1Z0-808 Mock Exam Solutions

ExamId: 100

August 5, 2025

(1) (questionId: 100127, topic: Main Method and Command Line Arguments)
Which of the following ‘main‘ method signatures will cause a ‘java.lang.NoSuchMethodError: main‘ exception at runtime, assuming the class is compiled successfully? (Choose all that apply)
Multiple correct choices.

- 0) ‘public static void main(String... args)‘
WRONG - This is a valid entry-point signature using varargs. It will run successfully.
- 1) ‘public void main(String[] args)‘
CORRECT - This signature is missing the ‘static‘ keyword. The class will compile, but the JVM will not recognize this instance method as the entry point.
- 2) ‘public static void Main(String[] args)‘
CORRECT - Java is case-sensitive. The method name must be ‘main‘, not ‘Main‘. This will compile but not run.
- 3) ‘public static void main(String args)‘
CORRECT - The parameter must be an array of ‘String‘ (‘String[]‘ or ‘String...‘), not a single ‘String‘. This will compile but not run.
- 4) ‘public static int main(String[] args)‘
CORRECT - The return type must be ‘void‘, not ‘int‘. This will compile but not run.

(2) (questionId: 100926, topic: Conditional Statements (if/else, switch))
Examine this code carefully. What is the result?

```
public class Test {
    public static void main(String[] args) {
        Integer i = 128;
        Integer j = 128;
        int k = 128;

        if (i == j) {
            System.out.print("A");
        }
        if (i == k) {
            System.out.print("B");
        }
    }
}
```

Only one correct choice.

- 0) A
WRONG - The first ‘if‘ condition is false.
- 1) B
CORRECT - This question tests autoboxing and ‘Integer‘ caching. 1. ‘if (i == j)‘: Java caches ‘Integer‘ objects for values from -128 to 127. Since 128

is outside this range, ‘i‘ and ‘j‘ are two separate ‘Integer‘ objects on the heap. The ‘==‘ operator compares their memory references, which are different, so the condition is ‘false‘.2. ‘if (i == k)‘: This compares an ‘Integer‘ object (‘i‘) with a primitive ‘int‘ (‘k‘). When this happens, the ‘Integer‘ object is automatically unboxed to its primitive ‘int‘ value. The comparison becomes a primitive comparison: ‘128 == 128‘, which is ‘true‘. ‘”B”‘ is printed.

- 2) AB
  WRONG - The first condition ‘i == j‘ evaluates to ‘false‘, so ‘”A”‘ is not printed.
- 3) No output is produced.
  WRONG - The second condition ‘i == k‘ evaluates to ‘true‘, so there is output.

(3) (questionId: 102427, topic: One-Dimensional and Multi-Dimensional Arrays)
What is the final value of `sum`?

```
long[][] arr = new long[2][2];
arr[0] = new long[]{1, 2};
arr[1] = arr[0];
arr[0][0] = 5;
long sum = arr[0][0] + arr[1][0];
```

Only one correct choice.

- 0) 6
  WRONG - This assumes ‘arr[1][0]‘ kept its original default value.
- 1) 7
  WRONG - This likely results from adding the modified ‘arr[0][0]‘ (5) and the original ‘arr[0][1]‘ (2).
- 2) 10
  CORRECT - This is a question about object references. The line `arr[1] = arr[0];` makes the reference ‘arr[1]‘ point to the *exact same* inner array object as ‘arr[0]‘. When `arr[0][0]` is set to 5, the single underlying array is modified. Since ‘arr[1]‘ points to that same array, ‘arr[1][0]‘ is also 5. The sum is ‘5 + 5 = 10‘.
- 3) Compilation fails.
  WRONG - The code is syntactically valid.

(4) (questionId: 101026, topic: Looping Constructs (for, while, do-while))
What will be printed after this code executes?

```
String[] data = {"a", "b", "c"};
int x = 0;
for(;;){
    try {
        System.out.print(data[x++]);
    } catch (ArrayIndexOutOfBoundsException e) {
        break;
    }
}
```

Only one correct choice.

- 0) abc
  CORRECT - The code uses an infinite ‘for(;;)‘ loop, meaning termination must occur via ‘break‘, ‘return‘, or an unhandled exception. The loop prints elements from the ‘data‘ array. It prints ‘data[0]‘ (‘a‘), ‘data[1]‘ (‘b‘), and ‘data[2]‘ (‘c‘). In the next iteration, ‘x‘ is 3, and ‘data[3]‘ throws an ‘ArrayIndexOutOfBoundsException‘. This exception is caught by the ‘catch‘ block, which then executes ‘break‘, terminating the loop.
- 1) ab
  WRONG - The loop successfully processes ‘data[2]‘ (‘c‘) before the exception is thrown.
- 2) a
  WRONG - The loop processes more than just the first element.
- 3) An infinite loop occurs.
  WRONG - The loop is not infinite because the ‘break‘ statement in the ‘catch‘ block provides a guaranteed exit condition.

(5) (questionId: 101129, topic: Break, Continue, and Labels)
Given the following code, which statements are true? (Choose all that apply)

```
public class Test {
    public static void main(String... args) {
        String result = "";
        loop:
        for (int i=0; i<4; i++) {
            if (i % 2 == 0) {
                continue;
            }
            switch(i) {
                case 1: result += "A"; break;
                case 3: result += "B"; break loop;
                case 5: result += "C";
            }
            result += "D";
        }
        System.out.println(result);
    }
}
```

Multiple correct choices.

- 0) The ‘continue‘ statement is executed when ‘i‘ is 0 and 2.
  CORRECT - The ‘if (i
- 1) The code enters the ‘switch‘ statement when ‘i‘ is 1 and 3.
  CORRECT - Because of the ‘continue‘ for even numbers, the ‘switch‘ statement is only reached when ‘i‘ is odd, which is ‘i=1‘ and ‘i=3‘.

- 2) The string ‘”D”‘ is appended to ‘result‘ exactly once.
  CORRECT - The line ‘result += ”D”;‘ is only reached if the ‘switch‘ statement completes without a ‘break loop‘. This happens when ‘i=1‘. The ‘break‘ in ‘case 1:‘ only exits the ‘switch‘, not the loop, so ‘”D”‘ is appended. When ‘i=3‘, ‘break loop;‘ is executed, so the line is not reached. Thus, ‘”D”‘ is appended only once.

- 3) The ‘break loop;‘ statement is executed.
  CORRECT - When ‘i=3‘, ‘case 3‘ is executed, which contains the ‘break loop;‘ statement. This statement terminates the entire ‘for‘ loop.

- 4) The final output is ‘ABD‘.
  WRONG - Let’s trace the ‘result‘ string. When ‘i=1‘, ‘result‘ becomes ‘”A”‘, then ‘”AD”‘. When ‘i=3‘, ‘result‘ becomes ‘”ADB”‘. The final output is ‘ADB‘.

- 5) The final output is ‘AB‘.
  WRONG - The final output is ‘ADB‘.

(6) (questionId: 102223, topic: Abstract Classes and Interfaces)
What is the result of attempting to access ‘MyDevice.NAME‘ in another class?

```
interface Device {
    String NAME = "Device";
}
interface Gadget {
    String NAME = "Gadget";
}
class MyDevice implements Device, Gadget {
    // Some code
}
// In another class:
// System.out.println(MyDevice.NAME);
```

Only one correct choice.

- 0) It prints ”Device”.
  WRONG - The reference is ambiguous.

- 1) It prints ”Gadget”.
  WRONG - The reference is ambiguous.

- 2) It results in a compile-time error due to an ambiguous field.
  RIGHT - The class ‘MyDevice‘ implements both ‘Device‘ and ‘Gadget‘. Both interfaces define a ‘public static final‘ field named ‘NAME‘. Because ‘MyDevice‘ inherits both fields, a reference to ‘MyDevice.NAME‘ is ambiguous. The compiler does not know whether to use the ‘NAME‘ from ‘Device‘ or the one from ‘Gadget‘. This results in a compile-time error. The ambiguity must be resolved by being more specific, e.g., ‘Device.NAME‘ or ‘Gadget.NAME‘.

- 3) It prints ‘null‘.
  WRONG - The issue is an ambiguity error at compile time, not a ‘null‘ value.

(7) (questionId: 100029, topic: Java Environment and Fundamentals)
Which of these are valid command line argument arrays in a main method signature? (Choose all that apply)
Multiple correct choices.

- 0) `String args[]`
  CORRECT - This is the classic C-style array declaration syntax, which is perfectly valid in Java.
- 1) `String... args`
  CORRECT - This is the varargs (variable arguments) syntax, introduced in Java 5. It is a valid and common way to declare the main method's parameter.
- 2) `String[] myArgs`
  CORRECT - The standard Java array declaration syntax is `Type[] name`. The name of the parameter can be any valid identifier, like `myArgs`.
- 3) `String[] _args`
  CORRECT - `_args` is a valid identifier in Java, so this declaration is syntactically correct.
- 4) `String..._args`
  WRONG - The varargs ellipsis (`...`) must be separated from the parameter name by whitespace. `..._args` is a syntax error.

(8) (questionId: 101826, topic: Garbage Collection and Object Lifecycle)
Select all lines of code after which at least one 'Gadget' object becomes eligible for garbage collection.

```
class Gadget {}
public class GadgetFactory {
    static Gadget staticGadget = new Gadget(); // Line 1
    Gadget instanceGadget = new Gadget();      // Line 2

    public static void main(String[] args) {
        GadgetFactory gf = new GadgetFactory(); // Line 3
        Gadget g1 = new Gadget();               // Line 4
        gf.build(g1);
        g1 = null;                              // Line 5
        gf = null;                              // Line 6
    }

    void build(Gadget g) {
        Gadget g2 = new Gadget();               // Line 7
    } // End of build method is effectively Line 8
}
```

Multiple correct choices.

- 0) Line 5
  CORRECT - After `Line 5`, the local reference `g1` is nulled. The `Gadget` object it was pointing to (created on Line 4) now has no more references and becomes

eligible for GC.

- 1) Line 6
  CORRECT - After `Line 6`, the local reference `gf` is nulled. This makes the `GadgetFactory` object eligible for GC. Because the `instanceGadget` was an instance member of that object, it also becomes unreachable and eligible for GC.

- 2) Line 8
  CORRECT - The variable `g2` is local to the `build` method. When the method execution ends (at Line 8), `g2` goes out of scope. The `Gadget` object it referenced (created on Line 7) becomes eligible for GC.

- 3) The line after the ‘main‘ method completes.
  CORRECT - The `staticGadget` is referenced by a static variable of the `GadgetFactory` class. This reference will persist as long as the class is loaded. When the `main` method completes and the application terminates, the class may be unloaded, at which point the static variable is gone and the `staticGadget` becomes eligible for collection.

- 4) Line 3
  WRONG - At Line 3, the `GadgetFactory` object is created and is actively referenced by `gf`. Nothing becomes eligible for GC at this point.

(9) (questionId: 102722, topic: Sorting and Searching Collections (Comparable, Comparator))
What is the result of this code?

```
Comparator<Integer> c = (i1, i2) -> i1 - i2;
List<Integer> list = Arrays.asList(Integer.MAX_VALUE, Integer.MIN_VALUE);
Collections.sort(list, c);
System.out.println(list);
```

Only one correct choice.

- 0) ‘[-2147483648, 2147483647]‘
  WRONG - This would be the correct, numerically sorted order. However, the provided comparator has a subtle bug.

- 1) ‘[2147483647, -2147483648]‘
  RIGHT - This question tests your knowledge of integer overflow. The lambda `(i1, i2) -> i1 - i2` is a common but unsafe way to write a comparator for integers. When the sort algorithm compares `Integer.MAX_VALUE` and `Integer.MIN_VALUE`, the expression becomes `Integer.MAX_VALUE - Integer.MIN_VALUE`. This calculation overflows the maximum value an `int` can hold and wraps around to become a negative number. Because `compare(MAX_VALUE, MIN_VALUE)` returns a negative value, the sort algorithm incorrectly concludes that `MAX_VALUE` is ’less than’ `MIN_VALUE`, resulting in the wrong sort order.

- 2) An ‘ArithmeticException‘ is thrown.
  WRONG - Integer overflow does not throw an `ArithmeticException` in Java; it silently wraps around.

- 3) The list remains unchanged.
  WRONG - The list will be sorted, but incorrectly due to the flawed comparator.

(10) (questionId: 100723, topic: Variable Scope and Lifetime)
Consider the following class. What is the outcome?

```
public class Test {
    static {
        i = 20; // Forward reference is ok in assignment
    }
    static int i = 10;

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

Only one correct choice.

- 0) 20
  WRONG - The final assignment to `i` overrides the value set in the static block.
- 1) 10
  CORRECT - This is a tricky question about the order of static initializers. The rules are executed top-to-bottom. 1. The static block `static { i = 20; }` is executed first. Assigning to a static field before its declaration (a forward reference) is legal for simple assignments. After this, `i` holds the value 20. 2. The static variable declaration `static int i = 10;` is executed next. This is also an assignment operation, and it *re-initializes* `i` to 10. 3. Static initialization is complete, and the final value of `i` is 10. The `main` method then prints this value.
- 2) Compilation fails due to illegal forward reference.
  WRONG - An 'illegal forward reference' error occurs when you try to *read* a variable before it's declared (e.g., `System.out.println(i);` in the static block). A simple assignment is permitted.
- 3) 0
  WRONG - The variable `i` does not retain its default value of 0.

(11) (questionId: 102126, topic: Polymorphism and Type Casting)
What is the result of attempting to compile this code snippet?

```
import java.util.*;

public class GenericsTest {
    public static void main(String[] args) {
        List<String> stringList = new ArrayList<>();
        if (stringList instanceof List<Integer>) {
            System.out.println("It's a list of Integers");
        }
        }
}
```

Only one correct choice.

- 0) The code compiles and runs, but the ‘if‘ block is never executed.
  WRONG - The code does not compile.
- 1) The code compiles and throws a ‘ClassCastException‘ at runtime.
  WRONG - The error is caught at compile time.
- 2) A compile-time error occurs.
  RIGHT - Due to a process called type erasure, the generic type parameters (like `<String>` or `<Integer>`) are removed by the compiler and are not present at runtime. At runtime, the object is just a raw `List`. Because the specific type information is unavailable, the `instanceof` operator cannot test against a parameterized type. The Java compiler enforces this by making any use of `instanceof` with a generic type parameter a compile-time error.
- 3) The code compiles and runs, and the ‘if‘ block is executed due to type erasure.
  WRONG - Type erasure is the reason for the compile-time error; it does not cause the code to run and enter the `if` block.

(12) (questionId: 100421, topic: Primitive Data Types and Literals)
What is the result of attempting to compile the following code snippet?

```
int i = 10;
byte b = i;
```

Only one correct choice.

- 0) It compiles successfully because the value of ‘i‘ (10) is within the range of a ‘byte‘.
  WRONG - This is a common misunderstanding. The rule for implicit narrowing applies only to compile-time constant *literals*, not variables.
- 1) It fails to compile because ‘i‘ is an ‘int‘ variable, and assigning it to a ‘byte‘ requires an explicit cast.
  RIGHT - This is a crucial distinction for the exam. While ‘byte b = 10;‘ compiles (assigning a literal), ‘byte b = i;‘ does not. When the right side of the assignment is a variable (here, the ‘int‘ variable ‘i‘), the compiler enforces the type-checking rules strictly. Assigning an ‘int‘ to a ‘byte‘ is a narrowing conversion and requires an explicit cast, ‘byte b = (byte)i;‘, regardless of the value held by the variable.
- 2) It compiles, but will throw a runtime exception if ‘i‘ were greater than 127.
  WRONG - The issue is a compile-time error, not a runtime exception.
- 3) It compiles because the compiler can determine the constant value of ‘i‘ at compile time.
  WRONG - Even though a modern compiler can often determine the value of ‘i‘, the Java Language Specification mandates that this type of assignment requires a cast.

(13) (questionId: 101227, topic: Enums)
Which of the following are true about enums in Java? (Choose all that apply)
Multiple correct choices.

- 0) An enum can be a generic type, e.g., ‘public enum MyEnum¡T¿ ... ‘
WRONG - Enums cannot be generic. A declaration like `public enum MyEnum<T>` is a syntax error. The enum type itself is the type.
- 1) Enum constants are implicitly ‘public‘, ‘static‘, and ‘final‘.
CORRECT - The constants declared in an enum are effectively `public static final` fields of that enum type. `public` so they are accessible, `static` so they belong to the type, and `final` so they cannot be reassigned.
- 2) An enum can contain a ‘main‘ method and can be executed as a standalone program.
CORRECT - An enum is a special type of class. It can have a `main` method and be run from the command line like any other Java application.
- 3) An enum type cannot be a subtype of another enum.
CORRECT - All enums implicitly extend `java.lang.Enum`. Due to Java’s single-inheritance model for classes, an enum cannot extend another class, which includes other enums.

(14) (questionId: 101622, topic: Constructors and Initialization Blocks)
What is the output of this program?

```
public class ForwardReference {
    {
        System.out.print(value + " ");
    }
    private int value = 1;
    {
        System.out.print(value + " ");
    }

    public ForwardReference() {
        System.out.print(value);
    }

    public static void main(String... args) {
        new ForwardReference();
    }
}
```

Only one correct choice.

- 0) 1 1 1
WRONG - The first print statement occurs before the explicit initialization of `value`.
- 1) 0 1 1
RIGHT - This demonstrates a ’legal forward reference’. The initialization

sequence is:
1. Instance variables get default values. `value` becomes 0.
2. The first instance block executes. It prints the current value of `value`, which is 0.
3. The instance variable initializer runs. `value` is set to 1.
4. The second instance block executes. It prints the current value of `value`, which is 1.
5. The constructor body executes. It prints the current value of `value`, which is 1.
The final output is ‘0 1 1‘.

- 2) 0 0 1
  WRONG - The second instance block runs after `value` has been initialized to 1.

- 3) The code fails to compile.
  WRONG - While it may seem like an error to reference a variable before its declaration, Java allows this for instance variables within instance initializers, using the default value.

(15) (questionId: 102022, topic: Inheritance and Method Overriding)
What is the result?

```
class SuperClass {
    static String ID = "Super";
    void printID() { System.out.println(ID); }
}

class SubClass extends SuperClass {
    static String ID = "Sub";
    void printID() { System.out.println(ID); }
}

public class TestHiding {
    public static void main(String[] args) {
        SuperClass sup = new SubClass();
        System.out.println(sup.ID);
        sup.printID();
    }
}
```

Only one correct choice.

- 0) Super
  CORRECT - This question tests the difference between static field hiding and instance method overriding. 1) ‘sup.ID‘: ‘ID‘ is a static field. Static members are resolved at compile time based on the reference type. ‘sup‘ is a ‘SuperClass‘ reference, so this resolves to ‘SuperClass.ID‘, which is ’Super’. 2) ‘sup.printID()‘: ‘printID‘ is an instance method. Its resolution is polymorphic, based on the runtime object type, which is ‘SubClass‘. Therefore, the overridden ‘printID‘ in ‘SubClass‘ is called. It prints the ‘ID‘ field that is visible in

its scope, which is the ‘ID‘ from ‘SubClass‘ (’Sub’).

- 1) Sub
  WRONG - The static field access ‘sup.ID‘ resolves to the ‘SuperClass‘ version.
- 2) Super
  WRONG - The instance method call ‘sup.printID()‘ resolves to the ‘SubClass‘ version.
- 3) Sub
  WRONG - This reverses both results.
- 4) Compilation fails.
  WRONG - Hiding static fields and overriding instance members are both valid Java concepts.

(16) (questionId: 103122, topic: Try-with-Resources)
An exception is thrown from a ‘try-with-resources‘ block, another from the resource’s ‘close()‘ method, and a third from the ‘finally‘ block. Which exception is ultimately propagated to the caller?
Only one correct choice.

- 0) The exception from the ‘try‘ block.
  WRONG - The exception from the `try` block would be suppressed by the exception from the `finally` block.
- 1) The exception from the ‘close()‘ method.
  WRONG - The exception from the `close()` method would be suppressed by the exception from the `finally` block.
- 2) The exception from the ‘finally‘ block.
  CORRECT - This is a critical rule of exception handling. An exception thrown from a `finally` block will always take precedence, suppressing any exception that was already thrown from the `try` block or the resource’s `close()` method. The exception from the `finally` block is the one that the caller will see.
- 3) A new wrapper exception containing all three.
  WRONG - Java’s mechanism suppresses the older exceptions in favor of the newest one from the `finally` block; it does not wrap them all.

(17) (questionId: 100424, topic: Primitive Data Types and Literals)
What value is stored in the variable ‘result‘ after this code is executed?

```
long result = 2_147_483_647 + 1;
```

Only one correct choice.

- 0) ‘2147483648‘
  WRONG - This would be the result if the calculation were done using ‘long‘ arithmetic, e.g., ‘2147483647L + 1‘.
- 1) ‘-2147483648‘

```
RIGHT - This is a tricky question about order of operations and integer over
```

- 2) The code fails to compile.
  WRONG - The code compiles, but the result is not what it appears to be due to overflow.
- 3) ‘21474836471‘
  WRONG - This is not how integer addition works.

(18) (questionId: 103452, topic: Static Imports)
What is the output of the following code, which uses a statically imported nested class?

```
// File: Encloser.java
public class Encloser {
    public static class Nested {
        public void hi() { System.out.println("Hi"); }
    }
}

// File: Main.java
import static Encloser.Nested;

public class Main {
    public static void main(String[] args) {
        Nested n = new Nested();
        n.hi();
    }
}
```

Only one correct choice.

- 0) ‘Hi‘
  CORRECT - A ‘public static‘ nested class is considered a static member and can be imported using ‘import static‘. This allows the nested class to be referenced by its simple name (‘Nested‘) without the enclosing class name prefix (‘Encloser.‘). The code correctly instantiates ‘Nested‘ and calls a method on it.
- 1) The code fails to compile because you cannot statically import a class.
  WRONG - This statement is too general. While you cannot statically import a *top-level* class, you can statically import a *static nested* class.
- 2) The code fails to compile because ‘Nested‘ must be instantiated via ‘Encloser.Nested‘.
  WRONG - The very purpose of the static import is to allow the use of the simple name ‘Nested‘ instead of ‘Encloser.Nested‘.
- 3) The code fails to compile for a different reason.
  WRONG - The code is valid.

(19) (questionId: 100722, topic: Variable Scope and Lifetime)
What will the following code print?

```
public class ScopePuzzle {
        int x = 5;

        public static void main(String[] args) {
            ScopePuzzle p = new ScopePuzzle();
            p.go();
        }

        void go() {
            int x;
            go2();
            // System.out.println(x); // Line X
        }

        void go2() {
            x = 10;
        }
    }
```

Only one correct choice.

- 0) If Line X is uncommented, the code will print 10.
  WRONG - The local variable `x` in `go()` is never initialized.
- 1) If Line X is uncommented, the code will print 5.
  WRONG - The local variable `x` in `go()` is not affected by the assignment in `go2()`, which modifies the instance variable.
- 2) If Line X is uncommented, the code will fail to compile.
  CORRECT - In method `go()`, a local variable `int x;` is declared but never initialized. The call to `go2()` modifies the *instance* variable `x` because that's the only `x` visible within `go2()`'s scope. When execution returns to `go()`, its local variable `x` remains uninitialized. Attempting to print it with `System.out.println(x);` would refer to this uninitialized local variable, causing a definite assignment compilation error.
- 3) The code as is will compile and run without error.
  WRONG - The code as is, with Line X commented out, does compile. But the question is about the *result* or implication of the code structure, best described by what would happen if Line X were active.

(20) (questionId: 102320, topic: The 'final' Keyword)
What is the output of the following code?

```
public class Finalizer {
    private final int value;
    public Finalizer(int v) {
        this.value = v;
    }
    public int getValue() {
        return this.value;
    }
    public static void main(String[] args) {
        final Finalizer f = new Finalizer(20);
        // Line X
        System.out.println(f.getValue());
    }
    public void modify(Finalizer fin) {
        fin = new Finalizer(30);
    }
}
```

What would happen if ‘modify(f);‘ was inserted at ’Line X’?
Only one correct choice.

- 0) The code would fail to compile because ‘f‘ is final.
  WRONG - The code compiles. Calling a method and passing a ‘final‘ reference is perfectly fine.
- 1) The code would print 30.
  WRONG - The ‘modify‘ method only changes its local copy of the reference, not the original ‘f‘ variable in ‘main‘.
- 2) The code would print 20.
  RIGHT - This question tests ‘final‘ and Java’s pass-by-value semantics. When ‘modify(f)‘ is called, a **copy** of the reference ‘f‘ is passed to the method. Inside ‘modify‘, the line ‘fin = new Finalizer(30);‘ reassigns this **local copy** (‘fin‘) to a new object. This action has no effect on the original ‘f‘ variable in the ‘main‘ method, which remains ‘final‘ and continues to point to the original object with ‘value = 20‘. Therefore, the output is 20.
- 3) The code would throw a runtime exception.
  WRONG - The code runs without any exceptions.

(21) (questionId: 100924, topic: Conditional Statements (if/else, switch))
What are the final values of ‘x‘ and ‘y‘ after this code snippet runs?

```
int x = 10;
int y = 20;
if (++x <= 10 && --y > 15) {
    x++;
    y++;
}
```

Only one correct choice.

- 0) ‘x‘ is 11, ‘y‘ is 20
  CORRECT - The code tests pre-increment and short-circuiting. 1. The left operand of “ is evaluated first: ‘++x ¡= 10‘. The pre-increment operator changes ‘x‘ to 11. The comparison ‘11 ¡= 10‘ is ‘false‘.2. Since the left operand of a logical AND (“) is ‘false‘, the entire expression must be ‘false‘. The “ operator short-circuits, and the right operand (‘–y ¿ 15‘) is never evaluated.3. The ‘if‘ block is skipped. 4. The final value of ‘x‘ is 11, and the final value of ‘y‘ remains unchanged at 20.

- 1) ‘x‘ is 11, ‘y‘ is 19
  WRONG - This would be the result if the right side of the ‘‘ were evaluated, but it is not due to short-circuiting.
- 2) ‘x‘ is 12, ‘y‘ is 20
  WRONG - This implies the ‘if‘ block was entered, which is incorrect.
- 3) ‘x‘ is 10, ‘y‘ is 20
  WRONG - The ‘++x‘ expression was evaluated, so ‘x‘ cannot be 10.

(22) (questionId: 101627, topic: Constructors and Initialization Blocks)
What is the result of compiling this class?

```
public class FinalChallenge {
    private final int value;

    public FinalChallenge() {
        this(10);
        // value = 20; // Line A
    }

    public FinalChallenge(int value) {
        this.value = value;
    }
}
```

Only one correct choice.

- 0) The code compiles successfully as is.
  WRONG - While the code does compile as is, this is not the best answer in an exam context, as choice 2 points out a critical rule violation in a ’what-if’ scenario, which is a common testing pattern.
- 1) The code fails to compile because a final field is assigned in one constructor but not the other.
  WRONG - The code does compile. The no-arg constructor properly delegates initialization to the one-arg constructor, so all paths lead to the final field being initialized.
- 2) If Line A is uncommented, the code will fail to compile.
  RIGHT - This is the best answer because it tests a critical rule. If Line A were uncommented, the code would be `this(10); value = 20;`. This fails to compile for two reasons: 1) the call to `this()` would no longer be the first statement, and 2) the `final` variable `value` would be assigned twice on this construction path (once in the called constructor, and again at Line A). This is illegal.
- 3) The code fails to compile because a final field cannot be assigned in a constructor that uses ‘this()‘.
  WRONG - It is perfectly legal for a constructor to delegate initialization of a final field using `this()`. The error only occurs if that constructor also tries to assign a value to the field itself.

(23) (questionId: 101729, topic: Static Members and ’this’ Keyword)
Given ‘public class Test static int x = 1; int y = 2; ‘, which of the following lines of code are valid if placed inside the ‘main‘ method of another class? (Choose all that apply)
Multiple correct choices.

- 0) ‘System.out.println(Test.x);‘
  CORRECT - `x` is a public/default static variable and can be accessed via its class name.
- 1) ‘System.out.println(Test.y);‘
  WRONG - This is a compile error. `y` is an instance variable and cannot be accessed in a static way via the class name.
- 2) ‘Test t = new Test(); System.out.println(t.x);‘
  CORRECT - This is valid but discouraged. The compiler allows accessing a static variable via an instance reference. It resolves the access based on the reference type, not the object itself.
- 3) ‘Test t = new Test(); System.out.println(t.y);‘
  CORRECT - This is the standard way to access an instance variable: through a valid reference to an object instance.
- 4) ‘Test t = null; System.out.println(t.x);‘
  CORRECT - This is the tricky case. Since `x` is static, the compiler resolves `t.x` to `Test.x` and does not need to dereference the null pointer `t`. No `NullPointerException` is thrown.
- 5) ‘Test t = null; System.out.println(t.y);‘
  WRONG - This line compiles, but it will throw a `NullPointerException` at runtime because it attempts to access an instance variable `y` through a `null` reference. The question requires that the code not cause a runtime exception.

(24) (questionId: 100524, topic: Type Conversion and Casting)
What is the final value of ‘s‘?

```
short s = 32767;
s++;
```

Only one correct choice.

- 0) ‘32768‘
  WRONG - The value ‘32768‘ cannot be stored in a ‘short‘.
- 1) ‘-32768‘

  `RIGHT - The ‘++‘ operator is a compound assignment operator, equivalent here`
- 2) ‘0‘
  WRONG - The value wraps around to the minimum value, not zero.
- 3) The code does not compile.
  WRONG - The code compiles because the ‘++‘ operator includes an implicit cast, which makes the narrowing conversion valid.

(25) (questionId: 100827, topic: Java Operators and Precedence)
Which statement best describes the evaluation of the following expression?

```
int a = 1, b = 2, c = 3, d = 4;
int result = a + b * c / d > a ? b + c : d - a;
```

Only one correct choice.

- 0) The expression evaluates to 5.
  WRONG - While the final result is indeed 5, this choice only describes the outcome, not the evaluation process. The question asks for the *best description of the evaluation*, which usually refers to the rules being applied, such as operator precedence.

- 1) The expression evaluates to 3.
  WRONG - The expression evaluates to 5. 3 would be the result if the ternary condition were false (‘d - a‘ equals ‘4 - 1 = 3‘).

- 2) The multiplication ‘b*c‘ is performed first.
  CORRECT - This statement accurately describes the first step in evaluating the expression according to Java’s operator precedence rules. Multiplicative operators (‘*‘, ‘/‘) have higher precedence than additive (‘+‘), relational (‘¿‘), and ternary (‘? :‘) operators. Therefore, ‘b * c‘ is the first calculation performed.

- 3) The ternary operator ‘? :‘ has higher precedence than ‘¿‘.
  WRONG - This is incorrect. Relational operators like ‘¿‘ have higher precedence than the ternary operator ‘? :‘. The entire expression to the left of the ‘¿ is evaluated first to serve as the boolean condition for the ternary operator.

(26) (questionId: 101924, topic: Encapsulation and Access Modifiers)
Given the code:

```
// In package company.parts
package company.parts;
public class Engine {
    // package-private constructor
    Engine() {}
}

// In package company.parts
package company.parts;
public class PartsFactory {
    public static Engine getEngine() {
        return new Engine();
    }
}

// In package company.vehicles
package company.vehicles;
import company.parts.*;
public class Car {
    public static void main(String[] args) {
        Engine e = PartsFactory.getEngine(); // Line X
        System.out.println("Engine acquired");
    }
}
```

What is the result?
Only one correct choice.

- 0) Compilation fails at Line X because ‘Engine‘’s constructor is not visible.
  WRONG - The call to the constructor happens inside `PartsFactory`, where it is visible. The `Car` class does not call the constructor directly.
- 1) Compilation fails at Line X because the ‘Engine‘ class is not visible.
  WRONG - The `Engine` class is `public`, so it is visible to the `Car` class.
- 2) Compilation succeeds, and ”Engine acquired” is printed.
  CORRECT - This demonstrates the Factory Pattern. The `Engine` class is `public` and visible everywhere. Its constructor is package-private, restricting direct instantiation to its own package. The `PartsFactory`, being in the same package, can legally call `new Engine()`. The factory's `getEngine()` method is `public`, so any class (like `Car`) can call it. The `Car` class receives a valid `Engine` object without needing access to its constructor. The code compiles and runs successfully.
- 3) Compilation fails because ‘PartsFactory.getEngine()‘ returns a type whose constructor is not public.
  WRONG - The compiler does not check the visibility of the constructor of the return type. It only checks that the factory method itself is accessible and that the returned type is visible.

(27) (questionId: 101528, topic: Classes and Objects Fundamentals)
Which statements are true regarding the initialization of a new object? (Choose all that apply)
Multiple correct choices.

- 0) The constructor body is executed before instance initializers.
  WRONG - Instance initializers and variable initializers run *before* the constructor body is executed.
- 1) If present, a call to another constructor using ‘this()‘ must be the very first statement in a constructor.
  CORRECT - This is a strict rule. A call to `this()` or `super()` can only appear as the very first statement inside a constructor.
- 2) Static variables are initialized after the constructor completes.
  WRONG - Static variables are initialized once, when the class is first loaded by the JVM, which happens long before any specific object is created and its constructor runs.
- 3) Instance variables are assigned their default values (e.g., 0, false, null) before any instance initializers or constructors are run.
  CORRECT - The very first step of instantiation, after memory allocation, is

that the JVM assigns all instance fields their default values (e.g., 0 for `int`, `false` for `boolean`, `null` for objects).

- 4) Instance initializers are executed in the order they appear in the source code.
CORRECT - If there are multiple instance initializers and instance variable declarations, they are executed in the sequence they appear in the source code, from top to bottom.
- 5) It is valid for a class to have multiple instance initializer blocks.
CORRECT - A class can have more than one instance initializer block. They are executed in the order they are written in the file.

(28) (questionId: 102520, topic: ArrayList and Basic Collections)
What is the result of executing the following code?

```
import java.util.List;
import java.util.ArrayList;

public class Test {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.remove(2);
        System.out.println(list);
    }
}
```

Only one correct choice.

- 0) [1, 2]
CORRECT - This is a classic trick question about method overloading. The list ‘[1, 2, 3]‘ contains ‘Integer‘ objects. However, the call is ‘list.remove(2)‘, where ‘2‘ is a primitive ‘int‘. Java will choose the ‘remove(int index)‘ method signature over ‘remove(Object o)‘. Therefore, it removes the element at index 2, which is the value ‘3‘. The final list is ‘[1, 2]‘.
- 1) [1, 3]
WRONG - This would be the result if ‘list.remove(Integer.valueOf(2))‘ were called, which would remove the object with the value ‘2‘.
- 2) [2, 3]
WRONG - This would be the result if the element at index 0 were removed.
- 3) An ‘IndexOutOfBoundsException‘ is thrown.
WRONG - Index 2 is a valid index for a list of size 3, so no exception is thrown.

(29) (questionId: 101425, topic: StringBuilder and StringBuffer)
Which line of code, when inserted at ‘// INSERT‘, will result in both ‘boolean‘ variables being ‘true‘?

```
StringBuilder sb1 = new StringBuilder("A");
StringBuilder sb2 = new StringBuilder("A");
String s1 = new String("A");

// INSERT

boolean b1 = sb1.toString().equals(s1);
boolean b2 = sb1 == sb2;
```

Only one correct choice.

- 0) ‘sb2 = sb1;‘
  CORRECT - The line ‘sb2 = sb1;‘ makes ‘sb2‘ point to the exact same object as ‘sb1‘. After this line executes: ‘b1‘ checks if ‘sb1.toString()‘ (‘”A”‘) equals ‘s1‘ (‘”A”‘), which is ‘true‘. ‘b2‘ checks if ‘sb1‘ and ‘sb2‘ refer to the same object (‘sb1 == sb2‘), which is also ‘true‘ because of the assignment.
- 1) ‘sb1 = new StringBuilder(s1);‘
  WRONG - This reassigns ‘sb1‘ but leaves ‘sb2‘ pointing to its original object, so ‘sb1 == sb2‘ would be false.
- 2) ‘s1 = sb1.toString(); sb2 = sb1;‘
  WRONG - This works, but it’s more complex than necessary. The simplest line that achieves the goal is option 0.
- 3) It’s impossible to make both ‘true‘.
  WRONG - Option 0 demonstrates that it is possible.

(30) (questionId: 102928, topic: Try-Catch-Finally Blocks)
What is the final output of this program?

```
public class Test {
    public static void main(String[] args) {
        try {
            System.out.print("A");
            danger();
        } catch (Exception e) {
            System.out.print("B");
        } finally {
            System.out.print("C");
        }
    }
    static void danger() {
        try {
            throw new Error();
        } finally {
            System.out.print("D");
        }
    }
}
```

Only one correct choice.

- 0) ‘ADBC‘
  WRONG - The ‘catch‘ block in ‘main‘ is never entered.
- 1) ‘ADC‘
  WRONG - The ‘finally‘ block in ‘main‘ is also executed.
- 2) ‘AD‘ followed by an ‘Error‘ being thrown.
  RIGHT - The flow is: 1) ‘main‘ prints ’A’. 2) ‘danger()‘ is called. 3) ‘danger()‘ throws an ‘Error‘. 4) ‘danger()‘’s ‘finally‘ block executes, printing ’D’. 5) The ‘Error‘ propagates from ‘danger()‘ back to ‘main‘. 6) The ‘catch (Exception e)‘ block in ‘main‘ does NOT catch the ‘Error‘ (they are siblings). 7) ‘main‘’s ‘finally‘ block executes, printing ’C’. 8) The uncaught ‘Error‘ is then thrown from ‘main‘, terminating the thread. The total output before termination is ‘ADC‘.
- 3) ‘A‘ followed by an ‘Error‘ being thrown.
  WRONG - Both ‘finally‘ blocks are executed before the program terminates.

(31) (questionId: 100125, topic: Main Method and Command Line Arguments)
Consider the following code:

```
package com.test;
public class Runner {
    public static void main(String[] args) {
        System.out.println("OK");
    }
}
```

After compiling with ‘javac -d . com/test/Runner.java‘, you are in the ‘com/test‘ directory. You execute ‘java Runner‘. What is the result?
Only one correct choice.

- 0) It prints ”OK”.
  WRONG - This command will fail.
- 1) A ‘ClassNotFoundException‘ is thrown.
  WRONG - A ‘ClassNotFoundException‘ occurs when the JVM cannot find the requested class file on the classpath. Here, the JVM finds ‘Runner.class‘, but the internal package name doesn’t match the request, leading to a different error.
- 2) A ‘NoClassDefFoundError‘ is thrown.
  CORRECT - This is a tricky classpath issue. When you are in ‘com/test‘ and run ‘java Runner‘, you are telling the JVM to load a class named ‘Runner‘ from the default (unnamed) package. The JVM finds ‘Runner.class‘ in the current directory. However, upon loading it, it reads the bytecode and sees that the class is declared to be in the package ‘com.test‘. This mismatch between the requested package (default) and the actual package (‘com.test‘) causes a ‘NoClassDefFoundError‘. To run it correctly, you must be at the root of the classpath (‘.‘ in this case) and execute ‘java com.test.Runner‘.
- 3) A ‘SecurityException‘ is thrown.
  WRONG - This is a class loading issue, not a security issue.

(32) (questionId: 100227, topic: Packages, Classpath, and JARs)
Which of the following statements about ‘import‘ declarations are true? (Choose all that apply)
Multiple correct choices.

- 0) ‘import‘ statements are required to use any class outside the current package.
WRONG - `import` statements are a convenience. You can always use the fully qualified name of a class (e.g., `java.util.ArrayList`) instead of importing it. Also, classes in `java.lang` never need to be imported.

- 1) A static import can import all static members of a class using a wildcard (‘*‘).
CORRECT - The statement `import static com.example.MyConstants.*;` will import all accessible static members (fields and methods) from the `MyConstants` class, allowing them to be used without the class name qualifier.

- 2) Importing a package, such as ‘java.util.*‘, also imports its subpackages, like ‘java.util.concurrent‘.
WRONG - A wildcard import (`*`) is not recursive. It imports types from the specified package only, not from any of its subpackages.

- 3) Importing a class with the same simple name from two different packages requires one of them to be referred to by its fully qualified name.
CORRECT - If you try to import two classes with the same simple name from different packages (e.g., `import java.util.Date;` and `import java.sql.Date;`), the compiler will report an error if you try to use the simple name `Date`. You must use the fully qualified name for at least one of them to resolve the ambiguity.

- 4) ‘import‘ statements increase the size of the final ‘.class‘ file.
WRONG - `import` statements are only instructions for the compiler. They are not included in the compiled bytecode and do not affect the size of the final `.class` file.

(33) (questionId: 100327, topic: Java Coding Conventions and Javadoc)
What is the result of compiling and running this code?

```
public class TrickyScope {
    public static void main(String[] args) {
        int i = 0;
        if (true) {
            // The following comment looks like it closes the block
            /*
                System.out.println("Inside comment");
            }
            */
            i = 1;
        }
        System.out.println(i);
    }
```

```
}
```

Only one correct choice.

- 0) It fails to compile due to a syntax error with braces.
  WRONG - The code is syntactically correct because the closing brace inside the comment is ignored.

- 1) It compiles and prints ‘0‘.
  WRONG - The code inside the `if` block is executed.

- 2) It compiles and prints ‘1‘.
  CORRECT - The compiler ignores all content inside the `/* ... */` block. This includes the line ‘System.out.println(”Inside comment”);‘ and the closing brace ‘‘. The actual code flow is: `i` is set to 0. The `if(true)` block is entered. The comment is skipped. The line `i = 1;` is executed. The ‘if‘ block is closed by the real brace. Finally, ‘System.out.println(i)‘ prints the current value of `i`, which is 1.

- 3) It compiles but throws a runtime exception.
  WRONG - The code is simple and contains no operations that would cause a runtime exception.

(34) (questionId: 103229, topic: Lambda Expressions and Functional Interfaces)
Which of the following functional interface declarations will compile successfully? (Choose all that apply)
Multiple correct choices.

- 0) ‘@FunctionalInterface interface A int m(); default int n() return 0; ‘
  CORRECT - Interface ‘A‘ has one abstract method (‘m‘) and one ‘default‘ method. Default methods don’t count towards the abstract method total, so this is a valid functional interface.

- 1) ‘@FunctionalInterface interface B extends A ‘
  CORRECT - Interface ‘B‘ extends ‘A‘ and does not add any new abstract methods. It inherits the single abstract method from ‘A‘, so it remains a valid functional interface.

- 2) ‘@FunctionalInterface interface C ¡T¿ T m(T t); ‘
  CORRECT - A functional interface can have a generic abstract method. Interface ‘C‘ has only one abstract method, ‘¡T¿ T m(T t)‘, making it a valid functional interface.

- 3) ‘@FunctionalInterface interface D extends java.util.Comparator ‘
  WRONG - This is a trick. The ‘java.util.Comparator‘ interface is itself a functional interface (its single abstract method is ‘compare‘). An interface ‘D‘ that merely extends ‘Comparator‘ without adding new abstract methods would also be a valid functional interface. Therefore, this code *should* compile. Its inclusion as an incorrect choice in some mock exams is often considered a flaw in the question, as there is no rule preventing this.

- 4) ‘@FunctionalInterface interface E void m(); String toString(); ‘
  CORRECT - Interface ‘E‘ declares one abstract method, ‘m()‘. It also re-

declares ‘toString()‘. However, since ‘toString()‘ is a public method in ‘java.lang.Object‘, it does not count towards the abstract method limit. Thus, ‘E‘ is a valid functional interface.

(35) (questionId: 101529, topic: Classes and Objects Fundamentals)
You have an encapsulated ‘MutableDate‘ class. Which of the following getter method implementations for a ‘Person‘ class would risk breaking the encapsulation of the ‘Person‘ object’s state? (Choose all that apply)

```
// Assume MutableDate is a class like java.util.Date
// with public methods to change its state.
class MutableDate { /* ... setters ... */ }

class Person {
    private String name;
    private MutableDate birthDate;

    public Person(String name, MutableDate birthDate) {
        this.name = name;
        this.birthDate = birthDate;
    }

    // ... getters ...
}
```

Multiple correct choices.

- 0) ‘public MutableDate getBirthDate() return this.birthDate; ‘
CORRECT - This is known as ’leaking a reference’. The getter returns a direct reference to the internal, mutable `birthDate` object. The caller who receives this reference can then call methods on it to change its state (e.g., `person.getBirthDate().setMonth(10)`), thereby modifying the `Person` object’s internal state without going through its methods. This breaks encapsulation.

- 1) ‘public String getName() return this.name; ‘
WRONG - `String` objects in Java are immutable. Returning a reference to a `String` is safe because the caller cannot change the object. Encapsulation is preserved.

- 2) ‘public MutableDate getBirthDate() return new MutableDate(this.birthDate.getTime()); ‘
WRONG - This implementation performs a ’defensive copy’. It creates a brand new `MutableDate` object that is a copy of the internal one. The caller gets a reference to the copy, not the original. Any modifications to the returned object do not affect the `Person`’s internal state. This preserves encapsulation.

- 3) ‘public MutableDate getBirthDate() return (MutableDate) this.birthDate.clone(); ‘ (Assume ‘clone()‘ is implemented correctly for a deep copy).
WRONG - Similar to the previous option, using a proper `clone()` method to create a copy is another form of defensive copying that preserves encapsulation.

- 4) ‘public void printBirthDate() System.out.println(this.birthDate); ‘
  WRONG - This method does not return anything, so it does not provide the caller with a reference to the internal object. Encapsulation is preserved.

(36) (questionId: 100823, topic: Java Operators and Precedence)
What is the result of this code snippet?

```
int mask = 0x000F;
int value = 0x2222;
System.out.println(value & mask);
```

Only one correct choice.

- 0) 15
  WRONG - 15 is the decimal representation of the mask ‘0x000F‘, not the result of the ‘‘ operation.
- 1) 2
  CORRECT - This question tests bitwise operators and hexadecimal literals. The ‘‘ operator performs a bitwise AND.- ‘mask = 0x000F‘ in binary is ‘...0000 1111‘.- ‘value = 0x2222‘ in binary is ‘...0010 0010 0010 0010‘.a bitwise AND means the resulting bit is 1 only if the corresponding bits in both operands are 1.‘...0010 0010 0010 0010‘ (‘value‘)‘‘ ‘...0000 0000 0000 1111‘ (‘mask‘)‘=‘ ‘...0000 0000 0000 0010‘result in binary is ‘10‘, which is 2 in decimal.
- 2) 0
  WRONG - The result would be 0 only if the last four bits of ‘value‘ were all 0 (e.g., ‘0x2220‘).
- 3) 2222
  WRONG - This is the original ‘value‘, not the result of the bitwise AND operation.

(37) (questionId: 101327, topic: String Immutability and Operations)
Which statements are true about string concatenation using the ‘+‘ operator in a loop? (Choose all that apply)

```
String result = "";
for (int i=0; i<100; i++) {
    result += i; // Line 3
}
```

Multiple correct choices.

- 0) A new ‘String‘ object is created in each iteration of the loop.
  CORRECT - From a theoretical, pre-optimization perspective, each ‘+=‘ operation creates a new String object. The expression ‘result + i‘ would create a new string, and the reference ‘result‘ would be updated to point to it, leaving the old string for garbage collection. This is conceptually correct and highlights the inefficiency.
- 1) The compiler automatically replaces this code with ‘StringBuilder‘ for efficiency.
  CORRECT - In practice, modern Java compilers are smart enough to optimize

this specific pattern. The compiler rewrites the loop to use a single ‘StringBuilder‘ instance, appending to it in each iteration and then converting it to a ‘String‘ once after the loop. This is a crucial real-world detail. The question is tricky because both statements describe the situation from different but valid perspectives (theoretical vs. actual compiled code).

- 2) This is the most memory-efficient way to build a string.
  WRONG - Without compiler optimization, this is one of the *least* memory-efficient ways. Using an explicit ‘StringBuilder‘ is far better.
- 3) After the loop, the original ‘result‘ object (the empty string) has been modified to contain the final value.
  WRONG - ‘String‘ objects are immutable. The original empty string object is never modified. The ‘result‘ *reference* is repeatedly reassigned to point to new ‘String‘ objects.

(38) (questionId: 101828, topic: Garbage Collection and Object Lifecycle)
Analyze the following code. At Point Y, how many ‘java.lang.String‘ objects are eligible for GC, assuming no string pooling optimizations for literals?

```
public class StringGC {
    public static void main(String[] args) {
        String s1 = "one";
        String s2 = new String("two");
        String s3 = "three";
        s3 = s1;
        s1 = s2;
        s2 = null;

        // What about the object referred to by s1 originally ("one")?
        // What about the object referred to by s2 originally ("two")?
        // What about the object referred to by s3 originally ("three")?
        // Point Y
    }
}
```

Only one correct choice.

- 0) 0
  WRONG - This would be the answer if standard String pooling were in effect, as all literals would be retained in the pool. However, the question explicitly tells you to ignore this optimization.
- 1) 1
  CORRECT - The question requires you to ignore string pooling, meaning each literal declaration acts like `new String(...)`. Let’s trace: 1. `s1` points to an object for ’one’ (O1). 2. `s2` points to an object for ’two’ (O2). 3. `s3` points to an object for ’three’ (O3). 4. `s3 = s1` makes `s3` point to O1. The only reference to O3 is now gone, so the ’three’ object is eligible for GC. 5. `s1 = s2` makes `s1` point to O2. 6. `s2 = null`. At Point Y, only the original ’three’ object is unreferenced.

- 2) 2
  WRONG - Only the 'three' object has lost all its references.
- 3) 3
  WRONG - The 'one' and 'two' objects are still referenced by `s3` and `s1` respectively.

(39) (questionId: 103022, topic: Throwing and Creating Exceptions)
What is the result of attempting to compile and run the following code?

```
public class StaticFail {
    static {
        if (true) {
            throw new RuntimeException("Initialization failed");
        }
    }

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

Only one correct choice.

- 0) The code compiles and prints 'Hello'.
  WRONG - The static initializer runs before the `main` method is called. Since it fails, `main` never executes.
- 1) The code does not compile.
  WRONG - The code is syntactically correct and will compile successfully. The error occurs at runtime.
- 2) The code compiles, but throws a 'RuntimeException' when run.
  WRONG - While a `RuntimeException` is the initial cause, the JVM wraps any exception thrown from a static initializer block in an `ExceptionInInitializerError`.
- 3) The code compiles, but throws an 'ExceptionInInitializerError' when run.
  CORRECT - When a class is first used, the JVM runs its static initializer block. If an exception is thrown from this block, the JVM catches it and throws a new `ExceptionInInitializerError`, which signals that a failure occurred during static initialization. This error prevents the class from being used and the `main` method from running.
- 4) The code compiles, but throws a 'NoClassDefFoundError' when run.
  WRONG - A `NoClassDefFoundError` typically occurs on a *second* attempt to use a class that previously failed to initialize. The first failure is always an `ExceptionInInitializerError`.

(40) (questionId: 101325, topic: String Immutability and Operations)
What is the output of the following code?

```
String text = "a.b.c";
String[] parts = text.split(".");
```

```
System.out.println(parts.length);
```

Only one correct choice.

- 0) 0
  CORRECT - This is a common trap. The `split()` method takes a regular expression (regex) as its argument. In regex, a single dot (‘.‘) is a special metacharacter that matches *any character*. Therefore, ‘text.split(”.”)‘ is splitting the string on every single character. This results in an array of empty strings. By default, trailing empty strings are removed, resulting in an empty array of length 0. To split on a literal dot, you must escape it: ‘text.split(”
  .”)‘.
- 1) 1
  WRONG - The split does not produce one part.
- 2) 3
  WRONG - This would be the result if you correctly split on the literal dot using ‘text.split(”
  .”)‘.
- 3) An exception is thrown at runtime.
  WRONG - No exception is thrown, this is valid (though likely unintended) behavior.

(41) (questionId: 103653, topic: Passing Data Among Methods)
What is the output of this code which passes and returns references?

```
class Num { public int val; }

public class ReturnTest {
    public static void main(String[] args) {
        Num a = new Num(); a.val = 1;
        Num b = new Num(); b.val = 2;
        b = process(a, b);
        System.out.println(a.val + "," + b.val);
    }

    public static Num process(Num x, Num y) {
        x.val = y.val;
        y = new Num();
        y.val = 3;
        return y;
    }
}
```

Only one correct choice.

- 0) ‘1,2‘
  WRONG - Both ‘a‘ and ‘b‘ are changed.
- 1) ‘2,3‘
  CORRECT - 1. ‘main‘ has ‘a‘ (val=1) and ‘b‘ (val=2). 2. ‘process‘ is called.

'x' points to 'a', 'y' points to 'b'. 3. 'x.val = y.val;' copies the value from 'b''s object to 'a''s object. 'a.val' is now 2. 4. 'y = new Num(); y.val = 3;' creates a new 'Num' object and makes the local 'y' parameter point to it. 5. 'return y;' returns this new object. 6. Back in 'main', 'b = process(...)' reassigns 'b' to the returned object. 'b.val' is now 3. Final state: 'a.val' is 2, 'b.val' is 3.

- 2) '2,2'
  WRONG - The variable 'b' in 'main' is reassigned to the new object returned by the method.
- 3) '1,3'
  WRONG - The state of the object 'a' is modified inside the 'process' method via the 'x' reference.

(42) (questionId: 103357, topic: Date and Time API (java.time))
Which of the following lines of code, if executed independently, will result in a runtime exception? (Choose all that apply)

```
// Assume all necessary imports from java.time and java.time.temporal
```

Multiple correct choices.

- 0) 'LocalDate.of(2025, 13, 1);'
  CORRECT - Throws 'DateTimeException' because 13 is not a valid month.
- 1) 'Duration.between(LocalDate.now(), LocalDateTime.now());'
  CORRECT - Throws 'DateTimeException' or 'UnsupportedTemporalTypeException'. 'Duration' measures time-based amounts (like seconds) and requires nanosecond precision. 'LocalDate' does not contain time information, so a 'Duration' cannot be calculated between it and a 'LocalDateTime'.
- 2) 'Period.of(1, 1, 1).plus(Duration.ofHours(1));'
  CORRECT - Throws 'UnsupportedTemporalTypeException'. A 'Period' is date-based. You cannot add a time-based 'Duration' to it.
- 3) 'LocalTime.now().truncatedTo(ChronoUnit.DAYS);'
  CORRECT - Throws 'UnsupportedTemporalTypeException'. A 'LocalTime' has no concept of 'DAYS'. You cannot truncate a time object to a unit that is larger than the units it contains.
- 4) 'Period.ofMonths(12).normalized();'
  WRONG - This code is valid. It creates a 'Period' of 12 months, and 'normalized()' converts it to 'P1Y' (1 year). No exception is thrown.

(43) (questionId: 100021, topic: Java Environment and Fundamentals)
Consider the following directory structure and files:

```
/project
    /src
        /com
            /example
                MyClass.java
    /bin
```

The file `MyClass.java` contains:

```
package com.example;

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

You are currently in the `/project` directory. Which sequence of commands will successfully compile and run `MyClass`?
Only one correct choice.

- 0)

  ```
  javac src/com/example/MyClass.java
  java -cp src com.example.MyClass
  ```

  WRONG - This sequence is not standard practice. The first command places the compiled `.class` file inside the `src` directory, mixing source files with binaries. While the second command would correctly run it from there, this approach is discouraged. Option 2 represents the correct, professional separation of concerns.

- 1)

  ```
  javac src/com/example/MyClass.java
  java -cp bin com.example.MyClass
  ```

  WRONG - The first command compiles the class and places the output in `src/com/example/MyClass.class`. The second command then looks for the class in the `bin` directory, where it does not exist. This will result in a `ClassNotFoundException`.

- 2)

  ```
  javac -d bin src/com/example/MyClass.java
  java -cp bin com.example.MyClass
  ```

  CORRECT - This is the standard and correct procedure. The `javac -d bin` command compiles the source file and places the resulting `.class` file in the specified destination directory (`bin`), creating the necessary package subdirectories (`com/example`). The `java -cp bin` command then correctly sets the classpath to the `bin` directory, allowing the JVM to find and run `com.example.MyClass`.

- 3)

  ```
  javac -d bin src/com/example/MyClass.java
  java com.example.MyClass
  ```

  WRONG - The compilation command is correct. However, the run command fails because it doesn't specify a classpath. The JVM defaults to the current directory (`.`), where it looks for `./com/example/MyClass.class`, which does not exist. The classpath must be set to `bin` using the `-cp` flag.

(44) (questionId: 100525, topic: Type Conversion and Casting)
Which of the following code snippets will compile successfully? (Choose all that apply)
Multiple correct choices.

- 0)

  ```
  short s = 10;
  s = s + 5;
  ```

  WRONG - This fails to compile. ‘s + 5‘ results in an ‘int‘. Assigning an ‘int‘ back to a ‘short‘ requires an explicit cast.

- 1)

  ```
  char c = ’a’;
  c += 5;
  ```

  CORRECT - This compiles. The compound assignment operator ‘+=‘ includes an implicit cast, so this is equivalent to ‘c = (char)(c + 5);‘.

- 2)

  ```
  final byte b1 = 10;
  final byte b2 = 20;
  byte b3 = b1 + b2;
  ```

  CORRECT - This compiles. Since both ‘b1‘ and ‘b2‘ are ‘final‘ variables initialized with literals, they are compile-time constants. The expression ‘b1 + b2‘ is a constant expression evaluated by the compiler to ‘30‘. Since ‘30‘ fits in a ‘byte‘, the assignment is allowed without a cast.

- 3)

  ```
  float f = 1.0f;
  double d = f;
  ```

  CORRECT - This compiles. Assigning a ‘float‘ to a ‘double‘ is a widening conversion and is always allowed.

(45) (questionId: 100623, topic: Wrapper Classes and Autoboxing/Unboxing)
Which of the following lines will compile without errors? (Choose all that apply)
Multiple correct choices.

- 0) `Integer i = new Integer(null);`
  WRONG (Will not compile) - The call `new Integer(null)` is ambiguous. The compiler cannot decide whether to call the `Integer(int)` constructor or the `Integer(String)` constructor, so it results in a compilation error.

- 1) `Double d = null; double d2 = d;`
  CORRECT (Will compile) - The syntax is valid. `Double d = null;` is fine. `double d2 = d;` is also syntactically valid; the compiler allows the unboxing assignment. Note: This line would throw a `NullPointerException` at *runtime*, but the question asks about compilation, and it compiles successfully.

- 2) `Byte b = 25;`
  CORRECT (Will compile) - This is a special case of autoboxing. While you can’t box an `int` variable into a `Byte`, you *can* assign an `int` literal if it’s a compile-time constant that fits within the range of a `byte` (-128 to 127). The compiler performs an implicit narrowing conversion before boxing.

- 3) `Short s = new Short((short)10);`
  CORRECT (Will compile) - This is a straightforward and valid use of the `Short` constructor, which takes a primitive `short` as an argument. The cast `(short)10` is valid.

- 4) `long l = new Integer(100);`
  CORRECT (Will compile) - This demonstrates unboxing followed by widening. The `new Integer(100)` object is first unboxed to a primitive `int 100`. Then, this `int` is widened to a `long` to be assigned to the variable `l`. This is a valid sequence of conversions.

(46) (questionId: 101222, topic: Enums)
Examine the following code. What is the result?

```
public enum Operation {
    PLUS {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS {
        public double apply(double x, double y) { return x - y; }
    };
    public abstract double apply(double x, double y);
}

class Test {
    public static void main(String[] args) {
        System.out.println(Operation.PLUS.apply(5, 3));
    }
}
```

Only one correct choice.

- 0) ‘8.0‘
  CORRECT - This pattern is a valid and powerful use of enums. The enum declares an `abstract` method, which forces every enum constant to provide a concrete implementation in a constant-specific class body. The call `Operation.PLUS.apply(5, 3` invokes the specific implementation for the `PLUS` constant, returning `5 + 3`, which is `8.0`.

- 1) The code fails to compile because an enum cannot be ‘abstract‘.
  WRONG - An enum itself cannot be declared `abstract`, but it *can* contain abstract methods as long as all of its constants provide implementations.

- 2) The code fails to compile because ‘apply‘ is not defined for the ‘Operation‘ enum itself.
  WRONG - The code compiles precisely because every constant *does* provide

an implementation, fulfilling the abstract contract.

- 3) The code fails to compile because an enum constant cannot provide a method implementation.
  WRONG - An enum constant can, and in this case must, provide a method implementation.

(47) (questionId: 100328, topic: Java Coding Conventions and Javadoc)
Which of the following code snippets will fail to compile due to issues with comment syntax? (Choose all that apply)
Multiple correct choices.

- 0)

  ```
  int x = 10; //* A special comment */
  ```

  WRONG - This will compile. The `//*` starts a single-line comment. The subsequent characters, `*` and `*/`, are just part of the ignored comment text.

- 1)

  ```
  String s = "This contains a comment end: */";
  ```

  WRONG - This will compile. The `*/` sequence is inside a string literal, so the compiler treats it as part of the string's text, not as a comment terminator.

- 2)

  ```
  /* Is this /* nested comment */ valid? */
  int y = 20;
  ```

  CORRECT - This will fail to compile. Java does not support nested multi-line comments. The first `*/` encountered (after 'comment') closes the entire comment block. This leaves the text ' valid? */' as un-commented, syntactically invalid code.

- 3)

  ```
  // Another comment \
  int z = 30;
  ```

  WRONG - This will compile. The backslash `\` at the end of the line is simply the last character in the single-line comment. It has no special meaning like line continuation. The next line, `int z = 30;`, is treated as a separate, valid statement.

(48) (questionId: 102622, topic: Generics)
Which of these lines causes a compilation error?

```
import java.util.*;

class Mammal {}
class Primate extends Mammal {}
class Human extends Primate {}

public class Test {
```

```
    public static void main(String[] args) {
        List<? super Primate> primates = new ArrayList<Mammal>(); // Line 1
        primates.add(new Human());                                // Line 2
        primates.add(new Primate());                              // Line 3
        primates.add(new Mammal());                               // Line 4
    }
}
```

Only one correct choice.

- 0) Line 1
  WRONG - Line 1 is a valid lower-bounded wildcard assignment because ‘Mammal‘ is a superclass of ‘Primate‘.
- 1) Line 2
  WRONG - Line 2 is valid. The list is guaranteed to accept ‘Primate‘ or any subtype, and ‘Human‘ is a subtype of ‘Primate‘.
- 2) Line 3
  WRONG - Line 3 is valid. The list can accept ‘Primate‘ itself.
- 3) Line 4
  CORRECT - This is a compilation error. ‘List¡? super Primate¿‘ is a ’consumer’ that can accept ‘Primate‘ and its subtypes. It cannot accept a ‘Mammal‘, which is a supertype. The compiler prevents this because the actual list object could be an ‘ArrayList¡Primate¿‘, into which you cannot add a ‘Mammal‘.

(49) (questionId: 101025, topic: Looping Constructs (for, while, do-while))
What is the output of the following code?

```
int[] a = {1, 2, 3};
int[] b = {4, 5, 6};
for (int i : a, j : b) {
    System.out.print(i + j);
}
```

Only one correct choice.

- 0) 579
  WRONG - The code does not compile, so it cannot produce any output.
- 1) 142536
  WRONG - The code does not compile.
- 2) The code does not compile.
  CORRECT - The syntax of the enhanced for loop (‘for-each‘) is ‘for (Type variable : arrayOrIterable)‘. It does not support declaring multiple loop variables or iterating over multiple collections in a single statement as shown. This is a syntax error, so the code fails to compile.
- 3) The code throws a runtime exception.
  WRONG - This is a compile-time error, not a runtime exception.

(50) (questionId: 103558, topic: Method Design and Variable Arguments)
Which of the following method declarations are valid in a concrete (non-abstract) class? (Choose all that apply)
Multiple correct choices.

- 0) ‘private final static void methodA();‘
WRONG - This is an illegal declaration. A method declared with a semicolon instead of a body ‘‘ is implicitly ‘abstract‘. However, ‘static‘ methods cannot be ‘abstract‘.

- 1) ‘protected abstract void methodB();‘
WRONG - This is an ‘abstract‘ method declaration. A concrete (non-abstract) class cannot contain any ‘abstract‘ methods.

- 2) ‘public final synchronized void methodC(String... s) ‘
CORRECT - This is a valid method declaration. It combines ‘public‘, ‘final‘, and ‘synchronized‘ modifiers, has a ‘void‘ return type, and a valid varargs parameter with a method body ‘‘.

- 3) ‘void methodD(final int... x) ‘
CORRECT - This is a valid method declaration. It uses default (package-private) access, and correctly declares a ‘final‘ varargs parameter. The ‘final‘ keyword on a varargs parameter makes the array reference itself final (i.e., you cannot reassign the parameter to a new array).

- 4) ‘static System.out.println(”I am not a method.”); ‘
WRONG - This is a ‘static initializer block‘, which is a valid construct in a class, but it is not a *method declaration*.

(51) (questionId: 101122, topic: Break, Continue, and Labels)
What is the result of attempting to compile this code?

```
public class InvalidContinue {
    public static void main(String[] args) {
        myLabel: {
            if (true) {
                continue myLabel;
            }
        }
    }
}
```

Only one correct choice.

- 0) It compiles successfully.
WRONG - The code contains a compilation error related to the use of ‘continue‘.

- 1) It fails to compile because ‘myLabel‘ is not on a loop.
CORRECT - The ‘continue‘ statement, whether labeled or not, can only be used inside a loop. Its function is to proceed to the next iteration, which is a concept that doesn’t apply to a simple code block. Since ‘myLabel‘ is

not attached to a ‘for‘, ‘while‘, or ‘do-while‘ loop, the ‘continue myLabel;‘ statement is a compilation error.

- 2) It fails to compile because a label cannot be on a simple block.
  WRONG - A label *can* be placed on a simple block. The error is with the ‘continue‘, not the label.
- 3) It fails to compile because of an unreachable statement.
  WRONG - While ‘continue‘ does create an unconditional jump, the primary compilation error is the illegal context for ‘continue‘, not unreachable code.

(52) (questionId: 100625, topic: Wrapper Classes and Autoboxing/Unboxing)
What is the output of the following code?

```
import java.util.ArrayList;
import java.util.List;

public class Test {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.remove(new Integer(2));
        System.out.println(list);
    }
}
```

Only one correct choice.

- 0) `[1, 2]`
  WRONG - This would be the result if the element at index 2 (the value 3) were removed.
- 1) `[1, 3]`
  CORRECT - The code calls `list.remove(new Integer(2));`. Because the argument is an `Integer` object, Java invokes the `remove(Object o)` method, not the `remove(int index)` method. This method searches the list for the first element that is `.equals()` to the given object and removes it. The list contains `[1, 2, 3]`, and the object `Integer(2)` is found and removed, resulting in the list `[1, 3]`.
- 2) `[2, 3]`
  WRONG - This would be the result if the element at index 0 (the value 1) were removed.
- 3) An `IndexOutOfBoundsException` occurs.
  WRONG - An `IndexOutOfBoundsException` would occur if we tried to call `remove(int)` with an index that is too large, for example `list.remove(3)`. The call here is valid and successful.

(53) (questionId: 101725, topic: Static Members and ’this’ Keyword)
What is the output of this code? This tests understanding of ‘this‘ within inner

classes.

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

    class Inner {
        String name = "Inner";
        void printNames() {
            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();
    }
}
```

Only one correct choice.

- 0) Inner
  RIGHT - This question tests name shadowing and the special `Outer.this` syntax. In `printNames()`:
  1. `name`: Refers to the name in the closest scope, the `Inner` class's field. Prints "Inner".
  2. `this.name`: `this` refers to the current `Inner` object. This also prints "Inner".
  3. `Outer.this.name`: This is the special syntax required to access a member of the enclosing `Outer` instance from within the `Inner` instance. It prints "Outer".
- 1) Outer
  WRONG - This would be the case if there were no shadowing.
- 2) Inner
  WRONG - The unqualified `name` refers to the inner class's field, not the outer's.
- 3) The code fails to compile.
  WRONG - The syntax is valid for inner classes.

(54) (questionId: 100223, topic: Packages, Classpath, and JARs)
The classpath is set to '-cp dirA:dirB'. 'dirA' contains 'com/test/Tool.class' version 1. 'dirB' contains 'com/test/Tool.class' version 2. A program uses 'com.test.Tool'. Which version of the class will be loaded by the JVM?
Only one correct choice.

- 0) Version 1 from 'dirA'.
  CORRECT - The JVM searches for classes by iterating through the classpath entries from left to right. It will first search in `dirA`. It will find `dirA/com/test/Tool.class`, load it, and stop searching. The version in `dirB` will be ignored.

- 1) Version 2 from ‘dirB‘.
WRONG - The class will be found and loaded from `dirA` before the JVM ever gets to search in `dirB`.
- 2) A compilation error will occur.
WRONG - This is a runtime class loading behavior, not a compilation error. The compiler would also use the first version it finds.
- 3) A runtime error will occur due to the conflict.
WRONG - The JVM does not consider this a conflict or an error. It deterministically loads the first version of the class it finds based on the classpath order. This can cause hard-to-diagnose bugs but is not a runtime error.

(55) (questionId: 101428, topic: StringBuilder and StringBuffer)
Which of these method calls can throw a ‘StringIndexOutOfBoundsException‘? (Choose all that apply)

```
StringBuilder sb = new StringBuilder("abc");
```

Multiple correct choices.

- 0) ‘sb.delete(1, 4);‘
CORRECT - For ‘delete(start, end)‘, the ‘end‘ index cannot be greater than the length. ‘sb‘ has length 3. An ‘end‘ of 4 is out of bounds.
- 1) ‘sb.insert(4, ”d”);‘
CORRECT - For ‘insert(offset, str)‘, the ‘offset‘ cannot be greater than the length. ‘sb‘ has length 3. An ‘offset‘ of 4 is out of bounds.
- 2) ‘sb.replace(0, 5, ”x”);‘
CORRECT - For ‘replace(start, end, str)‘, the ‘end‘ index cannot be greater than the length. ‘sb‘ has length 3. An ‘end‘ of 5 is out of bounds.
- 3) ‘sb.setCharAt(3, ’d’);‘
CORRECT - For ‘setCharAt(index, ch)‘, the ‘index‘ must be less than the length. ‘sb‘ has length 3. An ‘index‘ of 3 is out of bounds (valid indices are 0, 1, 2).

(56) (questionId: 102824, topic: Exception Hierarchy and Types)
What is the result of attempting to compile this code?

```
public class Test {
    public static void main(String[] args) {
        throw new String("This is an error");
    }
}
```

Only one correct choice.

- 0) It compiles, but throws a ‘ClassCastException‘ at runtime.
WRONG - The error is caught at compile time.
- 1) It compiles, but throws a ‘RuntimeException‘ at runtime.
WRONG - The error is caught at compile time.

- 2) It compiles, but throws an ‘Error‘ at runtime.
  WRONG - The error is caught at compile time.
- 3) It does not compile.
  RIGHT - The `throw` statement requires an object that is an instance of `java.lang.Throwable` or one of its subclasses. The class `java.lang.String` does not extend `Throwable`. Therefore, this is a syntax error that the compiler will catch, resulting in a compilation failure.