1Z0-808 Exam Topic Reviewer

TopicId: 1035
Topic: Method Design and Variable Arguments

August 5, 2025

Introduction: Crafting Your Methods

Methods are the verbs of our Java programs—they perform the actions. For the 1Z0-808 exam, you need to be precise about how they are constructed. This includes everything from who can access them (access modifiers) to how they handle a varying number of inputs (varargs). Let's master the blueprint of a Java method.

1 Method Signature Components

A method's declaration has several parts, but for the purposes of uniqueness and overloading, the **signature** is defined by the **method name** and the **parameter list** (the number, type, and order of parameters). **Exam Trap:** The return type, access modifier, and 'throws' clause are **not** part of the signature for overloading. You cannot have two methods that differ only by their return type.

```
public int calculate();
public String calculate(); // COMPILER ERROR: method calculate() is already defi
```

2 Access Modifiers: Controlling Visibility

You must know the four access levels cold. From most to least restrictive:

- `private`: Accessible **only** within the same class.
- **default** (Package-Private): No keyword is used. Accessible only by classes in the **same package**.
- `protected`: Accessible within the **same package**, and also by **subclasses** outside the package.
- `public`: Accessible from **anywhere**.

3 Variable Arguments (Varargs)

Varargs provide a way to create methods that can be called with a variable number of arguments (from zero to many). This feature reduces the need for creating multiple overloaded methods or forcing the caller to manually create an array.

3.1 Syntax and Behavior

The syntax uses three dots (an ellipsis) after the data type.

```
// The 'numbers' parameter is a varargs parameter.
public static void printNumbers(int... numbers) {
    System.out.println("Number of arguments: " + numbers.length);
    // Inside the method, 'numbers' is treated as an array: int[]
    for (int num : numbers) {
        System.out.print(num + " ");
    }
    System.out.println();
}
```

Calling a varargs method:

```
printNumbers();          // Called with zero arguments. length is 0.
printNumbers(10);        // Called with one argument.
printNumbers(1, 2, 3); // Called with three arguments.

// You can also pass an array explicitly
int[] data = {4, 5, 6};
printNumbers(data);
```

3.2 The Two Golden Rules of Varargs

The exam will absolutely test you on these rules. Memorize them.

(a) **A method can have at most ONE varargs parameter.**

```
// COMPILER ERROR: two varargs parameters
void invalidMethod(int... nums, String... names) { }
```

(b) **The varargs parameter must be the LAST parameter in the method signature.**

```
// COMPILER ERROR: varargs is not the last parameter
void invalidMethod(String... names, int count) { }

// VALID: varargs is the last parameter
void validMethod(int count, String... names) { }
```

3.3 Varargs and Overloading

When you have overloaded methods, one with a varargs parameter and one with a more specific parameter list, the compiler will always choose the most specific match available. The varargs method is the last resort.

```
public class OverloadTest {
    public static void fly(int numMiles) {
        System.out.println("int");
    }

    public static void fly(int... lengths) {
        System.out.println("varargs");
    }

    public static void main(String[] args) {
        fly(5);       // Prints "int". Exact match is preferred.
        fly(5, 10);   // Prints "varargs". No exact match, so varargs is used.
        fly(new int[]{2, 3}); // Prints "varargs". Explicit array is passed.
    }
}
```

Key Takeaways for the 1Z0-808 Exam

- A method signature is its name and parameter list. Return type is not part of it.
- Know the visibility rules: `private`, default, `protected`, `public`.
- A varargs parameter is declared with `...` and is treated as an **array** inside the method.
- The two varargs rules are non-negotiable: **only one** per method, and it **must be last**.
- In overloading, the compiler prioritizes exact matches over varargs matches.