1Z0-808 Exam Topic Reviewer

TopicId: 1002
Topic: Packages, Classpath, and JARs

August 5, 2025

Organizing Your Code: Beyond a Single File

Alright team, let's level up. So far, we've dealt with single Java files. In any real-world project, and certainly on the exam, you'll work with code organized into logical units. This lesson is about the nuts and bolts of that organization: packages for structure, the classpath for finding your code, and JAR files for deploying it. Mastering this is non-negotiable.

1 Packages: The Java Filing System

A package serves two primary purposes: organizing your classes into a manageable namespace and controlling access to them. Think of it like folders on your computer.

1.1 Declaration and Directory Structure

- **Declaration:** You declare a class's package with the `package` keyword. This **must** be the first non-comment statement in the file.

  ```
  // File: src/com/mycorp/utils/Calculator.java
  package com.mycorp.utils;

  public class Calculator { ... }
  ```

- **Directory Mapping (Crucial Exam Point):** The package name maps *directly* to a directory structure. The compiler (`javac`) and runtime (`java`) enforce this rule strictly. For the package `com.mycorp.utils`, your file system must look like this:

  ```
  src/
   com/
       mycorp/
           utils/
               Calculator.java
  ```

  A mismatch between the package name and the folder path will result in a compile-time or runtime error.

1.2 Compiling and Running Packaged Code

Your commands must now be aware of this structure.

- **Compiling:** You should run `javac` from the root directory of your source code (e.g., the `src` folder in the example above).

  ```
  // Assume we are inside the 'src' directory
  javac com/mycorp/utils/Calculator.java
  ```

  This creates `Calculator.class` inside `src/com/mycorp/utils/`.

- **A Better Way (Using -d):** To keep source and compiled files separate, use the `-d` flag to specify a destination directory.

  ```
  // From inside 'src', compile into a 'bin' directory
  // The 'bin' directory is at the same level as 'src'
  javac -d ../bin com/mycorp/utils/Calculator.java
  ```

  This will automatically create the `com/mycorp/utils` structure inside `bin` and place `Calculator.class` there.

- **Running:** To run the code, you use the **Fully Qualified Class Name (FQCN)**, which is `packageName.ClassName`. You also need to tell Java where to find the compiled files.

  ```
  // Assume we are in the project root (parent of 'src' and 'bin')
  // We must tell Java to look inside the 'bin' directory
  java -cp bin com.mycorp.utils.Calculator
  ```

2 The CLASSPATH: Telling Java Where to Look

The classpath is a list of directories and JAR files that the JVM searches for your compiled `.class` files. If a class isn't found on the classpath, you'll get a `ClassNotFoundException` or `NoClassDefFoundError`.

- **Default:** If you don't set it, the classpath defaults to the current directory (`.`).
- **Setting it:** The `-cp` (or `-classpath`) flag is the standard way to set it for both `javac` and `java`.
- **Syntax:** Paths are separated by `;` on Windows and `:` on Linux/macOS.

  ```
  // Look in the 'bin' directory AND in an external library 'libs/utils.ja
  java -cp "bin;libs/utils.jar" com.mycorp.Main  // Windows
  java -cp "bin:libs/utils.jar" com.mycorp.Main  // Linux/macOS
  ```

3 JAR Files: Bundling Your Application

A JAR (Java ARchive) file is essentially a ZIP file that bundles all your project's `.class` files, resources, and metadata into a single distributable unit.

- **Creating a JAR:** Use the `jar` tool from the JDK. The flags `c` (create) and `f` (file) are common.

  ```
  // Assume 'bin' contains our compiled classes
  // Create a file called 'app.jar' with the contents of 'bin'
  jar -cf app.jar -C bin .
  ```

The `-C bin .` part is a tricky but useful pattern: it means "change directory to `bin`, then grab everything (`.`)." This prevents the `bin` folder itself from being inside the JAR.

- **Making it Executable:** To run a JAR with `java -jar`, you need a **manifest file** that specifies the entry point.

  (a) Create a text file, e.g., `manifest.mf`, with this content (**the file MUST end with a newline character!**):

  ```
  Main-Class: com.mycorp.Main
  ```

  (b) Create the JAR using the `m` (manifest) flag.

  ```
  jar -cfm app.jar manifest.mf -C bin .
  ```

- **Running the JAR:** Now it's simple.

  ```
  java -jar app.jar
  ```

  When using `-jar`, the `-cp` flag is *ignored.* The classpath is defined inside the JAR's manifest if needed.

4 Key Takeaways for the 1Z0-808 Exam

- **Package vs. Path:** The package declaration `package a.b;` must correspond to the directory structure `a/b/`.
- **FQCN:** Run packaged classes with their full name, e.g., `java a.b.MyClass`.
- **Compiler/Runtime Flags:** Know `javac -d <outdir>`, `java -cp <path>`, and `java -jar <jarfile>`.
- **JARs:** An executable JAR requires a manifest with a `Main-Class` attribute. `java -jar` ignores the external classpath.