Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления» (ИУ)
КАФЕДРА «Информационная безопасность» (ИУ8)

Реферат на тему "Байт код в Java Virtual Machine"

по дисциплине «Управление программными продуктами»

Вариант 12

Студент ИУ8-104 (Группа) Мильченко И. Д. (И. О. Фамилия) (Подпись, дата)

Преподаватель Карондеева Андрей Михайлович (И. О. Фамилия) (Подпись, дата)

Москва, 2025 г.

1 Введение

Виртуальная машина Java (JVM) — это программная реализация абстрактного процессора, гарантирующая независимость исполняемого кода от аппаратной платформы и операционной системы. Центральной идеей является преобразование исходной программы в промежуточный байт-код — компактный поток инструкций, описывающий алгоритм на уровне стека операндов. Такое представление проверяется на безопасность, а затем либо интерпретируется, либо динамически компилируется в нативные инструкции.

Знание внутренних процессов JVM полезно не только создателям языков и инструментов, но и прикладным разработчикам: понимание работы сборщика мусора помогает диагностировать утечки, знание модели памяти упрощает написание конкурентных программ, а умение читать байт-код облегчает аудит сторонних библиотек.

2 Архитектура JVM

Общая архитектура изображена на рис. 1. Она условно разбивается на три крупных подсистемы.

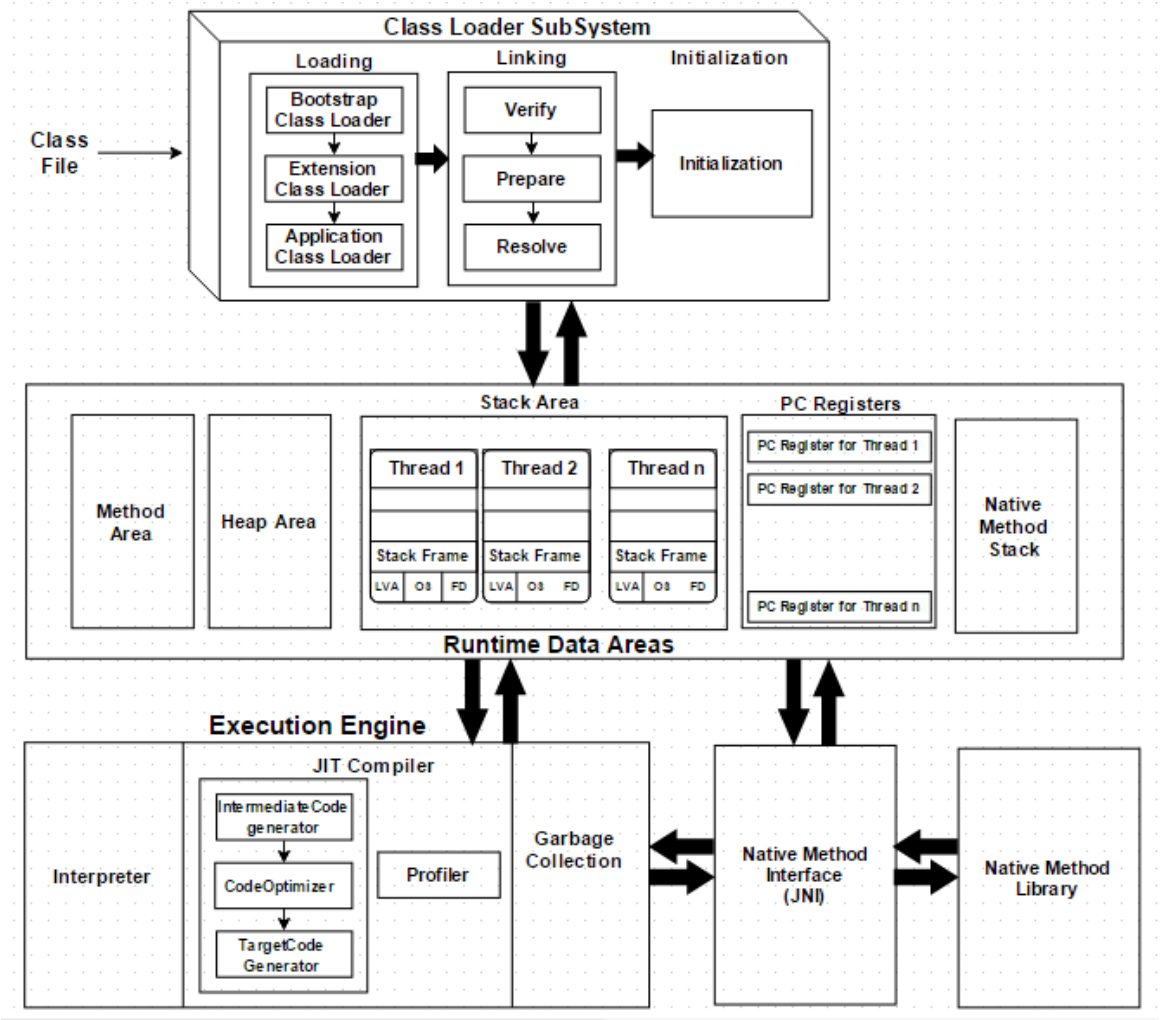

Рисунок 1 – Основные компоненты JVM

2.1 Загрузчик классов (Class Loader)

Процесс загрузки состоит из трёх фаз:

1) **Loading** — поиск и считывание байт-кода из JAR-архивов, модулей или сети.
2) **Linking** включает:
   - **Verification** — четырёхэтапная проверка корректности и безопасности, см. разд. 3;
   - **Preparation** — выделение памяти под статические поля и их инициализация значениями по умолчанию;

- **Resolution** — замена символических ссылок на прямые указатели.

3) **Initialization** — исполнение статических инициализаторов и блоков `<clinit>`.

Алгоритм делегирования *parent-first* предотвращает подмену ключевых классов (например, `java.lang.String`) и обеспечивает согласованность пространства имён.

2.2 Области данных времени выполнения

Каждый поток имеет собственные стековые структуры, а данные, разделяемые потоками, находятся в куче (*Heap*). Подробнее:

Heap — основное хранилище объектов. В HotSpot обычно разделяется на *Young Generation* (Eden + Survivor S0/S1) и *Old Generation*. Современные коллекторы (G1, ZGC, Shenandoah) используют более дробную разметку.

Method Area содержит метаданные классов, пул констант, JIT-компилированный код и профилировочную информацию (до Java 8 называлась Permanent Generation).

JVM Stack — набор кадров (frames). Каждый кадр хранит локальные переменные, стек операндов и ссылку на константный пул.

PC-register — указатель на текущую инструкцию в потоке.

Native Method Stack — стеки для вызовов JNI.

2.3 Исполнительный движок (Execution Engine)

Движок включает:

- Интерпретатор (Template Interpreter) — быстрый запуск, низкая оптимизация.
- C1-компилятор (Client) — выдаёт компактный, умеренно оптимизированный код.
- C2-компилятор (Server) — затратен при компиляции, но генерирует высокопроизводительный код.

HotSpot использует **tiered compilation**: метод начинает работу в интерпретаторе, затем переходит на C1, а при достаточной "горячести"— на C2. График переходов показан на рис. 2.

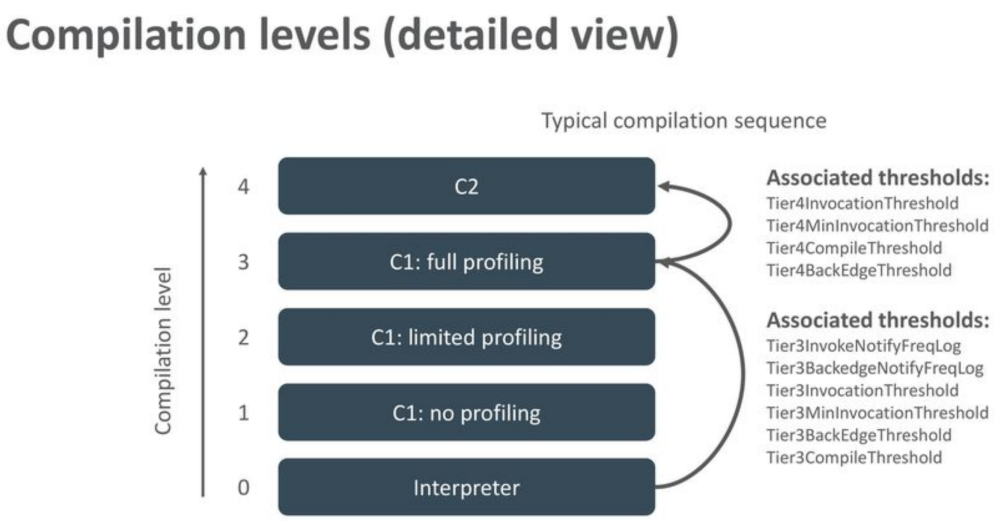

Рисунок 2 – Уровни JIT-компиляции в HotSpot

3 Верификация байт-кода

Перед исполнением класс проходит четыре уровня проверок:

A. **Проверка файла** — корректность заголовка, версии и контроль целостности потока.

B. **Проверка метаданных** — допустимы ли модификаторы, существуют ли супер-классы.

C. **Проверка байт-кода** — построение модели типизации стека и локалов, контроль баланса стека и исключений. Реализуется с помощью алгоритма DFA (data-flow analysis).

D. **Проверка загрузчика** — гарантирует, что разные классы одного имени загружены одним загрузчиком.

Проваливая любой этап, JVM выбрасывает `VerifyError`, предотвращая загрузку небезопасного кода.

4 Формат файла `.class`

4.1 Обзор структуры

Ниже приведена таблица с главными разделами:

Раздел	Назначение
Magic, Minor/Major	Идентификатор и версия формата
Constant pool	Таблица литералов и ссылок
Access flags	Модификаторы (public, final, enum ...)
This/Super	FQCN текущего и базового класса
Interfaces	Список реализованных интерфейсов
Fields	Описания полей
Methods	Сигнатуры и байт-код методов
Attributes	Расширения (аннотации, StackMapTable)

4.2 Константный пул: типы записей

Всего определено 15 тэгов (`CONSTANT_Utf8`, `CONSTANT_Methodref` ...). Начиная с Java 11 появился `CONSTANT_Dynamic`, позволяющий лениво вычислять константы под управлением bootstrap-метода.

4.3 Пример дизассемблирования

Ниже показан полный цикл: исходный Java-код, скомпилированный байт-код, а затем подробный разбор инструкций.

Листинг 1 – Класс Sum.java

```
public class Sum {
    public int sum(int n) {
        int acc = 0;
        for (int i = 0; i < n; i++) {
            acc += i;
        }
        return acc;
    }
}
```

Скомпилируйте файл командой `javac Sum.java`. Получится класс `Sum.class`.

Команда: `javap -c -v Sum.class`. Результат для метода `sum` — листинг 2.

Листинг 2 – Байт-код метода sum

```
0: iconst_0            // push int 0
1: istore_2            // local[2] = 0 (accumulator)
2: iconst_0
3: istore_3            // local[3] = 0 (index)
4: iload_3
5: iload_1
6: if_icmpge    22   // loop exit
9: iload_2
10: iload_3
11: iadd
12: istore_2           // acc += index
13: iinc       3, 1    // index++
16: goto       4
22: iload_2
23: ireturn
```

Подробный разбор инструкций

0: `iconst_0` Кладёт константу `0` на стек.

1: `istore_2` Снимает значение со стека и сохраняет в локальную переменную 2 (аккумулятор `acc`).

2: `iconst_0` Помещает `0` на стек для инициализации счётчика.

3: `istore_3` Записывает значение в локальную переменную 3 (счётчик `i`).

4: `iload_3` Загружает `i` на стек.

5: `iload_1` Загружает аргумент `n` на стек.

6: `if_icmpge 22` Сравнивает `i` и `n`; если `i >= n` — переход к метке 22 (выход из цикла).

9: `iload_2` Загружает текущую сумму `acc`.

10: `iload_3` Загружает текущий индекс `i`.

11: `iadd` Складывает верхние два элемента стека (`acc + i`).

12: `istore_2` Сохраняет результат обратно в `acc`.

13: `iinc 3, 1` Увеличивает локальную переменную 3 (`i`) на 1 без использования стека.

16: `goto 4` Переход к повторной проверке условия цикла.

22: `iload_2` Загружает итоговую сумму `acc` на стек.

23: `ireturn` Возвращает значение вершины стека вызывающему коду.

5 Набор инструкций и модель стека

Каждый байт-код читает или пишет стек операндов. Например, `iadd` снимает два int, складывает и кладёт результат. Благодаря такой модели JVM легко переносится на регистровые архитектуры: JIT может свободно распределять значения по физическим регистрам.

5.1 Классификация инструкций

1) **Константы и загрузка** (`ldc`, `bipush`).
2) **Арифметика и логика** (`ixor`, `lrem`).
3) **Контроль потока** (`lookupswitch`, `jsr`).
4) **Объектные операции** (`checkcast`, `putfield`).
5) **Синхронизация** (`synchronized`, реализовано парами `monitorenter/exit`).

6 Исполнение байт-кода

6.1 Интерпретация и JIT

При запуске новый метод помечается профилировочным счётчиком `InvocationCounter` = 0. С каждым вызовом он растёт. Достигнув порога `-XX:CompileThreshold` (по умолчанию 2000), метод попадает в очередь JIT. После компиляции JVM заменяет в call-site адрес интерпретируемой версии на адрес нативного кода.

6.2 Профилирование

HotSpot собирает:

- число вызовов, обратных переходов, исключений;
- вероятности ветвлений;
- наблюдаемые классы для виртуальных вызовов.

Эти данные хранятся в структуре `MDO` (MethodDataOop) и используются при повторной компиляции.

6.3 Оптимизации C2

Inlining — раскрытие мелких методов, уменьшает вызовы и раскрывает контекст для последующих оптимизаций.

Escape Analysis — выявляет объекты, не «уходящие» из метода; такие объекты могут быть размещены на стеке или устранены.

Loop Unrolling и Peeling — уменьшает количество проверок и повышает ILP (instruction-level parallelism).

Range Check Elimination — удаляет избыточные проверки массива.

7 Управление памятью и сборка мусора

7.1 Поколенческая теория

Большинство объектов «умирает молодыми», поэтому куча делится на поколения (рис. 3). Сборка молодого поколения выполняется чаще и быстрее.

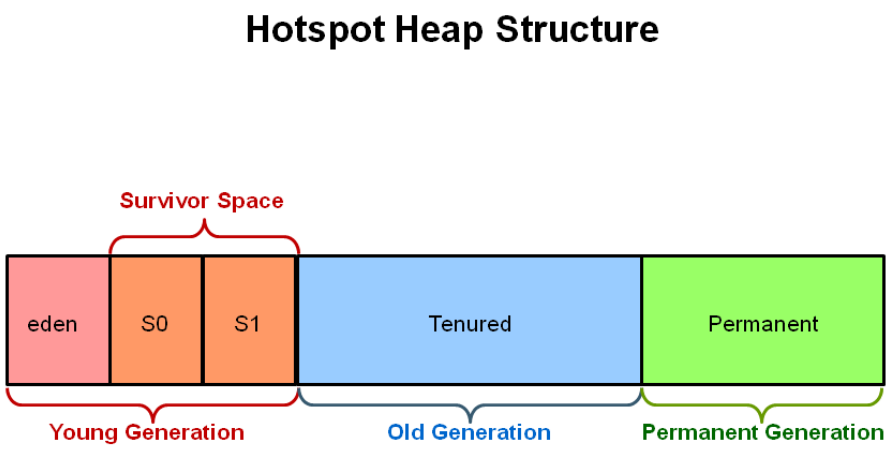

Рисунок 3 – Генерационный разрез кучи

На схеме показано стандартное логическое разбиение кучи HotSpot на поколения.

- *Eden* (розовый) — область, куда помещаются все вновь создаваемые объекты. При заполнении Eden запускается *минорная* (young) сборка.
- *Survivor S0/S1* (оранжевые) — две буферные области. Живые объекты из Eden копируются поочерёдно в один из них. После каждой минорной паузы роли S0 и S1 меняются.
- *Tenured (Old) Generation* (голубой) — здесь оказываются объекты, пережившие несколько минорных сборок. Для неё применяется *мажорная* (full) сборка.
- *Permanent/Metaspace* (зелёный) — хранит метаданные классов, пул констант, JIT-код и другие служебные структуры. Начиная с Java 8 эта область вынесена за пределы Java-кучи и называется Metaspace.

Главная идея генерационной гипотезы: большинство объектов «умирает» ещё до первой минорной сборки. Поэтому частые короткие паузы обрабатывают только Eden и Survivor-области, а редкие длительные — Old Generation. Такой подход значительно снижает среднюю латентность приложения без потери пропускной способности.

7.2 Обзор коллекторов

- **Serial GC** — однопоточный mark-copy, подходит для систем с 1 ЦП.
- **Parallel GC** — многопоточный, ориентирован на Throughput.
- **CMS** — concurrent mark-sweep, низкие паузы, но фрагментирует кучу.
- **G1** — разбивает кучу на регионы, предсказуемые паузы.
- **ZGC** — цветные указатели, паузы < 10 мс при терабайтных кучах.
- **Shenandoah** — алгоритм Region-based, полностью конкаррентный (паузы < 1 мс).

Сравнение средних пауз:

Коллектор	Java 21 default	Цель	Тип паузы
G1	Да	balanced	10–200 мс
ZGC		low-latency	< 10 мс
Shenandoah		ultra-low	< 1 мс

8 Динамические вызовы: `invokedynamic`

8.1 История и мотивация

До Java 7 JVM поддерживала четыре оп-кода вызова: `invokestatic`, `invokevirtual`, `invokespecial`, `invokeinterface`. Они обеспечивали статическое связывание. Для реализации динамических языков (JRuby, Jython) требовались хаки. В JSR-292 появился `invokedynamic`.

8.2 Bootstrap-метод

При первой встрече с инструкцией JVM вызывает bootstrap-метод, который возвращает *CallSite*. Далее вызовы осуществляются через кэшированную ссылку без дополнительных затрат.

8.3 Пример лямбды

Компилятор Java 8 преобразует лямбда-выражение в вызов `invokedynamic` с дескриптором, указывающим на метод `LambdaMetafactory`. Это позволило реализовать функциональный стиль без генерации анонимного класса.

9 Инструменты анализа и генерации байт-кода

1. **javap** — дизассемблер из JDK.
2. **JClassLib** — GUI-просмотрщик структуры `.class`.
3. **ASM** — низкоуровневое API для чтения/записи байт-кода.
4. **ByteBuddy** — DSL над ASM, декларативная генерация.
5. **BCEL** — библиотека Apache для статического анализа.
6. **JMH** — harness для микробенчмарков, измеряет влияние JIT.

10 Современные тенденции и будущие изменения

Project Loom — вводит лёгкие потоки (Fibers) и оп-коды `YIELD`, влияющие на планировщик.

Project Valhalla — inline-классы (value types) с дескриптором `QFoo;`. Это позволит хранить объекты без указателей и GC-заголовка.

Project Panama — FFI к нативному коду и SIMD-акселерация через Vector API.

GraalVM — JIT на Java, AOT-компиляция в образ Native Image, поддержка Truffle-языков.

Заключение

Развитие JVM сопровождается эволюцией формата байт-кода и JIT-технологий, сохраняя при этом обратную совместимость. Освоив внутренние механизмы загрузки, верификации и исполнения, разработчик получает рычаги для глубокого профилирования, оптимизации и даже создания собственных языков поверх JVM.