**Brief All Java Versions**

**Java 9 Version Enhancements: -**

Java 9, released in September 2017, brought a host of new features and enhancements, with the most prominent being the **Java Platform Module System (JPMS)**. Below is a

**Brief overview of all major enhancements in Java 9:**

**1. Java Platform Module System (JPMS)**

* **JEP 261: The Module System**: Introduced the **Java Platform Module System**, a major new feature that modularized the Java Development Kit (JDK) itself, dividing it into a set of interoperable modules. This enables:
  + **Encapsulation**: Stronger access control by exporting only specific packages, limiting what other code can access.
  + **Explicit dependencies**: Modules explicitly declare dependencies on other modules using requires.
  + **module-info.java**: A new file that defines the module’s dependencies and which packages it exports.
  + **Custom JREs**: Developers can create smaller runtime environments with only the modules they need using the jlink tool.
  + **Compatibility**: Introduced **automatic modules** to allow legacy JARs to be used as modules without modification.

**Example of a module-info.java:**

module myapp {

requires java.logging;

exports com.myapp.api;

}

**2. jlink Tool (JEP 282)**

* A new tool that allows developers to create custom runtime images containing only the necessary modules. This is particularly useful for creating lightweight Java applications with reduced footprint, important for containerized environments and microservices.
* **Example**: Create a custom JRE:

jlink --module-path $JAVA\_HOME/jmods --add-modules java.base,java.logging --output myCustomJRE

**3. JShell (JEP 222)**

* The introduction of **JShell**, an interactive **REPL (Read-Eval-Print Loop)** tool for Java, which allows you to execute Java code snippets without writing a full-fledged class or method.
* Useful for quick testing, prototyping, and learning.
* **Example**:

jshell> System.out.println("Hello, Java 9!");

Hello, Java 9!

**4. Improvements to the Process API (JEP 102)**

* Enhanced the java.lang.Process API, making it easier to manage and control operating system processes. It includes methods to:
  + Retrieve process information, like process ID, arguments, user, and start time.
  + Monitor and control process trees.
* **Example**:

ProcessHandle currentProcess = ProcessHandle.current();

System.out.println("Process ID: " + currentProcess.pid());

**5. Enhanced Stream API (JEP 266)**

* New methods were added to the Stream API for greater flexibility:
  + takeWhile(): Takes elements from a stream as long as a condition holds true.
  + dropWhile(): Drops elements while the condition is true and takes the remaining.
  + iterate(): Allows creation of infinite streams with a condition to terminate.
  + **Example**:

List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8);

numbers.stream().takeWhile(n -> n < 5).forEach(System.out::println); // Prints: 1, 2, 3, 4

**6. Private Interface Methods (JEP 213)**

* Java 9 introduced the ability to define **private methods** in interfaces. These methods can be used to share code between default and static methods within the same interface, improving code reuse.
* **Example**:

interface MyInterface {

default void method1() {

common();

}

default void method2() {

common();

}

private void common() {

System.out.println("Common logic");

}

}

**7. Collection Factory Methods (JEP 269)**

* New factory methods for creating immutable instances of commonly used collections (List, Set, Map), simplifying the creation of small, unmodifiable collections.
* **Example**:

java

Copy code

List<String> list = List.of("one", "two", "three");

Set<String> set = Set.of("a", "b", "c");

Map<String, Integer> map = Map.of("key1", 1, "key2", 2);

**8. Multi-Release JAR Files (JEP 238)**

* Multi-release JARs allow developers to ship JAR files that can be used across different Java versions while having version-specific class implementations. This makes libraries backward-compatible while using new features for newer versions.
* **Example**:
  + The JAR contains META-INF/versions/<version>/ for version-specific classes.

**9. Compact Strings (JEP 254)**

* Improved the internal representation of strings in memory. Java 9 introduced **Compact Strings**, which automatically encodes strings as **byte arrays** using UTF-8 encoding if all characters can be represented as single-byte values. This saves memory in many applications, as most strings are ASCII-compatible.

**Example**

Let's consider an example to illustrate how memory usage is reduced:

**Before Compact Strings (Java 8 and earlier):**

String latin1String = "Hello"; // Stored as char[] = {'H', 'e', 'l', 'l', 'o'}

Each character in the string is stored using 2 bytes, even though the Latin-1 characters could be stored more efficiently in 1 byte.

**With Compact Strings (Java 9 and later):**

String latin1String = "Hello"; // Stored as byte[] = {72, 101, 108, 108, 111}

Here, each character is stored using only 1 byte, saving memory. However, for strings containing non-Latin characters, such as:

String nonLatinString = "你好"; // Stored as byte[] = {encoded in UTF-16}

This string will use 2 bytes per character due to the Unicode representation.

**10. Unified JVM Logging (JEP 158)**

* Introduced a unified logging framework for all components of the JVM, providing a flexible way to handle log messages. Logging can be controlled with command-line flags, such as:

java -Xlog:gc\*:file=gc.log

**11. HTTP/2 Client (JEP 110 - Incubator)**

* Java 9 introduced an early, incubating version of a new **HTTP/2 Client** API to replace the old HttpURLConnection class. The new API supports both **HTTP/1.1** and **HTTP/2**, making it more suitable for modern web development.
* **Example**:

HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()

.uri(URI.create("https://example.com"))

.build();

HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());

System.out.println(response.body());

**12. Enhanced Deprecation (JEP 277)**

* Improved the way deprecation works in Java, including two new annotations:
  + @Deprecated(forRemoval = true): Indicates that the feature will be removed in a future release.
  + @Deprecated(since = "version"): Specifies the version since which the feature has been deprecated.

**13. Improved Garbage Collection (JEP 291)**

* Java 9 introduced several improvements to garbage collection, including **G1 GC** (Garbage-First Garbage Collector) becoming the default garbage collector. The G1 GC is designed for low-latency applications and is more efficient for managing large heaps compared to the previous default, Parallel GC.

**14. Ahead-of-Time Compilation (JEP 295)**

* Introduced **AOT (Ahead-of-Time) Compilation**, which allows Java code to be compiled into native machine code before execution. This reduces the startup time for applications. However, this feature was experimental and aimed at applications that require faster startup times.

**15. Variable Handles (JEP 193)**

* **Variable Handles** provide a more flexible and modern alternative to sun.misc.Unsafe, offering a standard way to perform low-level atomic and non-atomic operations on variables, improving thread safety.

**16. Stack-Walking API (JEP 259)**

* Introduced a new **Stack-Walking API**, which allows developers to efficiently traverse and inspect stack frames in a lazy and flexible way. This is an improvement over the existing Throwable.getStackTrace() method, which can be resource-intensive.
* **Example**:

StackWalker walker = StackWalker.getInstance();

walker.forEach(frame -> System.out.println(frame.getClassName()));

**17. Project Jigsaw (JEP 220)**

* Java 9 modularized the JDK itself using the module system, with over 90 modules, including java.base, java.sql, java.logging, and others. This marked a significant change in how Java is organized, and the entire JDK was refactored into smaller components.

**Summary of Key Java 9 Enhancements:**

1. **Modularity (JPMS)**: Modularized JDK, introduced module system (module-info.java).
2. **JShell**: Interactive REPL tool.
3. **Process API Improvements**: Enhanced process management.
4. **Stream API Enhancements**: New methods like takeWhile, dropWhile, and iterate.
5. **Private Interface Methods**: Allows private methods in interfaces.
6. **Collection Factory Methods**: Simplified collection creation.
7. **Multi-Release JARs**: Support for version-specific JARs.
8. **Compact Strings**: Memory-efficient string representation.
9. **Unified JVM Logging**: New logging framework for JVM components.
10. **HTTP/2 Client (Incubating)**: Modern HTTP client API.

**Java 11 Version Enhancements: -**

Java 11 introduced several enhancements and new features aimed at improving the language, the core libraries, and JVM performance. Some of the notable features and enhancements in **Java 11** include:

**1. New String Methods**

* **String::repeat**: Repeats the string a specified number of times.

String str = "Hello";

System.out.println(str.repeat(3)); // Output: HelloHelloHello

* **String::isBlank**: Checks if the string is empty or contains only whitespace characters.

System.out.println(" ".isBlank()); // Output: true

* **String::strip**: Removes leading and trailing whitespace (similar to trim() but more Unicode-compliant).

String str = " Hello ";

System.out.println(str.strip()); // Output: "Hello"

* **String::lines**: Converts a string into a stream of lines.

String str = "Hello\nWorld";

str.lines().forEach(System.out::println);

// Output:

// Hello

// World

**2. Local-Variable Syntax for Lambda Parameters (var)**

Java 11 allows the use of the var keyword for the lambda parameters to infer the type.

var list = List.of("a", "b", "c");

list.stream().map((var s) -> s.toUpperCase()).forEach(System.out::println);

**3. HTTP Client API (Standard)**

The new java.net.http package provides a more modern and efficient HTTP Client API for HTTP/2 and WebSocket communication. This was introduced as part of Java 9 but became standard in Java 11.

java

Copy code

HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()

.uri(URI.create("https://example.com"))

.build();

HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());

System.out.println(response.body());

**4. Run Single-File Programs Without Compilation**

You can now run Java source files directly using the java command without explicit compilation.

java HelloWorld.java

**5. Optional API Enhancements**

* **Optional::isEmpty**: This provides the opposite of isPresent(), simplifying conditions.

Optional<String> opt = Optional.empty();

System.out.println(opt.isEmpty()); // Output: true

**6. File Methods (New Methods in Files Class)**

* **Files::writeString** and **Files::readString**: Simplifies writing and reading String content from files.

java

Copy code

Path path = Files.writeString(Files.createTempFile("demo", ".txt"), "Hello World");

String content = Files.readString(path);

System.out.println(content); // Output: Hello World

**7. New Garbage Collector: ZGC (Z Garbage Collector)**

* Java 11 introduced the **ZGC**, a low-latency garbage collector that can handle large heaps with minimal pause times.
* It's designed to keep pause times consistently below 10ms, even with multi-terabyte heaps.

**8. Flight Recorder**

* **Java Flight Recorder (JFR)**, a performance monitoring tool that collects profiling data of running applications, became open-source in Java 11.

**9. Deprecation of Nashorn JavaScript Engine**

The **Nashorn JavaScript engine**, which was introduced in Java 8, is deprecated and removed in later versions. Nashorn was used for embedding JavaScript code in Java applications.

**10. Epsilon Garbage Collector**

* Epsilon GC, introduced in Java 11, is a "no-op" garbage collector. It allocates memory but does not reclaim it. It's mainly used for performance testing.

**11. Removal of Java EE and CORBA Modules**

* Java 11 removed the modules related to Java EE (e.g., java.xml.ws, java.xml.bind) and CORBA, which had already been marked as deprecated in earlier versions.

**12. Improved var with Lambda**

Java 11 allows the use of the var keyword inside lambda expressions to infer parameter types, as long as you explicitly use var for all parameters.

(var x, var y) -> x + y

**13. Unicode 10 Support**

Java 11 adds support for **Unicode 10.0** to handle additional characters and symbols across the platform.

These improvements in Java 11 aim to boost performance, simplify code, and provide modern tools for building efficient applications. Let me know if you want to dive deeper into any of these features!

**Java 17 version inhancements**

**Java 17**, released in **September 2021**, is a **Long-Term Support (LTS)** version, bringing several important features and updates. Below are some key new features and improvements introduced in this release:

**1. Sealed Classes (JEP 409):**

* **Sealed classes** restrict which other classes can extend or implement them, providing more control over the inheritance hierarchy.

public sealed class Shape permits Circle, Square {

// class definition

}

final class Circle extends Shape {

// Circle class details

}

final class Square extends Shape {

// Square class details

}

**Pattern Matching for instanceof (JEP 394):**

* Simplifies code by combining type casting and pattern matching in instanceof checks.
* Example

if (obj instanceof String s) {

System.out.println(s.toUpperCase()); // No need to cast 'obj' to 'String'

}

**Text Blocks (JEP 378):**

* Provides a simple way to write multi-line string literals, improving readability for code involving large blocks of text (like JSON, SQL, or XML).
* Example

String json = """

{

"name": "John",

"age": 30

}

""";

**Enhanced switch (JEP 361):**

* Improves the switch statement, making it more powerful and expressive by allowing it to return values and work with patterns.
* Example:

String result = switch (day) {

case MONDAY, FRIDAY, SUNDAY -> "Weekend";

case TUESDAY -> "Workday";

default -> "Invalid day";

};

**Records (JEP 395):**

* Introduces **Records**, a compact syntax for declaring classes meant to hold immutable data. They act as **data carriers** with minimal boilerplate code.
* Example

public record Point(int x, int y) {}

Point p = new Point(10, 20);

System.out.println(p.x()); // Accessor for x

Foreign Function & Memory API (Incubator, JEP 412):

* Introduces APIs to interact with native code and memory outside of the JVM. This provides safer and more efficient access to native libraries and off-heap memory without using JNI.

7. Deprecation of the SecurityManager (JEP 411):

* Marks the SecurityManager for future removal. The SecurityManager was a feature used to impose security policies on Java applications, but its use has significantly declined.

8. Strong Encapsulation by Default (JEP 403):

* Strengthens encapsulation by making all internal APIs in the java.\* modules inaccessible by default. This is part of the Java Platform Module System (introduced in Java 9) to improve modularity and security.

9. macOS/AArch64 Port (JEP 391):

* Adds support for running Java on Apple Silicon (M1) chips and AArch64-based architectures, ensuring future compatibility with ARM processors on macOS.

10. Vector API (Incubator, JEP 414):

* Provides an API to work with vector computations, offering performance improvements by leveraging SIMD (Single Instruction Multiple Data) instructions on modern CPUs.

11. Removal of RMI Activation (JEP 407):

* The RMI Activation mechanism was removed, as it is an outdated feature that sees very limited use.

12. New macOS Rendering Pipeline (JEP 382):

* This feature replaces the old rendering pipeline on macOS with one that uses Apple’s Metal API, improving performance on macOS systems.

**Java 23 version enhancements**

JDK 17 binaries are free to use in production and free to redistribute, at no cost, under the [Oracle No-Fee Terms and Conditions](https://www.java.com/freeuselicense) (NFTC).

Java is the world's most prolific programming language and development platform. Oracle Java SE Universal Subscription offers the most complete support and advanced management features available today. Protect your enterprise investment with security updates and world-class support from the stewards of Java. Now the subscription includes entitlement to Oracle GraalVM, Java SE Subscription Enterprise Performance Pack, and Java Management Service, an Oracle Cloud service that monitors and manages all your Java deployments.

**Java 23 is now available**

The next Java release improves the performance, stability, and security of Java application development.

JDK 23 binaries are free to use in production and free to redistribute, at no cost, under the [Oracle No-Fee Terms and Conditions](https://www.java.com/freeuselicense) (NFTC).

JDK 23 will receive updates under these terms, until March 2025, when it will be superseded by JDK 24.

**what is the java jdk latest release**

The latest release of Java is **JDK 23**, launched in September 2024. This version brings several improvements in terms of language features, performance, and security.

Notable features of JDK 23 include:

1. **Primitive Types in Patterns**: This enhancement simplifies pattern matching for primitive types, making it easier for developers to work with them across instanceof and switch contexts.
2. **Graal Just-In-Time (JIT) Compiler**: Integrated directly into the Oracle JDK, providing faster startup times and improving performance for applications that generate large amounts of garbage early on.
3. **Structured Concurrency**: This helps developers manage multithreaded code more efficiently by grouping related tasks into single units of work.
4. **Markdown Support in Javadoc**: JDK 23 allows developers to write documentation comments using Markdown, making it easier to maintain readable API documentation.
5. **Z Garbage Collector (ZGC)**: Enabled by default in generational mode, it improves efficiency by reducing the resource cost associated with garbage collection.

These improvements make JDK 23 a powerful tool for modern Java applications, with particular focus on performance, usability, and maintaining backward compatibility.

1. **Primitive Types in Patterns**

is a feature introduced as a **preview** in Java 23 under **Project Amber**. This feature enhances pattern matching, instanceof, and switch by extending their capabilities to work seamlessly with **primitive types**.

**Key Highlights:**

1. **Pattern Matching for Primitive Types**: Previously, pattern matching in Java worked well for reference types (like String, Object, etc.) but had limitations when dealing with primitive types (like int, double, etc.). With this enhancement, you can now use pattern matching with primitive types in all pattern contexts.

Example:

if (x instanceof int i) {

System.out.println(i \* 2); // x is now matched and cast as an int

}

1. **Switch Expression Support for Primitives**: Java’s switch expressions can now handle primitive patterns, improving the usability of switch for conditions involving primitive values.

Example:

switch (x) {

case int i -> System.out.println("Integer: " + i);

case double d -> System.out.println("Double: " + d);

default -> System.out.println("Other type");

}

1. **Improved Uniformity**: By allowing primitive types in patterns, instanceof, and switch, Java becomes more consistent and expressive. It reduces the friction between working with primitive and reference types in complex code.
2. The **Graal Just-In-Time (JIT) Compiler**

**this** is an advanced feature integrated into the Oracle JDK, including the latest **Java 23** release. It's designed to improve the performance of Java applications, particularly those that are resource-intensive or involve complex calculations.

**Key Aspects of Graal JIT:**

1. **Performance Boost**: The Graal JIT compiler enhances the speed of applications by optimizing bytecode at runtime. This is particularly useful for long-running applications like microservices or backend systems, which benefit from improved performance over time.
2. **Improved Startup Times**: Java traditionally uses the HotSpot JIT compiler, but Graal offers faster startup times for applications that generate a large amount of garbage early on. This can lead to quicker execution and better efficiency when deploying Java-based cloud services, which makes it particularly relevant for cloud-native solutions.
3. **Better Code Optimization**: Graal JIT aggressively optimizes bytecode by leveraging techniques like **inlining** (embedding frequently called methods directly into the code) and **escape analysis** (identifying objects that do not escape a method or thread). These optimizations reduce memory overhead and improve throughput, making Java programs run faster, especially in high-performance environments like web services and financial applications.
4. **Multi-language Support**: Graal JIT is also part of the **GraalVM** ecosystem, which supports multiple languages (like JavaScript, Python, and Ruby) within the same runtime. This makes it appealing to developers who are working in polyglot environments.
5. **Relevance for Cloud and Containerized Environments:**

If you're working with **containerized Java applications** in platforms like **AWS Lambda**, the Graal JIT compiler is an excellent tool. It helps optimize your Java workloads in resource-constrained environments like microservices, where startup time and runtime efficiency are critical.

With the Graal JIT, you can take advantage of optimizations without the need to modify your code significantly, and your applications will automatically benefit from better memory and execution performance.

1. **Structured Concurrency**

This is a feature introduced in **Java 23** (still in **preview**) to simplify working with concurrent code, specifically focusing on making it more manageable, reliable, and easier to maintain.

### Key Concepts:

1. **Simplified Thread Management**: Structured concurrency provides an API that allows developers to organize multiple related tasks into a single **unit of work**. This eliminates the complexity and risks that often come with manually managing threads, such as potential **memory leaks**, **thread cancellations**, and **shutdown issues**.
2. **Task Grouping**: Instead of managing individual threads or ExecutorService, structured concurrency uses a **scoped** model to group tasks that are related. If one task in the group fails or needs to be canceled, the entire group can be managed together, reducing potential issues with thread management.
3. **Error Handling and Cancellation**: One of the common challenges in multi-threaded programming is handling errors when tasks are canceled or encounter an exception. Structured concurrency improves upon this by providing mechanisms to automatically handle these situations without leaving orphaned threads, ensuring more robust and predictable behavior.
4. **Improved Observability**: Structured concurrency provides better observability into running tasks, which helps with debugging and monitoring. Since all related tasks are structured under a common unit, it’s easier to keep track of task progress, completion, or failure across multiple threads.

import java.util.concurrent.\*;

import java.util.List;

public class StructuredConcurrencyExample {

public static void main(String[] args) throws InterruptedException, ExecutionException {

// Using StructuredTaskScope for structured concurrency

try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {

// Start multiple tasks concurrently

Future<String> task1 = scope.fork(() -> fetchDataFromServiceA());

Future<String> task2 = scope.fork(() -> fetchDataFromServiceB());

// Wait for all tasks to complete

scope.join(); // Wait for both tasks to finish

scope.throwIfFailed(); // Propagate exceptions, if any

// Process results once all tasks have completed

String resultA = task1.resultNow();

String resultB = task2.resultNow();

System.out.println("Results: " + resultA + ", " + resultB);

}

}

// Simulated services for data fetching

static String fetchDataFromServiceA() throws InterruptedException {

Thread.sleep(1000); // Simulate delay

return "Data from Service A";

}

static String fetchDataFromServiceB() throws InterruptedException {

Thread.sleep(1200); // Simulate delay

return "Data from Service B";

}

}

**Explanation:**

1. **StructuredTaskScope**: The StructuredTaskScope is used to manage related tasks within a scope. When the scope is closed (using try-with-resources), it ensures that all tasks are handled appropriately (either completed or failed).
2. **Forking Tasks**: We use scope.fork() to run tasks concurrently. Each task runs in a separate thread, but they are managed as part of the same group.
3. **Join and ThrowIfFailed**:
   * scope.join() waits for all tasks to complete.
   * scope.throwIfFailed() propagates any exception that occurred during task execution.
4. **Result Retrieval**: Once all tasks have completed, results are obtained via task.resultNow(), ensuring the tasks have finished executing successfully.
5. **Markdown support in Javadoc**

Java 23 introduces **Markdown support in Javadoc** as part of its new features, allowing developers to write more readable and expressive documentation. It enables the use of **Markdown syntax** alongside traditional HTML in Javadoc comments, making it easier to write structured and aesthetically pleasing documentation.

/\*\*

\* # Markdown in Javadoc Example

\*

\* This is an example of using \*\*Markdown\*\* in Javadoc comments.

\*

\* ## Features

\* - Use of \*\*bold\*\* text

\* - Use of \*italic\* text

\* - Code blocks:

\*

\* ```java

\* System.out.println("Hello, World!");

\* ```

\*

\* ## Advantages

\* 1. Easier to write

\* 2. More readable

\* 3. Seamless integration with existing HTML tags

\*

\* Here's a link to [Java documentation](https://docs.oracle.com/javase/tutorial/).

\*/

public class MarkdownJavadocExample {

/\*\*

\* Main method demonstrating Javadoc Markdown support.

\*

\* @param args Command line arguments

\*/

public static void main(String[] args) {

System.out.println("Check out the Markdown support in Javadoc!");

}

}

**Key Markdown Features Supported:**

* **Headings** (#, ##, etc.)
* **Bold** (\*\*text\*\*)
* **Italic** (\*text\*)
* **Lists** (both ordered and unordered)
* **Code blocks** (using triple backticks ```)

1. **Z Garbage Collector (ZGC)**:

Enabled by default in generational mode, it improves efficiency by reducing the resource cost associated with garbage collection.

The **Z Garbage Collector (ZGC)** is a highly scalable, low-latency garbage collector introduced in **JDK 11** and improved in later versions like **Java 23**. It is designed to handle very large heaps of memory with minimal impact on application throughput and latency.

**Key Features of ZGC:**

1. **Low Latency**: ZGC is designed to keep **pause times under 10ms**, regardless of the heap size. This makes it especially suited for applications where responsiveness and predictable latency are critical, such as real-time systems, gaming, or financial services.
2. **Scalability**: ZGC can handle heap sizes ranging from a few megabytes to **multi-terabyte** heaps. It scales well without increasing pause times, which makes it ideal for applications that require a large memory footprint.
3. **Concurrent Operations**: Most of ZGC’s operations are performed **concurrently** with the execution of the application, meaning that it does not require the application to pause for long garbage collection cycles. Key operations like marking, relocating, and compacting memory are done in the background, reducing the impact on application performance.
4. **Generational Mode**: Starting in **Java 21**, ZGC supports **generational garbage collection**, which divides the heap into **young** and **old** generations. This results in more efficient memory management, as objects that are frequently created and destroyed (young generation) are collected more frequently, while long-lived objects (old generation) are managed separately.
5. **No Full GC**: ZGC avoids traditional **"stop-the-world"** full garbage collections. It continuously manages memory in small, incremental steps, ensuring that the application runs smoothly even during garbage collection.
6. **Coloring Pointers**: ZGC uses **colored pointers** to track object locations and states without pausing the application. This technique allows ZGC to move objects around in memory without disrupting ongoing application threads.

**Example of Enabling ZGC:**

To use ZGC, you can enable it with the following JVM options:

bash

Copy code

java -XX:+UseZGC -Xms10g -Xmx10g -jar your-application.jar

Here, -XX:+UseZGC activates ZGC, and -Xms and -Xmx set the initial and maximum heap sizes to 10GB.

**Benefits of ZGC:**

1. **Improved Throughput**: ZGC's concurrent approach ensures minimal impact on application throughput, especially for applications with large heaps.
2. **Predictable Performance**: ZGC minimizes unpredictable long pauses during garbage collection, leading to more consistent application performance.
3. **Ideal for Modern Applications**: It is well-suited for modern applications running in environments like **cloud services** or **microservices**, where low-latency and scalability are critical.

**Use Cases:**

* **Large-scale web applications** that handle high traffic and require minimal response time.
* **Real-time systems** that can’t afford long pauses due to garbage collection.
* **Big data applications** or **AI systems** using large heaps of memory.

#-------------------------------------------