Here is a **complete version-wise list of Java features and enhancements** from **Java 1.0 to Java 24**, focused on what developers gained in each release

**🔹 Java 1.0 (1996)**

* Object-Oriented Programming (OOP)
* Platform Independence
* Automatic Garbage Collection
* Thread support
* Applets

**🔹 Java 1.1 (1997)**

* Inner Classes
* JavaBeans
* JDBC
* Reflection API
* RMI

**🔹 Java 1.2 (1998) – Java 2**

* Swing API
* Collections Framework
* JIT Compiler
* Java Plug-in
* Security Model improvements

**🔹 Java 1.3 (2000)**

* HotSpot JVM
* RMI over IIOP
* JavaSound API

**🔹 Java 1.4 (2002)**

* assert keyword
* Regular Expressions
* Exception Chaining
* NIO (New I/O)
* Logging API

**🔹 Java 5 (2004) – J2SE 5.0**

* Generics
* Enhanced for-each loop
* Autoboxing/Unboxing
* Varargs
* Static Import
* Annotations
* Enums
* java.util.concurrent

**🔹 Java 6 (2006)**

* Scripting API (JSR 223)
* Compiler API
* Pluggable annotations
* JDBC 4.0
* Web Services (JAX-WS)

**🔹 Java 7 (2011) – Project Coin**

* Strings in switch
* Binary Literals
* Underscores in numeric literals
* Try-with-resources
* Diamond Operator (<>)
* NIO 2.0
* Fork/Join Framework
* Multi-catch exceptions

**🔹 Java 8 (2014)**

* Lambda Expressions
* Streams API
* Default/Static Methods in Interfaces
* Optional Class
* Date-Time API (java.time)
* Nashorn JS Engine
* Method References
* Functional Interfaces

**🔹 Java 9 (2017)**

* Modular System (Project Jigsaw)
* JShell (REPL)
* Private Methods in Interfaces
* Stream API Improvements
* Process API Updates
* Collection Factory Methods
* HTTP/2 Client

**🔹 Java 10 (2018)**

* var keyword
* Garbage Collector Interface
* Application Class-Data Sharing

**🔹 Java 11 (2018)**

* var in Lambda Parameters
* HTTP Client API (Standardized)
* String & File API Improvements
* Nest-Based Access Control
* Removed JavaFX from JDK

**🔹 Java 12 (2019)**

* Switch Expressions (Preview)
* Shenandoah GC
* JVM Constants API
* Compact Number Formatting

**🔹 Java 13 (2019)**

* Text Blocks (Preview)
* Legacy Socket API Reimplementation
* Switch Expressions (2nd Preview)

**🔹 Java 14 (2020)**

* Switch Expressions (Final)
* Text Blocks (Final)
* Records (Preview)
* Pattern Matching for instanceof (Preview)
* Helpful NullPointerExceptions

**🔹 Java 15 (2020)**

* Sealed Classes (Preview)
* Hidden Classes
* ZGC Enhancements

**🔹 Java 16 (2021)**

* Records (Final)
* Pattern Matching for instanceof (Final)
* Sealed Classes (2nd Preview)
* Unix-Domain Socket Channels
* Stream.toList()

**🔹 Java 17 (2021) – LTS**

* Sealed Classes (Final)
* Pattern Matching for switch (Preview)
* New macOS Rendering Pipeline
* Applet API Deprecated
* Strong Encapsulation of JDK Internals

**🔹 Java 18 (2022)**

* Simple Web Server
* UTF-8 by Default
* Pattern Matching for switch (2nd Preview)

**🔹 Java 19 (2022)**

* Virtual Threads (Preview)
* Structured Concurrency (Incubator)
* Pattern Matching for switch (3rd Preview)
* Foreign Function & Memory API (Preview)

**🔹 Java 20 (2023)**

* Virtual Threads (2nd Preview)
* Record Patterns (Preview)
* Pattern Matching for switch (4th Preview)
* Scoped Values (Incubator)

**🔹 Java 21 (2023) – LTS**

* Record Patterns (Final)
* Virtual Threads (Final)
* String Templates (Preview)
* Sequenced Collections
* Pattern Matching for switch (Final)

**🔹 Java 22 (2024)**

* Scoped Values (Final)
* Structured Concurrency (Final)
* String Templates (Final)

**🔹 Java 23 (2024)**

* Record Patterns (Final)
* Pattern Matching for switch (Enhanced)
* Foreign Function & Memory API (3rd Preview)
* Vector API (6th Incubator)

**🔹 Java 24 (2025)**

* Ahead-of-Time Class Loading
* Virtual Threads without Pinning
* Compact Object Headers (Experimental)
* Stream Gatherers (Final)
* Primitive Types in Patterns & Switch (2nd Preview)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The **Foreign Function & Memory (FFM) API** is a major feature in Java that allows developers to interact with native code and memory directly from Java, without using the older and more complex Java Native Interface (JNI). Here's a breakdown of its **features and benefits** for developers:

**🔧 Key Features of the FFM API**

**1. Foreign Function Access**

* Call native functions (e.g., from C libraries) directly from Java.
* No need for JNI boilerplate code.
* Uses MethodHandle and FunctionDescriptor for type-safe calls.

**2. Memory Access**

* Allocate and manipulate off-heap memory using MemorySegment.
* Safe and structured access with bounds checking.
* Supports both manual and automatic memory management.

**3. Memory Layouts**

* Define structured memory (like C structs) using MemoryLayout.
* Enables mapping complex native data structures to Java.

**4. Symbol Lookup**

* Locate native symbols (functions or variables) using SymbolLookup.
* Works with system libraries or custom native libraries.

**5. Linker API**

* Linker binds Java code to native functions.
* Supports downcalls (Java → native) and upcalls (native → Java).

**6. Upcalls**

* Pass Java functions to native code as callbacks.
* Enables event-driven or asynchronous native interactions.

**🚀 Developer Benefits**

* **Performance**: Direct memory access and native calls improve speed.
* **Safety**: Type-safe APIs and bounds checking reduce errors.
* **Simplicity**: Eliminates JNI complexity and improves readability.
* **Portability**: Works across platforms with minimal changes.
* **Modern API Design**: Fluent, modular, and easy to use.

**📅 Evolution Across Java Versions**

| **Java Version** | **Status of FFM API** | **Notes** |
| --- | --- | --- |
| Java 14–16 | Incubator | Early experimental phase |
| Java 17–20 | Preview | API refinement and stability improvements |
| Java 21 | Stable (partial) | Widely usable in production |
| Java 22–24 | Finalizing | Enhanced performance, usability, and safety |

Here's a **Java project template** that demonstrates the **Foreign Function & Memory API** with two examples:

1. **Calling a native C function (sqrt)**
2. **Allocating and accessing native memory**

**📁 Project Structure**

FFM\_API\_Example/

└── src/

└── FFMExample.java

**📄 FFMExample.java**

import java.lang.foreign.\*;

import java.lang.invoke.MethodHandle;

import java.lang.invoke.MethodHandles;

import java.lang.invoke.MethodType;

public class FFMExample {

    public static void main(String[] args) throws Throwable {

        Linker linker = Linker.nativeLinker();

        SymbolLookup lookup = SymbolLookup.systemLookup();

        // Lookup the native sqrt function

        MemorySegment sqrtFunc = lookup.lookup("sqrt").orElseThrow();

        // Define the function signature: double sqrt(double)

        FunctionDescriptor descriptor = FunctionDescriptor.of(ValueLayout.JAVA\_DOUBLE, ValueLayout.JAVA\_DOUBLE);

        // Create a method handle to call the native function

        MethodHandle handle = linker.downcallHandle(sqrtFunc, descriptor);

        // Call the native sqrt function

        double input = 49.0;

        double result = (double) handle.invoke(input);

        System.out.println("Native sqrt(" + input + ") = " + result);

        // Demonstrate native memory allocation

        try (Arena arena = Arena.ofConfined()) {

            MemorySegment segment = arena.allocate(ValueLayout.JAVA\_INT);

            segment.set(ValueLayout.JAVA\_INT, 0, 12345);

            int value = segment.get(ValueLayout.JAVA\_INT, 0);

            System.out.println("Value in native memory: " + value);

        }

    }

}

**🛠️ Requirements**

* Java **21 or later**
* No additional native libraries needed (uses system sqrt)

**▶️ How to Compile and Run**

**javac --enable-preview --release 21 src/FFMExample.java**

**java --enable-preview -cp src FFMExample**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Here's a step-by-step guide to Java Stream Gatherers, a powerful feature introduced in Java 24 that allows developers to create custom intermediate operations in stream pipelines.**

**🔍 What Are Stream Gatherers?**

**Stream Gatherers are like enhanced versions of intermediate operations (map, filter, etc.) but with more control. Unlike Collector (which is terminal), Gatherers operate mid-stream, allowing you to:**

* **Group elements into batches**
* **Accumulate values progressively**
* **Reorder or deduplicate elements**
* **Short-circuit stream processing**
* **Process elements in parallel**

**🧱 Built-in Gatherers in Java 24**

**1. windowFixed(int size)**

**Groups elements into non-overlapping fixed-size batches.**

**✅ Example:**

orders.stream()

    .gather(Gatherers.windowFixed(2))

    .forEach(batch -> {

        System.out.println("Batch:");

        batch.forEach(order -> System.out.println(" - " + order.customerName()));

    });

**2. windowSliding(int size)**

**Creates overlapping windows of elements.**

**✅ Example:**

orders.stream()

    .gather(Gatherers.windowSliding(3))

    .forEach(window -> {

        System.out.println("Window:");

        window.forEach(order -> System.out.println(" - " + order.customerName()));

    });

**3. fold(Supplier<A>, BiFunction<A, T, A>)**

**Accumulates values into a single result (like reduce but mid-stream).**

**✅ Example:**

orders.stream()

    .limit(3)

    .gather(Gatherers.fold(

        () -> "Summary:\n",

        (summary, order) -> summary + "- " + order.customerName() + " ordered " + order.product() + "\n"

    ))

    .findFirst()

    .ifPresent(System.out::println);

**4. scan(Supplier<A>, BiFunction<A, T, A>)**

**Like fold, but emits intermediate results after each element.**

**✅ Example:**

orders.stream()

    .limit(3)

    .gather(Gatherers.scan(

        () -> "Progress:\n",

        (summary, order) -> summary + "- " + order.customerName() + " ordered " + order.product() + "\n"

    ))

    .forEach(System.out::println);

**5. mapConcurrent(int parallelism, Function<T, R>)**

**Applies a function to each element concurrently.**

**✅ Example:**

orders.stream()

    .gather(Gatherers.mapConcurrent(2, order ->

        "Shipping Label: " + order.customerName() + " - " + order.product()))

    .forEach(System.out::println);

**🛠️ Custom Gatherer Example**

**You can build your own gatherer using the Gatherer.of(...) method:**

Gatherer.of(

    () -> new HashMap<String, List<Order>>(), // initializer

    (map, order, downstream) -> {

        map.computeIfAbsent(order.category(), k -> new ArrayList<>()).add(order);

        return true;

    },

    (map1, map2) -> map1, // combiner

    (map, downstream) -> {

        map.forEach((category, orders) -> {

            List<Order> topOrders = orders.stream()

                .sorted(Comparator.comparing(Order::orderDate).reversed())

                .limit(3)

                .toList();

            downstream.push(Map.entry(category, topOrders));

        });

    }

);

**🧪 How to Run Gatherer Code**

**Since Gatherers are a preview feature, compile and run with:**