a `new House()` calls the `House` constructor. The first (implicit) statement in the `House` constructor is a call to `super()`, which invokes the `Builder` constructor. The `Builder` constructor prints ”B”, then returns. The `House` constructor continues, printing ”H”. The final output is ”BH”.
- 2) The code compiles and prints ”HB”.
  WRONG - The superclass constructor always runs before the subclass constructor’s body.
- 3) The code fails to compile because ‘new Builder()‘ is not allowed.
  WRONG - The code does not attempt to instantiate the abstract class directly. It correctly instantiates a concrete subclass, which is allowed.

(46) (questionId: 100821, topic: Java Operators and Precedence)
What is the final value of ‘a‘?

```
int a = 2;
a = a++ * a++;
```

Only one correct choice.

- 0) 4
  WRONG - This might result from incorrectly evaluating the expression as ‘2 * 2‘. It doesn’t account for the side effects of the post-increment operator during the evaluation of the expression.
- 1) 6
  CORRECT - This is a classic tricky question about post-increment. The operands of the ‘*‘ operator are evaluated from left to right.1. The left operand ‘a++‘ is evaluated. Its value for the multiplication is the *current* value of ‘a‘, which is 2. After its value is used, ‘a‘ is incremented to 3.2. The right operand ‘a++‘ is evaluated. Its value is the *current* value of ‘a‘, which is now 3. After its value is used, ‘a‘ is incremented to 4.3. The multiplication is performed: ‘2 * 3‘ results in 6.4. The assignment ‘a = 6‘ happens. The final value of ‘a‘ is 6, overwriting the value of 4 it held after the increments.

- 2) 8
  WRONG - This might result from a misunderstanding, perhaps by multiplying the initial value of ‘a‘ (2) by its final value after both increments (4).
- 3) 9
  WRONG - This might result from thinking ‘a‘ is incremented before each use, leading to ‘3 * 3‘.

(47) (questionId: 101723, topic: Static Members and ’this’ Keyword)
What is the output of the following code? This question tests static initialization order.

```
public class Init {
    static { a = b * 2; }
    static int a = 10;
    static int b = 5;
    static { a = b * 3; }

    public static void main(String[] args) {
        System.out.println(a);
    }
}
```

Only one correct choice.

- 0) 10
  WRONG - The value of `a` is reassigned after its initial declaration.
- 1) 15
  RIGHT - Static initializers (both blocks and variable initializers) run in top-to-bottom order when the class is loaded.
  1. Default values are set: `a=0`, `b=0`.
  2. First static block: `a = b * 2;` becomes `a = 0 * 2;`, so `a` is 0.
  3. Next line: `static int a = 10;`. `a` is now 10.
  4. Next line: `static int b = 5;`. `b` is now 5.
  5. Second static block: `a = b * 3;` becomes `a = 5 * 3;`, so `a` is 15.
  Finally, the `main` method prints the final value of `a`, which is 15.
- 2) 30
  WRONG - The value of `b` at the time of the final calculation is 5, not 10.
- 3) The code fails to compile.
  WRONG - Reading a static variable in a block before it is initialized is a legal forward reference; it just uses the default value.

(48) (questionId: 100929, topic: Conditional Statements (if/else, switch))
What is true about the following code snippet? (Choose all that apply)

```
public class Tricky {
    public static void main(String[] args) {
        boolean a = true, b = false, c = false;
        if ( a || (b=true) && (c=true) )
            ;
        System.out.println(a + " " + b + " " + c);
    }
}
```

Multiple correct choices.

- 0) The output is ‘true false false‘.
  CORRECT - The key is operator precedence and short-circuiting. The ‘‘ operator has higher precedence than ‘——‘, so the expression is grouped as ‘a —— ((b=true) (c=true))‘. The ‘——‘ operator is evaluated first. Its left operand, ‘a‘, is ‘true‘. Since the left side of a logical OR is ‘true‘, the entire expression is guaranteed to be ‘true‘, and the ‘——‘ operator short-circuits. The right side of the ‘——‘, which is ‘(b=true) (c=true)‘, is never evaluated. Therefore, the assignments to ‘b‘ and ‘c‘ never happen, and they retain their initial values of ‘false‘.
- 1) The output is ‘true true true‘.
  WRONG - This would be the result if the entire expression were evaluated, which does not happen due to short-circuiting.
- 2) The variable ‘b‘ is assigned ‘true‘ during the evaluation of the ‘if‘ condition.
  WRONG - The assignment to ‘b‘ is part of the expression that is skipped due to short-circuiting.
- 3) The variable ‘c‘ is assigned ‘true‘ during the evaluation of the ‘if‘ condition.
  WRONG - The assignment to ‘c‘ is part of the expression that is skipped due to short-circuiting.
- 4) The code does not compile.
  WRONG - The code is syntactically valid and compiles.

(49) (questionId: 101629, topic: Constructors and Initialization Blocks)
Which statements correctly describe the complete order of initialization for an object ‘new Sub()‘ where ‘Sub extends Super‘? (Choose all that apply)
Multiple correct choices.

- 0) Static initializers of ‘Super‘ run before static initializers of ‘Sub‘.
  CORRECT - The parent class ‘Super‘ must be loaded before the child class ‘Sub‘, so its static initializers run first.
- 1) All instance initializers (both ‘Super‘ and ‘Sub‘) run before any constructor code.
  WRONG - This is incorrect. The ‘Super‘ constructor body runs *before* the ‘Sub‘ instance initializers. The full order is ‘Super‘ instance init -¿ ‘Super‘ constructor -¿ ‘Sub‘ instance init -¿ ‘Sub‘ constructor.
- 2) The constructor body of ‘Super‘ runs before the instance initializers of ‘Sub‘.
  CORRECT - The call to ‘super()‘ (implicit or explicit) ensures that the entire initialization of the superclass object, including its constructor body, completes before the subclass’s instance initializers are run.
- 3) The constructor body of ‘Sub‘ is the very last thing to run for the ‘Sub‘ object’s initialization.

CORRECT - The last step in the chain of instance creation is the execution of the subclass's own constructor body.

- 4) Static initializers of 'Sub' run before instance initializers of 'Super'.
  CORRECT - The static initialization phase for the entire class hierarchy completes before the instance initialization phase begins. Therefore, 'Sub' statics run before 'Super' instance initializers.
- 5) Instance initializers of 'Sub' run before the constructor body of 'Sub'.
  CORRECT - For any given class, its instance initializers always run just before its constructor body is executed.

(50) (questionId: 102921, topic: Try-Catch-Finally Blocks)
What is the value returned by the method 'check()'?

```
public class Test {
    public static int check() {
        try {
            return 1;
        } catch (Exception e) {
            return 2;
        } finally {
            return 3;
        }
    }
    public static void main(String[] args) {
        System.out.println(check());
    }
}
```

Only one correct choice.

- 0) '1'
  WRONG - A `return` in a `finally` block supersedes a `return` from the `try` block.
- 1) '2'
  WRONG - No exception is thrown, so the `catch` block is never entered.
- 2) '3'
  RIGHT - This demonstrates a critical rule: a `return` statement in a `finally` block will always override any `return` from the corresponding `try` or `catch` blocks. The `try` block prepares to return 1, but before it can, the `finally` block executes and its `return 3;` statement causes the method to exit immediately with the value 3.
- 3) The code does not compile.
  WRONG - The code is syntactically valid, although it is considered very poor programming practice.

(51) (questionId: 102725, topic: Sorting and Searching Collections (Comparable, Comparator))
What is the output?

```
class Legacy {
    public int compareTo(Object o) { return 0; }
}
class Generic extends Legacy implements Comparable<Generic> {
}
// in a method
Comparable c = new Generic();
System.out.println(c.compareTo("test"));
```

Only one correct choice.

- 0) ‘0‘
  WRONG - The code fails to compile.
- 1) A ‘ClassCastException‘ is thrown at runtime.
  WRONG - The issue is caught at compile time, not runtime.
- 2) The code does not compile.
  RIGHT - The class `Generic` states that it implements `Comparable<Generic>`. This is a contract that obligates the class to provide a method with the signature `public int compareTo(Generic o)`. However, the class is empty. It inherits a method `public int compareTo(Object o)` from `Legacy`. Because the inherited method's parameter is `Object`, not `Generic`, it does not satisfy the contract of `Comparable<Generic>`. Therefore, the compiler issues an error that `Generic` must implement the abstract method `compareTo(Generic)`.
- 3) The output is unpredictable.
  WRONG - The result is a predictable compilation failure.

(52) (questionId: 100528, topic: Type Conversion and Casting)
What is the result of the following code snippet?

```
float f = (float) Double.POSITIVE_INFINITY;
int i = (int) f;
System.out.println(i);
```

Only one correct choice.

- 0) ‘0‘
  WRONG - This would be the result if casting ‘NaN‘ to ‘int‘.
- 1) ‘-1‘
  WRONG - This is an incorrect value.
- 2) ‘2147483647‘

  ```
  RIGHT - This is a specific conversion rule defined in the Java Language Spec
  ```
- 3) A runtime ‘ArithmeticException‘ is thrown.
  WRONG - This special conversion does not throw an exception.

(53) (questionId: 100324, topic: Java Coding Conventions and Javadoc)
A class contains a method with the following Javadoc comment. What is the result of attempting to compile the source file containing this code?

```
/**
 * Processes a request.
 * @parameter name The name of the user.
 * @return The result of the processing.
 */
public String process(String name) { return "Processed: " + name; }
```

Only one correct choice.

- 0) Compilation fails because ‘@parameter‘ is not a valid Javadoc tag.
  WRONG - The Java compiler (`javac`) is not responsible for validating Javadoc tags. Its job is to compile Java code into bytecode.
- 1) Compilation succeeds.
  CORRECT - The content of a comment, including Javadoc tags, is ignored by the Java compiler. The code itself is syntactically valid. Therefore, compilation will succeed. It is the `javadoc` tool that would process the comment and report a warning or error about the unrecognized tag `@parameter`.
- 2) Compilation succeeds, but the ‘javadoc‘ tool will fail to execute.
  WRONG - The `javadoc` tool would likely report a warning about the unknown tag, but it might not necessarily fail to execute entirely. More importantly, this question is about compilation, which succeeds.
- 3) Compilation fails with a warning about the unrecognized tag.
  WRONG - The compiler does not issue warnings for Javadoc content; it ignores it.

(54) (questionId: 100826, topic: Java Operators and Precedence)
What is the output of the following program?

```
public class Test {
    public static void main(String[] args) {
        int x = 5;
        boolean b1 = true;
        boolean b2 = false;
        if ((x == 4) && !b2 )
            System.out.print("1 ");
        System.out.print("2 ");
        if ((b2 = true) && b1 )
            System.out.print("3 ");
    }
}
```

Only one correct choice.

- 0) 2
  WRONG - This output omits the ‘3‘. This would happen if the second ‘if‘ condition evaluated to ‘false‘.
- 1) 2 3
  CORRECT - Let’s trace the execution:1. The first ‘if‘ condition is ‘(x == 4) !b2‘. Since ‘x‘ is 5, ‘x == 4‘ is ‘false‘. Due to short-circuiting with ‘‘,

the rest of the condition is not evaluated. The block is skipped.2. The line ‘System.out.print(”2 ”);‘ is unconditional and executes. Output is now ‘”2 ”‘.3. The second ‘if‘ condition is ‘(b2 = true) b1‘. This is an *assignment*, not a comparison. ‘b2‘ is assigned ‘true‘. The value of an assignment expression is the assigned value, so ‘(b2 = true)‘ evaluates to ‘true‘. 4. The condition becomes ‘true b1‘. Since ‘b1‘ is ‘true‘, the whole condition is ‘true‘. The block executes and prints ‘”3 ”‘.5. Final output is ‘”2 3”‘.

- 2) 1 2 3
  WRONG - This implies the first ‘if‘ condition evaluated to ‘true‘, which is incorrect as ‘x‘ is 5, not 4.
- 3) 1 2
  WRONG - This implies the second ‘if‘ condition was false. It is ‘true‘ because of the assignment ‘b2 = true‘.

(55) (questionId: 101027, topic: Looping Constructs (for, while, do-while))
What is the final value of ‘count‘?

```
int count = 0;
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 5; j++) {
        if (j == 2)
            continue;
        count++;
    }
}
```

Only one correct choice.

- 0) 25
  WRONG - This would be the result if the ‘continue‘ statement were not present (‘5 * 5 = 25‘).
- 1) 20
  CORRECT - The outer loop runs 5 times (for ‘i‘ from 0 to 4). The inner loop is set to run 5 times (for ‘j‘ from 0 to 4). However, the ‘continue‘ statement skips the ‘count++‘ line whenever ‘j‘ is 2. This means for each of the 5 outer loop iterations, the inner loop only performs 4 increments (‘j=0,1,3,4‘). Therefore, the total number of increments is $5 \times 4 = 20$.
- 2) 15
  WRONG - This count is too low. The ‘continue‘ only removes one increment per outer loop iteration.
- 3) 10
  WRONG - This count is too low.

(56) (questionId: 101121, topic: Break, Continue, and Labels)
What is the result of attempting to compile and run this code?

```
public class LabeledBlock {
    public static void main(String[] args) {
        int x = 5;
        myBlock: {
            if (x == 5) {
                break myBlock;
            }
            System.out.print("Inside");
        }
        System.out.print("Outside");
    }
}
```

Only one correct choice.

- 0) It prints ‘InsideOutside‘.
  WRONG - The ‘break myBlock;‘ statement causes the code inside the block to be skipped.
- 1) It prints ‘Outside‘.
  CORRECT - This code is valid. A label can be applied to a simple block of code. The ‘break myBlock;‘ statement transfers control to the end of the labeled block. Therefore, ‘System.out.print(”Inside”);‘ is skipped, and execution continues with ‘System.out.print(”Outside”);‘.
- 2) It prints ‘Inside‘.
  WRONG - The ‘System.out.print(”Outside”);‘ statement is executed after the labeled block is exited.
- 3) It fails to compile.
  WRONG - Using a labeled ‘break‘ to exit a simple code block is a valid, though uncommon, feature of Java.