OOPS with Java AS1

Tuesday, September 3, 2024 11:47 AM

A Short History of Java :

• Birth: 1991.
• Origin: Sun Microsystems.
• Green Project: To explore opportunities in the consumer electronics market.
• Green Team(Key Members): James Gosling, Patrick Naughton, Mike Sheridan.
• From Oak to Java: The original language, named Oak, was later renamed to Java.
• Programming Paradigm: Object Oriented.
• The "*7" Device(1992): To showcase the technology potentials.
• Failure of "*7": Time-Warner denied set-top box OS and video-on-demand technology for demo.
• Breakthrough with the Web(1994): WebRunner (a Web browser), Applet.
• First public implementation: Java 1.0 in 1996.
• Acquisition of Java: Oracle Corporation acquired Sun Microsystems in 2010.
• Slogan: "Write Once, Run Anywhere".

The history and evolution of Java is kind of how technological innovation can sometimes precede the emergence of a clear market need. It is a fascinating study of technology development and adaptation.

Origins and Early Development

- **Java's Genesis**: Java's roots trace back to **1991** when a small group of Sun Microsystems engineers, led by **James Gosling**, embarked on a project to develop technology for the **consumer electronics market**. This project, known as the "**Green" project**, aimed to create a distributed system that could operate across various platforms.
- **Initial Focus**: The original goal was to **develop a distributed system for consumer electronics**, which required a technology that was reliable, cost-effective, and standards-compliant. This focus on consumer needs contrasted sharply with the demands of workstation users who prioritized power and were willing to tolerate higher costs and complexity.
- **Transition to Oak and Java**: **Initially extending C++**, Gosling and his team found it **insufficient** and **began developing Oak, which later became Java**. The language was created as a tool for **building a large, distributed network of devices** rather than as a direct competitor to C++.

Evolution and Adaptation

- **_*The 7 Device_:** In 1992, the Green project produced a prototype device, the *7, which was a handheld remote control-like device. This demonstrated the small, efficient codebase and impressed key figures within Sun Microsystems.
- **Shift in Strategy**: By early 1993, as the Green team (now FirstPerson Inc.) pursued opportunities in **set-top boxes**, they faced **setbacks** due to non-technical reasons. Despite having superior technology, **business politics** prevented them from securing key contracts.

Emergence and Impact on the Web

- **Web Integration**: By mid-1994, the World Wide Web was gaining traction. Sun recognized an opportunity to leverage Java in the web space, resulting in the development of the **WebRunner browser** (later **HotJava**). This browser showcased Java's capabilities in creating **interactive web experiences.**
- **Java's Role**: Java's **platform-independent nature** made it particularly suited for web applications. It allowed for a more dynamic interaction with web content compared to static HTML pages, transforming web browsers into frameworks for **more complex and interactive applications.**

Market Position and Competition

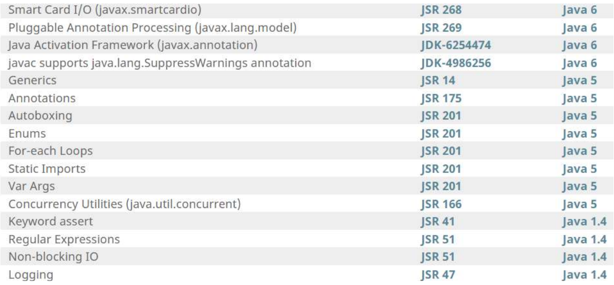
- **Strategic Licensing**: Sun Microsystems opted to license Java technology widely to various companies, including browser developers and OEMs, rather than keeping it proprietary. This strategy aimed to **establish Java as a universal standard for web-based applications**.
- **Competitive Landscape**: Despite Java's strengths, it faced competition from other technologies, including Microsoft's Visual Basic and General Magic's Telescript. However, **Java's cross-platform capabilities and emphasis on security distinguished it in the market.**
- **Challenges and Future Directions**: Java faced challenges such as the need for more efficient interpreters and the expansion of its application beyond the web. Sun recognized the importance of developing authoring tools and making Java lighter for broader applications, including embedded systems and interactive devices.

Strategic Implications

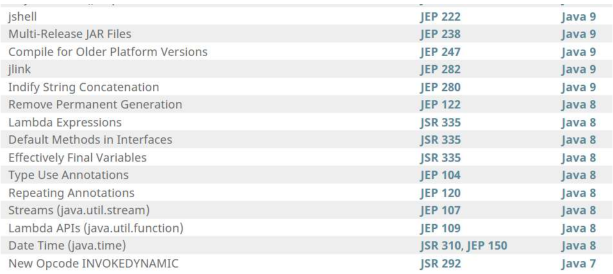
- **Marketing and Adoption**: Sun's strategy involved making Java accessible and attractive to a broad range of developers and users. The focus was on generating widespread interest and demonstrating Java's potential to transform web and networked applications.
- **Long-Term Vision**: Sun aimed to avoid the pitfalls experienced with its previous technology, **NeWS**, by collaborating with industry players and promoting Java's integration across various platforms and services.

In conclusion, Java's development was a strategic response to the need for a platform-independent, reliable technology capable of transforming how web and network applications are built and interacted with. Its evolution from a consumer electronics project to a key component of web technology reflects a successful adaptation to market demands and technological opportunities.

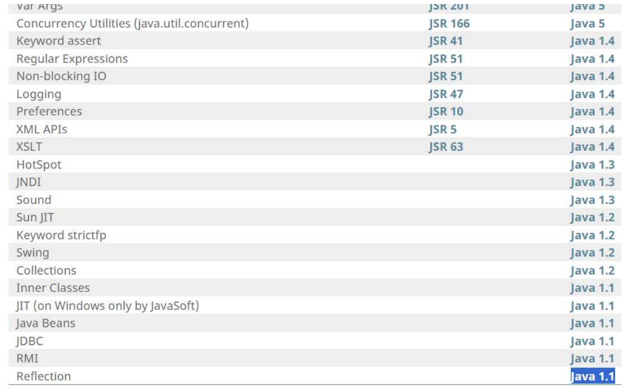
Java Language Features



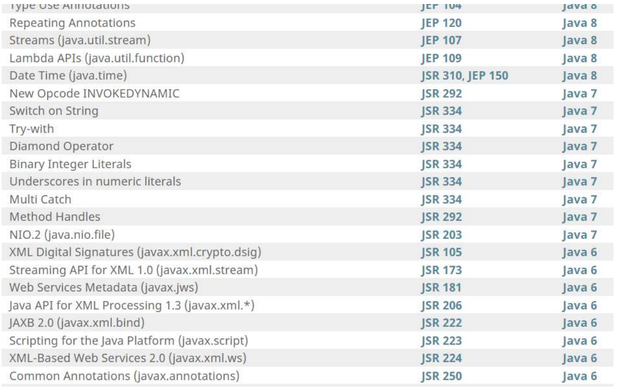
Smart Card I/O (javax.smartcardio)	JSR 268	Java 6
Pluggable Annotation Processing (javax.lang.model)	JSR 269	Java 6
Java Activation Framework (javax.annotation)	JDK-6254474	Java 6
javac supports java.lang.SuppressWarnings annotation	JDK-4986256	Java 6
Generics	JSR 14	Java 5
Annotations	JSR 175	Java 5
Autoboxing	JSR 201	Java 5
Enums	JSR 201	Java 5
For-each Loops	JSR 201	Java 5
Static Imports	JSR 201	Java 5
Var Args	JSR 201	Java 5
Concurrency Utilities (java.util.concurrent)	JSR 166	Java 5
Keyword assert	JSR 41	Java 1.4
Regular Expressions	JSR 51	Java 1.4
Non-blocking IO	JSR 51	Java 1.4
Logging	JSR 47	Java 1.4



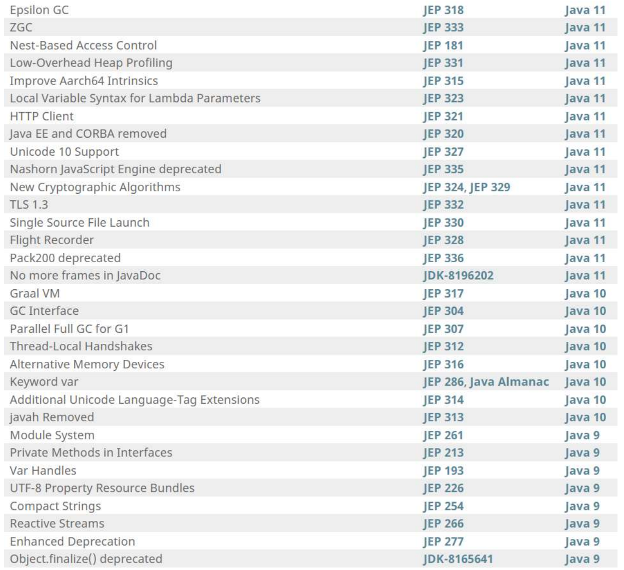
jshell	JEP 222	Java 9
Multi-Release JAR Files	JEP 238	Java 9
Compile for Older Platform Versions	JEP 247	Java 9
jlink	JEP 282	Java 9
Indify String Concatenation	JEP 280	Java 9
Remove Permanent Generation	JEP 122	Java 8
Lambda Expressions	JSR 335	Java 8
Default Methods in Interfaces	JSR 335	Java 8
Effectively Final Variables	JSR 335	Java 8
Type Use Annotations	JEP 104	Java 8
Repeating Annotations	JEP 120	Java 8
Streams (java.util.stream)	JEP 107	Java 8
Lambda APIs (java.util.function)	JEP 109	Java 8
Date Time (java.time)	JSR 310, JEP 150	Java 8
New Opcode INVOKEDYNAMIC	JSR 292	Java 7



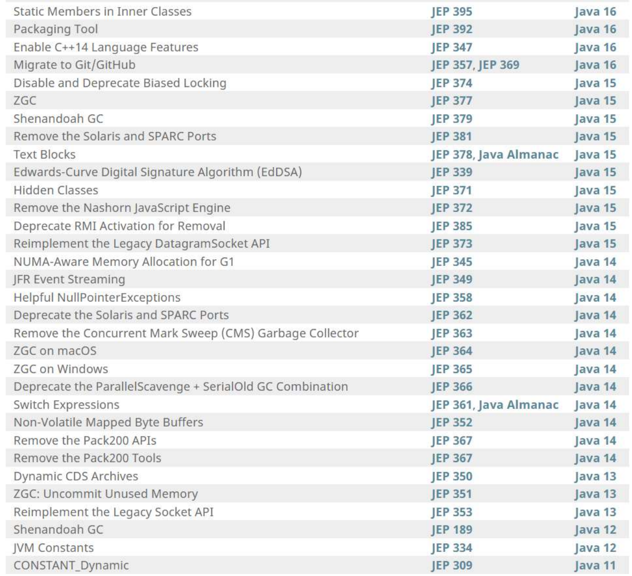
Var Args	JSR 201	Java 5
Concurrency Utilities (java.util.concurrent)	JSR 166	Java 5
Keyword assert	JSR 41	Java 1.4
Regular Expressions	JSR 51	Java 1.4
Non-blocking IO	JSR 51	Java 1.4
Logging	JSR 47	Java 1.4
Preferences	JSR 10	Java 1.4
XML APIs	JSR 5	Java 1.4
XSLT	JSR 63	Java 1.4
HotSpot		Java 1.3
JNDI		Java 1.3
Sound		Java 1.3
Sun JIT		Java 1.2
Keyword strictfp		Java 1.2
Swing		Java 1.2
Collections		Java 1.2
Inner Classes		Java 1.1
JIT (on Windows only by JavaSoft)		Java 1.1
Java Beans		Java 1.1
JDBC		Java 1.1
RMI		Java 1.1
Reflection		Java 1.1



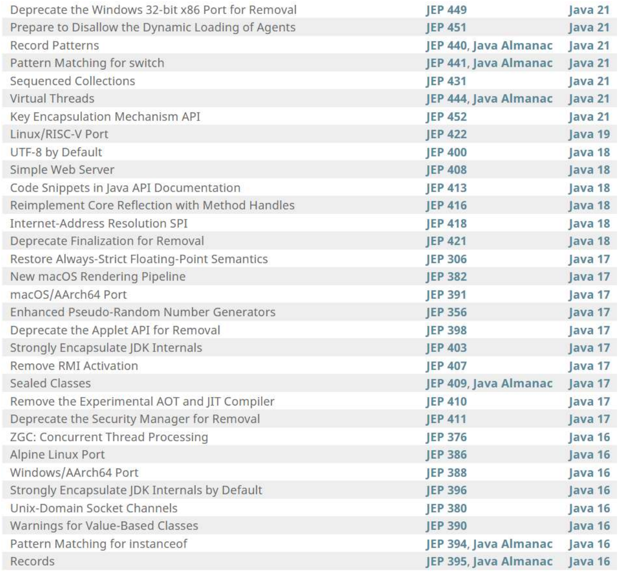
Type Use Annotations	JEP 104	Java 8
Repeating Annotations	JEP 120	Java 8
Streams (java.util.stream)	JEP 107	Java 8
Lambda APIs (java.util.function)	JEP 109	Java 8
Date Time (java.time)	JSR 310, JEP 150	Java 8
New Opcode INVOKEDYNAMIC	JSR 292	Java 7
Switch on String	JSR 334	Java 7
Try-with	JSR 334	Java 7
Diamond Operator	JSR 334	Java 7
Binary Integer Literals	JSR 334	Java 7
Underscores in numeric literals	JSR 334	Java 7
Multi Catch	JSR 334	Java 7
Method Handles	JSR 292	Java 7
NIO.2 (java.nio.file)	JSR 203	Java 7
XML Digital Signatures (javax.xml.crypto.dsig)	JSR 105	Java 6
Streaming API for XML 1.0 (javax.xml.stream)	JSR 173	Java 6
Web Services Metadata (javax.jws)	JSR 181	Java 6
Java API for XML Processing 1.3 (javax.xml.*)	JSR 206	Java 6
JAXB 2.0 (javax.xml.bind)	JSR 222	Java 6
Scripting for the Java Platform (javax.script)	JSR 223	Java 6
XML-Based Web Services 2.0 (javax.xml.ws)	JSR 224	Java 6
Common Annotations (javax.annotations)	JSR 250	Java 6



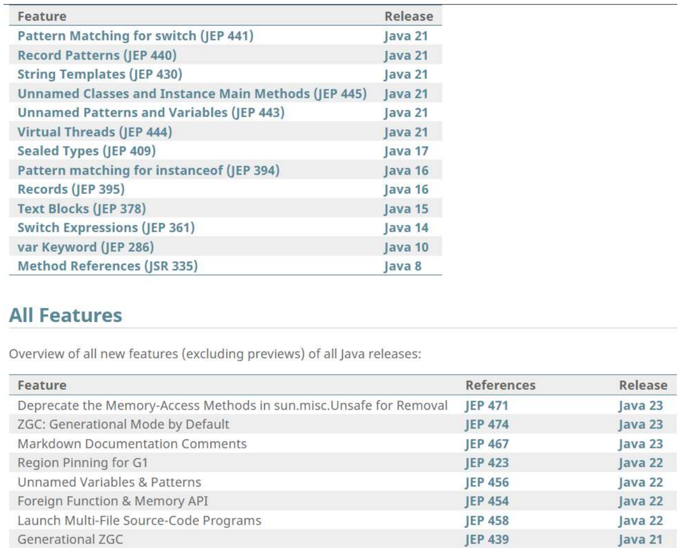
Epsilon GC	JEP 318	Java 11
ZGC	JEP 333	Java 11
Nest-Based Access Control	JEP 181	Java 11
Low-Overhead Heap Profiling	JEP 331	Java 11
Improve Aarch64 Intrinsics	JEP 315	Java 11
Local Variable Syntax for Lambda Parameters	JEP 323	Java 11
HTTP Client	JEP 321	Java 11
Java EE and CORBA removed	JEP 320	Java 11
Unicode 10 Support	JEP 327	Java 11
Nashorn JavaScript Engine deprecated	JEP 335	Java 11
New Cryptographic Algorithms	JEP 324, JEP 329	Java 11
TLS 1.3	JEP 332	Java 11
Single Source File Launch	JEP 330	Java 11
Flight Recorder	JEP 328	Java 11
Pack200 deprecated	JEP 336	Java 11
No more frames in JavaDoc	JDK-8196202	Java 11
Graal VM	JEP 317	Java 10
GC Interface	JEP 304	Java 10
Parallel Full GC for G1	JEP 307	Java 10
Thread-Local Handshakes	JEP 312	Java 10
Alternative Memory Devices	JEP 316	Java 10
Keyword var	JEP 286, Java Almanac	Java 10
Additional Unicode Language-Tag Extensions	JEP 314	Java 10
javah Removed	JEP 313	Java 10
Module System	JEP 261	Java 9
Private Methods in Interfaces	JEP 213	Java 9
Var Handles	JEP 193	Java 9
UTF-8 Property Resource Bundles	JEP 226	Java 9
Compact Strings	JEP 254	Java 9
Reactive Streams	JEP 266	Java 9
Enhanced Deprecation	JEP 277	Java 9
Object.finalize() deprecated	JDK-8165641	Java 9



Static Members in Inner Classes	JEP 395	Java 16
Packaging Tool	JEP 392	Java 16
Enable C++14 Language Features	JEP 347	Java 16
Migrate to Git/GitHub	JEP 357, JEP 369	Java 16
Disable and Deprecate Biased Locking	JEP 374	Java 15
ZGC	JEP 377	Java 15
Shenandoah GC	JEP 379	Java 15
Remove the Solaris and SPARC Ports	JEP 381	Java 15
Text Blocks	JEP 378, Java Almanac	Java 15
Edwards-Curve Digital Signature Algorithm (EdDSA)	JEP 339	Java 15
Hidden Classes	JEP 371	Java 15
Remove the Nashorn JavaScript Engine	JEP 372	Java 15
Deprecate RMI Activation for Removal	JEP 385	Java 15
Reimplement the Legacy DatagramSocket API	JEP 373	Java 15
NUMA-Aware Memory Allocation for G1	JEP 345	Java 14
JFR Event Streaming	JEP 349	Java 14
Helpful NullPointerExceptions	JEP 358	Java 14
Deprecate the Solaris and SPARC Ports	JEP 362	Java 14
Remove the Concurrent Mark Sweep (CMS) Garbage Collector	JEP 363	Java 14
ZGC on macOS	JEP 364	Java 14
ZGC on Windows	JEP 365	Java 14
Deprecate the ParallelScavenge + SerialOld GC Combination	JEP 366	Java 14
Switch Expressions	JEP 361, Java Almanac	Java 14
Non-Volatile Mapped Byte Buffers	JEP 352	Java 14
Remove the Pack200 APIs	JEP 367	Java 14
Remove the Pack200 Tools	JEP 367	Java 14
Dynamic CDS Archives	JEP 350	Java 13
ZGC: Uncommit Unused Memory	JEP 351	Java 13
Reimplement the Legacy Socket API	JEP 353	Java 13
Shenandoah GC	JEP 189	Java 12
JVM Constants	JEP 334	Java 12
CONSTANT_Dynamic	JEP 309	Java 11



Deprecate the Windows 32-bit x86 Port for Removal	JEP 449	Java 21
Prepare to Disallow the Dynamic Loading of Agents	JEP 451	Java 21
Record Patterns	JEP 440, Java Almanac	Java 21
Pattern Matching for switch	JEP 441, Java Almanac	Java 21
Sequenced Collections	JEP 431	Java 21
Virtual Threads	JEP 444, Java Almanac	Java 21
Key Encapsulation Mechanism API	JEP 452	Java 21
Linux/RISC-V Port	JEP 422	Java 19
UTF-8 by Default	JEP 400	Java 18
Simple Web Server	JEP 408	Java 18
Code Snippets in Java API Documentation	JEP 413	Java 18
Reimplement Core Reflection with Method Handles	JEP 416	Java 18
Internet-Address Resolution SPI	JEP 418	Java 18
Deprecate Finalization for Removal	JEP 421	Java 18
Restore Always-Strict Floating-Point Semantics	JEP 306	Java 17
New macOS Rendering Pipeline	JEP 382	Java 17
macOS/AArch64 Port	JEP 391	Java 17
Enhanced Pseudo-Random Number Generators	JEP 356	Java 17
Deprecate the Applet API for Removal	JEP 398	Java 17
Strongly Encapsulate JDK Internals	JEP 403	Java 17
Remove RMI Activation	JEP 407	Java 17
Sealed Classes	JEP 409, Java Almanac	Java 17
Remove the Experimental AOT and JIT Compiler	JEP 410	Java 17
Deprecate the Security Manager for Removal	JEP 411	Java 17
ZGC: Concurrent Thread Processing	JEP 376	Java 16
Alpine Linux Port	JEP 386	Java 16
Windows/AArch64 Port	JEP 388	Java 16
Strongly Encapsulate JDK Internals by Default	JEP 396	Java 16
Unix-Domain Socket Channels	JEP 380	Java 16
Warnings for Value-Based Classes	JEP 390	Java 16
Pattern Matching for instanceof	JEP 394, Java Almanac	Java 16
Records	JEP 395, Java Almanac	Java 16



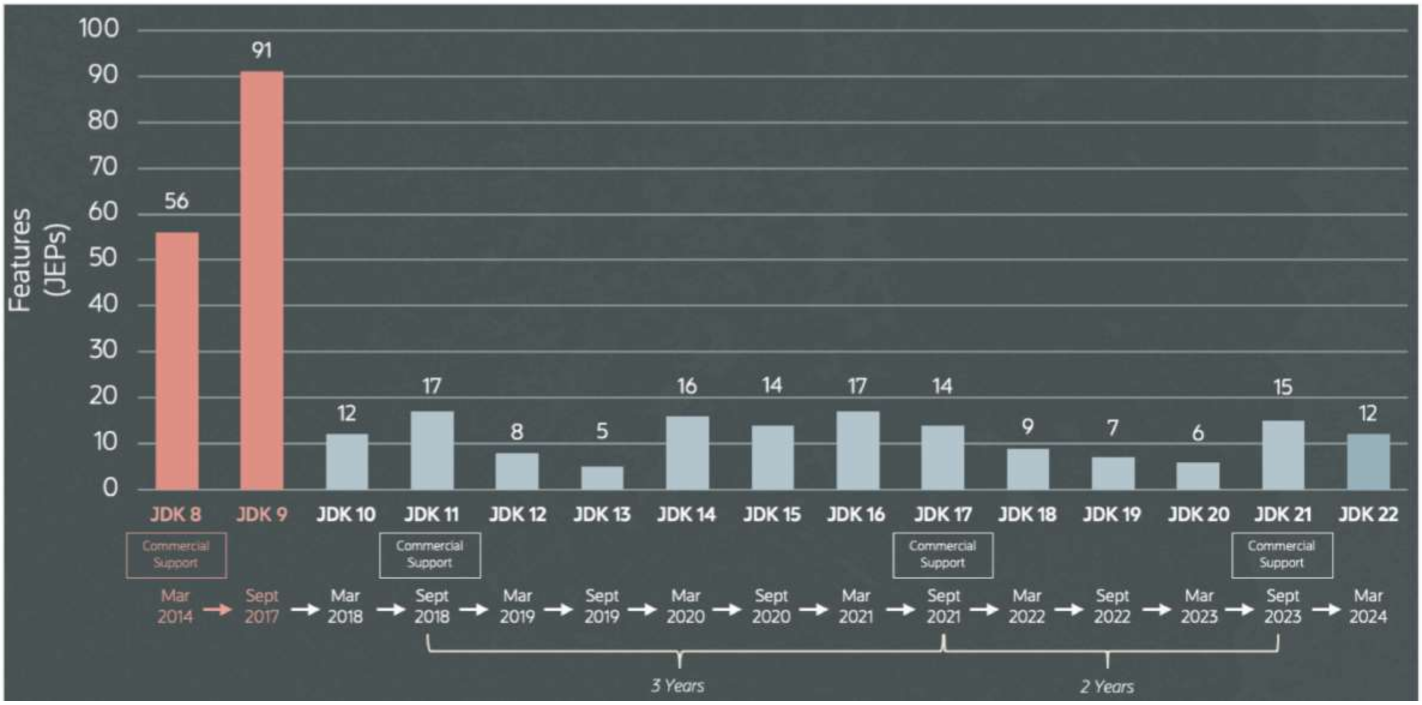
Feature	Release
Pattern Matching for switch (JEP 441)	Java 21
Record Patterns (JEP 440)	Java 21
String Templates (JEP 430)	Java 21
Unnamed Classes and Instance Main Methods (JEP 445)	Java 21
Unnamed Patterns and Variables (JEP 443)	Java 21
Virtual Threads (JEP 444)	Java 21
Sealed Types (JEP 409)	Java 17
Pattern matching for instanceof (JEP 394)	Java 16
Records (JEP 395)	Java 16
Text Blocks (JEP 378)	Java 15
Switch Expressions (JEP 361)	Java 14
var Keyword (JEP 286)	Java 10
Method References (JSR 335)	Java 8

All Features

Overview of all new features (excluding previews) of all Java releases:

Feature	References	Release
Deprecate the Memory-Access Methods in sun.misc.Unsafe for Removal	JEP 471	Java 23
ZGC: Generational Mode by Default	JEP 474	Java 23
Markdown Documentation Comments	JEP 467	Java 23
Region Pinning for G1	JEP 423	Java 22
Unnamed Variables & Patterns	JEP 456	Java 22
Foreign Function & Memory API	JEP 454	Java 22
Launch Multi-File Source-Code Programs	JEP 458	Java 22
Generational ZGC	JEP 439	Java 21

Java Version History



Choosing the right version of the JDK (Java Development Kit)

Choosing the right version of the JDK (Java Development Kit) and its distribution is important for ensuring compatibility, performance, and long-term support for your Java applications. Here’s a concise guide based on the provided information:

JDK Version Recommendations

- **JDK 21**:
  - **Type**: Long-Term-Support (LTS)
  - **Release Date**: September 2023
  - **Highlights**: Pattern Matching, Virtual Threads
  - **Recommendation**: Use JDK 21 for both new projects and existing applications. It's the latest LTS version and will receive updates for a longer period.
- **JDK 17**:
  - **Type**: Long-Term-Support (LTS)
  - **Release Date**: September 2021
  - **Highlights**: Sealed Classes
  - **Recommendation**: If you are using JDK 17, it is still a solid choice, but consider upgrading to JDK 21 in the near future to benefit from newer features and longer support.

Distribution Recommendations

1. **Adoptium Eclipse Temurin**:
   - **Recommendation**: Highly recommended. Provides high-quality, vendor-neutral OpenJDK builds with regular updates and support.
2. **Azul Zulu**:
   - **Recommendation**: Good choice. Offers no-cost, production-ready OpenJDK builds and wide platform support.
3. **BellSoft Liberica JDK**:
   - **Recommendation**: Good choice. Provides open-source builds with broad compatibility and good industry reputation.
4. **Amazon Corretto**:
   - **Recommendation**: Good choice, especially if you run Java applications on Amazon Linux 2 or AWS environments.
5. **Microsoft Build of OpenJDK**:
   - **Recommendation**: Use only if running Java applications directly on Azure. Other options are more established.

Distributions to Avoid

- **Oracle OpenJDK builds**: Limited support and updates after a short period. Not recommended for long-term use.
- **Oracle Java SE Development Kit (JDK)**: Licensing issues may arise, especially for commercial use.
- **AdoptOpenJDK**: Now succeeded by Adoptium Eclipse Temurin. Do not use.
- **Alibaba Dragonwell** and **SapMachine**: Limited recommendations unless specific conditions apply.
- **ojdkbuild**: Project is discontinued.

Special Considerations

- **Apple Silicon**: For development on Apple Silicon Macs, use a native macOS/AArch64 build of JDK 17 or later for optimal performance.
- **GraalVM**: Offers advanced features but may require additional considerations. Share experiences if using it in production.

Installation Tips

- **For Local Development**: Use SDKMAN! for easy management and installation of different JDK versions.
- **Check Installed Version**: Use java --version or which java followed by java --version to check your current JDK version.

Summary

For most users, JDK 21 with Adoptium Eclipse Temurin is the recommended choice due to its long-term support and reliable distribution. Consider your specific needs and environment when selecting a distribution and version.

The Java Development Kit (JDK) and Java Runtime Environment (JRE) have a structured directory layout that organizes their files and resources. Here's an overview of these directories and the key files they contain:

JDK Directory Structure

1. Root Directory (/jdk-1.8)

- **/jdk-1.8**: This is the root directory of your JDK installation. It contains:
  - copyright, license, and README files
  - src.zip: An archive of the Java platform source code

2. Bin Directory (/jdk-1.8/bin)

- **/jdk-1.8/bin**: Contains executable files for JDK tools:
  - java*: The Java application launcher
  - javac*: The Java compiler
  - javap*: The Java class file disassembler
  - javah*: The Java header file generator
  - javadoc*: The Javadoc documentation generator

3. Lib Directory (/jdk-1.8/lib)

- **/jdk-1.8/lib**: Contains essential files used by the development tools:
  - tools.jar: Contains non-core classes supporting JDK tools and utilities
  - dt.jar: Contains Bean Info files for IDEs

4. JRE Directory (/jdk-1.8/jre)

- **/jdk-1.8/jre**: This is the root directory of the JRE used by JDK tools. It represents the runtime environment:
  - **/jdk-1.8/jre/bin**: Contains executable files for the runtime environment. This includes:
    - java*: The runtime launcher, similar to the one in /jdk-1.8/bin
  - **/jdk-1.8/jre/lib**: Contains runtime libraries and resource files:
    - rt.jar: Contains core Java API classes
    - charsets.jar: Contains character-conversion classes
    - **/jdk-1.8/jre/lib/ext**: Default directory for Java extensions:
      - jfxrt.jar: JavaFX runtime libraries
      - localedata.jar: Locale data for java.text and java.util
    - **/jdk-1.8/jre/lib/security**: Contains security management files:
      - java.policy: Security policy file
      - java.security: Security properties file
    - **/jdk-1.8/jre/lib/applet**: Contains JAR files for applet support classes
    - **/jdk-1.8/jre/lib/fonts**: Contains font files used by the platform

Additional Files and Directories

1. Source Code (/jdk-1.8/src.zip)

- **/jdk-1.8/src.zip**: An archive containing the source code for the Java platform. Useful for developers who need to view or debug the Java standard library source code.

2. C Header Files (/jdk-1.8/include)

- **/jdk-1.8/include**: Contains C-language header files used for native-code programming with Java, such as:
  - JNI (Java Native Interface)
  - JVM TI (Java Virtual Machine Tool Interface)
  - Java Access Bridge API

3. Man Pages (/jdk-1.8/man)

- **/jdk-1.8/man**: Contains manual pages for JDK commands and tools. Useful for command-line reference.

About the Java Technology

Java technology is both a programming language and a platform.

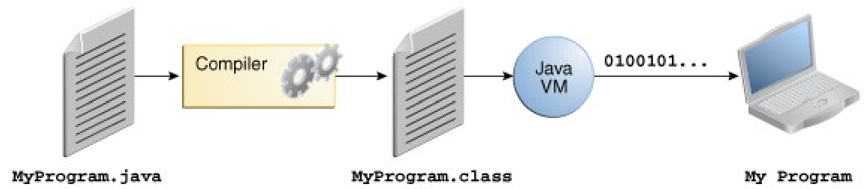
The Java Programming Language

The Java programming language is a high-level language that can be characterized by all of the following buzzwords:

- Simple
- Object oriented
- Distributed
- Multithreaded
- Dynamic
- Architecture neutral
- Portable
- High performance
- Robust
- Secure

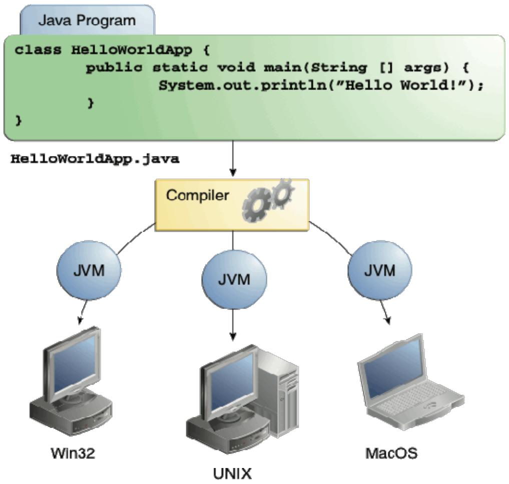
Each of the preceding buzzwords is explained in *The Java Language Environment* , a white paper written by James Gosling and Henry McGilton.

In the Java programming language, all source code is first written in plain text files ending with the .java extension. Those source files are then compiled into .class files by the javac compiler. A .class file does not contain code that is native to your processor; it instead contains *bytecodes* — the machine language of the Java Virtual Machine[1] (Java VM). The java launcher tool then runs your application with an instance of the Java Virtual Machine.



An overview of the software development process.

Because the Java VM is available on many different operating systems, the same .class files are capable of running on Microsoft Windows, the Solaris™ Operating System (Solaris OS), Linux, or Mac OS. Some virtual machines, such as the Java SE HotSpot at a Glance, perform additional steps at runtime to give your application a performance boost. This includes various tasks such as finding performance bottlenecks and recompiling (to native code) frequently used sections of code.



```
class HelloWorldApp {
        public static void main(String [] args) {
                System.out.println("Hello World!");
        }
}
```

Through the Java VM, the same application is capable of running on multiple platforms.

The Java Platform

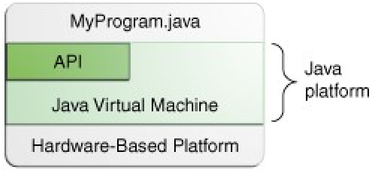
A *platform* is the hardware or software environment in which a program runs. We've already mentioned some of the most popular platforms like Microsoft Windows, Linux, Solaris OS, and Mac OS. Most platforms can be described as a combination of the operating system and underlying hardware. The Java platform differs from most other platforms in that it's a software-only platform that runs on top of other hardware-based platforms.

The Java platform has two components:

- The *Java Virtual Machine*
- The *Java Application Programming Interface* (API)

You've already been introduced to the Java Virtual Machine; it's the base for the Java platform and is ported onto various hardware-based platforms.

The API is a large collection of ready-made software components that provide many useful capabilities. It is grouped into libraries of related classes and interfaces; these libraries are known as *packages*. The next section, What Can Java Technology Do? highlights some of the functionality provided by the API.



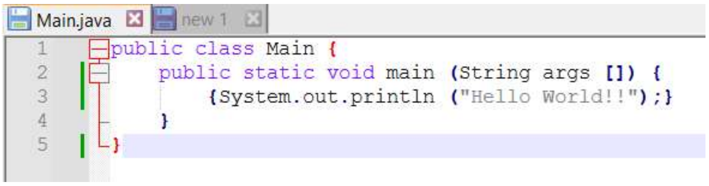
The API and Java Virtual Machine insulate the program from the underlying hardware.

As a platform-independent environment, the Java platform can be a bit slower than native code. However, advances in compiler and virtual machine technologies are bringing performance close to that of native code without threatening portability.

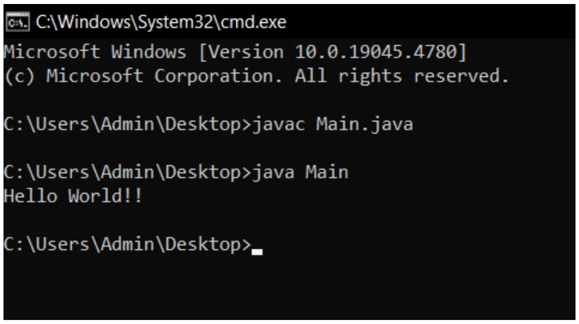
The terms"Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java platform.

From <https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html>

1. **Hello World Program**: Write a Java program that prints "Hello World!!" to the console.

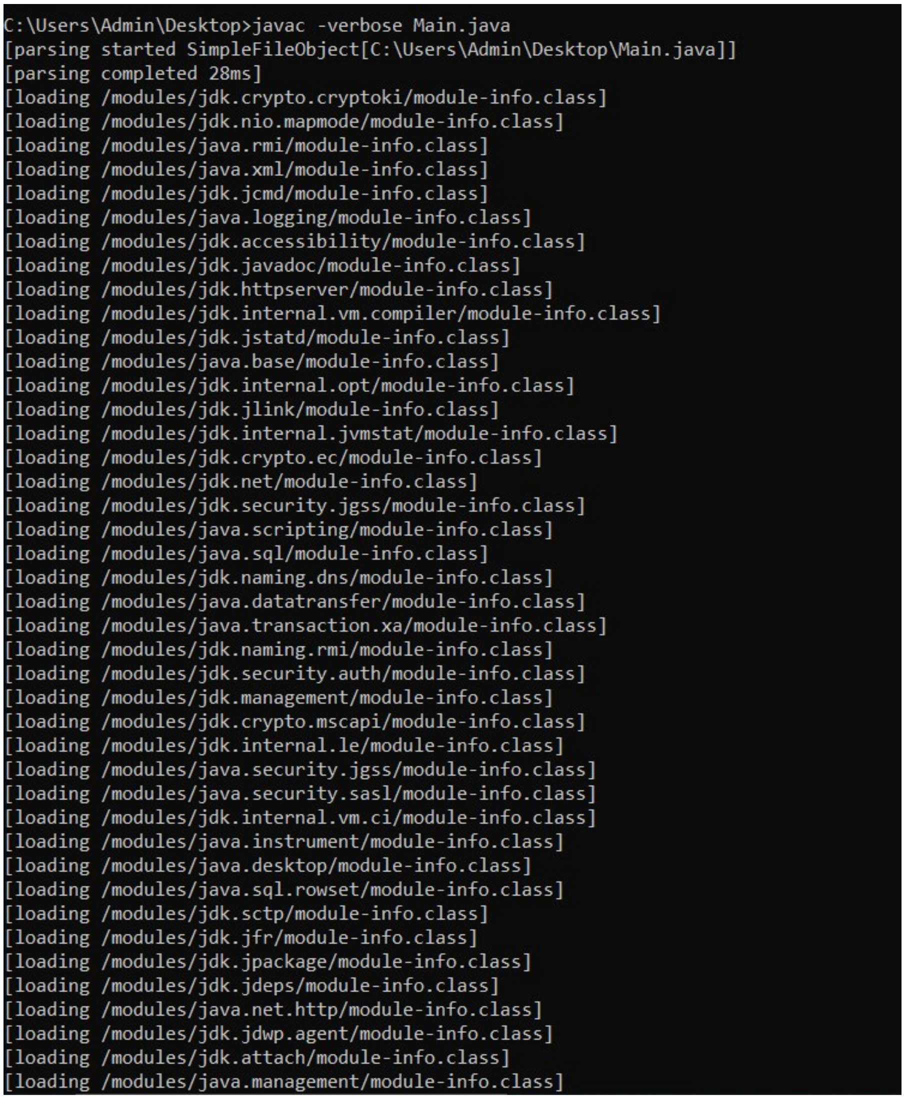


```
public class Main {
    public static void main (String args []) {
        {System.out.println ("Hello World!!");}
    }
}
```



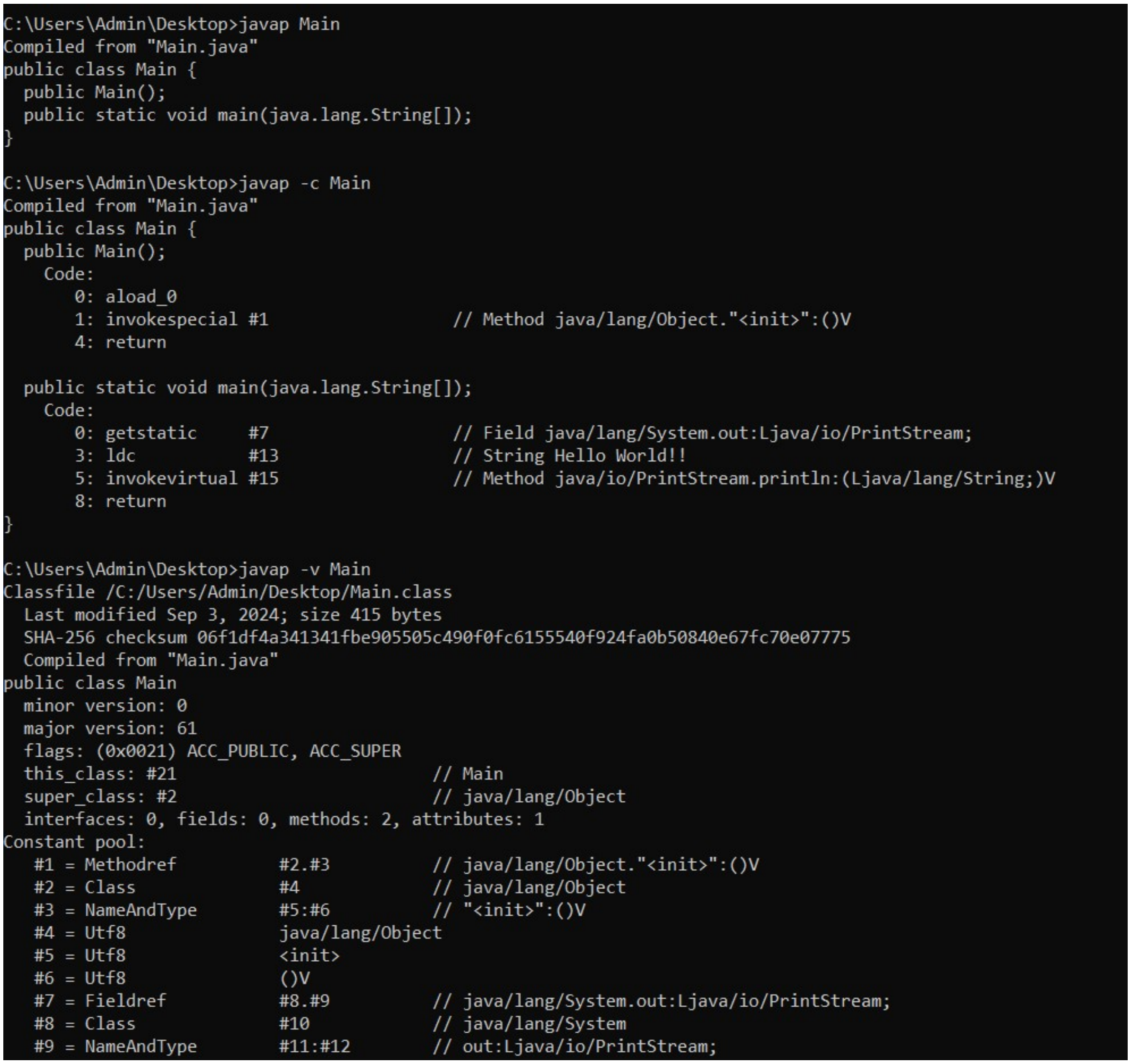
```
Microsoft Windows [Version 10.0.19045.4780]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Admin\Desktop>javac Main.java

C:\Users\Admin\Desktop>java Main
Hello World!!

C:\Users\Admin\Desktop>
```

2. **Compile with Verbose Option**: Compile your Java file using the `-verbose` option with `javac`. Check the output. It will show the complete process of compilation.

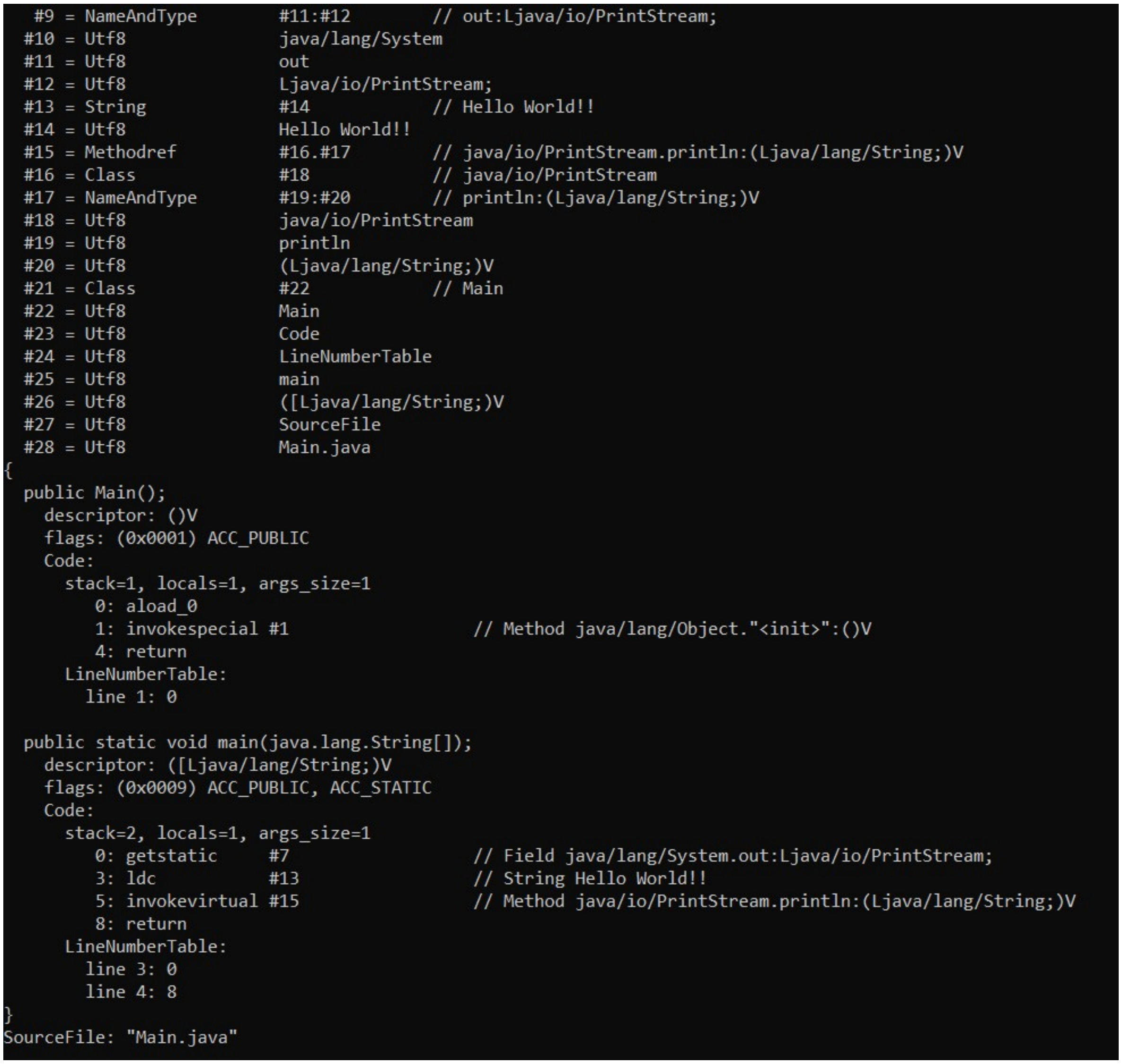


```
C:\Users\Admin\Desktop>javac -verbose Main.java
[parsing started SimpleFileObject[C:\Users\Admin\Desktop\Main.java]]
[parsing completed 28ms]
[loading /modules/jdk.crypto.cryptoki/module-info.class]
[loading /modules/jdk.nio.mapmode/module-info.class]
[loading /modules/java.rmi/module-info.class]
[loading /modules/java.xml/module-info.class]
[loading /modules/jdk.jcmd/module-info.class]
[loading /modules/java.logging/module-info.class]
[loading /modules/jdk.accessibility/module-info.class]
[loading /modules/jdk.javadoc/module-info.class]
[loading /modules/jdk.httpserver/module-info.class]
[loading /modules/jdk.internal.vm.compiler/module-info.class]
[loading /modules/jdk.jstatd/module-info.class]
[loading /modules/java.base/module-info.class]
[loading /modules/jdk.internal.opt/module-info.class]
[loading /modules/jdk.jlink/module-info.class]
[loading /modules/jdk.internal.jvmstat/module-info.class]
[loading /modules/jdk.crypto.ec/module-info.class]
[loading /modules/jdk.net/module-info.class]
[loading /modules/jdk.security.jgss/module-info.class]
[loading /modules/java.scripting/module-info.class]
[loading /modules/java.sql/module-info.class]
[loading /modules/jdk.naming.dns/module-info.class]
[loading /modules/java.datatransfer/module-info.class]
[loading /modules/java.transaction.xa/module-info.class]
[loading /modules/jdk.naming.rmi/module-info.class]
[loading /modules/jdk.security.auth/module-info.class]
[loading /modules/jdk.management/module-info.class]
[loading /modules/jdk.crypto.mscapi/module-info.class]
[loading /modules/jdk.internal.le/module-info.class]
[loading /modules/java.security.jgss/module-info.class]
[loading /modules/java.security.sasl/module-info.class]
[loading /modules/jdk.internal.vm.ci/module-info.class]
[loading /modules/java.instrument/module-info.class]
[loading /modules/java.desktop/module-info.class]
[loading /modules/java.sql.rowset/module-info.class]
[loading /modules/jdk.sctp/module-info.class]
[loading /modules/jdk.jfr/module-info.class]
[loading /modules/jdk.jpackage/module-info.class]
[loading /modules/jdk.jdeps/module-info.class]
[loading /modules/java.net.http/module-info.class]
[loading /modules/jdk.jdwp.agent/module-info.class]
[loading /modules/jdk.attach/module-info.class]
[loading /modules/java.management/module-info.class]
```

3. **Inspect Bytecode**: Use the `javap` tool to examine the bytecode of the compiled `.class` file. Observe the output.



```
C:\Users\Admin\Desktop>javap Main
Compiled from "Main.java"
public class Main {
  public Main();
  public static void main(java.lang.String[]);
}

C:\Users\Admin\Desktop>javap -c Main
Compiled from "Main.java"
public class Main {
  public Main();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #13                 // String Hello World!!
       5: invokevirtual #15                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}

C:\Users\Admin\Desktop>javap -v Main
Classfile /C:/Users/Admin/Desktop/Main.class
  Last modified Sep 3, 2024; size 415 bytes
  SHA-256 checksum 06f1df4a341341fbe905505c490f0fc6155540f924fa0b50840e67fc70e07775
  Compiled from "Main.java"
public class Main
  minor version: 0
  major version: 61
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #21                         // Main
  super_class: #2                         // java/lang/Object
  interfaces: 0, fields: 0, methods: 2, attributes: 1
Constant pool:
   #1 = Methodref          #2.#3          // java/lang/Object."<init>":()V
   #2 = Class              #4             // java/lang/Object
   #3 = NameAndType        #5:#6          // "<init>":()V
   #4 = Utf8               java/lang/Object
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Fieldref           #8.#9          // java/lang/System.out:Ljava/io/PrintStream;
   #8 = Class              #10            // java/lang/System
   #9 = NameAndType        #11:#12        // out:Ljava/io/PrintStream;
```



```
   #9 = NameAndType        #11:#12        // out:Ljava/io/PrintStream;
  #10 = Utf8               java/lang/System
  #11 = Utf8               out
  #12 = Utf8               Ljava/io/PrintStream;
  #13 = String             #14            // Hello World!!
  #14 = Utf8               Hello World!!
  #15 = Methodref          #16.#17        // java/io/PrintStream.println:(Ljava/lang/String;)V
  #16 = Class              #18            // java/io/PrintStream
  #17 = NameAndType        #19:#20        // println:(Ljava/lang/String;)V
  #18 = Utf8               java/io/PrintStream
  #19 = Utf8               println
  #20 = Utf8               (Ljava/lang/String;)V
  #21 = Class              #22            // Main
  #22 = Utf8               Main
  #23 = Utf8               Code
  #24 = Utf8               LineNumberTable
  #25 = Utf8               main
  #26 = Utf8               ([Ljava/lang/String;)V
  #27 = Utf8               SourceFile
  #28 = Utf8               Main.java
{
  public Main();
    descriptor: ()V
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 1: 0

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: (0x0009) ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #13                 // String Hello World!!
         5: invokevirtual #15                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
         8: return
      LineNumberTable:
        line 3: 0
        line 4: 8
}
SourceFile: "Main.java"
```

JVM Architecture :

very Java developer knows that bytecode will be executed by the **JRE** (Java Runtime Environment). But many don't know the fact that JRE is the implementation of **Java Virtual Machine** (JVM), which analyzes the bytecode, interprets the code, and executes it. It is very important, as a developer, that we know the architecture of the JVM, as it enables us to write code more efficiently. In this article, we will learn more deeply about the JVM architecture in Java and different components of the JVM.

What Is the JVM?

A **Virtual Machine** is a software implementation of a physical machine. Java was developed with the concept of **WORA (*Write Once Run Anywhere*)**, which runs on a **VM**. The **compiler** compiles the Java file into a Java **.class** file, then that .class file is input into the JVM, which loads and executes the class file.

How Does the JVM Work?

As shown in the above architecture diagram, the JVM is divided into three main subsystems:

1. ClassLoader Subsystem
2. Runtime Data Area
3. Execution Engine

1. ClassLoader Subsystem

Java's dynamic class loading functionality is handled by the ClassLoader subsystem. It loads, links. and initializes the class file when it refers to a class for the first time at runtime, not compile time.

1.1 Loading

Classes will be loaded by this component. BootStrap ClassLoader, Extension ClassLoader, and Application ClassLoader are the three ClassLoaders that will help in achieving it.

1. **BootStrap** ClassLoader – Responsible for loading classes from the bootstrap class path, nothing but **rt.jar. Highest priority** will be given to this loader.
2. **Extension ClassLoader** – Responsible for loading classes which are inside the ext folder **(jre\lib).**
3. **Application ClassLoader** –Responsible for loading Application Level Class path, path mentioned Environment Variable, etc.

The above Class Loaders will follow **Delegation Hierarchy Algorithm** while loading the class files.

1.2 Linking

1. **Verify** – Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the verification error.
2. **Prepare** – For all static variables memory will be allocated and assigned with default values.
3. **Resolve** – All symbolic memory references are replaced with the original references from Method Area.

1.3 Initialization

This is the final phase of ClassLoading; here, all static variables will be assigned with the original values, and the static block will be executed.

2. Runtime Data Area

The Runtime Data Area is divided into five major components:

1. **Method Area** – All the class-level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource.
2. **Heap Area** – All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe.
3. **Stack Area**– For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called Stack Frame. All local variables will be created in the stack memory. The stack area is thread-safe since it is not a shared resource. The Stack Frame is divided into three subentities:
    1. **Local Variable Array** – Related to the method how many local variables are involved and the corresponding values will be stored here.
    2. **Operand stack** – If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation.
    3. **Frame data** – All symbols corresponding to the method is stored here. In the case of any **exception**, the catch block information will be maintained in the frame data.
4. **PC Registers** – Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.
5. **Native Method stacks** – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

3. Execution Engine

The bytecode, which is assigned to the **Runtime Data Area,** will be executed by the Execution Engine. The Execution Engine reads the bytecode and executes it piece by piece.

1. **Interpreter** – The interpreter interprets the bytecode faster but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.
2. **JIT Compiler**– The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.
    1. **Intermediate Code Generator** – Produces intermediate code
    2. **Code Optimizer** – Responsible for optimizing the intermediate code generated above

    3. **Target Code Generator –** Responsible for Generating Machine Code or Native Code
    4. **Profiler –** A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.
3. **Garbage Collector**: Collects and removes unreferenced objects. Garbage Collection can be triggered by calling `System.gc()`, but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

Java Native Interface (JNI): JNI will be interacting with the Native Method Libraries and provides the Native Libraries required for the Execution Engine.

Native Method Libraries: This is a collection of the Native Libraries, which is required for the Execution Engine.