1Z0-808 Mock Exam Solutions

ExamId: 100

August 5, 2025

(1) (questionId: 101125, topic: Break, Continue, and Labels)
What is the output of the following code fragment?

```
int val = 0;
loop1:
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 4; j++) {
        val++;
        if (j >= 1) {
            continue loop1;
        }
    }
}
System.out.println(val);
```

Only one correct choice.

- 0) 2
  WRONG - The counter is incremented more than twice.
- 1) 3
  CORRECT - (Note: The actual output of this code is 4. The provided answer key appears to be incorrect). Let's trace: 'val' is 0. 'i=0, j=0': 'val' becomes 1. 'j¿=1' is false. 'i=0, j=1': 'val' becomes 2. 'j¿=1' is true, 'continue loop1'. 'i=1, j=0': 'val' becomes 3. 'j¿=1' is false. 'i=1, j=1': 'val' becomes 4. 'j¿=1' is true, 'continue loop1'. The final value is 4. For the answer to be 3, the second increment on the last pass ('val++' when 'i=1, j=1') would have to not happen, which is contrary to the code.
- 2) 4
  WRONG - Although this is the technically correct answer from a trace, it is not the keyed answer.
- 3) 8
  WRONG - The 'continue' statement limits the number of increments.

(2) (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.

(3) (questionId: 100226, topic: Packages, Classpath, and JARs)
What happens if a 'package-info.java' file contains a class declaration?
Only one correct choice.

- 0) The class is associated with the package as metadata.
  WRONG - This file is for package-level metadata, but a class declaration is not a valid form of metadata for this file.

- 1) It is treated like any other class in the package.
  WRONG - The file is treated specially by the compiler and has restrictions on its content.

- 2) A compilation error occurs.
  CORRECT - The file `package-info.java` is reserved for a single purpose: to contain a package declaration, package-level annotations, and a package-level Javadoc comment. It is a compilation error to declare any type (class, interface, or enum) within this file.

- 3) A runtime error occurs when the package is loaded.
  WRONG - The error is a violation of the language syntax rules and is caught by the compiler at compile time.

(4) (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.

(5) (questionId: 102728, topic: Sorting and Searching Collections (Comparable, Comparator))
What is the output of this code?

```
List<String> data = new ArrayList<>();
data.add("C");
data.add(null);
data.add("A");
data.sort(Comparator.nullsFirst(Comparator.naturalOrder()));
System.out.println(data);
```

Only one correct choice.

- 0) ‘[null, A, C]‘
  RIGHT - The static method `Comparator.nullsFirst()` is a comparator adaptor. It wraps another comparator (in this case, `Comparator.naturalOrder()`) and adds the behavior that any `null` elements are considered 'smaller' than non-null elements and should appear at the beginning of the sorted list. The remaining non-null elements ('A', 'C') are then sorted by the wrapped comparator, resulting in `[null, A, C]`.
- 1) ‘[A, C, null]‘
  WRONG - This would be the result of using `Comparator.nullsLast()`.
- 2) A ‘NullPointerException‘ is thrown.
  WRONG - A `NullPointerException` would be thrown if you used `Comparator.naturalOrde` directly on a list containing nulls. The entire purpose of `Comparator.nullsFirst()` is to prevent this exception by handling nulls gracefully.
- 3) The code does not compile.
  WRONG - The code is valid and compiles without issue.

(6) (questionId: 100222, topic: Packages, Classpath, and JARs)
You are in the directory ‘/app/bin/com/corp/‘, which contains ‘Main.class‘. The class is declared in package ‘com.corp‘. You run ‘java Main‘. What is the result?
Only one correct choice.

- 0) The program runs successfully.
  WRONG - The command fails for a very specific reason related to packages and the classpath.
- 1) A ‘ClassNotFoundException‘ is thrown.

WRONG - A `ClassNotFoundException` is a checked exception typically thrown by reflective operations like `Class.forName()`. The error from the JVM launcher is different.

- 2) A 'NoClassDefFoundError' is thrown with a message about 'com/corp/Main' being found in the wrong place.
  CORRECT - This is a classic exam trap. You must run a Java class from a directory that is the root of the package structure, not from within the package directory itself. By running `java Main`, you're asking the JVM to run a class named `Main` in the *default package*. The JVM finds `Main.class`, but sees that its code declares it to be in package `com.corp`. This mismatch between the requested package (default) and the actual package (`com.corp`) results in a `NoClassDefFoundError`. The correct way is to go to `/app/bin` and run `java com.corp.Main`.
- 3) A 'SecurityException' is thrown.
  WRONG - A `SecurityException` relates to violations of the Java security policy, which is not relevant here.

(7) (questionId: 103651, topic: Passing Data Among Methods)
What is the output of the following program?

```
public class ArrayOfObjects {
    static class Bulb { boolean on = false; }

    public static void main(String[] args) {
        Bulb[] bulbs = {new Bulb(), new Bulb()};
        turnOn(bulbs);
        System.out.println(bulbs[0].on + "," + bulbs[1].on);
    }

    public static void turnOn(Bulb[] lights) {
        lights[0].on = true;
        lights[1] = new Bulb();
        lights[1].on = true;
    }
}
```

Only one correct choice.

- 0) 'true,true'
  CORRECT - Let's trace carefully. 1. The 'turnOn' method receives a reference to the 'bulbs' array. 2. 'lights[0].on = true;' modifies the first 'Bulb' object in that array. This change is visible in 'main'. 3. 'lights[1] = new Bulb();' replaces the reference at index 1 of the array with a reference to a *new* 'Bulb' object. The original array is now modified. 4. 'lights[1].on = true;' modifies this new 'Bulb' object. 5. Back in 'main', 'bulbs[0]' is the first bulb, which is now on. 'bulbs[1]' refers to the *new* bulb created and modified in the method, which is also on. The output is 'true,true'. (Note: The provided answer key for this question was likely in error, as this is the definite behavior in Java).

- 1) ‘true,false‘
  WRONG - For this to be the output, the change to the second bulb would have to be invisible to ‘main‘. However, modifying the contents of an array (‘lights[1] = ...‘) is a state change to the array object itself, which is visible to the caller.
- 2) ‘false,true‘
  WRONG - The first bulb is definitely turned on.
- 3) ‘false,false‘
  WRONG - Both bulbs end up being turned on.

(8) (questionId: 100427, topic: Primitive Data Types and Literals)
Which statements are true about division in Java? (Choose all that apply)
Multiple correct choices.

- 0) Dividing a non-zero floating-point number by ‘0.0‘ results in ‘Infinity‘ or ‘-Infinity‘ and does not throw an exception.
  CORRECT - According to the IEEE 754 standard for floating-point arithmetic, dividing a positive or negative non-zero number by zero results in ‘Infinity‘ or ‘-Infinity‘, respectively. This is a defined operation and does not throw an exception.
- 1) Dividing any integer by ‘0‘ will always result in a compile-time error.
  WRONG - Dividing an integer by the *literal* ‘0‘ (e.g., ‘x / 0‘) is a compile-time error. However, dividing by an integer *variable* that happens to be zero at runtime (e.g., ‘int y = 0; x / y;‘) causes a runtime ‘ArithmeticException‘.
- 2) The expression ‘0.0 / 0.0‘ evaluates to ‘NaN‘ (Not a Number).
  CORRECT - The expression ‘0.0 / 0.0‘ is an indeterminate form in floating-point math. The IEEE 754 standard specifies that this operation should result in ‘NaN‘ (Not a Number).
- 3) The expression ‘10 / 4‘ evaluates to ‘2.5‘.
  WRONG - The expression ‘10 / 4‘ involves two integers, so Java performs integer division. The result is ‘2‘, with the fractional part (‘.5‘) being truncated. To get ‘2.5‘, at least one operand would need to be a floating-point type, e.g., ‘10.0 / 4‘.

(9) (questionId: 103351, topic: Date and Time API (java.time))
What is the result of attempting to compile and run the following code? This question checks your knowledge of object instantiation rules for the Date-Time API.

```
import java.time.LocalDate;

public class ConstructorTest {
    public static void main(String[] args) {
        LocalDate date = new LocalDate(2025, 8, 2);
        System.out.println(date);
    }
}
```

Only one correct choice.

- 0) It prints '2025-08-02'.
  WRONG - The code will not run because it does not compile.
- 1) It throws a 'DateTimeException' at runtime.
  WRONG - The error is a compilation error, not a runtime exception.
- 2) It fails to compile.
  CORRECT - This is a fundamental rule for the 'java.time' API. Classes like 'LocalDate', 'LocalTime', 'Period', etc., have private constructors. You cannot use the 'new' keyword to create an instance. The compiler will report an error, such as 'LocalDate(...) has private access'. You must use static factory methods (e.g., 'LocalDate.of(...)').
- 3) It prints a reference to the object.
  WRONG - The code fails to compile, so it cannot be executed to print anything.

(10) (questionId: 103123, topic: Try-with-Resources)
What is the output of this code?

```
class R implements AutoCloseable {
    public R() throws Exception { throw new Exception("R_INIT"); }
    public void close() { System.out.print("R_CLOSE"); }
}
public class TestFinal {
    public static void main(String[] args) {
        try (R r = new R()) {
            System.out.print("TRY");
        } catch (Exception e) {
            System.out.print(e.getMessage());
        } finally {
            System.out.print("_FINAL");
        }
    }
}
```

Only one correct choice.

- 0)

  'R_INIT_FINAL'

  CORRECT - 1. The code attempts to create resource `r`. 2. The constructor `new R()` immediately throws an `Exception` with message 'R$_I NIT'.3.Becausetheresourcewa$

- 1)

  'TRY_R_CLOSE_FINAL'

  WRONG - The `try` block is never entered and `close()` is never called.

- 2)

`'R_INIT_R_CLOSE_FINAL'`

WRONG - The `close()` method is not called if the resource constructor fails.

- 3)

  `'R_INIT'`

  WRONG - The `finally` block is guaranteed to execute.

(11) (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.

(12) (questionId: 101623, topic: Constructors and Initialization Blocks)
What is the output of this program?

```
public class StaticForward {
    static {
        System.out.print(x + " ");
    }
    private static int x = initX();
    static {
        System.out.print(x + " ");
    }

    private static int initX() {
        System.out.print("initX ");
        return 10;
    }

    public static void main(String[] args) {
        // Class loading is triggered by main method lookup
    }
}
```

Only one correct choice.

- 0) 0 initX 10
  RIGHT - This is a tricky static forward reference. The static initialization process runs top-to-bottom.
  1. Static variables get default values. `x` is 0.
  2. The first static block runs. It prints the current value of `x`, which is 0.
  3. The static variable initializer for `x` runs. It calls `initX()`.
  4. The `initX()` method runs, printing "initX ". It returns 10. The variable `x` is now assigned the value 10.
  5. The second static block runs. It prints the current value of `x`, which is 10.
  The final output is '0 initX 10 '.
- 1) initX 10 10
  WRONG - The first static block runs before `initX()` is ever called.
- 2) The code fails to compile due to illegal forward reference.
  WRONG - Just like with instance variables, a simple read of a static variable before its textual declaration is allowed; it will just use the default value.
- 3) initX 0 10
  WRONG - This shows an incorrect order of operations.

(13) (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.

(14) (questionId: 101324, topic: String Immutability and Operations)
How many ‘String‘ objects are created in the following code, not including any pre-existing literals in the string pool?

```
String s1 = new String("Hello");
String s2 = " World";
String s3 = s1 + s2;
```

Only one correct choice.

- 0) 1
  WRONG - More than one object is created.
- 1) 2
  WRONG - More than two objects are created.
- 2) 3
  CORRECT - This is a classic trick question. Let’s count them, assuming neither literal existed before: 1. ‘”Hello”‘: The string literal itself is created and placed in the String Constant Pool. 2. ‘new String(”Hello”)‘: The ‘new‘ keyword creates a second ‘String‘ object on the heap, which is what ‘s1‘ references. 3. ‘s1 + s2‘: The concatenation happens at runtime. This process creates a ‘StringBuilder‘ object behind the scenes, and its ‘toString()‘ method creates a third ‘String‘ object on the heap. So, 3 new String objects are created by this code. Note: the literal ‘” World”‘ would also be created in the pool, making a total of 4, but exam questions on this topic are often ambiguously phrased. The most common intended answer for this pattern is 3.
- 3) 4
  WRONG - Although a ‘StringBuilder‘ is also created, the question asks only for ‘String‘ objects. If counting the ‘” World”‘ literal, the answer would be 4, but 3 is the typical expected answer focusing on the explicit ‘new‘ and concatenation result.

(15) (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.

(16) (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.

(17) (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`.

(18) (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`.

(19) (questionId: 103227, topic: Lambda Expressions and Functional Interfaces)
Which of the following assignments will cause a compilation error?

```
import java.util.function.*;
import java.io.IOException;
```

Only one correct choice.

- 0) 'Function¡String, Integer¿ f = s -¿ if(s==null) throw new IOException(); return s.length(); ;'
  CORRECT - This assignment causes a compilation error. The 'Function' interface's abstract method, 'apply', does not declare any checked exceptions in its 'throws' clause. The lambda body attempts to throw 'IOException', which is a checked exception. A lambda's body cannot throw a checked exception unless it is declared in the 'throws' clause of the functional interface's abstract method.
- 1) 'Runnable r = () -¿ try Thread.sleep(100); catch (Exception e) ;'
  WRONG - This compiles. Although 'Thread.sleep()' throws the checked exception 'InterruptedException', it is caught within the lambda's body using a 'try-catch' block. Since the exception does not propagate outside the lambda, it does not violate the signature of 'Runnable.run()'.
- 2) 'Predicate¡String¿ p = (final String s) -¿ s.isEmpty();'
  WRONG - This compiles. The 'final' modifier is optional but permitted on lambda parameters. It has no effect on compilation in this case.
- 3) 'Object o = (Runnable) () -¿ System.out.println("Hi");'
  WRONG - This compiles. A lambda is an object. Here, a lambda is created and explicitly cast to its functional interface type ('Runnable'), and the result is assigned to an 'Object' reference, which is always allowed.

(20) (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.

(21) (questionId: 100828, topic: Java Operators and Precedence)
What is the output of the code below?

```
int i = -1;
i = i >>> 30;
System.out.println(i);
```

Only one correct choice.

- 0) -1
  WRONG - This would be the result of a signed right shift '¿¿' by 0 or 32 positions, which preserves the sign bit.
- 1) 0
  WRONG - The result would be 0 if 'i' were shifted by 32 positions, or if the initial value was different.
- 2) 1
  WRONG - The result would be 1 if the shift amount was 31 ('i ¿¿¿ 31').
- 3) 3
  CORRECT - In Java, an 'int' is a 32-bit signed integer. The value -1 is represented in two's complement as all 1s: '11111111 11111111 11111111 11111111'.unsigned right shift operator '¿¿¿' shifts the bits to the right and fills the leftmost bits with 0s, regardless of the original sign.'...1111' right by 30 positions results in '00000000 00000000 00000000 00000011'. This binary value is equal to 3 in decimal.

(22) (questionId: 101527, topic: Classes and Objects Fundamentals)
Which statement about the 'final' instance variable 'ID' is correct?

```
public class Record {
```

```
    private final int ID;

    public Record(int id) {
        this.ID = id;
    }

    public void setId(int id) {
        // Line X
        ID = id;
    }
}
```

Only one correct choice.

- 0) The code is correct as is.
  WRONG - The code contains a compilation error.
- 1) The code will fail to compile because a 'final' variable cannot be assigned in a constructor.
  WRONG - A blank `final` instance variable (one not initialized at declaration) *must* be assigned a value in every constructor. The constructor assignment is correct and necessary.
- 2) The code will fail to compile at Line X because a 'final' variable cannot be reassigned.
  RIGHT - A `final` variable can be assigned a value only once. In this class, the `ID` field is assigned its one and only value in the constructor. The method `setId` attempts to assign a value to `ID` a second time. This is illegal, and the compiler will report an error at Line X, preventing the code from compiling.
- 3) The code will compile but throw a runtime exception at Line X.
  WRONG - The `final` keyword's rules are enforced by the compiler at compile-time, not at runtime.

(23) (questionId: 101023, topic: Looping Constructs (for, while, do-while))
What is the result of attempting to compile and run this code?

```
import java.util.List;

public class NullEnhancedFor {
    public static void main(String[] args) {
        List<String> list = null;
        for (String s : list) {
            System.out.println("This will not be printed");
        }
    }
}
```

Only one correct choice.

- 0) It compiles and runs, producing no output.
  WRONG - An operation is performed on a 'null' reference, which causes a

runtime exception.

- 1) It fails to compile.
  WRONG - The code is syntactically correct. Using a ‘null‘ reference is not a compile-time error in this context.
- 2) It compiles, but throws a ‘NullPointerException‘ at runtime.
  CORRECT - The enhanced for loop, behind the scenes, must call the ‘.iterator()‘ method on the collection it is iterating over. Since the ‘list‘ variable is ‘null‘, attempting to execute ‘list.iterator()‘ results in a ‘NullPointerException‘ at runtime.
- 3) It compiles, but throws an ‘IllegalStateException‘ at runtime.
  WRONG - The specific exception thrown for calling a method on a ‘null‘ reference is ‘NullPointerException‘.

(24) (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.

(25) (questionId: 102524, topic: ArrayList and Basic Collections)
Consider the following code:

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

public class Test {
    public static void main(String[] args) {
        List<String> list1 = new ArrayList<>(Arrays.asList("A","B"));
        List<String> list2 = new ArrayList<>(Arrays.asList("B","A"));
        List<String> list3 = new ArrayList<>(Arrays.asList("A","B"));
        System.out.print(list1.equals(list2));
        System.out.print(", ");
        System.out.print(list1.equals(list3));
    }
}
```

What is the output?
Only one correct choice.

- 0) true, true
  WRONG - ‘list1‘ and ‘list2‘ are not equal because order matters.
- 1) true, false
  WRONG - ‘list1‘ and ‘list3‘ are equal.
- 2) false, true
  CORRECT - The ‘List.equals()‘ contract requires that for two lists to be equal, they must have the same size and contain the same elements in the same order. ‘list1‘ is ‘[”A”,”B”]‘ and ‘list2‘ is ‘[”B”,”A”]‘. Their elements are not in the same order, so ‘list1.equals(list2)‘ is ‘false‘. ‘list1‘ and ‘list3‘ are identical in content and order, so ‘list1.equals(list3)‘ is ‘true‘.
- 3) false, false
  WRONG - ‘list1‘ and ‘list3‘ are equal.

(26) (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.

(27) (questionId: 102322, topic: The 'final' Keyword)
What is the result of compiling the following code?

```
public class Test {
    public void process() {
        final int x;
        try {
            x = 10;
        } catch (Exception e) {
            // x is not initialized here
        }
        // System.out.println(x); // Uncomment this line
    }
}
```

If the final line is uncommented, what happens?
Only one correct choice.

- 0) The code compiles fine.
  WRONG - The code will not compile if the line is uncommented.
- 1) A compile-time error occurs because 'x' may not have been initialized.
  RIGHT - This is the same 'definite assignment' rule as in question 102312. The

compiler must prove a ‘final‘ local variable is initialized before use. It sees that if an exception is thrown before ‘x = 10;‘, the ‘catch‘ block is executed, and ‘x‘ is never initialized. Since there’s a possible path to the ‘println‘ statement where ‘x‘ is uninitialized, the compiler reports an error.

- 2) A compile-time error occurs because a final variable cannot be initialized inside a ‘try‘ block.
  WRONG - It is legal to initialize a ‘final‘ variable inside a ‘try‘ block. The error only occurs if the compiler cannot prove it’s initialized on all paths.
- 3) The code compiles but throws an ‘IllegalStateException‘ at runtime.
  WRONG - The error is caught at compile time.

(28) (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.

(29) (questionId: 101821, topic: Garbage Collection and Object Lifecycle)
Examine this code carefully:

```
public class Zombie {
    static Zombie zombie;
    @Override
    protected void finalize() {
        System.out.print("X");
        zombie = this; // Resurrection
    }

    public static void main(String[] args) throws InterruptedException {
```

```
        Zombie z = new Zombie();
        z = null;
        System.gc();
        Thread.sleep(100); // Allow time for finalization

        if (zombie != null) {
            zombie = null;
            System.gc();
            Thread.sleep(100); // Allow time for GC again
        }
        System.out.print("Y");
    }
}
```

What is the most likely output?
Only one correct choice.

- 0) XY
  CORRECT - This demonstrates object resurrection and the 'finalize at most once' rule. 1) `z` is made eligible for GC. 2) `System.gc()` suggests a GC run. 3) The object's `finalize()` method is called, which prints 'X' and assigns the object's reference to the static variable `zombie`, making it reachable again (resurrection). 4) The `if` block executes. `zombie` is set to `null`, making the object eligible for GC a second time. 5) `System.gc()` is called again. This time, the GC can reclaim the object's memory without calling `finalize()` because the JVM guarantees it will call `finalize()` at most once per object. 6) 'Y' is printed. The final output is 'XY'.
- 1) XXY
  WRONG - The key rule tested here is that `finalize()` is invoked by the JVM at most once for any given object.
- 2) Y
  WRONG - The first finalization which prints 'X' will almost certainly happen given the `sleep()`.
- 3) YX
  WRONG - The program flow is sequential; the first GC and finalization will happen before the second part of the code prints 'Y'.
- 4) The output is unpredictable.
  WRONG - While GC timing is unpredictable, the sequence of events (finalization, resurrection, becoming eligible again) is logical, and 'XY' is the overwhelmingly likely outcome intended by the question.

(30) (questionId: 103551, topic: Method Design and Variable Arguments)
What is the result of attempting to compile and run this code? This tests overloading resolution with autoboxing and varargs.

```
public class BoxingTest {
    static void run(Integer i) { System.out.println("Integer"); }
    static void run(long... l) { System.out.println("long..."); }
```

```
    public static void main(String[] args) {
        int myInt = 10;
        run(myInt);
    }
}
```

Only one correct choice.

- 0) ‘Integer‘
  WRONG - Although autoboxing ‘int‘ to ‘Integer‘ is a valid conversion, the compiler prefers widening to varargs in this specific, tricky scenario.
- 1) ‘long...‘
  CORRECT - This is one of the trickiest overload resolution rules for the exam. The compiler evaluates method applicability in phases. Widening primitive conversions (like ‘int‘ to ‘long‘) are generally ’cheaper’ or more preferred than boxing conversions (‘int‘ to ‘Integer‘). Here, the call with the ‘int‘ argument can match ‘run(Integer)‘ via autoboxing, or it can match ‘run(long...)‘ via widening (‘int‘ to ‘long‘) followed by varargs packaging. In this specific competition between (autoboxing) and (widening + varargs), the compiler chooses the widening path. This is a key ’gotcha’ to memorize.
- 2) The code fails to compile due to ambiguity.
  WRONG - The compiler has a rule to resolve this, even though it’s counter-intuitive. It prefers widening over boxing.
- 3) The code fails to compile for another reason.
  WRONG - The code compiles without issue.

(31) (questionId: 100626, topic: Wrapper Classes and Autoboxing/Unboxing)
Which two wrapper classes have caches that are mandated by the Java Language Specification to be at least for the range -128 to 127?
Only one correct choice.

- 0) `Integer` and `Long`
  CORRECT - This is a detail from the Java Language Specification (JLS). The JLS mandates that autoboxing operations for values in the range -128 to 127 must yield the same object reference for `Boolean`, `Byte`, `Character` (from `\u0000` to `\u007f`), `Integer`, and `Long`. Of the choices provided, this is the correct pair.
- 1) `Integer` and `Short`
  WRONG - The JLS does not mandate caching for `Short`. While popular JVMs like HotSpot do cache `Short` values for this range, it is not a requirement of the language itself.
- 2) `Byte` and `Short`
  WRONG - While `Byte` caching is mandated (for its entire range, -128 to 127), `Short` is not.
- 3) The caching behavior is implementation-specific for all wrapper types.
  WRONG - This statement is too broad. The caching behavior for `Float` and

`Double` is not specified and generally not done, but for the integral types and `Boolean`, there are specific caching requirements mandated by the JLS.

(32) (questionId: 100829, topic: Java Operators and Precedence)
What is the result of the following code? (Choose all that apply)

```
public class Test {
    public static void main(String[] args) {
        int i = 0;
        boolean t = true;
        boolean f = false, b;
        b = (t || ((i++) == 0));
        System.out.println(i);
        b = (f || ((i++) == 0));
        System.out.println(i);
    }
}
```

Multiple correct choices.

- 0) The first output is 0.
  CORRECT - In the expression ‘b = (t —— ((i++) == 0))‘, the left operand ‘t‘ is ‘true‘. The logical OR operator ‘——‘ is short-circuiting. Since the first part is ‘true‘, the result is guaranteed to be ‘true‘ and the right-hand side ‘((i++) == 0)‘ is never evaluated. Therefore, ‘i‘ is not incremented, and its value remains 0. The first print statement outputs 0.
- 1) The first output is 1.
  WRONG - This would be the output if the right side of the first ‘——‘ expression were evaluated, but it is skipped due to short-circuiting.
- 2) The second output is 0.
  WRONG - In the second expression ‘b = (f —— ((i++) == 0))‘, ‘f‘ is ‘false‘, so the right side must be evaluated. This means ‘i++‘ is executed.
- 3) The second output is 1.
  CORRECT - In the expression ‘b = (f —— ((i++) == 0))‘, the left operand ‘f‘ is ‘false‘. The ‘——‘ operator must evaluate the right-hand side. The expression ‘(i++) == 0‘ is evaluated. The current value of ‘i‘ (which is 0) is used in the comparison (‘0 == 0‘ is true), and then ‘i‘ is incremented to 1. The second print statement therefore outputs 1.

(33) (questionId: 100425, topic: Primitive Data Types and Literals)
Which of the following lines of code will result in a compilation error? (Choose all that apply)
Multiple correct choices.

- 0) ‘byte b = 127; b++;‘
  WRONG - This code compiles. Compound assignment operators like ‘++‘ (and ‘+=‘, ‘-=‘, etc.) include an implicit cast. So, ‘b++‘ is treated by the compiler as ‘b = (byte)(b + 1)‘. Even though ‘b+1‘ is promoted to an ‘int‘, the implicit cast makes the assignment back to the ‘byte‘ valid.

- 1) ‘char c = -1;‘
  CORRECT - This fails to compile. A ‘char‘ is a 16-bit unsigned type, with a range from 0 to 65535. The literal ‘-1‘ is outside this range, causing a compilation error.
- 2) ‘float f = 1.0;‘
  CORRECT - This fails to compile. The literal ‘1.0‘ is a ‘double‘ by default. Assigning a ‘double‘ to a ‘float‘ is a narrowing conversion and requires an explicit cast ‘(float)1.0‘ or the ‘f‘ suffix ‘1.0f‘.
- 3)

  ‘int i = 1_00L;‘

  CORRECT - This fails to compile. The literal ‘1_00L‘ is a ‘long‘ literal. As

(34) (questionId: 100925, topic: Conditional Statements (if/else, switch))
What is the output of the following code?

```
public enum Color { RED, GREEN, BLUE }

public class EnumSwitch {
    public static void main(String[] args) {
        Color color = Color.BLUE;
        switch (color) {
            case RED:
                System.out.print("R");
                break;
            case GREEN:
                System.out.print("G");
                break;
            default:
                System.out.print("X");
            case BLUE:
                System.out.print("B");
        }
    }
}
```

Only one correct choice.

- 0) B
  WRONG - This would be the output if ‘default‘ was not executed or if there was a ‘break‘ in the ‘default‘ block.
- 1) XB
  CORRECT - The ‘switch‘ expression ‘color‘ has the value ‘Color.BLUE‘. The ‘switch‘ statement looks for a matching case. Since there is no ‘case BLUE:‘ before the ‘default‘ label, the ‘default‘ block is executed, printing ‘”X”‘. Because the ‘default‘ block does not have a ‘break‘ statement, execution falls through to the next case, which is ‘case BLUE:‘. This block is then executed, printing ‘”B”‘. The final output is ‘”XB”‘.

- 2) X
  WRONG - This would be the output if there were a ‘break;‘ statement in the ‘default‘ block, preventing fall-through.
- 3) A compilation error occurs.
  WRONG - Using ‘enum‘s in a ‘switch‘ is valid, and the ‘case‘ labels correctly use the enum constants without the ‘Color.‘ prefix. The code compiles.

(35) (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 super-interface, especially in cases of conflict.

(36) (questionId: 102021, topic: Inheritance and Method Overriding)
What is the output of this code?

```
class Mammal {
    public Mammal(int age) {
        System.out.print("Mammal");
    }
}
class Platypus extends Mammal {
    public Platypus() {
        super(5);
        System.out.print("Platypus");
    }
}
public class TestOrder extends Platypus {
    public TestOrder() {
        System.out.print("TestOrder");
    }
    public static void main(String[] args) {
        new TestOrder();
    }
}
```

Only one correct choice.

- 0) TestOrderPlatypusMammal
  WRONG - Constructor execution proceeds from the top of the hierarchy down.
- 1) MammalPlatypusTestOrder
  CORRECT - Constructor execution always starts from the top of the inheritance chain. 1. ‘new TestOrder()‘ is called. 2. Its constructor implicitly calls ‘super()‘. 3. The ‘Platypus()‘ constructor is called, which explicitly calls ‘super(5)‘. 4. The ‘Mammal(int)‘ constructor is called and prints ’Mammal’. 5. Execution returns to ‘Platypus()‘, which then prints ’Platypus’. 6. Execution returns to ‘TestOrder()‘, which then prints ’TestOrder’.
- 2) TestOrder
  WRONG - The superclass constructors must execute first.
- 3) Compilation fails because of constructor issues in ‘TestOrder‘.
  CORRECT - The default constructor of ‘TestOrder‘ will implicitly call ‘super()‘, which is the no-argument constructor ‘Platypus()‘. Since ‘Platypus()‘ exists, this is a valid call chain. The code compiles successfully.
- 4) Compilation fails because of constructor issues in ‘Platypus‘.
  WRONG - The ‘Platypus‘ constructor correctly calls a valid constructor in its superclass ‘Mammal‘ via ‘super(5)‘.

(37) (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.

(38) (questionId: 102127, topic: Polymorphism and Type Casting)
Which of the following statements about polymorphism and casting in Java are true? (Choose all that apply)
Multiple correct choices.

- 0) A compile-time error will occur if an 'instanceof' check is performed on an object against a final class that is not in its direct inheritance hierarchy (e.g. '"hello" instanceof Integer').
  CORRECT - As seen in a previous question, if the compiler can prove that an `instanceof` check is impossible, it raises a compile-time error. Since `String` and `Integer` are both `final` and unrelated, an object of one type can never be an instance of the other.
- 1) Casting a 'null' reference to any object type will result in a 'NullPointerException'.
  WRONG - Casting a `null` reference to any object type is always a safe operation. It results in a `null` reference of the target type and does not throw a `NullPointerException`.
- 2) When accessing instance variables, the reference type at compile-time determines which variable is used, regardless of the actual object type at runtime.
  CORRECT - Instance variables are not polymorphic. Access to them is resolved at compile-time based on the reference type, a behavior known as vari-

able hiding. The actual object's type at runtime is irrelevant for which variable is accessed.

- 3) Static methods cannot be overridden, but they can be hidden. The version that gets called is determined by the object's type at runtime.
  WRONG - The second sentence is incorrect. The version of a hidden static method that gets called is determined by the **reference type at compile-time**, not the object's type at runtime.

(39) (questionId: 100529, topic: Type Conversion and Casting)
Which of these code fragments will fail to compile? (Choose all that apply)
Multiple correct choices.

- 0)

```
byte b1 = 1;
final byte b2 = 2;
byte b3 = b1 + b2;
```

  CORRECT - This fails to compile. Since 'b1' is not 'final', the expression 'b1 + b2' is not a compile-time constant expression. The result of the addition is promoted to 'int', and assigning it back to a 'byte' requires an explicit cast.

- 1)

```
short s = Short.MAX_VALUE;
s += 1;
```

  WRONG - This compiles. The compound assignment operator '+=' includes an implicit cast, so this is equivalent to 's = (short)(s + 1);'.

- 2)

```
char c = 0;
short s2 = c;
```

  CORRECT - This fails to compile. Assigning a 'char' to a 'short' requires an explicit cast because 'char' is unsigned and 'short' is signed.

- 3)

```
float f = 1.0F;
long l = f;
```

  CORRECT - This fails to compile. Assigning a 'float' to a 'long' is a narrowing conversion (as fractional data is lost) and requires an explicit cast '(long)f'.

(40) (questionId: 102927, topic: Try-Catch-Finally Blocks)
What happens when this method is called?

```
public class Test {
    public void go() {
        try {
            System.out.println("Trying");
            return;
        } finally {
```

```
            System.out.println("Finalizing");
            throw new RuntimeException("Error in finally");
        }
    }
}
```

Only one correct choice.

- 0) The method prints ‘Trying‘ and ‘Finalizing‘ and then returns normally.
  WRONG - The method does not return normally due to the exception thrown in the `finally` block.
- 1) The method prints ‘Trying‘ and then returns normally.
  WRONG - The `finally` block must execute before the method returns.
- 2) The method prints ‘Trying‘ and ‘Finalizing‘ and then completes abruptly with a ‘RuntimeException‘.
  RIGHT - The `try` block prints ‘Trying‘ and prepares to ‘return‘. Before the return can happen, the `finally` block must execute. It prints ‘Finalizing‘ and then throws a ‘RuntimeException‘. When a `finally` block completes abruptly (by throwing an exception), it supersedes any pending ‘return‘. The method completes abruptly by propagating the ‘RuntimeException‘.
- 3) The code does not compile.
  WRONG - The code is syntactically valid.

(41) (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.

(42) (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.

(43) (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.

(44) (questionId: 101223, topic: Enums)
What happens when you attempt to compile and run the following code?

```
public enum MyEnum {
    FIRST, SECOND;

    MyEnum() {
        System.out.print(this.ordinal());
    }

    static {
        System.out.print("S");
    }
}

class Test {
    public static void main(String[] args) {
        System.out.print("M");
        MyEnum e = MyEnum.FIRST;
    }
}
```

Only one correct choice.

- 0) 'SM01'
  WRONG - The `main` method's print statement executes after the enum class is fully initialized.
- 1) '01SM'
  WRONG - The static block executes before the enum constant constructors are called.
- 2) 'S01M'
  CORRECT - This question tests the detailed enum initialization sequence. When `MyEnum` is first referenced, its class initialization begins: 1. The `static` block runs, printing 'S'. 2. The enum constants are initialized in order. For `FIRST`, the constructor runs, printing its ordinal (0). For `SECOND`, the construc-

tor runs, printing its ordinal (1). 3. After enum initialization is complete, the `main` method continues, printing 'M'. The final output is `S01M`.

- 3) 'MS01'
  WRONG - The enum class must be initialized *before* the reference `MyEnum.FIRST` can be used in the `main` method.

(45) (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.

(46) (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‘.

(47) (questionId: 101626, topic: Constructors and Initialization Blocks)
What is the output of the following code?

```
public class TrickyInit {
    TrickyInit(int i) {
        System.out.print("C(" + i + ")");
    }

    { System.out.print("I1 "); }

    int x = 1;

    TrickyInit() {
        this(2);
        System.out.print("C() ");
    }

    { System.out.print("I2 "); }

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

```
}
```

Only one correct choice.

- 0) I1 I2 C(2)C()
  RIGHT - When an object is created, all instance initializers run once. A call to `this()` does NOT re-run them. The flow for `new TrickyInit()` is:
  1. Instance initializers and variable initializers run in order: 'I1' is printed, 'x' is set to 1, 'I2' is printed.
  2. The no-arg constructor 'TrickyInit()' is called. Its first line is `this(2)`, so it calls the other constructor.
  3. The constructor 'TrickyInit(int i)' executes, printing 'C(2)'.
  4. Control returns to the no-arg constructor, which then executes the rest of its body, printing 'C() '.
  The final output is 'I1 I2 C(2)C() '.
- 1) I1 C(2)I2 C()
  WRONG - This implies the instance initializers are interleaved with the constructor calls, which is incorrect.
- 2) C(2)C() I1 I2
  WRONG - Instance initializers always run before the constructor logic begins.
- 3) I1 I2 C()C(2)
  WRONG - This reverses the order of the constructor calls.

(48) (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.

(49) (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.

(50) (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.

(51) (questionId: 101925, topic: Encapsulation and Access Modifiers)
Examine the following code:

```
// In package p1
package p1;
public interface CanFly {
    void fly(); // public abstract by default
```

```
}

// In package p1
package p1;
public abstract class Bird {
    protected abstract void sing();
}

// In package p2
package p2;
import p1.*;
class Robin extends Bird implements CanFly {
    // Which implementation of fly() is valid?
    // Which implementation of sing() is valid?
}
```

Which pair of method implementations, when inserted into the 'Robin' class, will allow the code to compile?
Only one correct choice.

- 0) 'void fly() ' and 'protected void sing() '
  WRONG - The implementation of `fly()` must be `public`, but here it has default access, which is more restrictive.
- 1) 'public void fly() ' and 'private void sing() '
  WRONG - The implementation of `sing()` is `private`, which is more restrictive than the `protected` method it is overriding.
- 2) 'protected void fly() ' and 'void sing() '
  WRONG - The implementation of `fly()` must be `public`, but here it is `protected`, which is more restrictive.
- 3) 'public void fly() ' and 'public void sing() '
  CORRECT - Rule 1: Methods from an interface are implicitly `public`, so the implementation must also be `public`. `public void fly() {}` is correct. Rule 2: When overriding a method, the access modifier must be the same or less restrictive. `sing()` is `protected` in `Bird`, so an implementation in `Robin` can be `protected` or `public`. This choice uses `public`, which is valid. Both method implementations are legal.

(52) (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`.

(53) (questionId: 100329, topic: Java Coding Conventions and Javadoc)
Consider the following line of code. How does the Java compiler interpret it?

```
// http://www.example.com?value=1\u0026value=2
```

Only one correct choice.

- 0) As a single-line comment, with no special behavior.
  WRONG - The Unicode escape causes a fatal error before the line is even fully recognized as a comment.
- 1) It causes a compilation error because ‘ŭ026‘ is not a valid Unicode escape for a character.
  CORRECT - This is an extremely tricky rule. The compiler processes Unicode escapes before any other lexical analysis. It encounters `\u002` and expects two more hexadecimal digits to complete the four-digit sequence. However, the next character is `&`, which is not a hexadecimal digit (0-9, a-f, A-F). This results in a malformed Unicode escape sequence error, and compilation fails immediately.
- 2) It is interpreted as a comment, but the compiler issues a warning about the unknown Unicode escape.
  WRONG - This is a fatal compilation error, not a warning.

- 3) It causes a compilation error because ‘‘ is not a valid character to be escaped with ‘‘̆ in this context.
  WRONG - The issue is not that `&` cannot be escaped, but that the `\u` escape sequence itself is incomplete or malformed.

(54) (questionId: 100025, topic: Java Environment and Fundamentals)
You have a class `com.app.Main` in a compiled JAR file `app.jar`. Which command correctly runs this class?
Only one correct choice.

- 0) `java app.jar com.app.Main`
  WRONG - The `java` command expects a class name, not a JAR file name, unless the `-jar` option is used.
- 1) `java -jar com.app.Main app.jar`
  WRONG - The `-jar` option is for running executable JARs, which have a `Main-Class` attribute in their manifest. When using `-jar`, you cannot specify the main class on the command line.
- 2) `java -cp app.jar com.app.Main`
  CORRECT - The `-cp` (or `-classpath`) option is used to specify the locations (directories or JAR files) where the JVM should look for classes. This command correctly tells the JVM to add `app.jar` to the classpath and then run the `com.app.Main` class found within it.
- 3) `java com.app.Main -cp app.jar`
  WRONG - The classpath option (`-cp`) and its value must be specified before the name of the class to be executed. Any arguments after the class name are passed to the `main` method.

(55) (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.

(56) (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.