



Java Básico

Cátedra de Programación Avanzada

Qué es la JVM?

La JVM Máquina Virtual de Java (*en Inglés Java Virtual Machine*) es un programa ejecutable en una plataforma específica (*Sistema Operativo, p. ej; Windows, Unix, etc.*) que es capaz de interpretar y ejecutar instrucciones escritas en un código especial (*Java ByteCode*) el cual es generado por el compilador de Java.

Es un entorno en el cual se ejecutan los programas escritos en lenguaje Java. Este además de interpretar y ejecutar instrucciones debe asegurarse de realizar algunas tareas principales:

-) Reservar espacio en memoria para los objetos creados.
-) Liberar la memoria no utilizada (*Garbage Collector*).
-) Asignar variables a registros y pilas.
-) Llamar al sistema operativo para ciertas funciones, como accesos a dispositivos.
-) Verificar restricciones de seguridad y robustez de las aplicaciones en ejecución

Si nos focalizamos en la última tarea, es una de las características más importantes del lenguaje:

-) Las referencias en memoria son verificadas en tiempo de ejecución.
-) No existe forma de manipular directamente los punteros de memoria.
-) La JVM gestiona automáticamente el uso de la memoria.

Cómo funciona la JVM?

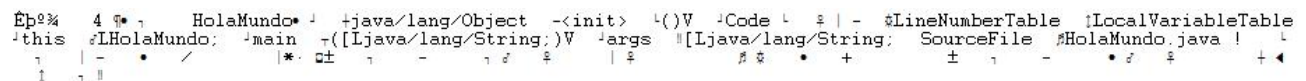
La JVM actúa como un motor en tiempo de ejecución para ejecutar aplicaciones Java. Como mencionamos anteriormente en las características de java se destaca por ser independiente de la plataforma y esto se debe precisamente a la JVM.

La JVM es la que realmente llama al método principal presente en un código java. La JVM es una parte de la JRE (*Java Runtime Environment*).

Cuando escribimos nuestro código en lenguaje Java, lo guardamos en un archivo “.java”. p ej; HolaMundo.java

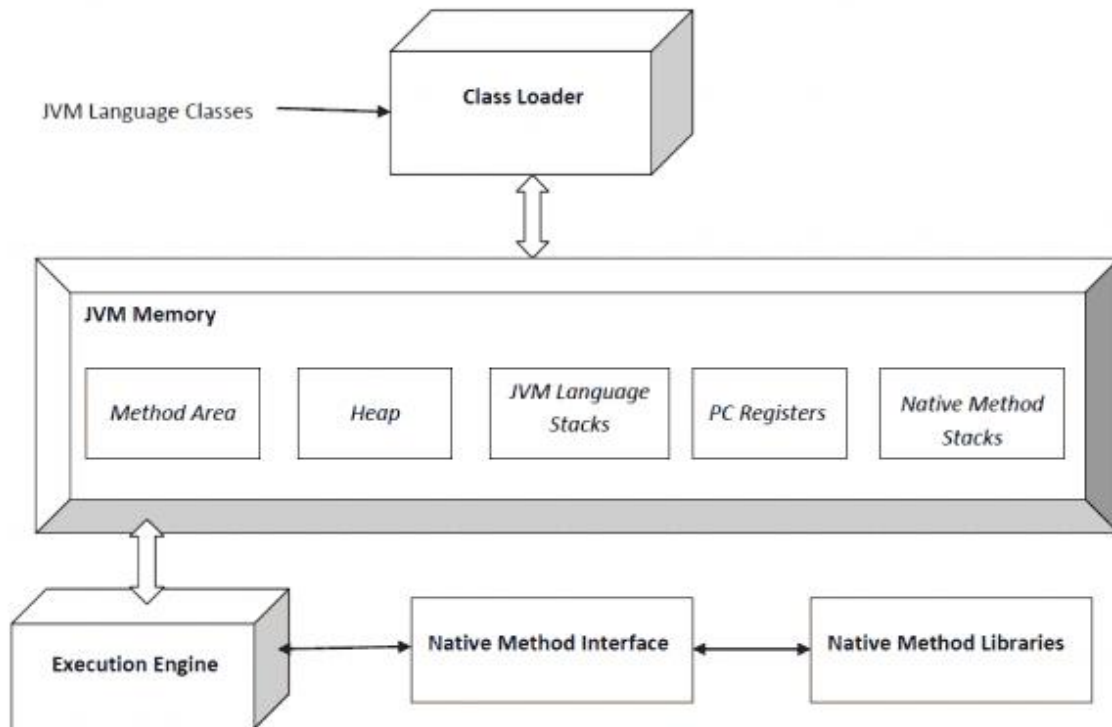
```
public class HolaMundo {
    public static void main(String[] args) {
        System.out.println("Hola Mundo!!!");
    }
}
```

Lo compilamos con un entorno de desarrollo (Eclipse) generando el archivo ByteCode “.class”. p. ej; HolaMundo.class



Vista del archivo HolaMundo.class desde un editor de texto.

Este archivo “.class” generado por el compilador pasa por varios pasos cuando lo ejecutamos en la JVM. La siguiente imagen nos muestra los mismos:



1. Java Class Loader

Es responsable de tres actividades:

a. Carga

El cargador de clases lee el archivo `HolaMundo.class` genera los datos binarios correspondientes y los almacena en el área de métodos. *Method Area*
Para cada archivo .class la JVM almacena la siguiente información:

- i. Nombre completo de la clase y su clase primaria inmediata.
- ii. Si el archivo .class está relacionado con *Class* o *Interface* o *Enum*.
- iii. Información sobre métodos, variables, etc.

Después de cargar el archivo `HolaMundo.class` crea un objeto de tipo *Class* para representar el mismo en la memoria *Heap*.

b. Enlace

Realiza la verificación, preparación y resolución.

- i. Verificación: Comprueba si este archivo está formado correctamente y generado por un compilador válido o no. Si la verificación falla obtenemos la excepción en tiempo de ejecución.
- ii. Preparación: La JVM asigna memoria para las variables de clase e inicializa la memoria a valores predeterminados.
- iii. Resolución: es el proceso de reemplazar referencias simbólicas del tipo con referencias directas. Se realiza buscando en el área del método (*Method Area*) para

localizar la entidad a la que se hace referencia.

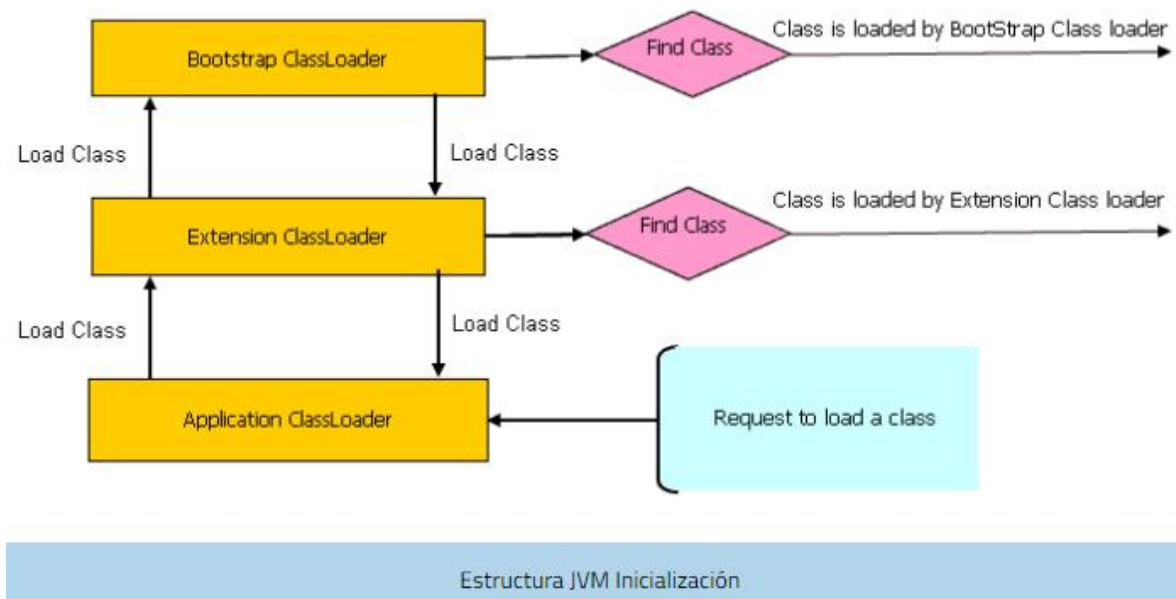
c. Inicialización

En esta fase, todas las variables estáticas se asignan con sus valores definidos en el código y en el bloque estático (si corresponde). Esto se ejecuta de arriba a abajo en una clase y de padres a hijos en la jerarquía de clases.

En general, hay tres cargadores de clase (*Class Loaders*):

-) Cargador de clases Bootstrap (*Bootstrap Class Loader*): cada implementación de JVM debe tener un cargador de clases de arranque, capaz de cargar clases confiables. Carga las clases API centrales de java presentes en el directorio `JAVA_HOME/jre/lib`. Esta ruta se conoce popularmente como ruta de arranque. Se implementa en lenguajes nativos como C, C++.
-) Cargador de clases de extensiones (*Extension Class Loader*): es un elemento secundario del Bootstrap Class Loader. Carga las clases presentes en los directorios de extensiones `JAVA_HOME/jre/lib/ext` (Ruta de extensión) o cualquier otro directorio especificado por la propiedad del sistema `java.ext.dirs`. Está implementado en Java por la clase `sun.misc.Launcher$ExtClassLoader`.
-) Cargador de clases de Sistema/Aplicación (*System/Application Class Loader*): es secundario del cargador de clases de extensión. Es responsable de cargar las clases desde la ruta de la clase de aplicación. Internamente utiliza la variable de entorno asignada a `java.class.path`. También se implementa en Java mediante la clase `sun.misc.Launcher$AppClassLoader`.

La JVM sigue el principio Delegación-Jerarquía para cargar clases. El cargador de clases del sistema delega la solicitud de carga al cargador de clases de extensión y la solicitud de delegado del cargador de clases de extensión al cargador de clases bootstrap. Si la clase se encuentra en la ruta boot-strap, la clase se carga; de lo contrario, solicita de nuevo la transferencia al cargador de clases de extensión y luego al cargador de clases del sistema. Por último, si el cargador de clases del sistema no puede cargar la clase, entonces obtenemos la excepción de tiempo de ejecución `java.lang.ClassNotFoundException`.



2. JVM Memory

Este se compone de varios módulos:

- a. **Área de método** (*Method area*): en el área de método se almacena toda la información del nivel de clase, como el nombre de clase, el nombre inmediato de la clase principal, la información de métodos y variables, etc., incluidas las variables estáticas. Solo hay un área de método por JVM, y es un recurso compartido.
- b. **Área Heap** (*Heap area*): la información de todos los objetos se almacena en el área heap. También hay un área heap por JVM. También es un recurso compartido.
- c. **Área de pila** (*Stack area*): para cada subproceso, JVM crea una pila en tiempo de ejecución que se almacena aquí. Cada bloque de esta pila se llama registro de activación/marco de pila que almacena los métodos de llamadas. Todas las variables locales de ese método se almacenan en su marco correspondiente. Una vez que finaliza un hilo, JVM destruirá la pila en tiempo de ejecución. No es un recurso compartido.
- d. **Registros de PC** (*PC Registers*): Almacena la dirección de la instrucción de ejecución actual de un hilo. Obviamente, cada hilo tiene registros de PC separados.
- e. **Pilas de métodos nativos** (*Native method stacks*): para cada hilo, se crea una pila nativa separada. Almacena información del método nativo.

3. Motor de Ejecución (Execution Engine)

El motor de ejecución ejecuta `HolaMundo.class` (*archivo ByteCode*). Lee el código línea por línea, usa datos e información presente en varias áreas de memoria y ejecuta instrucciones. Se puede clasificar en tres partes:

- a. **Intérprete**: interpreta el bytecode línea por línea y luego lo ejecuta.
- b. **Compilador Just-In-Time (JIT)**: se usa para aumentar la eficiencia del intérprete. Compila todo el ByteCode y lo cambia a código nativo para que cada vez que el intérprete vea llamadas a métodos repetidos, JIT proporcione código nativo directo para esa parte, de modo que la reinterpretación no sea necesaria, por lo tanto, se mejora la eficiencia.
- c. **Recolector de basura**: destruye los objetos no referenciados.

4. Interfaz nativa de Java (JNI – Java Native Interface)

Es una interfaz que interactúa con las Bibliotecas de métodos nativos y proporciona las bibliotecas nativas (C, C++) necesarias para la ejecución. Permite a la JVM llamar a bibliotecas C/C++ y ser llamado por bibliotecas C/C++ que pueden ser específicas del hardware.

5. Bibliotecas de métodos nativos (Native Method Libraries)

Es una colección de bibliotecas nativas (C, C++) que requiere el motor de ejecución.

Tipos de Datos en Java

En java tenemos dos tipos de datos:

-) Tipos de datos primitivos
-) Tipos de datos Objetos (Wrapper)
-) Tipos especiales

Tipos de Datos Primitivos

Estos son tipos de datos del lenguaje, almacenan un valor único y no poseen ningún tipo de característica especial.

TIPO	DESCRIPCIÓN	DEFAULT	TAMAÑO	EJEMPLOS
boolean	true o false	false	1 bit	true, false
byte	entero complemento de dos	0	8 bits	100, -50
char	carácter unicode	\u0000	16 bits	'a', '\u0041', '\101', '\\'
short	entero complemento de dos	0	16 bits	10000, -20000
int	entero complemento de dos	0	32 bits	100000, -2, -1, 0, 1, 2, -200000
long	entero complemento de dos	0	64 bits	-2L, -1L, 0L, 1L, 2L
float	coma flotante IEEE 754	0.0	32 bits	1.23e100f, -1.23e-100f, .3ef, 3.14f
double	coma flotante IEEE 754	0.0	64 bits	1.2345e300d, -1.2345e-300f, 1e1d

Se pueden organizar en 4 grupos:

1. Numéricos Enteros

1.1. byte

Representa un tipo de dato de 8 bits con signo. De tal manera que puede almacenar los valores numéricos de -128 a 127.

1.2. short

Representa un tipo de dato de 16 bits con signo.
De esta manera almacena valores numéricos de -32768 a 32767.

1.3. int

Es un tipo de dato de 32 bits con signo para almacenar valores numéricos.
Cuyo valor mínimo es -2^{31} y el valor máximo $2^{31}-1$.

1.4. long

Es un tipo de dato de 64 bits con signo que almacena valores numéricos entre -2^{63} a $2^{63}-1$.

2. Carácter

2.1. char

Es un tipo de datos que representa a un carácter Unicode sencillo de 16 bits.

3. Numéricos Decimal

3.1. float

Es un tipo de dato para almacenar números en coma flotante con precisión simple de 32 bits.
Sufijo: F/f Ejemplo: 9.8f

3.2. double

Es un tipo de dato para almacenar números en coma flotante con doble precisión de 64 bits.

4. Lógicos

4.1. boolean

Es un tipo de dato lógico. Sólo puede almacenar los valores lógicos *true* o *false*.

Tipos de datos Objetos (Wrapper)

Un tipo de dato Wrapper es una clase que envuelve o contiene tipos de datos primitivos. Esto permite disponer de características especiales que incrementan su usabilidad y necesidad.

Para cada tipo de dato primitivo tenemos su equivalencia como Objeto:

Tipo Primitivo	Tipo Objeto
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Ejemplo: Clase Integer

Método	Descripción
Integer(int valor) Integer(String valor)	Constructores a partir de int y String
int intValue() / byte byteValue() / float floatValue() . . .	Devuelve el valor en distintos formatos, int, long, float, etc.
boolean equals(Object obj)	Devuelve true si el objeto con el que se compara es un Integer y su valor es el mismo.
static Integer getInteger(String s)	Devuelve un Integer a partir de una cadena de caracteres. Estático
static int parseInt(String s)	Devuelve un int a partir de un String. Estático.
static String toBinaryString(int i) static String toOctalString(int i) static String toHexString(int i) static String toString(int i)	Convierte un entero a su representación en String en binario, octal, hexadecimal, etc. Estáticos
String toString()	
static Integer valueOf(String s)	Devuelve un Integer a partir de un String. Estático.

Tipos especiales

Además de los tipos de datos primitivos y Objetos se tiene tipos especiales:

) void

Este tipo especial se usa para representar la ausencia de datos. Se usa en la definición de métodos. Datos de este tipo no pueden existir.