VM (Java Virtual Machine)

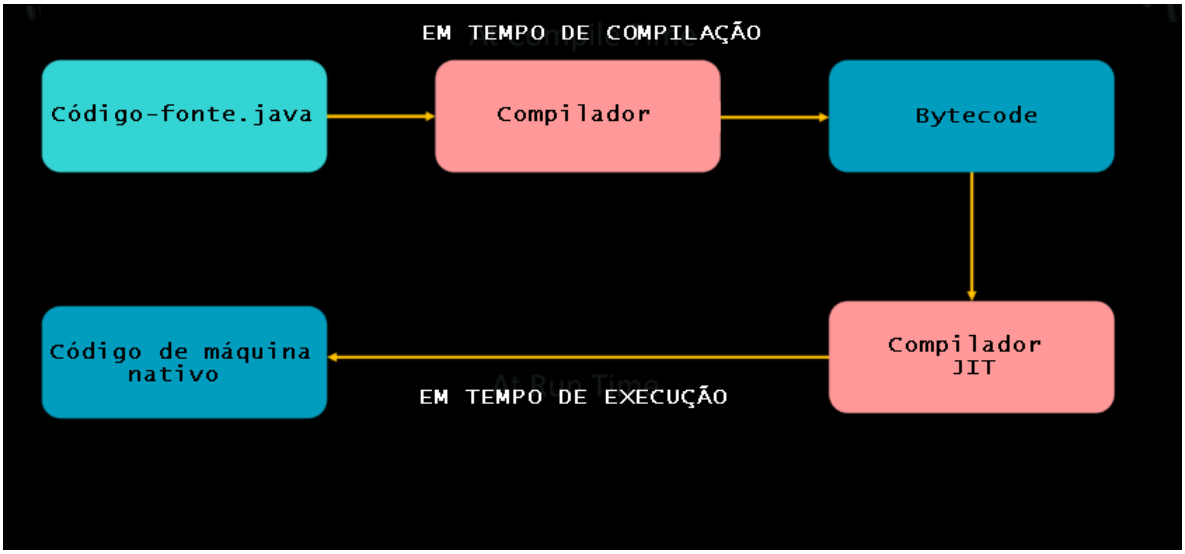
A JVM, ou Máquina Virtual Java, é uma peça fundamental para que os programas escritos em Java possam ser executados em diferentes sistemas operacionais sem a necessidade de reescrever o código. Aqui está como ela funciona, de forma simplificada:

1. **Código Java (Fonte)**: Você escreve seu código em Java (.java).
2. **Compilador Java**: Esse código é compilado pelo compilador Java (javac) para gerar um bytecode (.class), que é um formato intermediário.
3. **JVM**: A JVM executa esse bytecode. Ela é como um intermediário que traduz o bytecode em instruções específicas para o sistema operacional e hardware em que está rodando.

JIT Compiler (Just-In-Time Compiler)

Agora, vamos falar sobre o JIT compiler, que é uma parte da JVM. O JIT é uma técnica que melhora a performance dos programas Java. Aqui está como ele funciona:

1. **Execução Inicial**: Quando você roda um programa Java, a JVM começa interpretando o bytecode, linha por linha, o que pode ser um pouco lento.
2. **Identificação de Hot Spots**: Enquanto o programa está sendo executado, a JVM identifica partes do código que são executadas com frequência, chamadas de "hot spots".
3. **Compilação Just-In-Time**: Quando a JVM encontra um hot spot, o JIT compiler entra em ação. Ele compila essas partes frequentemente executadas do bytecode para código de máquina (código nativo) enquanto o programa ainda está rodando.
4. **Execução Rápida**: O código nativo é muito mais rápido de executar que o bytecode interpretado, então essas partes do programa agora rodam muito mais rápido.



Para que servem

Os *JIT compilers* servem para melhorar a performance dos programas Java. Ao compilar partes do código para código de máquina nativo durante a execução, eles permitem que o programa rode de forma mais eficiente e rápida.

Aqui estão alguns pontos chave sobre o JIT compiler:

- **Performance**: Aumenta a velocidade de execução do programa ao compilar dinamicamente os hot spots.
- **Otimizações**: Pode aplicar várias otimizações específicas durante a compilação para melhorar ainda mais a performance.
- **Adaptabilidade**: Como a compilação acontece durante a execução, o JIT pode otimizar o código com base no comportamento real do programa em tempo de execução.

Na realidade existem diferentes tipos de JIT compilers e cada um tem suas características e objetivos específicos. Vamos explorar isso um pouco mais.

Tipos de JIT Compilers na JVM

Na JVM, existem principalmente dois tipos de JIT compilers:

1. **C1 (Client Compiler)**
2. **C2 (Server Compiler)**

C1 (Client Compiler)

- **Foco**: Desempenho rápido de inicialização e compilação rápida.
- **Uso**: É usado em aplicações onde o tempo de inicialização rápida é crucial, como em aplicações desktop ou ambientes com recursos limitados.
- **Otimizações**: Realiza otimizações mais simples e rápidas para compilar o código rapidamente e permitir uma execução rápida logo de início.

C2 (Server Compiler)

- **Foco**: Alto desempenho de longo prazo.
- **Uso**: É usado em aplicações onde o desempenho de execução é mais importante que o tempo de inicialização, como em servidores e aplicações de longo prazo.
- **Otimizações**: Realiza otimizações mais complexas e intensivas em tempo para produzir um código muito mais eficiente, mas leva mais tempo para compilar.

Tiered Compilation

A JVM usa uma técnica chamada **Tiered Compilation** que combina os benefícios dos dois compiladores C1 e C2. Aqui está como funciona:

1. **Inicialização Rápida com C1**: Quando um programa Java começa a rodar, o compilador C1 entra em ação, compilando o bytecode rapidamente para que a aplicação possa iniciar rapidamente.
2. **Monitoramento de Hot Spots**: Durante a execução, a JVM monitoriza o comportamento do programa e identifica os hot spots (partes frequentemente executadas do código).
3. **Transição para C2**: Para esses hot spots, o compilador C2 entra em ação e recompila essas partes do código com otimizações mais complexas para maximizar o desempenho.

Níveis de Compilação (Tiers)

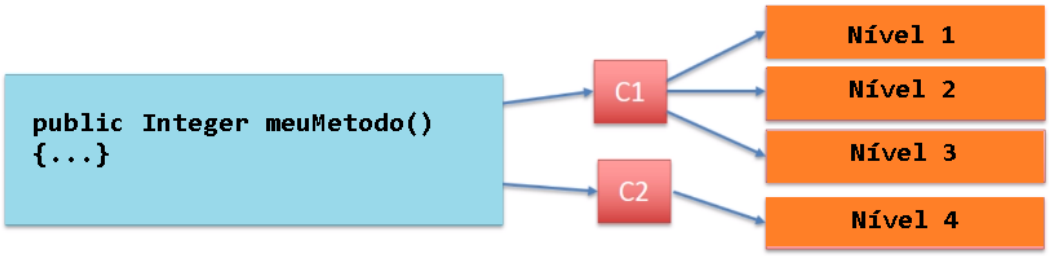
A JVM usa uma abordagem chamada **Tiered Compilation**, que consiste em diferentes níveis de compilação. Cada nível oferece um balanço diferente entre a velocidade de compilação e o grau de otimização. Aqui estão os níveis mais comuns:

1. **Interpretação**: O bytecode é interpretado diretamente pela JVM, sem compilação para código nativo. É o mais lento, mas não requer tempo de compilação.
2. **Nível 1 (C1)**: O bytecode é compilado rapidamente com otimizações básicas pelo compilador C1. Isso permite que o programa rode mais rápido que na interpretação pura.
3. **Nível 2 (C1)**: O compilador C1 aplica otimizações um pouco mais avançadas, ainda focadas na rapidez de compilação.
4. **Nível 3 (C1)**: O compilador C1 aplica otimizações mais agressivas, balanceando rapidez de compilação e eficiência do código gerado.
5. **Nível 4 (C2)**: O compilador C2 recompila os hot spots identificados com otimizações avançadas e intensivas, visando o máximo desempenho.

Como Funciona o Processo

1. **Interpretação Inicial**: Quando o programa é executado pela primeira vez, o bytecode é interpretado pela JVM. Isso permite que o programa comece a rodar imediatamente, mas a execução é lenta.
2. **Primeira Compilação (C1)**: À medida que o código é executado repetidamente, a JVM começa a compilar partes frequentemente executadas (hot spots) usando o compilador C1. Inicialmente, o C1 faz isso com otimizações básicas (Nível 1).
3. **Otimizações Progressivas (C1)**: Conforme a execução continua e mais dados de desempenho são coletados, o compilador C1 pode aplicar otimizações mais avançadas (Nível 2 e Nível 3).
4. **Compilação Avançada (C2)**: Para hot spots que são identificados como extremamente críticos para o desempenho, o compilador C2 recompila esses trechos de código com

otimizações muito mais complexas e detalhadas (Nível 4), resultando em código nativo altamente eficiente.



Vantagens dos Diferentes Níveis de Compilação

- **Rapidez Inicial**: A interpretação e os níveis mais baixos de compilação (C1) permitem que o programa comece a rodar rapidamente.
- **Desempenho Progressivo**: À medida que o programa é executado, ele se torna progressivamente mais rápido devido às otimizações aplicadas.
- **Máximo Desempenho**: O compilador C2 garante que as partes mais críticas do código rodem com a máxima eficiência.

Vamos usar um exemplo de código, para calcular uma determinada quantidade de número primos, para mostrar como o JIT otimiza a execução do código, dependendo do número de vezes que ele é chamado.

```
public class Main {

    public static void main(String[] args) throws Exception {
        PrimeNumbers primeNumbers = new PrimeNumbers();
        Integer max = Integer.parseInt(args[0]);
        primeNumbers.generateNumbers(max);
    }

}
```

Este código simplesmente inicia a aplicação, cria uma instância da classe `PrimeNumbers`, recebe um argumento da linha de comando (o máximo de números a serem gerados), e chama o método `generateNumbers` para gerar números primos até o valor máximo especificado.

```
package main;

import java.util.ArrayList;
import java.util.List;

public class PrimeNumbers {

    private List<Integer> primes;

    private Boolean isPrime(Integer testNumber) {
        for (int i = 2; i < testNumber; i++) {
            if (testNumber % i == 0) {
                return false;
            }
        }
        return true;
    }

    private Integer getNextPrimeAbove(Integer previous) {
        Integer testNumber = previous + 1;
        while (!this.isPrime(testNumber)) {
            testNumber++;
        }
        return testNumber;
    }

    public void generateNumbers(Integer max) {
        this.primes = new ArrayList<Integer>();
        this.primes.add(2);

        Integer next = 2;
        while (this.primes.size() <= max) {
            next = this.getNextPrimeAbove(next);
            this.primes.add(next);
        }
        // System.out.println(this.primes); // imprime os números primos
    }

}
```

Este código define a classe `PrimeNumbers`, que gera números primos até um valor máximo especificado. Ele utiliza um método `isPrime` para verificar se um número é primo e um método `getNextPrimeAbove` para obter o próximo número primo maior que um dado valor. A lista de números primos gerados é armazenada no campo `primes`.

Para rodar o programa Java utilizaremos o parâmetro `-XX:+PrintCompilation`, é importante entender o que essa opção faz e como ela influencia a execução do programa.

O que `-XX:+PrintCompilation` faz?

`-XX:+PrintCompilation` é uma opção de linha de comando que pode ser passada para a JVM (Java Virtual Machine) ao executar um programa Java. Ela ativa a impressão de informações detalhadas sobre as compilações Just-In-Time (JIT) realizadas pela JVM durante a execução do programa. Aqui estão os principais pontos sobre o que essa opção faz:

1. **Registro de Compilações**: Quando ativada, a JVM imprime no console informações sobre cada método compilado em tempo de execução.
2. **Detalhes de Compilação**: As informações, entre outras, incluem o número de milissegundos desde que a máquina virtual foi iniciada, além da ordem na qual o método ou bloco de código foi compilado. Os número não precisam aparecer em ordem. Isso significa apenas que algumas partes demoraram mais para compilar do que outras. Isso pode ser devido a problemas de *multithreading*, complexidade ou comprimento do código sendo compilado.
3. **Utilidade para Otimização**: É útil para desenvolvedores e administradores de sistema entenderem como a JVM está otimizando o código em tempo real. Isso pode ajudar na identificação de métodos frequentemente compilados, otimizações aplicadas e até mesmo problemas de desempenho relacionados à compilação JIT.

Como utilizar `-XX:+PrintCompilation` com seu programa:

Para utilizar `-XX:+PrintCompilation` com seu programa, você precisa executar o Java com essa opção junto com o caminho da classe principal (`Main` no seu caso) e quaisquer argumentos necessários. Por exemplo:

```
java -XX:+PrintCompilation Main 10
```

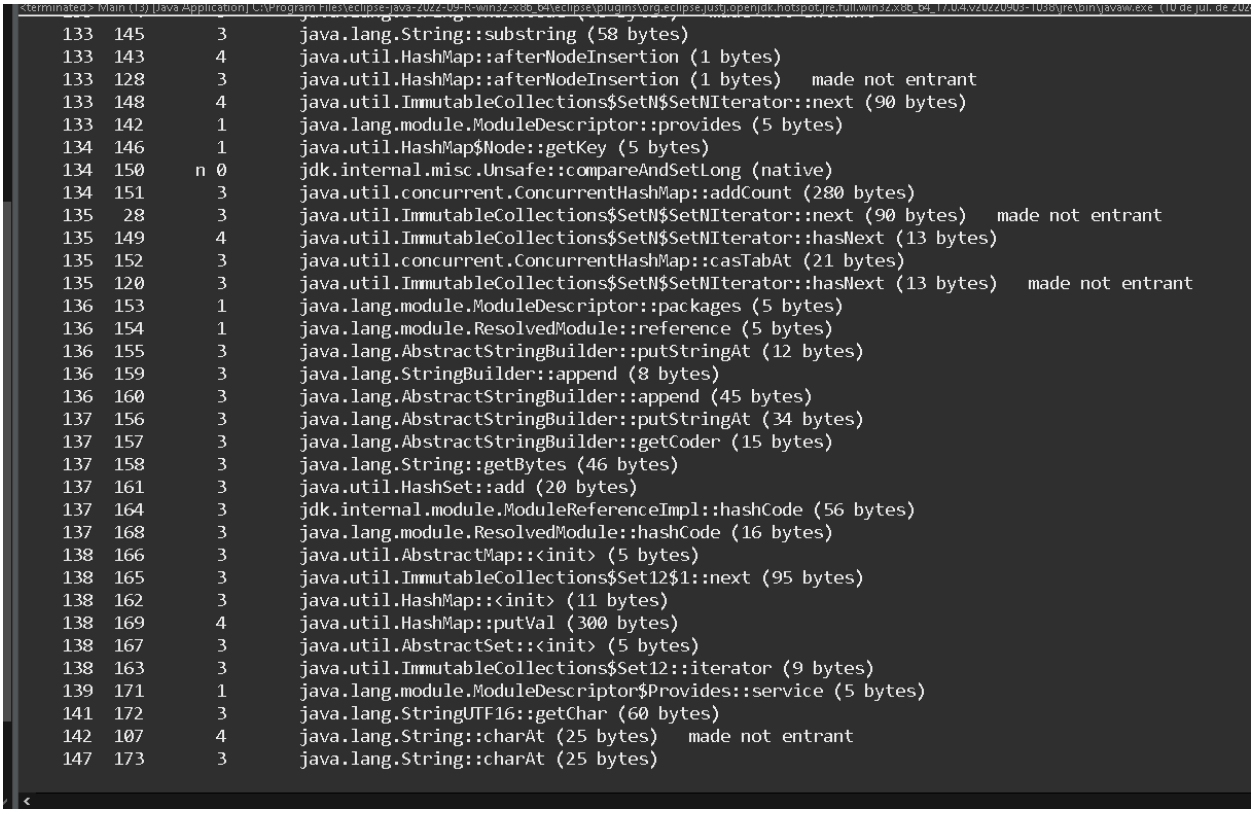
Neste comando:

- `java`: Comando para executar a JVM.
- `-XX:+PrintCompilation`: Ativa a impressão de informações de compilação JIT.
- `Main`: Nome da classe principal que contém o método `main`.
- `10`: Exemplo de argumento que você pode passar para seu programa, como o valor máximo de números primos a serem gerados.

Resultado esperado:

Ao executar seu programa com `-XX:+PrintCompilation`, o console imprimirá linhas de saída detalhando cada método que é compilado pela JVM.

Vamos executar o programa gerando apenas 10 número primos, abaixo vemos as últimas linhas apresentadas no console:



Temos algumas colunas diferentes, na maioria em branco, mas temos uma que tem um ***n*** significa método nativo, um método nativo em Java é aquele cuja implementação é escrita em uma linguagem de programação de nível mais baixo, como C ou C++. Em outras palavras, o código do método nativo não é escrito em Java puro, mas sim em uma linguagem que pode ser compilada diretamente para código de máquina executável pelo sistema operacional.

Caso aparecesse um ***s*** significaria que é um método sincronizado.

Um ponto de exclamação ***!*** significaria que há algum tratamento de exceção acontecendo e um símbolo de ***%*** significaria que o código foi compilado nativamente e está agora sendo executado em uma parte especial da memória chamada cache de código (*code cache*).

Quando o código possui métodos que não são executados com frequência, o JIT (compilador just-in-time) não aciona o C2 no nível quatro imediatamente. O JIT pode optar por não compilar esses métodos a níveis mais altos de otimização, como o C2, se eles não forem considerados críticos o suficiente para beneficiar significativamente da otimização mais agressiva. Em vez disso, esses

métodos podem permanecer em níveis de compilação mais baixos ou até mesmo serem interpretados diretamente pelo JVM, dependendo do perfil de execução e das políticas de otimização configuradas.

Vamos agora re-executar o código, mas com a quantidade de números primos igual a 5000

```
java -XX:+PrintCompilation Main 5000
```

O resultado em console agora é:

```
   144  160       3       jdk.internal.module.ModuleReferenceImpl::hashCode (56 bytes)
   144  164       3       java.lang.module.ResolvedModule::hashCode (16 bytes)
   145  165       3       java.util.HashMap$HashIterator::<init> (79 bytes)
   145  167       4       java.util.HashMap::putVal (300 bytes)
   145  162       3       java.util.AbstractMap::<init> (5 bytes)
   145  163       3       java.util.AbstractSet::<init> (5 bytes)
   145  161       3       java.util.ImmutableCollections$Set12$1::next (95 bytes)
   145  166       3       java.util.HashMap::<init> (11 bytes)
   146  145       1       java.util.HashMap$Node::getKey (5 bytes)
   146  169       1       java.lang.module.ModuleDescriptor$Provides::service (5 bytes)
   146  170       3       java.util.ImmutableCollections$Set12::iterator (9 bytes)
   146  171       3       java.util.ImmutableCollections$Set12$1::<init> (32 bytes)
   148  172       3       java.lang.StringUTF16::getChar (60 bytes)
   148   89       4       java.lang.String::charAt (25 bytes)   made not entrant
   149  173       3       java.lang.String::charAt (25 bytes)
   153  174       3       java.lang.AbstractStringBuilder::<init> (39 bytes)
   154  175       1       java.lang.Boolean::booleanValue (5 bytes)
   154  176       3       main.PrimeNumbers::isPrime (35 bytes)
   154  177       3       java.lang.Integer::valueOf (32 bytes)
   154  178       3       java.lang.Number::<init> (5 bytes)
   155  179       3       java.lang.Integer::<init> (10 bytes)
   155  180       1       java.util.ArrayList::size (5 bytes)
   155  123       3       java.util.HashMap::putVal (300 bytes)   made not entrant
   155  181 %     4       main.PrimeNumbers::isPrime @ 5 (35 bytes)
   155  182       3       java.util.ArrayList::add (25 bytes)
   156  183       3       java.util.ArrayList::add (23 bytes)
   156  184       3       main.PrimeNumbers::getNextPrimeAbove (36 bytes)
   158  168       4       java.util.HashMap::newNode (13 bytes)
   158  184       3       main.PrimeNumbers::getNextPrimeAbove (36 bytes)   made not entrant
   158  185       3       main.PrimeNumbers::getNextPrimeAbove (36 bytes)
   159  119       3       java.util.HashMap::newNode (13 bytes)   made not entrant
   159  186       4       main.PrimeNumbers::isPrime (35 bytes)
   161  176       3       main.PrimeNumbers::isPrime (35 bytes)   made not entrant
   218  187       4       main.PrimeNumbers::getNextPrimeAbove (36 bytes)
   224  185       3       main.PrimeNumbers::getNextPrimeAbove (36 bytes)   made not entrant
```

Perceba que existe agora referência aos métodos da classe `PrimeNumbers` como `getNextPrimeAbove` e `isPrime`. Esses são métodos que existem em nosso código e também temos o símbolo de porcentagem aparecendo para `isPrime` , significando que o método alcançou o nível mais alto de otimização possível sendo colocado no *code cache.* Os número entre 0 e 4 informam que tipo de compilação ocorreu, um zero significaria nenhuma compilação, o código foi apenas interpretado. Os números de 1 a 4 significam que ocorreu um nível progressivamente mais profundo de compilação.

O método `isPrime` portanto foi compilado no nível mais alto possível de compilação, no nível quatro, e foi colocado na cache de código. Podemos ver que `getNextPrimeAbove` também foi compilado até o nível quatro, embora não tenha sido colocado na cache de código.

A JVM decide qual nível de compilação aplicar a um bloco de código específico usando como base a frequência que ele é executado e em quão complexo ou demorado ele é.

À medida que o programa é executado, o perfil de execução pode mudar e também qual compilador está operando C1 ou C2. A JVM pode detectar mudanças no comportamento do programa e realizar recompilações just-in-time adicionais para se adaptar a essas mudanças. Isso é conhecido como reotimização e permite que o código seja otimizado com base nas condições atuais de execução.

Então, para qualquer método que tenha um número de 1, 2, 3, o código foi compilado usando o compilador C1, e quanto maior o número, mais otimizado o código foi.

Se o código foi executado várias vezes, como no nosso exemplo ao rodar o programa para calcular 5000 números primos, então alcançamos o nível 4. Nesse ponto, o compilador C2 foi utilizado e o código pode estar incluso na cache de código, representando o máximo de otimização possível.

Nota: Existe uma opção de compilação que gera um arquivo de log mais detalhado e você pode observar a mudança do nível de compilação a medida que o código é executado (C1 nivel 1,2,3 C2 nível 4) A opção é:

```
java -XX:+UnlockDiagnosticVMOptions -XX:+LogCompilation Main 5000
```

1