1Z0-808 Exam Topic Reviewer

TopicId: 1010
Topic: Looping Constructs (for, while, do-while)

August 5, 2025

Topic 1010: Looping Constructs (for, while, do-while)

Thinking Like the Compiler: Controlling Repetition

Loops are the workhorses of programming, designed to execute a block of code repeatedly. The 1Z0-808 exam won't just ask you to write a simple loop. It will test your understanding of their precise execution order, termination conditions, variable scope, and the subtle differences between each type. Questions often involve infinite loops or loops with unusual syntax—code a developer might not write, but which is perfectly valid for the compiler.

The `while` Loop

This is the simplest form of loop. It repeats a block of code as long as a condition is true. **Syntax:** `while (booleanExpression) { ... }`

- **Execution Flow:** The `booleanExpression` is evaluated *before* each iteration. If it's false to begin with, the loop body is never executed.
- **Exam Trap (Infinite Loop):** If the condition never becomes false, you have an infinite loop. The exam will test this by presenting loops where the update logic is flawed or missing.
- **Exam Trap (Unreachable Code):** A condition that is a compile-time constant `false` will cause a compile error because the loop body is unreachable. E.g., `while(false) {...}`

The `do-while` Loop

This is a variation of the `while` loop where the condition is checked at the end.

Syntax: `do { ... } while (booleanExpression);`

- **Execution Flow:** The loop body executes once, *then* the `booleanExpression` is evaluated. This guarantees the body runs **at least one time**.
- **Exam Trap (The Semicolon):** A `do-while` loop must end with a semicolon. Forgetting it is a compile error, and a likely trick question.

```
// Example: while vs. do-while
int x = 10;
while (x < 10) { System.out.println("while loop"); } // Prints nothing

do {
    System.out.println("do-while loop"); // Prints "do-while loop" once
} while (x < 10);
```

The Basic `for` Loop

The most structured loop, with three distinct parts in its declaration.

Syntax: `for (initialization; condition; update) { ... }`

- **Initialization:** Executes **once** when the loop starts. You can declare variables here; their scope is limited to the loop itself.
- **Condition:** A boolean expression evaluated **before** each iteration. The loop continues as long as this is true.
- **Update:** Executed **at the end** of each iteration, after the body runs.

Exam Traps for Basic `for` Loops

- **Optional Parts:** All three sections are optional. `for( ; ; ) {...}` is a valid, infinite loop.
- **Multiple Statements:** The initialization and update sections can contain multiple statements, separated by commas.

```
for (int i=0, j=5; i < j; i++, j--) {
    System.out.println(i + " " + j);
}
```

- **Variable Scope:** A variable declared in the initialization section cannot be accessed after the loop.

```
for (int k = 0; k < 3; k++) { ... }
// System.out.println(k); // COMPILE ERROR: cannot find symbol 'k'
```

The Enhanced `for` Loop (For-Each)

This provides a simpler syntax for iterating through all elements of an array or a collection.

Syntax: `for (Type var : iterableCollectionOrArray) { ... }`

- **Exam Trap (Variable is a Copy):** For each iteration, the loop variable `var` gets a copy of the element. If you reassign the loop variable itself, it has no effect on the original array or collection. However, you *can* call methods on an object reference to change its state.

```
StringBuilder[] names = { new StringBuilder("A"), new StringBuilder("B") };
for (StringBuilder name : names) {
    name.append("X"); // This WORKS, modifies the object in the array
    name = new StringBuilder(); // This has NO effect on the array itself
}
// After loop, names array contains ["AX", "BX"]
```