Java 11

Java 11 was released in September 2018 and brought a wealth of new features and enhancements to the Java platform. This release is an LTS (Long-Term Support) version, making it a popular choice for enterprises.

1. Local-Variable Syntax for Lambda Parameters

Java 11 allows the use of the var keyword in lambda expressions, providing local variable type inference for lambda parameters. This makes lambda expressions more flexible and consistent with local variable declarations.

Explanation

- var item in the lambda expression allows type inference for the lambda parameter, similar to how var is used for local variables.

2. HTTP Client (Standard)

What is the HTTP Client?

Java 11 standardizes the HTTP client, which was introduced as a preview feature in Java 9. This API supports HTTP/1.1 and HTTP/2 and provides a modern, feature-rich client for making HTTP requests.

Explanation

- HttpClient is used to send HTTP requests and receive responses.
- HttpRequest constructs an HTTP GET request.
- HttpResponse contains the response body as a string.

3. New Collection.toArray() Method

What is the New Collection.toArray() Method?

Java 11 adds a new toArray(IntFunction<T[]>) method to the Collection interface, allowing collections to be converted to arrays with improved type safety and performance.

Explanation

- list.toArray(String[]::new) converts the list to an array using the new toArray() method, specifying the array type with a constructor reference.

4. New String Methods

What are the New String Methods?

Java 11 introduces several new methods to the String class, enhancing its functionality and convenience.

isBlank(): The isBlank() method checks if a string is empty or contains only whitespace.

lines(): The lines() method returns a stream of lines from the string.

strip(), stripLeading(), and stripTrailing(): These methods remove whitespace from the beginning and end of the string.

repeat(): The repeat() method repeats the string a specified number of times.

5. Files.readString() and writeString()

Java 11 adds new methods to the Files class for reading and writing strings to and from files, simplifying file I/O operations.

Explanation

- Files.writeString() writes a string to a file.
- Files.readString() reads a string from a file, simplifying file I/O operations.

6. Path.of()

Java 11 introduces the Path.of() method as a convenient way to create Path instances. This method simplifies the creation of Path objects by providing a more concise and readable alternative to Paths.get(), which was used in earlier Java versions.

Explanation

- Path.of(): This method is used to create a Path instance by joining the specified path components. It enhances readability and usability when working with file paths.
- Comparing with Paths.get(): Although Paths.get() is still available, Path.of() is preferred for its cleaner syntax. Both methods achieve the same result, but Path.of() is more concise and aligns with the introduction of factory methods for other Java APIs.

Advantages of Path.of()

- Readability: Path.of() provides a clearer and more intuitive way to construct file paths, enhancing code readability.
- Consistency: The introduction of factory methods like Path.of() is part of a broader effort in Java to provide consistent and expressive APIs, making code easier to write and maintain.
- Modern Approach: Path.of() represents the modern approach to handling file paths, aligning with Java's ongoing improvements in usability and developer experience.

Use Cases for Path.of()

- File Manipulation: Use Path.of() to create paths for reading, writing, or manipulating files and directories in a more readable manner.
- Cross-Platform Compatibility: Path.of() automatically handles the appropriate file separators for different operating systems, making it easier to write cross-platform Java applications.
- Code Modernization: When updating legacy code, replacing Paths.get() with Path.of() can improve readability and maintainability.

7. Epsilon: A No-Op Garbage Collector

What is Epsilon?

Epsilon is a no-op garbage collector introduced in Java 11. It handles memory allocation without reclaiming memory, making it useful for performance testing, memory profiling, and short-lived applications where garbage collection is not needed. Epsilon allows developers to focus on application throughput and performance characteristics without the interference of garbage collection pauses.

Example: Enable Epsilon Garbage Collector

To enable Epsilon, use the following JVM options:

```
java -XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC -jar MyApp.jar
```

Explanation

- No-Op Garbage Collection: Epsilon does not perform any memory reclamation. This means that while it allocates memory for objects, it does not collect garbage, leading to predictable performance without pauses.
- **Use Cases:**
  - Performance Testing: Epsilon is ideal for benchmarking and testing application performance, as it removes the variable of garbage collection pauses.
  - Memory Profiling: It can be used to understand memory usage patterns in applications and identify memory leaks.
  - Short-Lived Applications: Epsilon is suitable for applications with a known lifespan that can terminate before exhausting memory.

Considerations

- Memory Management: Since Epsilon does not reclaim memory, applications using it will eventually run out of memory unless terminated or restarted. This requires careful consideration of application lifecycle and resource management.
- Use with Caution: Epsilon should be used in scenarios where garbage collection is either not needed or managed through other means. It is not suitable for long-running applications that require memory management.

8. Launch Single-File Source-Code Programs

What is Launch Single-File Source-Code Programs?

Java 11 introduces the ability to run a single Java source file directly without the need for explicit compilation. This feature simplifies the execution of small programs and scripts, making Java more accessible for quick tasks and experimentation

Explanation

- **Direct Execution**: The java command can directly execute a Java source file, compiling it in memory and running it in one step. This feature is particularly useful for small scripts, educational purposes, and quick testing of code snippets.

- **Simplified Workflow**: Developers can quickly test ideas and perform tasks without the need to set up a full project or compile code explicitly.
- **Integration with Editors**: This feature can be leveraged in text editors and IDEs that support integrated terminal execution, streamlining development workflows.

Benefits of Single-File Execution

- **Rapid Prototyping**: Quickly test code snippets and ideas without creating a full project structure.
- **Educational Use**: Ideal for teaching and learning Java concepts, as it reduces setup complexity and allows students to focus on code logic.
- **Convenience**: Developers can write and execute Java code on the fly, enhancing productivity for small tasks and scripts.

Limitations

- **Single-Class Limitation**: This feature is best suited for simple programs and may not support complex applications with multiple classes and dependencies.
- **Not for Production**: While useful for development and testing, single-file execution is not typically used for deploying production applications.

9. Nest-Based Access Control

What is Nest-Based Access Control?

Nest-Based Access Control is a feature introduced in Java 11 that allows classes that are logically grouped into a "nest" to access each other's private members. This feature enhances encapsulation and provides more flexibility for code organization by allowing inner classes and their enclosing classes to share private fields and methods.

Explanation

- **Nests and Nest Members**: A nest is a group of classes that are logically related, typically consisting of an outer class and its nested inner classes. Nest members can access each other's private fields and methods directly, as shown in the example
where OuterClass and InnerClass access each other's private members.
- **Improved Encapsulation**: Nest-Based Access Control improves encapsulation by maintaining the logical grouping of classes while allowing controlled access to private members.
- **Flexibility**: This feature provides more flexibility in code organization, allowing developers to structure their classes in a way that best fits their design while maintaining access control.

Use Cases for Nest-Based Access Control

- **Inner Classes**: Facilitate communication between inner classes and their enclosing classes without exposing private members to unrelated classes.
- **Encapsulation**: Enhance encapsulation by keeping related classes together while allowing them to access necessary private members.
- **Code Organization**: Organize code more effectively by grouping related classes into nests and simplifying access to shared resources.

Considerations

- **Logical Grouping**: Ensure that only logically related classes are grouped into nests to maintain a clear and maintainable code structure.
- **Access Control**: Use nest-based access control judiciously to avoid exposing private members unnecessarily, maintaining the integrity of encapsulation.

10. Analysis Tools

Java 11 introduces new analysis tools to help developers monitor and analyze the performance of their applications, providing insights into application behavior and resource usage.

What is Java Flight Recorder?

Java Flight Recorder (JFR) is a monitoring tool that records information about a running Java application. It provides insights into the application's performance, allowing developers to identify and diagnose issues effectively.

Enable Java Flight Recorder

To start a Java application with JFR enabled, use the following JVM options:

```
java -XX:StartFlightRecording=duration=60s,filename=recording.jfr -jar MyApp.jar
```

Use Cases for Java Flight Recorder

- **Performance Tuning**: Analyze performance data to identify and address bottlenecks, optimizing application efficiency and responsiveness.
- **Production Monitoring**: Use JFR in production environments to monitor applications and ensure they meet performance requirements.
- **Troubleshooting**: Diagnose issues and identify the root causes of performance problems, enabling faster resolution and improved application stability.

Considerations

- **Configuration**: Configure JFR appropriately for the application's needs, balancing the level of detail captured with the desired performance impact.
- **Data Analysis**: Use tools like Java Mission Control to analyze JFR recordings and extract actionable insights from the data.

Low-Overhead Heap Profiling

Low-overhead heap Profiling is a feature in Java 11 that provides detailed information about heap usage with minimal performance impact. This feature helps developers identify memory leaks, optimize memory usage, and improve application performance.

Enable Heap Profiling

To enable heap profiling, use the JVM option:

```
java -XX:+UnlockExperimentalVMOptions -XX:+HeapDumpOnOutOfMemoryError -
XX:HeapDumpPath=heap.hprof -jar MyApp.jar
```

Explanation

- **Heap Profiling**: The -XX:+HeapDumpOnOutOfMemoryError option generates a heap dump when an OutOfMemoryError occurs, capturing a snapshot of the application's heap memory at the time of the error.
    - The heap dump file (heap.hprof) can be analyzed using tools like VisualVM or Eclipse Memory Analyzer to understand memory usage patterns, identify memory leaks, and optimize memory allocation.
- **Low Overhead**: Heap profiling is designed to have minimal impact on application performance, allowing developers to gather detailed memory usage data without significant disruption.

Use Cases for Low-Overhead Heap Profiling

- **Memory Leak Detection**: Identify and address memory leaks by analyzing heap dumps and understanding memory allocation patterns.
- **Performance Optimization**: Optimize memory usage by identifying inefficient memory allocation and improving resource management.
- **Troubleshooting**: Diagnose memory-related issues and improve application stability by understanding heap usage and identifying potential problems.

Considerations

- **Heap Dump Size**: Heap dumps can be large, especially for applications with significant memory usage. Ensure sufficient storage is available for heap dump files.
- **Analysis Tools**: Use appropriate tools to analyze heap dumps and extract actionable insights from the data, improving application performance and stability.

11. Experimental and Preview Features

Java 11 includes experimental and preview features that allow developers to explore new capabilities before they become standard. These features provide insights into future Java developments and enable developers to experiment with cutting-edge functionality.

ZGC: A Scalable Low-Latency Garbage Collector (Experimental)

ZGC (Z Garbage Collector) is an experimental garbage collector introduced in Java 11. It is designed for applications with large heaps, providing low-latency garbage collection and minimizing pause times. ZGC aims to scale efficiently with heap sizes ranging from gigabytes to terabytes, making it suitable for memory-intensive applications.

Example: Enable ZGC

To enable ZGC, use the JVM option:

```
java -XX:+UnlockExperimentalVMOptions -XX:+UseZGC -jar MyApp.jar
```

Explanation

- **Low-Latency Garbage Collection**: ZGC minimizes pause times by performing most of its work concurrently with the application, reducing the impact of garbage collection on application performance.
- **Scalability**: ZGC is designed to scale with large heap sizes, making it suitable for applications that require significant memory resources.
- **Concurrent Processing**: ZGC performs most of its operations concurrently, allowing the application to continue running with minimal disruption.

Use Cases for ZGC

- **Memory-Intensive Applications**: Use ZGC for applications with large memory requirements, such as data processing, analytics, and machine learning workloads.
- **Low-Latency Applications**: ZGC is ideal for applications where low latency and responsiveness are critical, such as real-time systems and high-frequency trading platforms.
- **Scalable Infrastructure**: Deploy ZGC in environments where scalability is essential, such as cloud-native applications and microservices architectures.

Considerations

- **Experimental Status**: ZGC is an experimental feature, and its behavior and performance characteristics may change in future Java releases.
- **Tuning and Configuration**: Optimize ZGC settings and configuration to achieve the desired balance between performance and resource utilization.

12. Deprecations and Deletions

Java 11 removes or deprecates several features and modules that are no longer needed or have been replaced by better alternatives. These changes reflect Java's ongoing evolution and commitment to providing a modern and efficient platform.

Remove the Java EE and CORBA Modules

Java EE (Enterprise Edition) and CORBA (Common Object Request Broker Architecture) modules are removed from the JDK in Java 11. These modules have been superseded by modern alternatives and are no longer needed in the JDK.

Deprecate the Nashorn JavaScript Engine

Nashorn is a JavaScript engine introduced in Java 8. Java 11 deprecates Nashorn as more modern JavaScript engines and solutions are available. This deprecation reflects the shift towards using JavaScript engines that are better integrated with modern web development practices.

Deprecate the Pack200 Tools and API

Pack200 is a compression tool and API for Java archives (JARs). Java 11 deprecates Pack200, as more efficient compression formats and tools are available. This deprecation reflects the availability of modern compression techniques that offer better performance and compression ratios.

JavaFX Goes Its Own Way

JavaFX is a platform for building rich desktop applications. Java 11 removes JavaFX from the JDK, allowing it to evolve independently. This separation enables JavaFX to be updated and improved at a pace different from that of the JDK.

13. Other Changes in Java 11

Java 11 introduces several other changes that enhance the platform and improve performance. These changes reflect Java's commitment to providing a modern, efficient, and secure platform for application development.

Unicode 10

Java 11 supports Unicode 10, which includes additional characters, scripts, and emoji. This support enhances Java's ability to handle diverse character sets and languages, improving internationalization capabilities.

Improve Aarch64 Intrinsics

Java 11 improves Aarch64 intrinsics, providing better performance and optimization for ARM-based architectures. These improvements enhance the performance of Java applications running on ARM-based devices, making Java a more viable option for embedded and mobile development.

Transport Layer Security (TLS) 1.3

Java 11 supports TLS 1.3, the latest version of the Transport Layer Security protocol, providing enhanced security and performance for secure communications. TLS 1.3 offers improved security features and reduced handshake latency, enhancing the performance and security of Java applications.

ChaCha20 and Poly1305 Cryptographic Algorithms

Java 11 adds support for the ChaCha20 and Poly1305 cryptographic algorithms, providing alternative encryption and authentication options. These algorithms offer improved performance and security, especially in scenarios where hardware acceleration is unavailable.

Key Agreement with Curve25519 and Curve448

Java 11 supports key agreement with Curve25519 and Curve448, providing modern elliptic curve cryptography options. These curves offer enhanced security and performance for key exchange operations, making Java applications more secure and efficient.

Dynamic Class-File Constants

Java 11 introduces dynamic class-file constants, allowing a more flexible and efficient representation of constant pool entries. This feature improves the performance and flexibility of Java applications by reducing the overhead of constant pool management.