30分でわかる！JVM簡単ツアー

Java/JVM Meetup

2025/05/23

by [@kmizu](https://x.com/kmizu)

はじめに

- **発表者:** kmizu (@kmizu)
- JavaはXX年くらい書いています（例: 15年以上）
  - ※自己紹介に合わせて適宜変更してください
- 普段はソフトウェアエンジニアとして、主にバックエンド開発などに携わっています。
  - ※自己紹介に合わせて適宜変更してください

本日は **Java Virtual Machine (JVM)** の世界へようこそ！

30分という短い時間ですが、JVMの基本を巡るツアーにご案内します。

本日のアジェンダ

1. **JVMとは何か？**
    - JVMの役割と「Write Once, Run Anywhere」
2. **JVMのアーキテクチャ**
    - クラスローダー、ランタイムデータエリア、実行エンジン
3. **クラスファイルフォーマット**
    - JavaコードがJVMで動く形になるまで
4. **JVMバイトコード命令入門**
    - JVMが理解する「言葉」
5. **JVMの進化と現在**
    - Javaの進化を支えるJVM

Javaの基本知識がある方を対象に、JVMの内部を少しだけ覗いてみましょう。

JavaとJVMの約30年

- **1995年:** Java 1.0 (Oakとして開発スタート) とJVMが登場
  - 当時のキャッチフレーズ「Write Once, Run Anywhere」
- 以来、Java言語は驚くほど進化を遂げました。
  - Generics (Java 5), Lambda Expressions (Java 8), Modules (Java 9), Records (Java 16), Virtual Threads (Java 21)...
- JVMの性能も大幅に向上し、最適化が進みました。
  - HotSpot VMの登場、JITコンパイラの進化、多様なGCアルゴリズム
- しかし、JVMの**基本アーキテクチャ自体は驚くほど変わっていません。**
  - この安定した基盤が、Javaエコシステムの発展を支えています。

1. JVMとは何か？

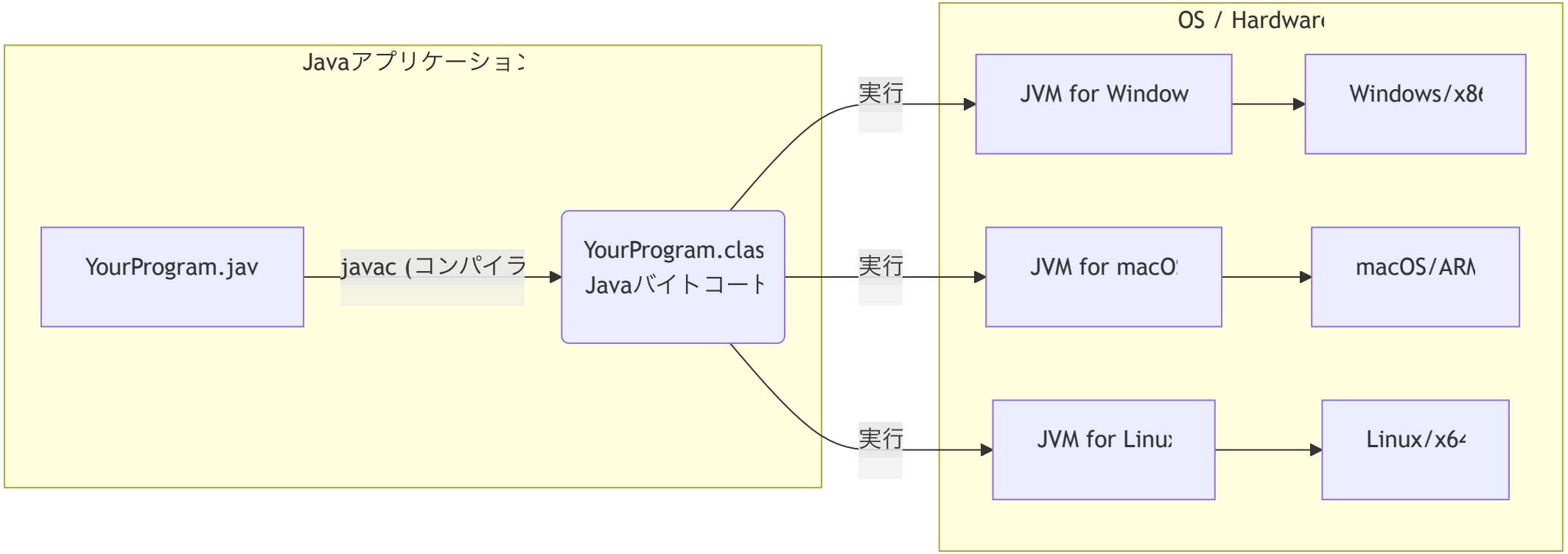
Java Virtual Machine (Java仮想マシン)

- Javaプログラムを実行するための**仮想的なコンピュータ**です。
- 物理的なハードウェアやOSに直接依存せず、Javaバイトコードを実行します。

"Write Once, Run Anywhere" (WORA) を実現する中核技術

プラットフォーム非依存の実現

OSやCPUアーキテクチャの違いをJVMが吸収します。



開発者は一度、.class、ファイルを作成すれば、異なる環境でも同じように動作させることが期待できます。

JVM言語

JVMはJava言語専用ではありません。多くの言語がJVM上で動作します。

- **Scala:** オブジェクト指向 + 関数型。静的型付け。
- **Kotlin:** 簡潔で安全なJava代替。Android公式開発言語。静적型付け。
- **Groovy:** 動的型付け。スクリプティングやDSL構築に強み。
- **Clojure:** Lisp方言。関数型プログラミング、不変データ構造。動的型付け。
- 他にも JRuby, Jython, Frege など多数...

これらの言語もコンパイルされるとJavaバイトコードになり、JVM上で動作します。
Javaの豊富なライブラリやツールといったエコシステムの恩恵を受けられます。

2. JVMのアーキテクチャ

JVMは、大きく分けて以下のコンポーネントで構成されています。

クラスローダー (Class Loader)

役割:

- Javaのクラスファイル (`.class`) をファイルシステムやネットワークから探し出し、JVMのメモリ（ランタイムデータエリア）にロードします。
- 単にロードするだけでなく、検証や準備も行います。

主なプロセス:

1. **Loading (ロード):**
    - クラスのバイナリデータを読み込み、メソッドエリアにクラスの内部表現を作成。
2. **Linking (リンク):**
    - **Verification (検証):** クラスファイルがJVM仕様に準拠し、安全か検証。
    - **Preparation (準備):** クラス変数（静的変数）のメモリ割り当てとデフォルト初期化

クラスローダーの種類 (階層構造)

1. **Bootstrap Class Loader (ブートストラップクラスローダー)**
   - JVM自身の起動に必要なコアクラス (`java.lang.Object` 等) をロード。
   - 通常、ネイティブコードで実装。
2. **Platform Class Loader (プラットフォームクラスローダー)** (Java 9以降)
   - (旧 Extension Class Loader) Java SEプラットフォームAPIやその実装クラスをロード。
3. **System Class Loader (システムクラスローダー) / Application Class Loader**
   - アプリケーションのクラスパス (`-cp` や環境変数 `CLASSPATH`) 上のクラスをロード。
   - 私たちが書くクラスの多くはこれでロードされます。

これらは親子関係を持ち、クラス探索は子から親へ委譲されます (Delegation Model)。

ランタイムデータエリア (Runtime Data Areas)

JVMがプログラム実行中に使用するメモリ領域です。スレッド共有のものと、スレッド毎のものがあります。

スレッド共有:

- **メソッドエリア (Method Area)**
- **ヒープ (Heap)**

スレッド毎 (Thread-specific):

- **JVMスタック (JVM Stacks)**
- **PCレジスタ (Program Counter Registers)**
- **ネイティブメソッドスタック (Native Method Stacks)**

メソッドエリア & ヒープ (スレッド共有)

メソッドエリア (Method Area):

- ロードされたクラスの情報（型情報、フィールド情報、メソッド情報、メソッドのバイトコード）、静的変数、**実行時定数プール**などを格納。
- HotSpot VMでは、Java 8以降「メタスペース (Metaspace)」としてネイティブメモリに配置。

ヒープ (Heap):

- **オブジェクトインスタンス** (`new` されたもの) や**配列**が格納される最大のメモリ領域。
- **ガベージコレクション (GC)** の主要な対象。
    - Young領域 (Eden, Survivor) と Old領域 (Tenured) に分かれることが多い。

JVMスタック (スレッド毎)

- メソッド呼び出しの情報を管理。スレッド開始時に作成。
- メソッドが呼び出されるたびに、新しい**フレーム (Frame)** がスタックに積まれます。
- メソッドが終了すると、対応するフレームがポップされます。
- 各フレームには以下が含まれます：
  - **ローカル変数配列 (Local Variables):** メソッドの引数やローカル変数を格納。
  - **オペランドスタック (Operand Stack):** バイトコード命令の実行時に値を一時的に置く作業領域 (LIFO)。
  - **フレームデータ (Frame Data):** 現在のメソッドが属するクラスへの参照 (実行時定数プールへの参照)、例外処理情報など。

```
  Thread 1 Stack
 +---------------+
```

PCレジスタ & ネイティブメソッドスタック (スレッド毎)

PCレジスタ (Program Counter Registers):

- 現在実行中のJVMバイトコード命令の**アドレス**を指すポインタ。
- メソッドがネイティブ (Java以外の言語) の場合、PCレジスタの値は未定義。

ネイティブメソッドスタック (Native Method Stacks):

- Javaコードからネイティブメソッド (例: C/C++で書かれたライブラリ関数) を呼び出す際に使用されるスタック。
- JNI (Java Native Interface) を介してネイティブコードを実行する場合に利用。
- 多くのJVM実装では、JVMスタックと統合されているか、同様の仕組みで提供。

実行エンジン (Execution Engine)

クラスローダーによってロードされ、ランタイムデータエリアに配置されたバイトコードを実行します。

- **インタープリタ (Interpreter):**
  - バイトコード命令を1つずつ解釈し、逐次実行する。
  - 起動は速いが、実行速度は比較的遅い。
- **JITコンパイラ (Just-In-Time Compiler):**
  - プログラム実行中に、頻繁に実行される「ホットスポット」と呼ばれるコード部分を特定。
  - その部分のバイトコードを、実行環境のCPUに最適化された**ネイティブコードにコンパイル**して高速化。
  - 例: HotSpot VMのC1 (Client), C2 (Server) コンパイラ。
- **ガベージコレクタ (Garbage Collector, GC):**

3. クラスファイルフォーマット

Javaソースコード (`.java`) は、Javaコンパイラ (`javac`) によってコンパイルされ、JVMが実行できる形式である**クラスファイル (`.class`)** に変換されます。

```
MyProgram.java  ----- (javac MyProgram.java) ----->  MyProgram.class
```

- クラスファイルは、特定の構造を持つバイナリファイルです。
- その構造はJVM仕様で厳密に定義されており、プラットフォーム非依存性を保証します。
- 1つの `.java` ファイルから、複数の `.class` ファイルが生成されることもあります (内部クラスなど)。

クラスファイルの主要な構成要素

`MyProgram.class` の中身（概要）：

フィールド	説明
`magic`	`0xCAFEBABE` という4バイトのマジックナンバー (クラスファイルであることの印)
`minor_version`	クラスファイルのマイナーバージョン
`major_version`	クラスファイルのメジャーバージョン (Javaのバージョンと対応)
`constant_pool_count`	定数プールのエントリ数
`constant_pool`	文字列リテラル、クラス名、メソッド名、フィールド名など、様々な定数を格納するテーブル
	クラスのアクセス修飾子 (public, final, interface, abstractな

`javap` コマンドでクラスファイルを見る

Javaには `javap` というクラスファイル逆アセンブラが付属しています。
これを使うと、クラスファイルの構造やバイトコードを確認できます。

簡単なJavaクラス:

```java
// Sample.java
public class Sample {
    private String message = "Hello";

    public void printMessage() {
        System.out.println(this.message);
    }
}
```

コンパイル: `javac Sample.java`

詳細表示: `javap -v Sample` (`-c` でバイトコードのみ、 `-p` でprivateメンバも表示)

`javap -v Sample` 出力例 (抜粋) - クラス情報

```
Classfile /path/to/Sample.class
  Last modified 2025/05/07; size XXX bytes // 日付やサイズは環境による
  SHA-256 checksum ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890 // ダミーのチェックサム
  Compiled from "Sample.java"
public class Sample
  minor version: 0
  major version: 61 // Java 17 でコンパイルした場合（例）
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER // public class であることを示す
  this_class: #5                          // constant_pool の #5 が Sample
  super_class: #2                         // constant_pool の #2 が java/lang/Object
  interfaces: 0, fields: 1, methods: 2, attributes: 1
```

- `major version` : 52(Java 8), 55(Java 11), 61(Java 17), 65(Java 21), 66(Java 22), 68(Java 24)
- `flags` : クラスのアクセス権や性質を示します。

`javap -v Sample` 出力例 (抜粋) - 定数プール

```
Constant pool:
   #1 = Methodref          #2.#3          // java/lang/Object."<init>":()V
   #2 = Class              #4             // java/lang/Object
   #3 = NameAndType        #20:#21        // "<init>":"()V"
   #4 = Utf8               java/lang/Object
   #5 = Class              #22            // Sample
   #6 = Fieldref           #5.#23         // Sample.message:Ljava/lang/String;
   #7 = Fieldref           #24.#25        // java/lang/System.out:Ljava/io/PrintStream;
   #8 = Methodref          #26.#27        // java/io/PrintStream.println:(Ljava/lang/String;)V
   // ... (中略) ...
  #18 = Utf8               message
  #19 = Utf8               Ljava/lang/String; // 型ディスクリプタ
  #20 = Utf8               <init> // コンストラクタ名
  #21 = Utf8               ()V    // voidメソッド、引数なし
  #22 = Utf8               Sample
  // ... (さらに続く)
```

- クラスファイル内で使われる文字列、クラス名、メソッド名、型などが集約されています。
- バイトコードからは、この定数プールへのインデックスで参照されます。

`javap -v Sample` 出力例 (抜粋) - フィールドとメソッド (1/2)

```
{ // Fields
  private java.lang.String message;
    descriptor: Ljava/lang/String; // 型ディスクリプタ
    flags: (0x0002) ACC_PRIVATE
}

{ // Methods
  public Sample(); // コンストラクタ
    descriptor: ()V
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=2, locals=1, args_size=1
         0: aload_0       // this をスタックにロード
         1: invokespecial #1 // super() (Object."<init>") 呼び出し
         4: aload_0       // this をスタックにロード
         5: ldc           #9 // 定数プールから "Hello" をロード
         7: putfield      #6 // this.message = "Hello"
        10: return
      // ... LineNumberTable, LocalVariableTableなど (デバッグ情報)
```

`javap -v Sample` 出力例 (抜粋) - フィールドとメソッド (2/2)

```
  public void printMessage();
    descriptor: ()V
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #7  // System.out をスタックにロード
         3: aload_0           // this をスタックにロード
         4: getfield      #6  // this.message をスタックにロード
         7: invokevirtual #8  // System.out.println(this.message) 呼び出し
        10: return
      // ... LineNumberTable, LocalVariableTableなど
}
SourceFile: "Sample.java" // 元のソースファイル名
```

このように `javap` を使うことで、コンパイル後のクラスファイルがどのような構造で、どのようなバイトコードを持っているかを確認できます。

4. JVMバイトコード命令入門

バイトコードとは？

- JVMが直接実行できる命令セットのこと。人間が直接読むアセンブリ言語のようなもの。
- 多くは1バイトのオペコード (命令の種類を示すコード) と、それに続く0個以上のオペランド (命令の対象データや参照) から構成されます。
- **スタックベースのマシン語**: 多くの命令は、メソッドフレーム内の**オペランドスタック**上の値を操作します。データはスタックに積まれ(push)、使われ(pop)、結果がまた積まれます。

例:

- `iload_0` : 0番目のローカル変数 (int型) をオペランドスタックにプッシュ。 (オペコードのみ)

主なバイトコード命令カテゴリ (1/2)

- **ロード命令 (Load):** ローカル変数から値をオペランドスタックにロード。
  - `iload_n` (int), `lload_n` (long), `fload_n` (float), `dload_n` (double), `aload_n` (参照) (n=0-3)
  - `iload <index>`, `lload <index>`, ... (より広い範囲のローカル変数用)
- **ストア命令 (Store):** オペランドスタックの値をローカル変数にストア。
  - `istore_n`, `lstore_n`, `fstore_n`, `dstore_n`, `astore_n` (n=0-3)
  - `istore <index>`, `lstore <index>`, ...
- **定数ロード命令 (Constant):** 定数をオペランドスタックにロード。
  - `aconst_null` (null), `iconst_m1` (-1), `iconst_0` (0), ..., `iconst_5` (5)
  - `bipush <byte>` (byte値), `sipush <short>` (short値)
  - `ldc`, `ldc_w`, `ldc2_w` (定数プールからint, float, String, Class, MethodType, MethodHandle, Dynamic Constant)

主なバイトコード命令カテゴリ (2/2)

- **演算命令 (Math):** スタック上の値を計算。
  - `iadd` (int加算), `lsub` (long減算), `fmul` (float乗算), `ddiv` (double除算), `irem` (int剰余)
  - `ineg` (int符号反転), `ishl` (int左シフト), `iand` (int論理積)
- **型変換命令 (Conversion):** 数値型同士の変換。
  - `i2l` (int to long), `i2f` (int to float), `l2d` (long to double), `d2i` (double to int)
- **オブジェクト操作命令 (Object):**
  - `new <class>` (オブジェクト生成), `newarray <type>` (プリミティブ型配列生成), `anewarray <class>` (参照型配列生成)
  - `getfield <field>`, `putfield <field>` (インスタンスフィールドアクセス)
  - `getstatic <field>`, `putstatic <field>` (スタティックフィールドアクセス)

バイトコード例: 簡単な加算 - Java

```java
// Calculator.java
public class Calculator {
    public int add(int x, int y) {
        int sum = x + y;
        return sum;
    }
}
```

コンパイル: `javac Calculator.java`

逆アセンブル (バイトコードのみ): `javap -c Calculator`

バイトコード例: 簡単な加算 - `add` メソッドのバイトコード

`javap -c Calculator` の `add` メソッド部分:

```
public int add(int, int);
  Code:
     0: iload_1       // ローカル変数スロット1（引数x）の値をスタックにロード
     1: iload_2       // ローカル変数スロット2（引数y）の値をスタックにロード
     2: iadd          // スタックトップの2つのintを加算し、結果をスタックにプッシュ
     3: istore_3      // スタックトップのintをローカル変数スロット3（sum）にストア
     4: iload_3       // ローカル変数スロット3（sum）の値をスタックにロード
     5: ireturn       // スタックトップのintを返り値としてメソッド終了
```

ローカル変数スロットについて:

- インスタンスメソッドの場合:
  - スロット0: `this` (現在のオブジェクト参照)
  - スロット1: 1番目の引数 (`x`)
  - スロット2: 2番目の引数 (`y`)

`add(5, 10)` のオペランドスタックの動き - 1

前提:

- `Calculator` のインスタンスメソッド `add` が `add(5, 10)` として呼び出された。
- ローカル変数 (LV): `[LV0: this, LV1: 5 (x), LV2: 10 (y), LV3: ? (sum)]`
- オペランドスタック (OS): `[]` (空)

`add(5, 10)` のオペランドスタックの動き - 2

`0: iload_1` (ローカル変数スロット1の値 (5) をロード)

- OS: `[5]`

`1: iload_2` (ローカル変数スロット2の値 (10) をロード)

- OS: `[5, 10]` (10がスタックトップ)

`add(5, 10)` のオペランドスタックの動き - 3

`2: iadd` (スタックトップの2つのint (10と5) をポップし、加算 (15)、結果をプッシュ)

- OS: `[15]`

`3: istore_3` (スタックトップのint (15) をポップし、ローカル変数スロット3 (`sum`) にストア)

- OS: `[]`
- LV: `[LV0: this, LV1: 5 (x), LV2: 10 (y), LV3: 15 (sum)]`

`add(5, 10)` のオペランドスタックの動き - 4

`4: iload_3` (ローカル変数スロット3の値 (15) をロード)

- OS: `[15]`

`5: ireturn` (スタックトップのint (15) をポップし、メソッドの返り値とする)

- OS: `[]` (このフレームのスタックはクリアされる)
- 呼び出し元に `15` が返る。

バイトコード例: `System.out.println("Hi");`

Javaコード:

```
System.out.println("Hi");
```

バイトコード (簡略化):

```
   0: getstatic     #SYS_OUT  // System.out（PrintStreamオブジェクト）をスタックにロード
   3: ldc           #HI_STR   // 文字列 "Hi" をスタックにロード
   5: invokevirtual #PRINTLN  // PrintStream.println(String) メソッドを呼び出し
```

1. `getstatic` : `java.lang.System` クラスの静的フィールド `out` (これは `PrintStream` 型のオブジェクト) の参照をスタックに積む。
   - OS: `[<PrintStream obj ref>]`
2. `ldc` : 定数プールから文字列リテラル `"Hi"` の参照をスタックに積む。
   - OS: `[<PrintStream obj ref>, "Hi"]`

5. JVMの進化と現在

JavaとJVMは誕生から約30年、絶えず進化を続けています。

パフォーマンス向上の歴史:

- **JITコンパイラの高度化:**
  - 初期のJVMはほぼインタプリタ実行。HotSpot VM (Java 1.3頃から標準) で高性能なJITコンパイラが導入。
  - より積極的な最適化技術: インライン化、エスケープ解析、ループ最適化、投機的最適化など。
  - 階層型コンパイル (Tiered Compilation, Java 7/8〜): 実行プロファイルに応じてインタプリタ -> C1 (高速起動) -> C2 (最大性能) へと段階的に最適化。
  - GraalVM: 高性能な多言語対応VM。JavaプログラムのAOT (Ahead-of-Time) コンパイルによるネイティブイメージ生成も可能に。
- **ガベージコレクション (GC) の進化:**

言語機能サポートとエコシステムの発展

新しいJava言語機能の効率的なサポート:

- ラムダ式、Stream API (Java 8): `invokedynamic` 命令の活用など。
- モジュールシステム (Project Jigsaw, Java 9): クラスロードの改善。
- ローカル変数型推論 (`var`), Records, Sealed Classes, Pattern Matchingなど、現代的な言語機能を効率よく実行するためのJVM側の対応。

エコシステムの成熟:

- 膨大な数のライブラリとフレームワーク (Spring, Jakarta EE, Apache Commonsなど)。
- 強力な開発ツール (IDE: IntelliJ IDEA, Eclipse, VS Code; ビルドツール: Maven, Gradle)。
- 大規模で活発な開発者コミュニティ。

近年の注目機能 (Java 21 LTS 中心)

Java 21 (2023年9月リリース - 最新LTS):

- **JEP 444: Virtual Threads (仮想スレッド)** (正式機能)
  - 従来のプラットフォームスレッドよりもはるかに軽量なスレッド。
  - 多数のI/Oバウンドなタスクを、少ないOSスレッドで効率的に扱える。
  - サーバサイドアプリケーションのスループットとスケーラビリティ向上に大きく貢献。
- **JEP 440: Record Patterns (レコードパターン)** (正式機能)
- **JEP 441: Pattern Matching for switch** (正式機能)
  - より表現力豊かで安全な条件分岐やデータ分解が可能に。
- **JEP 453: Structured Concurrency (構造化された並行性)** (Preview)
  - 複数の並行タスクをグループとして扱い、エラーハンドリングやキャンセルを容易にするAPI。仮想スレッドと相性が良い。

Java 22 / 23 / 24 と今後

Java 22 (2024年3月リリース):

- JEP 423: Region Pinning for G1 (G1 GCでJNI利用時のレイテンシ削減)
- JEP 458: Launch Multi-File Source-Code Programs (複数ソースファイルをコンパイルなしで直接起動)
- JEP 461: Stream Gatherers (Preview) (Stream APIにカスタム中間操作を追加しやすく)
- JEP 456: Unnamed Variables & Patterns (無名変数・パターン)
- JEP 447: Statements before super(...) (Preview) (`super()`呼び出し前の文)

Java 23 (2024年9月リリース予定):

- JEP 455: Primitive Types in Patterns, instanceof, and switch (Preview) (パターンマッチングでのプリミティブ型)

6. まとめ

- JVMは「Write Once, Run Anywhere」を実現するJavaプラットフォームの中核です。
- **クラスローダー**が `.class` ファイルをロード・リンク・初期化し、**ランタイムデータエリア**に展開します。
- **実行エンジン** (インタプリタ, JITコンパイラ, GC) がバイトコードを実行・最適化し、メモリを管理します。
- **クラスファイル**はJVMが理解できる共通のバイナリフォーマットです。
- **バイトコード**はJVMの命令セットで、スタックベースで動作します。
- JVMの基本アーキテクチャは30年近く安定していますが、その実装は性能向上と新機能サポートのために絶えず進化し続けています。

JVMの仕組みを少しでも理解することで、Javaプログラミングの解像度が上がり、よ

もっとJVMを知るために

書籍:

- 『Javaパフォーマンス詳解』(Scott Oaks 著, オライリー・ジャパン)
- 『プログラマの教養としてのJava VM』(きしだなおき 著, SBクリエイティブ) - 分かりやすい入門書
- 『Java Performance: The Definitive Guide』 (Scott Oaks, O'Reilly Media) - 最新情報はこちら
- 『The Java Virtual Machine Specification, Java SE X Edition』 (Oracle) - 公式仕様書 (Xはバージョン。例: SE 24)

Webサイト & 記事:

- OpenJDK 公式サイト (各JEPの詳細、プロジェクト情報)
- きしださんのブログ/Qiita記事 (https://qiita.com/nowokayなど) - 最新Java情報

ご清聴ありがとうございました

質疑応答

発表者: kmizu ([@kmizu](https://x.com/kmizu))
Java/JVM Meetup - 2025/05/23

(予備) JVMの起動プロセス概略

1. **JVMの初期化:**
    - `java <MainClassName>` コマンド実行。
    - OSがJVMライブラリ (例: `jvm.dll`, `libjvm.so`) をロードし、JVMインスタンスを作成。
    - JVM自身の初期設定 (ヒープサイズ、GC選択など) を行う。
2. **メインクラスのロード:**
    - システムクラスローダーが、指定された `MainClassName` の `.class` ファイルを探してロード。
    - クラスの検証、準備、(必要なら)解決を行う。
3. **`main` メソッドの検索と実行:**
    - ロードされたメインクラスから `public static void main(String[] args)` メソッドを探す。

(予備) ガベージコレクタ (GC) の簡単な分類

- **シリアルGC (Serial GC):** `-XX:+UseSerialGC`
  - 単一スレッドでGCを実行。Stop-The-World (STW) が発生。
  - クライアントマシンや小規模なヒープ向け。
- **パラレルGC (Parallel GC / Throughput GC):** `-XX:+UseParallelGC`
  - 複数のスレッドでGCを実行 (主にYoung領域)。STWは発生するが、スループット重視。
  - Java 8までのデフォルト。
- **G1 GC (Garbage-First GC):** `-XX:+UseG1GC`
  - Java 9以降のデフォルト。ヒープを多数のリージョンに分割して管理。
  - 大きなヒープサイズで、STW時間を予測可能にすることを目指す。
- **ZGC:** `-XX:+UseZGC` (Java 11で実験的導入、Java 15で製品版)
- **Shenandoah GC:** `-XX:+UseShenandoahGC` (OpenJDKプロジェクト、Red Hatなど