UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS ILMU KOMPUTER

Module 05: Java Profiling
Advanced Programming

Muhammad Yusuf Khadafi

Editor: Teaching Team (Lecturer and Teaching Assistant)

Advanced Programming

Semester Genap 2024

Fasilkom UI

Author: *Muhammad Yusuf Khadafi*
Email: m.yusuf03@cs.ui.ac.id
© 2024 Fakultas Ilmu Komputer Universitas Indonesia
This work uses license: Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

Table of Contents

Contents

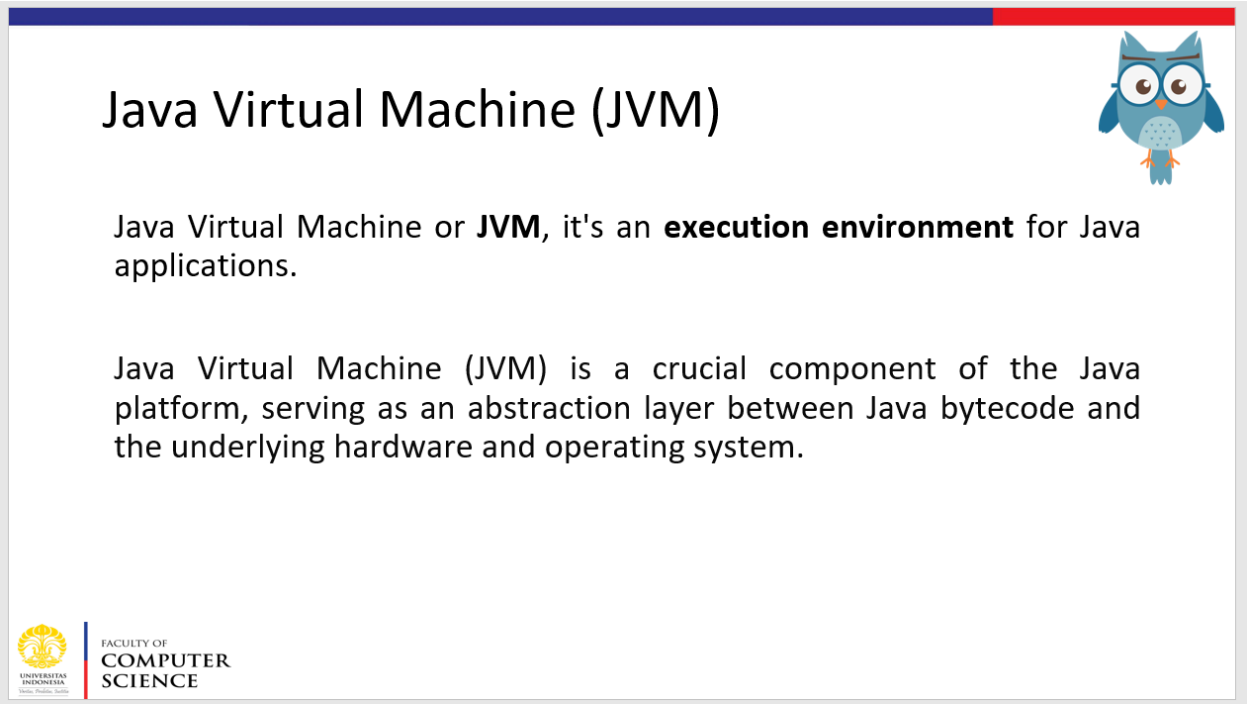
Learning Objectives

1. Student is able to comprehend the memory model in Java, including JVM components, JIT compiler, and Java garbage collector.
2. Student is able to identify performance issue
    - Recognize common performance bottlenecks in Java programs.
    - Utilize JMeter for measuring and diagnosing performance issues.
3. Student is able to perform profiling and logging:
    - Use profiling tools to analyze CPU and memory usage.
    - Implement effective logging strategies for Java applications.

References

1. Oaks, Scott. Java Performance The Definitive Guide Getting the Most Out of Your Code. O'Reilly Media, Inc., 2014 .
2. Putten, M. van, & Kennedy, S. Java Memory Management: A comprehensive guide to garbage collection and JVM tuning. Packt Publishing, 2022.

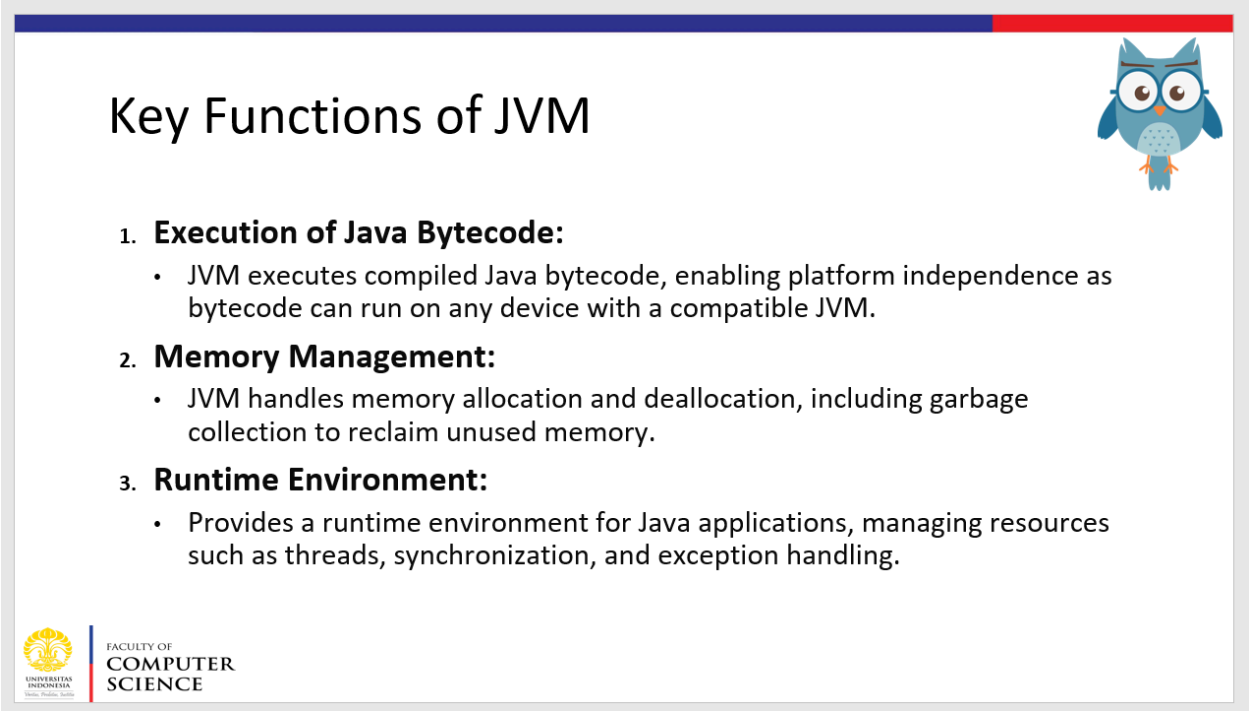
Java Virtual Machine



The Java Virtual Machine (JVM) stands as a cornerstone of the Java programming platform, bridging the gap between Java applications and the diverse landscapes of hardware and operating systems on which they operate. At its core, the JVM is an execution environment designed to offer a uniform experience across various computing platforms, thereby embodying the principle of "write once, run anywhere." This universality is achieved through the use of Java bytecode, an intermediate representation of Java programs. When a Java application is compiled, it is transformed into bytecode, which is platform-independent. The JVM then interprets or compiles this bytecode into native machine code at runtime, ensuring optimal performance on the host system.

Furthermore, the JVM plays a pivotal role in providing a secure execution environment, managing memory through garbage collection, and facilitating seamless integration of new software components. It encapsulates the complexities of dealing with different operating systems and hardware configurations, allowing developers to focus on the logic of their applications rather than on platform-specific idiosyncrasies. By doing so, the JVM not only simplifies development and deployment but also enhances the portability and scalability of Java applications, making it an indispensable tool in the vast ecosystem of Java development.

Key Functions of JVM

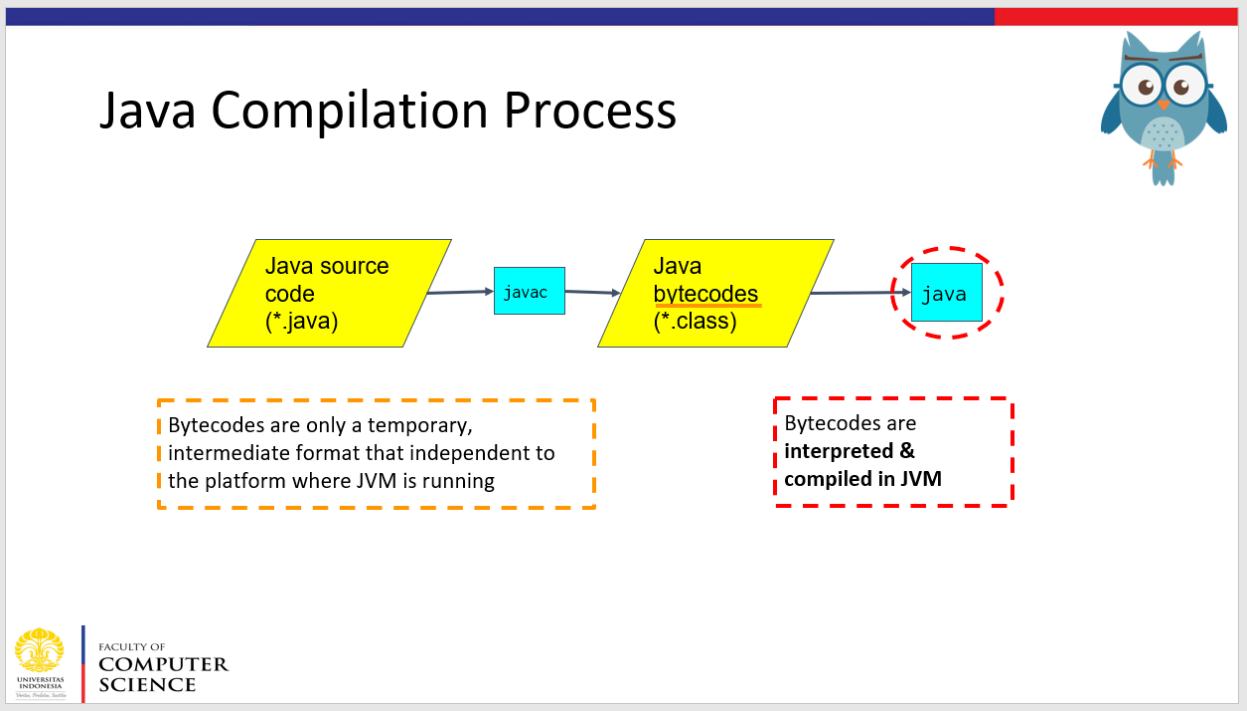


The Java Virtual Machine (JVM) is an integral part of the Java runtime environment, providing a versatile platform for executing Java applications. Its key functions are centered around ensuring that Java maintains its core promise of write once, run anywhere. By executing compiled Java bytecode, the JVM allows applications to transcend the barriers of diverse operating systems and hardware configurations. This bytecode, a middle-ground representation of Java code, is designed to be platform-independent, enabling it to operate seamlessly across any device equipped with a compatible JVM.

In addition to code execution, the JVM excels in memory management. It autonomously handles the allocation and deallocation of memory, significantly reducing the likelihood of memory leaks and other related issues. A pivotal aspect of this process is garbage collection, wherein the JVM identifies and disposes of objects that are no longer in use, thereby reclaiming unused memory and optimizing application performance.

Moreover, the JVM furnishes a comprehensive runtime environment that supports the execution of Java applications. This includes managing essential resources such as threads for concurrent execution, synchronization to control access to shared resources, and robust exception handling mechanisms. These functionalities not only simplify the development process but also enhance the reliability and efficiency of Java applications, making the JVM a fundamental component for Java developers.

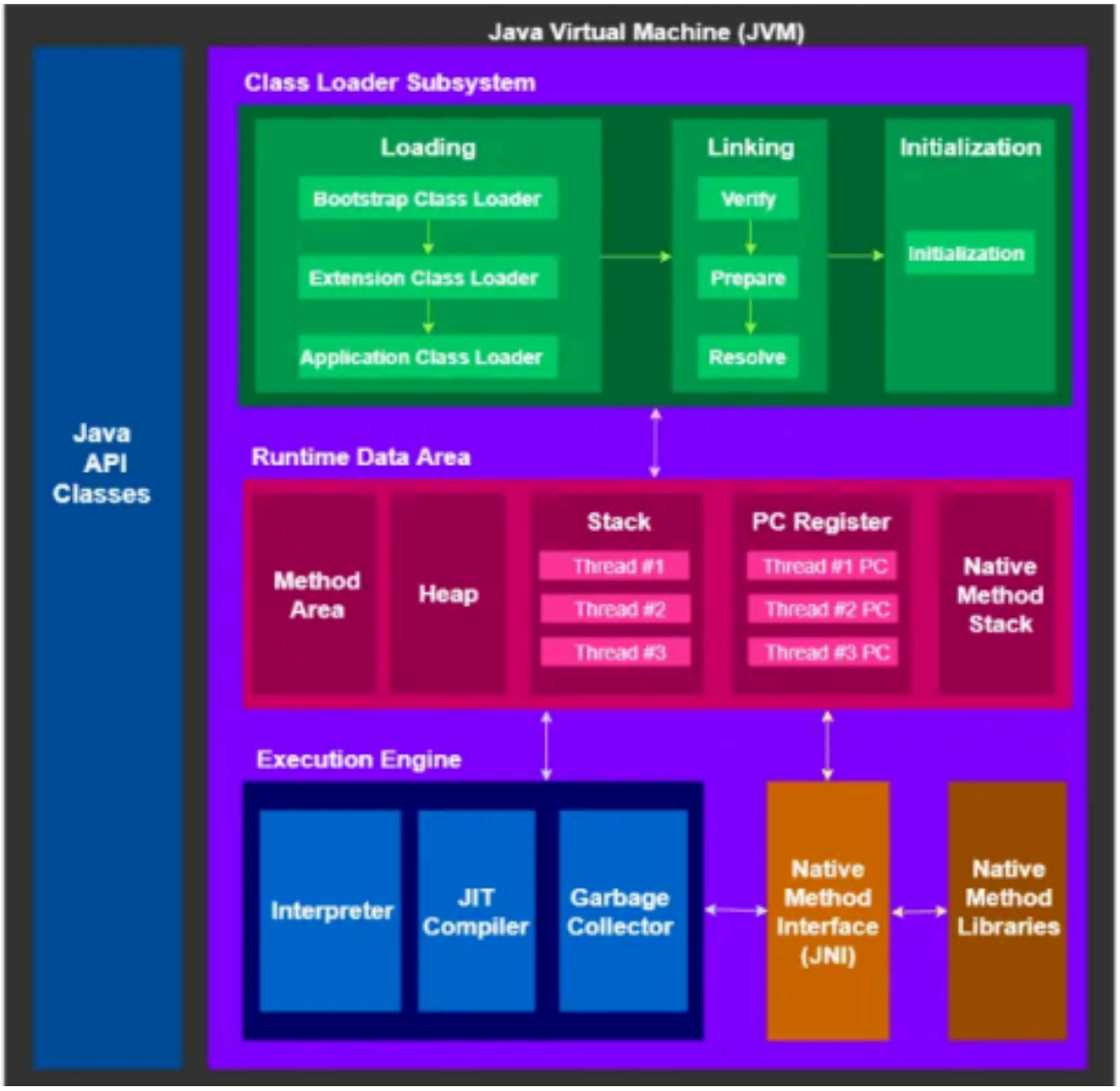
Java Compilation Process



The Java compilation process serves as the backbone for Java's cross-platform capabilities, enabling Java applications to run on any device that hosts a Java Virtual Machine (JVM). This process starts with the Java source code, written in ".java" files, which developers compile using the "javac" command. The outcome of this compilation is not machine code but Java bytecodes, encapsulated in ".class" files. These bytecodes represent a platform-independent, intermediate format, specifically designed to be executed by the JVM.

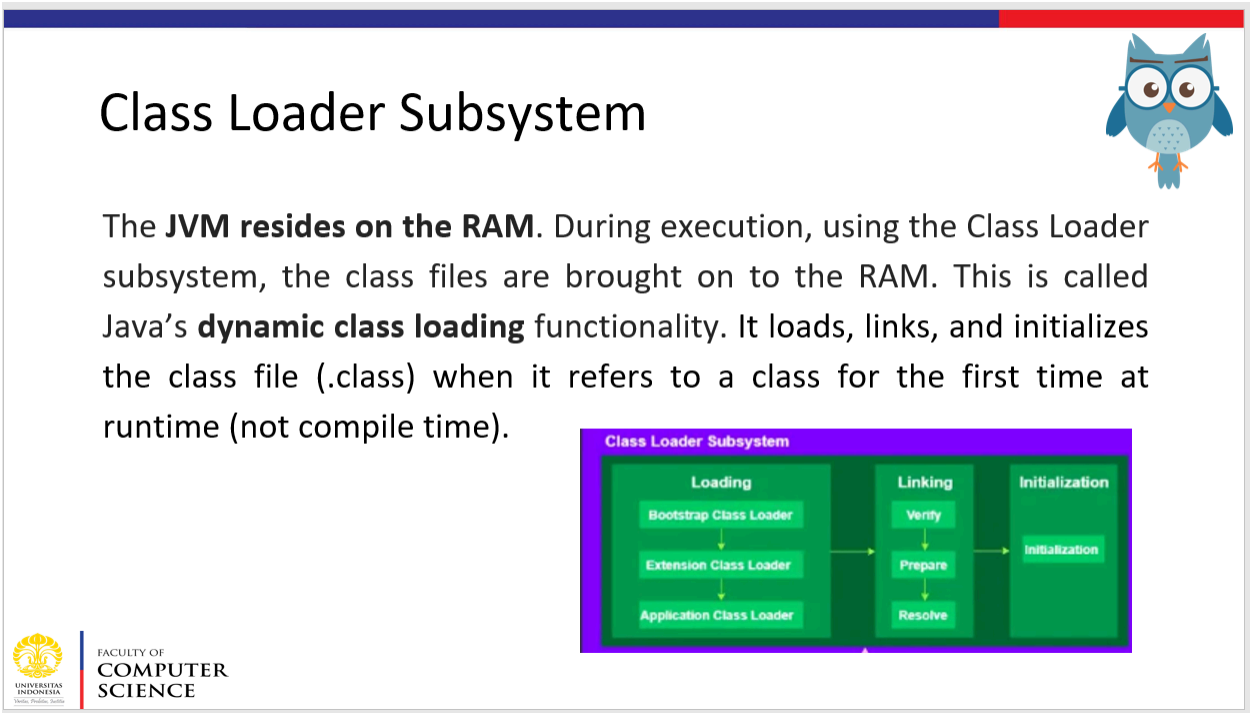
Upon execution, using the "java" command, the JVM interprets or just-in-time compiles these bytecodes into the native code of the host machine, allowing the program to run as if it were written specifically for that platform. This level of abstraction is key to Java's "write once, run anywhere" philosophy, ensuring that developers need not tailor their code for each specific platform. Bytecodes act as a universal language for the JVM, making it possible for a Java application to be run on any device with a JVM, regardless of its underlying hardware or operating system architecture. This process underscores the versatility and portability that have made Java one of the most widely used programming languages.

JVM Architecture



The Java Virtual Machine (JVM) architecture is a sophisticated framework designed to run Java applications across various platforms. At its core, the JVM encompasses several key components: Java API Classes for standard programming interfaces, a Class Loader Subsystem to dynamically load Java classes, a Runtime Data Area for memory allocation during execution, and an Execution Engine that interprets Java bytecode into machine instructions. Additionally, it includes a Native Method Interface and Native Method Libraries, enabling Java applications to interact with software and libraries specific to the host operating system. This comprehensive structure ensures that Java applications can run efficiently and seamlessly on any device equipped with a JVM, maintaining Java's "write once, run anywhere" promise.

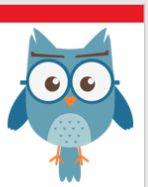
Class Loader Subsystem



The Class Loader Subsystem within the Java Virtual Machine (JVM) is a fundamental mechanism that underscores Java's ability to dynamically load, link, and initialize classes. This functionality is pivotal for Java's runtime efficiency, security, and the robust management of class dependencies and namespaces. Situated in the RAM, the JVM utilizes this subsystem to dynamically bring class files into memory, adhering to Java's principle of loading classes only when they are explicitly referenced during program execution. This contrasts sharply with static loading methodologies, where all classes are loaded at compile time, regardless of their eventual use within the application, leading to increased memory consumption and decreased efficiency.

Java's dynamic class loading is facilitated through a multi-stage process involving loading, linking, and initialization phases. Initially, the Class Loader Subsystem retrieves the .class files from the file system or network sources, making them available in the RAM. Following this, the linking phase takes over, verifying the correctness of the classes, preparing them for execution, and resolving any symbolic references to other classes, ensuring that the class's binary data is in a suitable state for execution.

Class Loader Subsystem



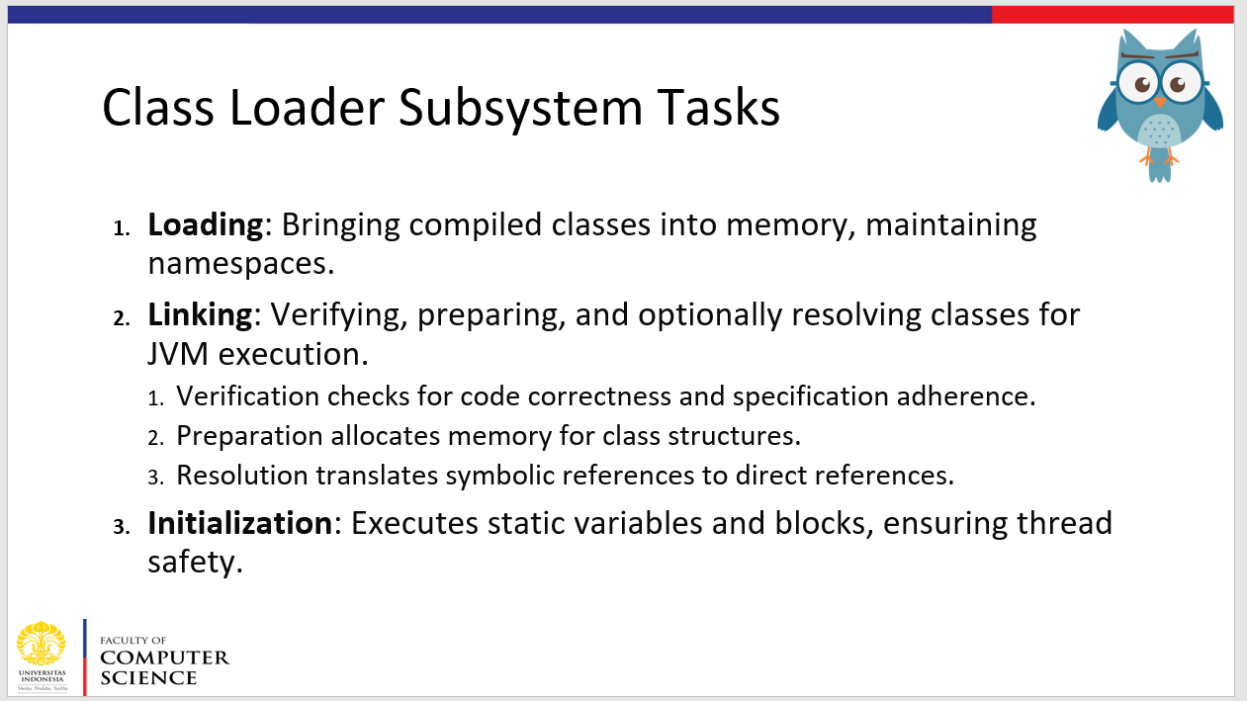
- Class Loaders enhance Java's dynamic capabilities, allowing for flexible, secure, and efficient application development.
- The delegation model prevents class conflicts and ensures a clear hierarchy and namespace management.
- Initialization is carefully synchronized to prevent concurrent issues in a multi-threaded environment.

FACULTY OF COMPUTER SCIENCE

Moreover, the Class Loader Subsystem employs a delegation model for class loading, which significantly enhances Java's dynamic capabilities by ensuring that classes are loaded in a hierarchical manner. This model prevents class conflicts by assigning a clear class loading hierarchy and namespace management, where the Class Loader Subsystem delegates class loading tasks to its parent class loader before attempting to load the class itself. This approach not only ensures a secure runtime environment by preventing the loading of unauthorized classes but also optimizes memory usage and application performance by avoiding the duplication of class data in memory.

In summary, the Class Loader Subsystem is instrumental in enhancing Java's dynamic capabilities, providing a flexible, secure, and efficient framework for application development. Its ability to dynamically load, link, and initialize classes at runtime—coupled with a sophisticated delegation model and careful synchronization mechanisms—enables Java applications to run in a robust, scalable, and thread-safe manner, embodying the essence of Java's design philosophy.

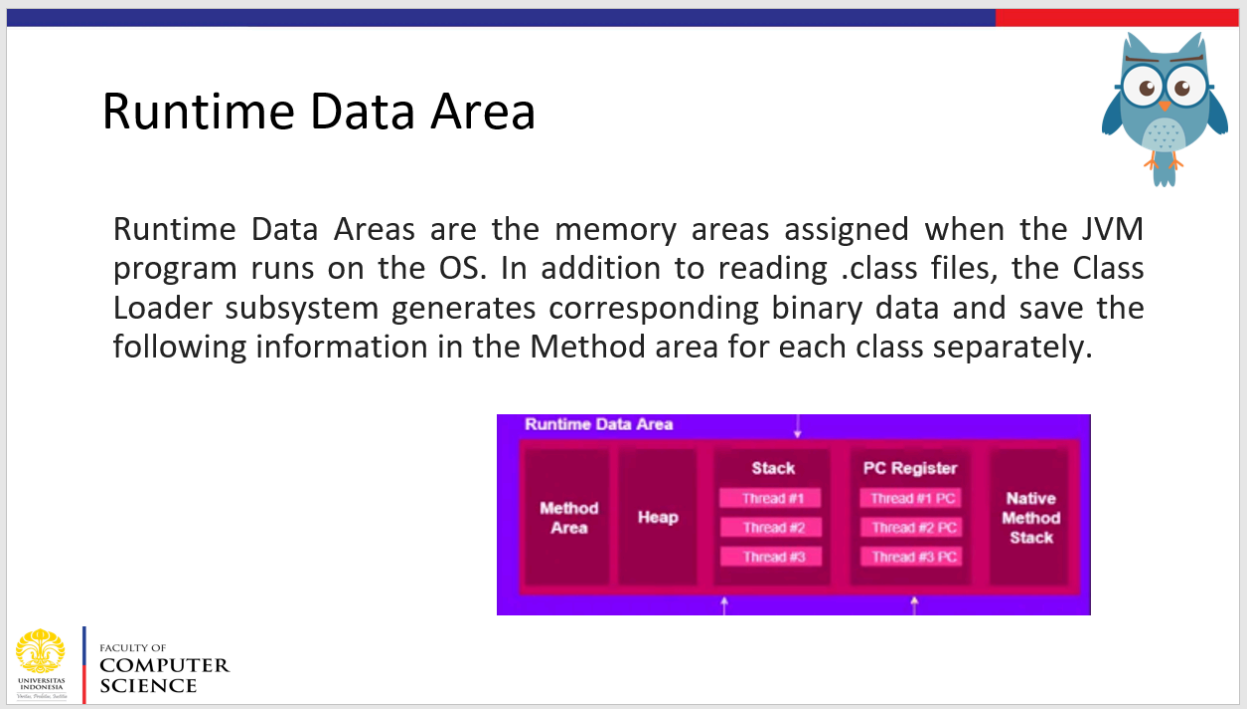
Class Loader Subsystem Tasks



The Class Loader Subsystem performs critical tasks essential for Java application execution, emphasizing efficiency, security, and reliability. Its primary responsibilities include loading, linking, and initializing classes within the Java Virtual Machine (JVM) environment. During the loading phase, it brings compiled classes (.class files) into memory and establishes unique namespaces to prevent conflicts. Linking is a threefold process involving verification, preparation, and resolution. Verification ensures the code's correctness and adherence to Java specifications, while preparation allocates necessary memory for class structures. Resolution then translates symbolic references within the code into direct references, facilitating JVM execution.

The final task, initialization, is crucial for setting up the runtime environment. It involves executing static variables and blocks, with a strong emphasis on maintaining thread safety to prevent issues in multi-threaded applications. Through these processes, the Class Loader Subsystem guarantees that Java applications are executed securely and efficiently, with classes being dynamically loaded, linked, and initialized as needed. This dynamic loading mechanism underpins Java's "write once, run anywhere" capability, allowing for flexible and scalable application development across diverse computing environments.

Runtime Data Area

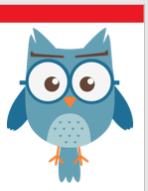


The Runtime Data Areas constitute a fundamental framework within the Java Virtual Machine (JVM), delineating the structured allocation of memory spaces essential for executing Java applications on any operating system. This architecture ensures that Java applications can run efficiently and securely, managing the complex dynamics of memory allocation and garbage collection with precision. When a JVM instance initiates, it meticulously allocates memory to these runtime data areas, each serving distinct roles in the application's lifecycle, from loading classes to executing methods and managing runtime data.

Key among the Runtime Data Areas is the Method Area, a dedicated memory space where the Class Loader subsystem deposits information for each class. This information is not limited to metadata about the class itself but extends to the runtime constant pool, details about methods and fields, and the bytecode for methods and constructors. This comprehensive storage solution enables the JVM to access critical class information rapidly, facilitating swift execution of Java applications.

In addition to the Method Area, the Runtime Data Areas encompass several other crucial components: Heap, Stack, PC Register, and Native Method Stack

Runtime Data Area



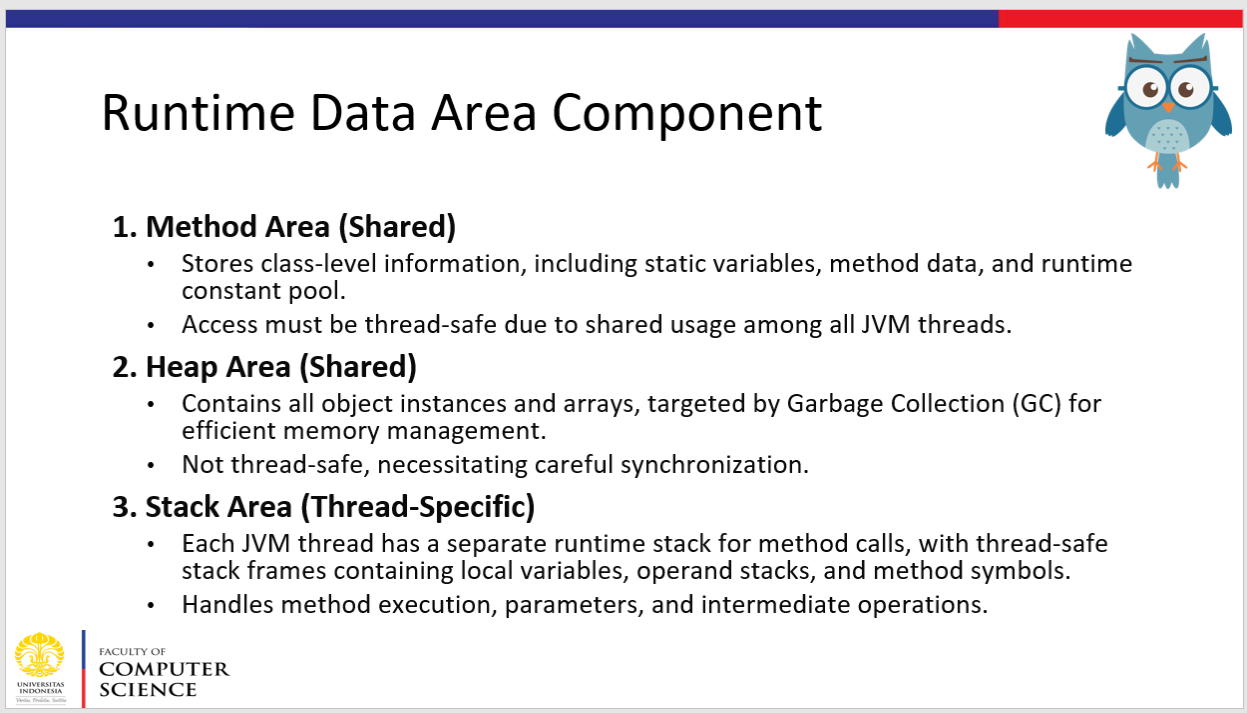
- Runtime Data Areas are allocated during JVM operation, handling the execution of programs.
- Class Loader subsystem reads .class files, generating binary data stored across various memory areas.

UNIVERSITAS INDONESIA
FACULTY OF COMPUTER SCIENCE

These components work in concert to manage the execution and memory needs of Java applications. The Class Loader subsystem's role in reading .class files and generating the necessary binary data to populate these areas is crucial. It ensures that each class's data is methodically stored in the Method Area, ready for quick retrieval during execution. This systematic approach to memory management underpins the JVM's capability to run complex Java applications across diverse computing environments, maintaining optimal performance and reliability.

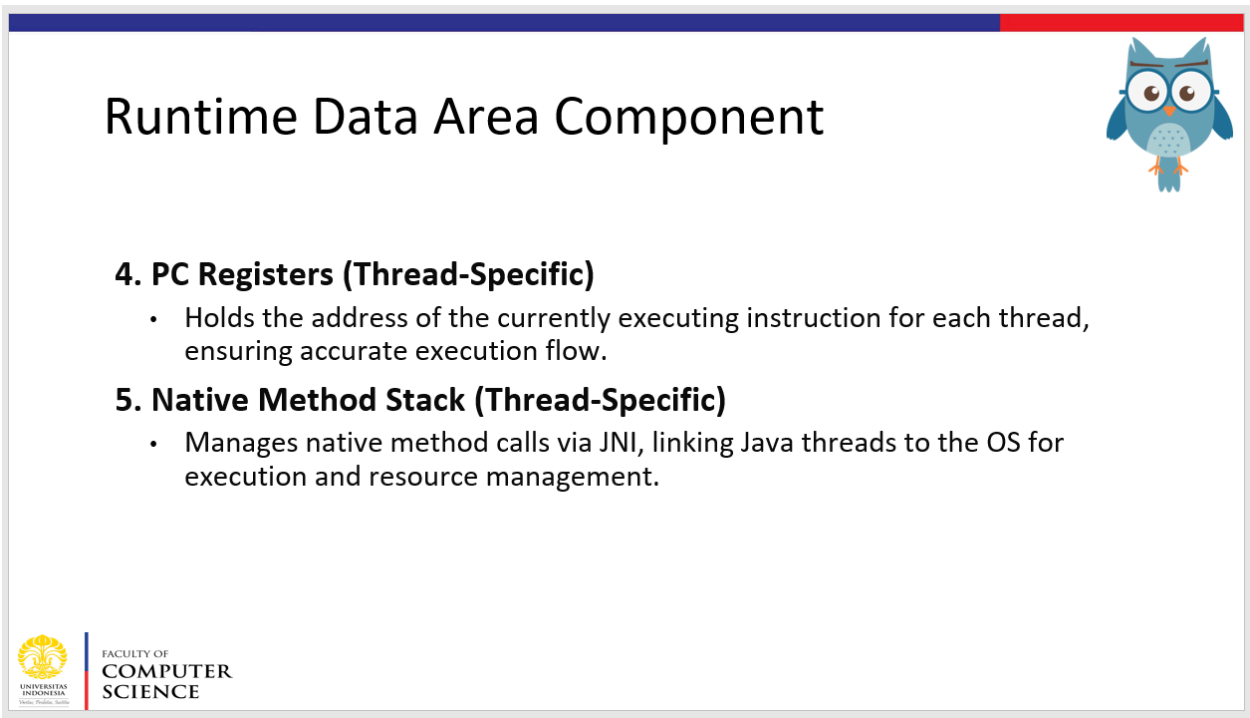
By meticulously organizing memory in this way, the JVM not only guarantees the efficient execution of Java applications but also enhances their security and stability. The Runtime Data Areas' design reflects the JVM's commitment to scalability and performance, ensuring that Java remains a robust platform for software development across various computing platforms, true to its "write once, run anywhere" philosophy.

Runtime Data Area Component



The Runtime Data Area within the Java Virtual Machine (JVM) is a complex structure, meticulously designed to support the execution of Java applications across various computing environments. This area is segregated into five distinct components, each serving a unique purpose in the program's execution lifecycle, ensuring efficiency, thread safety, and effective memory management.

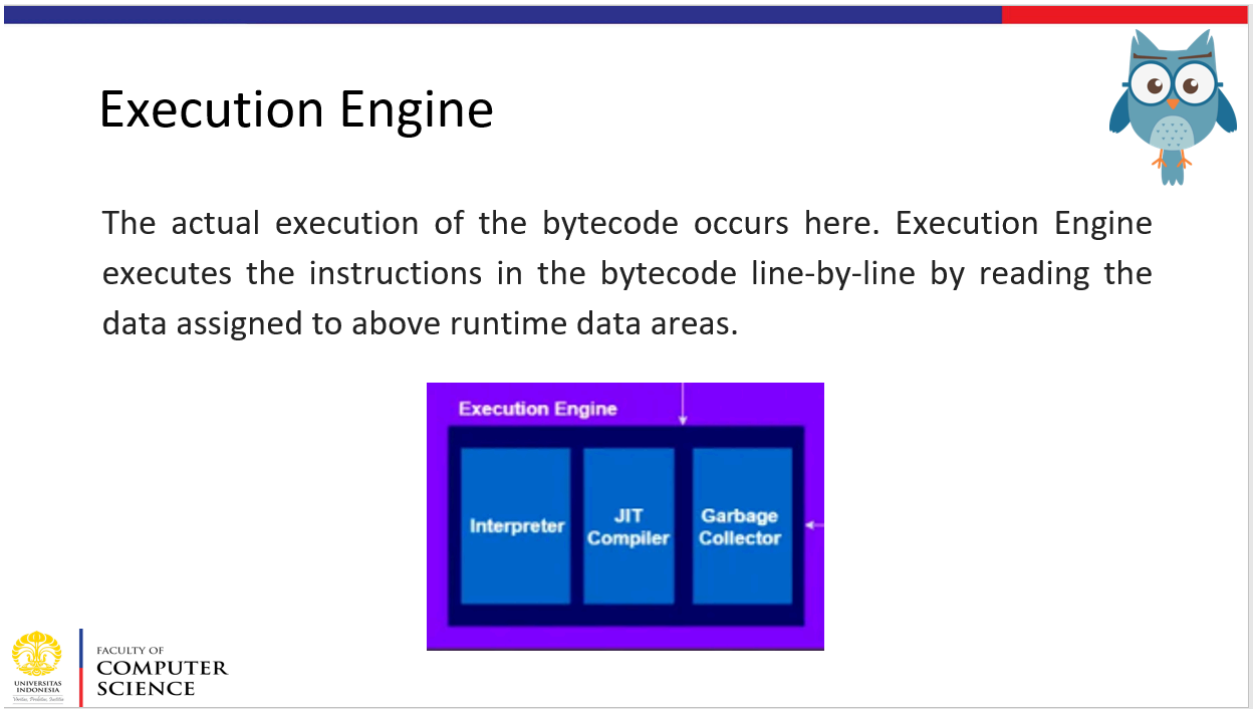
1. Method Area (Shared): This component is a central repository storing class-level information such as static variables, method data, and the runtime constant pool. Given its shared nature among all JVM threads, access to the Method Area is meticulously synchronized to ensure thread safety. This area is fundamental in holding the blueprint of classes, enabling the JVM to instantiate objects and execute class methods accurately.
2. Heap Area (Shared): The Heap Area is the JVM's memory pool containing all object instances and arrays created by a Java application. It is the focal point of Garbage Collection (GC), a critical process for reclaiming unused memory, thereby preventing memory leaks and ensuring efficient memory utilization. Although shared across threads, this area is not inherently thread-safe, necessitating explicit synchronization in multi-threaded scenarios to avoid data corruption.



3. Stack Area (Thread-Specific): Each JVM thread possesses its runtime stack, a linear collection of stack frames. These frames store local variables, operand stacks, and method references, providing a thread-safe environment for method calls, parameters handling, and execution of intermediate operations. The Stack Area is pivotal for managing method invocations and returns, maintaining a clear execution path for each thread.
4. PC Registers (Thread-Specific): Program Counter (PC) Registers are critical for the JVM's execution engine, holding the address of the currently executing instruction for each thread. This ensures that the JVM can accurately track the progression of thread execution, maintaining an orderly flow of execution and facilitating context switching between threads.
5. Native Method Stack (Thread-Specific): This component handles the execution of native methods, which are methods implemented in a language other than Java (typically C or C++), through the Java Native Interface (JNI). The Native Method Stack bridges Java applications with the underlying operating system, allowing for the execution of OS-specific functionalities and access to system resources, thereby extending the capabilities of Java applications beyond the confines of the JVM.

Together, these components form the backbone of the JVM's Runtime Data Area, orchestrating the complex interplay between Java application code, memory management, and execution control.

Execution Engine

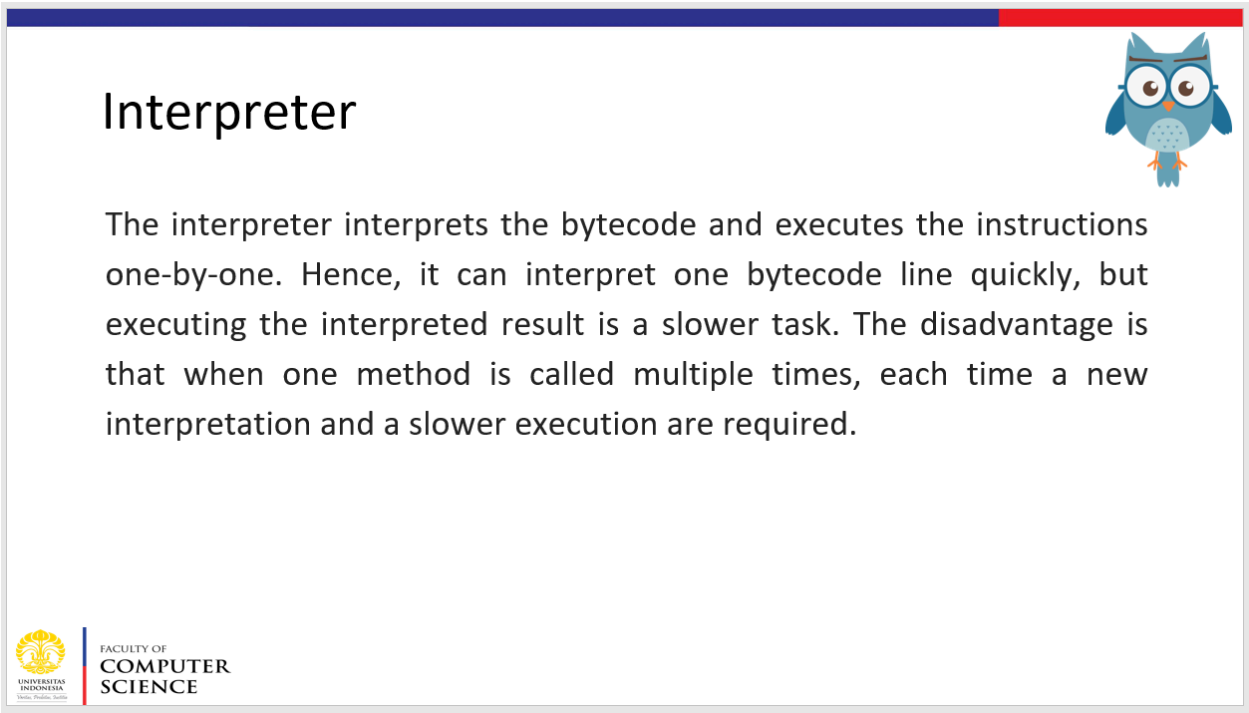


The Execution Engine is the core of the Java Virtual Machine (JVM) that brings Java applications to life by turning bytecode into actions. It operates by executing the bytecode instructions line-by-line, a meticulous process that involves interacting with the various Runtime Data Areas allocated by the JVM. As the engine reads through the bytecode, it utilizes the data stored in areas such as the Method Area, Heap, Stack, PC Registers, and Native Method Stacks to perform operations, manage memory, and execute methods.

This engine is adept at navigating the complex landscape of Java bytecode, an intermediate representation of Java code that is platform-independent. Through a combination of interpretation, Just-In-Time (JIT) compilation, and, in some implementations, Ahead-Of-Time (AOT) compilation, the Execution Engine optimizes program performance dynamically, ensuring that Java applications run efficiently on any device. It translates the abstract bytecode into concrete machine-level instructions that the host system's processor can execute, enabling the seamless execution of Java programs across varied computing environments.

In essence, the Execution Engine is the dynamic force that enables the JVM to execute Java applications with precision and efficiency, leveraging the structured data within the Runtime Data Areas to deliver a robust, platform-independent execution environment. This intricate process underscores the JVM's capability to provide a consistent runtime experience, embodying the essence of Java's write once, run anywhere philosophy.

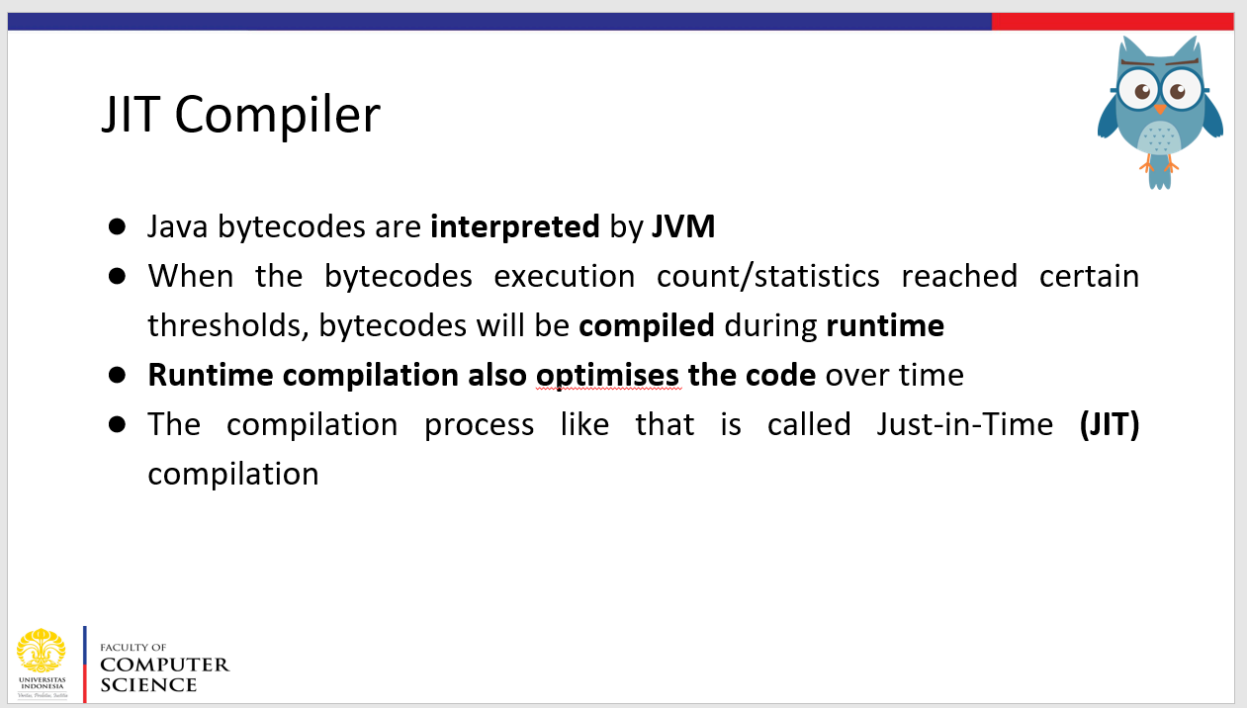
Interpreter



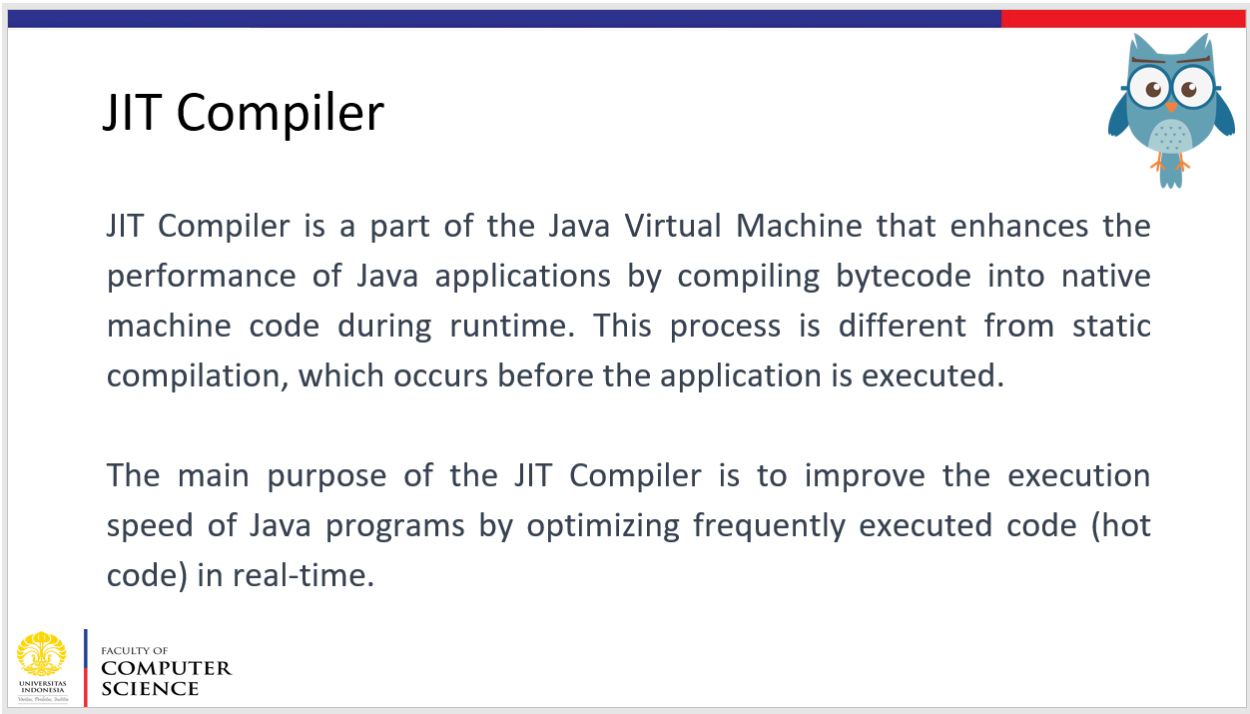
The Interpreter within the JVM's Execution Engine plays a pivotal role in the execution of Java programs by sequentially interpreting bytecode instructions. This component of the JVM ensures that Java applications can start running quickly by translating one bytecode line at a time into machine code, making it possible to execute Java programs on a wide variety of hardware platforms without the need for recompilation. However, this approach has its drawbacks, primarily in terms of execution speed. While the Interpreter can quickly understand and begin executing bytecode, its one-by-one processing model results in slower overall performance, especially noticeable when a method is invoked multiple times.

Each time a method is called, the Interpreter reinterprets the bytecode from scratch, leading to repetitive processing and inefficiency. This characteristic is particularly disadvantageous for performance-critical applications that rely on frequent method calls. Such repeated interpretation not only slows down execution but also increases the CPU's workload, contrasting with compiled languages where methods, once compiled, run directly as machine code. This highlights a critical area within the JVM where optimization techniques, like Just-In-Time (JIT) compilation, become essential to enhance performance by compiling frequently called methods into machine code, thus bypassing the need for repeated interpretation and significantly speeding up execution.

JIT Compiler

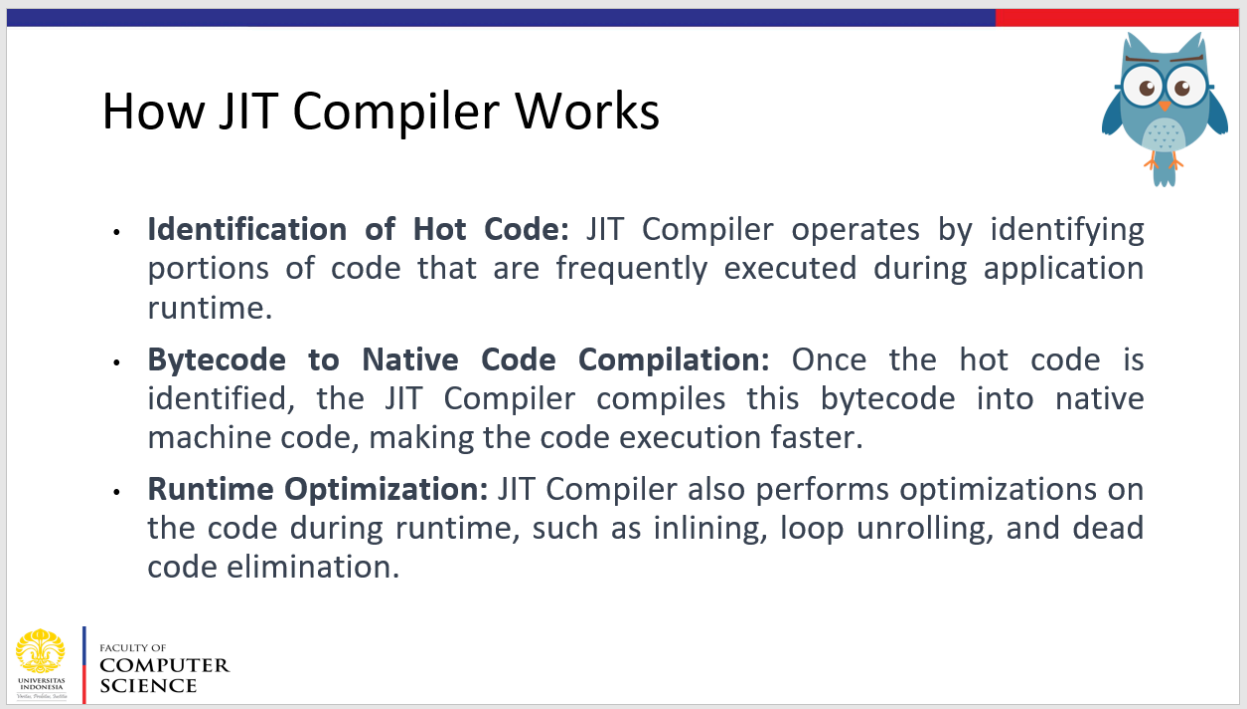


Within the Java Virtual Machine (JVM), a sophisticated mechanism known as the JIT (Just-in-Time) Compiler plays a pivotal role in optimizing the execution of Java applications. Initially, the JVM interprets Java bytecodes, a process that ensures Java's celebrated platform independence and facilitates immediate execution across various environments. However, this method of execution, while versatile, can be less efficient for code that is executed repeatedly. Recognizing this, the JIT Compiler intervenes when certain execution count or performance statistics thresholds are met, indicating that specific portions of bytecode are being frequently executed. At this juncture, it compiles those bytecodes into native machine code directly during runtime. This strategic compilation not only bypasses the overhead associated with repeated interpretation but also implements advanced code optimizations based on actual runtime performance. By integrating these runtime compilations and optimizations at strategic points, the JIT Compiler significantly enhances the execution speed and efficiency of Java applications, seamlessly marrying the interpretive flexibility of the JVM with the raw performance benefits of native code execution,



The Just-in-Time (JIT) Compiler is an essential component of the Java Virtual Machine (JVM), designed to enhance the performance of Java applications by converting bytecode into native machine code at runtime. This contrasts with static compilation, which translates code before execution, locking it to a specific platform. The JIT Compiler dynamically improves Java programs' execution speed by focusing on "hot code," or code segments frequently executed, optimizing them in real-time. This process allows Java applications to achieve high performance while maintaining platform independence. By compiling and optimizing hot code on-the-fly, the JIT Compiler ensures efficient program execution, tailoring optimizations to the application's actual usage patterns. This dynamic approach enables Java applications to run faster and more efficiently, significantly narrowing the performance gap between interpreted bytecode and native applications.

How JIT Compiler Works



The Just-in-Time (JIT) Compiler enhances Java application performance through a sophisticated process that unfolds in real-time during application execution. Initially, the JIT Compiler identifies "hot code," which are segments of code executed frequently, a key factor that signals the need for optimization. This identification is crucial as it allows the JIT Compiler to focus its resources efficiently, targeting the optimization of code that most impacts performance.

Following the identification of hot code, the JIT Compiler proceeds to compile this bytecode into native machine code. This step is pivotal, transforming the platform-independent bytecode into platform-specific machine code, which can be directly executed by the CPU. This direct execution significantly speeds up the application by eliminating the need for bytecode interpretation during these critical sections of code.

Moreover, the JIT Compiler doesn't stop at mere compilation; it also implements a series of runtime optimizations on the identified hot code. These optimizations include inlining, which integrates the code of frequently called methods directly into their call sites; loop unrolling, which reduces the overhead of looping constructs; and dead code elimination, which removes code segments that do not affect the program's outcome. Through these targeted optimizations, the JIT Compiler ensures that Java applications run with enhanced efficiency and speed, leveraging the dynamic nature of runtime compilation and optimization to deliver superior performance.

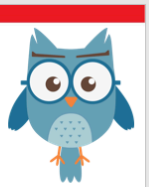
Java Garbage Collector

The Java Garbage Collector (GC) is a crucial component of the Java Virtual Machine (JVM), designed to automate memory management and optimize application performance. Its primary function is to oversee memory allocation for Java objects and to identify and dispose of those objects that are no longer in use by the application. This process of identifying unused objects, also known as garbage collection, helps in reclaiming memory space, thereby preventing memory leaks and ensuring the efficient use of resources.

Operating in the background, the GC continuously monitors object references within the JVM, using various algorithms to determine which objects are unreachable or no longer needed. Once an object is identified as garbage, the GC reclaims the heap space it occupied, making this space available for new objects. This automatic memory management relieves developers from the manual task of memory allocation and deallocation, reducing the risk of errors and enhancing application stability and performance.

By efficiently managing memory allocation and freeing up unnecessary memory resources, the Java Garbage Collector plays a vital role in ensuring that Java applications run smoothly, without running out of memory or experiencing slowdowns due to memory bloat. Its integration into the JVM architecture embodies Java's commitment to providing a robust platform for developing high-performance applications with minimal overhead for developers.

Java Garbage Collector

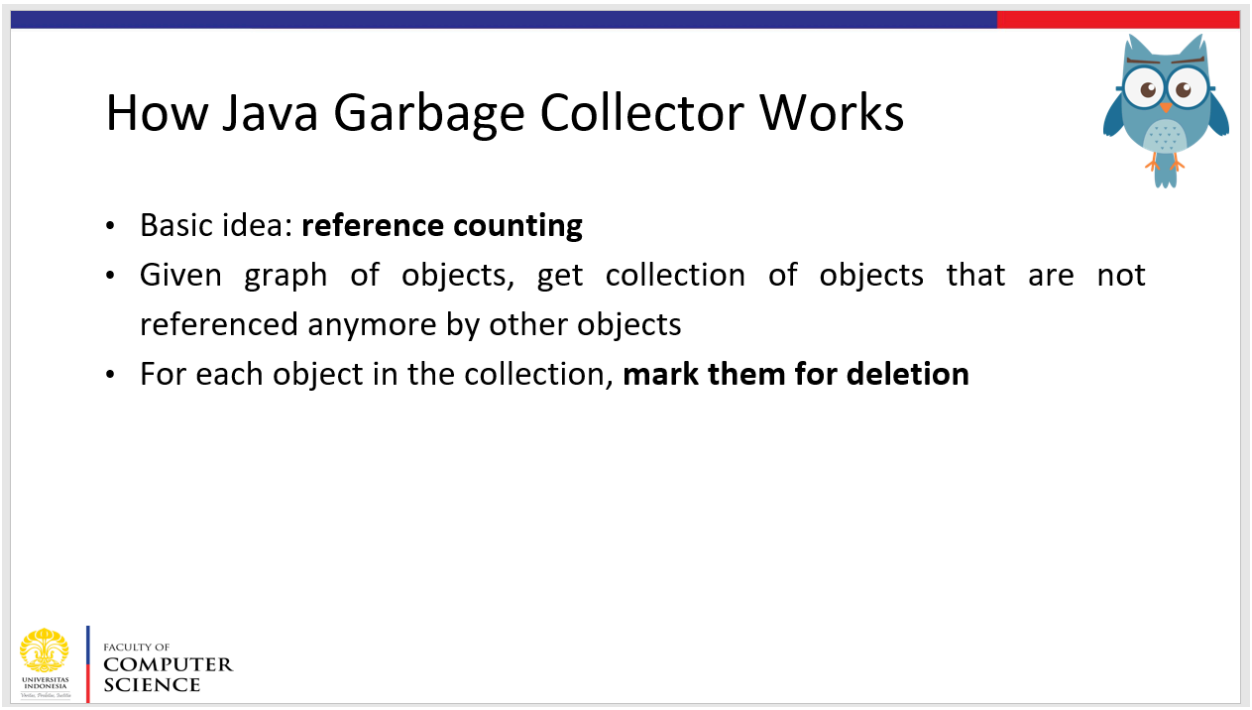


- Objects are instantiated and referenced during runtime
- At some point, some objects will get unused
  - E.g. 'deleting' a node in linked list, data objects that have been serialised into persistence
- Unused objects will accumulate in the memory **if they are not cleaned up**

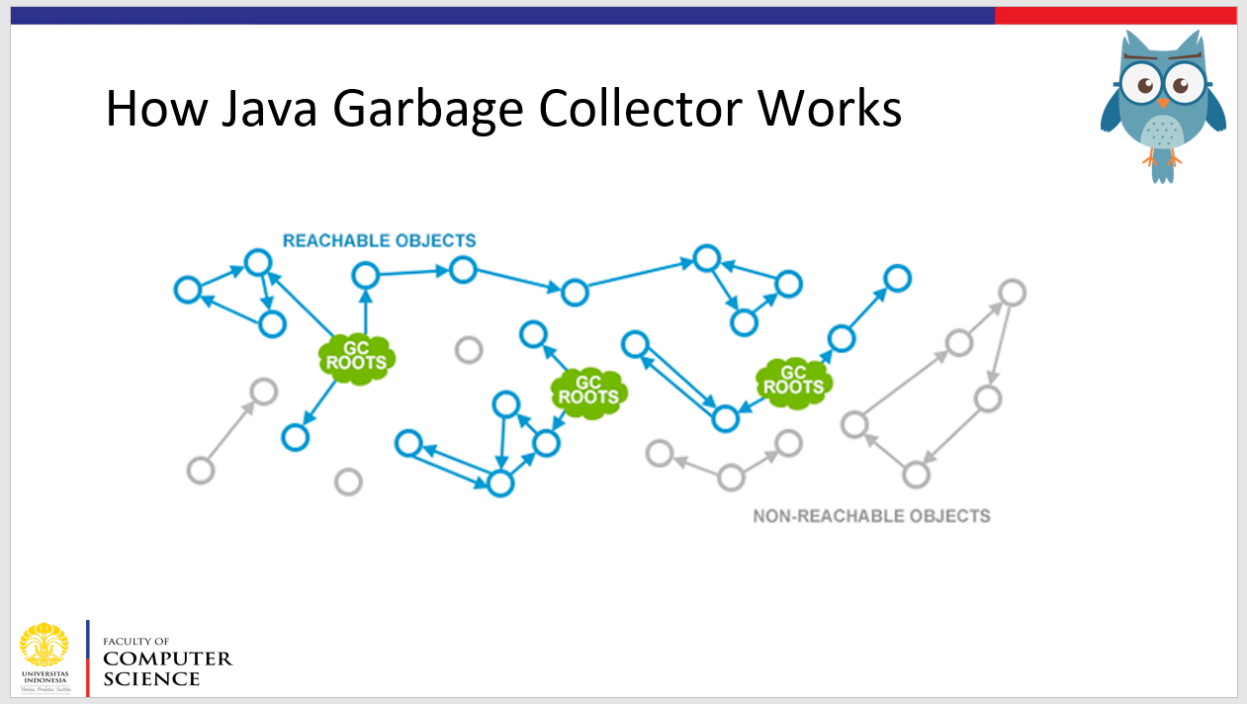
FACULTY OF COMPUTER SCIENCE

In Java, objects are dynamically created and utilized during runtime, serving various purposes from data storage to controlling the flow of application logic. However, as the application evolves, certain objects become obsolete—for instance, when a node is removed from a linked list or data objects are serialized into a persistent format. These unused objects, if left unchecked, begin to accumulate in memory, leading to inefficient memory use and potential memory leaks. The Java Garbage Collector (GC) plays a pivotal role in addressing this issue by automatically identifying and removing these objects from memory. By clearing unused objects, the GC ensures that memory is efficiently managed and available for newly instantiated objects, maintaining the application's performance and stability. This automated process is a cornerstone of Java's approach to memory management, allowing developers to focus on application logic without worrying about the underlying memory allocation details.

How Java Garbage Collector Works

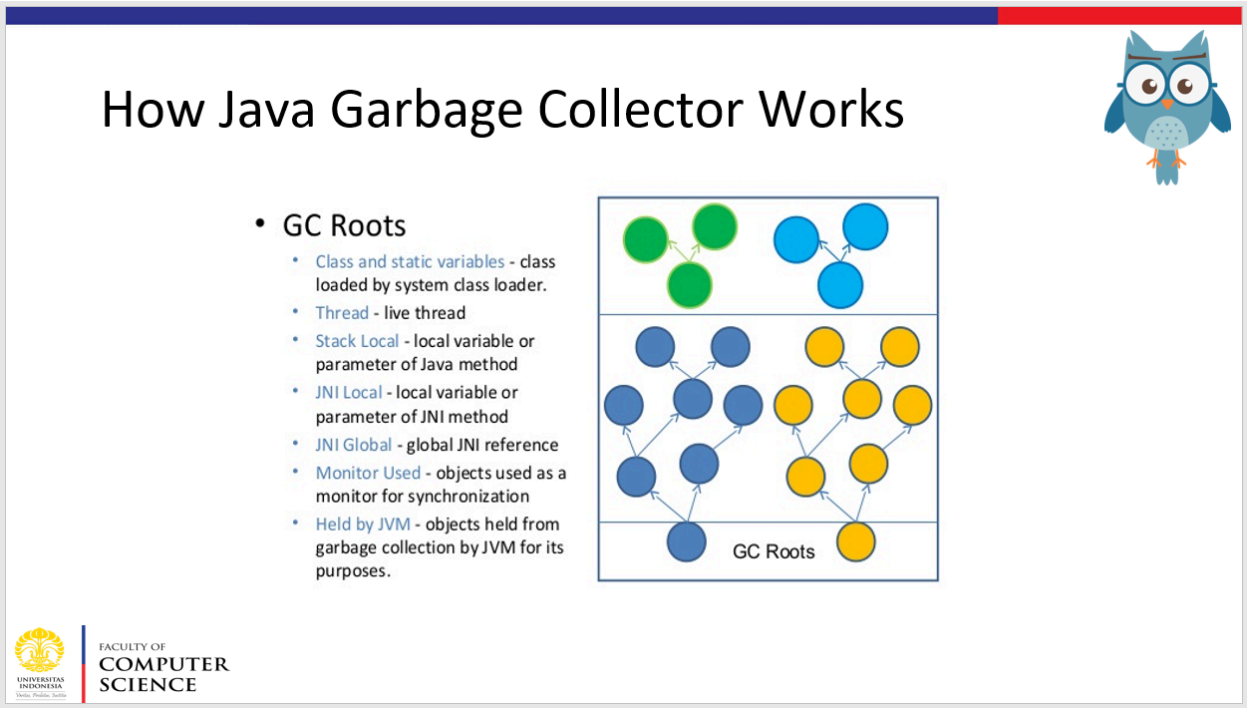


The Java Garbage Collector (GC) operates on a fundamental principle of managing memory through the identification and elimination of objects that are no longer needed by a Java application. At its core, one of the basic techniques it uses is reference counting, which involves tracking how many references exist to each object in memory. The process begins by constructing a graph of objects that maps out the relationships between them, highlighting how they reference each other.



In this graph, certain objects will inevitably become unreferenced over time due to the application's evolution, such as objects falling out of scope or explicitly being set to null. These unreferenced objects are the primary targets for garbage collection, as they are deemed no longer necessary for the application's functionality. The GC meticulously scans through the graph to collect a subset of objects that are not referenced by any other objects in the application. Once identified, these objects are marked for deletion.

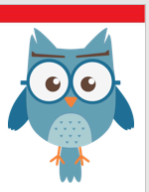
The marking phase is crucial as it precisely identifies which objects in the heap are eligible for garbage collection, ensuring that only unused objects are removed, thereby preventing any accidental deletion of objects still in use. Following the marking, the actual collection process begins, where the marked objects are removed, and the memory they occupied is reclaimed. This reclaimed memory is then made available for new objects, thereby optimizing the application's memory usage and enhancing performance. The Java Garbage Collector's ability to automate this process of memory management is a key advantage, allowing developers to concentrate on application logic rather than the complexities of memory allocation and deallocation.



The Java Garbage Collector (GC) operates through a sophisticated mechanism designed to efficiently manage memory within the Java Virtual Machine (JVM). At the heart of this process are the Garbage Collector Roots, which serve as the starting points for the GC to track and manage object references. These roots include class and static variables, threads, stack local variables, JNI (Java Native Interface) local and global references, monitors used in synchronized blocks, and objects held by the JVM itself.

The GC begins its task by identifying objects that are reachable through a chain of references from these roots. Any object that cannot be reached in this manner is considered eligible for garbage collection, implying it's no longer in use and its memory can be reclaimed. This approach ensures that only objects with no remaining ties to the running application are marked for deletion, preventing active objects from being accidentally collected. By systematically tracing references from GC roots, the Java Garbage Collector efficiently identifies unused objects, clears them from memory, and thus helps in maintaining optimal application performance and memory usage. This process is critical for the continuous, smooth operation of Java applications, allowing them to run without manual memory management interventions by the developer.

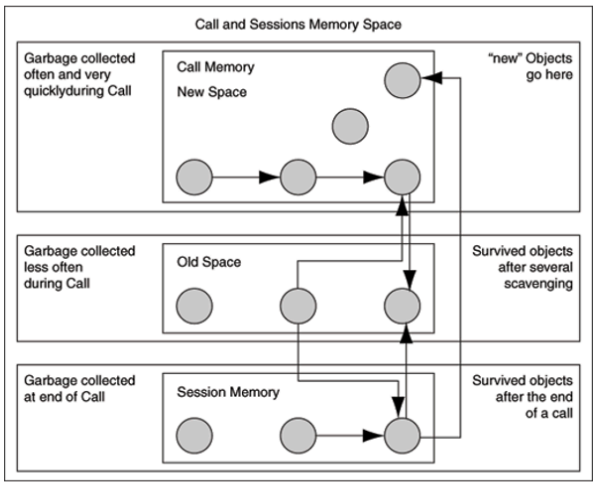
Generational Garbage Collection



- Java uses different approach in GC
- JVM organised heap (i.e. where objects live in memory) into three generations: **Young, Old, Permanent** generations
- Long-lived objects can get promoted to the next generation

FACULTY OF COMPUTER SCIENCE

Generational Garbage Collection

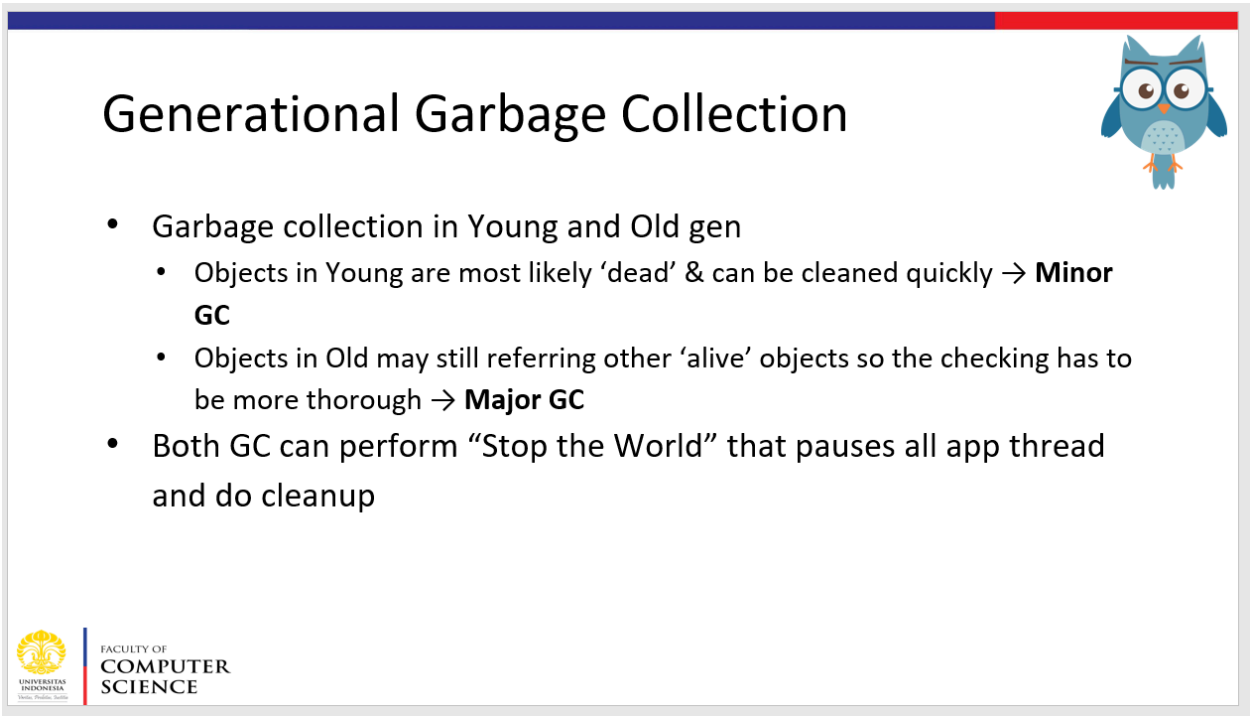


FACULTY OF COMPUTER SCIENCE

Java adopts a unique approach to Garbage Collection (GC) by structuring the JVM heap, the memory area where objects reside, into three distinct generations: Young, Old, and Permanent. This method, known as Generational Garbage Collection, is based on the observation that most objects are short-lived and can be collected quickly after their use is complete, while others may need to remain in memory for a longer duration.

The Young Generation, or "new space," is where new objects are allocated. Most objects die here, and the GC can reclaim their memory efficiently. Objects that survive the garbage collection process in the Young Generation are then promoted to the Old Generation, indicating that they have longer lifetimes. This space is collected less frequently, reflecting the stability of its objects. The Permanent Generation, although not directly part of the generational collection in later versions of Java, traditionally held metadata describing the classes and methods used in the application.

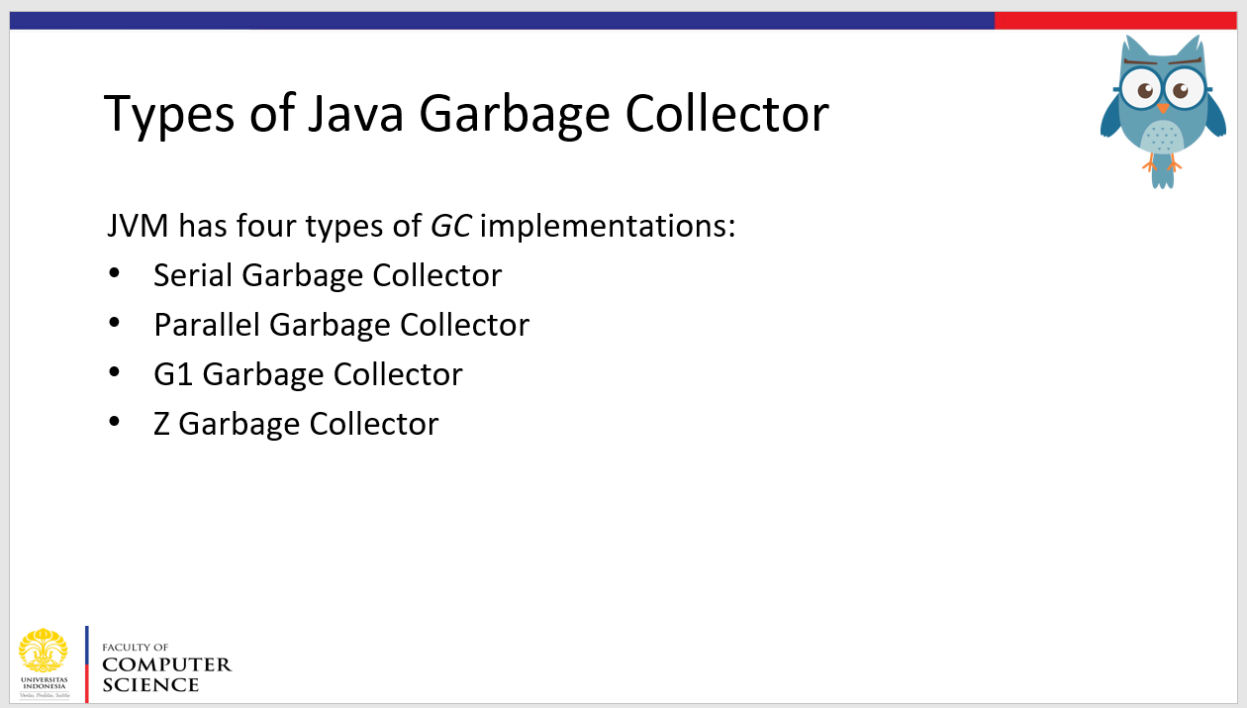
Garbage collection in Java is optimized by this division, allowing the JVM to focus its efforts on the areas with the most garbage to collect, typically the Young Generation, and use more intensive GC methods on the Old Generation less frequently. This generational approach significantly improves the efficiency of memory management, ensuring that short-lived and long-lived objects are handled appropriately according to their lifetimes, ultimately enhancing application performance and reliability.



Generational Garbage Collection in Java strategically divides the JVM heap into Young and Old generations to optimize memory management and improve application performance. This method hinges on the premise that objects in the Young generation, where new objects are allocated, are often short-lived and thus can be collected more swiftly and with less overhead—this process is known as Minor Garbage Collection (GC). As objects survive Minor GC, they are promoted to the Old generation, indicating a longer life expectancy. Here, objects are less frequently collected due to the assumption that they are more likely to be in use, making the garbage collection process, referred to as Major GC, more comprehensive and thorough.

Both Minor and Major GC operations have the potential to initiate a "Stop the World" event, where all application threads are temporarily paused to allow the GC to perform the necessary cleanup tasks without interference. This pause is crucial for the GC to accurately determine which objects are no longer in use and can be safely reclaimed. Although "Stop the World" events can impact application responsiveness, they are essential for freeing up memory and preventing memory leaks. By employing Generational Garbage Collection, Java balances the need for efficient memory management with the goal of minimizing disruption to the application's execution, ensuring that both short-lived and long-lived objects are handled appropriately.

Types of Java Garbage Collector



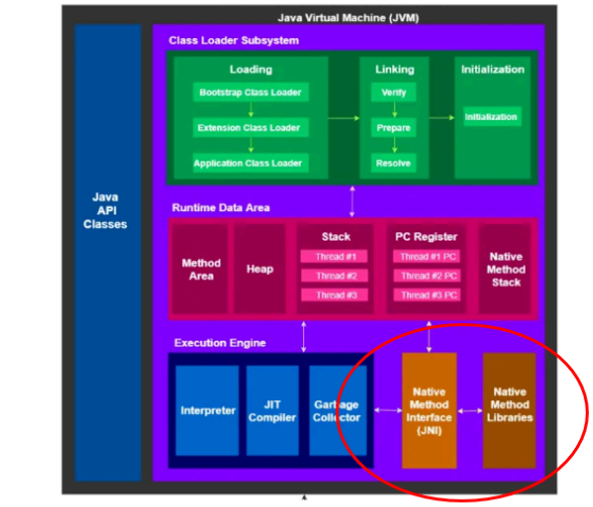
The Java Virtual Machine (JVM) offers four distinct implementations of the Garbage Collector (GC) mechanism, each designed to cater to different types of applications and system configurations, optimizing garbage collection in ways that best suit the operational requirements and goals of an application.

1. Serial Garbage Collector: This is the simplest form of GC, designed for single-threaded environments. It operates by pausing all application threads (a "Stop the World" event) while it collects garbage, making it suitable for applications with smaller datasets or those running on single-core machines where minimal pause times are not critical.
2. Parallel Garbage Collector: Also known as the Throughput Collector, it utilizes multiple threads for garbage collection, significantly speeding up the GC process compared to the Serial Garbage Collector. It is best suited for multi-threaded applications and servers with multiple processors, aiming to maximize application throughput by utilizing available CPU cores for garbage collection.
3. G1 Garbage Collector: The Garbage-First Collector is designed for applications with large heaps, focusing on minimizing pause times by predicting which set of garbage collection regions will be most efficient to collect. It divides the heap into regions and collects those that are most full, allowing for more predictable garbage collection times and improving application performance in large-scale environments.

4. Z Garbage Collector: The ZGC is a scalable, low-latency garbage collector designed to handle large heaps with minimal pause times, regardless of the heap size. It aims to limit GC pause times to not exceed a few milliseconds by using load barriers and colored pointers for memory management, making it ideal for applications where consistent low latency is critical.

Each of these GC implementations offers unique advantages, allowing developers to choose the most appropriate garbage collection strategy based on the specific needs of their Java applications, whether they prioritize throughput, minimal pause times, or scalability.

JNI & Native Method Libraries



FACULTY OF COMPUTER SCIENCE

JNI & Native Method Libraries

JNI

This interface is used to interact with Native Method Libraries required for the execution and provide the capabilities of such Native Libraries (often written in C/C++). This enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

Native Method Lib

Collection of C/C++ Native Libraries which is required for the Execution Engine and can be accessed through the provided Native Interface.

FACULTY OF COMPUTER SCIENCE

The Java Native Interface (JNI) is a critical framework within the Java ecosystem, enabling Java applications to interact seamlessly with native code written in other programming languages, such as C or C++. This interface is pivotal for extending the capabilities of the Java Virtual

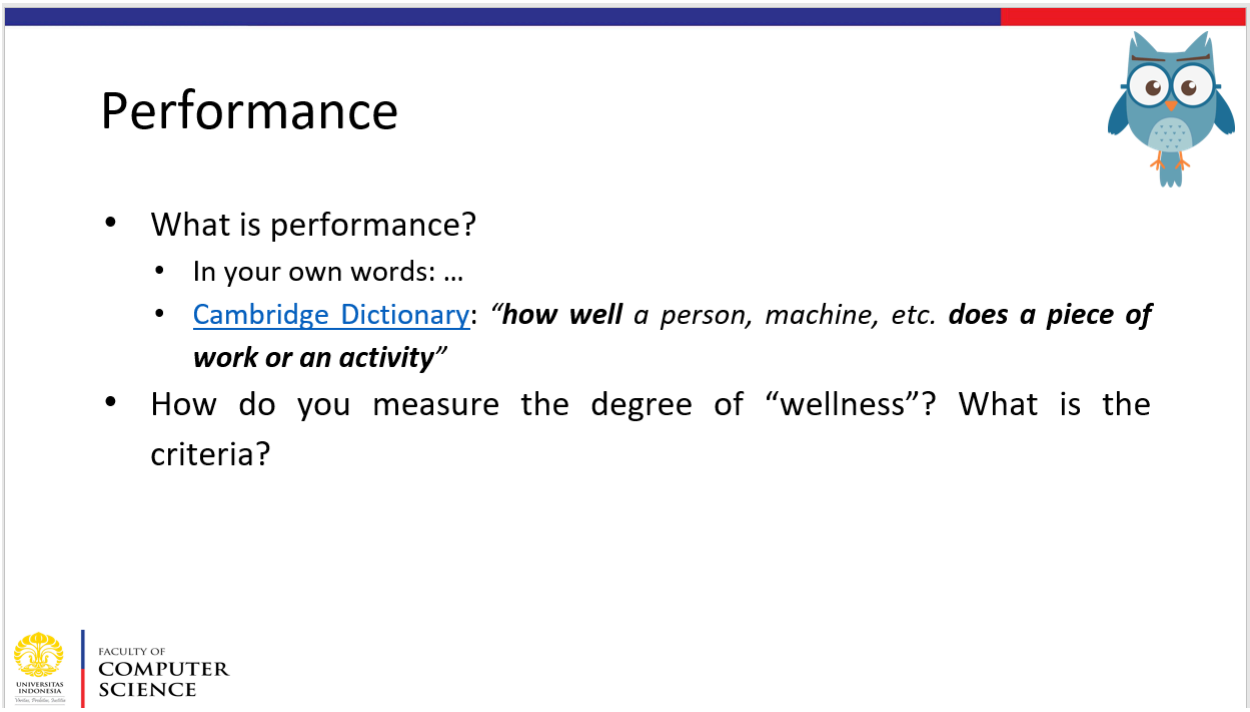
Machine (JVM) beyond its standard library, allowing for the execution of tasks that Java alone might not be able to perform efficiently or at all. JNI acts as a bridge, facilitating communication between the JVM and native method libraries, which are collections of functions written in languages like C or C++.

Native method libraries encompass a wide array of C/C++ libraries required by the Execution Engine of the JVM. These libraries can include functionalities that are either too low-level for Java to handle directly, such as direct memory management or system calls, or that require close integration with the underlying hardware, like graphical rendering or hardware-accelerated computation. By leveraging JNI, Java applications can call functions from these native libraries, thus inheriting their efficiency and capabilities. Conversely, it also permits native applications to invoke Java methods, providing a bidirectional communication channel between Java and native code.

The integration of JNI within Java applications is not without its complexities. Developers must be mindful of the differences in memory management and data type conventions between Java and native languages. Java operates within a garbage-collected environment, whereas C/C++ requires explicit memory management. Additionally, data types in Java may not have direct equivalents in C/C++, necessitating careful data translation and marshalling to maintain type safety and prevent memory leaks or application crashes.

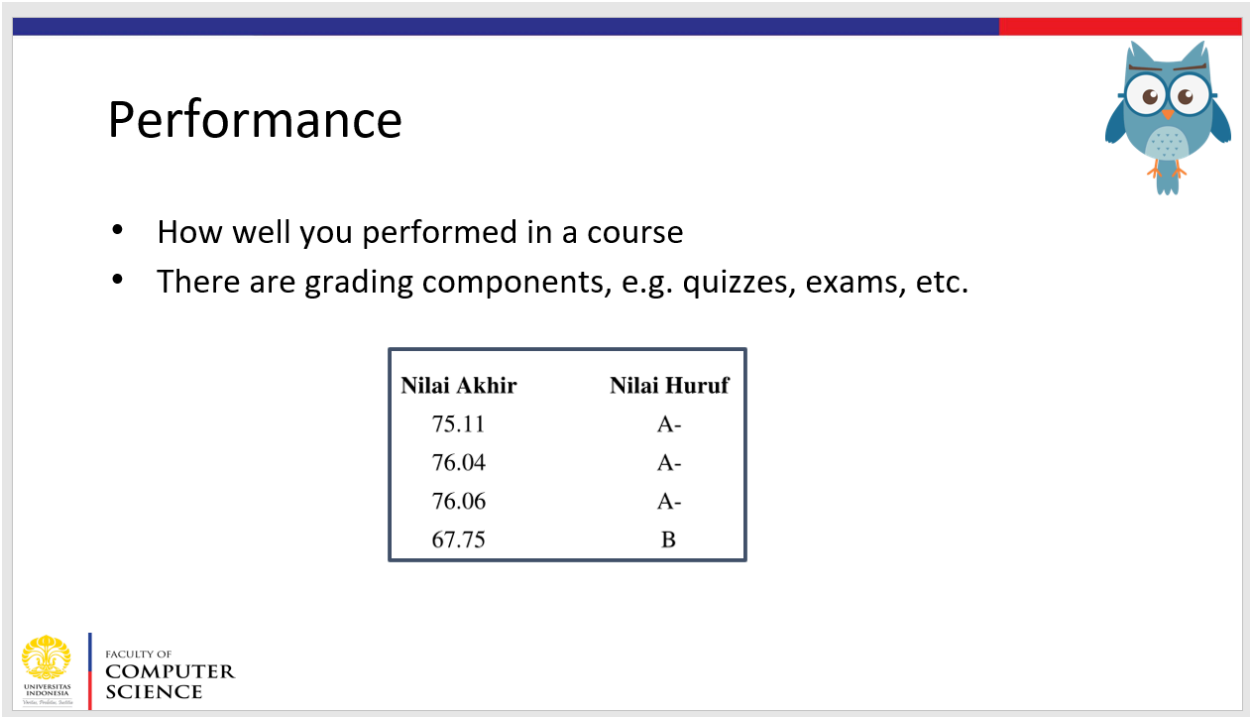
Despite these challenges, JNI remains an invaluable tool for Java developers, enabling the creation of high-performance applications by combining the platform independence and ease of use of Java with the speed and power of native libraries. This symbiosis allows developers to extend the reach of Java applications into areas traditionally dominated by lower-level languages, from high-performance computing to real-time system interaction, thereby significantly enhancing the versatility and capability of Java-based solutions.

Performance Issues



Performance, in a broad sense, encompasses the effectiveness and efficiency with which a person, machine, or system completes tasks or activities. It's a measure of how well objectives are achieved, often evaluated against a set of standards or expectations. In the context of computing, performance can be quantified through various metrics, such as speed (how fast a task is completed), throughput (the amount of work done in a given period), and resource utilization (how effectively the system uses its resources like CPU, memory, and disk).

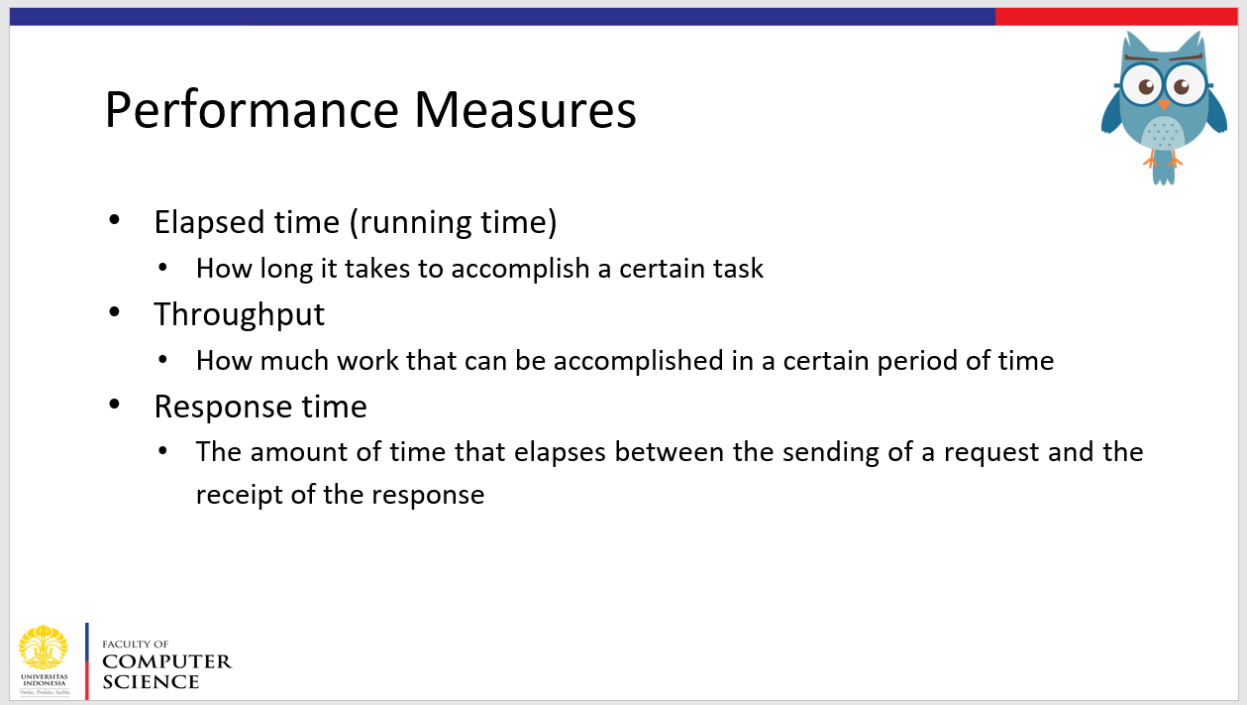
Measuring the degree of "wellness" or performance involves setting specific criteria that are relevant to the task or system at hand. For software applications, this could include response time for user actions, the latency of data processing tasks, and the number of concurrent users the system can support without degradation in service. The criteria chosen depend on the goals of the system and the expectations of its users. By establishing clear benchmarks for performance, developers and administrators can identify areas for improvement and optimize the system to better meet its objectives, ensuring a smoother and more efficient operation.



In the context of system performance, the analogy of grading in a course can be aptly applied. Just as a student's performance is assessed through quizzes, exams, and other components, a system's performance is evaluated through various metrics like response time, throughput, and resource utilization. Each metric, akin to a grading component, offers insights into different aspects of the system's efficiency and effectiveness. For example, response time can be likened to quiz scores, indicating how quickly the system handles tasks; throughput might be compared to exam scores, reflecting the volume of work it can process over time; and resource utilization could parallel participation grades, showing how well the system manages its computational resources.

The final scores (75.11, 76.04, etc.) and corresponding alphabet scores (B+, A, B-) in the educational analogy can correlate to performance benchmarks or targets set for the system. Achieving an 'A' might represent surpassing performance expectations, a 'B+' could indicate satisfactory performance, and a 'B-' might signal areas needing improvement. This approach allows administrators and developers to evaluate where the system excels and where it falls short, guiding optimizations and adjustments to enhance overall performance, similar to how a student would focus on improving in areas where their grades indicate room for improvement.

Performance Measures



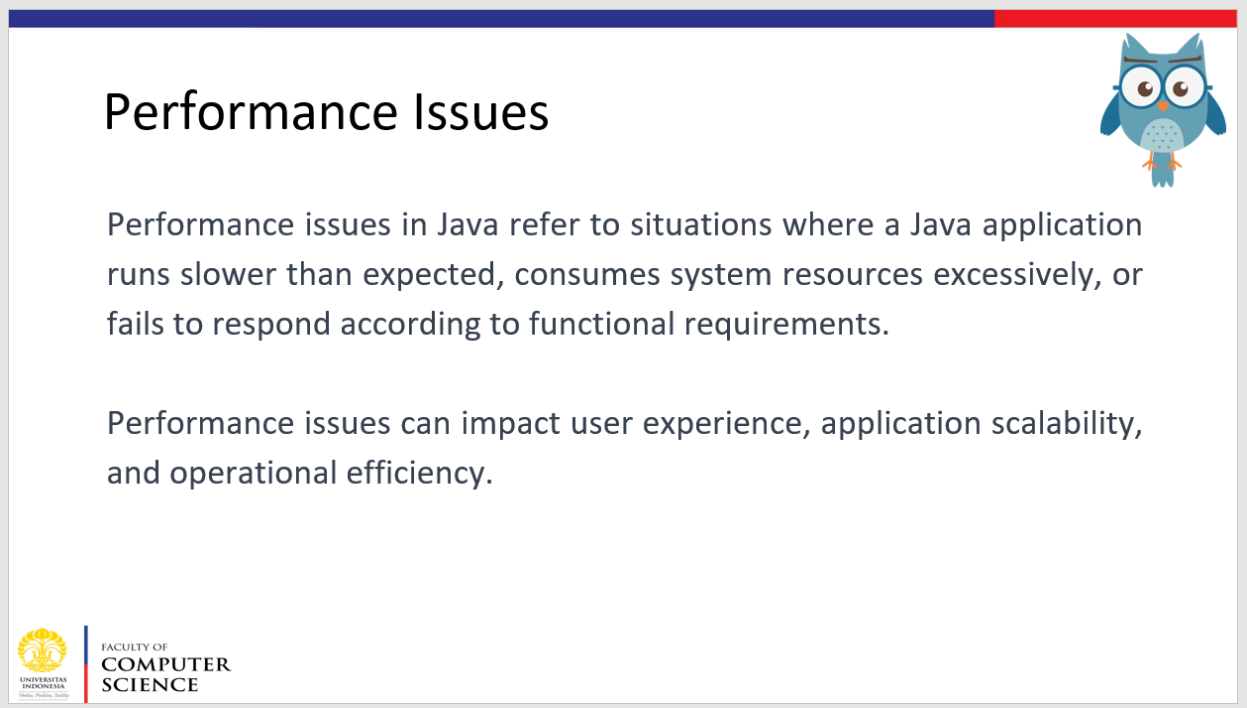
Performance measures are crucial metrics in evaluating the efficiency and effectiveness of a system, offering insights into various aspects of its operation. Elapsed time, or running time, is a measure of the duration taken to complete a specific task. This metric is vital in scenarios where time efficiency is critical, such as in computing tasks or processes that require quick execution to meet user expectations or system requirements.

Throughput refers to the volume of work or transactions a system can handle within a given timeframe. It's an essential measure in environments where the capability to process a high volume of tasks efficiently is paramount, such as in data processing centers or high-traffic web services. High throughput rates indicate a system's ability to handle heavy loads effectively, making it a key performance indicator for system scalability and efficiency.

Response time measures the interval between the initiation of a request and the receipt of the corresponding response. It's particularly relevant in user-facing applications and services, where quick responses are crucial for a positive user experience. Shorter response times are often associated with higher system performance, as they reflect the system's ability to promptly process requests and deliver results, thereby enhancing user satisfaction and engagement.

Together, these performance measures provide a comprehensive overview of a system's operational capabilities, identifying areas of strength and pinpointing opportunities for optimization to ensure that the system meets its performance objectives efficiently.

Performance Issues

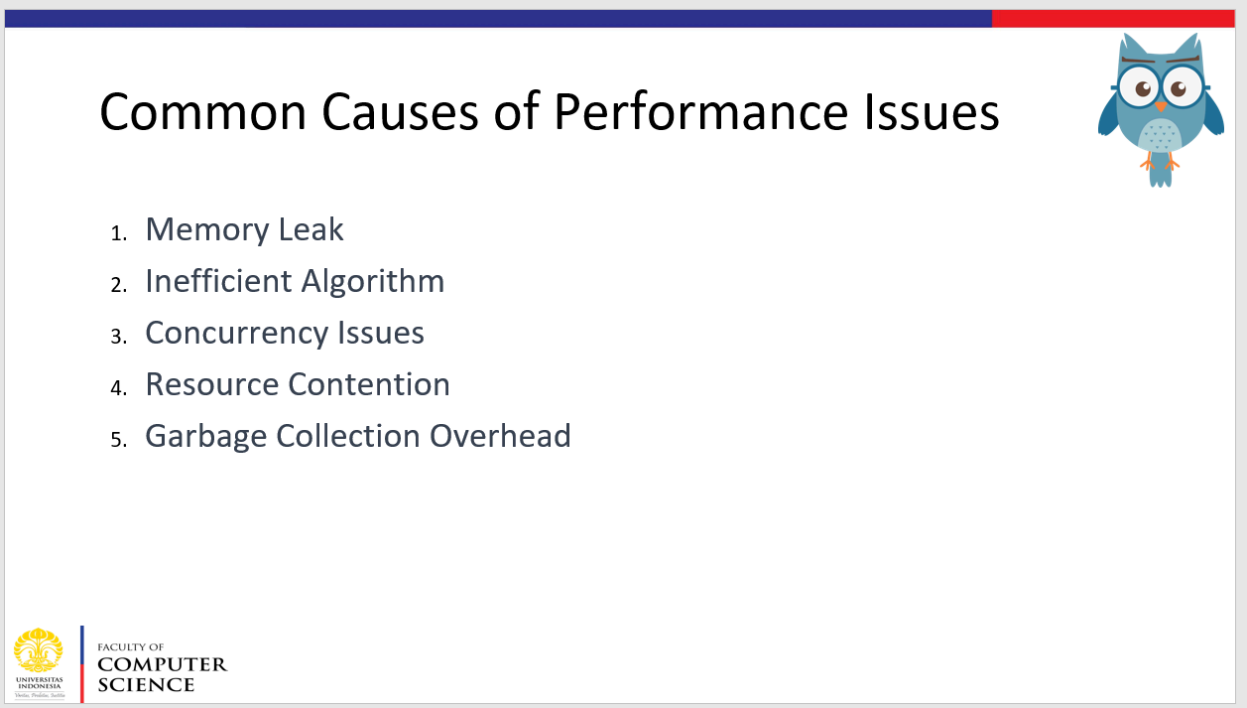


Performance issues in Java applications are critical concerns that can significantly hinder their effectiveness, scalability, and user satisfaction. These issues manifest when an application operates slower than anticipated, excessively utilizes system resources, or does not meet its functional responsiveness requirements. Such problems can arise from various factors, including inefficient code algorithms, inadequate memory management, or suboptimal utilization of system resources.

The consequences of performance issues extend beyond mere operational inefficiencies. From a user perspective, slow response times or frequent application freezes can lead to frustration and decreased engagement. For businesses, these issues can affect the scalability of applications, as system resources may become overstretched as user demand increases. Additionally, operational efficiency suffers when more computing power or memory is required to maintain performance levels, leading to increased costs and potential system instability.

Addressing performance issues in Java necessitates a thorough analysis of the application's architecture, codebase, and runtime environment. Optimizing algorithms, refining garbage collection processes, and ensuring efficient use of resources are essential steps toward mitigating these issues. By proactively identifying and resolving performance bottlenecks, developers can significantly enhance the user experience, ensure application scalability, and maintain operational efficiency, thereby securing the application's success and longevity.

Common Causes of Performance Issues



Common causes of performance issues in Java applications encompass a range of problems that can severely degrade application efficiency and user experience. Memory leaks occur when objects that are no longer needed are still referenced by the application, preventing the garbage collector from reclaiming their memory. This can lead to excessive memory usage and, eventually, to application slowdowns or crashes due to out-of-memory errors.

Inefficient algorithms can also cause significant performance degradation, especially in applications that process large data sets or require complex computations. An algorithm that is poorly optimized may consume unnecessary CPU cycles, leading to slower execution times and a poor user experience.

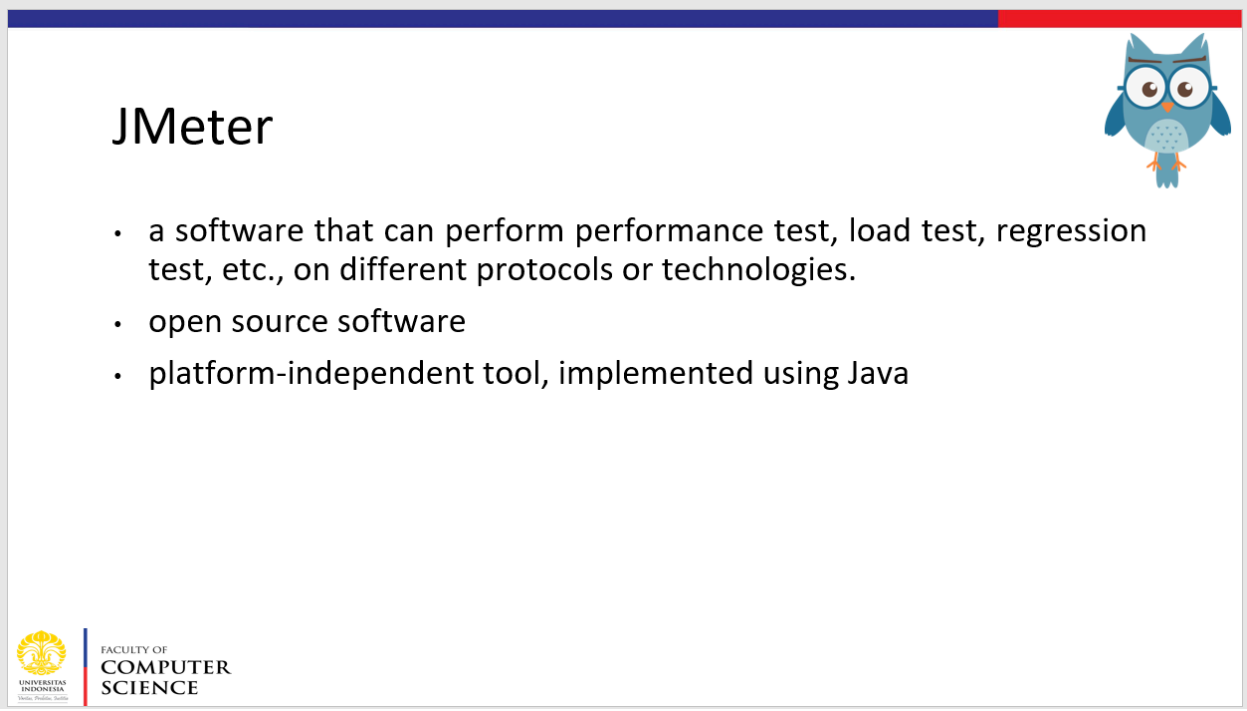
Concurrency issues arise when multiple threads access shared resources without proper synchronization, leading to race conditions, deadlocks, or inconsistent data states. These issues can complicate debugging and significantly impact application performance and reliability.

Resource contention occurs when multiple applications or processes compete for system resources, such as CPU, memory, or I/O bandwidth. This competition can throttle the performance of individual applications, leading to slower response times and reduced throughput.

Garbage collection overhead refers to the performance impact of the garbage collection process itself. While garbage collection helps manage memory automatically, its activities can pause application execution, especially if the garbage collector frequently runs or takes a long time to complete its task. Tuning the garbage collection process is often necessary to balance memory management efficiency against minimal impact on application performance.

Addressing these common causes requires a comprehensive understanding of Java application behavior, diligent code profiling, and optimization strategies to ensure applications run efficiently, scale effectively, and deliver a seamless user experience.

JMeter



JMeter is a powerful, open-source software tool designed for performance testing, load testing, regression testing, and more, across various protocols and technologies. As a platform-independent tool, it is implemented in Java, allowing it to run on any system that supports Java. This versatility makes JMeter an essential tool for developers and testers aiming to evaluate the performance of web applications, databases, and servers under different conditions and loads.

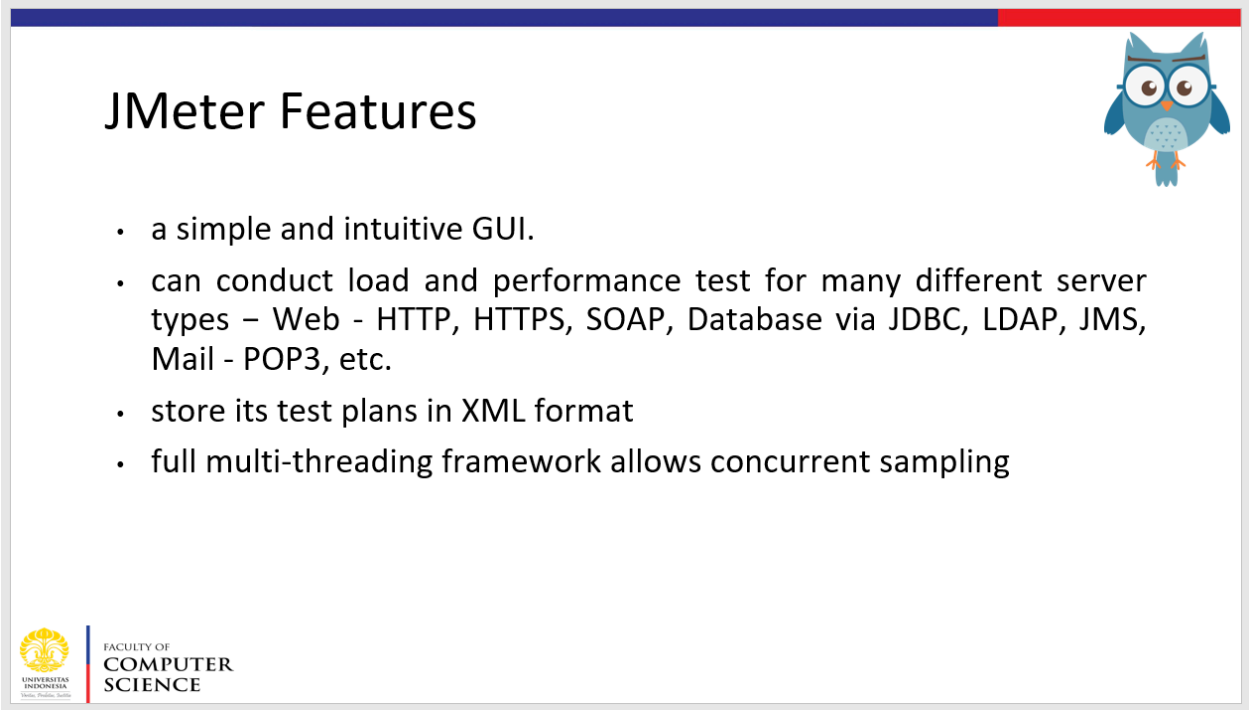
The broad testing capabilities of JMeter enable it to simulate a heavy load on a server, network, or object to test its strength or analyze overall performance under different load types. Beyond load and performance testing, JMeter is also adept at conducting regression tests, helping developers to catch bugs early by automatically testing the application's functionality after changes have been made.

Thanks to its open-source nature, JMeter has a large community of users and developers who continuously contribute to its enhancement and extend its functionality through plugins and extensions. This community support ensures that JMeter stays up-to-date with the latest technologies and testing practices.

Furthermore, its user-friendly interface allows for easy creation of test plans without the need for extensive programming knowledge. Testers can visually design tests, which can then be

executed to generate detailed reports and insights into the application's performance, scalability, and reliability.

JMeter Features



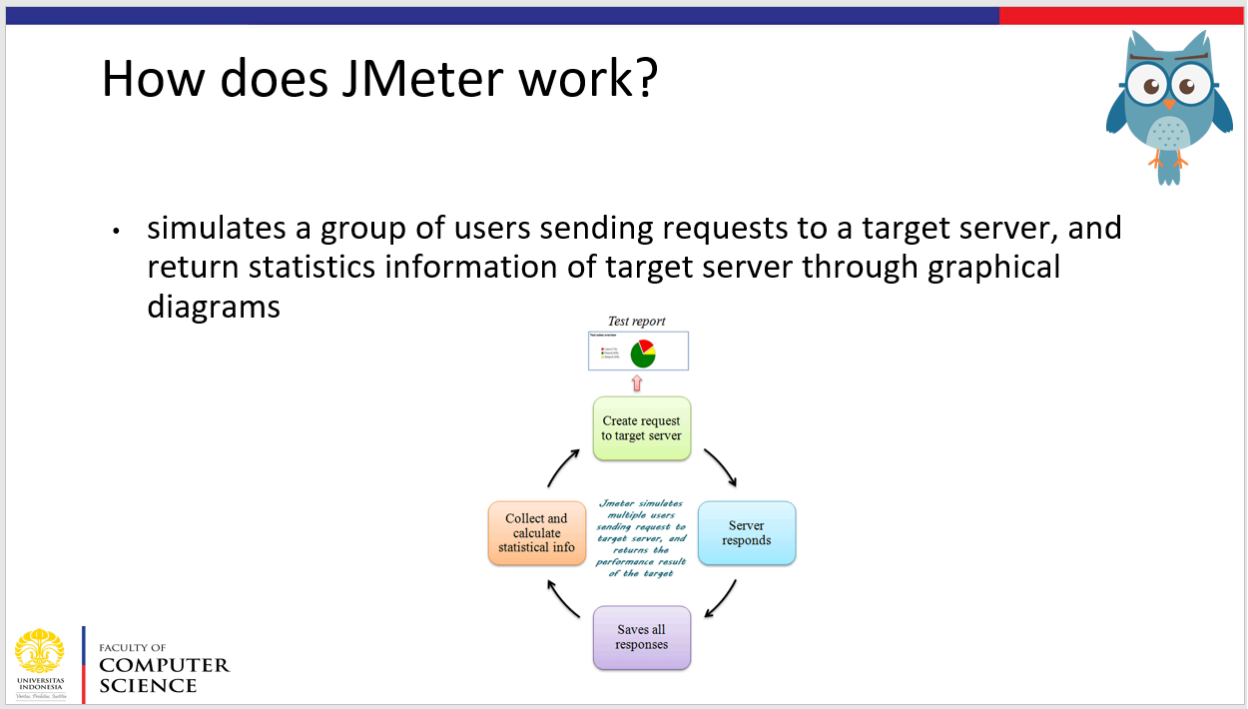
JMeter, renowned for its comprehensive testing capabilities, offers a suite of features designed to facilitate efficient and effective performance evaluations across a variety of server types and protocols. Its simple and intuitive graphical user interface (GUI) stands out, making it accessible for both experienced testers and novices. Users can easily create, configure, and execute test plans without delving into complex configurations, significantly reducing the learning curve associated with performance testing tools.

One of JMeter's strengths lies in its versatility. It can perform load and performance tests on a wide range of server types, including web servers (HTTP, HTTPS), web services (SOAP), databases (via JDBC), directory services (LDAP), messaging services (JMS), and mail servers (POP3). This capability ensures that JMeter can be used in virtually any testing scenario, from web applications to enterprise-level software systems.

JMeter stores its test plans in XML format, which not only makes them easy to read and edit but also facilitates version control and collaboration among team members. Test plans can be shared, reviewed, and modified without proprietary software, enhancing teamwork and efficiency.

The tool's full multithreading framework is another critical feature, enabling concurrent sampling by simulating multiple users or threads. This allows JMeter to accurately mimic real-world usage scenarios, providing insights into how applications behave under various levels of demand

How Does JMeter Work



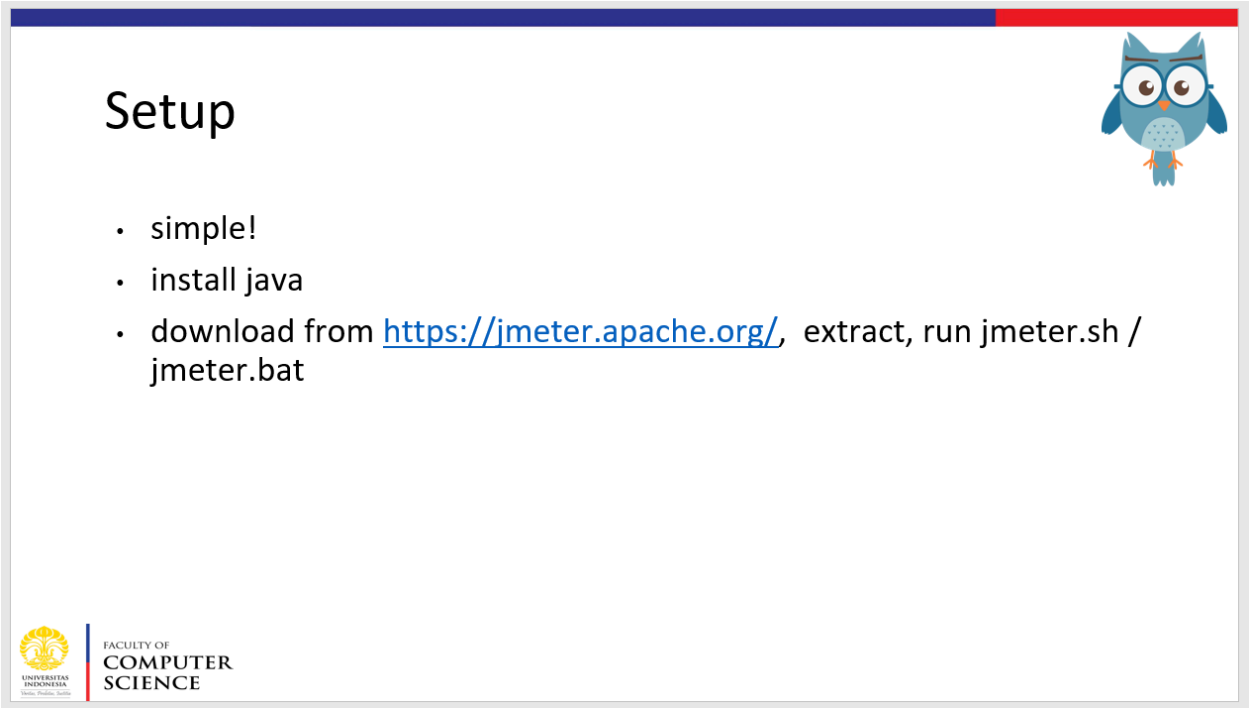
JMeter operates by simulating a group of users sending requests to a target server, mimicking the behavior of multiple users accessing a website or application simultaneously. This simulation begins with JMeter crafting requests that are sent to the server, which then processes these requests and sends back responses. JMeter captures all these responses, providing a comprehensive dataset of the server's handling of requests under varying conditions.

The power of JMeter lies in its ability to collect and analyze the data from these interactions. After capturing the responses from the server, JMeter aggregates this data to calculate statistical information, such as response times, success rates, error rates, and throughput, among other metrics. This statistical analysis is crucial for identifying bottlenecks, understanding performance under load, and determining the scalability of the application or website.

The cycle of request creation, response saving, and statistical analysis is iterative. JMeter continuously cycles through this process, adjusting the load and types of requests based on the test plan's configuration. This iterative approach allows for dynamic testing that can adapt to the results being observed in real-time.

The results of these simulations are presented through graphical diagrams, making it easier for testers and developers to visualize performance issues and improvements. These visuals aid in pinpointing specific areas that require optimization, facilitating data-driven decisions to enhance the application's performance.

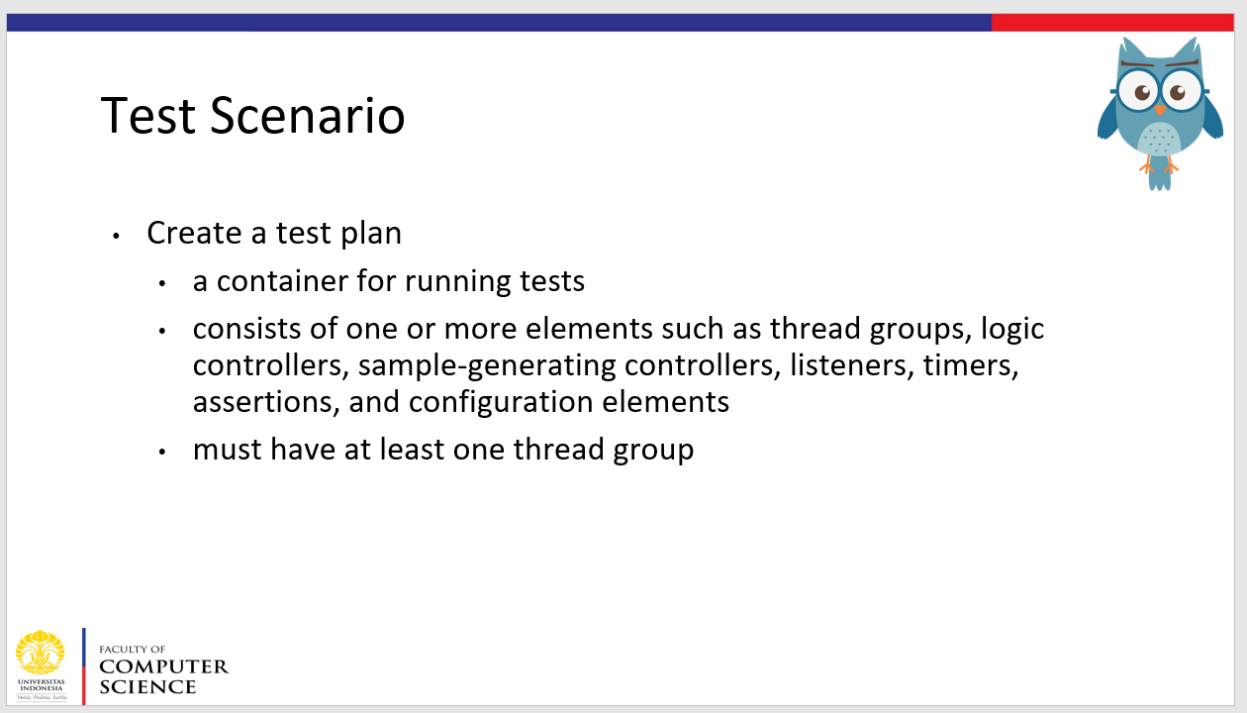
Setup



Setting up JMeter is a straightforward process, requiring minimal steps to get started with performance testing. The first step is to ensure Java is installed on your system, as JMeter is a Java-based application. Once Java is set up, the next step involves downloading JMeter from its official website, https://jmeter.apache.org/. After downloading, you simply need to extract the files from the downloaded package to a desired location on your computer.

To launch JMeter, navigate to the extracted folder and run the appropriate script for your operating system. If you are using a Unix-based system like Linux or MacOS, execute the 'jmeter.sh' script. Windows users should run the 'jmeter.bat' file. This action will start JMeter, presenting you with its graphical user interface, where you can begin creating and executing your test plans. This simplicity in setup makes JMeter an accessible tool for users looking to perform performance testing without a complex installation process.

Test Scenario



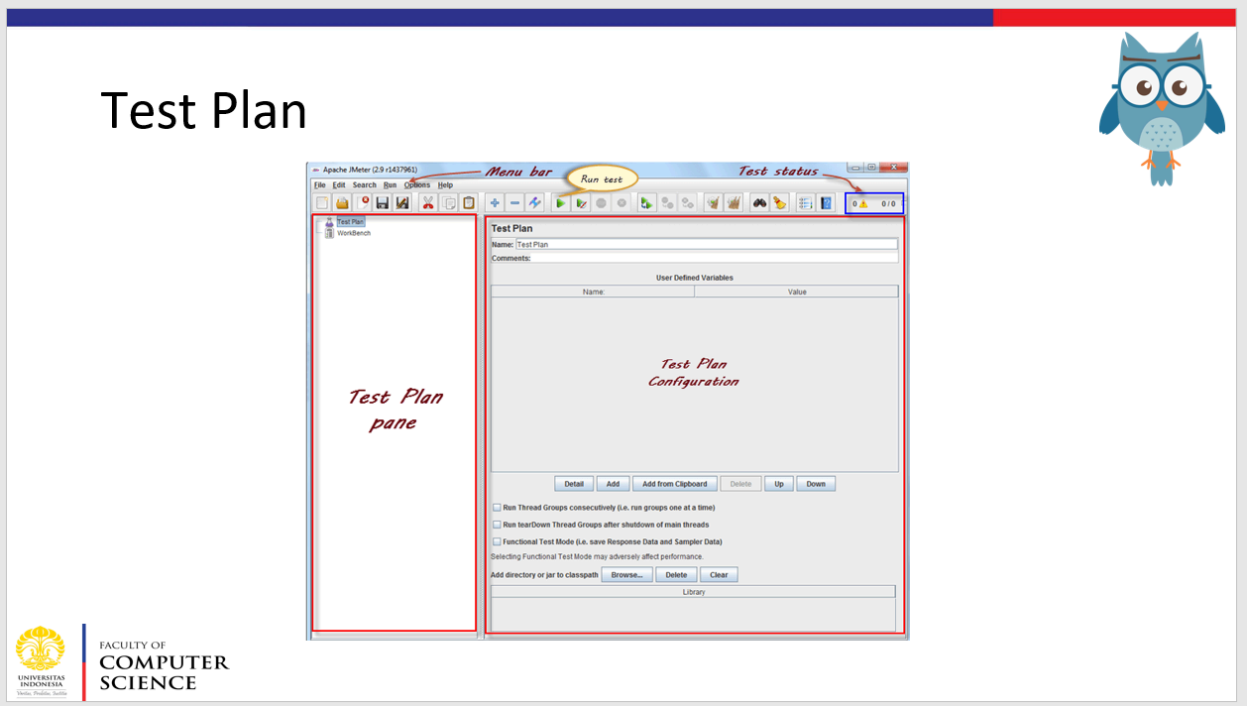
Creating a test plan is the foundational step in leveraging JMeter for performance testing. A test plan acts as a container for all the tests you wish to run, encapsulating the various components required to simulate user behavior and monitor server responses. Essentially, it's a structured document that outlines the sequence of actions JMeter will execute during the testing process.

A comprehensive test plan in JMeter includes a variety of elements, each serving a specific purpose. At its core, every test plan must have at least one thread group. Thread groups define a pool of user simulations, specifying the number of users to simulate, the ramp-up period (how quickly those users are to start), and the number of times to execute the test.

In addition to thread groups, test plans may also contain logic controllers to define the flow of execution, sample-generating controllers such as HTTP requests to simulate user requests to the server, and listeners to collect and visualize test results. Timers can be used to add delays between requests, mimicking real user interaction, while assertions validate the response of the server against expected outcomes. Configuration elements set up defaults and variables for use in several components within the test.

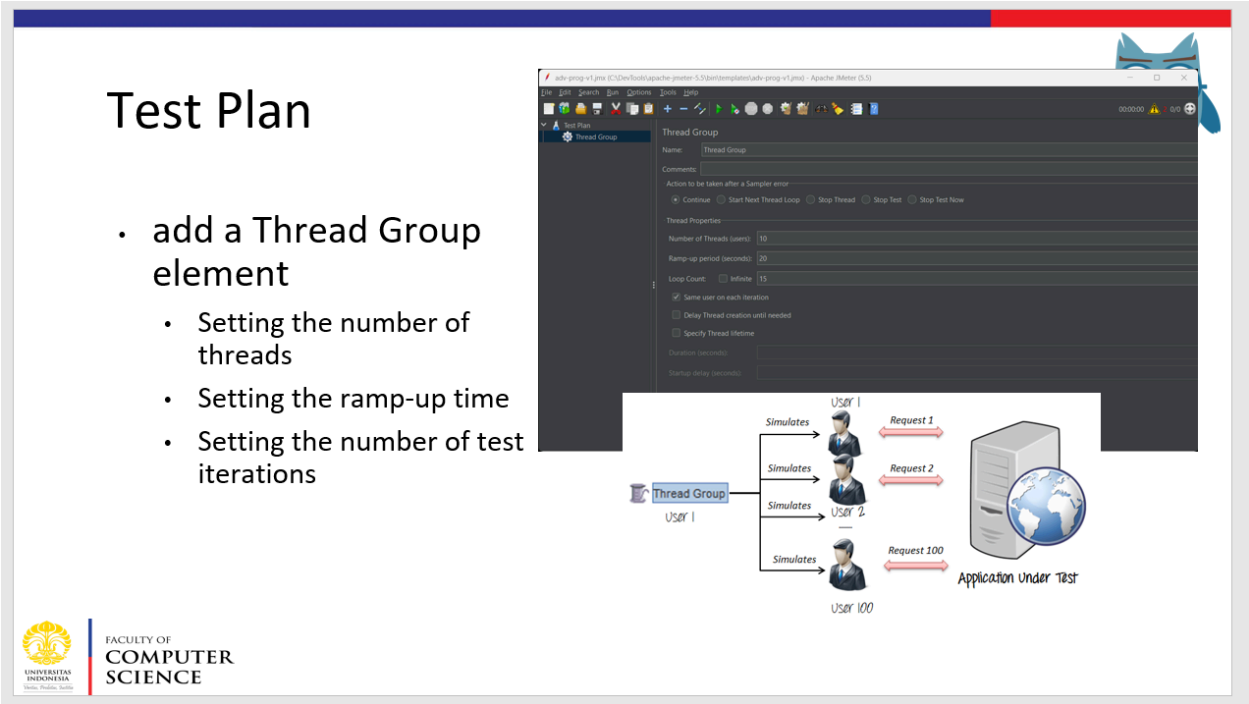
This structured approach ensures that every aspect of the application's behavior can be meticulously tested and optimized based on real-world usage patterns.

Test Plan



The test plan window in JMeter is a well-structured interface designed for efficient setup and management of performance tests. It comprises several key sections: the test plan pane, test plan configuration, menu bar, tool bar, and test status, each offering distinct functionalities. The test plan pane displays the hierarchical structure of the test plan, allowing users to organize and navigate through various testing components like thread groups and listeners. The configuration section is where specific settings for each selected component are adjusted, enabling precise control over test execution parameters.

Above these sections, the menu bar provides comprehensive access to JMeter's features, including creating, saving, and executing test plans. The tool bar beneath offers shortcuts for common actions, enhancing usability. The test status area, typically at the bottom, delivers real-time feedback on the test's progress and results. This organized interface ensures that users can effortlessly configure, execute, and monitor their performance tests, making JMeter an invaluable tool for optimizing application performance.



In JMeter, the addition of a Thread Group element is crucial for simulating user actions on a web application or server. This element acts as the backbone of any test plan, dictating the scale and intensity of the test. When configuring a Thread Group, three primary settings must be adjusted to tailor the test to specific testing requirements.

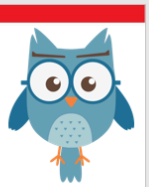
Firstly, the "number of threads" setting determines the total count of virtual users (VUs) that JMeter will simulate. This number directly influences the load exerted on the application, allowing testers to mimic varying levels of user traffic and observe how the application behaves under different load conditions.

Secondly, the "ramp-up time" specifies the duration over which all the specified threads will be launched. A shorter ramp-up time means a more sudden influx of users, which is useful for stress testing, whereas a longer ramp-up time simulates a more gradual increase in traffic, closely mimicking real-world scenarios.

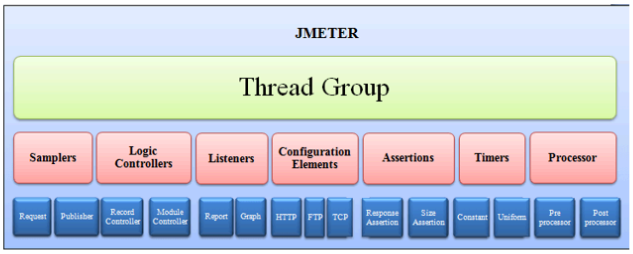
Lastly, the "number of test iterations" controls how many times the test cycle will be executed. This can range from a single iteration, suitable for basic functionality tests, to multiple iterations, ideal for endurance or stability testing.

Together, these settings enable testers to fine-tune their performance tests, providing flexibility in simulating various user behaviors and accurately assessing the application's scalability, reliability, and performance thresholds.

Test Plan Elements

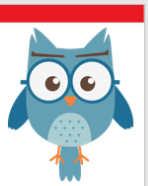


- Samplers: allow JMeter to send specific types of requests to a server
- Logic Controllers: control the order of processing of Samplers in a Thread
- Listeners: view the results of Samplers in the form of tables, graphs, trees, or simple text in some log files



FACULTY OF COMPUTER SCIENCE

Test Plan Elements



- Assertions: include some validation test on the response of your request made using a Sampler
- Configuration Elements: create defaults and variables to be used by Samplers

FACULTY OF COMPUTER SCIENCE

JMeter's test plan elements are the building blocks of any performance testing scenario, each serving a distinct purpose in the creation and execution of a test. Understanding these elements is crucial for effectively leveraging JMeter's capabilities.

Samplers are one of the core components of JMeter, enabling it to make various types of requests to a server. These requests can range from HTTP requests to web pages, SOAP requests to web services, or JDBC queries to databases. By simulating the requests that users would make in a real-world scenario, Samplers allow testers to assess how a server or application responds under different conditions.

Logic Controllers dictate the flow of execution within a test. They determine the order in which Samplers are processed, enabling complex testing scenarios like conditional execution or looping. Logic Controllers can simulate real-user decision-making processes, adding depth and realism to the testing scenario.

Listeners are the window into the performance of the test, providing visual feedback through tables, graphs, trees, or logs. They allow testers to analyze the results of Samplers, offering insights into response times, throughput rates, and error percentages. This data is invaluable for identifying bottlenecks and areas for improvement within the application being tested.

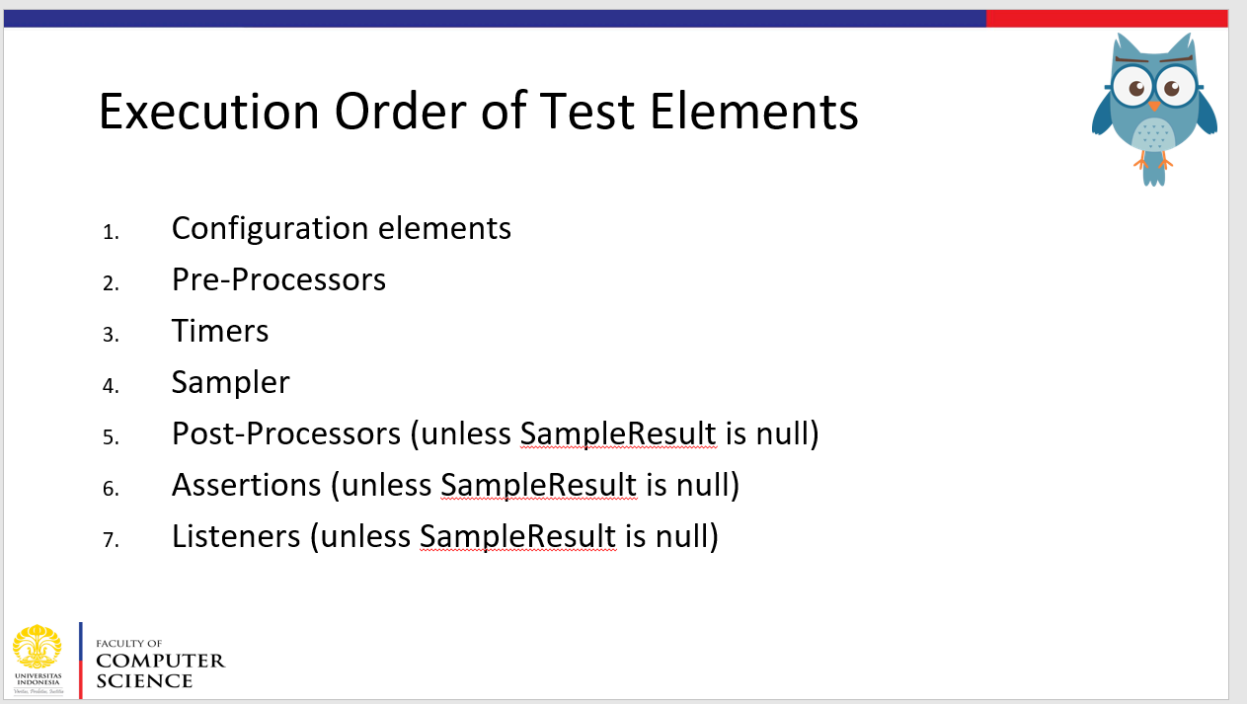
Assertions are used to perform validation checks on the responses received from Samplers. They ensure that the application is returning the correct data, status codes, or response times, aligning with the expected outcome. Assertions are critical for automated testing, allowing JMeter to automatically verify the correctness of application responses.

Configuration Elements set up and manage variables and defaults that Samplers use in their requests. They can define user data, server names, ports, or any other parameter required by the test, ensuring that Samplers operate within the correct context.

The picture also mentions Timers and Processors, which play their roles in shaping the test execution. Timers introduce delays between requests, simulating think time between user actions, while Pre-Processors and Post-Processors manage tasks before and after Sampler execution, such as setting up test conditions or processing the data extracted from responses.

Together, these elements form a comprehensive toolkit for designing and executing detailed performance tests with JMeter, allowing testers to simulate a wide range of user behaviors and assess the performance of web applications and services under various conditions.

Execution Order of Test Elements



The execution order of test elements within a JMeter test plan is critical for understanding how tests are conducted and how data flows through the testing process. This sequence ensures that each element contributes accurately to the test execution, influencing the behavior of samplers and the interpretation of results.

1. Configuration Elements are executed first. They set up or define variables, server settings, and other prerequisites required by the samplers to execute requests. Configuration elements lay the groundwork for the test, ensuring that all subsequent elements operate within the correct context.

2. Pre-Processors come next, performing operations before a sampler executes. They can modify or set up requests, add or alter headers, or manipulate user variables, tailoring the request to specific testing needs.

3. Timers are then triggered, introducing delays between requests. This delay simulates real-user think time, ensuring that the load on the server mimics actual user behavior as closely as possible.

4. The Sampler itself executes following the timer, making the actual request to the server or application under test. The sampler is the core component that simulates user requests and gathers performance data.

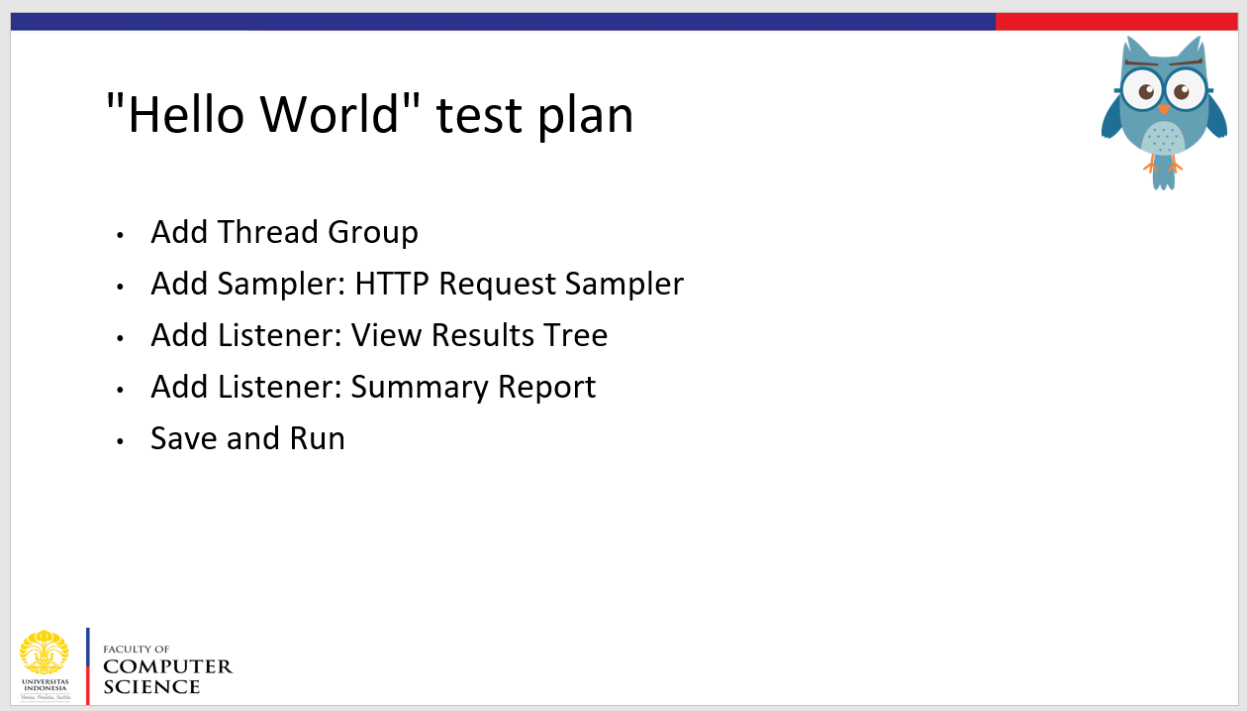
5. Post-Processors are executed after the sampler (unless the SampleResult is null). They process the response from the server, extracting data, managing variables, or handling server responses for use in later stages of the test plan.

6. Assertions evaluate the sampler response to ensure it meets expected criteria. This step is crucial for validating that the application's response is as expected, based on the test's objectives.

7. Finally, Listeners are executed (unless the SampleResult is null). They provide a visual representation of the test results, displaying data on response times, success rates, and other metrics in various formats for analysis.

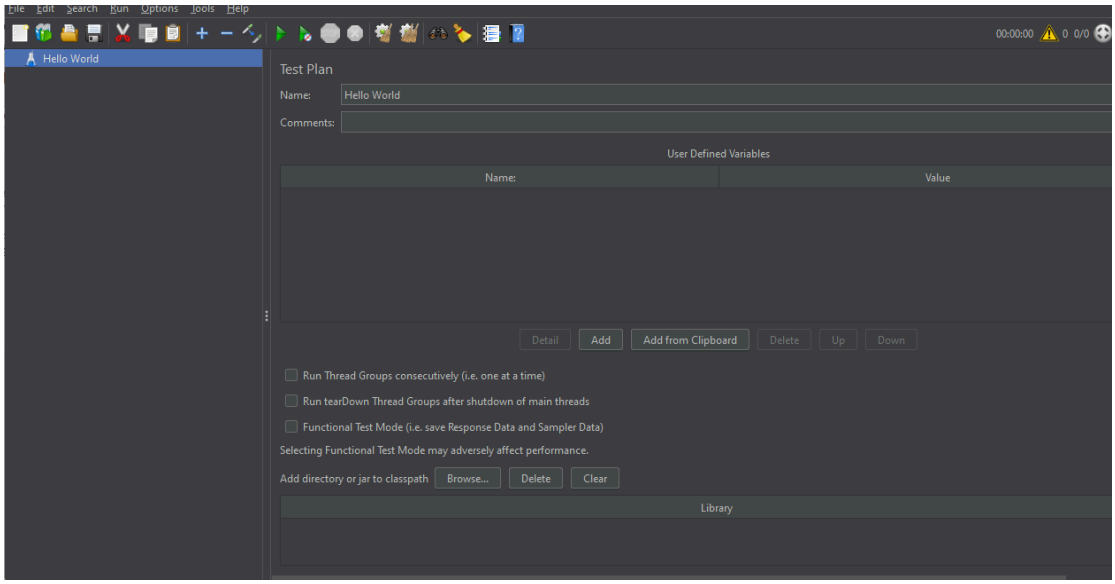
This structured execution order allows JMeter to accurately simulate user actions, assess application performance, and provide detailed feedback on how the application behaves under specific conditions.

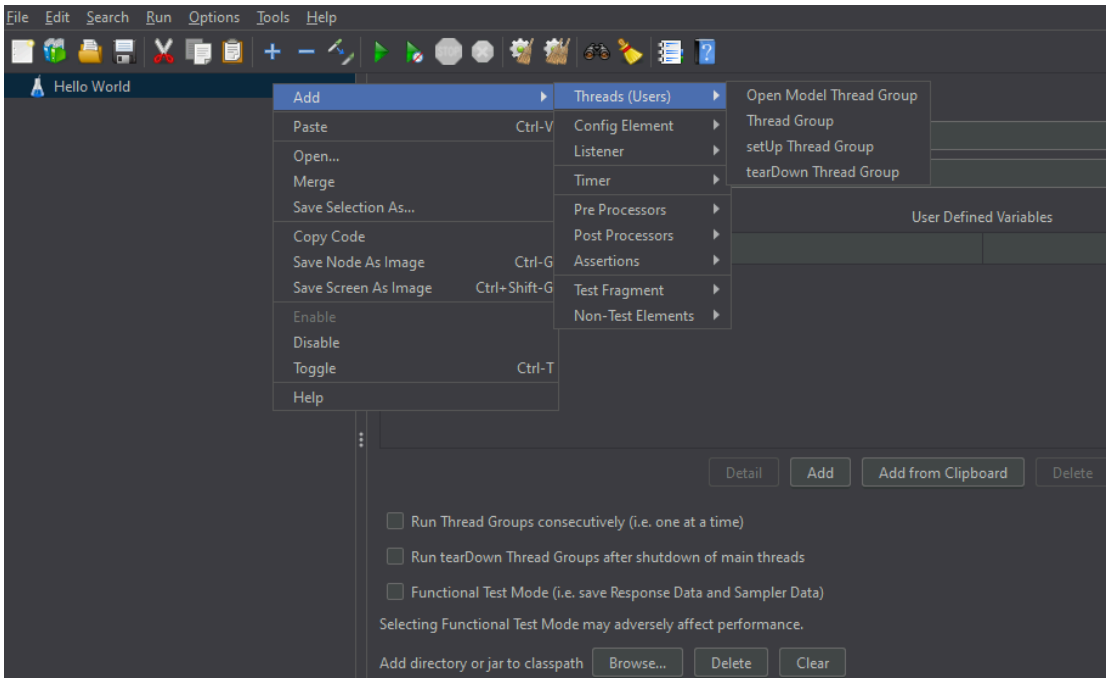
Hello World Test Plan

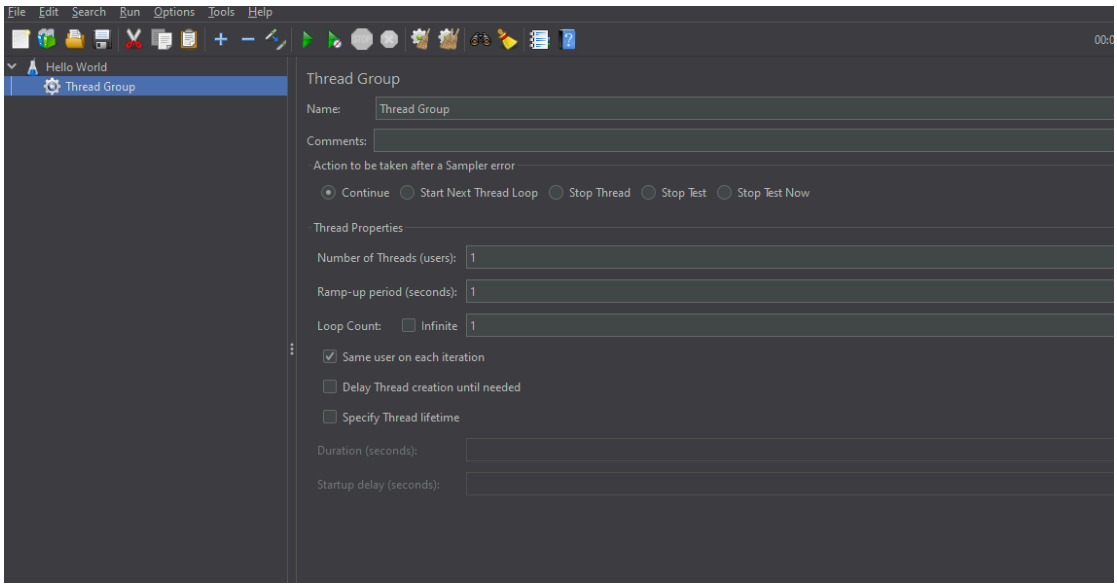


Creating a "Hello World" test plan in JMeter is a straightforward way to get acquainted with its fundamental components and execute a simple performance test. This basic test plan includes a sequence of steps that simulate a user visiting a webpage, showcasing how JMeter sends requests and processes responses.

1. Add Thread Group: Begin by adding a Thread Group to your test plan. This element acts as the starting point for your test, allowing you to specify the number of virtual users (threads), the ramp-up period for starting these users, and the number of times to execute the test.

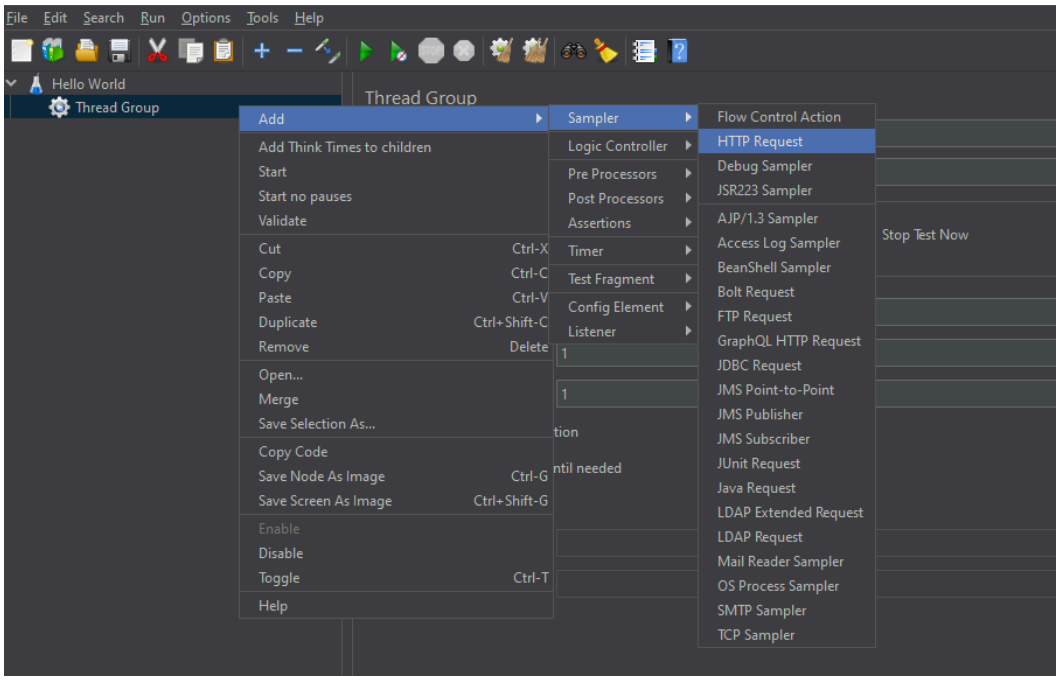

File Edit Search Run Options Tools Help
00:00:00 0 0/0
Hello World
Test Plan
Name: Hello World
Comments:
User Defined Variables
Name:
Value
Detail Add Add from Clipboard Delete Up Down
Run Thread Groups consecutively (i.e. one at a time)
Run tearDown Thread Groups after shutdown of main threads
Functional Test Mode (i.e. save Response Data and Sampler Data)
Selecting Functional Test Mode may adversely affect performance.
Add directory or jar to classpath Browse... Delete Clear
Library

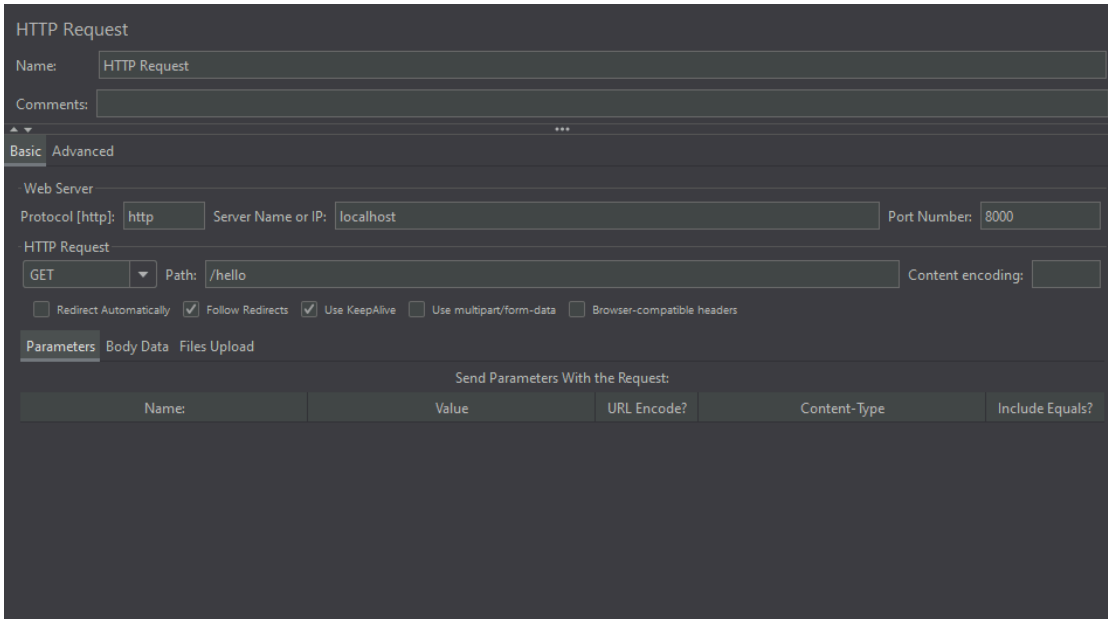




2. Add Sampler: HTTP Request Sampler: Next, incorporate an HTTP Request Sampler into the Thread Group. This sampler simulates a user request to a web server. Configure it by specifying the website's URL you wish to test, which could be the homepage of your application or any

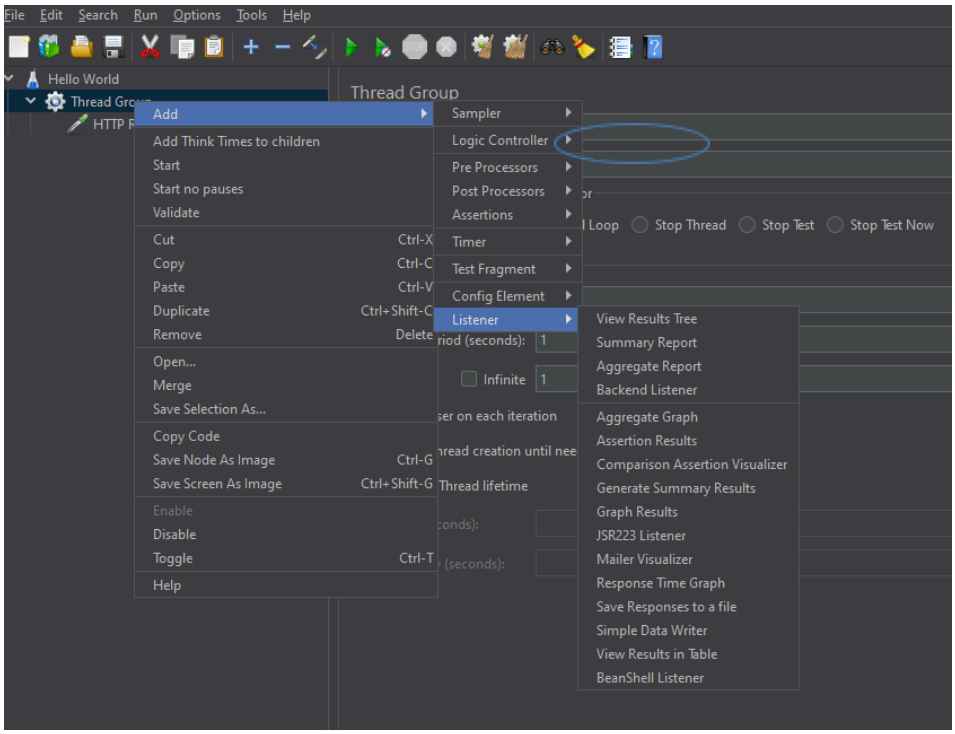
other page of interest.



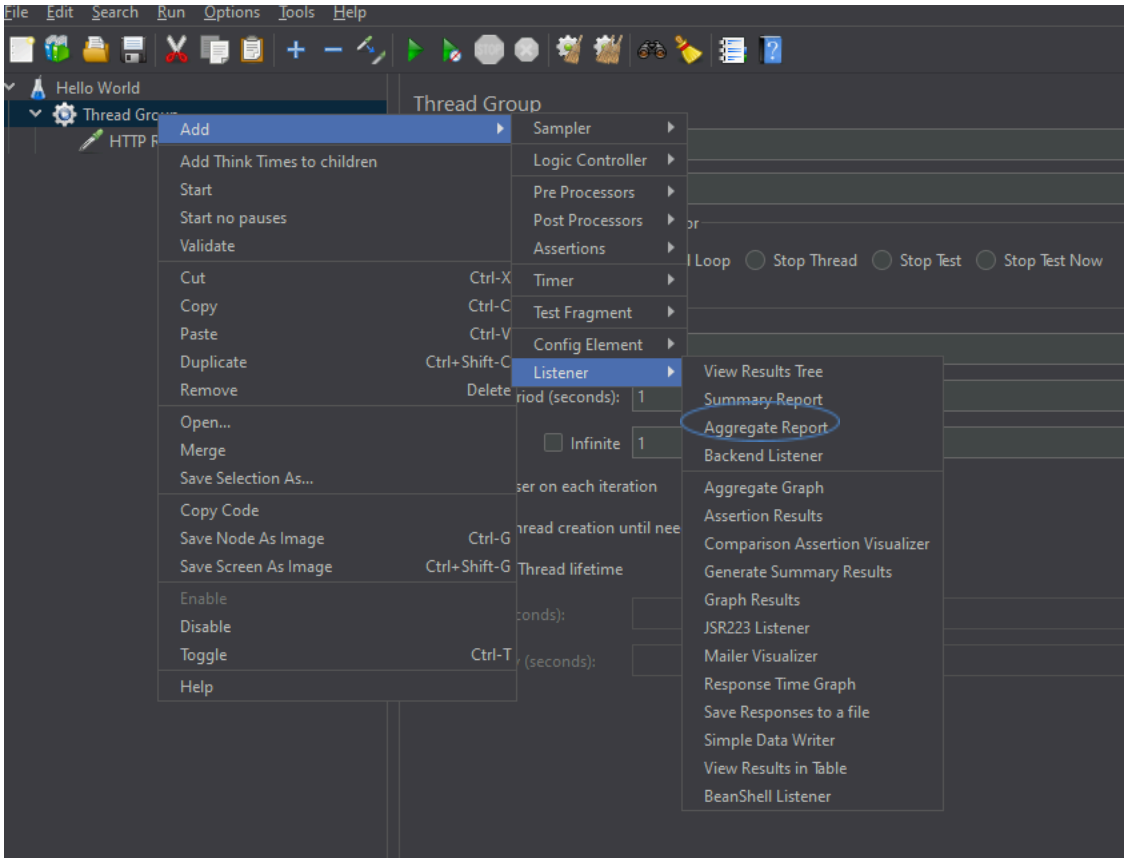


3. Add Listener: View Results Tree: To visualize the response received from the server for each request, add a View Results Tree listener. This listener provides detailed insights into the request sent, the response received, and any errors encountered, making it invaluable for debugging and

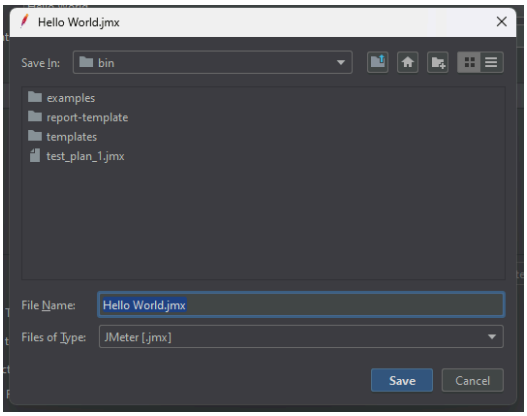
understanding the server's behavior.

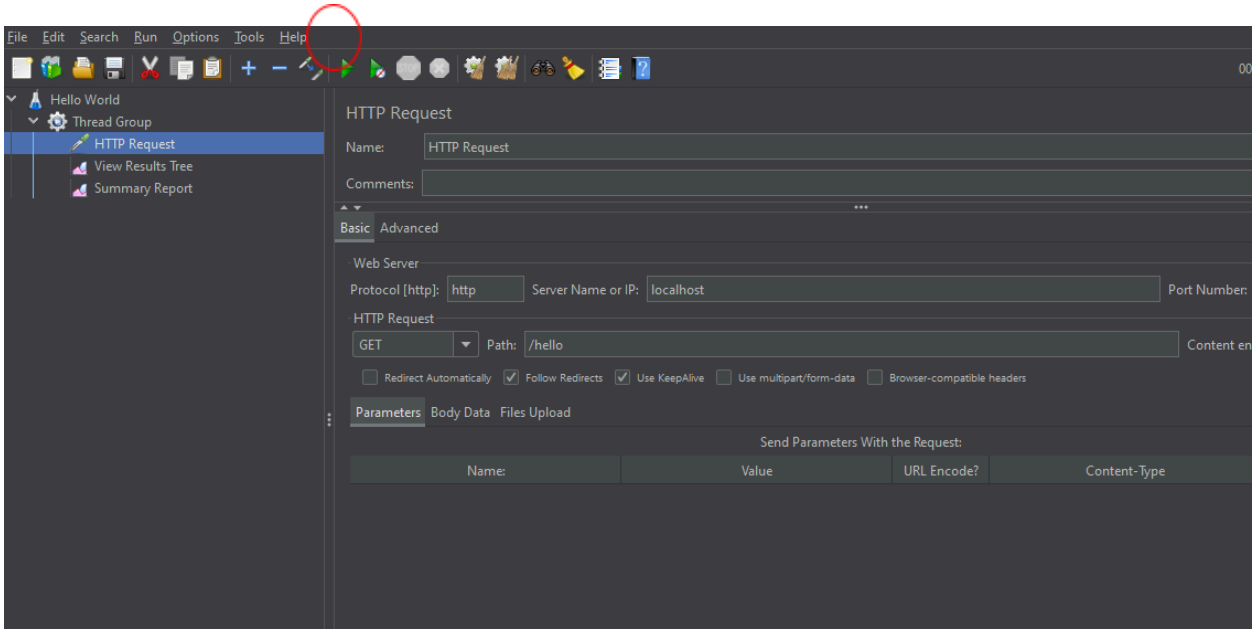


4. Add Listener: Summary Report: For a concise overview of the test's performance, include a Summary Report listener. It aggregates the results of the test execution, offering statistics on response times, throughput, error rates, and other key performance indicators.



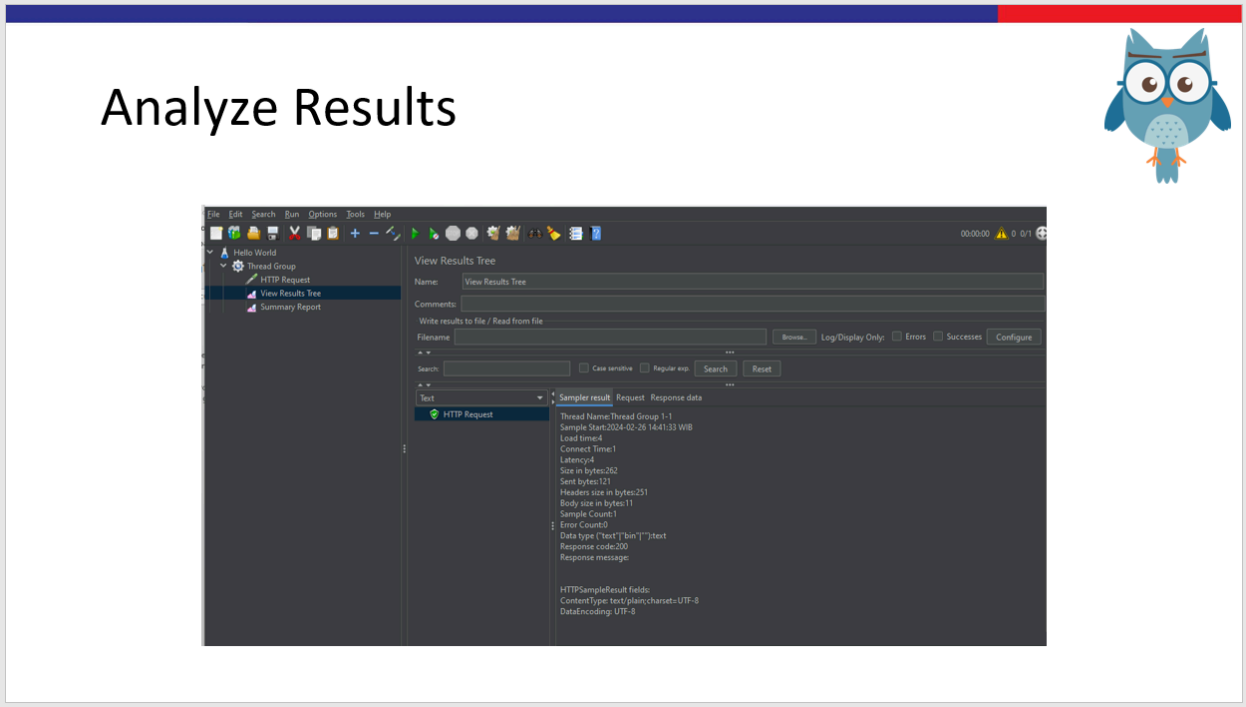
5. Save and Run: Once the components are configured, save your test plan and run it. JMeter will execute the specified number of requests to the target server, and the listeners you've added will display the results.





This "Hello World" test plan is an excellent introduction to JMeter's capabilities, demonstrating the ease with which you can set up and run performance tests. It lays the foundation for more complex testing scenarios, allowing for the gradual incorporation of additional elements and configurations to meet specific testing requirements.

Analyze Result



The View Results Tree listener in JMeter provides a detailed breakdown of each request made during a test, offering comprehensive insights into how the server responds to each simulated user action. For the given result, the output displays several key metrics that are crucial for evaluating the performance and reliability of the application under test.

The "Thread Name" indicates the specific thread (virtual user) that made the request, here labeled as "Thread Group 1-1," suggesting it's the first thread from the first group. The "Sample Start" timestamp marks the exact time the request was initiated, providing context for when the test was conducted.

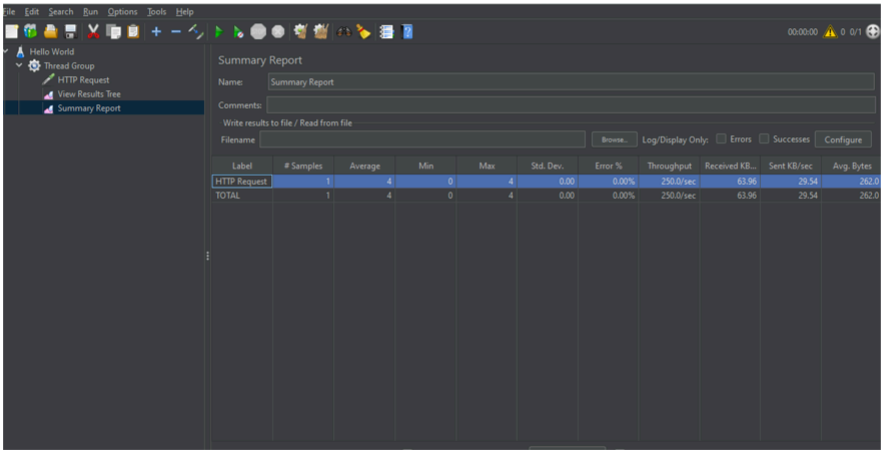
Notably, the "Load time" of 4 milliseconds signifies the total time taken to receive the response after sending the request, reflecting the server's efficiency in processing and responding. The "Connect Time" of 1 millisecond shows how quickly the connection to the server was established, which is essential for understanding the network latency.

"Latency," also 4 milliseconds, represents the time to the first byte received, indicating the responsiveness of the server. The "Size in bytes" details, including the total response size (262 bytes), the request size (121 bytes), headers size (251 bytes), and body size (11 bytes), offer insights into the payload exchanged during the interaction.

With "Sample Count" at 1 and "Error Count" at 0, it confirms that the request was successfully executed without errors. The "Response code" of 200 indicates a successful HTTP request, and the "ContentType" of text/plain;charset=UTF-8, along with "DataEncoding" of UTF-8, describes the format and encoding of the response, confirming the server returned a text response in UTF-8 encoding.

This detailed output from the View Results Tree is invaluable for developers and testers, providing a granular view of each request's performance and outcome, aiding in diagnosing issues and optimizing server responses.

Analyze The Summary



Analyze The Summary

- General statistics: the total number of requests and average response time.
- Response time distribution: the distribution of response times, including the minimum, maximum
- Throughput: the number of requests per second and the average bytes per second.
- Error percentage: the percentage of failed requests and the types of errors encountered.

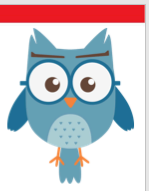
The Summary Report in JMeter offers a concise overview of key performance metrics from testing sessions. It presents general statistics such as the total number of requests made during the test and the average response time, which helps in assessing the overall efficiency of the server. The report also details the distribution of response times, including the fastest (minimum) and slowest (maximum) response times, providing insight into the variability and reliability of server responses. Throughput is another critical metric covered, indicating the number of requests processed per second and the average data transfer rate, which reflects the server's capacity to handle load. Additionally, the report includes the error percentage, summarizing the proportion of requests that failed and identifying common error types. This comprehensive snapshot allows developers and testers to quickly gauge the performance and stability of the application or server under test.

The Summary Report in JMeter succinctly aggregates performance testing metrics for HTTP requests. It shows a single request was made, with both the minimum and maximum response times recorded at 4 milliseconds, indicating a fast and consistent server response for this particular test. The reported 90th percentile response time at 250 milliseconds suggests a potential anomaly or reporting error, as it significantly deviates from the other metrics for a single-request test.

Throughput is calculated at approximately 64 requests per second, with the average amount of data transferred per request being 262 bytes. This indicates the server's capability to handle requests efficiently within the tested parameters. The standard deviation value, 29.541015625, is intended to measure variability but is less relevant here due to the singular nature of the test.

This condensed report highlights key aspects of server performance, such as response time and throughput, offering a quick reference to assess the server's efficiency in handling HTTP requests. For comprehensive analysis, further testing with multiple requests is recommended to validate these initial findings.

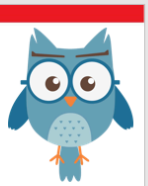
Best Practice



- run jmeter in non-GUI mode for more accurate results
- disable some listeners to speed up
- write results to text, analyze later
- Ensure that the test environment is configured to closely match the production environment, including hardware, software, network, and database configurations.
- Use appropriate load generation and distribution strategies to simulate realistic user behavior and avoid overloading the system.

FACULTY OF COMPUTER SCIENCE

Best Practice



- Use parameterization to vary input data and ensure that the load testing is not biased towards a specific set of data.
- Pay attention to your client environment: computer performance, network bandwidth
- run Distributed Testing
- The maximum number of threads you can effectively run with JMeter is 300. Limit to 100.

FACULTY OF COMPUTER SCIENCE

Adopting best practices when using JMeter can significantly enhance the accuracy and relevance of performance testing results. One of the fundamental recommendations is to run JMeter in non-GUI mode for test execution, as the GUI consumes considerable resources and can affect

test outcomes. This mode ensures more accurate results and reduces resource consumption, making it ideal for load testing.

Disabling unnecessary listeners during test execution is another critical practice. Listeners can slow down the testing process, especially when handling a large volume of data. Instead, it's advisable to write the results to a file for post-test analysis. This approach not only speeds up the testing process but also allows for a detailed review of the results without the overhead of live data visualization.

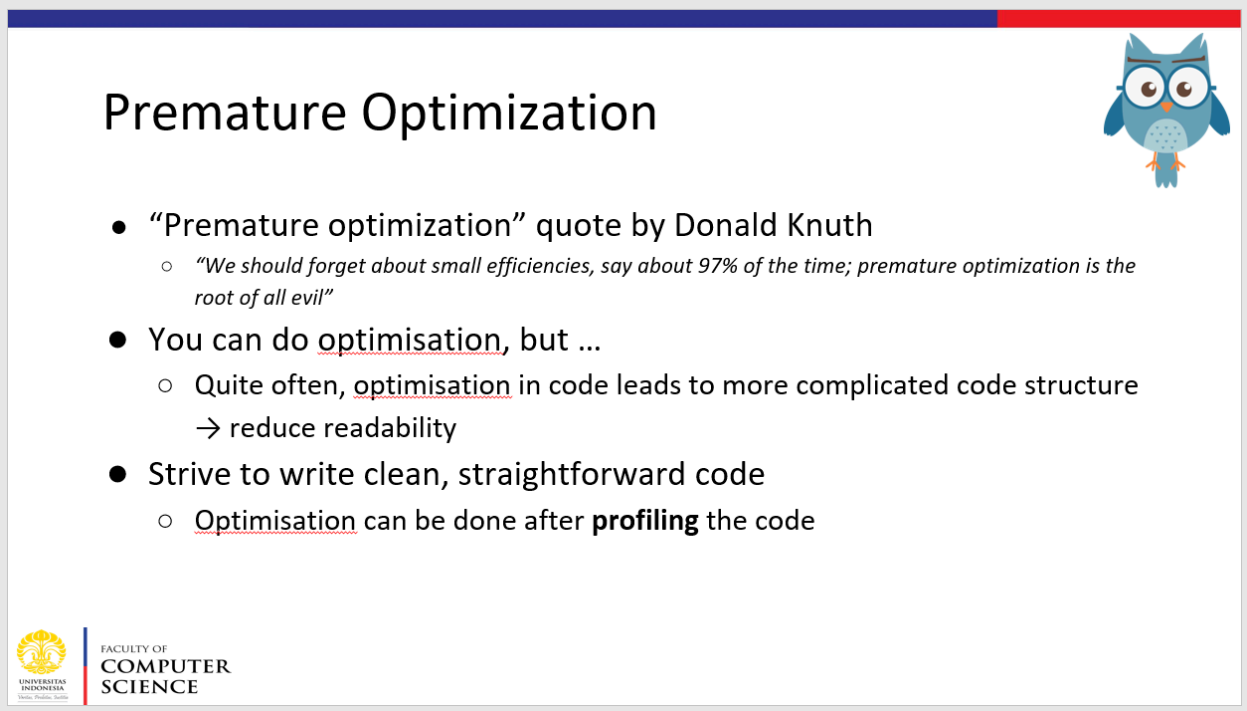
Ensuring that the test environment mirrors the production environment as closely as possible is vital for obtaining relevant insights. This similarity should encompass hardware, software, network configurations, and database setups to accurately predict how the application will perform under real-world conditions.

Adopting realistic load generation and distribution strategies is crucial. Simulating user behavior accurately, without overloading the system, requires thoughtful planning and configuration. Parameterization of input data ensures tests cover a broad range of user interactions, preventing bias towards specific data sets and scenarios.

Considering the client environment's capabilities, such as computer performance and network bandwidth, is essential to avoid bottlenecks that could skew test results. For large-scale tests, distributed testing is recommended, where test load is distributed across multiple machines to simulate more users and achieve higher scalability.

Lastly, while JMeter can theoretically handle thousands of threads, practical limits are much lower due to resource constraints. A general guideline is to limit the number of threads to 300 per test instance, with an optimal number around 100 threads for most environments, ensuring that the testing resources are not overstretched and that the results remain reliable and indicative of actual performance. Following these best practices allows testers to efficiently utilize JMeter's capabilities, providing accurate and actionable performance insights.

Premature Optimization

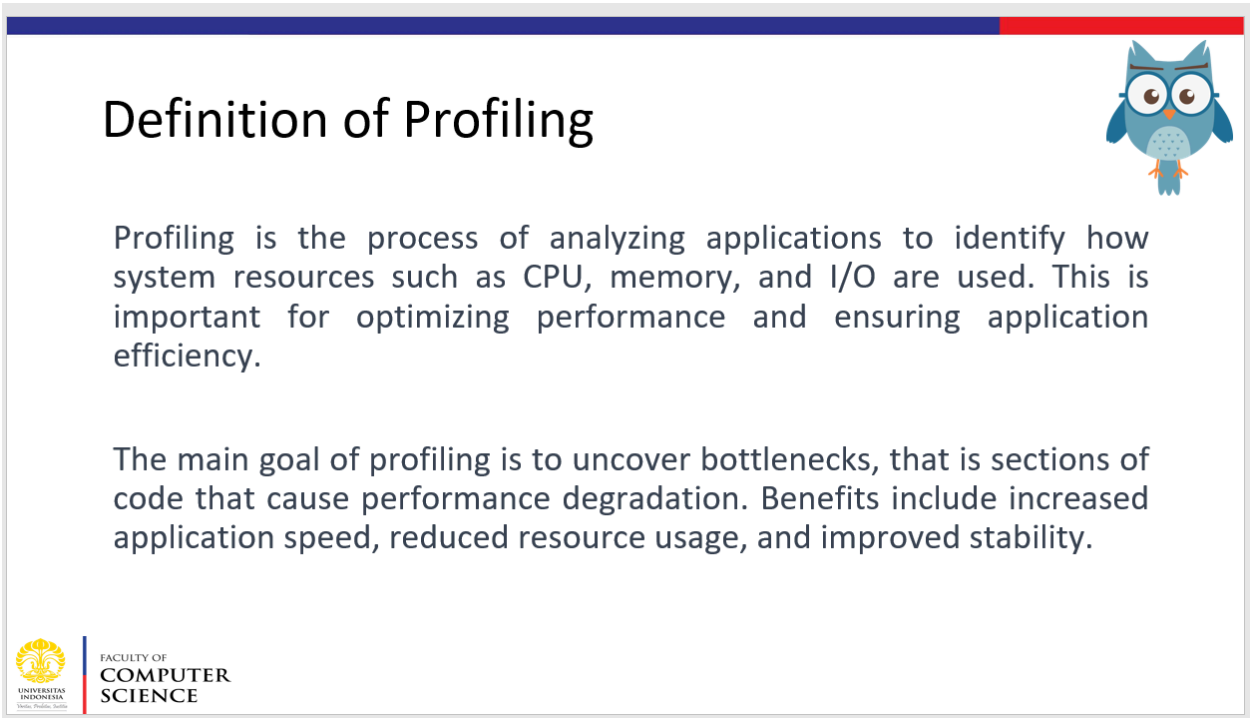


Donald Knuth's quote on "premature optimization" underscores a critical principle in software development, emphasizing the importance of focusing on code clarity and functionality before delving into optimization efforts. Knuth advises that for about 97% of the time, developers should prioritize writing clean and straightforward code over seeking small efficiencies. This approach is grounded in the reality that optimizing too early in the development process often results in a more complex and less readable code structure, which can complicate maintenance and future development.

The essence of this wisdom is not to dissuade optimization altogether but to caution against its premature application. Optimization should be a deliberate process, informed by insights gained from profiling the code to identify actual performance bottlenecks. By profiling, developers can pinpoint where optimizations will have the most significant impact, ensuring that efforts to enhance performance do not come at the expense of code quality and maintainability.

Striving for simplicity in code writing lays a solid foundation for building robust and efficient software. It allows for easier understanding, testing, and modification of the code. Once the software is functional and its performance characteristics are well understood, targeted optimizations can be made to refine efficiency without undermining the codebase's integrity. This balanced approach to optimization ensures that enhancements contribute positively to the software's overall quality and performance.

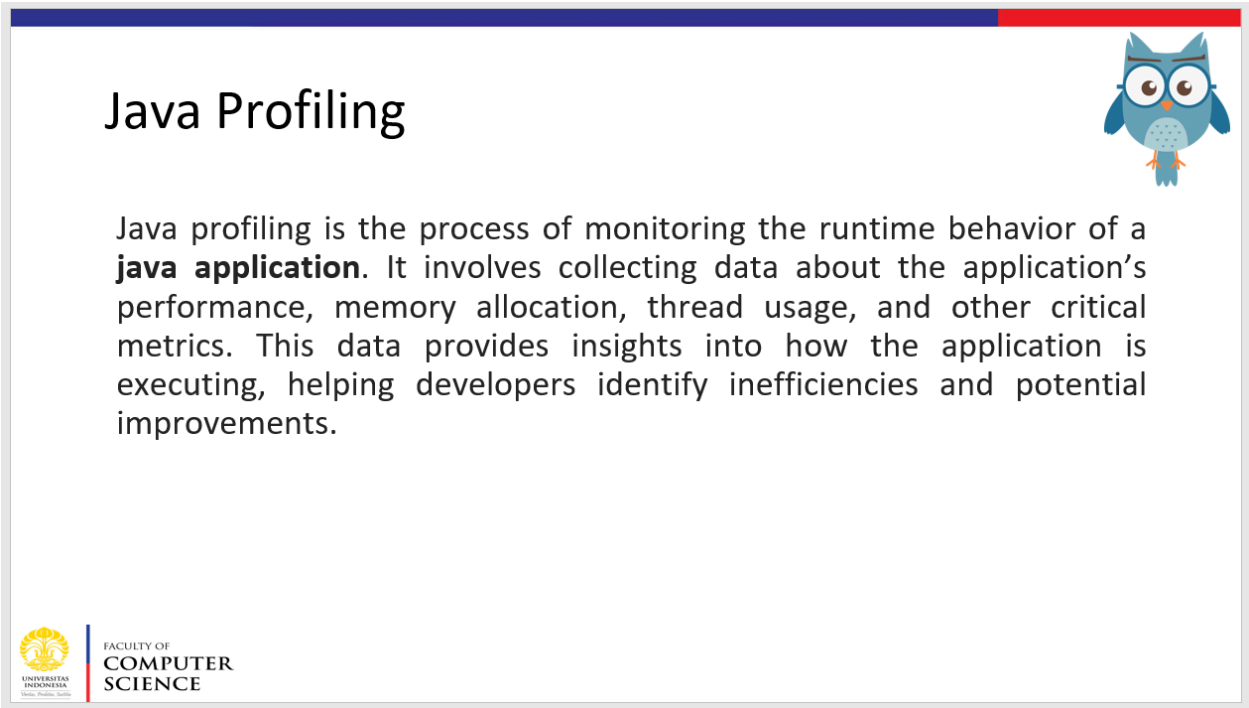
Java Profiling



Profiling stands as a crucial technique in software development, aimed at enhancing an application's performance by meticulously analyzing its interaction with system resources like CPU, memory, and I/O. This process is not just about monitoring; it's a detailed investigation into how an application behaves under various conditions, identifying the most resource-intensive parts of the code. The primary objective of profiling is to pinpoint bottlenecks—specific areas in the code that significantly slow down performance or excessively consume resources.

By identifying these critical sections, developers can focus their optimization efforts where they are needed most, leading to substantial improvements in application speed, efficiency, and overall stability. This targeted approach ensures that resources are used judiciously, which is particularly important in environments where resource constraints are a factor. Moreover, profiling aids in preempting potential performance issues before they escalate into more significant problems, contributing to a smoother user experience and more reliable software operation. As a result, profiling is an indispensable step in the optimization process, enabling developers to make informed decisions about where and how to refine their code for maximum efficiency and effectiveness.

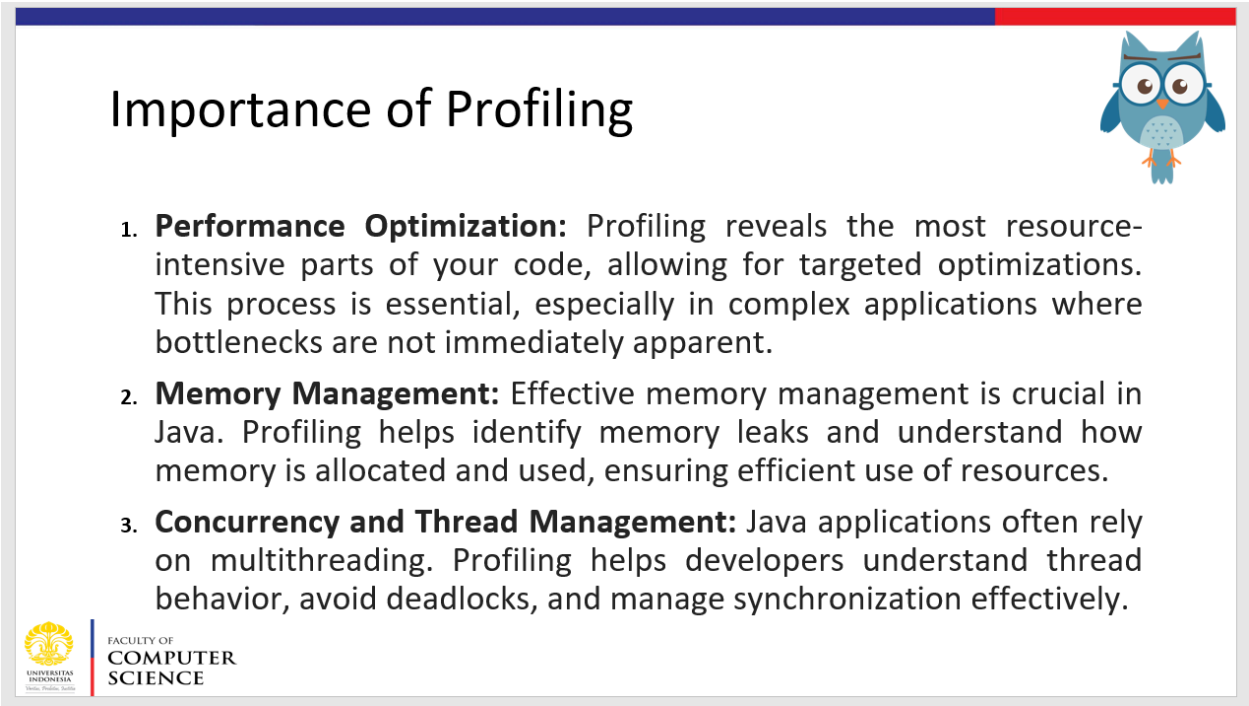
Java Profiling



Java profiling is a vital process for developers seeking to optimize their applications. It delves into the intricate details of an application's runtime behavior, tracking various performance metrics such as memory usage, execution speed, thread activity, and more. This comprehensive data collection sheds light on the inner workings of the application, revealing areas where performance may not be optimal. By analyzing these metrics, developers can pinpoint specific parts of the code that may be causing bottlenecks or consuming excessive resources.

The insights gained from Java profiling enable developers to make targeted improvements, enhancing the application's overall efficiency and responsiveness. Whether it's refining memory allocation to prevent leaks, optimizing slow-running code segments, or managing thread synchronization more effectively, Java profiling provides the empirical evidence needed to guide these enhancements. This methodical approach to performance tuning not only leads to more robust and scalable Java applications but also significantly improves the user experience by reducing latency and increasing reliability. In essence, Java profiling is an indispensable tool in the developer's arsenal for creating high-performing, efficient, and stable Java applications.

Importance of Profiling

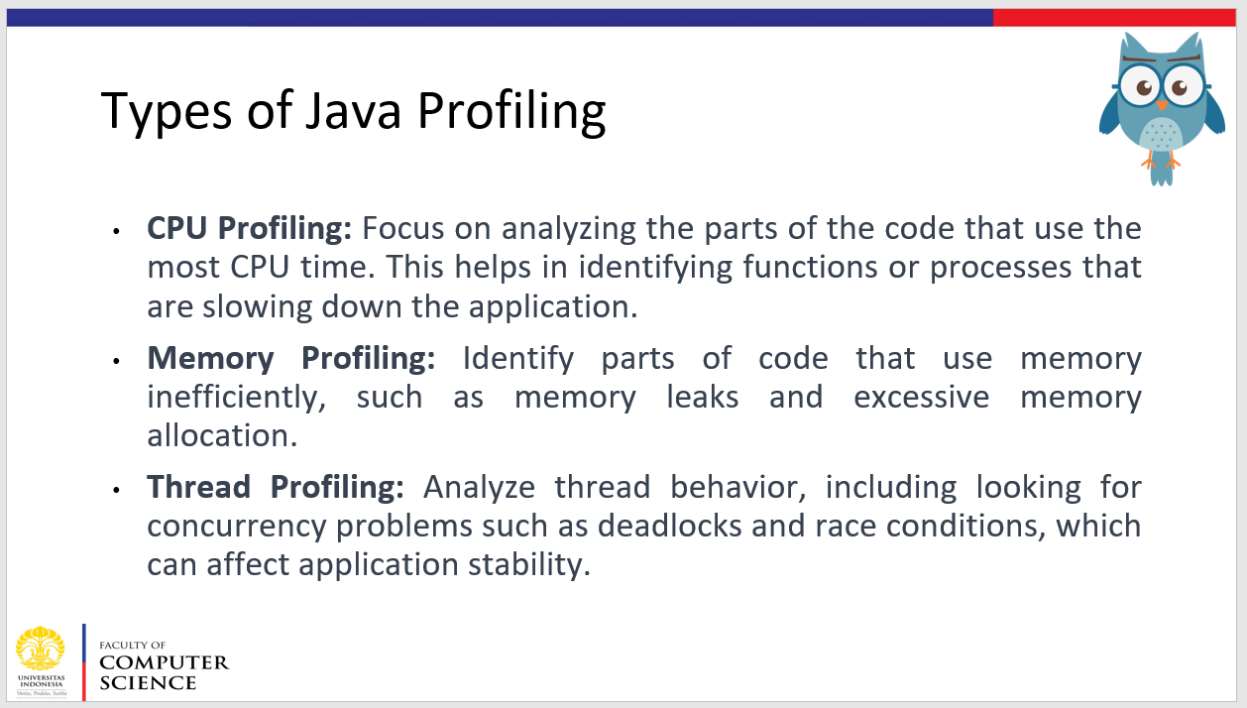


Profiling stands as a cornerstone in the optimization of Java applications, offering a granular view into the runtime operations and resource utilization. By pinpointing the segments of code that consume the most resources, profiling empowers developers to undertake precise optimizations, enhancing the application's performance. This meticulous approach is particularly invaluable in complex systems where identifying bottlenecks is not straightforward.

Beyond performance, profiling plays a critical role in memory management. Java, with its garbage-collected environment, demands efficient memory usage to prevent bloating and ensure smooth operation. Through profiling, developers can detect memory leaks—where objects are not properly disposed of—and analyze memory allocation patterns. This insight enables the refinement of memory usage, ensuring that the application does not waste valuable system resources.

Moreover, Java's capability to run concurrent processes through multithreading introduces challenges in managing threads safely and efficiently. Profiling offers a window into the behavior of threads within the application, facilitating the detection of deadlocks and the optimization of synchronization mechanisms. By understanding how threads interact and compete for resources, developers can devise strategies to mitigate contention, improve responsiveness, and enhance the scalability of their applications.

Types of Java Profiling



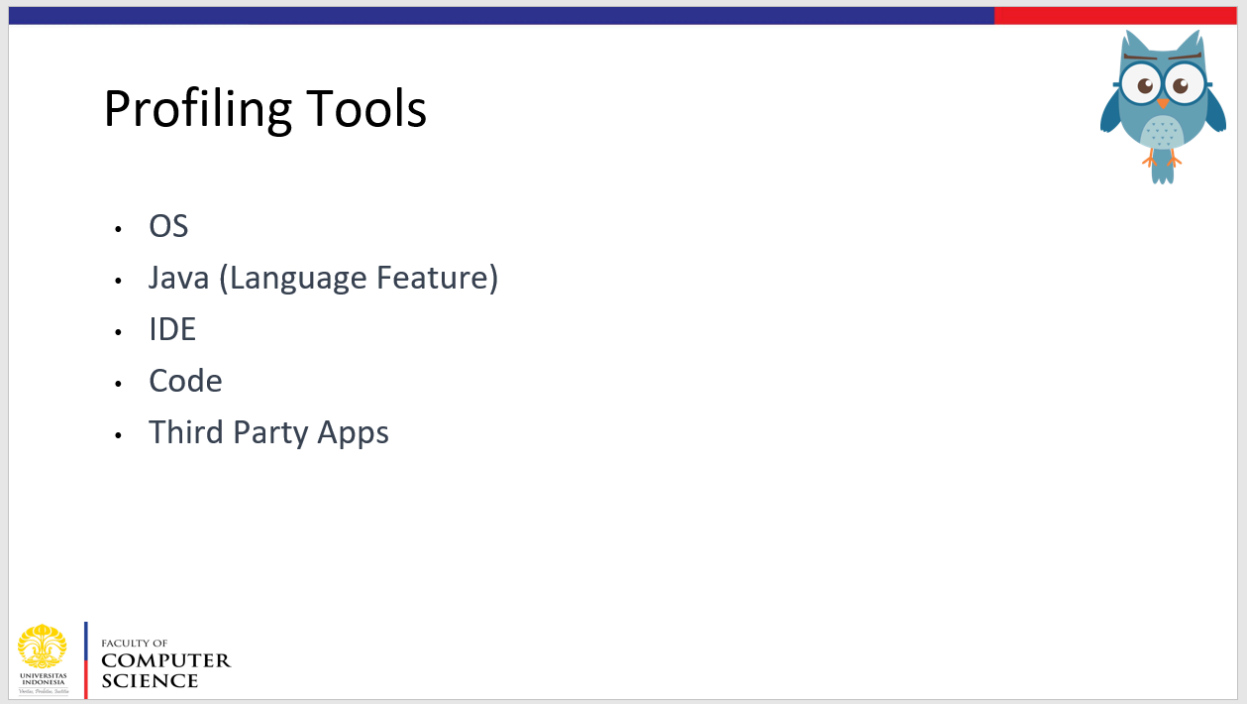
Java profiling is an essential practice for optimizing application performance and stability, categorized into CPU, memory, and thread profiling, each focusing on distinct aspects of application behavior.

CPU profiling is crucial for identifying hotspots in the code—sections that consume significant CPU time. By pinpointing these areas, developers can optimize or refactor them to enhance the application's overall efficiency. This type of profiling is instrumental in ensuring that an application runs smoothly, especially in CPU-intensive tasks or environments.

Memory profiling, on the other hand, targets the efficient use of memory. It helps in detecting memory leaks—where the application fails to release unused objects, leading to wasted memory and potential application slowdowns over time. Additionally, it identifies excessive memory allocation, guiding developers to optimize memory usage, crucial for applications running in limited-memory environments.

Thread profiling addresses the complexities of multithreading in Java applications. It focuses on analyzing thread behavior, identifying concurrency issues such as deadlocks, where two or more threads block each other by holding resources the other needs, and race conditions, which occur when the outcome of a process depends on the sequence or timing of other uncontrollable events. By resolving these issues, thread profiling ensures that the application remains stable and responsive under concurrent operations.

Profiling Tools



Profiling tools are instrumental in optimizing and debugging software by analyzing its runtime behavior to identify areas for improvement. These tools can be categorized based on their integration level, from operating systems (OS) to development environments and third-party applications.

Operating System Tools: These tools provide a low-level view of the application's performance, including CPU usage, memory allocation, and disk I/O operations. They are useful for understanding how an application interacts with the underlying hardware and OS.

Java Language Features: Java offers built-in features like the *jcmd* and *jconsole* that allow for monitoring and managing resource consumption, thread states, and application performance directly within the Java ecosystem.

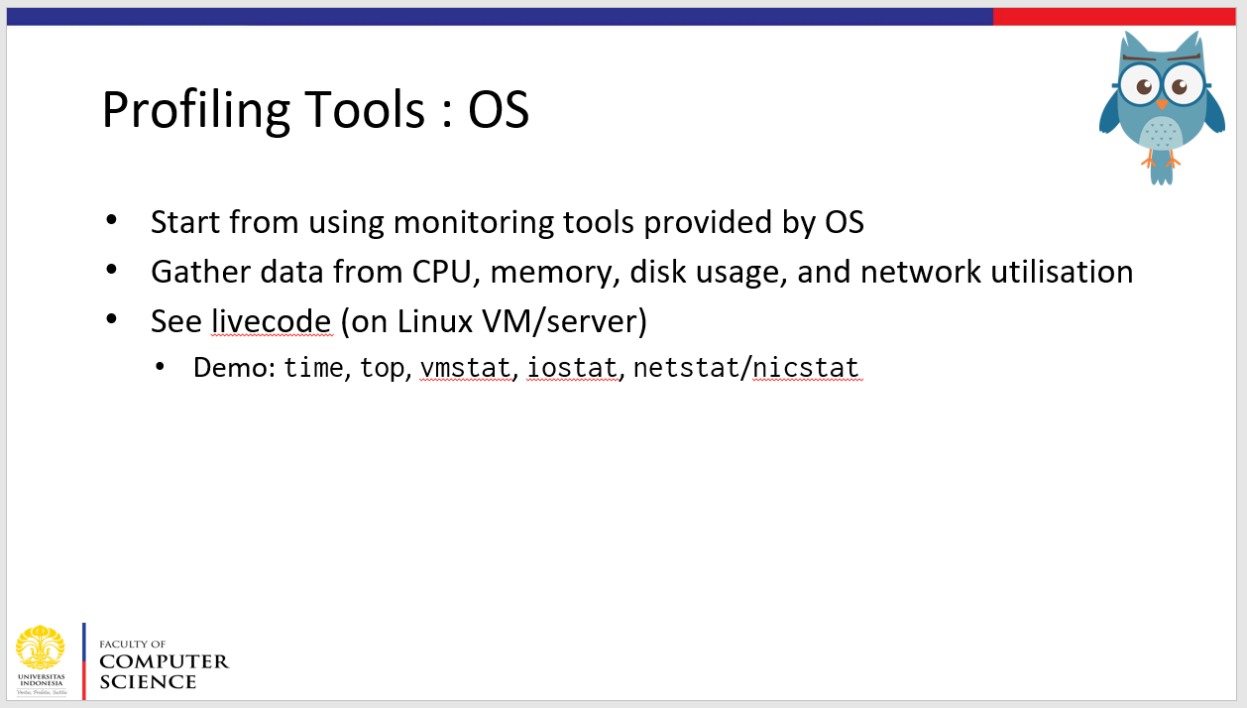
Integrated Development Environment (IDE) Tools: When it comes to IDEs, tools such as the IntelliJ profiler offer deep integration with the development environment, facilitating seamless profiling during the coding process. These IDE-based profilers allow for on-the-fly analysis of CPU, memory, and threading issues without leaving the development environment, thus streamlining the optimization process.

Code-Level Tools: These are libraries or frameworks integrated into the application codebase to provide insights into specific aspects of application performance, such as execution time for methods or frequency of function calls.

Third-Party Applications: A wide range of third-party profiling tools offer advanced features for performance analysis, including detailed memory usage analysis, garbage collection behavior, and thread deadlock detection. These tools often provide more comprehensive analysis capabilities and can be used across different programming languages and environments.

Together, these tools form a comprehensive suite that developers can leverage to ensure their applications are efficient, reliable, and scalable. By carefully selecting and utilizing these tools, developers can pinpoint performance bottlenecks, understand resource usage patterns, and make informed decisions to optimize their software.

Profiling Tools : OS

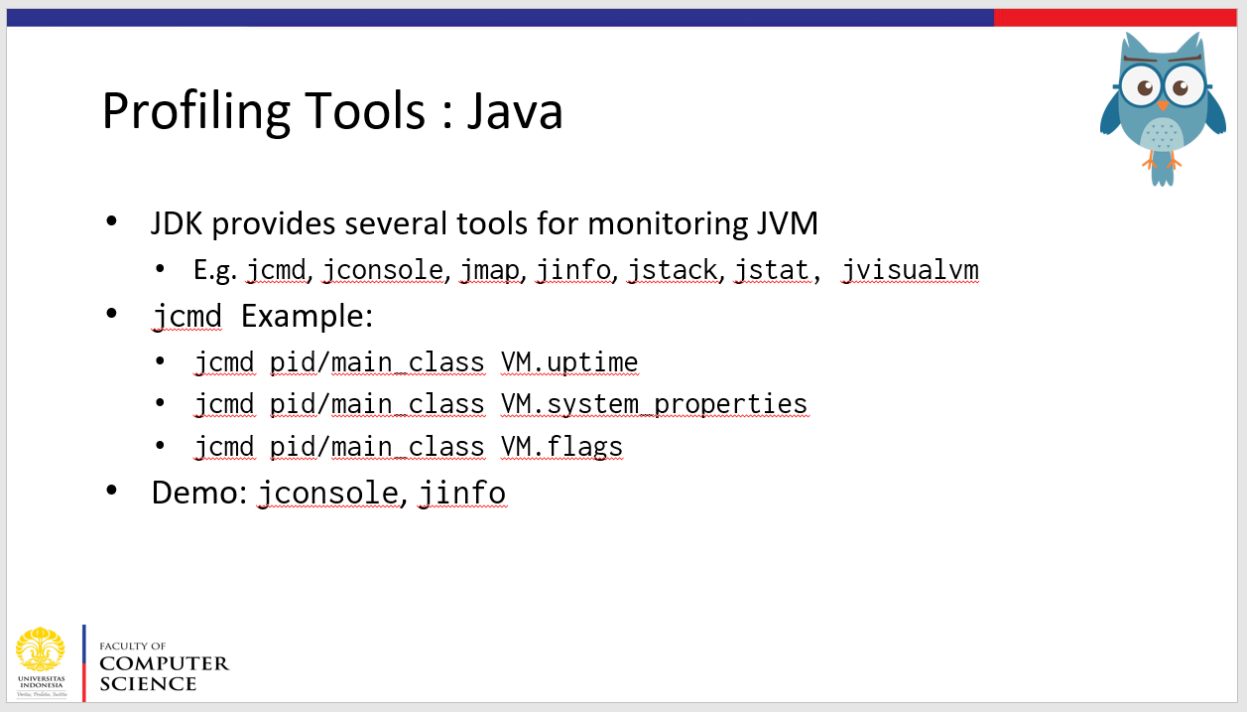


In the domain of software optimization, beginning with operating system (OS)-level monitoring tools is a foundational step. These tools are instrumental in collecting comprehensive data on the utilization of critical system resources such as CPU, memory, disk usage, and network utilization. By leveraging the monitoring capabilities provided by the OS, developers can gain valuable insights into how an application interacts with the underlying hardware, identifying potential bottlenecks and areas for improvement.

For instance, on a Linux VM or server, developers can utilize command-line tools like ‘top’ for real-time CPU and memory monitoring, ‘iostat’ for disk I/O statistics, ‘vmstat’ for memory usage, and ‘netstat’ or ‘iftop’ for network utilization. This live code observation enables developers to pinpoint inefficiencies in resource usage that could impact application performance.

This approach of starting with OS-level profiling tools serves as a preliminary yet powerful method for understanding the broader context of application performance. By identifying high-level resource usage patterns, developers can then delve deeper into specific areas with more targeted profiling tools, making OS-level monitoring an essential first step in the optimization process.

Profiling Tools : Java

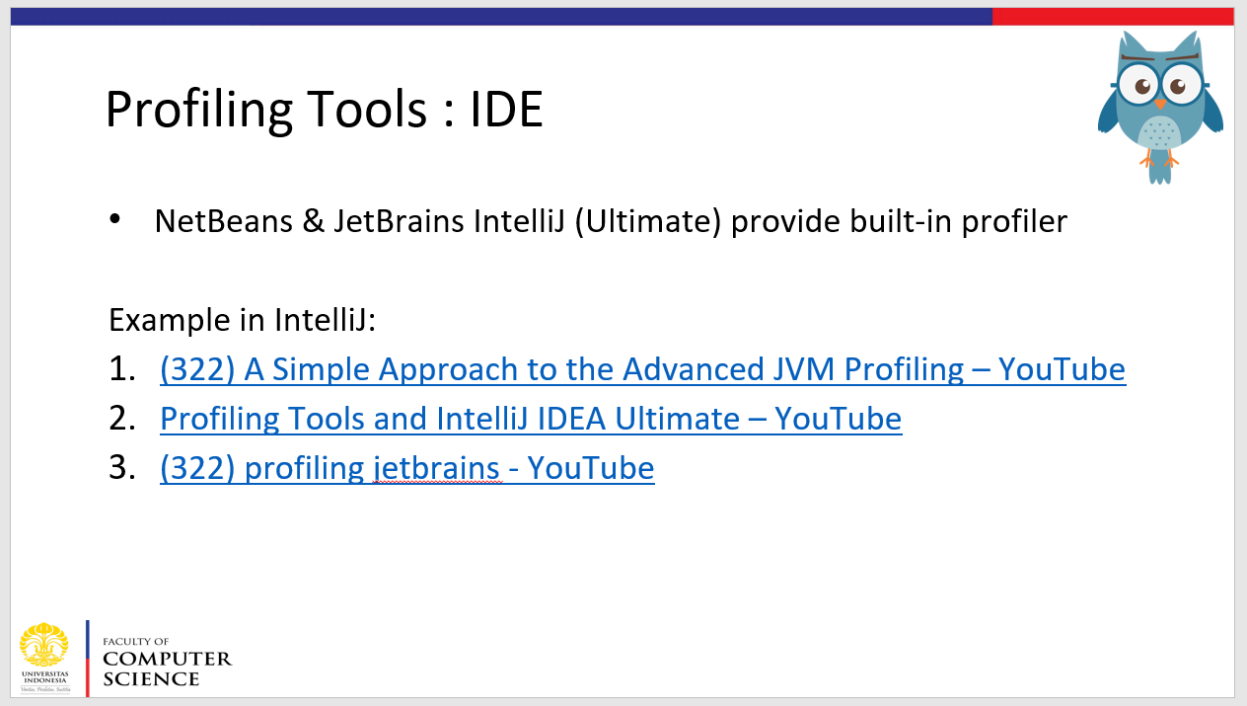


The Java Development Kit (JDK) comes equipped with a comprehensive suite of tools designed for monitoring and profiling the Java Virtual Machine (JVM), enabling developers to fine-tune application performance and resolve issues. Among these tools, ‘jcmd’, ‘jconsole’, ‘jmap’, ‘jinfo’, ‘jstack’, ‘jstat’, and ‘jvisualvm’ stand out for their utility in different contexts.

‘jcmd’ is a versatile tool that allows for issuing diagnostic command requests to the JVM, making it possible to perform a wide range of monitoring and management tasks. For instance, using ‘jcmd’ with the syntax ‘jcmd pid/main_class VM.uptime’ lets developers ascertain the uptime of the JVM, providing insights into the duration of the application's operation. Similarly, ‘jcmd pid/main_class VM.system_properties’ reveals the system properties of the JVM, offering a detailed view of the environment settings. Additionally, ‘jcmd pid/main_class VM.flags’ lists the JVM startup flags, aiding in the understanding of the JVM's configuration and operational parameters.

These tools, integral to the JDK, furnish developers with powerful capabilities for JVM monitoring, profiling, and tuning. By leveraging such tools, developers can gain deep insights into the JVM's behavior, optimize application performance, and ensure efficient resource utilization, thereby enhancing the overall quality and stability of Java applications.

Profiling Tools : IDE



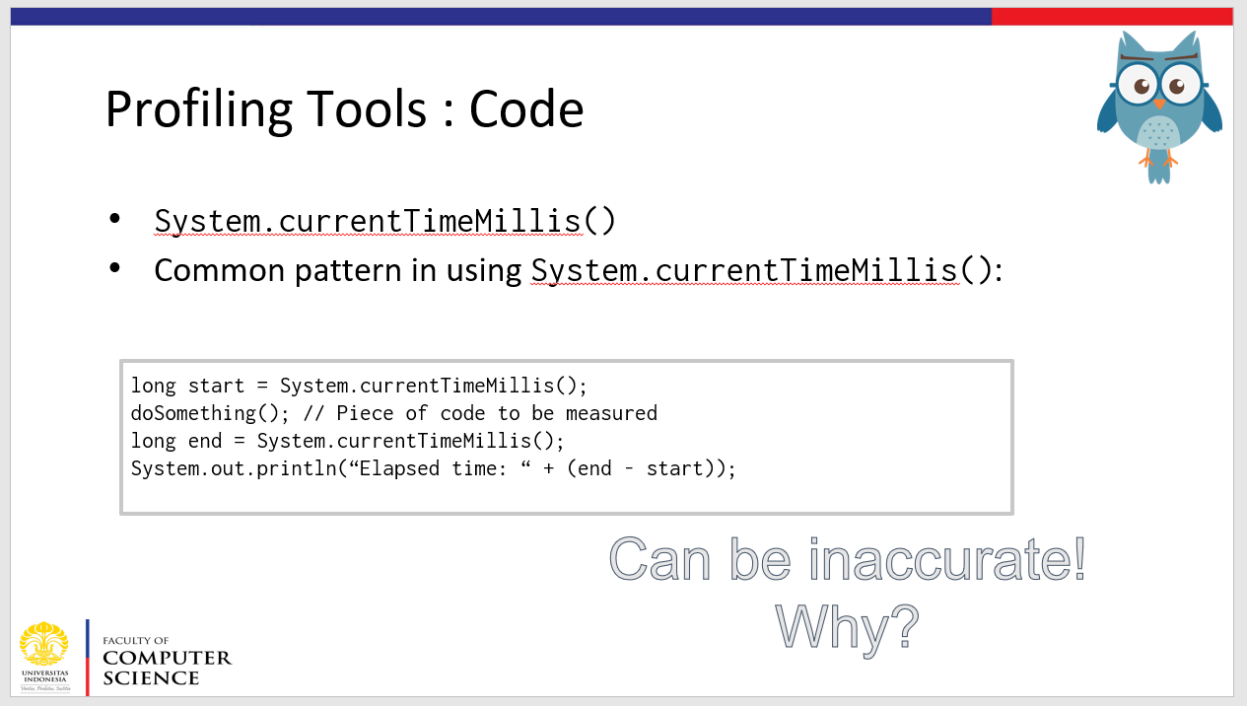
Integrated Development Environments (IDEs) like NetBeans and IntelliJ IDEA Ultimate edition incorporate built-in profilers, offering developers a seamless and integrated experience for optimizing and troubleshooting Java applications. These profilers are designed to provide detailed insights into the application's performance, including CPU usage, memory allocation, and thread activity, without the need for external tools or complex setup procedures.

With these IDEs, developers can easily initiate profiling sessions directly within their development environment, allowing for real-time analysis and immediate feedback on the performance impact of their code changes. This integrated approach simplifies the process of identifying and resolving performance bottlenecks, memory leaks, and concurrency issues.

NetBeans profiler, for instance, offers a comprehensive suite of profiling features, including CPU, memory, and threads profiling, enabling developers to pinpoint performance issues quickly. Similarly, the JetBrains IntelliJ IDEA Ultimate edition includes a powerful profiler that aids in detecting memory leaks, understanding memory allocation, and analyzing CPU usage.

By leveraging the built-in profiling capabilities of these IDEs, developers can significantly enhance the efficiency, reliability, and performance of their Java applications, ensuring a smoother and more productive development workflow.

Profiling Tools : Code



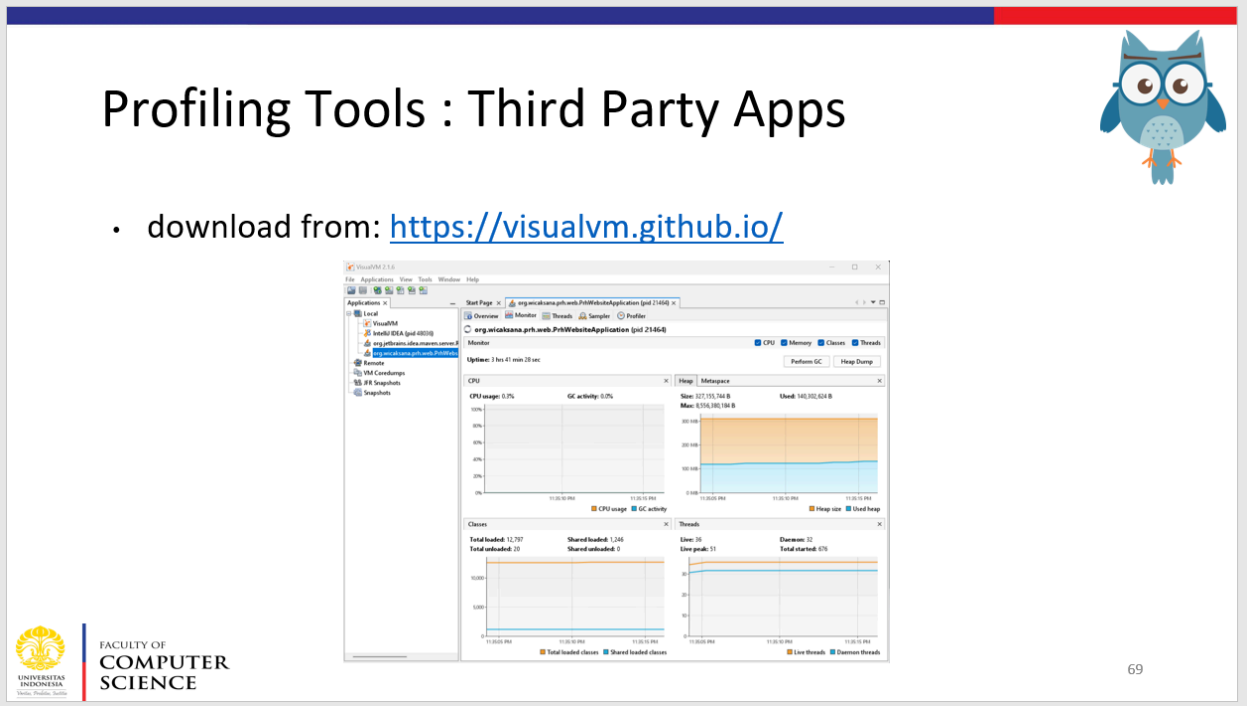
In the quest for optimizing Java applications, developers often leverage profiling tools to pinpoint performance bottlenecks. One such rudimentary yet powerful tool is the use of 'System.currentTimeMillis()', a method that captures the current time in milliseconds. This method is instrumental in measuring the execution time of specific code segments, providing a simple way to assess performance efficiency.

The sample code provided illustrates a common pattern for employing 'System.currentTimeMillis()'

Here, 'start' captures the time before the execution of 'doSomething()', a placeholder for any block of code whose performance is being evaluated. Following the execution, 'end' captures the time again. The difference between 'end' and 'start' gives the total execution time in milliseconds, which is then printed to the console. This method is exceptionally straightforward and can be easily inserted into various parts of an application to measure the performance of distinct operations.

While this approach offers a quick and easy way to identify slow-executing parts of code, it's important to note that it measures only execution time. For a more comprehensive analysis, including memory allocation, developers may need to explore advanced profiling tools.

Profiling Tools : Third Party Apps

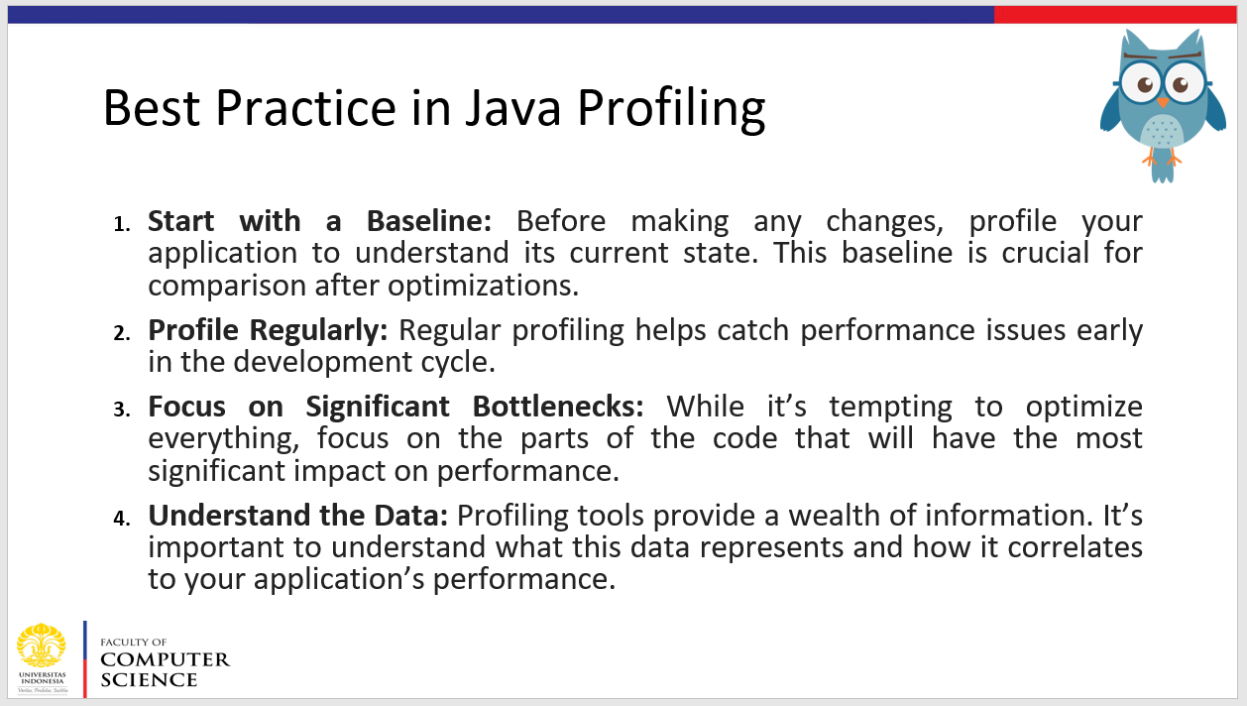


VisualVM is a powerful third-party profiling tool that provides a visual interface for monitoring, troubleshooting, and profiling Java applications. It combines several JDK command-line tools and lightweight profiling capabilities into a single intuitive graphical user interface, making it accessible to both developers and system administrators. With VisualVM, users can monitor application memory consumption, track garbage collection processes, analyze heap dumps, and observe thread states in real-time.

One of the key features of VisualVM is its ability to provide detailed CPU and memory profiling information. This includes the ability to take and analyze heap dumps, perform garbage collection analysis, and monitor the execution of threads. Additionally, VisualVM allows for the capture of snapshots of the JVM's state, which can be invaluable for post-mortem analysis or for sharing information with others.

VisualVM supports local and remote monitoring. The tool can attach to local applications automatically and monitor applications running on remote machines, provided the appropriate setup for remote communication is configured. This versatility makes VisualVM a go-to tool for diagnosing performance issues and optimizing Java application performance across various environments. Its extensible architecture also allows for plugins, further enhancing its capabilities to meet specific user needs.

Best Practice in Java Profiling

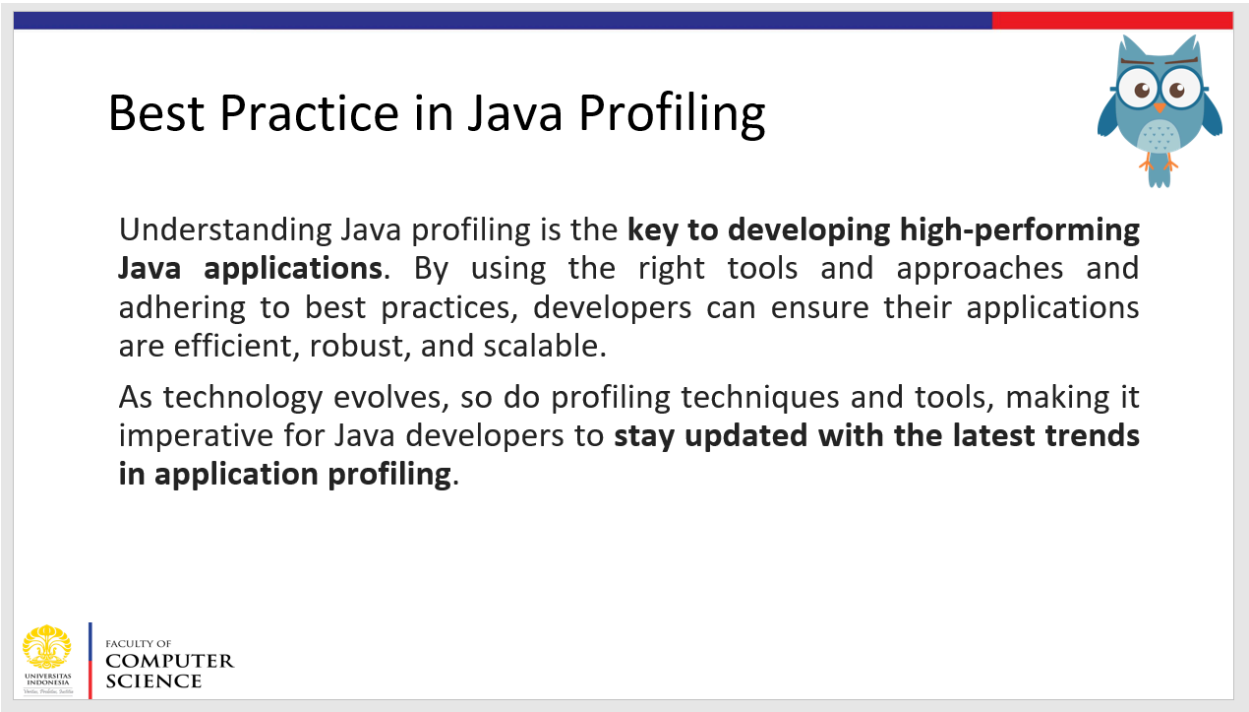


Adopting best practices in Java profiling is essential for optimizing application performance efficiently and effectively. Establishing a baseline is the first critical step; it involves profiling your application before implementing any changes to understand its initial performance characteristics. This baseline serves as a reference point, enabling you to measure the impact of subsequent optimizations accurately.

Regular profiling is another cornerstone of effective performance management. By integrating profiling into the development cycle, you can identify and address performance issues promptly, preventing them from escalating into more significant problems. This proactive approach ensures continuous performance improvement and helps maintain application efficiency.

However, it's essential to prioritize your efforts. Focus on resolving significant bottlenecks—areas in your code that have the most substantial impact on overall performance. Optimizing these critical paths can yield significant improvements, enhancing user experience and application scalability. Given the limited resources and time, concentrating on these areas ensures that optimization efforts deliver maximum return on investment.

Lastly, the data generated by profiling tools is only as valuable as your ability to interpret it. Understanding the nuances of the profiling data, such as CPU usage, memory allocation patterns, and thread activity, is crucial. This knowledge allows you to make informed decisions about where and how to optimize, translating complex data into actionable insights.



Java profiling stands as a critical component in the development of high-performing applications, offering a pathway to optimize efficiency, ensure robustness, and enhance scalability. In the realm of Java development, the choice of profiling tools and methodologies plays a pivotal role in identifying performance bottlenecks, memory leaks, and threading issues. These tools provide valuable insights into the runtime behavior of applications, enabling developers to make informed decisions about optimizations.

Moreover, the landscape of Java profiling is continuously evolving, with new tools and techniques emerging to address the challenges of modern application development. This dynamic environment demands that Java developers remain abreast of the latest advancements in profiling technologies and methodologies. Staying updated with these trends is not just about leveraging new tools but also about adopting a mindset geared towards performance optimization from the outset of development.

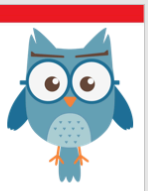
Embracing the best practices in Java profiling—such as regular performance checks, focusing on critical bottlenecks, and understanding the intricacies of profiling data—ensures that applications not only meet their performance criteria but also adapt to changing requirements and environments. As technology progresses, the ability of developers to effectively profile and optimize Java applications will continue to be a defining factor in the success of software projects, emphasizing the importance of continuous learning and adaptation in the field of Java development.

Profiling Practice with IntelliJ Profiler

- IntelliJ Profiler is an integrated tool within the **IntelliJ IDEA Ultimate Edition**, designed to aid developers in analyzing and improving the performance of their Java applications.
- This profiler provides real-time insights into various aspects of application performance, including CPU usage, memory allocation, garbage collection activity, and thread concurrency.

FACULTY OF COMPUTER SCIENCE

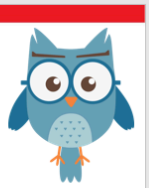
Profiling Practice with IntelliJ Profiler



Source Code : https://github.com/muhammad-khadafi/demo-profiling

FACULTY OF COMPUTER SCIENCE

Profiling Practice with IntelliJ Profiler



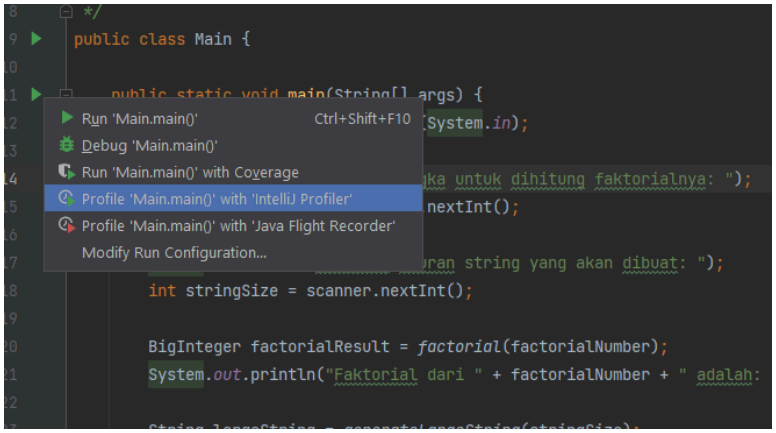
- This source code is an example of a simple program that contains a factorial function and a function to create a string with a specified number of characters.
- To observe performance issues, try providing inputs with sufficiently large numbers for both functions. For example, calculate the factorial of 100 and create a string with 100,000 characters.

```
Masukkan angka untuk dihitung faktorialnya: 100
Masukkan ukuran string yang akan dibuat: 100000
```

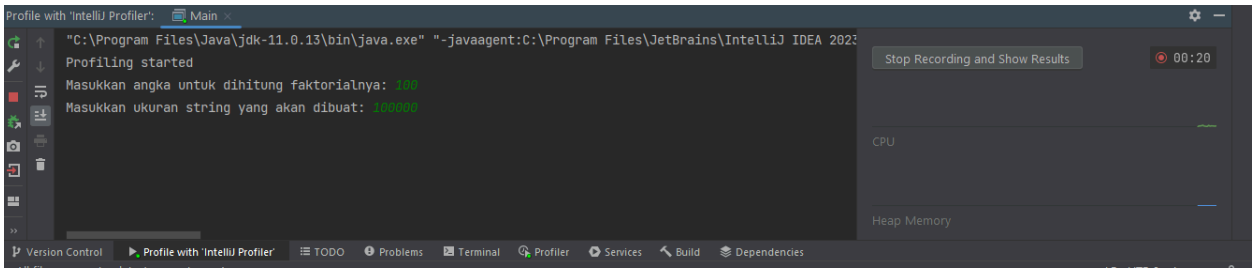
FACULTY OF COMPUTER SCIENCE

1. Clone the source code from git repository
   https://github.com/muhammad-khadafi/demo-profiling

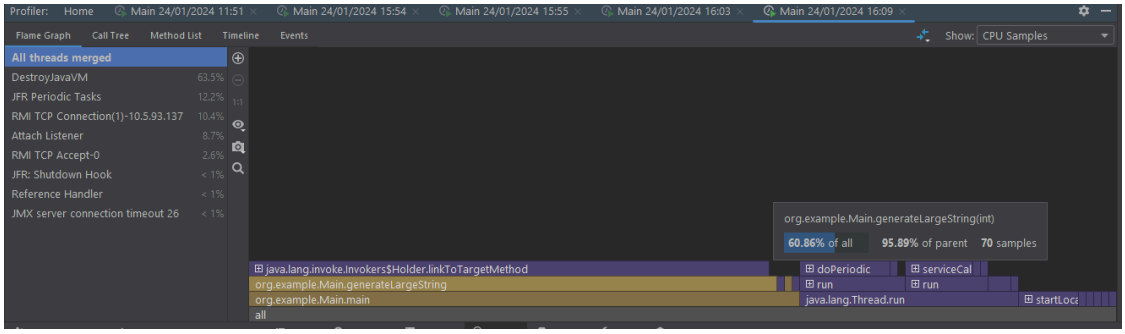
2. Start Profiling : Right click on run icon -> Profile..



3. Profiler start to record the application performance

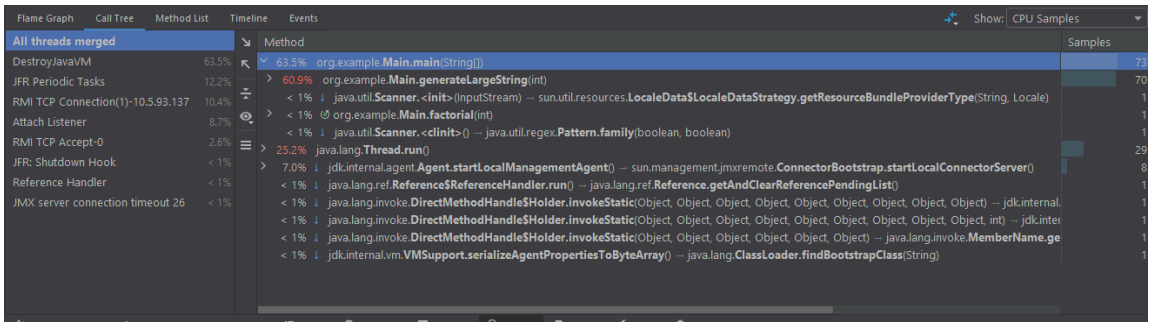


4. View the result (profiler data). this is flame graph view



The Flame Graph view in IntelliJ Profiler is a visualization tool that provides a graphical representation of application performance, focusing on the call stack of profiled code. It displays the most resource-intensive methods and call paths as wider, "hotter" flames, making it easier to identify performance bottlenecks. Each block on the graph represents a function in the stack, with the width indicating the function's relative execution time. By offering a quick, at-a-glance understanding of where the application spends most of its time, developers can efficiently pinpoint inefficiencies and optimize their code. This method simplifies the complex task of performance analysis by highlighting critical paths through visual cues.

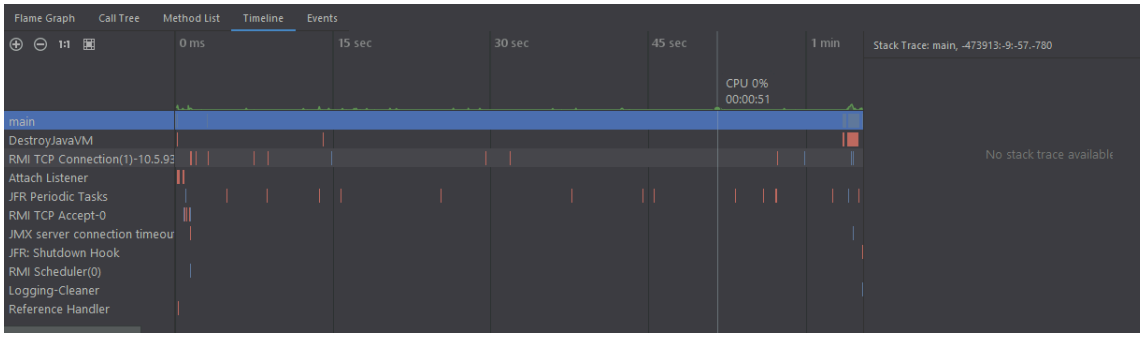
5. View others presentation. this is call tree view



The Call Tree view in IntelliJ Profiler offers a detailed hierarchical representation of the calls made by your application during profiling. It breaks down the execution path of your application into a tree structure, showing each method call, its callers, and callees. This allows developers to see not just which methods are consuming the most time, but also

how they are being called within the application's workflow. The view helps in identifying performance hotspots by displaying the cumulative execution time spent in each method and its children, making it easier to pinpoint inefficiencies and understand the application's behavior in depth.

6. Timeline view



The Timeline view in IntelliJ Profiler provides a chronological visualization of your application's performance data, allowing you to observe how metrics like CPU usage, memory consumption, and garbage collection events change over time. This view is essential for understanding the dynamics of your application under load or during specific operations. By examining the timeline, developers can identify patterns, spikes, or anomalies in resource usage, pinpoint when performance issues occur, and correlate them with specific actions or events within the application. This insight is invaluable for diagnosing issues and optimizing application performance.

7. Refactor the Code to Increase Performance
    - Factorial : using for loop instead of recursive and remove the Thread.sleep

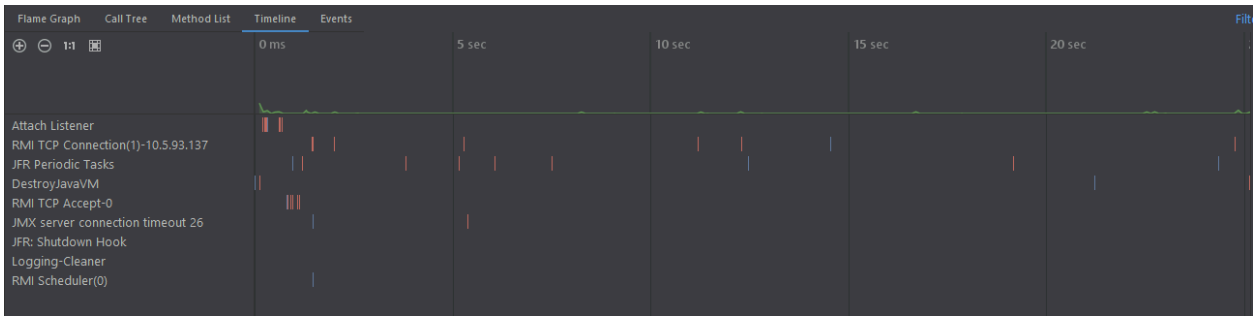
```
36      private static BigInteger factorial(int n) {
37          BigInteger result = BigInteger.ONE;
38          for (int i = 2; i <= n; i++) {
39              result = result.multiply(BigInteger.valueOf(i));
40          }
41          return result;
42      }
```

- GenerateLargeString : Using StringBuilder instead of String.

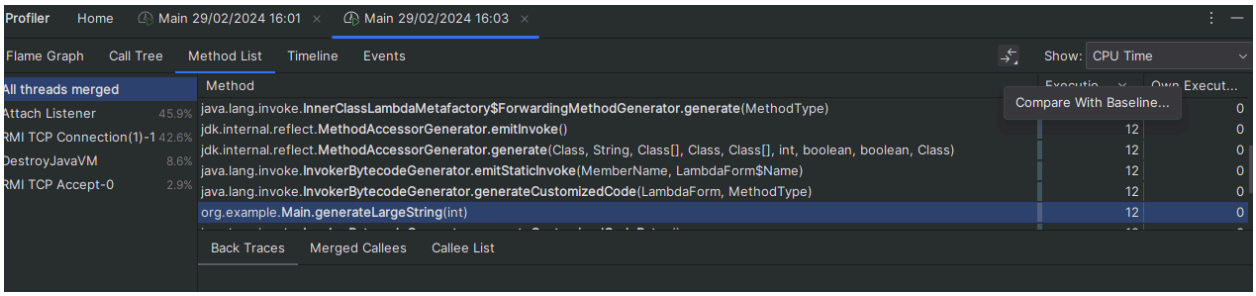
```
private static String generateLargeString(int size) {
    StringBuilder builder = new StringBuilder(size);
    for (int i = 0; i < size; i++) {
        builder.append("a");
    }
    return builder.toString();
}
```

- **The code has been written for you. Change to branch 'refactor'*

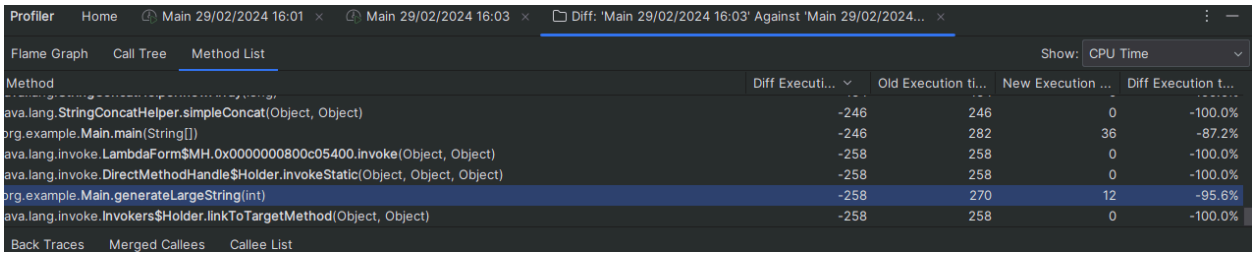
8. See the difference



9. You can display a view that compares the results of one profiling session with another. Select a profiling results tab, then click the "compare with baseline" button and choose another profiling result you want to compare.

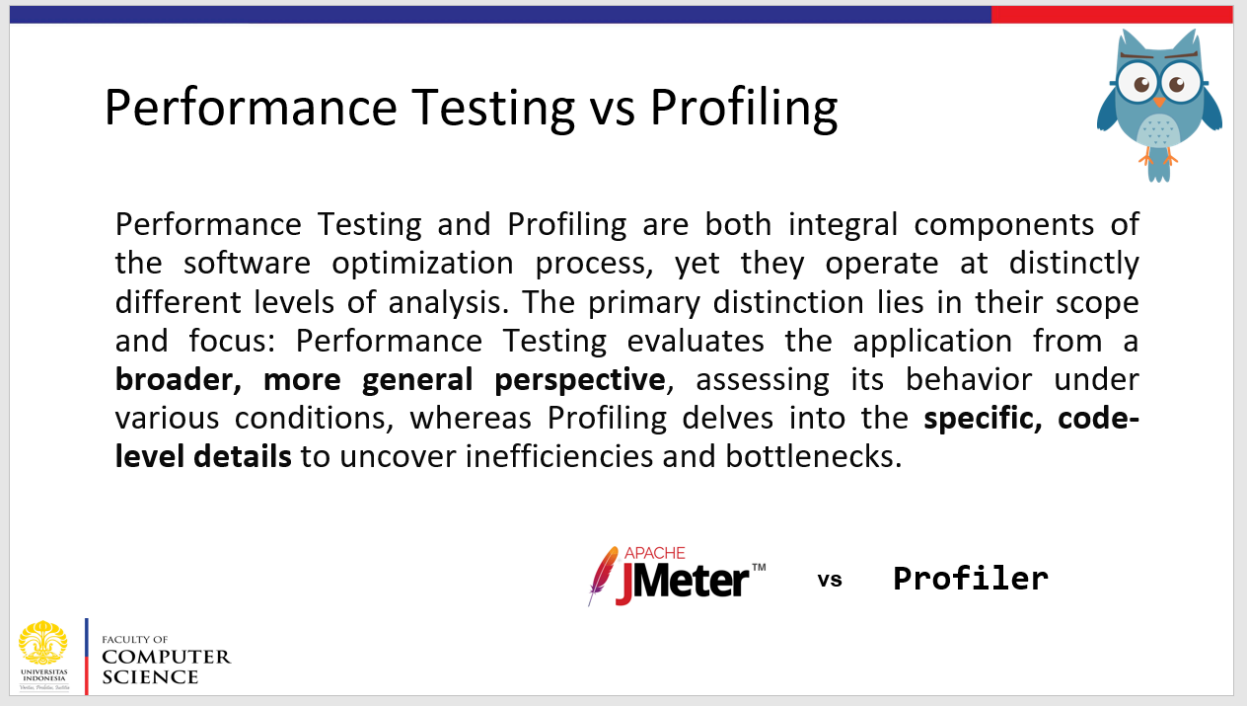


10. You can immediately see the results of the differences between the two profiling sessions. In the screenshot below, the improvement in CPU time between before and after refactoring is 95.6%.



The 95.6% improvement was achieved from the CPU Time performance before refactoring, which was 270 seconds, and then reduced to 12 seconds after refactoring. This indicates that the optimization process carried out had a very significant impact on performance enhancement.

Performance Testing vs Profiling



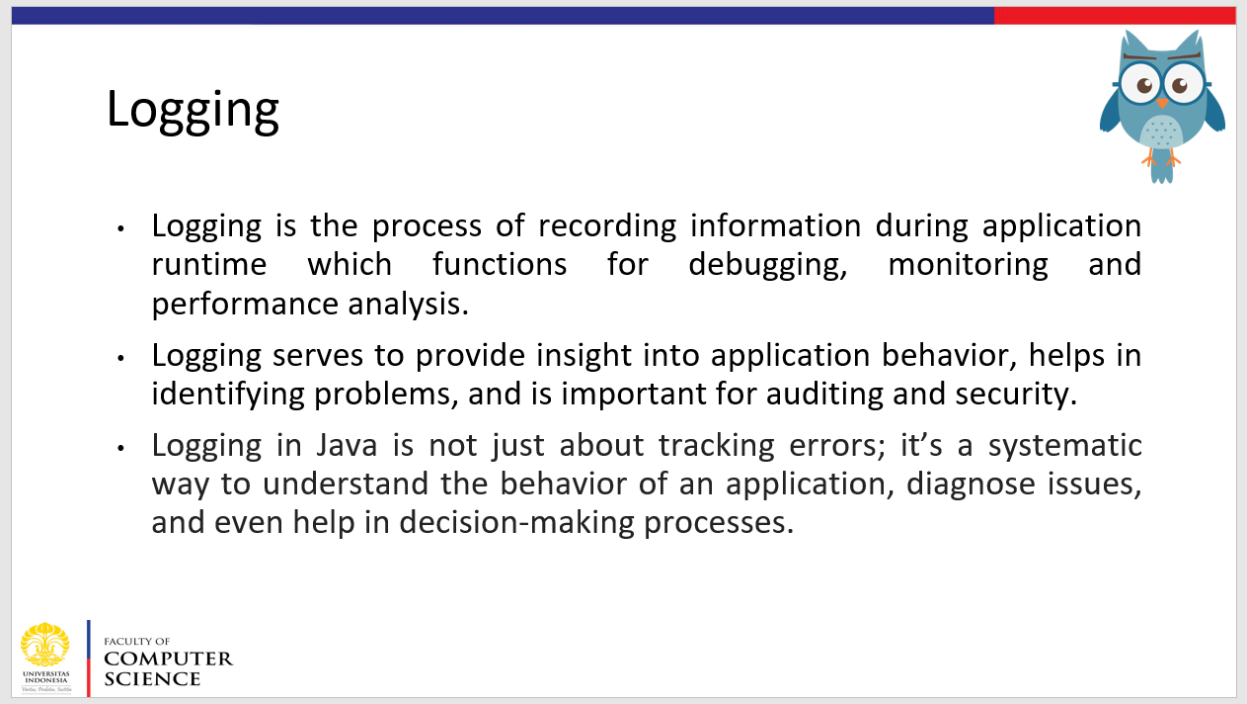
Performance testing and profiling are two complementary approaches in the software development lifecycle aimed at enhancing application performance, yet they operate at distinctly different levels of granularity. Performance testing is broader in scope, designed to assess the application's overall behavior and efficiency under various load conditions. It simulates real-world usage scenarios to ensure the application can handle expected traffic volumes and performs well

under stress. This type of testing is critical for identifying system-wide issues that could impact user experience, such as slow response times or scalability concerns.

Conversely, profiling offers a deep dive into the application's internal operations, focusing specifically on code-level performance. It is a more detailed and precise technique that identifies the exact lines of code or processes that are inefficient or consuming disproportionate resources. Profiling is instrumental in uncovering hidden bottlenecks, memory leaks, or excessive CPU usage that could degrade application performance. By providing insights into the intricacies of code execution, profiling enables developers to make targeted optimizations, enhancing code efficiency and application responsiveness.

In essence, while performance testing evaluates the application from a macro perspective, profiling delves into the fine details, offering a comprehensive and nuanced understanding of performance across all levels of the application.

Logging



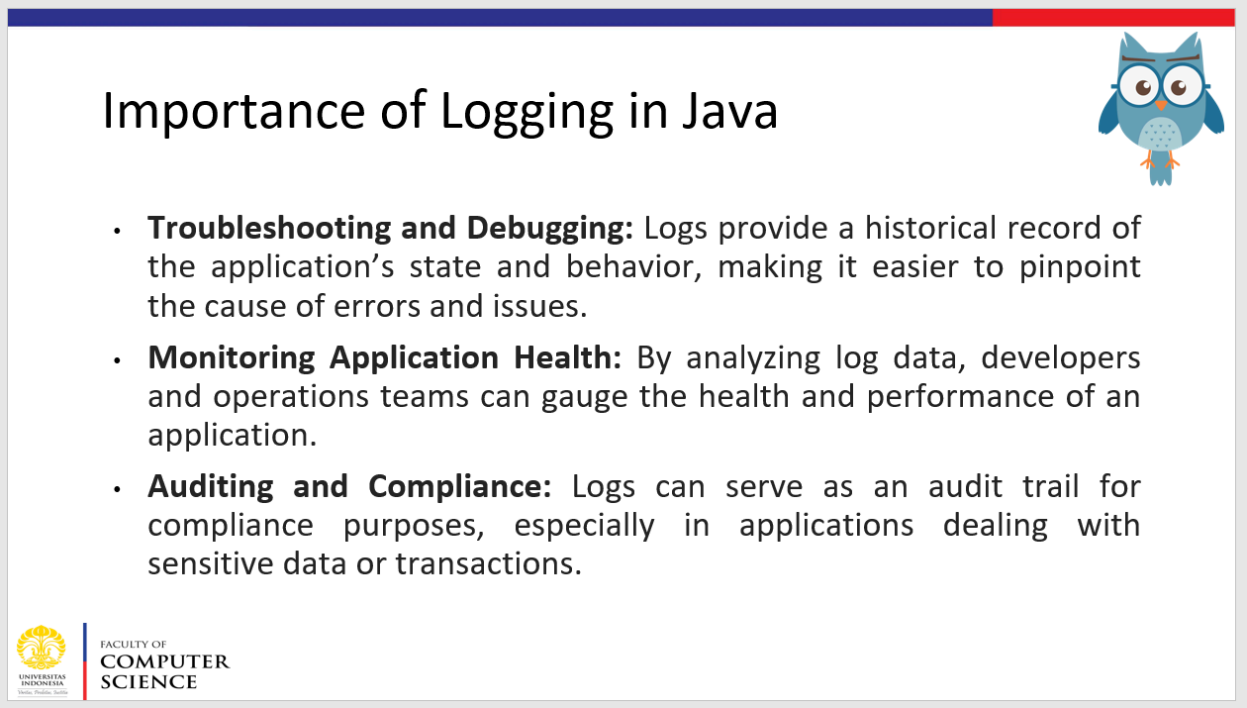
Logging is a critical aspect of software development, serving as the backbone for a myriad of essential activities, including debugging, monitoring, and performance analysis. By recording detailed information about an application's runtime operations, logging provides invaluable insights into the inner workings of the system. It is not merely a tool for capturing errors; it plays a

pivotal role in understanding application behavior, identifying and diagnosing problems, and facilitating effective decision-making.

In the context of Java development, logging transcends basic error tracking to become a comprehensive diagnostic mechanism. Developers can leverage logging to trace the flow of execution, monitor system state changes, and assess performance metrics. This level of detail is crucial for pinpointing the root causes of issues, optimizing application performance, and ensuring the system operates as intended.

Moreover, logging is indispensable for auditing and enhancing security. It allows for the historical analysis of application activity, helping to identify suspicious behavior or unauthorized access attempts. By maintaining detailed logs, organizations can meet compliance requirements, conduct thorough investigations when security incidents occur, and strengthen their overall security posture.

Importance of Logging in Java



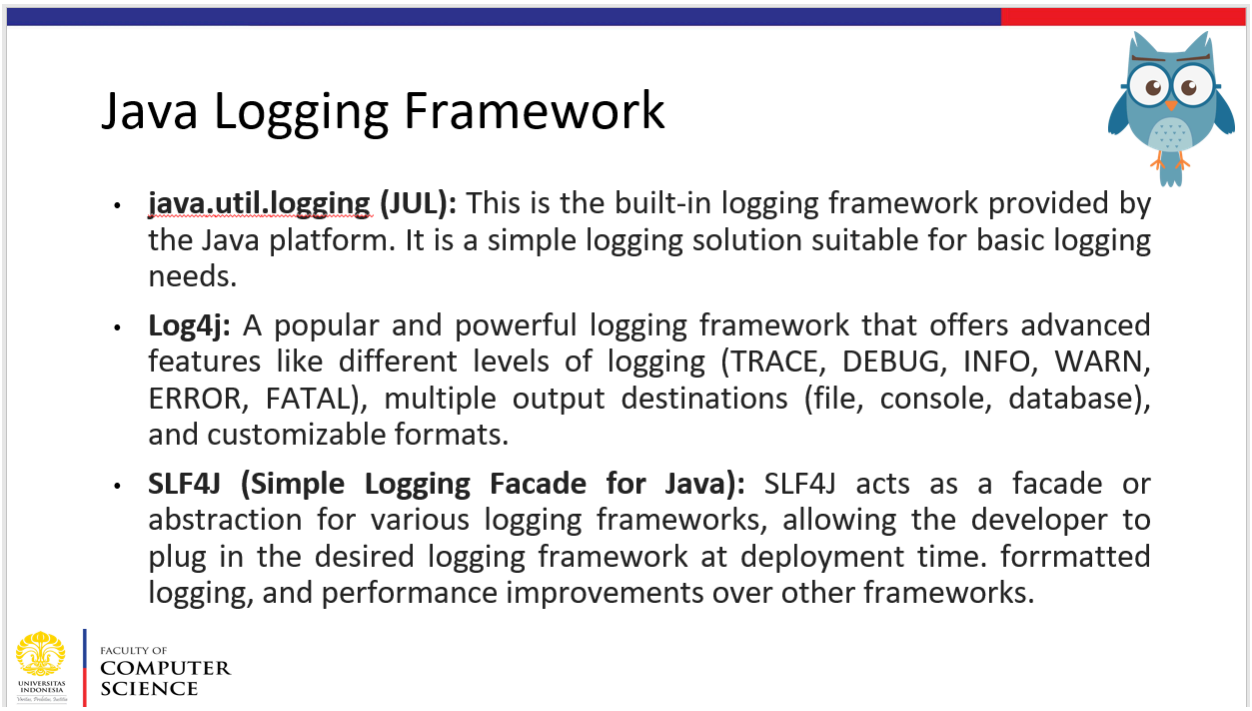
The importance of logging within Java applications cannot be overstated, as it plays a crucial role across various facets of software development and maintenance. Logging facilitates troubleshooting and debugging by providing a detailed historical record of an application's state

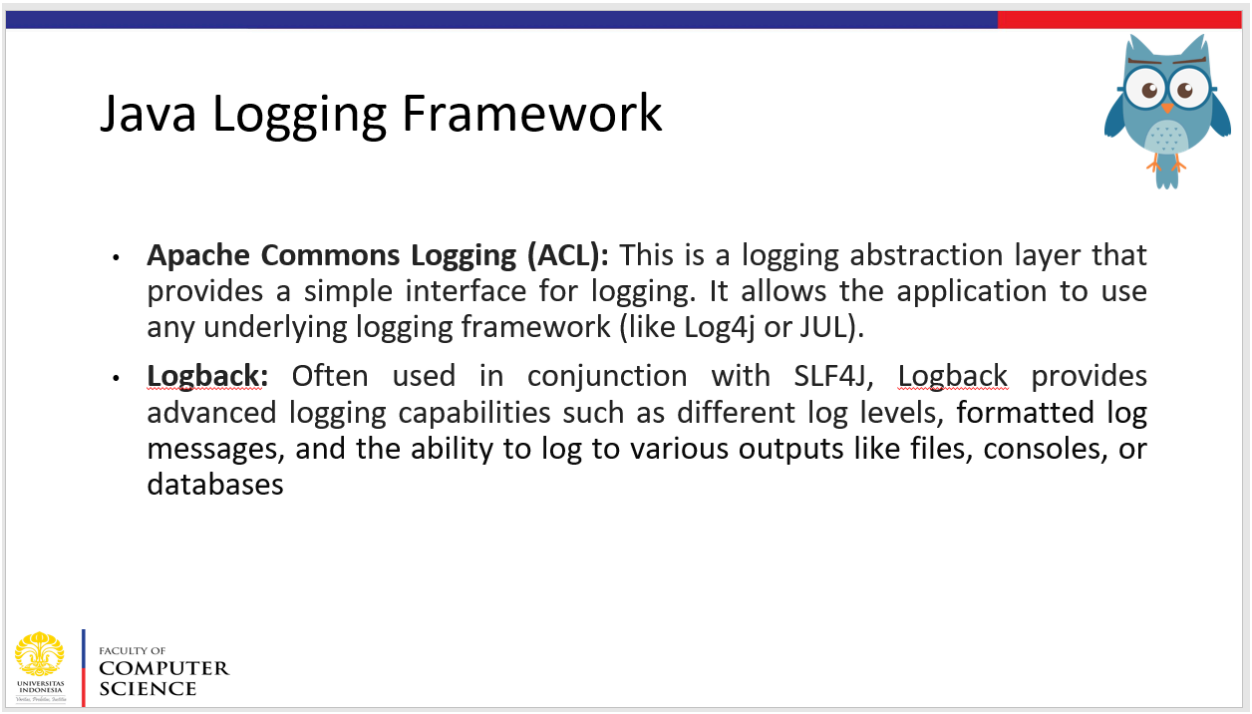
and behavior over time. This record is instrumental in identifying and pinpointing the root causes of errors and issues, enabling developers to resolve problems more efficiently and effectively.

Beyond debugging, logging is vital for monitoring the health and performance of an application. Through the systematic analysis of log data, developers and operations teams can assess the application's operational status, identify performance bottlenecks, and detect early signs of potential issues before they escalate. This proactive monitoring ensures the application remains reliable and performs optimally under varying conditions.

Furthermore, logging serves a critical function in auditing and compliance, especially for applications that handle sensitive data or conduct transactions. Logs can act as a comprehensive audit trail, documenting every action, event, and interaction within the application. This documentation is essential for demonstrating compliance with regulatory standards, ensuring accountability, and maintaining the integrity of the application. In summary, logging in Java is indispensable for effective troubleshooting, monitoring application health, and adhering to auditing and compliance requirements, underscoring its significance in the development and operation of secure and reliable software.

Java Logging Framework





In the Java ecosystem, logging frameworks play a crucial role in the development and maintenance of applications by providing tools for recording runtime information, which is vital for debugging, monitoring, and performance analysis. Among these, several frameworks stand out due to their features, flexibility, and ease of use.

- java.util.logging (JUL): Integrated directly into the Java platform, JUL offers a straightforward logging solution that caters to basic needs. It's an accessible option for developers looking for a simple way to log information without external dependencies. Despite its simplicity, it provides a range of functionality suitable for many standard applications.

- Log4j: Recognized for its robustness and flexibility, Log4j is a favorite among developers for advanced logging requirements. It supports various log levels—TRACE, DEBUG, INFO, WARN, ERROR, FATAL—allowing detailed control over log output. Moreover, Log4j facilitates logging to multiple destinations, including files, consoles, and databases, with the ability to customize log formats according to the developer's needs. Its comprehensive feature set makes it ideal for complex applications requiring nuanced logging strategies.

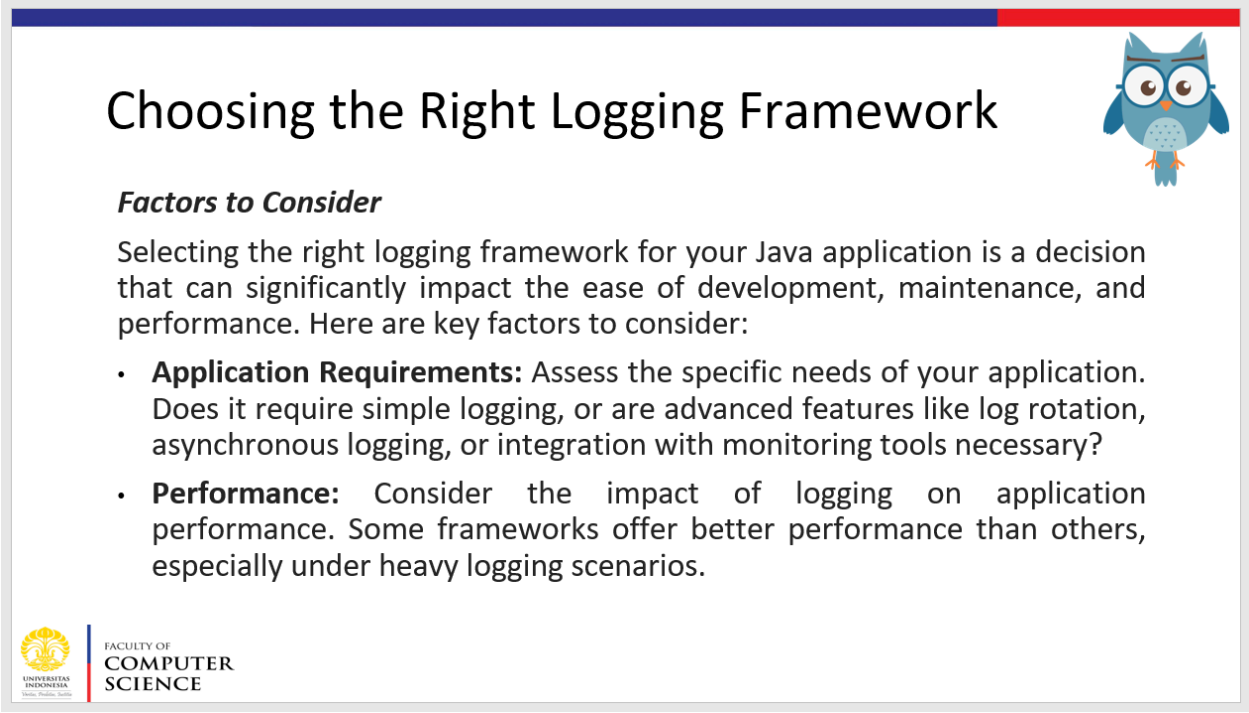
- SLF4J (Simple Logging Facade for Java): Serving as an abstraction layer, SLF4J enables developers to interface with different logging frameworks, such as Log4j or JUL, seamlessly. This flexibility allows choosing or switching the underlying logging implementation without changing the application code, offering a versatile approach to logging. SLF4J is particularly beneficial in environments where the logging framework might change based on deployment contexts or performance considerations.

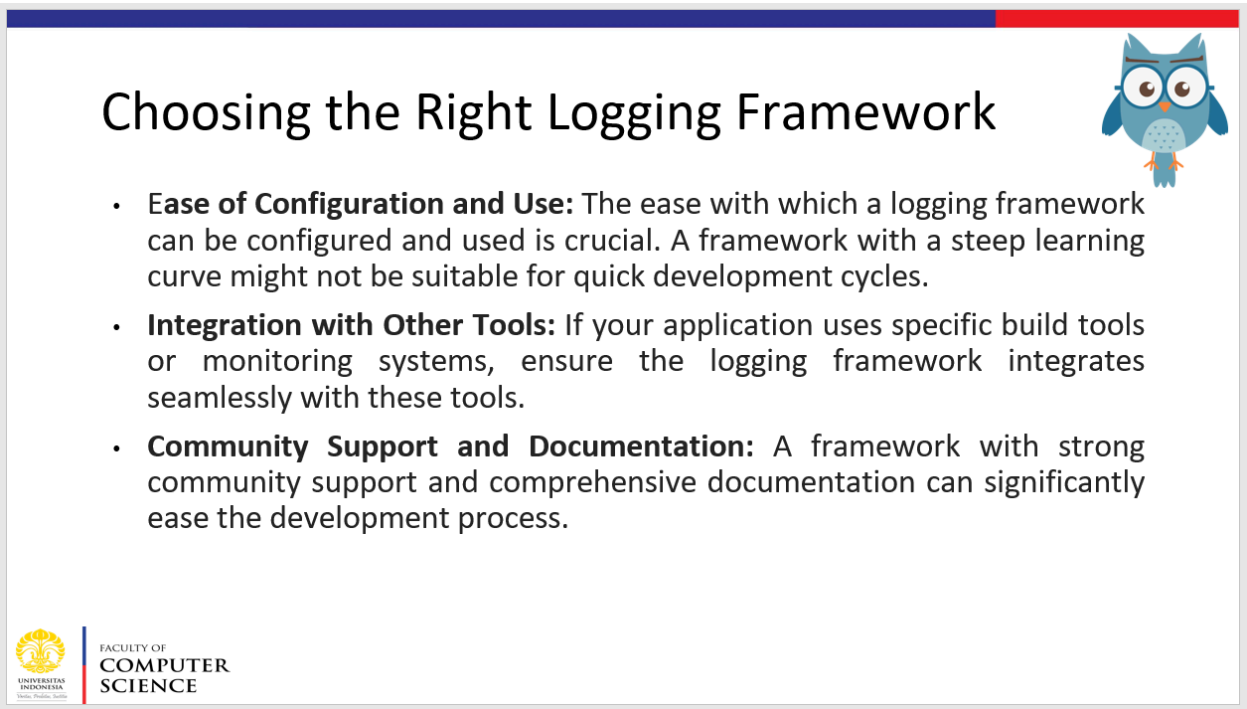
- Apache Commons Logging (ACL): Similar to SLF4J, ACL provides an abstraction layer that decouples the application from the actual logging framework. By offering a uniform logging interface, it ensures that applications remain independent of the logging implementation, supporting flexibility and ease of maintenance.

- Logback: As the designated successor to Log4j, Logback is often used in conjunction with SLF4J. It excels in providing advanced features like varied log levels and the ability to format log messages and direct them to different outputs. Logback is praised for its speed, configuration flexibility, and small memory footprint, making it a powerful choice for modern Java applications seeking efficient and effective logging solutions.

Together, these frameworks form the backbone of Java logging, each offering distinct advantages tailored to different development needs. From simple logging with JUL to advanced, flexible logging with Log4j, SLF4J, ACL, and Logback, Java developers have a comprehensive set of tools to ensure their applications are not only functional but also maintainable and performant.

Choosing Logging Framework





Selecting the appropriate logging framework for a Java application is a pivotal decision that influences not just the developmental workflow but also the application's maintainability and operational performance. To make an informed choice, several critical factors need to be evaluated:

1. Application Requirements: The initial step involves a thorough assessment of the application's logging needs. While some applications may suffice with basic logging functionalities, others might necessitate advanced features like log rotation (to manage log file sizes), asynchronous logging (for minimizing impact on application performance), or compatibility with specific monitoring tools for real-time analysis. Understanding these requirements is essential for selecting a framework that aligns with the application's complexity and operational demands.

2. Performance: The efficiency of a logging framework under load is a paramount consideration. The impact of logging on the overall performance of an application cannot be understated, especially in high-throughput environments where logging operations could potentially become bottlenecks. Frameworks vary in their performance optimization techniques; hence, choosing one that offers optimizations like asynchronous logging or minimal overhead can be crucial for maintaining application responsiveness.

3. Ease of Configuration and Use: A logging framework's learning curve and ease of integration into the development process are significant factors. Frameworks that offer straightforward configuration and intuitive APIs can accelerate development and facilitate easier maintenance.

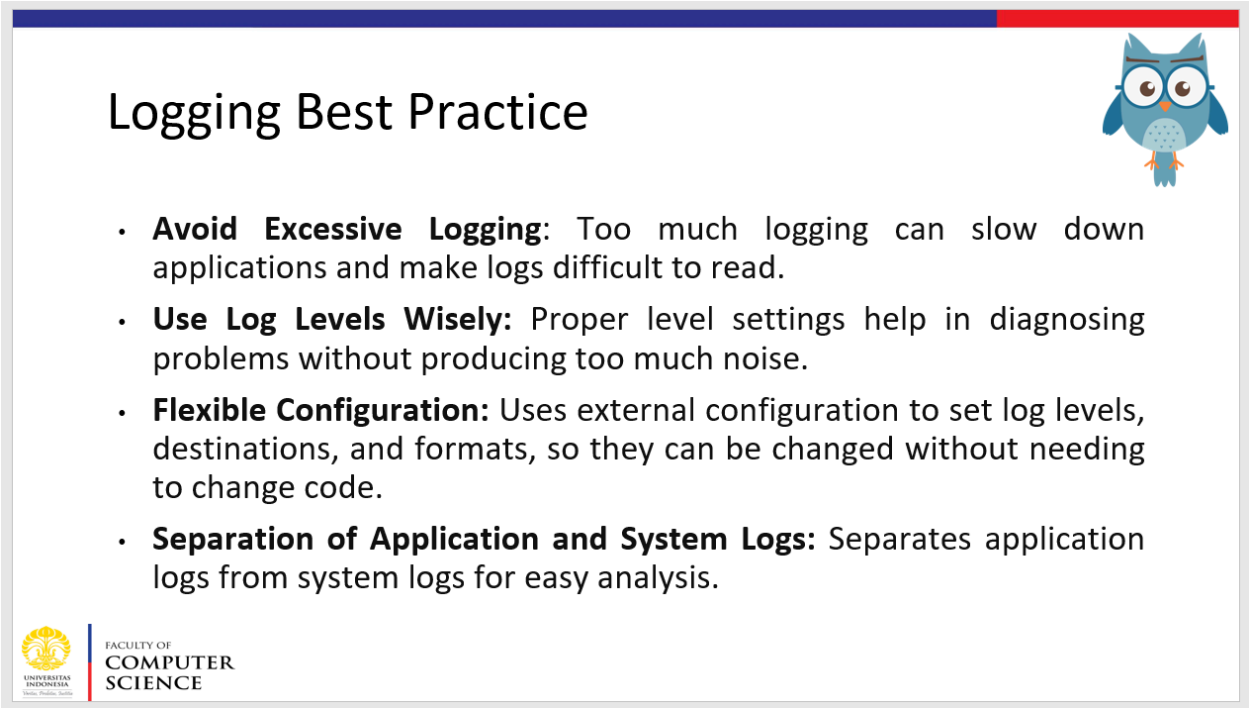
Complex frameworks, while powerful, may not be the best fit for projects with tight deadlines or teams with limited experience in logging practices.

4. Integration with Other Tools: The logging framework's compatibility with the existing ecosystem—such as build tools (Maven, Gradle), IDEs, and monitoring systems (Splunk, ELK)—is vital. Seamless integration ensures that logging does not become a disruptive factor and leverages the full potential of the development and monitoring infrastructure.

5. Community Support and Documentation: Robust community support and well-maintained documentation are invaluable resources for resolving issues and implementing best practices. A well-supported logging framework not only facilitates smoother development and troubleshooting but also ensures long-term viability through regular updates and community-driven enhancements.

By carefully considering these factors, developers can choose a logging framework that not only meets the technical requirements of their Java application but also complements their development process, ensuring efficient logging practices that enhance application performance and maintainability.

Logging Best Practice



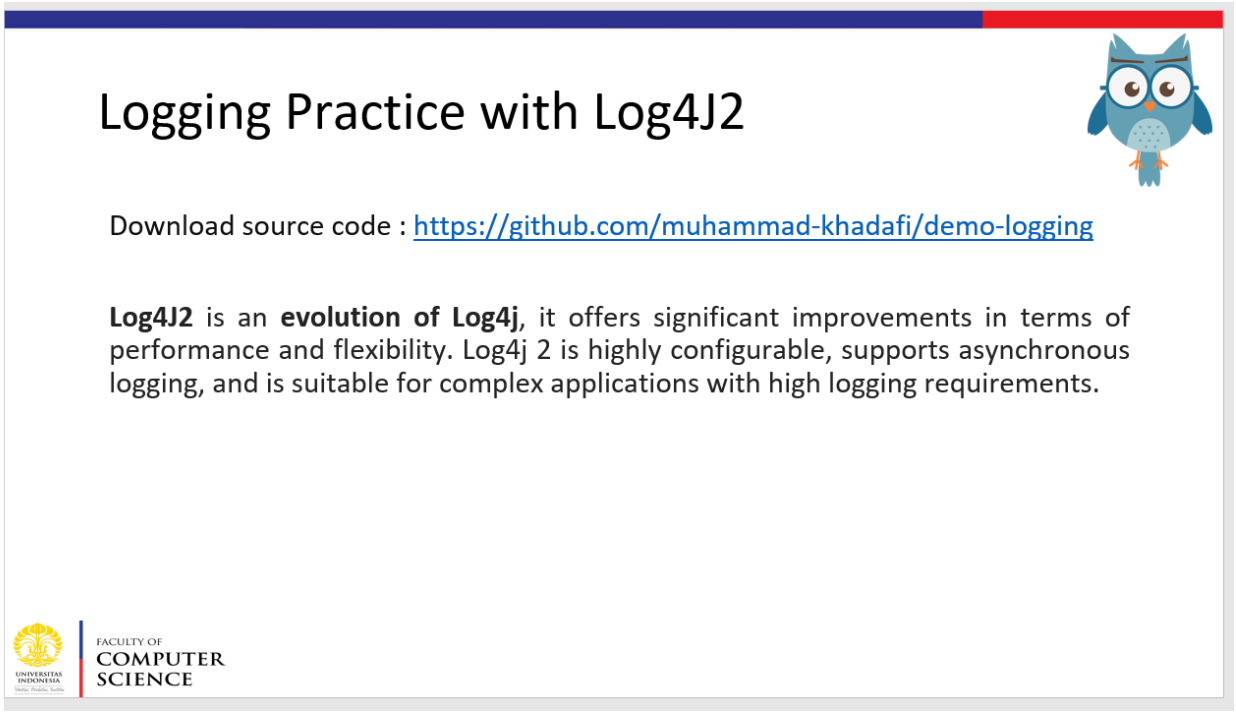
Adopting best practices in logging is essential for maintaining the performance, readability, and manageability of your Java applications. One critical recommendation is to avoid excessive logging. Overlogging can burden your application, leading to slower performance and creating log files that are cumbersome to navigate. It's vital to strike a balance, ensuring that logs capture essential information without overwhelming the system or the developers.

Equally important is the wise use of log levels. By categorizing log messages according to their severity (DEBUG, INFO, WARN, ERROR, FATAL), you can streamline problem diagnosis and keep the log output manageable. Proper level settings ensure that critical issues are highlighted without being drowned out by less significant messages.

Implementing a flexible configuration for your logging framework is another best practice. By externalizing settings such as log levels, output destinations, and message formats into configuration files, you can adjust logging behavior without altering the application's codebase. This flexibility allows for easy log management across different environments and situations.

Lastly, separating application logs from system logs is crucial for effective log analysis. This separation helps in quickly identifying issues specific to the application's logic versus those stemming from the underlying system or infrastructure. By organizing logs in this manner, you enhance their clarity and usefulness, facilitating faster troubleshooting and analysis.

Logging Practice with Log4J2



Log4J2 represents a significant upgrade from its predecessor, Log4j, providing a more powerful and flexible logging framework. It is engineered for high performance, particularly in demanding environments, and its design allows for extensive customization to fit various application needs. With its support for asynchronous logging, Log4J2 efficiently handles extensive logging tasks without compromising application performance, making it the ideal choice for complex applications that generate a lot of log data.

The code for this practice is already available, check it out this git repository : https://github.com/muhammad-khadafi/demo-logging

1. Create spring boot based project with maven building tools

2. Add dependencies

pom.xml (demo)

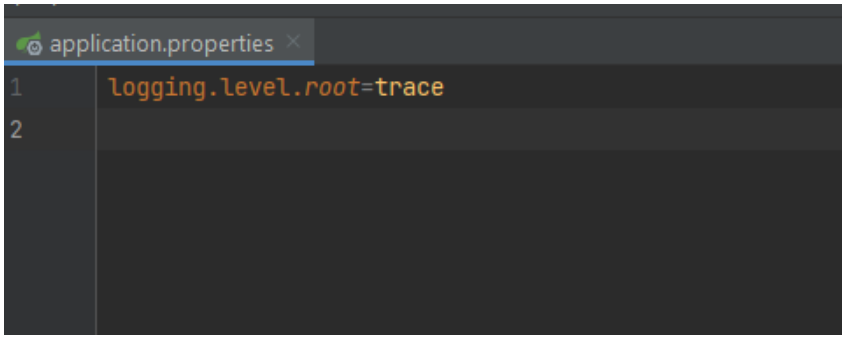
```
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
            <exclusions>
                <exclusion>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-starter-logging</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-log4j2</artifactId>
        </dependency>

    </dependencies>
```

3. Write the Log

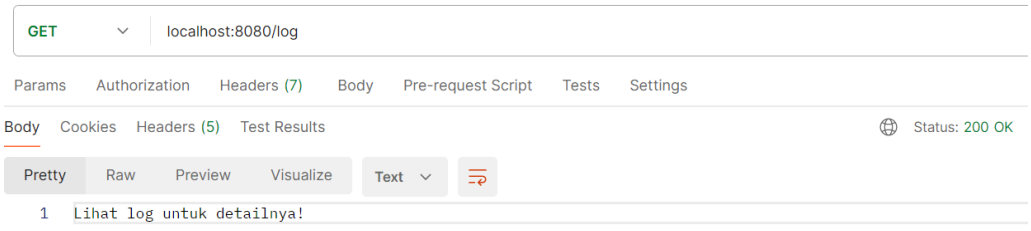
LoggingController.java

```
@RestController
public class LoggingController {

    5 usages
    private static final Logger logger = LogManager.getLogger(LoggingController.class);

    @GetMapping("/log")
    public String index() {
        logger.info("Pesan log INFO");
        logger.error("Pesan log ERROR");
        logger.warn("Pesan log WARNING");
        logger.debug("Pesan log DEBUG");
        logger.trace("Pesan log TRACE");

        return "Lihat log untuk detailnya!";
    }
}
```

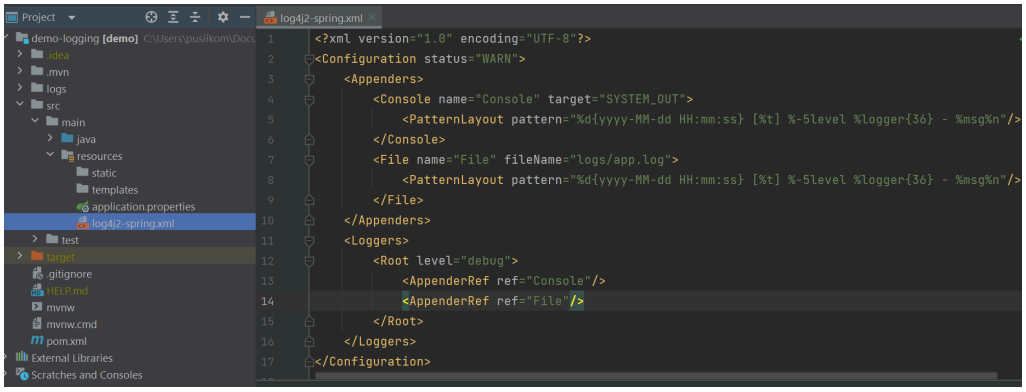
4. Display the log until trace level in application.properties, By default, the displayed logs are only up to the warning level



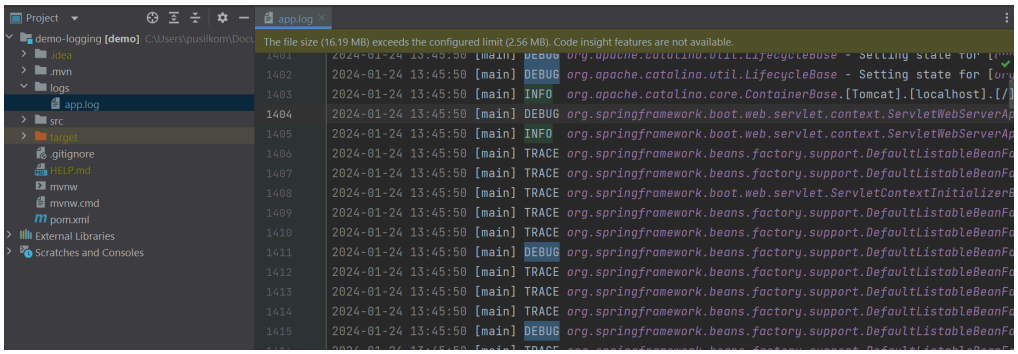
5. Run the app



6. Add config to write the log into file



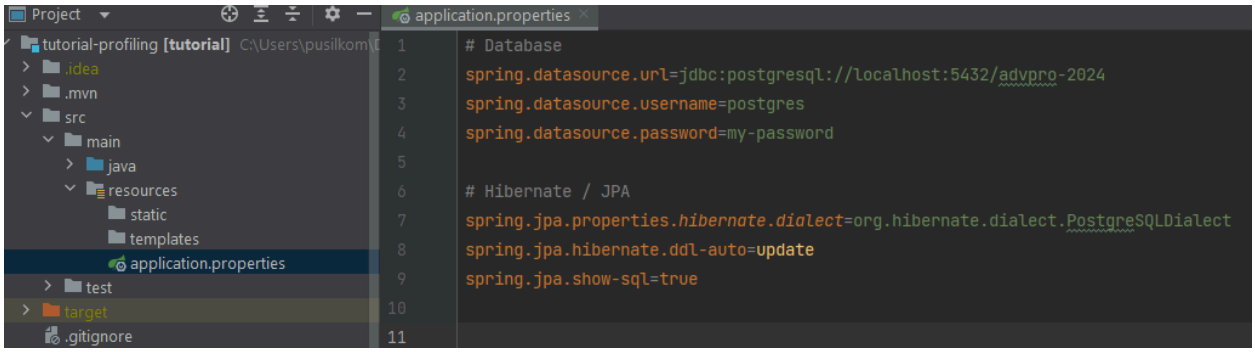
7. Run the app, the log will be automatically written to a file



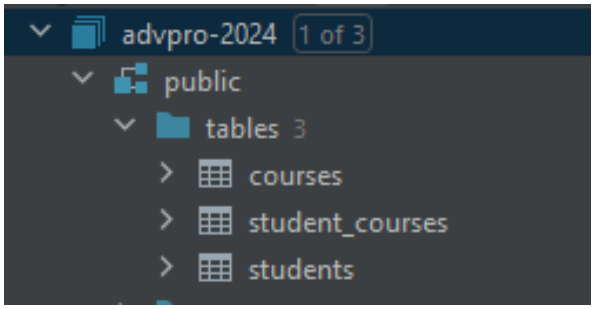
Tutorial & Exercise

Project Setup

1. Fork the source code. Fork the source code for this exercise at the following repository: https://github.com/muhammad-khadafi/exercise-profiling
2. Set up a PostgreSQL database, ensuring its configuration is correctly listed in the application.properties file.

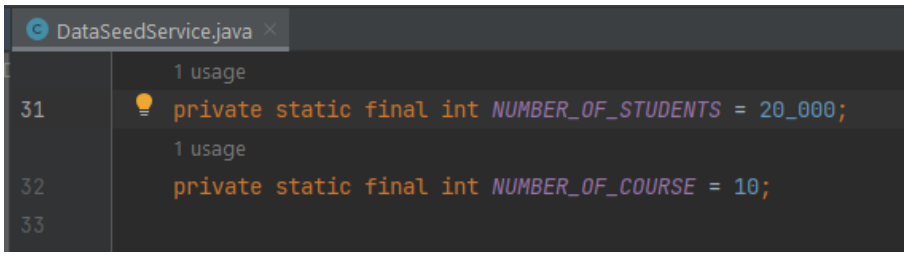


3. If your IDE does not automatically install dependencies, then execute the command: `mvn install`
4. Run the application using the command mvn spring-boot:run or click the run button in your IDE (IntelliJ).
5. This application uses DDL from JPA, so when it is first run, it will automatically create tables in the database. Ensure that these three tables are successfully created in your database.



6. After the application is successfully running, perform data seeding for the courses and students tables by accessing the following endpoint: ***http://localhost:8080/seed-data-master***

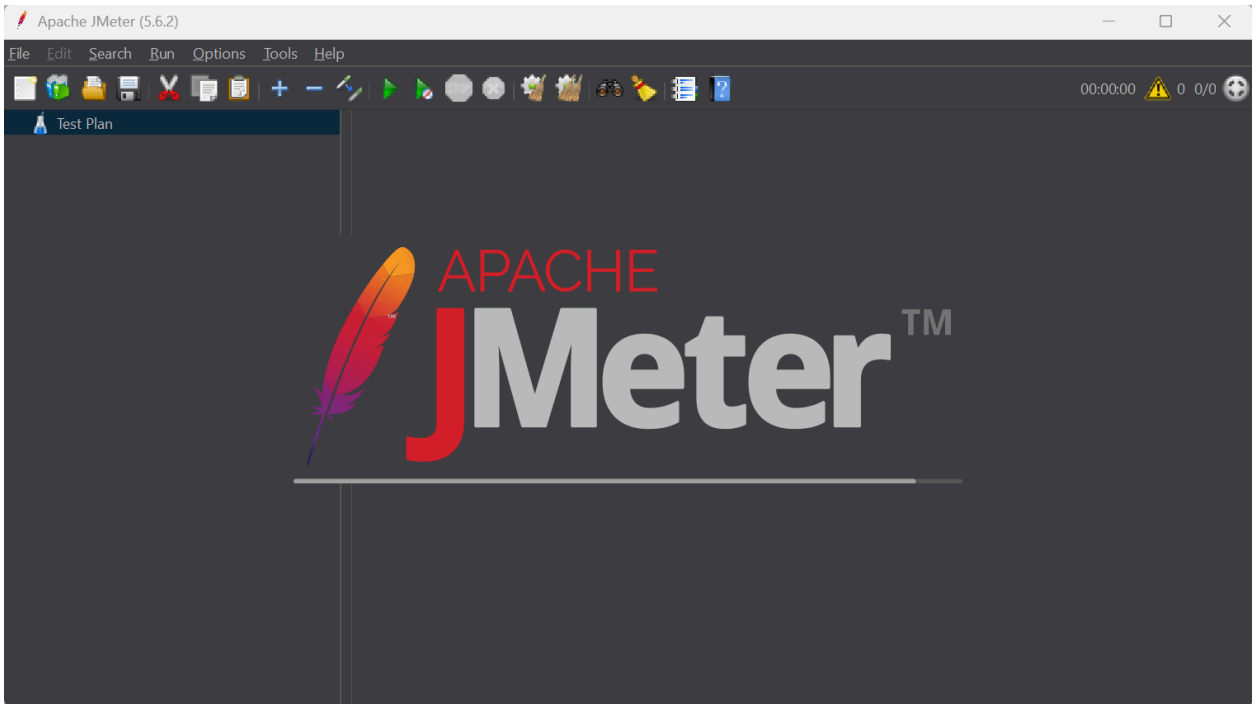
7. Also perform data seeding for the student_courses table by accessing the following endpoint: ***http://localhost:8080/seed-student-course***
8. If the data seeding process takes too long, you are allowed to reduce the number of student data generated, but ensure it is not too small so that the essence of performance testing and profiling processes can still be experienced. The data seeding settings are found in the DataSeedService.java file



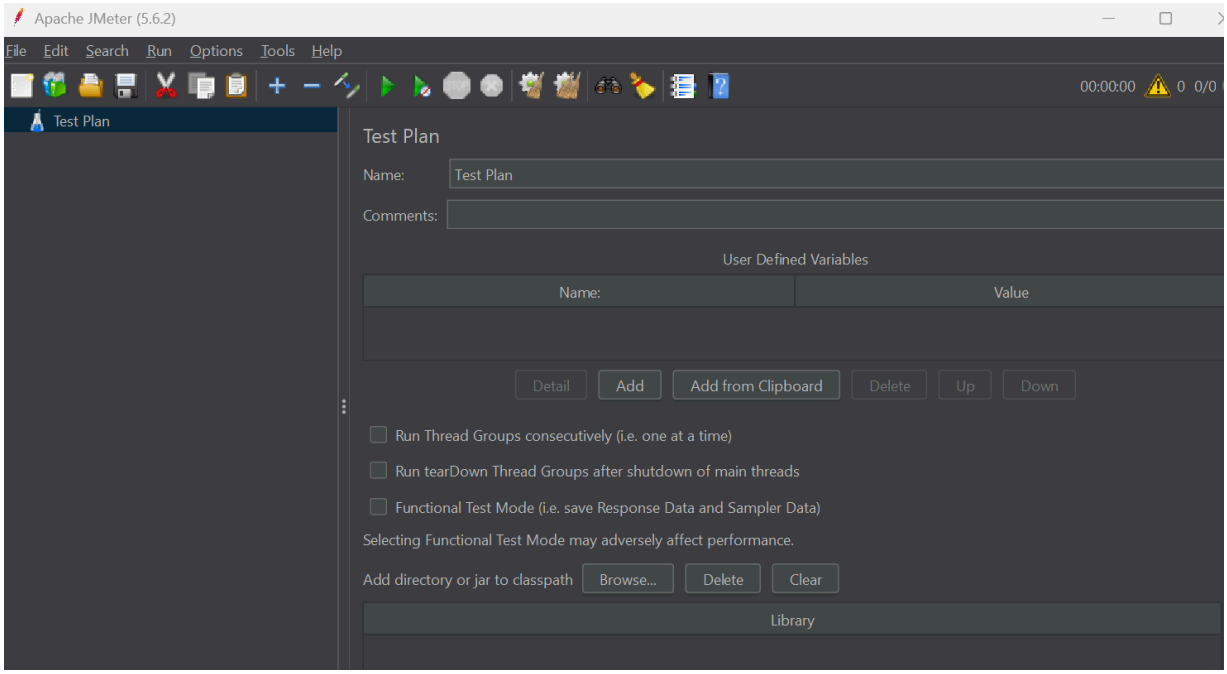
€ Commit your source code into your main branch in your repository

Performance Testing

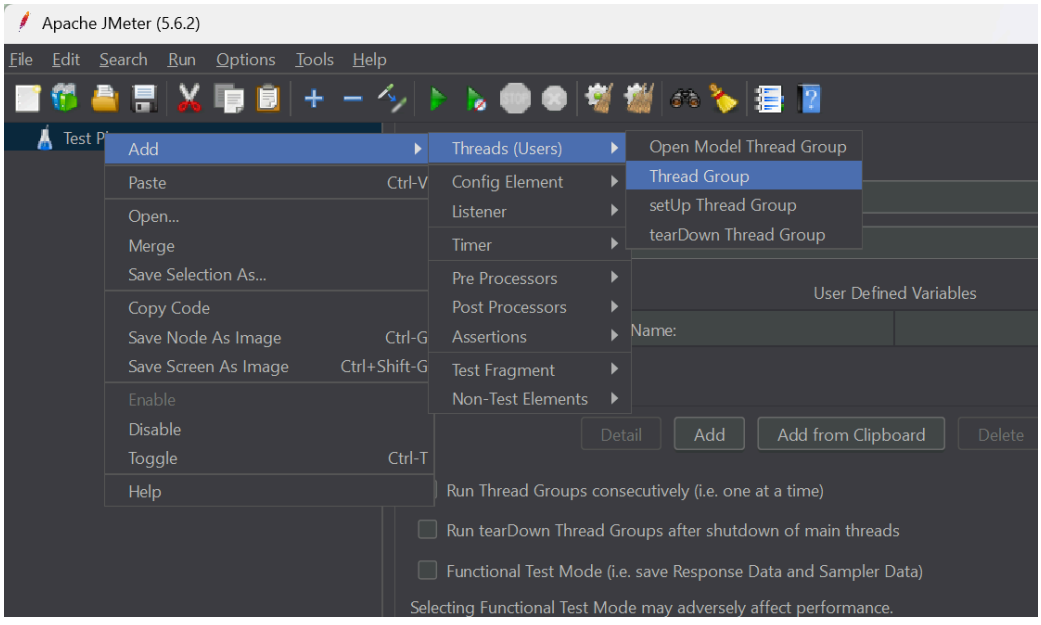
9. Run JMeter in GUI mode to create a test plan.

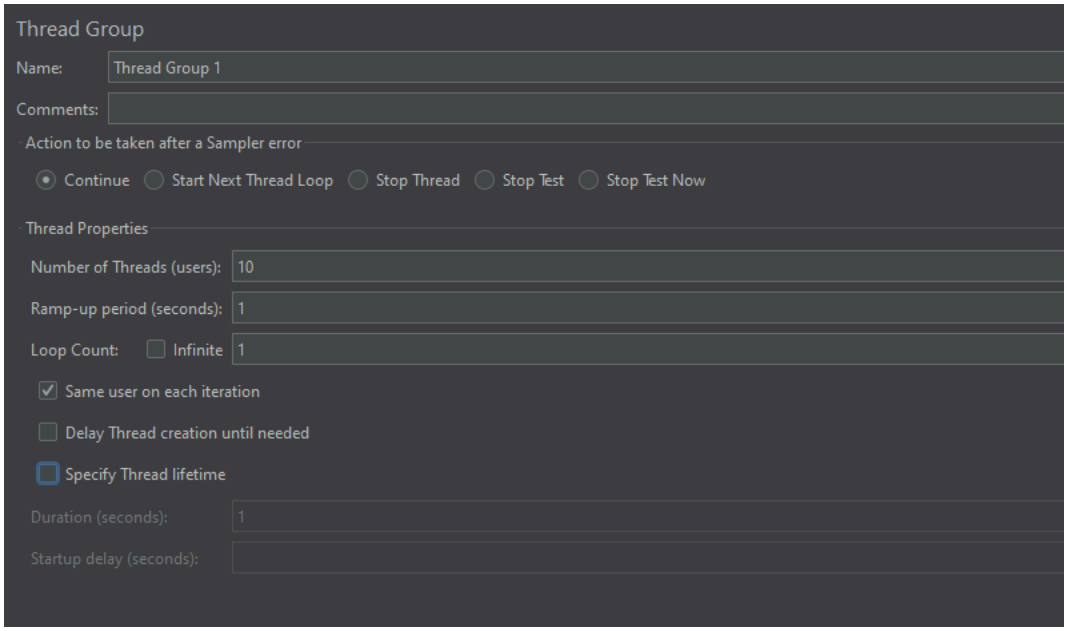


10. Create new test plan

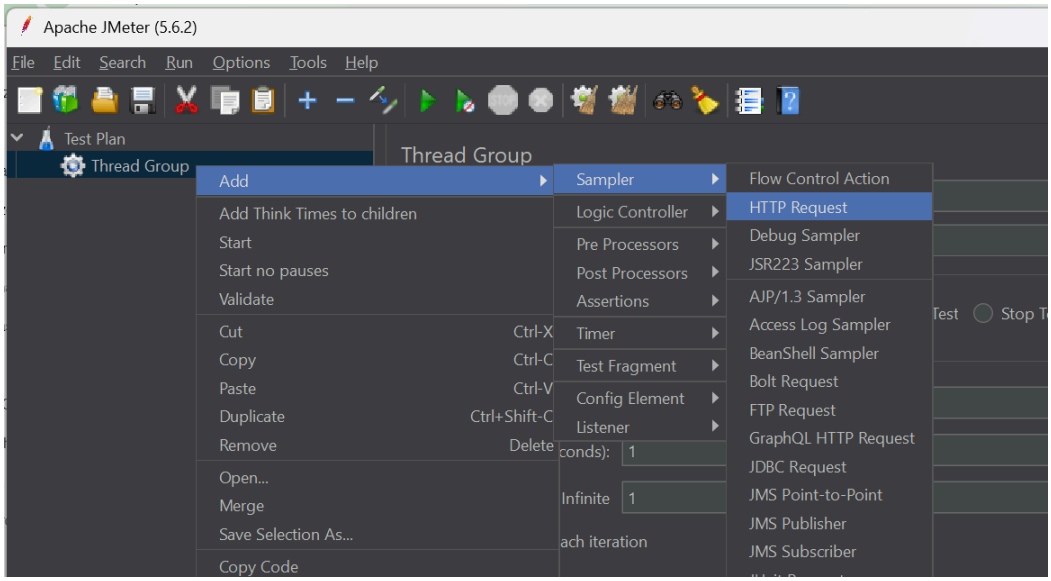


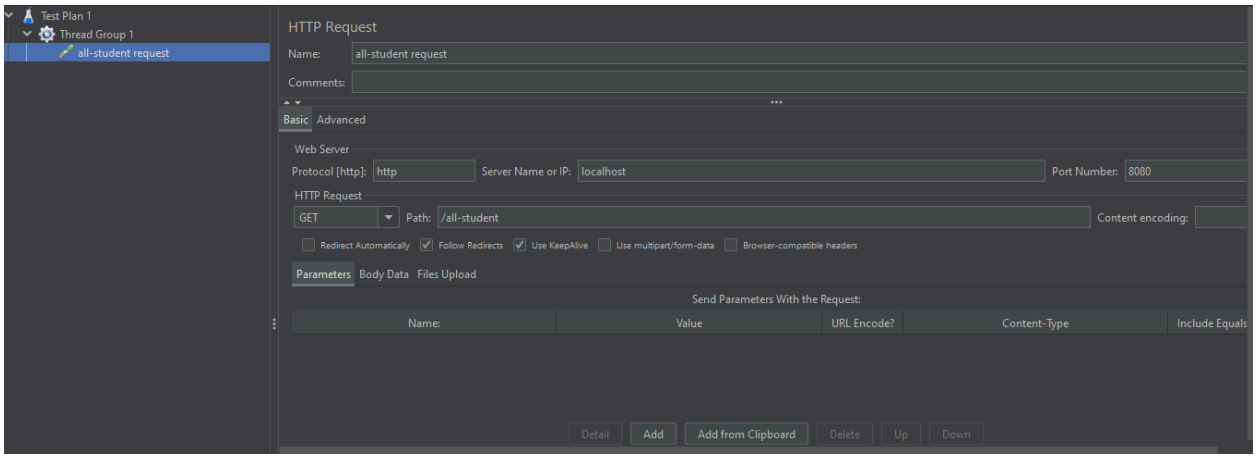
11. Create a thread group, simulating 10 users with a ramp-up period of 1 second.



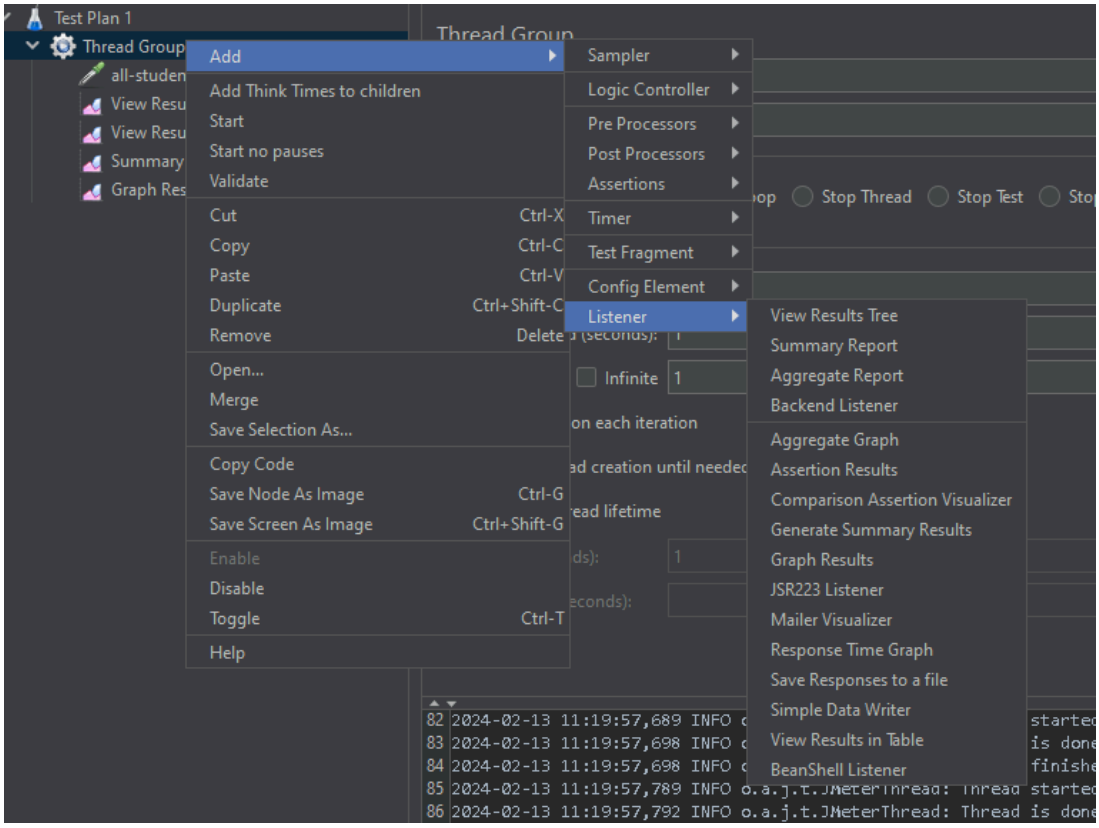


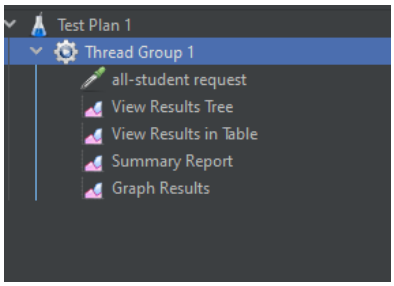
12. Add a sampler as an HTTP Request for the ***/all-student*** endpoint.



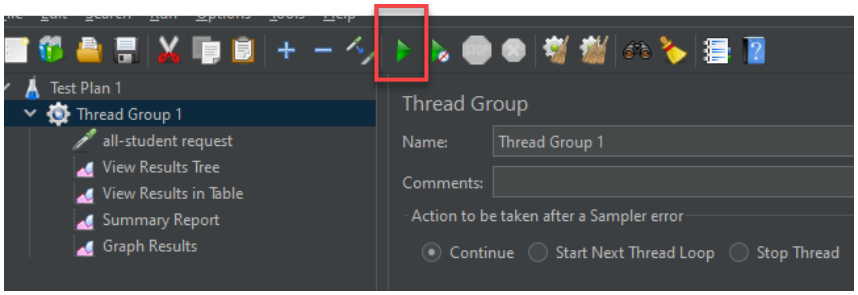


13. To view the test results we need listeners, add four listeners: View Results Tree, View Results in Table, Summary Report, and Graph Results.

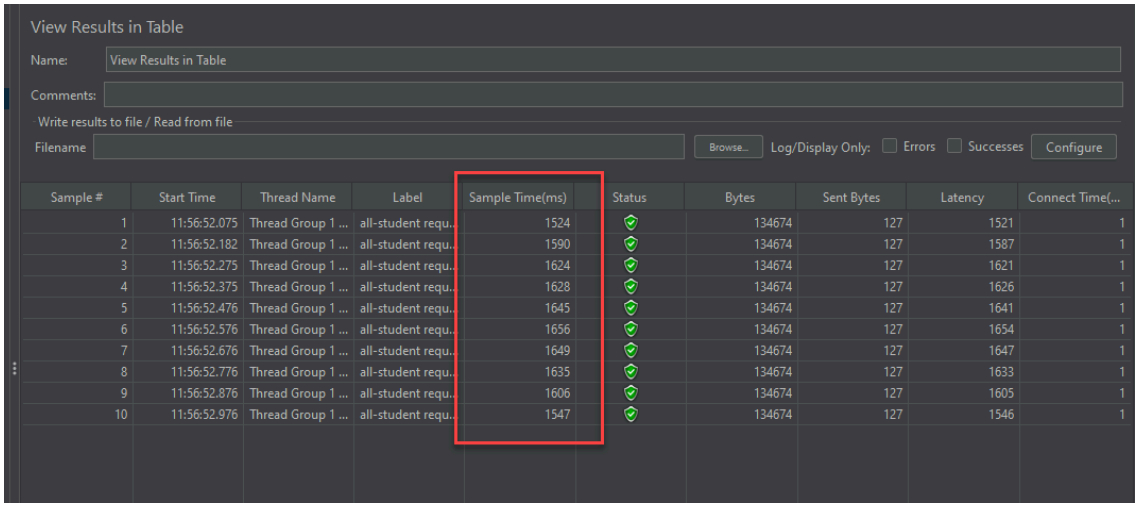




14. Run the Test Plan via GUI.



15. View the test results on each listener.
16. For example, let's look at the results in View Results in Table. Sample Time is the time it takes for your sampler to complete a request. Later, we will perform optimization to achieve shorter sample times in test results.



Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(...
1	11:56:52.075	Thread Group 1 ...	all-student requ...	1524		134674	127	1521	1
2	11:56:52.182	Thread Group 1 ...	all-student requ...	1590		134674	127	1587	1
3	11:56:52.275	Thread Group 1 ...	all-student requ...	1624		134674	127	1621	1
4	11:56:52.375	Thread Group 1 ...	all-student requ...	1628		134674	127	1626	1
5	11:56:52.476	Thread Group 1 ...	all-student requ...	1645		134674	127	1641	1
6	11:56:52.576	Thread Group 1 ...	all-student requ...	1656		134674	127	1654	1
7	11:56:52.676	Thread Group 1 ...	all-student requ...	1649		134674	127	1647	1
8	11:56:52.776	Thread Group 1 ...	all-student requ...	1635		134674	127	1633	1
9	11:56:52.876	Thread Group 1 ...	all-student requ...	1606		134674	127	1605	1
10	11:56:52.976	Thread Group 1 ...	all-student requ...	1547		134674	127	1546	1

17. Save your test plan, name it “test_plan_1.jmx”

€ Create 2 other test plan for endpoints ***/all-student-name*** and ***/highest-gpa***. Perform performance testing as has been done above. Take a screenshot of the results and place it in the README.md file

18. The best practice for using JMeter to simulate a very large number of users is not through the GUI but via the command line. Therefore, we will attempt to run the test plan we have created via the command line.
19. Run the command **`jmeter -n -t [test_plan_file.jmx] -l [test_result_log.jtl]`**

```
PS C:\Program Files\apache-jmeter-5.6.2\bin> .\jmeter -n -t test_plan_1.jmx -l testresults1.jtl
WARN StatusConsoleListener The use of package scanning to locate plugins is deprecated and will be removed in a future r
elease
WARN StatusConsoleListener The use of package scanning to locate plugins is deprecated and will be removed in a future r
elease
WARN StatusConsoleListener The use of package scanning to locate plugins is deprecated and will be removed in a future r
elease
WARN StatusConsoleListener The use of package scanning to locate plugins is deprecated and will be removed in a future r
elease
Creating summariser <summary>
Created the tree successfully using test_plan_1.jmx
Starting standalone test @ 2024 Feb 16 20:22:48 WIB (1708089768091)
Waiting for possible Shutdown/StopTestNow/HeapDump/ThreadDump message on port 4445
summary =     10 in 00:00:02 =    4.2/s Avg:  1530 Min:  1433 Max:  1581 Err:     0 (0.00%)
Tidying up ...    @ 2024 Feb 16 20:22:50 WIB (1708089770542)
... end of run
```

20. Once the test plan script has been successfully executed, you can view the results in the log file

testresults1.jtl

File Edit View

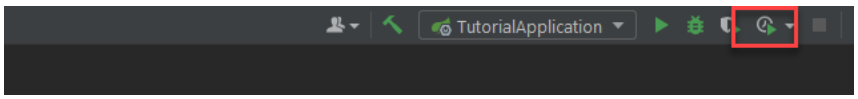
```
timeStamp,elapsed,label,responseCode,responseMessage,threadName,dataType,success,failureMessage,bytes,sentBytes,grpThreads,allThreads,URL,Latency,IdleTime,Connect
1708089768344,1552,all-student request,200,,Thread Group 1 1-1,text,true,,134674,127,10,10,http://localhost:8080/all-student,1547,0,16
1708089768345,1559,all-student request,200,,Thread Group 1 1-2,text,true,,134674,127,9,9,http://localhost:8080/all-student,1557,0,16
1708089768407,1550,all-student request,200,,Thread Group 1 1-3,text,true,,134674,127,8,8,http://localhost:8080/all-student,1548,0,1
1708089768506,1581,all-student request,200,,Thread Group 1 1-4,text,true,,134674,127,7,7,http://localhost:8080/all-student,1579,0,1
1708089768606,1568,all-student request,200,,Thread Group 1 1-5,text,true,,134674,127,6,6,http://localhost:8080/all-student,1566,0,2
1708089768707,1550,all-student request,200,,Thread Group 1 1-6,text,true,,134674,127,5,5,http://localhost:8080/all-student,1549,0,1
1708089768805,1533,all-student request,200,,Thread Group 1 1-7,text,true,,134674,127,4,4,http://localhost:8080/all-student,1531,0,3
1708089768907,1496,all-student request,200,,Thread Group 1 1-8,text,true,,134674,127,3,3,http://localhost:8080/all-student,1494,0,3
1708089769005,1483,all-student request,200,,Thread Group 1 1-9,text,true,,134674,127,2,2,http://localhost:8080/all-student,1481,0,1
1708089769106,1433,all-student request,200,,Thread Group 1 1-10,text,true,,134674,127,1,1,http://localhost:8080/all-student,1432,0,2
```

€ Run both test plans that you have previously created (for endpoint ***/highest-gpa*** and ***/all-student-name***) via the command line, take a screenshot of the results, and include them in the README.md file.

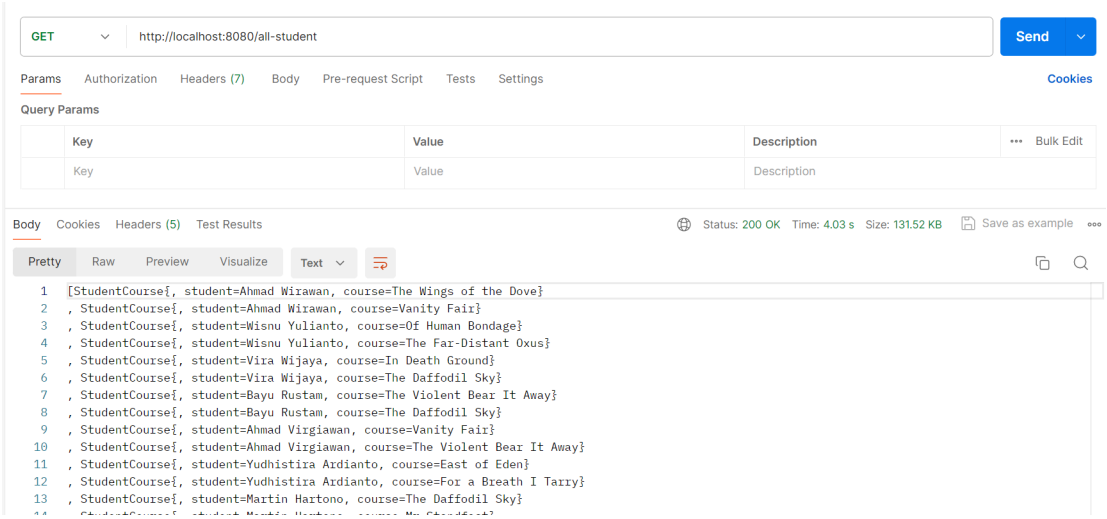
Profiling

The profiling process is carried out so that we can look more closely into the source code of the application whose performance we are measuring. The goals include obtaining more detailed information and also finding code sections/methods that consume a lot of resources so that we can optimize those parts of the code.

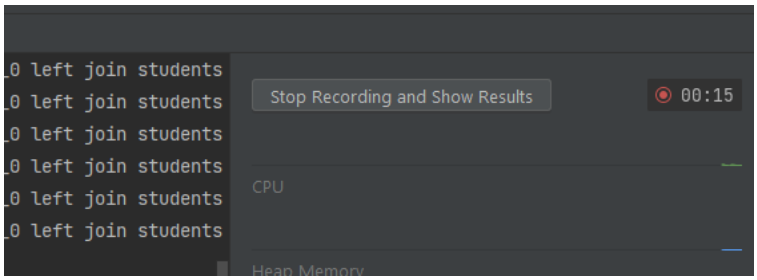
21. Running the Application using IntelliJ Profiler



22. Access the ***/all-student*** endpoint. You can use a REST client application such as postman or only with web browser.

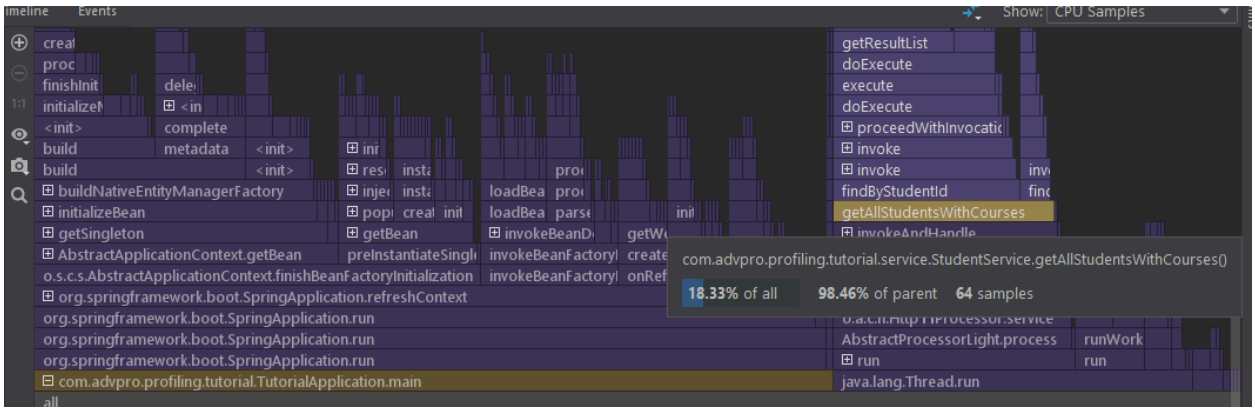


23. Click stop recording and show results in IntelliJ.

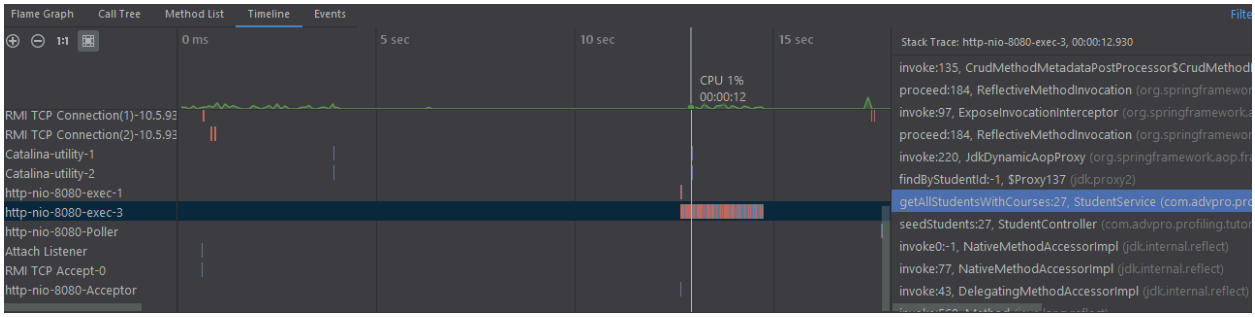


24. Find the method that consumes the most resources. It can be seen on the flame graph that the method consuming the most resources is ***getAllStudentWithCourses***. The main

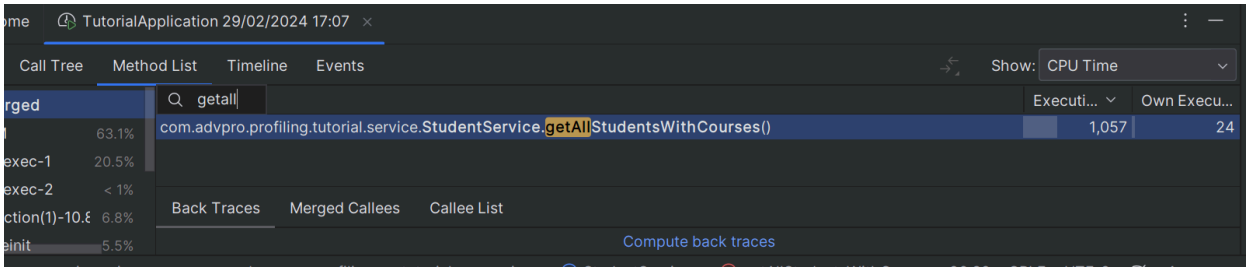
method consumes resources for starting the application, so you can ignore it.



25. You can see the processing time on the timeline tab, look for the active thread at the time the method was executed.



26. To see more detailed processing times for each method, you can look at the method list tab and then view the execution time column.



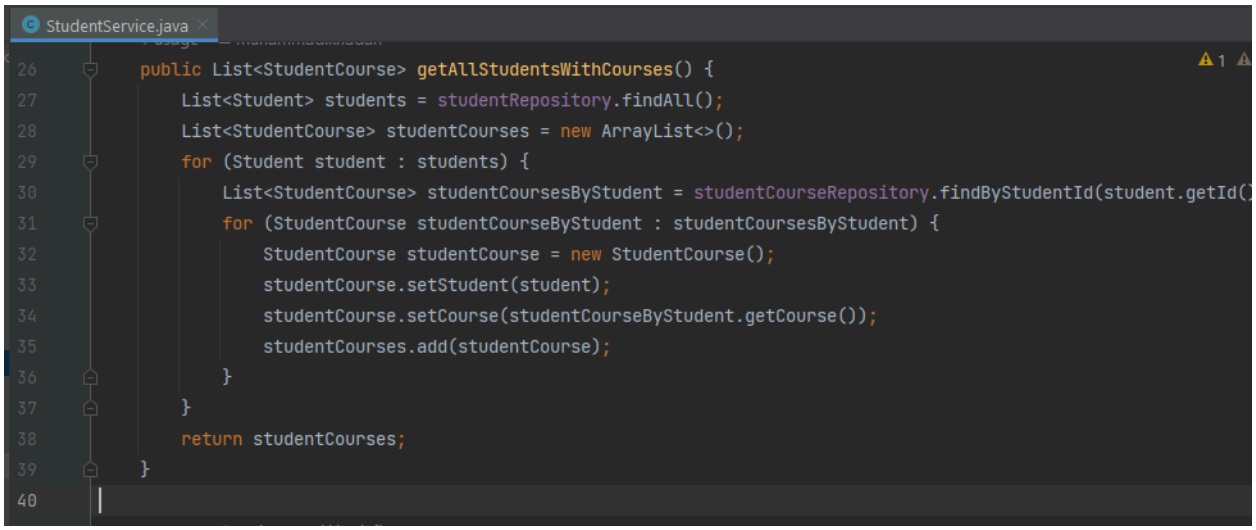
It contains information based on Total time and CPU time. Here are the differences:

- Total Time: This refers to the cumulative time spent executing a particular method or set of methods, including the time spent in the method itself and in calling other methods (i.e., the time spent in both the method and its child methods). Total time is a measure of the overall time impact of a method on application performance,

providing insight into which methods are the most time-consuming parts of your application.

- CPU Time: CPU time, on the other hand, measures only the time during which the CPU is actively executing instructions for a particular method, excluding any time spent waiting for resources, I/O operations, or time when the process was inactive or waiting. CPU time gives a more focused view of how much processor time is consumed by specific methods, helping developers identify areas where code efficiency can be improved to reduce CPU usage.

27. Open the source code of that method and analyze it.



€ Refactor that code to achieve better performance. Commit in a new branch named "**optimize**". Create representative commit messages. Do not use commit messages like "OK", "Cool", "Great", but instead create commit messages like "[Refactoring] - Optimizing method getAllStudentWithCourse".

28. You must to optimize the code to achieve at least a 20% performance improvement. If previously the process time from the /all-student endpoint was 2 seconds, at least you should be able to improve it so the process time reaches 1.6 seconds. The 2 seconds is not fix because performance can vary depending on your HW/SW, the reference point is a 20% improvement regardless of the initial performance value.

29. Do not use measurements from the first application run because when run for the first time, the JIT compiler on the JVM is not yet optimal and the processing time is runs a bit slower. Maybe you should to start the application, hit the endpoint, then turn off the application, and then start again for several times first.

30. The processing time we use is the CPU time reported in the method list tab.

31. Use the comparison view for both profiling sessions, before optimizing and after optimizing, to make it easier to see.

€ Perform the profiling process and optimizing the code for others endpoint (***/all-student-name*** and ***/highest-gpa)***. Also achieve a 20% performance improvement for both of those endpoints. Commit in "**optimize"** branch also create representative commit messages. Do not use commit messages like "OK", "Cool", "Great", but instead create commit messages like "[Refactoring] - Optimizing method getAllStudentWithCourse".

€ After the profiling and performance optimization process is completed, perform a performance test again using JMeter, see the results, and compare with the first measurement. Is there an improvement from JMeter measurements? Write your conclusion in the README.md file.

Reflection

Please answer the following questions:

1. What is the difference between the approach of performance testing with JMeter and profiling with IntelliJ Profiler in the context of optimizing application performance?
2. How does the profiling process help you in identifying and understanding the weak points in your application?
3. Do you think IntelliJ Profiler is effective in assisting you to analyze and identify bottlenecks in your application code?
4. What are the main challenges you face when conducting performance testing and profiling, and how do you overcome these challenges?
5. What are the main benefits you gain from using IntelliJ Profiler for profiling your application code?
6. How do you handle situations where the results from profiling with IntelliJ Profiler are not entirely consistent with findings from performance testing using JMeter?
7. What strategies do you implement in optimizing application code after analyzing results from performance testing and profiling? How do you ensure the changes you make do not affect the application's functionality?

Please write the answer in the **README.md file.**

Grading Scheme

Scale

All components will be scored on scale 0-4. Grade 4 is the highest grade, equivalent to A. Score 0 is for not submitting.

Components

- 30% - Performance Testing
- 30% - Commit
- 30% - Profiling and Performance Optimization
- 10% - Reflection

Rubrics

	Score 4	Score 3	Score 2	Score 1
Performance Testing				
There are screenshots of 2 JMeter test plans and their execution through the GUI for the endpoints /all-student-name and /highest-gpa	Present, clear, and easy to understand	-	Present but unclear or not appropriate	Not Present
There are screenshots of the execution results of the JMeter test plan through the command line	Present, clear, and easy to understand	-	Present but unclear or not appropriate	Not Present

for the endpoints /all-student-name and /highest-gpa				
Commit				
	All requested commits are registered with representative message and correct.	At least 50% of the requested commits are registered with representative message and correct.	At least 25% of the requested commits are registered with representative message and correct.	Less than 25%.
Profiling & Performance Optimization				
Optimizing endpoint /all-student	Performance optimized by at least 20%	Performance optimized by at least 15%	Performance optimized by at least 10%	Performance optimized by less than 10%
Optimizing endpoint /highest-gpa	Performance optimized by at least 20%	Performance optimized by at least 15%	Performance optimized by at least 10%	Performance optimized by less than 10%
Optimizing endpoint /all-student-name	Performance optimized by at least 20%	Performance optimized by at least 15%	Performance optimized by at least 10%	Performance optimized by less than 10%
There are screenshots of the re-measurement through JMeter after performance optimization along with	There are screenshots of the retesting results, with good, clear, and easy-to-understand conclusions	There are screenshots of the retesting results, but there are no conclusions/lack of explanation	There are screenshots but they are unclear/not appropriate, without explanation	Not present

conclusions/explanations				
Reflection				
Reflection (for each question)	The description is sound.	The description is sound.	The description is not sound although still related.	The description is not sound. It is not related to the topics.