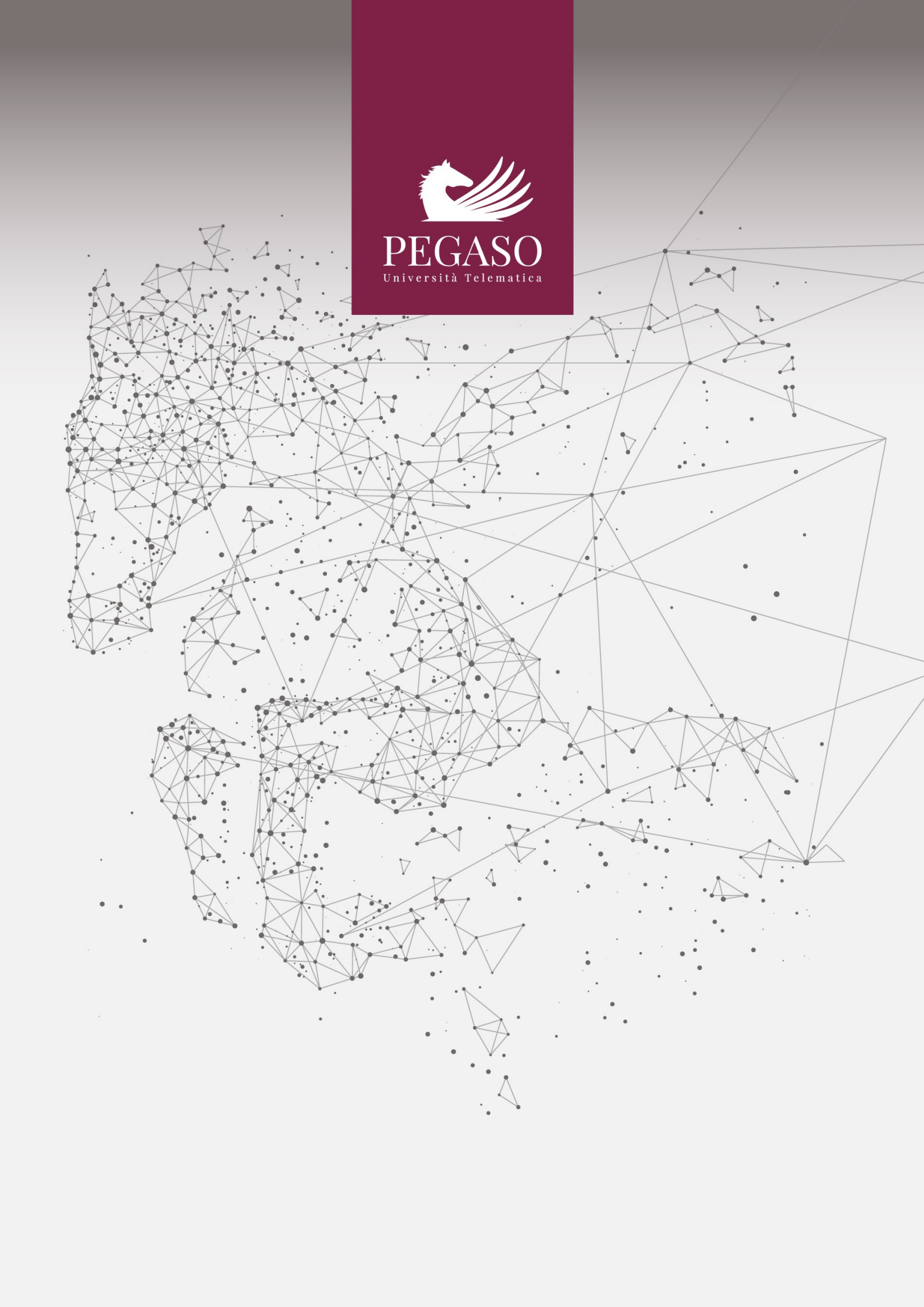
PEGASO
Università Telematica

Indice

1. Java: un linguaggio compilato e interpretato

Il compilatore e l'interprete rappresentano due approcci differenti all'esecuzione di programmi scritti in un linguaggio di programmazione o di scripting. Entrambi hanno l'obiettivo di trasformare il codice sorgente, ovvero il testo scritto dallo sviluppatore, in istruzioni comprensibili ed eseguibili dal computer. Tuttavia, lo fanno seguendo modalità diverse, con implicazioni importanti sul piano delle prestazioni, della portabilità e del flusso di sviluppo.

Un compilatore è un programma che analizza l'intero codice sorgente e lo traduce completamente in un altro linguaggio, solitamente in codice oggetto o codice macchina. Questo processo genera un file binario eseguibile che può essere lanciato direttamente dal sistema operativo senza che sia necessario il codice sorgente originale. Il vantaggio principale di questa strategia è che, una volta compilato, il programma è in genere molto più veloce da eseguire perché il computer può processarlo direttamente senza ulteriori trasformazioni. Linguaggi come C e C++ seguono questo approccio: il codice viene scritto, compilato e poi eseguito autonomamente, rendendo questi linguaggi ideali per applicazioni ad alte prestazioni o che devono interagire in modo efficiente con l'hardware.

l contrario, un interprete è un programma che legge e analizza il codice sorgente riga per riga, traducendolo ed eseguendolo in tempo reale, senza produrre un file binario intermedio. Questo significa che ogni volta che si vuole eseguire il programma, l'interprete ha bisogno del codice sorgente originale. I linguaggi interpretati offrono una maggiore flessibilità, permettendo una più semplice sperimentazione, debugging e modifica del codice, ma in genere presentano una minore efficienza in fase di esecuzione. Esempi tipici di linguaggi interpretati includono Python, Perl e Matlab, che sono molto apprezzati in ambiti come l'analisi dei dati, la prototipazione rapida e la ricerca scientifica.

Ebbene la distinzione tra compilatore e interprete sia storicamente netta, in realtà i linguaggi moderni tendono spesso a combinare le due strategie per ottenere un buon compromesso tra prestazioni e flessibilità. Java, ad esempio, adotta un approccio ibrido. Il codice Java viene inizialmente compilato in bytecode, una forma intermedia ottimizzata, mediante il compilatore javac. Questo bytecode non è direttamente eseguibile dalla macchina, ma è progettato per essere interpretato da una componente chiamata Java Virtual Machine (JVM). La JVM si occupa di eseguire il bytecode in modo portabile, garantendo che il programma Java possa funzionare su qualsiasi sistema operativo dotato di una JVM compatibile.

A questo si aggiunge un ulteriore livello di ottimizzazione, chiamato JIT (Just-In-Time) compiler, che entra in azione durante l'esecuzione del programma. Il JIT prende parti del bytecode che vengono utilizzate frequentemente e le compila dinamicamente in codice macchina nativo, migliorando così le prestazioni

complessive. In questo modo, Java combina la portabilità tipica degli interpreti con l'efficienza dei compilatori, offrendo una soluzione equilibrata per lo sviluppo di applicazioni moderne.

Per ricapitolare, l'architettura di Java combina il processo di compilazione e interpretazione:

- Il codice scritto in Java viene convertito in bytecode dal compilatore Java
- I byte code vengono poi convertiti in codice macchina dalla JVM
- Il codice macchina viene eseguito direttamente dalla macchina.

Vediamo questo concetto più in dettaglio con il Java Programming Model (vedi Fig. 1).

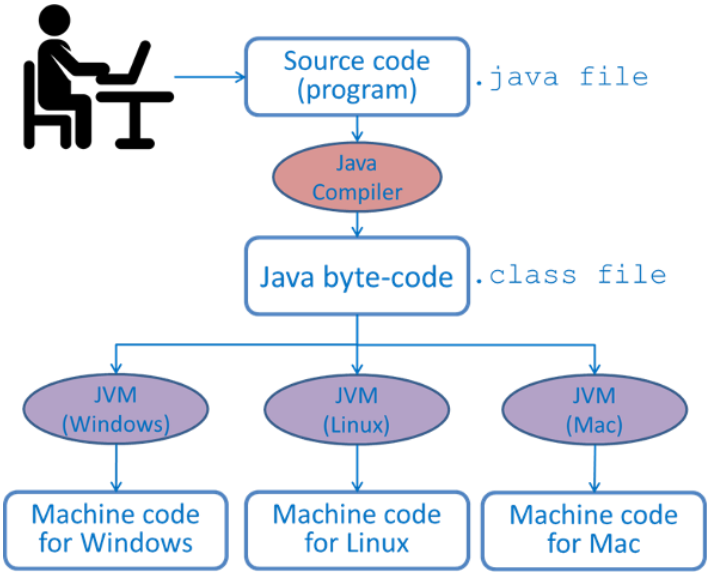

Fig. 1: *Java Programming Model*

Dopo aver scritto il programma Java, lo sviluppatore deve per prima cosa compilarlo ottenendo così non direttamente un file eseguibile (ovvero la traduzione in un linguaggio macchina del file sorgente scritto in Java) ma un file che contiene la traduzione del listato in linguaggio bytecode. Una volta ottenuto il bytecode, la JVM interpreterà il bytecode ed il programma andrà quindi in esecuzione. Quindi se una piattaforma qualsiasi possiede una JVM ciò sarà sufficiente a renderla esecutrice del bytecode.

Si fa notare che:

- Il compilatore Java è lo stesso per ogni sistema operativo/piattaforma. Mentre l'interprete ovvero la Java Virtual Machine cambia a seconda della piattaforma.
- Il file sorgente deve essere salvato con suffisso .java mentre il file risultato della compilazione avrà lo stesso nome del file sorgente ma con suffisso .class e conterrà il byte-code.

Si potrebbe affermare che il linguaggio macchina sta ad un computer come il bytecode sta ad una JVM. Oltre a permettere l'indipendenza dalla piattaforma la JVM garantisce a java anche (i) la capacità di supportare fra gli altri (i) il multi-threading ovvero capacità di mandare in esecuzione più processi in

maniera parallela, (ii) meccanismi di sicurezza molto potenti, (iii) la supervisione del codice da parte del Garbage Collector.

Per utilizzare il linguaggio Java dalla riga di comando, è fondamentale conoscere alcuni comandi di base messi a disposizione dal JDK (Java Development Kit). Il primo passo è scrivere il codice sorgente in un file con estensione .java. Una volta scritto, il file può essere compilato con il comando javac. Ad esempio, se si ha un file chiamato Main.java, è sufficiente eseguire:

```
javac Main.java
```

Questo comando compila il sorgente e genera un file Main.class contenente i bytecode Java. Una volta compilato, è possibile eseguire il programma tramite il comando java, specificando il nome della classe principale (senza estensione .class):

```
java Main
```

Un altro strumento molto utile è il comando jar, che consente di creare archivi Java (.jar) contenenti più classi e risorse. Questi archivi possono essere eseguiti direttamente se contengono un file MANIFEST.MF con l'indicazione della classe principale. Ad esempio, per creare un file eseguibile mioProgramma.jar, si può usare:

```
jar cfe mioProgramma.jar Main Main.class
```

dove Main è la classe con il metodo main. Successivamente, il programma può essere eseguito con:

```
java -jar mioProgramma.jar
```

Questi comandi sono la base per la compilazione, esecuzione e distribuzione di applicazioni Java da terminale, e rappresentano strumenti essenziali nello sviluppo professionale con Java.

2. JDK, JVM, JRE

Il mondo della programmazione Java si fonda su tre componenti fondamentali: JVM (Java Virtual Machine), JRE (Java Runtime Environment) e JDK (Java Development Kit). Comprendere a fondo la loro funzione, interazione e contesto d'uso è essenziale per ogni sviluppatore Java, sia principiante che professionista.

La Java Virtual Machine (JVM) è il cuore dell'architettura Java. Si tratta di una macchina virtuale che esegue il bytecode generato dalla compilazione del codice sorgente Java. Il ruolo della JVM è quello di astrarre l'esecuzione del programma dall'hardware e dal sistema operativo sottostante, garantendo la tanto nota portabilità del linguaggio Java: lo stesso programma può essere eseguito su sistemi Windows, macOS, Linux o qualsiasi altro ambiente dotato di una JVM compatibile. La JVM si occupa anche di molte operazioni cruciali come la gestione della memoria (attraverso il garbage collector), la verifica del bytecode per garantire la sicurezza, e l'ottimizzazione in fase di esecuzione grazie al compilatore JIT (Just-In-Time).

La JRE (Java Runtime Environment) rappresenta l'ambiente necessario per eseguire applicazioni Java. Include la JVM, oltre a tutte le librerie di classi (API) e risorse richieste per il corretto funzionamento del programma. La JRE non contiene strumenti di sviluppo, quindi non consente di compilare o creare applicazioni: è destinata principalmente agli utenti finali o agli ambienti di produzione in cui i programmi Java devono essere solo eseguiti, non sviluppati.

Il JDK (Java Development Kit) è il pacchetto completo per lo sviluppo in Java. Include tutto ciò che si trova nella JRE (e quindi anche la JVM), ma fornisce in aggiunta strumenti indispensabili per lo sviluppatore: il compilatore javac, il debugger, il monitor di prestazioni, l'interprete java, il documentatore javadoc, il packager jar, e molti altri tool. Il JDK è disponibile in diverse edizioni, come la Standard Edition (SE), la Enterprise Edition (EE) per applicazioni aziendali distribuite, e la Micro Edition (ME) per dispositivi embedded e mobili.

In scenari di sviluppo, il JDK viene installato localmente sull'ambiente di lavoro dello sviluppatore. Oggi è comune installare più versioni del JDK sullo stesso sistema, selezionando la versione da utilizzare tramite variabili di ambiente come JAVA_HOME. In contesti server o cloud, dove è richiesta solo l'esecuzione di applicazioni, si preferisce installare la JRE per ridurre la dimensione dell'ambiente runtime e contenere il memory footprint, ovvero l'impronta che il software lascia sulla memoria del sistema. In alternativa, con l'introduzione del modulo jlink (a partire da Java 9), è possibile creare runtime personalizzati che contengano solo le componenti strettamente necessarie all'applicazione, migliorando ulteriormente l'efficienza e la sicurezza.

Il linguaggio Java e i suoi strumenti hanno subito un'evoluzione significativa dalla loro prima versione rilasciata nel 1996 da Sun Microsystems. Dopo l'acquisizione di Sun da parte di Oracle nel 2010, Java ha adottato un ciclo di rilascio semestrale, con versioni LTS (Long-Term Support) ogni tre anni, come Java 8, Java 11, Java 17 e più recentemente Java 21. Ogni nuova versione del JDK introduce ottimizzazioni prestazionali, nuove API, e miglioramenti nella JVM, come la riduzione della latenza del garbage collector (es. ZGC, Shenandoah), una modularizzazione più granulare del JDK, e miglioramenti al linguaggio come var, records, pattern matching, ecc.

Uno dei principali vantaggi dell'uso della JVM è la gestione automatica della memoria, ma questo ha anche un costo in termini di consumo. La JVM ha una certa impronta di memoria iniziale che può variare a seconda delle impostazioni. Per applicazioni leggere o ambienti con risorse limitate, è fondamentale saper configurare i parametri della JVM, come la dimensione della heap (-Xmx, -Xms), l'uso di garbage collector alternativi (-XX:+UseG1GC, -XX:+UseZGC) e il monitoraggio dell'utilizzo della memoria in fase di esecuzione.

3. Funzionamento

La Fig. 2 mostra l'architettura generale di Java.

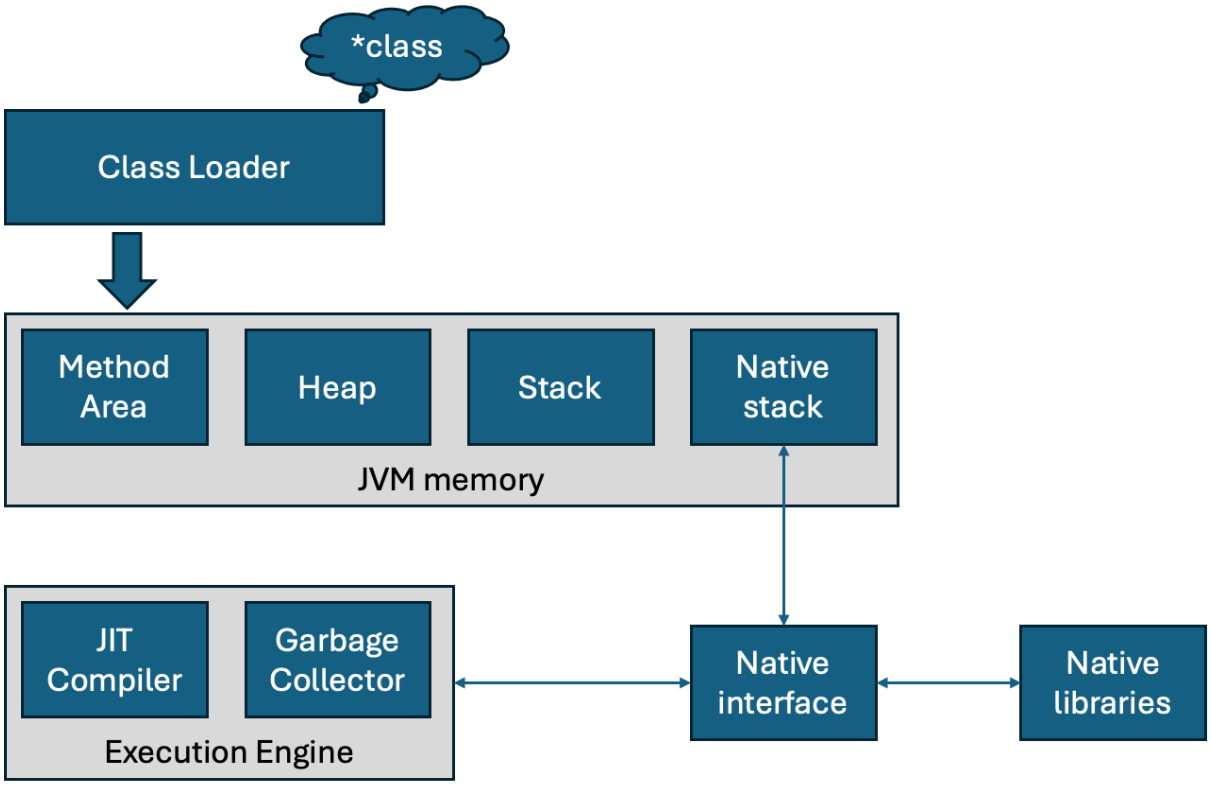

***Fig. 2** - Architettura Java*

In dettaglio:

- Class Loader: Il class loader è un sottosistema della JVM. Viene utilizzato per caricare i file di classe. Ogni volta che eseguiamo un programma Java, il class loader lo carica per primo.
- Class Method Area: È una delle aree di memoria della JVM, in cui vengono memorizzati i dati delle classi. Variabili statiche, blocchi statici, metodi statici e metodi di istanza sono memorizzati in quest'area.
- Heap: L'heap viene creato quando la JVM si avvia. Può aumentare o diminuire di dimensioni durante l'esecuzione dell'applicazione.
- Stack: Lo stack della JVM è anche noto come thread stack. È un'area di dati nella memoria della JVM creata per un singolo thread di esecuzione. Lo stack della JVM di un thread viene utilizzato dal thread per memorizzare vari elementi, cioè: variabili locali, risultati parziali e dati per la chiamata e il ritorno dei metodi.

- Native Stack: Include tutti i metodi nativi utilizzati nella tua applicazione.

Execution Engine:

- JIT Compiler: Il compilatore Just-In-Time (JIT) è una parte dell'ambiente di esecuzione. Aiuta a migliorare le prestazioni delle applicazioni Java compilando i bytecode in codice macchina durante l'esecuzione. Il compilatore JIT è abilitato di default. Quando un metodo viene compilato, la JVM richiama direttamente il codice compilato di quel metodo. Il JIT compila il bytecode del metodo in codice macchina "just in time" per l'esecuzione.
- Garbage Collector: Come suggerisce il nome, il Garbage Collector si occupa di raccogliere il materiale inutilizzato. Nella JVM, questo lavoro viene svolto dalla raccolta dei rifiuti (garbage collection). Tiene traccia di ogni oggetto presente nello spazio heap della JVM e rimuove quelli non più necessari.

Il Garbage Collector (GC) in Java è una componente fondamentale della Java Virtual Machine (JVM), responsabile della gestione automatica della memoria. Il suo scopo è liberare la memoria occupata da oggetti che non sono più utilizzati, evitando così problemi come il consumo eccessivo di memoria o i memory leak. Grazie al GC, gli sviluppatori non devono preoccuparsi di allocare o deallocare la memoria manualmente, a differenza di quanto avviene in linguaggi come C o C++.

Il processo di garbage collection in Java si basa su diverse strategie, ma tutte si fondano sul principio di identificare gli oggetti non più raggiungibili. Un oggetto è considerato "raggiungibile" se è accessibile direttamente o indirettamente da un root set (ad esempio, le variabili statiche, le variabili locali nei frame dello stack, o i registri della CPU). Quando un oggetto non è più raggiungibile, il GC lo considera "garbage", ovvero spazzatura da rimuovere.

Il meccanismo di base si articola in due fasi principali:

- Mark (Marcatura): in questa fase, il GC attraversa l'intero grafo degli oggetti partendo dalle root e marca tutti gli oggetti ancora raggiungibili.
- Sweep (Pulizia): successivamente, il GC libera tutta la memoria occupata dagli oggetti che non sono stati marcati nella fase precedente.

Questo approccio è semplice, ma può essere inefficiente in applicazioni di grandi dimensioni o in ambienti real-time, dove le pause causate dal GC (stop-the-world) devono essere minimizzate.

Per ottimizzare il comportamento del GC, la JVM supporta diversi algoritmi di raccolta, ognuno con caratteristiche adatte a differenti contesti:

- Serial GC: utilizza un singolo thread per la raccolta. È semplice e adatto per applicazioni con un singolo core o con bassa quantità di memoria.

- Parallel GC (o Throughput Collector): utilizza più thread per la raccolta, migliorando le performance complessive. È il garbage collector predefinito nelle versioni più vecchie di Java (fino a Java 8).
- CMS (Concurrent Mark-Sweep): esegue la marcatura in modo concorrente con il programma, riducendo i tempi di pausa. È stato deprecato in Java 9 e rimosso in Java 14.
- G1 GC (Garbage First): divide l'heap in regioni e cerca di raccogliere prima quelle con più oggetti "garbage". È progettato per offrire tempi di pausa prevedibili e gestire heap di grandi dimensioni. Dal JDK 9, è diventato il GC predefinito.
- ZGC (Z Garbage Collector): è un garbage collector a bassa latenza in grado di gestire heap molto grandi (anche superiori ai 16 TB) con pause inferiori a 10 ms, indipendentemente dalla dimensione dell'heap. È stato introdotto in Java 11.
- Shenandoah GC: simile a ZGC, è progettato per basse latenze e garbage collection concorrente. È disponibile a partire da Java 12.

La maggior parte degli algoritmi moderni implementa il concetto di generational GC, che sfrutta l'osservazione empirica che gli oggetti più giovani tendono a vivere meno. L'heap è diviso in tre aree:

- Young Generation: dove vengono allocati nuovi oggetti. Quando si riempie, avviene una Minor GC, che è molto veloce.
- Old Generation (o Tenured): contiene oggetti sopravvissuti a diverse minor GC. Qui avviene la Major GC o Full GC, che è più costosa in termini di tempo.
- Permanent Generation (o Metaspace): usata fino a Java 7 per contenere le classi caricate e i metadati. Dalla versione 8 in poi è stata sostituita dal Metaspace, che usa la memoria nativa.

Gli sviluppatori possono configurare il comportamento del GC tramite opzioni JVM, ad esempio:

```
-XX:+UseG1GC
-XX:+UseZGC
-Xms512m -Xmx2048m
```

Per analizzare le performance della garbage collection, Java fornisce strumenti come JVisualVM, JConsole, o il comando jstat. Esistono anche strumenti avanzati come Java Flight Recorder e GC logs analizzabili con GCEasy o Garbagecat.