**Java 9-10-11 Improvements**

**A) Smaller Java (New Modular System in Java)**

**What is Project Jigsaw?**

Project Jigsaw is an umbrella project with new features aimed at the introduction of a module system to the Java language, and its implementation in JDK source code, modularize JDK libraries and the Java runtime.

* Project Jigsaw is a Compile-Time Module System.
* Encapsulate internal APIs which are not meant to be publicly accessible.
* It helps to improve security, performance and maintainability of Java Platform.

**Why do we need modules?**

Modularity is a design principle that helps us to achieve -

* Loose Coupling
* Reliable configuration - Clear contracts and dependencies between components
* Strong Encapsulation - Hidden implementation details
* Fix issues with classpath
* Fix issues with monolithic nature of the JDK

**What are issues with old classpath and how modularity fixes these issues?**

**Classpath System --**

* If there are multiple versions of dependent jar sitting on a classpath, then classloader loads the first jar it finds which may not be relevant jar. This leads to unexpected and unpredictable results.
* The other issue with JVM using classpath was that the compilation of the application would be successful, but the application will fail at runtime with a ClassNotFound Exception, due to a missing version of that class at runtime.
* Unwieldy and lengthy classpath makes it difficult to detect conflicts or missing dependencies.

**Modular System --**

* The re-envisioning of the Java platform as a modular system made these classpath issues a thing of the past.
* In a compile-time module system the resolution of dependencies happens when the application is compiled. This way runtime errors can be avoided.
* Modules solve the pre-Java 9 classpath problem by providing reliable configuration.

**What are issues with old monolithic nature of the JDK and how modularity fix these issues?**

**Monolithic JDK --**

* Over the years, the utilities of the Java platform have continued to evolve and increase, making it into one big monolith.
* Large JDK did not fit on very small devices.
* Entire oversized JDK was loaded, even when only a small subset of the JDK was required. This resulted in memory and performance issues on devices, networks and cloud.
* The existence of internal APIs made it difficult to make the JDK secure and scalable.

**Modular System --**

* In that regard, the need for a modular system came in as a vital requirement, not only to address the modularization of the Java utilities, but also to provide a mechanism for developers to create

and manage modular applications using the same module system as that used in the JDK.

* Modules solve the pre-Java 9 JDK monolithic issue by providing strong encapsulation.
* One of the great updates to the platform is that only the modules needed are compiled, as opposed to the entire JDK. This improves scalability of an application.
* Modular system with strong encapsulation improves security and performance.

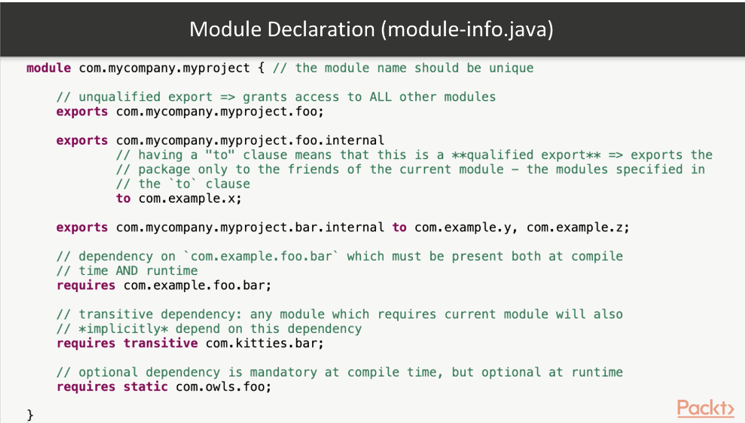
**How module system is implemented in Java?**

**Modular architecture -**

* Supports module as a top-level construct, one level above package.
* A module can have multiple packages defined
* A module configuration file (module-info.java) is used to define (at a minimum) a module name, dependencies on packages from other modules and exposed packages to outside modules.

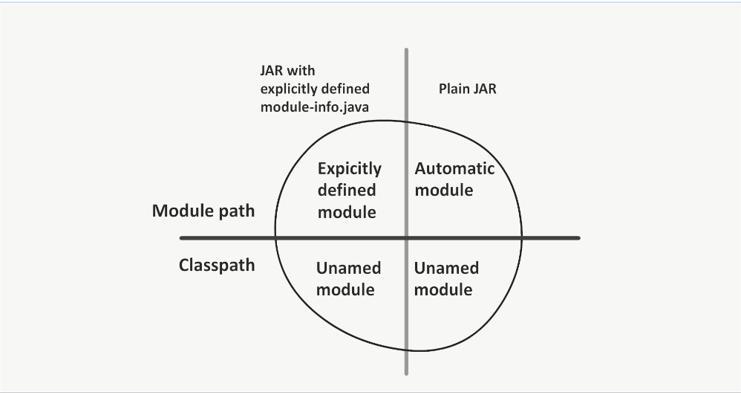
**Keywords --**

* Module - Module name
* Requires - Dependencies on packages from other modules
* Exports - Packages within this module available publicly
* Opens - Packages accessible only at runtime



**What is a module-path?**

* JVM uses module-path to verify that all necessary modules are present both at compile-time and runtime.
* Module path does not replace class-path. Module path and class-path can work together.
* Modules (Jars) defined on the module-path become the named-modules. Whereas, everything(jars) on the classpath becomes unnamed module.
* Unnamed module helps us in incremental modularization. Migrate first to Java 9, test it and then add module declarations.
* An automatic module (plain jar without module declaration on module path) can be used as a dependency for other modules.



**What are different strategies for Pre-Java-9 project migration to modular Java?**

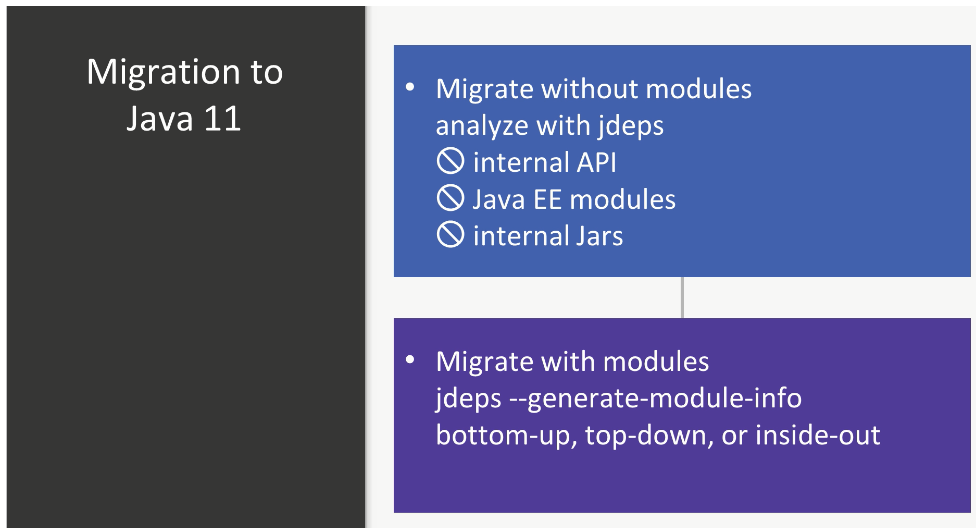
Migration to Java 9 and later can be tedious but there are good tools like Jdpes and online documentation.

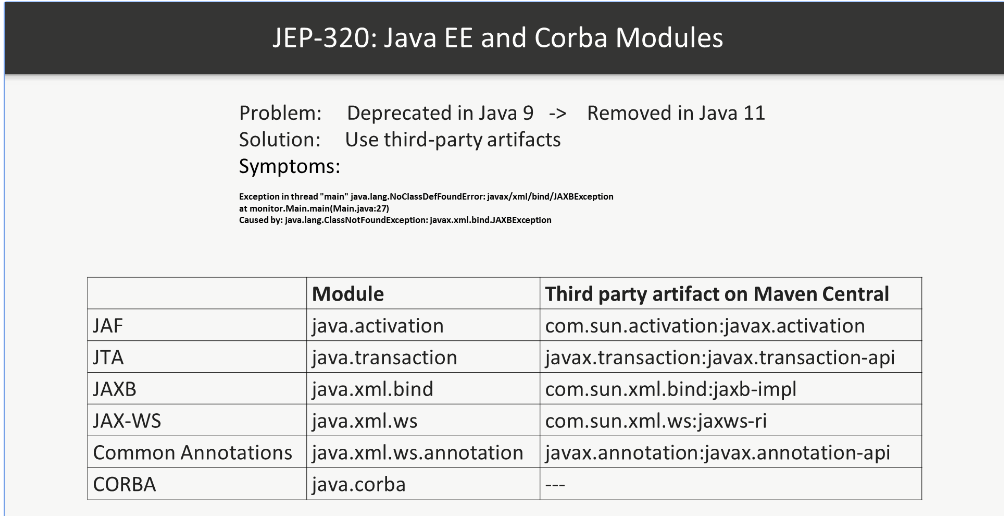
**a) Migration without modules**

* Initially migrate existing project without adding any module declarations.
* Some artifacts are no longer available due to modularization of JDK (like internal API/jars and Java EE components). FIX these dependencies!
* Jdeps tool is useful for analyzing these missing Java dependencies.
* Once all dependencies are fixed and code is working fine on Java9 then add module declaration for each component.

**b) Migration with modules**

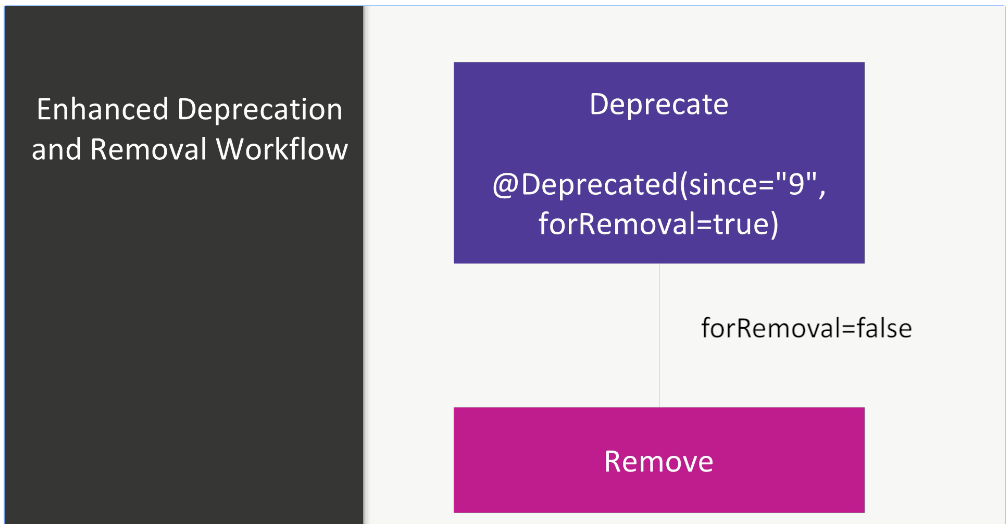
* For a large monolith application with many interconnected components, it is beneficial to add module declaration explicitly.
* Jdeps tool can help to generate initial version of module-info.java file.
* Based on a type of your project (framework or a library) or if project dependencies are already modularized or not, you can choose bottom-up, top-down or inside-out strategy for modularization.





**Which components were removed to make Java Cleaner and Smaller?**

* Java eliminated some dead weight such as Corba or Applets, but also removed modules that are still used by many developers, like Java EE modules.
* Java EE - JAF, JTA, JAXB, JAX-WS, Common Annotations. Solution is to use third party alternatives
* Java FX - OpenJFX an open source counterpart of Java FX
* Some formerly deprecated classes and methods are completely removed. Instead of sun. misc.Base64 use java.util.Base64
* Some classes and methods are moved to internal API to hide them from developers (strong encapsulation). like sun.\*, com.sun.\*
* Prior to Java 9, the JDK included the JRE as well as other tools and libraries. A notable change in Java 9 is that the JRE subdirectory is no longer part of the JDK image.
* New deprecation/removal workflow with @Deprecated annotation.



**How to create lightweight custom runtime images with JLink?**

The Jlink tool links a set of modules along with their transitive dependencies to create a custom runtime image.

Developers can create lightweight runtime environment using Jlink tool. This will improve application start-up time.

Such lightweight custom runtime images are useful in IoT devices.

There is no auto-update for Jlink images. Developers are responsible for updating their custom runtime images.

jlink [options] --module-path modulepath --add-modules module [,module...]

**B) Better Java**

**Which problem is addressed by new Local Variable Type Inference (var) feature?**

* Local Variable Type Inference feature is introduced in Java 10 to address the problem of verbose and redundant Java Syntax.
* Developers no longer have to include manifest declarations of local variable types; rather, declarations can be inferred through use of the new var identifier.
* var can be used for local variables, in for-loop, try-with-resources block. var is especially useful for verbose type declarations of generics.

For e.g. -

***var myList = new ArrayList<String>();***

***The preceding code infers ArrayList<String>, so we no longer need to use below verbose syntax.***

***It infers a type (LHS) based on value (RHS)***

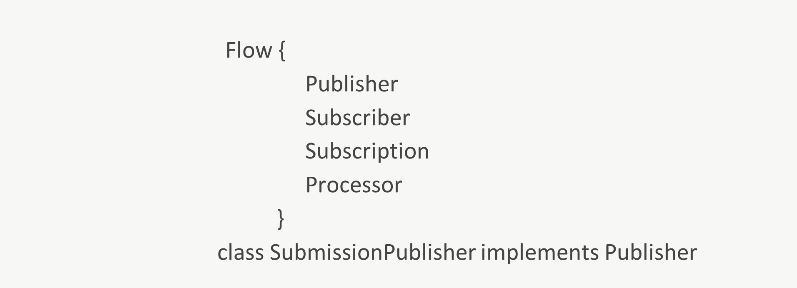
***ArrayList<String> myList = new ArrayList<String>();***

**Some key points to note:**

* Introduced in Java 10.
* Only type inference, not related to final to non-final.
* Local variable only. Cannot be used in method signature or class fields.
* Can be used with Lambda expressions starting Java 11. This way var can be combined with annotations which are helpful for input validations like @Nullable, @ZipCode etc.
* Makes code readable and maintainable.
* Var is not a new keyword introduce in java. The var identifier is technically a reserved type name.

**What are reactive streams/reactive programming/Java Flow API or Java Flow interfaces?**

* Purpose of Java Flow API or Java Flow interfaces is to Standardize asynchronous stream processing with non-blocking back-pressure.
* When different popular frameworks (Spring, Akka, RxJava2, Reactor) supporting reactive streams emerged, they were not sharing the same interfaces making migrating tedious and the learning curve steeper.
* The new interfaces from the Flow API introduced in Java 9 provide a golden standard for reactive streams to be supported by these frameworks.
* Flow.Java (Flow API) is the main entry point of every reactive application. This class has 4 interfaces defined inside it as shown below.



**Why Jshell was introduced in Java?**

* JShell is Java's REPL (Read Evaluate and Print Loop) command-line tool.
* Jshell is running Java from command line. JShell is an important tool that is relatively new to the Java platform. It was introduced with JDK 9.
* With frequent releases and new features in Java delivered faster than ever, it’s becoming more important to test the new features quickly and easily. JShell allows us to do just that.
* Prototyping and debugging becomes much easy with Jshell
* No new class with main method or other IDE related overhead to try out new features
* We can create new variables, methods, and classes in JShell. REPL is used to evaluate the declarations, statements, and expressions in Java.
* We can also write JShell scripts, which are a sequence of JShell commands and snippets.

**What are other important improvements in Java?**

1) Improved Core API: String and Files Improvements

2) Going More Functional: Improvements to Collections, Optional, and Predicate

**3) Taking control of external processes--**

Approaches to create a Java process and handle process input/output before Java9 were inflexible and non-portable, as the set of commands executed by the external processes were highly dependent on the operating system.

An additional effort has been exerted in Java 9 in order to make the process operations portable across multiple operating systems.

**4) Getting ready for HTTP 2.0--**

HTTP 2.0 is the successor to the HTTP 1.1 protocol, and this new version of the protocol addresses some limitations and drawbacks of the previous one. HTTP 2.0 improves performance in several ways and provides capabilities such as request/response multiplexing in a single TCP connection, sending of responses in a server push, flow control, and request prioritization, among others.

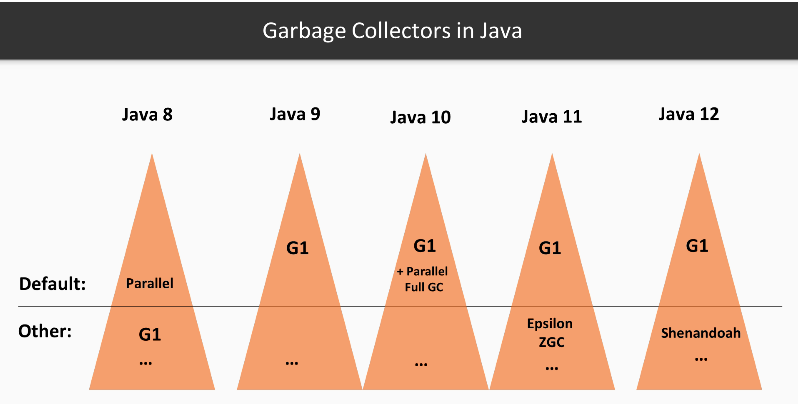
Java provides the *java.net.HttpURLConnection* utility that can be used to establish a non-secure HTTP 1.1 connection. However, the API was considered difficult to maintain, an issue which was further complicated by the need to support HTTP 2.0 and, so, an entirely new client API was introduced in Java 9 in order to establish a connection via the HTTP 2.0 or the web socket protocols.

**C) Faster Java**

**How GC evolved in post Java 8 versions?**

**New Improved Garbage Collectors --**

* Garbage collectors (GCs) help the developers to allocate and de-allocate the memory. The pauses created when GC collects unused memory can be annoying and influence the application performance.
* Over the period, GC evolved in Java for better application performance and memory management.
* In Java 8, default garbage collector was Parallel GC. G1 was available for use with VM arguments.
* Since Java 9, G1 (Garbage-First) is the default garbage collector. As the name indicates, it targets regions with most garbage first for memory de-allocation.
* In Java 10, G1 is upgraded. Full GC becomes parallel improving its worst-case latencies. All heap operations are performed concurrently with application threads. G1 is targeted for multi-processor machines with a large amount of memory.
* Java 11 (latest long-term support release) introduces Epsilon GC – A no-op GC. There is no memory reclamation mechanism in Epsilon. Once available heap is exhausted, JVM will shut down. This GC can be useful for performance testing or memory pressure testing. Epsilon is not meant for production use unless you absolutely know what you are doing.
* In addition to Epsilon, an initial version of ZGC is available in Java 11.



**What is difference between CDS and ApsCDS?**

**Class Data Sharing (CDS):**

* The Class data sharing (CDS) feature helps reduce the startup time and memory footprint between multiple Java Virtual Machines (JVM).
* When you use the installer to install the Oracle Java Runtime Environment (JRE), the installer loads a default set of classes from the system Java Archive (JAR) file into a private internal representation and dumps that representation to a file called a shared archive.
* When the JVM starts, the shared archive is memory-mapped to allow sharing of read-only JVM metadata for these classes among multiple JVM processes.
* Because accessing the shared archive is faster than loading the classes, startup time is reduced.
* Class-Data sharing (CDS) was available since Java 5 but was used only by bootstrap class loader.

**Application Class-Data Sharing (ApsCDS):**

* To further reduce the startup time and the footprint, Application Class-Data Sharing (ApsCDS) is introduced that extends the CDS to include selected classes from the application class path.
* This feature allows application classes to be placed in a shared drive. The common class metadata is shared across different Java processes.
* AppCDS allows the built-in system class loader, built-in platform class loader, and custom class loaders to load the archived classes.
* When multiple JVMs share the same archive file, memory is saved, and the overall system response time improves.
* Application Class-Data Sharing (ApsCDS) is introduced in Java 10. It can be used for application classes by application class loader.

**What new features and tools are introduced to improve performance in post Java8 versions?**

* New improved garbage collectors
* Use of efficient data-types with immutable collections for performance improvement and maintainability.
* String improvements – Starting Java 9, String has changed its internal representation (uses byte [] instead of char[]). String is now more compact and thus reduces memory footprint and improves performance. In addition, string concatenation is improved.

~~Char []~~ byte[] value 🡪 Reduced memory footprint 🡪 fewer GC calls (fewer GC pauses)

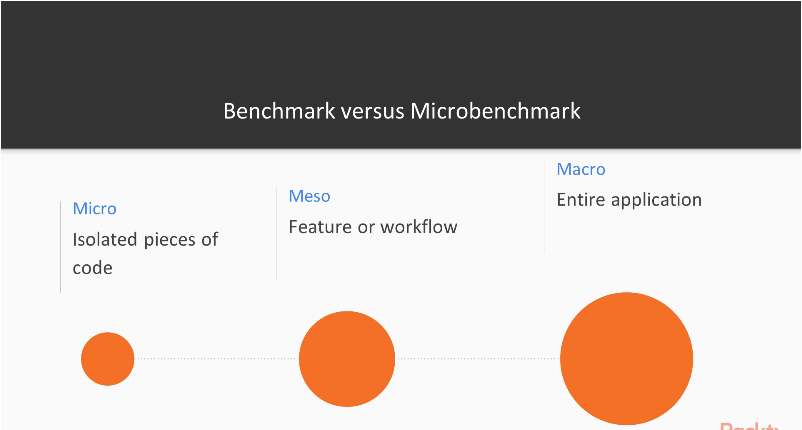
~~StringBuilder::append~~ invokedynamic

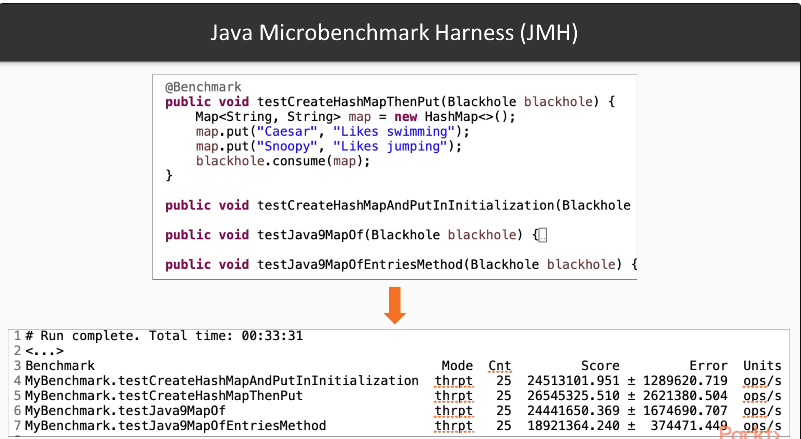
* Modules and Application Class Data Sharing (ApsCDS) help reduce the startup time and memory footprint.
* Graal compiler for faster applications, high-performance and extensibility.

**Monitoring and benchmarking –**

**Benchmarking:** New Java Microbenchmark Harness tool (JMH - introduced in Java 9) will help to analyze performance. Optimize critical parts of the code by measuring and comparing performance of different implementations of same critical code.

Highly configurable with @Benchmark, @Fork and @BenchmarkMode annotations.





**Monitoring**: New low-overhead heap profiling (since Java 11) to sample Java Heap allocations.