1Z0-808 Mock Exam Solutions

ExamId: 100

August 5, 2025

(1) (questionId: 103207, topic: Lambda Expressions and Functional Interfaces)
What is the result of executing the following code?

```
import java.util.function.Predicate;

public class CheckString {
    public static void main(String[] args) {
        Predicate<String> p = (s) -> s.isEmpty();
        System.out.println(p.test(""));
    }
}
```

Only one correct choice.

- 0) 'true'
  CORRECT - The code compiles and runs successfully. `Predicate<String>` has the method `boolean test(String s)`. The lambda implements this by calling the `isEmpty()` method on the input string. When `p.test("")` is called, `"".isEmpty()` is executed, which returns `true`.
- 1) 'false'
  WRONG - The `String.isEmpty()` method returns `true` for a string of length 0, such as `""`.
- 2) A 'NullPointerException' is thrown.
  WRONG - A `NullPointerException` would be thrown if the method were called with a `null` argument (i.e., `p.test(null)`), because the lambda would attempt to call `null.isEmpty()`. However, it is called with an empty string object `""`, which is not `null`.
- 3) The code does not compile.
  WRONG - The code is valid. It correctly imports `java.util.function.Predicate`, uses a valid lambda expression, and the method call is appropriate.

(2) (questionId: 100507, topic: Type Conversion and Casting)
What is the value of 'i' after this code is executed?

```
double d = 12.9;
int i = (int)d;
```

Only one correct choice.

- 0) 12
  RIGHT - When a floating-point value (like a 'double') is cast to an integer type (like 'int'), the fractional part is truncated (i.e., simply cut off), not rounded. Therefore, '(int)12.9' evaluates to '12'.
- 1) 13
  WRONG - The value is truncated, not rounded up.
- 2) 12.9
  WRONG - An 'int' cannot hold a fractional value.

- 3) The code does not compile.
  WRONG - The explicit cast '(int)' makes the code compile successfully.

(3) (questionId: 101820, topic: Garbage Collection and Object Lifecycle)
What happens if an exception is thrown from within a 'finalize()' method?
Only one correct choice.

- 0) The exception propagates to the 'main' thread and terminates the application if not caught.
  WRONG - The exception is handled by the GC's finalizer thread and is not propagated to other application threads.
- 1) The garbage collector catches the exception, ignores it, and halts finalization for that object.
  CORRECT - The finalization process is run by a special JVM thread. If an exception is thrown from a `finalize()` method, that thread catches the exception, effectively ignoring it (it's not propagated), and halts the finalization for that specific object. The object is still considered 'finalized' and will be reclaimed by the GC later.
- 2) The object is not garbage collected.
  WRONG - The object has already been marked for collection and its finalizer has been run (or attempted to run). It will be garbage collected in a subsequent cycle.
- 3) The JVM will shut down immediately.
  WRONG - An exception in `finalize()` does not cause the JVM to shut down.
- 4) It causes a compilation error.
  WRONG - The `finalize()` method is declared with `throws Throwable`, so throwing an exception from it is syntactically valid.

(4) (questionId: 102816, topic: Exception Hierarchy and Types)
Which exception will be thrown by the following code?

```
public class Test {
    public static void main(String[] args) {
        String[] array = {"a", "b"};
        System.out.println(array[getIdx()].length());
    }
    public static int getIdx() {
        return 2;
    }
}
```

Only one correct choice.

- 0) 'NullPointerException'
  WRONG - A `NullPointerException` would occur if `array[getIdx()]` evaluated to `null`, but the expression doesn't get that far.
- 1) 'ArrayIndexOutOfBoundsException'
  RIGHT - In the expression `array[getIdx()].length()`, Java must first eval-

uate the array index. The method `getIdx()` is called and returns `2`. Then, the array access `array[2]` is attempted. Since the array's valid indices are 0 and 1, this immediately throws an `ArrayIndexOutOfBoundsException`. The `.length()` method is never called.

- 2) 'StringIndexOutOfBoundsException'
  WRONG - This would be thrown if you tried to access a character at an invalid index within a `String`, which doesn't happen here.
- 3) No exception is thrown.
  WRONG - An exception is definitely thrown.

(5) (questionId: 100619, topic: Wrapper Classes and Autoboxing/Unboxing)
Which of these lines of code will compile successfully? (Choose all that apply)
Multiple correct choices.

- 0) `Float f = 10.0;`
  WRONG (Will not compile) - The literal `10.0` is a `double` by default. Java does not automatically narrow a `double` to a `float`, nor does it autobox a `double` to a `Float`. To fix this, you would need to use a float literal: `Float f = 10.0f;`.
- 1) `Character c = 65;`
  WRONG (Will not compile) - You cannot autobox a primitive `int` literal like `65` into a `Character` reference. A `Character` must be initialized with a `char` literal (e.g., `'A'`) or by an explicit cast (e.g., `(char)65`).
- 2) `double d = new Double(10.5);`
  CORRECT (Will compile) - This is a valid **unboxing** operation. A new `Double` object is created with the value 10.5, and then it is automatically unboxed to a primitive `double` to be assigned to the variable `d`.
- 3) `Boolean b = null;`
  CORRECT (Will compile) - Wrapper classes are object types. Any object reference variable can be assigned the value `null`, indicating that it does not currently refer to any object.
- 4) `int i = new Integer(5);`
  CORRECT (Will compile) - This is a valid **unboxing** operation. A new `Integer` object is created with the value 5. This object is then automatically unboxed to a primitive `int` to be assigned to the variable `i`.

(6) (questionId: 101310, topic: String Immutability and Operations)
What is printed by the following code?

```
String s = " a b c ";
s = s.trim();
s = s.replace(" ", "");
System.out.println(s.length());
```

Only one correct choice.

- 0) 3
  CORRECT - Let's trace the value of 's': 1. Initially, `s` is `" a b c "`. 2.

`s.trim()` removes leading and trailing whitespace, returning `"a b c"`. This is assigned back to `s`. 3. `s.replace(" ", "")` removes all remaining spaces, returning `"abc"`. This is assigned back to `s`. 4. The length of the final string `"abc"` is 3.

- 1) 5
  WRONG - This would be the length of `"a b c"` after trimming but before replacing.
- 2) 7
  WRONG - This would be the length of the original string.
- 3) 9
  WRONG - This count is incorrect.

(7) (questionId: 102516, topic: ArrayList and Basic Collections)
Given the following list, which of the options will result in the list ‘[X, Z]‘? (Choose all that apply)

```
List<String> list = new ArrayList<>();
list.add("X");
list.add("Y");
list.add("Z");
```

Multiple correct choices.

- 0) `list.remove(1);`
  CORRECT - The list is ‘[”X”, ”Y”, ”Z”]‘. ‘remove(1)‘ removes the element at index 1, which is ”Y”. The result is ‘[”X”, ”Z”]‘.
- 1) `list.remove("Y");`
  CORRECT - ‘remove(”Y”)‘ finds the first object that is ‘equal‘ to ”Y” and removes it. The result is ‘[”X”, ”Z”]‘.
- 2) `list.remove(new String("Y"));`
  CORRECT - This is functionally identical to option 1. A new ‘String‘ object is created, but ‘remove(Object o)‘ uses the ‘.equals()‘ method for comparison, so it finds and removes the ”Y” in the list. The result is ‘[”X”, ”Z”]‘.
- 3) `list.set(1, "Z"); list.remove(2);`
  WRONG - While this sequence of operations does result in the list ‘[”X”, ”Z”]‘, it’s not a direct removal of ”Y”. It first changes the list to ‘[”X”, ”Z”, ”Z”]‘ and then removes the last element. Exam questions about achieving a state usually favor the most direct methods.

(8) (questionId: 102208, topic: Abstract Classes and Interfaces)
What is the result of attempting to compile this code?

```
public abstract class Shape {
    private abstract void draw();
}
```

Only one correct choice.

- 0) It compiles successfully.
  WRONG - The combination of ‘private‘ and ‘abstract‘ modifiers is illegal.
- 1) It fails to compile because an abstract method cannot be ‘private‘.
  RIGHT - The purpose of an ‘abstract‘ method is to be implemented by a subclass. The ‘private‘ modifier makes a method inaccessible to subclasses. These two modifiers are mutually exclusive, as a subclass cannot implement a method it cannot see or access. This results in a compile-time error.
- 2) It fails to compile because the class is abstract but has no concrete methods.
  WRONG - An abstract class is not required to have any concrete methods. It can have only abstract methods.
- 3) It fails to compile because ‘draw()‘ has no method body.
  WRONG - An abstract method is *defined* as a method without a body. The lack of a body is the correct syntax for an abstract method.

(9) (questionId: 100416, topic: Primitive Data Types and Literals)
Which of the following character literals are valid in Java? (Choose all that apply)
Multiple correct choices.

- 0) ‘’ŏ0041’‘
  CORRECT - ‘’ŏ0041’‘ is a valid ‘char‘ literal using a Unicode escape. It represents the character ’A’.
- 1) ‘”‘
  CORRECT - ‘”‘ is a valid ‘char‘ literal using an escape sequence for a special character (newline).
- 2) ‘’ab’‘
  WRONG - A ‘char‘ literal must be enclosed in single quotes and contain exactly one character or one valid escape sequence. ‘’ab’‘ contains two characters and will cause a compilation error.
- 3) ‘””‘
  CORRECT - A double quote character is a valid single character. It does not need to be escaped within a ‘char‘ literal’s single quotes. Therefore, ‘””‘ is a valid literal.

(10) (questionId: 103434, topic: Static Imports)
Consider the following code. Which of the import statements, if inserted at line 1, will allow the code to compile? (Choose all that apply)

```
// line 1: INSERT IMPORT HERE

public class Main {
    public static void main(String[] args) {
        List<String> data = asList("x", "y");
        out.println(data);
    }
}
```

Multiple correct choices.

- 0) ‘import java.util.*; import static java.lang.System.*;‘
  WRONG - This set of imports is missing a way to resolve ‘asList‘. The ‘java.util.*‘ import provides the ‘List‘ type, and ‘static java.lang.System.*‘ provides ‘out‘, but ‘asList‘ (a static method in ‘java.util.Arrays‘) is not imported.
- 1) ‘import java.util.List; import static java.util.Arrays.asList; import static java.lang.System.out;‘
  CORRECT - This set of imports correctly provides all three required elements. ‘import java.util.List;‘ for the ‘List‘ type. ‘import static java.util.Arrays.asList;‘ for the ‘asList‘ method. ‘import static java.lang.System.out;‘ for the ‘out‘ field.
- 2) ‘import static java.util.Arrays.*; import static java.lang.System.out; import java.util.List;‘
  CORRECT - This set also works. The wildcard ‘import static java.util.Arrays.*;‘ imports all static members of ‘Arrays‘, including ‘asList‘. The other two imports fulfill the remaining requirements.
- 3) ‘import static java.util.Arrays.asList; import static java.lang.System.*;‘
  WRONG - This set of imports is missing a regular import for the ‘List‘ type. The compiler will fail with a ’cannot find symbol’ error for ‘List‘.

(11) (questionId: 102617, topic: Generics)
Which of the following statements are true about raw types in Java? (Choose all that apply)
Multiple correct choices.

- 0) Using raw types is completely forbidden in Java 8.
  WRONG - Raw types are permitted in Java 8 to ensure backward compatibility with code written before generics were introduced in Java 5.
- 1) Using raw types bypasses compile-time generic type checking.
  CORRECT - This is the main characteristic and danger of raw types. They opt out of the compile-time checks that generics provide, potentially leading to runtime exceptions.
- 2) The compiler issues a warning when raw types are used.
  CORRECT - The compiler issues ’unchecked’ warnings whenever raw types are used, to alert the developer that type safety is being compromised.
- 3) Raw types are necessary for backward compatibility with pre-Java 5 code.
  CORRECT - This is the sole reason for their existence: to allow new, generic code to interoperate with legacy, pre-Java 5 code.
- 4) A ‘List‘ is equivalent to a ‘List¡Object¿‘.
  WRONG - A raw ‘List‘ and a ‘List¡Object¿‘ are not equivalent. The former circumvents type checking entirely for that reference, while the latter is a fully type-checked generic that can happen to hold any object.

(12) (questionId: 102907, topic: Try-Catch-Finally Blocks)
What is the output of this code?

```
public class Test {
```

```
    public static void main(String[] args) {
        System.out.print(getValue());
    }
    public static int getValue() {
        try {
            return 10;
        } finally {
            System.out.print("Finally ");
        }
    }
}
```

Only one correct choice.

- 0) '10 Finally'
  WRONG - The print statement from the `finally` block executes before the value is returned to the caller's `print` method.
- 1) 'Finally 10'
  RIGHT - When a `return` statement is encountered in a `try` block, the return value (10) is prepared. Before the method actually returns control to the caller, the `finally` block is executed. This prints "Finally ". After the `finally` block completes, the method returns the prepared value (10), which is then printed by the `main` method.
- 2) '10'
  WRONG - The `finally` block always executes if the `try` block is entered.
- 3) 'Finally'
  WRONG - The method successfully returns a value after the `finally` block completes.

(13) (questionId: 102115, topic: Polymorphism and Type Casting)
What is the result of executing the following code?

```
interface Readable {}
class Book implements Readable {}
class EBook extends Book {
    public void read() { System.out.println("Reading EBook"); }
}
public class Test {
    public static void main(String[] args) {
        Readable r = new EBook();
        if (r instanceof EBook) {
            ((EBook) r).read();
        }
    }
}
```

Only one correct choice.

- 0) Reading EBook

RIGHT - The code compiles and runs correctly. First, an `EBook` is assigned to a `Readable` reference, which is valid since `EBook` extends `Book`, which implements `Readable`. The `instanceof EBook` check correctly returns `true`. The code then safely downcasts the reference to `EBook` and calls the `read()` method.

- 1) The code fails to compile because 'r' does not have a 'read()' method.
  WRONG - This error would occur if you tried to call `r.read()` directly without casting, because the `Readable` interface does not define a `read()` method. However, the code correctly casts `r` to `((EBook) r)` before making the call, which is valid.
- 2) The code fails to compile because an interface reference cannot be cast to a class.
  WRONG - An interface reference can be cast to a class type. The cast will succeed at runtime if the object being referenced is an instance of that class.
- 3) A 'ClassCastException' is thrown at runtime.
  WRONG - The `instanceof` check ensures that the cast is safe, thus preventing a `ClassCastException`.

(14) (questionId: 101506, topic: Classes and Objects Fundamentals)
What is the result of compiling and running the following code?

```
public class Test {
    public static void main(String[] args) {
        Test t;
        t.go();
    }

    void go() {
        System.out.println("Going!");
    }
}
```

Only one correct choice.

- 0) Going!
  WRONG - The code will not run because it will not compile.
- 1) The code compiles but throws a 'NullPointerException' at runtime.
  WRONG - A `NullPointerException` is a runtime issue. This code has a compile-time issue, which is caught before the program can run.
- 2) The code fails to compile because 't' is not initialized.
  RIGHT - The variable `t` is a local variable within the `main` method. Unlike instance variables, local variables are not given default values and must be explicitly initialized before they are used. The compiler detects that `t` is used in `t.go()` without ever being assigned a value, resulting in a compilation error: 'variable t might not have been initialized'.
- 3) The code compiles but throws an 'IllegalStateException' at runtime.
  WRONG - The code will not compile, so no runtime exceptions can be thrown.

(15) (questionId: 101109, topic: Break, Continue, and Labels)
What is the result of executing this code snippet?

```
int i = 0;
while (i < 10) {
    if (i == 5) {
        continue;
    }
    System.out.print(i);
    i++;
}
```

Only one correct choice.

- 0) 012346789
  WRONG - The loop does not terminate normally.
- 1) 01234
  WRONG - The loop gets stuck when ‘i‘ reaches 5.
- 2) An infinite loop occurs.
  CORRECT - The loop prints values ‘01234‘. When ‘i‘ is 5, the ‘if (i == 5)‘ condition is true, and ‘continue‘ is executed. The ‘continue‘ statement immediately jumps to the start of the next loop iteration, skipping the ‘i++;‘ statement. Since ‘i‘ is never incremented beyond 5, the condition ‘i ¡ 10‘ remains true, and the ‘continue‘ is executed in every subsequent iteration, creating an infinite loop.
- 3) A compilation error occurs.
  WRONG - The code is syntactically correct; the error is a logical one that leads to an infinite loop at runtime.

(16) (questionId: 101412, topic: StringBuilder and StringBuffer)
What is the result of executing this code snippet?

```
StringBuilder sb = new StringBuilder("Test");
String s = "Test";
System.out.println(s.equals(sb.toString()) + " " + sb.toString().equals(s));
```

Only one correct choice.

- 0) ‘true true‘
  CORRECT - The ‘sb.toString()‘ method creates a new ‘String‘ object with the value ‘"Test"‘. The first expression, ‘s.equals(sb.toString())‘, compares two ‘String‘ objects with the same content, so it evaluates to ‘true‘. The second expression, ‘sb.toString().equals(s)‘, does the exact same comparison and also evaluates to ‘true‘. The output is ‘true true‘.
- 1) ‘true false‘
  WRONG - Both comparisons are identical and yield ‘true‘.
- 2) ‘false true‘
  WRONG - Both comparisons are identical and yield ‘true‘.

- 3) ‘false false‘
  WRONG - Both comparisons yield ‘true‘.

(17) (questionId: 102719, topic: Sorting and Searching Collections (Comparable, Comparator))
Which of these expressions will cause a compilation error?

```
class Animal { int age; public int getAge() { return age; } }
class Dog extends Animal {}
List<Dog> dogs = new ArrayList<>();
```

Only one correct choice.

- 0) ‘Comparator¡Animal¿ c1 = (a1, a2) -¿ a1.getAge() - a2.getAge(); dogs.sort(c1);‘
  WRONG - The `sort` method on a `List<Dog>` takes a `Comparator<? super Dog>`. Since `Animal` is a superclass of `Dog`, a `Comparator<Animal>` is valid. This compiles.
- 1) ‘Comparator¡Dog¿ c2 = (d1, d2) -¿ d1.getAge() - d2.getAge(); dogs.sort(c2);‘
  WRONG - A `Comparator<Dog>` is obviously a valid comparator for a `List<Dog>`. This compiles.
- 2) ‘Comparator¡Object¿ c3 = (o1, o2) -¿ 1; dogs.sort(c3);‘
  WRONG - Since `Object` is a superclass of `Dog`, a `Comparator<Object>` is a valid comparator for a `List<Dog>`. This compiles.
- 3) ‘Comparator¡String¿ c4 = (s1, s2) -¿ s1.length() - s2.length(); dogs.sort(c4);‘
  RIGHT - A `Comparator<String>` cannot be used to compare `Dog` objects. `String` is not a supertype of `Dog`. The compiler will report that the method `sort(Comparator<String>)` is not applicable for the type `List<Dog>`, causing a compilation error.

(18) (questionId: 101710, topic: Static Members and ’this’ Keyword)
What is the output of the following code?

```
import static java.lang.Integer.MAX_VALUE;

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

Only one correct choice.

- 0)

  `MAX_VALUE`

  WRONG - `MAX_VALUE` is a variable name, not a string literal. The program prints its value.
- 1) 2147483647
  RIGHT - The statement `import static java.lang.Integer.MAX_VALUE;` allows the static final field `MAX_VALUE` to be used without the `Integer.` prefix.

The value of this constant is 2147483647, which is the maximum value for a 32-bit signed integer. The program will print this number.

- 2) The code does not compile because of the import statement.
  WRONG - The static import syntax is correct.
- 3)

  `The code does not compile because ‘MAX_VALUE‘ is ambiguous.`

  WRONG - There are no other variables named `MAX_VALUE` in scope, so the reference is not ambiguous.

(19) (questionId: 103633, topic: Passing Data Among Methods)
What are the final values of ‘x‘, ‘y.value‘, and ‘z‘ at the end of the ‘main‘ method? (Choose all that apply)

```
class Wrapper { public int value; }

public class FinalValues {
    public static void main(String[] args) {
        int x = 10;
        Wrapper y = new Wrapper(); y.value = 20;
        String z = "30";
        modify(x, y, z);
        // What are the values here?
    }
    public static void modify(int x, Wrapper y, String z) {
        x = 15;
        y.value = 25;
        z = "35";
    }
}
```

Multiple correct choices.

- 0) ‘x‘ is 10
  CORRECT - ‘x‘ is a primitive. Its value is copied to the method. The original ‘x‘ in ‘main‘ is not changed and remains 10.
- 1) ‘y.value‘ is 20
  WRONG - The state of the ‘Wrapper‘ object ‘y‘ is changed by the method.
- 2) ‘y.value‘ is 25
  CORRECT - The method receives a copy of the reference to the ‘Wrapper‘ object. It uses this reference to change the object’s ‘value‘ field to 25. This change is visible in ‘main‘.
- 3) ‘z‘ is ”30”
  CORRECT - ‘z‘ is a reference to an immutable ‘String‘. The method reassigns its local copy of the reference to a new ‘String‘ ‘”35”‘. The original ‘z‘ in ‘main‘ is unaffected and still refers to ‘”30”‘.
- 4) ‘z‘ is ”35”

WRONG - The original ‘String‘ reference ‘z‘ is not affected by the reassignment inside the method.

(20) (questionId: 100014, topic: Java Environment and Fundamentals)
A Java source file contains two classes, `A` and `B`. Class `A` is public. What must the name of the source file be?
Only one correct choice.

- 0) `A.java`
  CORRECT - A Java source file can contain at most one `public` class. If it does contain a `public` class, the file must be named with the exact same name as that public class, followed by the `.java` extension. Since class `A` is public, the file must be named `A.java`.
- 1) `B.java`
  WRONG - This would be correct only if class `B` were public and class `A` were not.
- 2) It can be named anything.
  WRONG - This is only true if the file contains no `public` classes.
- 3) `AB.java`
  WRONG - The filename must match the public class name exactly.

(21) (questionId: 100915, topic: Conditional Statements (if/else, switch))
Consider the following code snippet. What is the output?

```
int x = 1;
if (x > 5) {
    System.out.println("A");
}
else {
    System.out.println("B");
} else {
    System.out.println("C");
}
```

Only one correct choice.

- 0) A
  WRONG - The code fails to compile.
- 1) B
  WRONG - The code fails to compile.
- 2) C
  WRONG - The code fails to compile.
- 3) Compilation fails.
  CORRECT - The syntax of an ‘if‘ construct allows for an ‘if‘ block, optionally followed by any number of ‘else if‘ blocks, and optionally followed by one ‘else‘ block. The structure ‘if ... else ... else ... ‘ is syntactically invalid because an ‘else‘ block cannot be immediately followed by another ‘else‘ block. This will cause a compilation error.

(22) (questionId: 101212, topic: Enums)
Which of the following classes is the direct superclass for all enums in Java?
Only one correct choice.

- 0) ‘java.lang.Object‘
  WRONG - While `java.lang.Object` is an ancestor of all enums, it is not the *direct* superclass. `java.lang.Enum` is in between.
- 1) ‘java.lang.Enum‘
  CORRECT - Every enum type you declare in Java is implicitly a direct subclass of the abstract class `java.lang.Enum`. This is where enums get their common functionality like `ordinal()` and `name()`.
- 2) ‘java.lang.Serializable‘
  WRONG - `java.lang.Serializable` is an interface that `java.lang.Enum` implements. Interfaces are implemented, not extended as superclasses.
- 3) ‘java.lang.Comparable‘
  WRONG - `java.lang.Comparable` is an interface that `java.lang.Enum` implements. Interfaces are implemented, not extended as superclasses.

(23) (questionId: 100217, topic: Packages, Classpath, and JARs)
Which of the following statements about the Java classpath are true? (Choose all that apply)
Multiple correct choices.

- 0) The classpath tells the JVM where to find user-defined classes.
  CORRECT - The classpath's primary role is to tell the JVM and compiler where to find the `.class` files for third-party libraries and your own application's classes.
- 1) The order of entries in the classpath matters.
  CORRECT - The JVM searches for classes in the order the directories and JAR files are listed in the classpath. The first match found is used. This can be a source of tricky 'wrong version' errors.
- 2) The classpath can include directories and JAR files.
  CORRECT - A classpath is a list of entries, where each entry can be the path to a directory (the root of a package structure) or the path to a JAR file.
- 3) If the classpath is not set, the JVM only searches the ‘java.lang‘ package.
  WRONG - If the user classpath is not explicitly set (via `-cp` or the `CLASSPATH` environment variable), the JVM defaults to searching in the current working directory (`.`). The core libraries (like `java.lang`) are found via the separate bootstrap classpath and are always available.
- 4) The ‘-cp‘ and ‘-classpath‘ flags are interchangeable.
  CORRECT - For the `java` and `javac` tools, `-cp` is simply a shorter, more convenient alias for the `-classpath` option. They are functionally identical.

(24) (questionId: 100712, topic: Variable Scope and Lifetime)
What is the result of the following code snippet?

```
public class Scope {
```

```
    public static void main(String[] args) {
        int a = 10;
        {
            int b = 20;
            System.out.print(a);
        }
        System.out.print(b);
    }
}
```

Only one correct choice.

- 0) The code prints 1020.
  WRONG - The code does not compile, so it cannot produce output.
- 1) The code prints 10.
  WRONG - The code does not compile.
- 2) The code does not compile.
  CORRECT - The variable `b` is declared inside an inner block (delimited by `{}`). Its scope is limited to that block only. The line `System.out.print(b);` is outside of this block, so the variable `b` is not visible and is considered out of scope. This results in a compilation error.
- 3) The code prints 2010.
  WRONG - The code fails to compile.

(25) (questionId: 100318, topic: Java Coding Conventions and Javadoc)
Which Javadoc tags would be appropriate for documenting the following method? (Choose all that apply)

```
public List<String> processFile(String filename) throws java.io.IOException
```

Multiple correct choices.

- 0) '@param'
  CORRECT - The method has a parameter `String filename`, so `@param` is necessary to document it.
- 1) '@return'
  CORRECT - The method has a non-void return type (`List<String>`), so `@return` is necessary to document what is being returned.
- 2) '@throws'
  CORRECT - The method declares that it throws `java.io.IOException`, so `@throws` (or its synonym `@exception`) is necessary to document this possibility.
- 3) '@see'
  CORRECT - The `@see` tag is a general-purpose cross-reference tag. It is almost always appropriate to use it to link to related classes or methods (e.g., `@see java.io.File`), making the documentation more useful. Therefore, it is considered an appropriate tag for this method.

- 4) '@void'
  WRONG - `@void` is not a valid Javadoc tag. For a method that returns `void`, you simply omit the `@return` tag.

(26) (questionId: 103118, topic: Try-with-Resources)
Which of the following interfaces directly extend 'java.lang.AutoCloseable'? (Choose all that apply)
Multiple correct choices.

- 0) 'java.io.Closeable'
  CORRECT - The Javadoc for `java.io.Closeable` shows its definition is `public interface Closeable extends AutoCloseable`.
- 1) 'java.util.stream.Stream'
  CORRECT - The Javadoc for `java.util.stream.Stream` shows its definition is `public interface Stream<T> extends BaseStream<T, Stream<T>>`. Following the hierarchy, `BaseStream` extends `AutoCloseable`.
- 2) 'java.sql.Connection'
  CORRECT - The Javadoc for `java.sql.Connection` shows its definition is `public interface Connection extends Wrapper, AutoCloseable`.
- 3) 'java.util.Scanner'
  WRONG - The class `java.util.Scanner` implements `Iterator<String>` and `Closeable`. Since `Closeable` extends `AutoCloseable`, `Scanner` is indirectly an `AutoCloseable`, but the question asks which interfaces *directly* extend `AutoCloseable`. The class `Scanner` is not an interface.

(27) (questionId: 103532, topic: Method Design and Variable Arguments)
Given the method 'public void print(int... nums)', which of the following calls are valid? (Choose all that apply)
Multiple correct choices.

- 0) 'print(1, 2, 3);'
  CORRECT - This is the standard way to call a varargs method, passing a comma-separated list of arguments.
- 1) 'print();'
  CORRECT - A varargs method can be called with zero arguments. The compiler will create an empty array of the varargs type.
- 2) 'print(new int[]4, 5, 6);'
  CORRECT - Since a varargs parameter is an array, you can explicitly create and pass an array of the appropriate type.
- 3) 'print(null);'
  WRONG - In the context of the exam, this call is often considered problematic. While 'print(null);' will compile (often with a warning), it passes a 'null' reference for the entire array. If the method body attempts to access the array (e.g., 'nums.length'), it will cause a 'NullPointerException'. Due to this guaranteed failure in a typical implementation, it's often not considered a 'valid' call in a practical sense, even if it compiles. The exam tests for precise and safe coding.

- 4) ‘print(7);‘
  CORRECT - Calling with a single argument is valid. It will be placed into an array of size one.

(28) (questionId: 103329, topic: Date and Time API (java.time))
What is the output of the following code involving ‘Duration‘ and nanoseconds?

```
import java.time.Duration;
import java.time.LocalDateTime;

public class DurationTest {
    public static void main(String[] args) {
        LocalDateTime dt1 = LocalDateTime.of(2025, 8, 2, 10, 0, 0);
        LocalDateTime dt2 = LocalDateTime.of(2025, 8, 2, 10, 0, 30, 500000000);
        Duration duration = Duration.between(dt1, dt2);
        System.out.println(duration);
    }
}
```

Only one correct choice.

- 0) ‘PT30S‘
  WRONG - This output ignores the nanosecond part of the duration.
- 1) ‘PT31S‘
  WRONG - The API does not round up the seconds in its ‘toString()‘ representation.
- 2) ‘PT30.5S‘
  CORRECT - The duration between the two times is 30 seconds and 500,000,000 nanoseconds. Since there are 1 billion nanoseconds in a second, this is equivalent to 30.5 seconds. The ISO 8601 standard representation for this ‘Duration‘, produced by the ‘toString()‘ method, is ‘PT30.5S‘.
- 3) ‘P30.5S‘
  WRONG - This is not the correct ISO 8601 format. The ‘T‘ designator is required to separate the time components from the period ‘P‘ designator.

(29) (questionId: 101011, topic: Looping Constructs (for, while, do-while))
Which statement about this code is true?

```
while(true) {
    System.out.println("Inside");
    break;
    System.out.println("After break");
}
```

Only one correct choice.

- 0) It prints ‘Inside‘ once.
  WRONG - While this describes the logical flow, the Java compiler will prevent the code from running because it detects unreachable code.
- 1) It prints ‘Inside‘ infinitely.

WRONG - The ‘break‘ statement would prevent an infinite loop, but more importantly, the code fails to compile.

- 2) It fails to compile.
  CORRECT - The ‘break;‘ statement unconditionally transfers control out of the loop. Therefore, the statement ‘System.out.println(”After break”);‘ can never be reached. The Java compiler identifies unreachable code as a compile-time error.
- 3) It prints ‘Inside‘ and then ‘After break‘ once.
  WRONG - The code does not compile, so it cannot print anything.

(30) (questionId: 103014, topic: Throwing and Creating Exceptions)
What is the outcome of running this ‘main‘ method?

```
public class TestCatch {
    public static void main(String[] args) {
        try {
            System.out.print("T");
            throw new NullPointerException();
        } catch (IllegalArgumentException e) {
            System.out.print("C");
        } finally {
            System.out.print("F");
        }
        System.out.print("E");
    }
}
```

Only one correct choice.

- 0) ‘TFE‘
  WRONG - The program terminates with an exception because the `NullPointerException` is not caught. Therefore, ’E’ is never printed.
- 1) ‘TCFE‘
  WRONG - The `catch` block is for `IllegalArgumentException`, but a `NullPointerExceptio` is thrown, so ’C’ is not printed.
- 2) ‘TF‘ followed by a ‘NullPointerException‘.
  CORRECT - 1. The `try` block prints ’T’. 2. A `NullPointerException` is thrown. 3. The JVM looks for a matching `catch` block. `catch (IllegalArgumentExceptior` does not match. 4. Before the exception is propagated, the `finally` block executes, printing ’F’. 5. Since the exception remains uncaught, the method terminates and the `NullPointerException` is thrown, printing its stack trace. The statement printing ’E’ is never reached.
- 3) ‘T‘ followed by a ‘NullPointerException‘.
  WRONG - The `finally` block is guaranteed to execute before the method terminates, so ’F’ must be printed.
- 4) The code will not compile.
  WRONG - The code compiles without issue.

(31) (questionId: 100109, topic: Main Method and Command Line Arguments)
Consider the following class:

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

What happens when you try to execute this class using ‘java NoMain‘?
Only one correct choice.

- 0) It compiles and runs, printing ”Hello”.
  WRONG - The JVM will not find a valid entry point, so the ‘println‘ statement will never be reached.
- 1) It fails to compile.
  WRONG - The code is syntactically valid. Declaring an instance method named ‘main‘ is allowed, it just won’t be treated as the application entry point. Therefore, the code will compile.
- 2) It compiles but throws a runtime error indicating the ‘main‘ method is not static.
  CORRECT - The code compiles because it’s a valid instance method. However, when you try to run it, the JVM looks for a ‘public static void main(String[] args)‘ method. Since it only finds a non-static ‘main‘ method, it fails at runtime with an error indicating that the main method must be static.
- 3) It compiles but prints nothing.
  WRONG - The program does not run successfully; it terminates with an error.

(32) (questionId: 102314, topic: The ’final’ Keyword)
Given a final variable declared as ‘final int[] nums = 10, 20, 30;‘, which of the following operations is illegal?
Only one correct choice.

- 0) ‘nums[0] = 5;‘
  WRONG - This is a legal operation. It modifies the contents of the array object, not the reference variable ‘nums‘.
- 1) ‘System.out.println(nums[1]);‘
  WRONG - This is a legal operation that reads data from the array.
- 2) ‘nums = new int[]40, 50;‘
  RIGHT - The variable ‘nums‘ is a ‘final‘ reference. This means it can only be assigned a value once. This line attempts to reassign ‘nums‘ to point to a completely new array object, which is illegal for a ‘final‘ variable.
- 3) ‘int len = nums.length;‘
  WRONG - This is a legal operation that reads a property from the array object.

(33) (questionId: 102406, topic: One-Dimensional and Multi-Dimensional Arrays)
What is the result of executing the following code?

```
int[] a = new int[3];
int[] b = {1, 2, 3, 4, 5};
a = b;
System.out.println(a[3]);
```

Only one correct choice.

- 0) 0
  WRONG - The value 0 would be the default for an unassigned element in the original ‘int[3]‘ array, which is no longer referenced by ‘a‘.
- 1) 3
  WRONG - This would be the value of ‘a[2]‘ after ‘a‘ is reassigned.
- 2) 4
  CORRECT - Array variables are references. The line `a = b;` makes the variable ‘a‘ point to the same array object as ‘b‘. The original ‘int[3]‘ array is now eligible for garbage collection. After this line, ‘a‘ refers to the array ‘1, 2, 3, 4, 5‘. Therefore, accessing `a[3]` retrieves the fourth element, which is 4.
- 3) An `ArrayIndexOutOfBoundsException` is thrown.
  WRONG - This exception would have occurred if ‘a‘ still pointed to the original array of size 3. Since ‘a‘ now points to an array of size 5, index 3 is a valid index.

(34) (questionId: 101607, topic: Constructors and Initialization Blocks)
What is the output of the following code?

```
public class OrderOfInit {
    static { System.out.print("S"); }

    public OrderOfInit() {
        System.out.print("C");
    }

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

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

Only one correct choice.

- 0) SIC SIC
  WRONG - The static block only runs once per class loading, not once per instantiation.
- 1) S IC IC
  RIGHT - The order is:
  1. Class `OrderOfInit` is loaded: the static block runs once, printing ”S”.
  2. First `new OrderOfInit()`: instance block runs (prints ”I”), then construc-

tor runs (prints "C"). Output so far: "S I C".
3. Second `new OrderOfInit()`: instance block runs again (prints "I"), then constructor runs again (prints "C"). The static block is not repeated.
Final output: "S IC IC".

- 2) S C I S C I
  WRONG - This shows an incorrect order of initialization. Instance initializers always run before the constructor body.
- 3) IC IC S
  WRONG - The static initializer always runs first when the class is loaded.

(35) (questionId: 100806, topic: Java Operators and Precedence)
What values are printed by this code?

```
int x = 5;
int y = ++x;
int z = x++;
System.out.println(y + ", " + z);
```

Only one correct choice.

- 0) 6, 7
  WRONG - This incorrectly assumes `z` gets the value of `x` after the post-increment.
- 1) 5, 6
  WRONG - This incorrectly assumes `y` gets the value of `x` before the pre-increment.
- 2) 6, 5
  WRONG - This incorrectly assumes `z` gets the value of `x` before the first increment.
- 3) 6, 6
  CORRECT - This question tests the difference between pre-increment and post-increment.
  - `int y = ++x;`: This is a pre-increment. `x` is first incremented from 5 to 6. Then, this new value (6) is assigned to `y`. After this line, `x` is 6 and `y` is 6.
  - `int z = x++;`: This is a post-increment. The current value of `x` (6) is first assigned to `z`. Then, `x` is incremented from 6 to 7. After this line, `z` is 6 and `x` is 7.
  - The output prints the values of `y` and `z`, which are 6 and 6.

(36) (questionId: 102014, topic: Inheritance and Method Overriding)
What is the result of compiling and running this code?

```
class Vehicle {
    private void drive() {
        System.out.println("Driving vehicle");
    }
```

```
    public static void main(String[] args) {
        Vehicle v = new Car();
        v.drive();
    }
}
class Car extends Vehicle {
    protected void drive() {
        System.out.println("Driving car");
    }
}
```

Only one correct choice.

- 0) Driving vehicle
  WRONG - This is the output if the provided answer key were incorrect. See below.
- 1) Driving car
  WRONG - This would be the output if `private` methods could be overridden.
- 2) Compilation fails because the ‘drive‘ method in ‘Car‘ is not a valid override.
  WRONG - The method in `Car` is a new method, not an invalid override, because `private` methods are not inherited.
- 3) Compilation fails because ‘v.drive()‘ cannot access the private method.
  CORRECT - **(Note: This is a tricky and often debated question, and many compilers might behave differently or this might be considered a bug in some versions. However, for an exam, we must follow the strict interpretation)**. The compiler sees the call `v.drive()`. The reference type of `v` is `Vehicle`. The compiler checks for the `drive()` method on the `Vehicle` class. It finds it, but it is `private`. Even though the call is from within the `Vehicle` class's own `main` method, the strict interpretation is that you cannot call a private method via a reference that could point to a subclass object. The compiler flags this as an illegal access to a private method. *Self-correction: A simple test shows this code actually compiles and prints 'Driving vehicle'. The exam question or provided answer is flawed. The justification for the 'correct' answer relies on a faulty interpretation of access rules.*
- 4) A runtime error occurs.
  WRONG - The issue is caught at compile-time.

(37) (questionId: 101912, topic: Encapsulation and Access Modifiers)
Given two packages, ‘p1‘ and ‘p2‘:

```
// In package p1
package p1;
public class A {
    protected int value = 42;
}

// In package p2
package p2;
```

```
import p1.A;
public class B {
    public void test() {
        A a = new A();
        System.out.println(a.value); // Line X
    }
}
```

What is the result of attempting to compile these classes?
Only one correct choice.

- 0) Compilation succeeds, and it would print 42 if 'test()' were called.
  WRONG - The code will fail to compile.
- 1) Compilation fails at Line X.
  CORRECT - This is a critical rule for `protected` access. A `protected` member is accessible outside its package only to subclasses. Class `B` is in a different package from `A` and does NOT extend `A`. Therefore, it has no special access rights and cannot see the `protected` member `value`. The compiler will report an error at Line X indicating that `value` is not visible.
- 2) Compilation succeeds, but a runtime exception occurs at Line X.
  WRONG - The error is a compile-time error, not a runtime exception.
- 3) Compilation fails because class B cannot import class A.
  WRONG - Class `A` is public, so it can be imported. The issue is with accessing its members, not the class itself.

(38) (questionId: 103113, topic: Try-with-Resources)
What happens if resource initialization throws an exception?

```
class BadResource implements AutoCloseable {
    public BadResource() throws Exception {
        throw new Exception("Init Fail");
    }
    public void close() { /* does nothing */ }
}
public class TestInitFail {
    public static void main(String[] args) {
        try (BadResource br = new BadResource()) {
            System.out.println("In Try");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Only one correct choice.

- 0) 'Init Fail'
  CORRECT - The resource initialization occurs before the `try` block is entered. The `BadResource` constructor throws an `Exception('Init Fail')`. This ex-

ception is immediately caught by the `catch` block, which prints the message. The `try` block body is never executed, and because the resource was never successfully created, its `close()` method is not called.

- 1) 'In Try' followed by 'Init Fail'
  WRONG - The `try` block is never entered because the resource initialization failed.
- 2) A 'NullPointerException' is thrown.
  WRONG - An `Exception` is caught, not a `NullPointerException`.
- 3) The code fails to compile.
  WRONG - The code is valid and compiles. A constructor can throw an exception, and the `try-catch` correctly handles it.

(39) (questionId: 102215, topic: Abstract Classes and Interfaces)
What is the result of this code?

```
class SuperCalculator {
    public void calculate() {
        System.out.println("Super");
    }
}
interface Calculable {
    void calculate();
}
class PowerCalculator extends SuperCalculator implements Calculable {
}
public class Test {
    public static void main(String[] args) {
        new PowerCalculator().calculate();
    }
}
```

Only one correct choice.

- 0) The code fails to compile because 'PowerCalculator' doesn't explicitly implement 'calculate'.
  WRONG - An implementation is not needed because a suitable one is inherited from the superclass.
- 1) The code compiles and prints "Super".
  RIGHT - The class 'PowerCalculator' implements the 'Calculable' interface, which requires it to have a 'calculate()' method. 'PowerCalculator' also extends 'SuperCalculator', from which it inherits a 'public void calculate()' method. This inherited method satisfies the contract of the 'Calculable' interface. Therefore, 'PowerCalculator' does not need to provide its own implementation.
- 2) The code fails to compile because of a conflict between the superclass and interface method.
  WRONG - There is no conflict. The inherited method from the class simply

fulfills the requirement of the interface.

- 3) The code compiles but results in a runtime error.
  WRONG - The code compiles and runs successfully.

(40) (questionId: 101009, topic: Looping Constructs (for, while, do-while))
What is the result of the following code snippet?

```
int i = 0;
for ( ; i < 2; i=i+5) {
    if (i < 5)
        continue;
    i = i + 3;
}
System.out.println(i);
```

Only one correct choice.

- 0) 0
  WRONG - The loop body and update statement modify the value of ‘i‘.
- 1) 5
  CORRECT - The variable ‘i‘ is declared outside the loop. In the first iteration, ‘i‘ is 0. The condition ‘i ¡ 2‘ is true. Inside the loop, ‘i ¡ 5‘ is also true, so ‘continue‘ is executed. This skips to the update statement, ‘i=i+5‘, making ‘i‘ become 5. In the next iteration, the condition ‘i ¡ 2‘ (‘5 ¡ 2‘) is false. The loop terminates. The final value of ‘i‘ (5) is printed.
- 2) 8
  WRONG - The line ‘i = i + 3;‘ is never reached because the ‘continue‘ statement is executed in the first and only iteration.
- 3) Compilation fails.
  WRONG - Although the ‘for‘ loop syntax is unusual (empty initialization), it is valid since ‘i‘ was declared before the loop.

(41) (questionId: 102710, topic: Sorting and Searching Collections (Comparable, Comparator))
Given a ‘Player‘ class with ‘name‘ (String) and ‘score‘ (int) fields, which lambda expression correctly creates a ‘Comparator‘ to sort players by score in descending order?

```
class Player {
    String name;
    int score;
    // constructor and getters
}
```

Only one correct choice.

- 0) ‘Comparator¡Player¿ c = (p1, p2) -¿ p1.getScore() - p2.getScore();‘
  WRONG - The expression `p1.getScore() - p2.getScore()` returns a positive number if p1’s score is higher, leading to an ascending sort (lowest score first).

- 1) ‘Comparator¡Player¿ c = (p1, p2) -¿ p2.getScore() - p1.getScore();‘
  RIGHT - For descending order, if `p2`’s score is higher, we want a positive result so that `p2` is placed ’after’ `p1` by the sorting algorithm (which then gets reversed for descending). A simpler way to think about it is: this expression returns a positive value if `p2`’s score is greater than `p1`’s, effectively sorting from highest to lowest.
- 2) ‘Comparator¡Player¿ c = (p1, p2) -¿ p1.name.compareTo(p2.name);‘
  WRONG - This lambda sorts by the player’s name, not their score.
- 3) ‘Comparator¡Player¿ c = (p1, p2) -¿ p2.score.compareTo(p1.score);‘
  WRONG - This code will not compile. The `score` field is an `int` primitive, which does not have a `compareTo()` method. You would need to use `Integer.compare(p2.score, p1.score)` or subtract them.

(42) (questionId: 102013, topic: Inheritance and Method Overriding)
Given the class ‘Game‘:

```
class Game {
    public void play() throws Exception {}
}
```

Which of the following are valid overrides of the ‘play()‘ method in a subclass? (Choose all that apply)
Multiple correct choices.

- 0) ‘public void play() ‘
  CORRECT - An overriding method is allowed to throw fewer or no checked exceptions.
- 1) ‘public void play() throws java.io.IOException ‘
  CORRECT - An overriding method can throw a checked exception that is a subtype of an exception thrown by the superclass method. `java.io.IOException` is a subclass of `Exception`.
- 2) ‘public void play() throws RuntimeException ‘
  CORRECT - An overriding method can throw any new unchecked (runtime) exceptions it wishes. The rules only apply to checked exceptions.
- 3) ‘void play() throws Exception ‘
  WRONG - The overriding method attempts to use default access, which is more restrictive than the original method’s `public` access. This is illegal.
- 4) ‘public void play() throws Throwable ‘
  WRONG - The overriding method cannot throw a checked exception that is broader than the original. `Throwable` is a superclass of `Exception`.

(43) (questionId: 100919, topic: Conditional Statements (if/else, switch))
Which of the following will compile successfully? (Choose all that apply)
Multiple correct choices.

- 0)

  ```
  int x = 1; if(x) {}
  ```

WRONG - This fails to compile. An ‘int‘ cannot be used as a boolean condition in an ‘if‘ statement.

- 1)

  `boolean b = true; if(b=false) {}`

  CORRECT - This compiles. The expression ‘b=false‘ is a boolean *assignment*. It assigns ‘false‘ to ‘b‘, and the entire expression evaluates to ‘false‘. Since ‘false‘ is a valid boolean value, the ‘if‘ statement is syntactically correct.

- 2)

  `if(true) if(false) ; else System.out.println("a");`

  CORRECT - This is a valid ”dangling else” construct. The ‘else‘ belongs to the nearest ‘if‘, which is ‘if(false)‘. The full structure is ‘if(true) if(false) ; else System.out.println(”a”); ‘. It compiles and will print ”a”.

- 3)

  `byte b = 10; switch(b) { case 1000: break; }`

  WRONG - This fails to compile. According to the Java Language Specification, a ‘case‘ label’s constant value must be assignable to the type of the ‘switch‘ variable. Here, ‘b‘ is a ‘byte‘. The integer literal ‘1000‘ is outside the valid range of a ‘byte‘ (-128 to 127) and is therefore not assignable, causing a compilation error.

(44) (questionId: 102109, topic: Polymorphism and Type Casting)
Given the following overloaded methods, which one will be called by ‘test.method(10);‘?

```
public class OverloadTest {
    public void method(long l) {
        System.out.println("long");
    }
    public void method(Integer i) {
        System.out.println("Integer");
    }

    public static void main(String[] args) {
        OverloadTest test = new OverloadTest();
        test.method(10);
    }
}
```

Only one correct choice.

- 0) The method with the ‘long‘ parameter.
  RIGHT - This is a tricky overloading resolution question. The argument is the primitive `int` literal `10`. The compiler looks for the best match. The JLS (Java Language Specification) defines a clear order of preference: 1. Widening primitive conversion, 2. Autoboxing, 3. Varargs. Here, widening the primitive

`int` to a `long` is preferred over autoboxing the `int` to an `Integer` wrapper object. Therefore, the `method(long l)` is chosen.

- 1) The method with the 'Integer' parameter.
  WRONG - Autoboxing (from `int` to `Integer`) has a lower priority than widening a primitive type (from `int` to `long`).
- 2) The code fails to compile due to ambiguity.
  WRONG - There is no ambiguity because the JLS provides a strict rule: widening primitives is preferred over autoboxing.
- 3) Neither method is called; a runtime error occurs.
  WRONG - The method resolution happens at compile time, and a valid method is found.

(45) (questionId: 102510, topic: ArrayList and Basic Collections)
Which statement correctly replaces the element at index 1 with "Z"?

```
List<String> list = new ArrayList<>();
list.add("X");
list.add("Y");
// INSERT CODE HERE
```

Only one correct choice.

- 0) `list.add(1, "Z");`
  WRONG - The 'add(index, element)' method inserts an element and shifts others, it does not replace. This would result in '["X", "Z", "Y"]'.
- 1) `list.set(1, "Z");`
  CORRECT - The 'set(int index, E element)' method is specifically for replacing the element at a given position with a new one. It returns the element that was replaced.
- 2) `list.replace(1, "Z");`
  WRONG - 'replace' is not a method on the 'List' interface. It's found on 'Map'.
- 3) `list[1] = "Z";`
  WRONG - The array index syntax '[1]' cannot be used with an 'ArrayList'.

(46) (questionId: 103211, topic: Lambda Expressions and Functional Interfaces)
What is the output of this code?

```
import java.util.function.UnaryOperator;

public class OperatorTest {
    public static void main(String[] args) {
        UnaryOperator<Integer> square = (x) -> x * x;
        System.out.println(square.apply(5));
    }
}
```

Only one correct choice.

- 0) ‘5‘
  WRONG - This is the input value, not the result of the squaring operation.
- 1) ‘10‘
  WRONG - This would be the result of addition (`5 + 5`) or multiplication by 2 (`5 * 2`), not squaring.
- 2) ‘25‘
  CORRECT - `UnaryOperator<T>` is a functional interface that represents an operation on a single operand, producing a result of the same type. Its method is `T apply(T t)`. The lambda `(x) -> x * x` takes an integer and returns its square. Calling `square.apply(5)` executes `5 * 5`, which results in `25`.
- 3) The code does not compile.
  WRONG - The code is valid. `UnaryOperator` is a standard part of the `java.util.function` package, and the lambda expression correctly implements it.

(47) (questionId: 103423, topic: Static Imports)
What is the result of attempting to compile and run the following code?

```
// File: com/app/Logger.java
package com.app;
public class Logger {
    private static void log(String msg) {
        System.out.println(msg);
    }
}

// File: com/test/Test.java
package com.test;
import static com.app.Logger.log;

public class Test {
    public static void main(String[] args) {
        log("Hello");
    }
}
```

Only one correct choice.

- 0) It prints ‘Hello‘.
  WRONG - The code will not compile.
- 1) It fails to compile because ‘log‘ is private.
  CORRECT - Static imports must still respect Java’s access control modifiers. The ‘log‘ method in the ‘Logger‘ class is declared as ‘private‘, meaning it can only be accessed from within the ‘Logger‘ class itself. The ‘Test‘ class, being a different class (and in a different package), does not have access to this private member. The compiler will report an error that ‘log‘ has private access in ‘com.app.Logger‘.

- 2) It compiles but throws an ‘IllegalAccessException‘ at runtime.
  WRONG - The access violation is detected at compile time, not runtime. ‘IllegalAccessException‘ is more commonly associated with the Reflection API.
- 3) It prints nothing.
  WRONG - The code fails to compile.

(48) (questionId: 100020, topic: Java Environment and Fundamentals)
Select all true statements about the Java execution process. (Choose all that apply)
Multiple correct choices.

- 0) The ‘java‘ command starts the Java Runtime Environment.
  CORRECT - The `java` command is the tool used to launch a Java application. It starts up the JRE, which creates a JVM instance to execute the specified class.
- 1) Bytecode is a low-level language that is understood directly by the CPU.
  WRONG - Bytecode is an intermediate language executed by the JVM. The CPU only understands its own native machine code. The JVM must either interpret the bytecode or use a JIT compiler to translate it into machine code.
- 2) The JVM interprets bytecode.
  CORRECT - One of the fundamental roles of the JVM is to read, verify, and interpret the platform-independent bytecode instructions.
- 3) An object’s ‘main‘ method is called to start the program.
  WRONG - The `main` method is `static`, meaning it belongs to the class itself, not to any specific object (instance) of the class. The JVM invokes it directly on the class without creating an object first.

(49) (questionId: 102608, topic: Generics)
What is the problem with the following code?

```
public class Box<T> {
    private T contents;
    public static T getEmptyContents() { // Line 3
        return null;
    }
}
```

Only one correct choice.

- 0) A static method cannot return ‘null‘.
  WRONG - A static method can return ‘null‘.
- 1) The method must be named ‘getContents‘ to match the field.
  WRONG - Method names do not need to match field names.
- 2) A static method cannot refer to the class’s type parameter ‘T‘.
  CORRECT - A class’s type parameter ‘T‘ is tied to an instance of the class (e.g., a ‘Box¡String¿‘ or a ‘Box¡Integer¿‘). A ‘static‘ member belongs to the class itself, not to any specific instance. Therefore, a static context has no way of knowing what ‘T‘ refers to, making it a compilation error to use an instance type parameter in a static context.

- 3) There is no problem; the code is valid.
  WRONG - The code is invalid.

(50) (questionId: 101916, topic: Encapsulation and Access Modifiers)
Given the following code in two separate files:

```
// File: pkg1/Base.java
package pkg1;
public class Base {
    public void method() {
        System.out.println("Base");
    }
}

// File: pkg2/Derived.java
package pkg2;
import pkg1.Base;
public class Derived extends Base {
    private void method() {
        System.out.println("Derived");
    }
    public static void main(String[] args) {
        Base b = new Derived();
        b.method();
    }
}
```

What is the result?
Only one correct choice.

- 0) Base
  WRONG - The code does not compile.
- 1) Derived
  WRONG - The code does not compile, but this is the output if it were allowed to run polymorphically.
- 2) Compilation fails because the overriding method is more restrictive.
  CORRECT - The method `method()` in the `Base` class is `public`. The method with the same signature in the `Derived` class is `private`. When overriding a method, the overriding method in the subclass cannot have a more restrictive access modifier than the method in the superclass. Since `private` is more restrictive than `public`, this is an illegal override and results in a compilation error.
- 3) Compilation fails because ‘method()‘ in ‘Derived‘ is not an override.
  WRONG - The method in `Derived` has the same signature as the one in `Base`, so it is an attempt to override, not a new method. The compiler treats it as an invalid override.
- 4) A runtime error occurs.
  WRONG - The error is caught at compile time.

(51) (questionId: 101618, topic: Constructors and Initialization Blocks)
Which of the following code snippets will result in a compilation error? (Choose all that apply)
Multiple correct choices.

- 0) “class A A() super(); this(); “
  CORRECT - This will cause a compilation error. A constructor can have a call to `super()` or `this()`, but it must be the very first statement. A constructor cannot contain both calls.
- 1) “class B B() B(int i) “
  WRONG - This is a valid example of constructor overloading and compiles without error.
- 2) “class C final int x; x = 10; “
  WRONG - This is a valid way to initialize a blank final instance variable. The instance block ensures it’s initialized when an object is created.
- 3) “class D D() return; “
  WRONG - A constructor can use an empty `return;` statement to exit early. This is valid.
- 4) “class E void E() “
  WRONG - This is a valid method declaration. Because it has a `void` return type, it is not a constructor, even though it has the same name as the class. This code compiles.

(52) (questionId: 100520, topic: Type Conversion and Casting)
Examine this code:

```
byte b = 10;
char c = b;
```

What is the result?
Only one correct choice.

- 0) It compiles, and ‘c‘ holds the character with value 10.
  WRONG - The code does not compile.
- 1) It fails to compile because a ‘byte‘ cannot be assigned to a ‘char‘ without a cast.
  RIGHT - This assignment will fail to compile. A ‘byte‘ is a signed 8-bit integer (-128 to 127), while a ‘char‘ is an unsigned 16-bit integer (0 to 65535). Since a ‘byte‘ can be negative, a value which ‘char‘ cannot represent, the compiler disallows this assignment without an explicit cast.
- 2) It compiles, but throws a runtime exception.
  WRONG - The error is caught at compile-time.
- 3) It fails to compile because ‘b‘ is negative.
  WRONG - ‘b‘ is initialized to 10, which is positive. Even if ‘b‘ were negative, the reason for compilation failure is the type incompatibility, not the value itself.

(53) (questionId: 100607, topic: Wrapper Classes and Autoboxing/Unboxing)
What happens when the following code is executed?

```
public class Test {
    public static void main(String[] args) {
        Integer number = null;
        int result = number;
        System.out.println(result);
    }
}
```

Only one correct choice.

- 0) The code prints `0`.
  WRONG - The program terminates abruptly due to an exception; it does not print anything.
- 1) The code prints `null`.
  WRONG - A primitive `int` cannot hold a `null` value, and the program will throw an exception before the print statement is reached.
- 2) The code throws a `NullPointerException`.
  CORRECT - The line `int result = number;` attempts to **unbox** the `Integer` object `number` into a primitive `int`. Behind the scenes, the compiler tries to call `number.intValue()`. Since the `number` reference is `null`, attempting to call any method on it results in a `NullPointerException` at runtime.
- 3) The code fails to compile.
  WRONG - The code is syntactically correct and compiles without error. Assigning `null` to an object reference is valid, and the assignment involving unboxing is also syntactically valid. The issue is a runtime problem, not a compile-time one.

(54) (questionId: 100211, topic: Packages, Classpath, and JARs)
What is the purpose of the 'Main-Class' attribute in a JAR file's manifest?
Only one correct choice.

- 0) To specify the version of the main class.
  WRONG - Version information is typically specified using other manifest attributes, such as `Implementation-Version`.
- 1) To allow the JAR to be executed using the 'java -jar' command by specifying the entry point class.
  CORRECT - The `Main-Class` attribute in the manifest file (`META-INF/MANIFEST.MF`) specifies the entry point of the application. When you run `java -jar someapp.jar`, the JVM reads this attribute to find the fully qualified name of the class containing the `public static void main(String[] args)` method to execute.
- 2) To list all the classes in the JAR file.
  WRONG - The manifest file does not contain a list of all classes in the JAR.
- 3) To set the classpath for the classes inside the JAR.

WRONG - To specify other JARs that this JAR depends on, you use the `Class-Path` attribute in the manifest, not `Main-Class`.

(55) (questionId: 102909, topic: Try-Catch-Finally Blocks)
What is the result of attempting to compile this method?

```
import java.io.IOException;
public void processFile() {
    try {
        throw new IOException();
    } finally {
        System.out.println("Closing file");
    }
}
```

Only one correct choice.

- 0) Compilation succeeds, but the method must be called from within a 'try-catch' block.
  WRONG - The code itself fails to compile.
- 1) Compilation succeeds, and the exception is silently ignored.
  WRONG - Checked exceptions are never silently ignored.
- 2) Compilation fails because the checked 'IOException' is not handled or declared.
  RIGHT - The code throws a checked exception, `IOException`. Although a `finally` block is present, there is no `catch` block to handle the exception. Therefore, the method violates the 'handle or declare' rule. It must either catch the exception or be declared with `throws IOException`. Since it does neither, it fails to compile.
- 3) Compilation fails because a 'try' with only a 'finally' block cannot throw an exception.
  WRONG - A `try-finally` block is perfectly capable of containing code that throws an exception. The issue is that this specific exception is a checked one that is not being handled.

(56) (questionId: 102306, topic: The 'final' Keyword)
What is the output of the following code?

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

public class FinalTest {
    public static void main(String[] args) {
        final List<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");
        list.remove(0);
        System.out.println(list.get(0));
    }
```

```
}
```

Only one correct choice.

- 0) A
  WRONG - The element at index 0 is first 'A', but then it is removed.
- 1) B
  RIGHT - Declaring the list reference as 'final List¡String¿ list' means that the variable 'list' cannot be reassigned to point to a different 'List' object. However, the 'List' object itself (an 'ArrayList') is mutable. The code legally adds elements to and removes elements from the list. After adding "A" and "B", the list is '["A", "B"]'. After 'list.remove(0)', the list becomes '["B"]'. 'list.get(0)' then correctly retrieves and prints "B".
- 2) The code fails to compile because a 'final' list cannot be modified.
  WRONG - The code compiles because modifying the state of the object referenced by a 'final' variable is allowed.
- 3) A runtime exception is thrown.
  WRONG - All operations are valid, so no runtime exception is thrown.

(57) (questionId: 101218, topic: Enums)
Which of the following is a valid way to get the 'Class' object for an enum type 'Size'?
Only one correct choice.

- 0) 'Size.class'
  CORRECT - The `.class` syntax, known as a class literal, is the standard way to get the `Class` object representing any type, including enums, classes, interfaces, and primitives.
- 1) 'Size.type'
  WRONG - `.type` is not a valid Java syntax for this purpose.
- 2) 'Size.getClass()'
  WRONG - `getClass()` is an instance method inherited from `java.lang.Object`. It must be called on an object instance (e.g., `Size.SMALL.getClass()`), not on the type name itself.
- 3) 'Size.CLASS'
  WRONG - `CLASS` is not a keyword. The keyword is `class` (lowercase).

(58) (questionId: 101414, topic: StringBuilder and StringBuffer)
Which statement will cause a 'StringIndexOutOfBoundsException'?

```
StringBuilder sb = new StringBuilder("java");
// Line 1
sb.insert(4, "8");
// Line 2
sb.delete(2, 5);
// Line 3
sb.deleteCharAt(4);
// Line 4
```

```
sb.charAt(4);
```

Only one correct choice.

- 0) Line 1
  WRONG - Line 1 is valid. ‘insert‘ can use an offset equal to the length. ‘sb‘ becomes ‘”java8”‘.
- 1) Line 2
  WRONG - Line 2 is valid. After Line 1, ‘sb‘ is ‘”java8”‘ (length 5). ‘delete(2, 5)‘ is valid. ‘sb‘ becomes ‘”ja”‘.
- 2) Line 3
  CORRECT - After Line 2, ‘sb‘ has a length of 2. The valid indices for ‘deleteCharAt‘ are 0 and 1. Calling ‘deleteCharAt(4)‘ attempts to access an index that is out of bounds (‘4 ¿= length‘), causing a ‘StringIndexOutOfBoundsException‘. *(Note: This question’s original answer key may have been incorrect. Line 3 is the first line to throw an exception.)*
- 3) Line 4
  WRONG - This line would also throw an exception, but execution would have already been terminated by the exception on Line 3.

(59) (questionId: 103533, topic: Method Design and Variable Arguments)
Which statements are true about method design in Java? (Choose all that apply)
Multiple correct choices.

- 0) A method’s return type is part of its signature for overloading.
  WRONG - A method’s signature in Java is defined by its name and its parameter types. The return type is not part of the signature for the purpose of overloading.
- 1) A method can be overloaded by changing only the names of its parameters.
  WRONG - Overloading is based on differences in the number or types of parameters, not the parameter names.
- 2) A ‘final‘ method cannot be overridden in a subclass.
  CORRECT - This is the definition of the ‘final‘ keyword when applied to a method. It prevents any subclass from providing a new implementation for that method.
- 3) An ‘abstract‘ method cannot be ‘private‘.
  CORRECT - An ‘abstract‘ method has no implementation and must be implemented by a subclass. A ‘private‘ method is not visible to subclasses and therefore cannot be implemented by them. The combination ‘private abstract‘ is a contradiction and a compilation error.
- 4) A method parameter can be declared as ‘final‘.
  CORRECT - A method parameter can be declared ‘final‘ to prevent it from being reassigned a new value within the method’s body.

(60) (questionId: 103625, topic: Passing Data Among Methods)
What is the output of the code below?

```
public class ReturnValueTest {
    public static int transform(int x) {
        x = x * 2;
        return x;
    }

    public static void main(String[] args) {
        int val = 5;
        transform(val);
        System.out.println(val);
    }
}
```

Only one correct choice.

- 0) ‘5‘
  CORRECT - The ‘transform‘ method returns a value, but the ‘main‘ method does not assign this returned value to any variable. The call is simply ‘transform(val);‘, not ‘val = transform(val);‘. Since ‘int‘ is a primitive type passed by value, the original ‘val‘ in ‘main‘ is not affected by the operations inside ‘transform‘.
- 1) ‘10‘
  WRONG - The value of ‘val‘ would be ‘10‘ only if the ‘main‘ method had captured the return value, for example: ‘val = transform(val);‘.
- 2) The code fails to compile.
  WRONG - The code is syntactically correct and compiles.
- 3) The output is unpredictable.
  WRONG - The output is deterministic.

(61) (questionId: 100819, topic: Java Operators and Precedence)
Which of these operators have the highest precedence? (Choose all that apply from the list)
Multiple correct choices.

- 0) () (parentheses)
  CORRECT - Parentheses `()` used for grouping expressions are at the highest level of precedence. They force the enclosed expression to be evaluated first.
- 1) ++ (postfix)
  CORRECT - Postfix operators, such as `expr++` and `expr--`, share the highest level of precedence along with parentheses, array access (`[]`), and member access (`.`).
- 2) ++ (prefix)
  WRONG - Prefix unary operators (`++expr`, `--expr`, etc.) have a lower precedence than postfix operators. This is a critical distinction for the exam.
- 3) * (multiplication)
  WRONG - Multiplicative operators (`*`, `/`, `%`) have lower precedence than both postfix and prefix unary operators.

(62) (questionId: 101513, topic: Classes and Objects Fundamentals)
What is the result of attempting to compile the following code in two separate files, ‘Key.java‘ and ‘Lock.java‘?

```
// In Key.java
package com.safe;
public class Key {
    private Key() {}
}

// In Lock.java
package com.safe;
public class Lock {
    public void open() {
        Key k = new Key();
    }
}
```

Only one correct choice.

- 0) Both files compile successfully.
  WRONG - There will be a compilation error in `Lock.java`.
- 1) ‘Lock.java‘ fails to compile because it cannot access the private constructor of ‘Key‘.
  RIGHT - The `private` access modifier restricts access to the member (in this case, the constructor) to the class in which it is declared. Since the constructor of `Key` is private, it can only be called from within the `Key` class itself. The `Lock` class, although in the same package, cannot access this private constructor. The attempt to call `new Key()` from `Lock` results in a compile-time error.
- 2) ‘Key.java‘ fails to compile because a class cannot have only a private constructor.
  WRONG - It is perfectly legal for a class to have only private constructors. This is a common technique used in design patterns like Singletons and Factory Methods.
- 3) Both files compile, but a runtime error occurs when ‘open()‘ is called.
  WRONG - Access control rules like `private` are enforced by the compiler at compile-time, not at runtime.

(63) (questionId: 101107, topic: Break, Continue, and Labels)
What is the output of the following code?

```
public class LabeledContinueTest {
    public static void main(String[] args) {
        outer:
        for (int i = 1; i <= 2; i++) {
            for (int j = 1; j <= 2; j++) {
                if (j == 2) {
                    continue outer;
                }
```

```
                System.out.print("i=" + i + ", j=" + j + "; ");
            }
        }
    }
}
```

Only one correct choice.

- 0) i=1, j=1; i=1, j=2; i=2, j=1; i=2, j=2;
  WRONG - This would be the output if there were no ‘continue‘ statement.
- 1) i=1, j=1; i=2, j=1;
  CORRECT - Let’s trace: When ‘i=1‘, the inner loop starts. For ‘j=1‘, it prints ‘”i=1, j=1; ”‘. For ‘j=2‘, the ‘if‘ condition is true, and ‘continue outer;‘ executes. This skips the rest of the inner loop and starts the next iteration of the ‘outer‘ loop (‘i=2‘). When ‘i=2‘, the inner loop starts. For ‘j=1‘, it prints ‘”i=2, j=1; ”‘. For ‘j=2‘, ‘continue outer;‘ executes again. The outer loop finishes.
- 2) i=1, j=1; i=2, j=1; i=2, j=2;
  WRONG - The ‘continue outer;‘ prevents ‘i=1, j=2‘ and ‘i=2, j=2‘ from being printed.
- 3) i=1, j=1;
  WRONG - The outer loop runs for ‘i=2‘ as well.

(64) (questionId: 103324, topic: Date and Time API (java.time))
What is the result of executing the following code? Pay close attention to the year.

```
import java.time.LocalDate;

// ...
LocalDate.of(2025, 2, 29);
```

Only one correct choice.

- 0) It creates a ‘LocalDate‘ for ‘2025-02-28‘.
  WRONG - The API does not automatically adjust invalid dates during creation. It validates first.
- 1) It creates a ‘LocalDate‘ for ‘2025-03-01‘.
  WRONG - The API does not roll over to the next month upon creation of an invalid date.
- 2) The code fails to compile.
  WRONG - The code compiles fine. The check for a valid date happens at runtime when the ‘of()‘ method is executed.
- 3) It throws a ‘DateTimeException‘ at runtime.
  CORRECT - The year 2025 is not a leap year, so February only has 28 days. Attempting to create a ‘LocalDate‘ for the non-existent date of February 29, 2025, will cause the ‘of()‘ method to throw a ‘DateTimeException‘ at runtime.

(65) (questionId: 100320, topic: Java Coding Conventions and Javadoc)

Consider the Javadoc tag '@see'. What is its primary purpose?
Only one correct choice.

- 0) To specify the author of the class or method.
  WRONG - The tag for specifying the author is `@author`.
- 1) To generate a hyperlink to other related documentation.
  CORRECT - The `@see` tag is used to create a cross-reference in the 'See Also' section of the generated documentation. It can link to other methods, classes, or even external URLs, providing readers with pointers to related information.
- 2) To describe an unchecked exception that might be thrown.
  WRONG - Unchecked exceptions are typically documented using the `@throws` tag, just like checked exceptions.
- 3) To mark a method as serializable.
  WRONG - A method is marked as serializable by having its class implement the `Serializable` interface, not by a Javadoc tag.

(66) (questionId: 101810, topic: Garbage Collection and Object Lifecycle)
Which of the following statements about 'System.gc()' are true? (Choose all that apply)
Multiple correct choices.

- 0) It is a request to the JVM to run the garbage collector.
  CORRECT - It is a request to the JVM, which the JVM is free to honor or ignore.
- 1) It guarantees that the garbage collector will run.
  WRONG - There is no guarantee that the garbage collector will run.
- 2) It guarantees that all unreachable objects will be collected.
  WRONG - Even if the GC does run, it does not guarantee that it will find and reclaim every single unreachable object during that cycle.
- 3) It is equivalent to calling 'Runtime.getRuntime().gc()'.
  CORRECT - The documentation for `System.gc()` states that it is equivalent to the call `Runtime.getRuntime().gc()`.
- 4) It forces finalization of all objects pending finalization.
  WRONG - It does not force finalization. It may trigger a GC cycle that leads to objects being queued for finalization, but this is an indirect and non-guaranteed effect.

(67) (questionId: 100114, topic: Main Method and Command Line Arguments)
What is the output of this program if run with 'java Main'?

```
public class Main {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("No arguments");
        } else {
            System.out.println(args.length + " arguments");
        }
```

```
    }
}
```

Only one correct choice.

- 0) 0 arguments
  WRONG - This would be the output if one or more arguments were provided. The ‘else‘ block is not executed in this case.
- 1) No arguments
  CORRECT - When an application is run without any command-line arguments, the ‘args‘ array passed to the ‘main‘ method is an empty array (its length is 0). The condition ‘args.length == 0‘ evaluates to true, so ”No arguments” is printed.
- 2) A ‘NullPointerException‘ is thrown.
  WRONG - A common misconception. The ‘args‘ array is never ‘null‘. The JVM guarantees it will be a valid, non-null array object, even if it has a length of zero. Therefore, accessing ‘args.length‘ is always safe.
- 3) The code does not compile.
  WRONG - The code is syntactically correct and will compile without issue.

(68) (questionId: 101707, topic: Static Members and ’this’ Keyword)
What is the result of trying to compile this class?

```
public class Validator {
    private boolean valid;

    public static void validate() {
        this.valid = true;
    }
}
```

Only one correct choice.

- 0) Compilation is successful.
  WRONG - The code contains a compilation error.
- 1) Compilation fails because ‘this‘ cannot be used in a static context.
  RIGHT - The method `validate()` is declared `static`. Static methods belong to the class, not to a specific instance. The `this` keyword is a reference to the current instance. Therefore, `this` cannot be used inside a static method. This results in a compilation error.
- 2) Compilation fails because the ‘valid‘ field cannot be accessed from ‘validate()‘.
  WRONG - This is also true (a static method can’t access an instance field directly), but choice 1 is more specific to the syntax used (`this.valid`), which makes it the better answer.
- 3) Compilation fails because a static method cannot have a ‘void‘ return type.
  WRONG - Static methods can have a `void` return type. For example, the `main` method does.

(69) (questionId: 103013, topic: Throwing and Creating Exceptions)
Which definition creates a custom unchecked exception?
Only one correct choice.

- 0) ‘public class MyUnchecked extends Exception ‘
  WRONG - Extending `Exception` creates a *checked* exception.
- 1) ‘public class MyUnchecked extends Throwable ‘
  WRONG - Extending `Throwable` is generally discouraged for application exceptions. One should extend `Exception` or `RuntimeException`.
- 2) ‘public class MyUnchecked extends Error ‘
  WRONG - Extending `Error` is for critical, unrecoverable system errors and should not be used for application logic.
- 3) ‘public class MyUnchecked extends RuntimeException ‘
  CORRECT - By definition, any class that extends `RuntimeException` (or one of its subclasses) is an unchecked exception. The compiler will not require it to be caught or declared.

(70) (questionId: 100407, topic: Primitive Data Types and Literals)
Which of the following lines of code will fail to compile?
Only one correct choice.

- 0) ‘int i = 0b101;‘
  WRONG - This code compiles. ‘0b‘ denotes a binary literal (introduced in Java 7). ‘0b101‘ is the binary representation of the decimal number 5.
- 1)

  ‘`double d = 3.14_15;`‘

  WRONG - This code compiles. Underscores can be placed between digits in numeric literals (since Java 7) to improve readability. This is equivalent to ‘double d = 3.1415;‘.
- 2) ‘float f = 1.2e3f;‘
  WRONG - This code compiles. ‘1.2e3f‘ is a valid ‘float‘ literal using scientific notation, representing $1.2 imes 10^3$, or ‘1200.0f‘.
- 3)

  ‘`long l = 100_L;`‘

  RIGHT - This code fails to compile. According to the rules for using underscores in numeric literals, an underscore cannot be placed immediately before a type suffix (‘L‘, ‘F‘, ‘D‘). It must be placed between digits. ‘long l = 100L;‘ would be valid.

(71) (questionId: 102809, topic: Exception Hierarchy and Types)
What is the output of the following code snippet?

```
public class Test {
    public static void main(String[] args) {
        try {
```

```
            Object[] arr = new String[2];
            arr[0] = "Hello";
            arr[1] = 100; // Line 5
            System.out.println("End of try");
        } catch (Exception e) {
            System.out.println(e.getClass().getSimpleName());
        }
    }
}
```

Only one correct choice.

- 0) 'NumberFormatException'
  WRONG - This exception is related to parsing strings into numbers, which is not what's happening here.
- 1) 'IllegalArgumentException'
  WRONG - This exception is typically for passing invalid arguments to methods, but a more specific exception applies here.
- 2) 'ArrayStoreException'
  RIGHT - The variable `arr` is of type `Object[]`, but the actual object it refers to is a `String[]`. At compile time, assigning an `Integer` to an `Object[]` seems fine. However, at runtime, the JVM knows the array's true type is `String[]` and that it cannot store an `Integer`. This specific type mismatch when storing an element in an array throws an `ArrayStoreException`.
- 3) 'ClassCastException'
  WRONG - `ClassCastException` is for invalid type casts (e.g., `(String) new Integer(5)`), not for storing the wrong type in an array.

(72) (questionId: 100714, topic: Variable Scope and Lifetime)
What will be printed by the following code?

```
public class Test {
    public static void main(String[] args) {
        int x;
        // line 1
        if (args.length > 0) {
            x = 5;
        }
        // line 2
        // System.out.println(x);
    }
}
```

Only one correct choice.

- 0) If line 2 is uncommented, the code will compile and print 0 if no arguments are passed.
  WRONG - The code will not compile, so it cannot run.
- 1) If line 2 is uncommented, the code will compile and print 5 if at least one

argument is passed.
WRONG - The code will not compile, so it cannot run.

- 2) If line 2 is uncommented, the code will fail to compile regardless of arguments passed.
  CORRECT - This is a classic 'definite assignment' problem. The local variable `x` is only initialized inside the `if` block. The compiler cannot guarantee that the `if` condition (`args.length > 0`) will be true at runtime. Since there is a possible execution path where `x` is never assigned a value, the compiler will report an error on the line `System.out.println(x);` stating that 'variable x might not have been initialized'.
- 3) If line 1 is changed to `int x=0;`, the uncommented code will compile.
  WRONG - While this statement is true (initializing `x=0;` would make the code compile), it does not correctly describe the outcome of the original code as asked by the question.

(73) (questionId: 102412, topic: One-Dimensional and Multi-Dimensional Arrays)
Which of these array declarations and initializations is NOT legal?
Only one correct choice.

- 0) `int[] arr = new int[2];`
  LEGAL - This correctly declares and instantiates an array of size 2, with elements defaulting to 0.
- 1) `int[] arr = new int[] {1, 2};`
  LEGAL - This correctly uses anonymous array syntax to declare, instantiate, and initialize an array.
- 2) `int[] arr = {1, 2};`
  LEGAL - This correctly uses the array initializer shortcut, which is valid only on the line of declaration.
- 3) `int[] arr = new int[2] {1, 2};`
  NOT LEGAL - This is a syntax error. When using the ‘new‘ keyword with an initializer block ‘‘, you cannot specify the array size in the brackets ‘[]‘. The compiler infers the size from the initializer. This redundancy causes a compilation failure.

(74) (questionId: 101311, topic: String Immutability and Operations)
What is the output of this code snippet?

```
String s1 = "1";
String s2 = s1.concat("2");
s2.concat("3");
System.out.println(s2);
```

Only one correct choice.

- 0) ‘1‘
  WRONG - The variable `s2` is assigned the result of the first concatenation.
- 1) ‘12‘
  CORRECT - 1. ‘s1‘ is ‘”1”‘. 2. `s2` is assigned the result of `s1.concat("2")`, so

`s2` becomes “"12"‘. 3. The line `s2.concat("3")` creates a new string “"123"‘, but since the result is not assigned to any variable, it is discarded. 4. The variable `s2` remains unchanged from step 2, so “"12"‘ is printed.

- 2) ‘123‘
  WRONG - This would be the output if the last line was `s2 = s2.concat("3");`.
- 3) The code does not compile.
  WRONG - The code is syntactically correct.

(75) (questionId: 102112, topic: Polymorphism and Type Casting)
What is the outcome of compiling and running this code?

```
interface Flyable {
    void fly();
}
class Bird implements Flyable {
    public void fly() { System.out.println("Bird flying"); }
}
class Plane implements Flyable {
    public void fly() { System.out.println("Plane flying"); }
}
public class Test {
    public static void main(String[] args) {
        Flyable flyer = new Plane();
        flyer.fly();
    }
}
```

Only one correct choice.

- 0) Bird flying
  WRONG - The object created is an instance of `Plane`, not `Bird`, so the `Plane`'s implementation of `fly()` is called.
- 1) Plane flying
  RIGHT - This demonstrates polymorphism using an interface. A reference of an interface type (`Flyable`) can hold an object of any class that implements that interface (`Plane`). At runtime, the JVM invokes the method implementation from the actual object's class, which is `Plane`.
- 2) A compile-time error occurs.
  WRONG - The code is valid. It's a standard use of interfaces and polymorphism.
- 3) A ‘ClassCastException‘ is thrown.
  WRONG - No casting is performed in this code, so a `ClassCastException` cannot occur.

(76) (questionId: 100611, topic: Wrapper Classes and Autoboxing/Unboxing)
Consider the following code. Which statement is true?

```
public class Test {
```

```
    public static void main(String[] args) {
        Long l1 = 10L;
        long l2 = 10;
        Integer i1 = 10;

        // Statement goes here
    }
}
```

Only one correct choice.

- 0) `if (l1 == i1)` will not compile.
  WRONG - This code will compile. When comparing wrapper types of different kinds (like `Long` and `Integer`) using `==`, both are unboxed to their primitive types. The comparison becomes `10L == 10`, which is valid.
- 1) `if (l1.equals(i1))` will return `true`.
  WRONG - The `.equals()` method for wrapper types first checks if the other object is of the same type. Since `i1` is an `Integer` and not a `Long`, `l1.equals(i1)` will return `false` without even checking the values.
- 2) `if (l1.equals(l2))` will not compile.
  WRONG - This code will compile. The primitive `long l2` will be autoboxed to a `Long` object to be passed as an argument to the `equals(Object obj)` method.
- 3) `if (l1 == l2)` will evaluate to `true`.
  CORRECT - When a comparison with `==` is made between a wrapper type and a primitive type, the wrapper is **unboxed**. The statement becomes a comparison between two primitives: `10L == 10L`, which evaluates to `true`.

(77) (questionId: 101520, topic: Classes and Objects Fundamentals)
What are the characteristics of a class that correctly follows the principle of encapsulation? (Choose all that apply)
Multiple correct choices.

- 0) All instance variables are declared ‘public‘ for easy access.
  WRONG - Making instance variables `public` is the opposite of good encapsulation, as it allows uncontrolled external access to the object's internal state.
- 1) Instance variables are declared ‘private‘.
  CORRECT - A key principle of encapsulation is data hiding. Instance variables are typically declared `private` to prevent direct access from outside the class.
- 2) Public accessor methods (getters) and mutator methods (setters) are provided to access and modify the private instance variables.
  CORRECT - To allow controlled access to the private data, `public` methods (getters and setters) are provided. This allows the class to enforce validation or logic when its state is read or modified.
- 3) The class cannot be instantiated.
  WRONG - Encapsulation does not prevent instantiation; it is a principle for

designing robust and maintainable objects that are meant to be instantiated.

- 4) The internal state of the object is hidden from the outside.
  CORRECT - This is the primary goal of encapsulation: to protect an object's internal data from arbitrary external modification, ensuring the object remains in a valid state.
- 5) All methods are declared 'static'.
  WRONG - Encapsulation deals with the state of individual objects (instances), so it primarily involves instance methods, not `static` methods which belong to the class.

(78) (questionId: 101010, topic: Looping Constructs (for, while, do-while))
What is printed by this nested loop?

```
public class NestedLoop {
    public static void main(String[] args) {
        for (int i = 0; i < 2; i++) {
            for (int j = 2; j > 0; j--) {
                if (j == 1)
                    break;
                System.out.print(j);
            }
            System.out.print(i);
        }
    }
}
```

Only one correct choice.

- 0) 2021
  CORRECT (Note: The provided answer key is incorrect). Let's trace the execution. Outer loop 'i=0': The inner loop starts with 'j=2'. It prints '2', then 'j' becomes 1. When 'j=1', the 'break' is hit, terminating the *inner* loop. Then, the outer loop prints 'i' (which is 0). Output is now '20'. Outer loop 'i=1': The inner loop runs again, printing '2' and breaking. The outer loop then prints 'i' (which is 1). The final output is '2021'.
- 1) 202
  WRONG - This output '202' is impossible to achieve with the given code. For the code to produce '202', the final 'System.out.print(i)' for 'i=1' would have to be skipped, and there is no logic in the code to cause this. This is likely an error in the question's provided answer key.
- 2) 21021
  WRONG - The 'break' statement prevents the inner loop from printing '1'.
- 3) 210
  WRONG - This would be the output if the outer loop only ran once, which is incorrect as the condition 'i ¡ 2' allows it to run for 'i=0' and 'i=1'.

(79) (questionId: 103421, topic: Static Imports)
What is the result of attempting to compile and run the following code?

```
// File: pkg/A.java
package pkg;
public class A {
    public static void run() { System.out.println("A"); }
}

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

// File: Main.java
import static pkg.A.*;
import static pkg.B.*;

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

Only one correct choice.

- 0) It prints ‘A‘.
  WRONG - The code does not compile.
- 1) It prints ‘B‘.
  WRONG - The code does not compile.
- 2) It fails to compile due to an ambiguous call.
  CORRECT - Both wildcard static imports, ‘import static pkg.A.*;‘ and ‘import static pkg.B.*;‘, bring a method named ‘run()‘ into the current scope. When ‘run()‘ is called, the compiler finds two potential matching methods: ‘A.run()‘ and ‘B.run()‘. Since it cannot decide which one to use, it raises an ’ambiguous method call’ compilation error.
- 3) It throws an exception at runtime.
  WRONG - The ambiguity is resolved at compile time, resulting in a compilation failure.

(80) (questionId: 100115, topic: Main Method and Command Line Arguments)
Consider this code:

```
public class Logic {
    public static void main(String... logic) {
        System.out.println(logic[1]);
    }
}
```

What is the result of running ‘java Logic true false‘?
Only one correct choice.

- 0) true
  WRONG - The argument "true" is passed as the first element, so it is at index 0 (‘logic[0]‘).
- 1) false
  CORRECT - The command-line arguments are "true" and "false". They are stored in the ‘logic‘ array as ‘"true", "false"‘. The code prints the element at index 1, which is the string "false".
- 2) Compilation fails due to the parameter name ‘logic‘.
  WRONG - ‘logic‘ is not a reserved keyword in Java and is a perfectly valid name for a variable or parameter.
- 3) An ‘ArrayIndexOutOfBoundsException‘ is thrown.
  WRONG - The array has a length of 2, so indices 0 and 1 are both valid. No exception will be thrown.

(81) (questionId: 100210, topic: Packages, Classpath, and JARs)
Given a source file with these two imports:

```
import java.util.Date;
import java.sql.Date;
```

What is the result?
Only one correct choice.

- 0) A compilation error occurs due to the ambiguous ‘Date‘ class.
  CORRECT - You cannot import two types with the same simple name. If you do, the compiler doesn't know which `Date` to use when you refer to it in your code, leading to an 'ambiguous reference' compilation error. To resolve this, you must use the fully qualified name (e.g., `java.util.Date`) for at least one of the types in your code.
- 1) ‘java.util.Date‘ takes precedence.
  WRONG - No import takes precedence. The conflict itself causes the compilation to fail.
- 2) ‘java.sql.Date‘ takes precedence.
  WRONG - No import takes precedence. The conflict itself causes the compilation to fail.
- 3) The code compiles, but a runtime error will occur if ‘Date‘ is used.
  WRONG - This is a compile-time error, not a runtime error. The compiler detects the ambiguity before the program can be run.

(82) (questionId: 102615, topic: Generics)
What happens when you try to use ‘instanceof‘ with a generic type?

```
public <T> void check(Object obj) {
    if (obj instanceof T) { // Line 2
        System.out.println("It's a T!");
    }
}
```

Only one correct choice.

- 0) The code works as expected.
  WRONG - It fails to compile.
- 1) The code compiles, but throws an exception at runtime.
  WRONG - It is a compile-time error, not a runtime exception.
- 2) The code fails to compile at Line 2.
  CORRECT - This is a compilation error because of type erasure. At runtime, the type ‘T‘ is erased and replaced by its bound (in this case, ‘Object‘). The ‘instanceof‘ operator needs to check against a specific, reifiable type at runtime, which is not possible with a generic type parameter ‘T‘.
- 3) The code compiles only if ‘T‘ is a final class.
  WRONG - The rule applies regardless of whether the type is final.

(83) (questionId: 102311, topic: The 'final' Keyword)
Which statement about the following code is true?

```
final class Algorithm {
    public final void perform() {
        // ...
    }
}
```

Only one correct choice.

- 0) The ‘final‘ keyword on the ‘perform‘ method is required for the code to compile.
  WRONG - The ‘final‘ keyword on the method is not required. The class would compile fine without it.
- 1) The ‘final‘ keyword on the ‘perform‘ method is redundant.
  RIGHT - A ‘final‘ class cannot be subclassed. By definition, this means none of its methods can ever be overridden. Therefore, declaring a method inside a ‘final‘ class as ‘final‘ is legal but redundant, as the method is already implicitly final in its behavior.
- 2) The ‘final‘ keyword on the class ‘Algorithm‘ is redundant.
  WRONG - The ‘final‘ keyword on the class has a distinct and important meaning: it prevents inheritance. It is not redundant.
- 3) This code will not compile.
  WRONG - The code is syntactically correct and compiles.

(84) (questionId: 103115, topic: Try-with-Resources)
What is the output of the code?

```
class Resource implements AutoCloseable {
    public void close() { System.out.print("Close"); }
}

public class TryFinally {
```

```
    public static void main(String[] args) {
        try (Resource r = new Resource()) {
            System.out.print("Try ");
        } finally {
            System.out.print("Finally");
        }
    }
}
```

Only one correct choice.

- 0) 'Try FinallyClose'
  WRONG - The resource is closed before the `finally` block is executed.
- 1) 'Try CloseFinally'
  CORRECT - The execution order is: 1. Resource `r` is created. 2. The `try` block executes, printing 'Try '. 3. The `try` block completes. 4. The resource's `close()` method is automatically called, printing 'Close'. 5. The `finally` block executes, printing 'Finally'. The final output is 'Try CloseFinally'.
- 2) 'FinallyTry Close'
  WRONG - The `try` block executes before the `finally` block.
- 3) The code does not compile.
  WRONG - The code is syntactically correct and compiles.

(85) (questionId: 101617, topic: Constructors and Initialization Blocks)
Which of the following statements about static initializer blocks are true? (Choose all that apply)
Multiple correct choices.

- 0) They are executed only once, when the class is first loaded by the JVM.
  CORRECT - Static initializers are tied to the class, not an instance, and run only once per class loader.
- 1) They can access 'this' to refer to the current object.
  WRONG - The `this` keyword refers to the current instance. Static blocks are executed in a class context where no instance exists.
- 2) A class can have multiple static initializer blocks.
  CORRECT - A class can have multiple static blocks, and they are executed in the order they appear in the source code.
- 3) They can access non-static instance variables of the class.
  WRONG - Static blocks cannot access instance (non-static) variables directly, as they are not associated with any specific object.
- 4) They are guaranteed to execute before any instance of the class is created.
  CORRECT - Class loading and static initialization must complete before the JVM can create an instance of that class.
- 5) They can throw checked exceptions without a 'throws' clause.
  WRONG - A static initializer block cannot throw a checked exception; this

would be a compilation error. It can throw an unchecked exception, which would cause an `ExceptionInInitializerError` at runtime.

(86) (questionId: 102018, topic: Inheritance and Method Overriding)
Which statement best describes the difference between method overriding and method hiding?
Only one correct choice.

- 0) Overriding applies to instance methods, while hiding applies to static methods. Method resolution for overriding is at runtime; for hiding, it's at compile-time.
  CORRECT - This is the precise definition. Overriding applies to instance methods and is a runtime concept (polymorphism). Hiding applies to static methods (and fields), and the method called is determined at compile time based on the reference type.
- 1) Overriding applies to static methods, while hiding applies to instance methods. Method resolution for overriding is at compile-time; for hiding, it's at runtime.
  WRONG - This inverts the concepts. Overriding is for instance methods.
- 2) Overriding involves changing the method signature, while hiding keeps it the same.
  WRONG - This describes overloading, not overriding or hiding. Both overriding and hiding require the method signature to be the same.
- 3) There is no difference; they are two terms for the same concept.
  WRONG - They are distinct concepts in Java with different binding rules and behaviors.

(87) (questionId: 101914, topic: Encapsulation and Access Modifiers)
What is the output of the following code?

```
class Parent {
    public String name = "Parent";
    void printName() { System.out.println(name); }
}

class Child extends Parent {
    public String name = "Child";
    void printName() { System.out.println(name); }
}

public class Test {
    public static void main(String[] args) {
        Parent p = new Child();
        System.out.println(p.name);
        p.printName();
    }
}
```

Only one correct choice.

- 0) Parent
  CORRECT - This question tests a crucial distinction. Variable access in Java is resolved at compile time based on the reference type, while method calls are resolved at runtime based on the actual object type. 1) `p.name`: The reference `p` is of type `Parent`. Therefore, this expression accesses the `name` field defined in the `Parent` class, printing 'Parent'. Fields do not exhibit polymorphic behavior. 2) `p.printName()`: This is a method call. At runtime, the JVM sees that `p` points to a `Child` object. Due to polymorphism (dynamic method dispatch), the overridden `printName()` method from the `Child` class is executed. This method prints the `name` field from the `Child` class's scope, which is 'Child'.
- 1) Child
  WRONG - This would be the output if variables were polymorphic like methods, but they are not.
- 2) Parent
  WRONG - This would be the output if the method call was not polymorphic.
- 3) Child
  WRONG - This reverses the correct results for variable access and method invocation.
- 4) Compilation fails.
  WRONG - The code is valid, although it demonstrates a potentially confusing practice (field hiding).

(88) (questionId: 100008, topic: Java Environment and Fundamentals)
You have a file named `Test.java`:

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

What happens when you try to compile this file with `javac Test.java`?
Only one correct choice.

- 0) It compiles successfully, creating `test.class`.
  WRONG - Compilation will fail. The generated class file would be named after the class (`test.class`), but the compiler won't even get that far.
- 1) It compiles successfully, creating `Test.class`.
  WRONG - Compilation will fail, so no `.class` file will be created.
- 2) It fails to compile because the class name `test` does not match the filename `Test.java`.
  CORRECT - A fundamental rule in Java is that if a class is declared `public`, the source file name must exactly match the class name, including case. Here, the filename is `Test.java` but the public class is named `test`, which is a mismatch. The compiler will report an error.

- 3) It fails to compile because of a syntax error in the main method.
  WRONG - The main method signature is syntactically correct. The error is the filename-classname mismatch, which is caught first.

(89) (questionId: 102515, topic: ArrayList and Basic Collections)
What is the output?

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

public class Test {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");
        list.add("C");
        System.out.println(list.remove("B") + " " + list.size());
    }
}
```

Only one correct choice.

- 0) B 2
  WRONG - The ‘remove(Object o)‘ method returns a ‘boolean‘, not the element that was removed.
- 1) true 2
  CORRECT - The ‘remove(Object o)‘ method returns ‘true‘ if an element was removed as a result of the call, and ‘false‘ otherwise. Since ”B” is in the list, it is removed, and the method returns ‘true‘. After removal, the list’s size is 2. The output is ‘true 2‘.
- 2) B 3
  WRONG - The method does not return the removed element, and the size is 2 after removal.
- 3) true 3
  WRONG - The size is 2 after removal.

(90) (questionId: 101319, topic: String Immutability and Operations)
Given ‘String str = ”Java SE 8”;‘, which expressions will evaluate to ‘true‘? (Choose all that apply)
Multiple correct choices.

- 0) ‘str.startsWith(”Java”)‘
  CORRECT - The string ‘”Java SE 8”‘ does start with the prefix ‘”Java”‘.
- 1) ‘str.endsWith(” 8”)‘
  CORRECT - The string does end with the suffix ‘” 8”‘ (note the leading space).
- 2) ‘str.contains(”SE”)‘
  CORRECT - The string does contain the character sequence ‘”SE”‘.

- 3) ‘str.equalsIgnoreCase(”java se 8”)‘
  CORRECT - The `equalsIgnoreCase()` method compares the content of two strings while ignoring differences in case, so this evaluates to ‘true‘.

(91) (questionId: 101709, topic: Static Members and ’this’ Keyword)
What is the result of compiling and running the following code?

```
public class StaticAccess {
    static String GREETING = "Hello";

    public static void main(String[] args) {
        StaticAccess sa = null;
        System.out.println(sa.GREETING);
    }
}
```

Only one correct choice.

- 0) Hello
  RIGHT - This is a classic exam trick. When a static member is accessed using an object reference, the compiler resolves the call based on the reference’s *declared type*, not the object itself. The compiler effectively rewrites `sa.GREETING` to `StaticAccess.GREETING`. Because the object instance is never actually used for the access, it doesn’t matter that the reference is `null`. No `NullPointerException` is thrown.
- 1) null
  WRONG - The static variable holds the value ”Hello”.
- 2) A ‘NullPointerException‘ is thrown at runtime.
  WRONG - No `NullPointerException` is thrown because the instance itself is not dereferenced to access a static member.
- 3) The code fails to compile.
  WRONG - Although accessing a static member through an instance reference is discouraged, it is perfectly legal syntax.

(92) (questionId: 100311, topic: Java Coding Conventions and Javadoc)
Which Javadoc tag is used to indicate that a method or class is outdated and may be removed in a future version?
Only one correct choice.

- 0) ‘@obsolete‘
  WRONG - `@obsolete` is not a standard Javadoc tag.
- 1) ‘@deprecated‘
  CORRECT - The `@deprecated` Javadoc tag is used to indicate that an API element (class, method, field) is no longer recommended for use and may be removed in a future version. This tag is often used in conjunction with the `@Deprecated` annotation, which causes the compiler to issue a warning if the deprecated element is used.
- 2) ‘@version‘

WRONG - The `@version` tag is used to specify the version of the software that the code belongs to.

- 3) '@legacy'
  WRONG - `@legacy` is not a standard Javadoc tag.

(93) (questionId: 101213, topic: Enums)
Which of the following code snippets will result in a compilation error?
Only one correct choice.

- 0)

  ```
  public enum A { X, Y; public void m() {} }
  ```

  VALID - An enum can have methods. A semicolon is required after the last enum constant if the enum body contains other members.

- 1)

  ```
  public enum B implements java.io.Serializable { X, Y; }
  ```

  VALID - All enums implicitly implement `Serializable`. Stating it explicitly is redundant but perfectly legal.

- 2)

  ```
  public enum C { X, Y; private C() {} }
  ```

  VALID - Enum constructors are implicitly private. Making it explicitly `private` is allowed.

- 3)

  ```
  public enum D extends java.lang.Enum { X, Y; }
  ```

  COMPILATION ERROR - This is the key restriction on enums. The compiler automatically makes every enum extend `java.lang.Enum`. You are not allowed to specify this explicitly, nor can you extend any other class.

(94) (questionId: 100518, topic: Type Conversion and Casting)
Which of the following lines of code require an explicit cast to compile successfully? (Choose all that apply)
Multiple correct choices.

- 0) 'long l = 10;'
  WRONG - This is a widening conversion from an 'int' literal to a 'long', so no cast is needed.

- 1) 'byte b = 10;'
  WRONG - This is an implicit narrowing of an 'int' literal to a 'byte'. Since '10' is within the 'byte' range, no cast is needed.

- 2) 'float f = 10.0;'
  CORRECT - The literal '10.0' is a 'double'. Assigning a 'double' to a 'float' is a narrowing conversion and requires an explicit cast like '(float)10.0' or using a float literal '10.0f'.

- 3) ‘int i = 10L;‘
  CORRECT - The literal ‘10L‘ is a ‘long‘. Assigning a ‘long‘ to an ‘int‘ is a narrowing conversion and requires an explicit cast like ‘(int)10L‘.

(95) (questionId: 102715, topic: Sorting and Searching Collections (Comparable, Comparator))
Which ‘Comparator‘ static methods can be used to create a ‘Comparator‘ instance? Multiple correct choices.

- 0) ‘Comparator.comparing(Function)‘
  CORRECT - `Comparator.comparing(Function)` is a static factory method introduced in Java 8 that takes a function extractor and returns a `Comparator` that compares by that extracted key.
- 1) ‘Comparator.naturalOrder()‘
  CORRECT - `Comparator.naturalOrder()` is a static factory method that returns a `Comparator` that uses the natural ordering of objects (i.e., calls their `compareTo` method).
- 2) ‘Comparator.reversed()‘
  WRONG - `reversed()` is a default instance method, not a static method. It is called on an existing `Comparator` instance to get a new comparator with the reverse ordering (e.g., `myComp.reversed()`).
- 3) ‘Comparator.thenComparing(Comparator)‘
  WRONG - `thenComparing()` is a default instance method, not a static method. It is used to chain comparators for secondary, tertiary, etc. sorting criteria (e.g., `comp1.thenComparing(comp2)`).

(96) (questionId: 100706, topic: Variable Scope and Lifetime)
What is the result of compiling and running this class?

```
public class ScopeTest {
    private int x = 10;

    public void process() {
        int x = 20;
        System.out.println(x);
    }

    public static void main(String[] args) {
        new ScopeTest().process();
    }
}
```

Only one correct choice.

- 0) 10
  WRONG - The instance variable is ’shadowed’ by the local variable.
- 1) 20
  CORRECT - This demonstrates variable shadowing. There are two variables named `x`: the instance variable (value 10) and the local variable inside the

`process` method (value 20). Within the `process` method, the local variable takes precedence. Therefore, `System.out.println(x)` refers to the local `x`, printing its value of 20.

- 2) Compilation fails due to a duplicate variable 'x'.
  WRONG - This is not a duplicate variable error because the variables are in different scopes (class scope vs. method scope). This is allowed in Java.
- 3) A runtime exception is thrown.
  WRONG - The code is valid and executes without any exceptions.

(97) (questionId: 102217, topic: Abstract Classes and Interfaces)
Which statements are true about 'default' methods in Java 8 interfaces? (Choose all that apply)
Multiple correct choices.

- 0) They must be marked with the 'default' keyword.
  CORRECT - The 'default' keyword is used to signify that an interface method provides a body (a default implementation).
- 1) They are implicitly 'public'.
  CORRECT - All methods in an interface, including 'default' methods, are implicitly 'public'. You cannot declare them as 'protected' or 'private' (until Java 9 private methods).
- 2) A class can implement two interfaces with the same default method signature without providing its own implementation.
  WRONG - This scenario is known as the 'diamond problem'. If a class implements two interfaces that provide a default method with the same signature, it results in a compile-time error. The class must override the method to explicitly resolve the ambiguity.
- 3) They cannot be 'static' or 'final'.
  WRONG - A 'default' method is an instance method, so it cannot be 'static'. It also cannot be 'final' because 'default' methods are designed to be overridable by implementing classes.

(98) (questionId: 101419, topic: StringBuilder and StringBuffer)
Which statements are true? (Choose all that apply)
Multiple correct choices.

- 0) 'String' objects are immutable.
  CORRECT - 'String' objects cannot be changed after creation.
- 1) 'StringBuilder' objects are mutable.
  CORRECT - 'StringBuilder' objects are designed to be changed (mutated).
- 2) 'StringBuffer' is thread-safe.
  CORRECT - 'StringBuffer' achieves thread safety by having its key methods synchronized.
- 3) Concatenating 'String' objects in a loop is generally less efficient than using 'StringBuilder'.
  CORRECT - Using the '+' operator in a loop can create many intermediate

'String' objects, which is inefficient. An explicit 'StringBuilder' is the preferred, more performant approach for building strings in loops.

(99) (questionId: 102914, topic: Try-Catch-Finally Blocks)
What is the output of the following code?

```
public class Test {
    public static void main(String[] args) {
        String s = "";
        try {
            s += "t";
            throw new Exception();
        } catch (Exception e) {
            s += "c";
        } finally {
            s += "f";
        }
        s += "a";
        System.out.println(s);
    }
}
```

Only one correct choice.

- 0) 'tfa'
  WRONG - The `catch` block is executed.
- 1) 'tcfa'
  RIGHT - This is a standard execution flow. 1) `try` block: 's' becomes "t". 2) An exception is thrown. 3) `catch` block: The exception is caught, 's' becomes "tc". 4) `finally` block: This always executes, 's' becomes "tcf". 5) The program continues after the construct, 's' becomes "tcfa". Finally, 'tcfa' is printed.
- 2) 'tca'
  WRONG - The `finally` block always executes after the `catch` block.
- 3) 'tcf'
  WRONG - The code after the `try-catch-finally` construct is executed because the exception was handled.

(100) (questionId: 102819, topic: Exception Hierarchy and Types)
What is the result of attempting to compile and run this class?

```
public class Test {
    public void go() throws java.sql.SQLException {
        System.out.println("Going");
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.go();
    }
```

```
}
```

Only one correct choice.

- 0) Compilation fails because ‘main‘ must declare ‘throws java.sql.SQLException‘.
  RIGHT - The method `go()` is declared with `throws java.sql.SQLException`. This is a checked exception. The `main` method calls `t.go()` but does not handle this checked exception with a `try-catch` block, nor does it declare it in its own `throws` clause. This is a violation of the ’handle or declare’ rule, resulting in a compilation error.
- 1) Compilation fails for a different reason.
  WRONG - The reason for compilation failure is specifically the unhandled checked exception.
- 2) It compiles and prints ‘Going‘.
  WRONG - The code does not compile.
- 3) It compiles but throws ‘java.sql.SQLException‘ at runtime.
  WRONG - The code does not compile, so it cannot run to throw an exception.