Advanced Java : Day-24:

Task 1: Build Lifecycle

Demonstrate the use of Maven lifecycle phases (clean, compile, test, package, install, deploy) by executing them on a sample project and documenting what happens in each phase.

Sample Maven Project

Let's assume a simple Maven project structure with the following files:

sample-project

```
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── com
    │           └── example
    │               └── Main.java
    └── test
        └── java
            └── com
                └── example
                    └── MainTest.java
```

`pom.xml` (Project Object Model)

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>sample-project</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- Dependencies can be specified here -->
    </dependencies>

    <build>
        <sourceDirectory>src/main/java</sourceDirectory>
        <testSourceDirectory>src/test/java</testSourceDirectory>
        <plugins>
            <!-- Maven Compiler Plugin -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>1.8</source>
```

```
            <target>1.8</target>
          </configuration>
        </plugin>
      </plugins>
    </build>
</project>
```

Maven Lifecycle Phases

1. **Clean Phase (`clean`)**

   ```
   bash
   Copy code
   mvn clean
   ```

   - **What happens**: Cleans the project by deleting the `target` directory, which contains compiled classes, generated JARs, and other artifacts.
   - **Output**: The `target` directory is deleted.
2. **Compile Phase (`compile`)**

   ```
   bash
   Copy code
   mvn compile
   ```

   - **What happens**: Compiles the main source code (`src/main/java`) and generates `.class` files in the `target/classes` directory.
   - **Output**: Java source code is compiled into bytecode.
3. **Test Phase (`test`)**

   ```
   bash
   Copy code
   mvn test
   ```

   - **What happens**: Executes unit tests found in `src/test/java` using a test framework like JUnit. Maven compiles test sources and runs the tests.
   - **Output**: Test results are displayed in the console. If tests pass, build proceeds; otherwise, it stops.
4. **Package Phase (`package`)**

   ```
   bash
   Copy code
   ```

```
mvn package
```

   - **What happens**: Packages the compiled code (both main and test classes) into a distributable format, typically a JAR (Java Archive) file located in the `target` directory.
   - **Output**: A JAR file (`sample-project-1.0-SNAPSHOT.jar`) is created in the `target` directory.

5. **Install Phase (`install`)**

```
bash
Copy code
mvn install
```

   - **What happens**: Installs the packaged artifact (JAR file) into the local Maven repository (`~/.m2/repository`). The artifact is now available to other projects locally.
   - **Output**: The JAR file is copied to the local Maven repository.

6. **Deploy Phase (`deploy`)**

```
bash
Copy code
mvn deploy
```

   - **What happens**: Copies the final package (e.g., JAR file) to a remote repository for sharing with other developers or projects.
   - **Output**: Typically used with a Maven repository manager like Nexus or Artifactory to deploy artifacts to a remote repository.

Summary

- **Lifecycle Phases**: Maven follows a sequence of phases (`clean`, `compile`, `test`, `package`, `install`, `deploy`) to manage the build process.
- **Execution**: Each phase performs specific tasks, progressing from cleaning the project to compiling, testing, packaging, and optionally deploying artifacts.
- **Customization**: Maven's flexibility allows customization of build processes through plugins and configurations in the `pom.xml`.

By running these Maven commands on your sample project, you can observe how Maven automates the build process, manages dependencies, and produces deployable artifacts efficiently. Adjustments and enhancements can be made to the `pom.xml` to suit specific project requirements and workflows.

Task 2:

Create test cases with assertEquals, assertTrue, and assertFalse to validate the correctness of a custom String utility class.

```
package wipro.com.assignment16;

public class StringUtil {

    public static boolean isEmpty(String str) {
        return str == null || str.trim().isEmpty();
    }

    public static String reverse(String str) {
        if (str == null) {
            return null;
        }
        return new StringBuilder(str).reverse().toString();
    }

    public static boolean isPalindrome(String str) {
        if (str == null) {
            return false;
        }
        String cleanStr = str.replaceAll("\\s+", "").toLowerCase();
        int length = cleanStr.length();
        for (int i = 0; i < length / 2; i++) {
            if (cleanStr.charAt(i) != cleanStr.charAt(length - i - 1)) {
                return false;
            }
        }
        return true;
    }
}
```

Task 3:

Research and present a comparison of different garbage collection algorithms (Serial, Parallel, CMS, G1, ZGC) in Java.

Garbage collection (GC) in Java is the automatic process of reclaiming memory occupied by objects that are no longer reachable or in use by the application. Different garbage collection algorithms have been developed over the years to improve performance, reduce pauses, and handle varying workloads efficiently. Here's a comparison of several prominent garbage collection algorithms used in Java:

1. Serial Garbage Collector

- **Type**: Single-threaded, stop-the-world (STW)
- **Use Case**: Suitable for small-scale applications or environments with limited resources.
- **Operation**: Uses a single thread to perform garbage collection, halting all application threads during the process.
- **Advantages**:
    - Simple and straightforward.
    - Minimal overhead due to single-threaded operation.
- **Disadvantages**:
    - Not suitable for multi-core systems as it cannot take advantage of parallel processing.
    - Longer pause times as garbage collection is performed in one thread.

2. Parallel Garbage Collector

- **Type**: Multi-threaded, stop-the-world (STW)
- **Use Case**: Designed for throughput, suitable for applications requiring high throughput and acceptable pause times.
- **Operation**: Utilizes multiple threads for garbage collection, stopping application threads during major GC cycles.
- **Advantages**:
    - Utilizes multiple cores, leading to reduced pause times compared to the Serial collector.
    - Suitable for server-side applications where throughput is critical.
- **Disadvantages**:
    - Longer pause times during full GC cycles compared to more advanced collectors like CMS or G1.

3. Concurrent Mark Sweep (CMS) Garbage Collector

- **Type**: Concurrent, low-latency, generational
- **Use Case**: Aimed at reducing pause times for applications sensitive to latency.
- **Operation**: Concurrently marks reachable objects while application threads continue running. Uses multiple threads for the initial mark and remark phases but pauses the application during the final sweep phase.
- **Advantages**:
    - Reduced pause times compared to stop-the-world collectors like Serial and Parallel.
    - Suitable for applications where minimizing pause times is critical, such as interactive applications.
- **Disadvantages**:
    - More complex to tune compared to simpler collectors.
    - May lead to fragmentation of the heap, potentially impacting performance.

4. G1 (Garbage-First) Garbage Collector

- **Type**: Region-based, concurrent, low-latency

- **Use Case**: Designed to handle large heaps with minimal pause times and improved throughput.
- **Operation**: Divides the heap into regions and performs garbage collection concurrently with the application. Adapts dynamically to meet application goals for pause times or throughput.
- **Advantages**:
    - Predictable pause times due to its incremental and concurrent approach.
    - Efficient for large heaps, minimizing full GC pauses.
- **Disadvantages**:
    - Increased CPU overhead compared to CMS in certain scenarios.
    - Requires careful tuning to achieve optimal performance.

5. ZGC (Z Garbage Collector)

- **Type**: Low-latency, scalable, generational
- **Use Case**: Designed for applications requiring very low pause times, typically less than 10ms.
- **Operation**: Uses colored pointers to achieve concurrent garbage collection. It operates concurrently with application threads and supports very large heaps (multi-terabyte) with minimal performance impact.
- **Advantages**:
    - Extremely low pause times, making it suitable for latency-sensitive applications.
    - Scalable to large heaps without sacrificing performance.
- **Disadvantages**:
    - Higher CPU overhead compared to other collectors, particularly on smaller heaps.
    - Still relatively new compared to other collectors, with ongoing improvements and optimizations.

Comparison Summary:

- **Pause Times**: Serial and Parallel collectors have longer pause times during full GC cycles compared to CMS, G1, and ZGC.
- **Throughput**: Parallel and G1 collectors excel in throughput scenarios, suitable for applications prioritizing overall application throughput.
- **Latency**: CMS, G1, and ZGC focus on reducing pause times, with ZGC achieving the lowest pause times at the expense of increased CPU overhead.
- **Scalability**: G1 and ZGC are designed for large heaps and can scale well with increasing heap sizes.
- **Complexity**: CMS and G1 require more tuning compared to Serial and Parallel collectors, while ZGC introduces new complexity due to its concurrent approach.

Choosing the right garbage collection algorithm depends on the specific requirements of your application, including throughput, latency sensitivity, heap size, and available hardware resources. Java provides flexibility in selecting the appropriate collector based on these factors, allowing developers to optimize performance and responsiveness accordingly.

3.5