3. Introducción a la tecnología Java: la maquina virtual.

Desde ya hace mucho tiempo venimos escuchado en todas partes sobre la tecnología Java, pero ¿Qué es exactamente esta tecnología? Empecemos conociendo que está compuesta por 2 partes:

- El lenguaje de programación.
- La plataforma.

3.1. Orígenes del lenguaje Java.

La plataforma Java y el lenguaje Java empezaron como un proyecto interno de *Sun Microsystem* en diciembre de 1990. *Patrick Naughton*, ingeniero de *Sun*, estaba decepcionado con el estado de C++ y la API de C y sus herramientas. Mientras consideraba migrar a *NeXT*, *Naughton* recibió la oferta de trabajar en una nueva tecnología, y así comenzó el proyecto *Stealth.*

El Proyecto *Stealth* fue rebautizado como *Green Project* (o *Proyecto Verde*) cuando *James Gosling* y *Mike Sheridan* se unieron a *Naughton*. Con la ayuda de otros ingenieros, empezaron a trabajar en una pequeña oficina en *Sand Hill Road* en Menlo Park, California. Intentaban desarrollar una nueva tecnología para programar la siguiente generación de *dispositivos inteligentes*, en los que Sun veía un campo nuevo a explotar.

El equipo pensó al principio usar C++, pero se descartó por varias razones. Al estar desarrollando un sistema empotrado con recursos limitados, C++ no es adecuado por necesitar mayor potencia además de que su complejidad conduce a errores de desarrollo. La ausencia de un *recolector de basura* (*garbage collector*) obligaba a los programadores a manejar manualmente el sistema de memoria, una tarea peligrosa y proclive a fallos. El equipo también se encontró con problemas por la falta de herramientas portables en cuanto a seguridad, programación distribuida, y programación concurrente. Finalmente abogaban por una plataforma que fuese fácilmente portable a todo tipo de dispositivo.

Bill Joy había concebido un nuevo lenguaje que combinase lo mejor de *Mesa* y C. En un escrito titulado *Further* (más lejos), proponía a Sun que sus ingenieros crearan un entorno *orientado a objetos* basado en C++. Al principio Gosling intentó modificar y ampliar C++, a b que llamó C++ ++ --, pero pronto descartó la idea para crear un lenguaje completamente nuevo, al que llamó *Oak*, en referencia al roble que tenía junto a su oficina.

El equipo dedicó largas horas de trabajo y en el verano de 1992 tuvieron lista algunas partes de la plataforma, incluyendo el Sistema Operativo Green, el lenguaje Oak, las librerías y el hardware. La primera prueba, llevada a cabo el 3 de Septiembre de 1992, se centró en construir una PDA (*Personal Digital Assistant* o Asistente Digital Personal) llamada *Star7*, que contaba con una interfaz gráfica y un asistente apodado "Duke" para guiar al usuario.

En noviembre de ese mismo año, el Proyecto Verde se convirtió en **FirstPerson, Inc**, una división propiedad de Sun MicroSystem, y el equipo se trasladó a *Palo Alto* (California). El interés se centró entonces en construir dispositivos interactivos, hasta que Time Warner publicó una solicitud de oferta para un adaptador de televisión. Es decir, un aparato que se sitúa entre la televisión y una fuente de señal externa y que adapta el contenido de ésta (video, audio, páginas Web, etc.) para verse en la pantalla. Entonces, Firsperson cambió de idea y envió a Warner una propuesta para el dispositivo que deseaban. Sin embargo, la industria del cable consideró que esa propuesta daba demasiado control al usuario, con lo que FirstPerson perdió la puja a favor de Silicon Graphics Incorporated. Un trato con la empresa 3DO para el mismo tipo de dispositivo tampoco llegó a buen puerto. Viendo que no había muchas posibilidades en la industria de la televisión, la compañía volvió al seno de Sun.

3.2. El lenguaje de programación Java.

Podemos empezar diciendo que el lenguaje Java es de alto nivel y sus características más importantes son:

- Lenguaje orientado a objetos.
- Java es un lenguaje sencillo.
- Independiente de plataforma
- Brinda un gran nivel de seguridad
- Capacidad multihilo
- Gran rendimiento
- Creación de aplicaciones distribuidas

Su robustez o lo integrado que tiene el protocolo *TCP/IP* lo que lo hace un lenguaje ideal para Internet.

Tradicionalmente se han dividido los lenguajes en compilados e interpretados. Los primeros necesitan ser traducidos por un programa llamado compilador al lenguaje máquina, que es el que entiende el ordenador. Como ejemplo de estos lenguajes podríamos citar a *C*, *C++*, *Visual Basic*, *Clipper*, etc. Los interpretados, en cambio, son traducidos mientras se ejecutan, por ejemplo *HTML*, *WML* o *XML*, por lo cual no necesitan ser compilados.

Así pues la diferencia entre estos lenguajes radica en la manera de ejecutarlos. Mientras que los compilados sólo se compilan una vez y lo hacen pasando todo el programa a código máquina (si da un error aunque sea en la ultima línea no podríamos ejecutar nada de nada), en el momento que lo hemos compilado correctamente se genera un archivo *.exe* que se puede ejecutar tantas veces como queramos sin tener que volver a compilar. Los interpretados en cambio, cada vez que los queramos ejecutar tendremos que interpretarlos línea a línea, es más lento, pero puede ocurrir un error en la ultima línea y a diferencia de los compilados, el programa se ejecuta justo hasta la línea que produce el error.

Java está diseñado para que un programa escrito en este lenguaje sea ejecutado independientemente de la plataforma (hardware, software y sistema operativo) en la que se esté actuando. Esta portabilidad se consigue haciendo de Java un lenguaje medio interpretado medio compilado. ¿Cómo se come esto? Pues se coge el código fuente, se

compila a un lenguaje intermedio cercano al lenguaje máquina pero independiente del ordenador y el sistema operativo en que se ejecuta (llamado en el mundo Java *bytecodes*).

Figura 3.1: Compilación de un programa Java.

Finalmente, se interpreta ese lenguaje intermedio por medio de un programa denominado máquina virtual de Java (JVM), que sí depende de la plataforma.

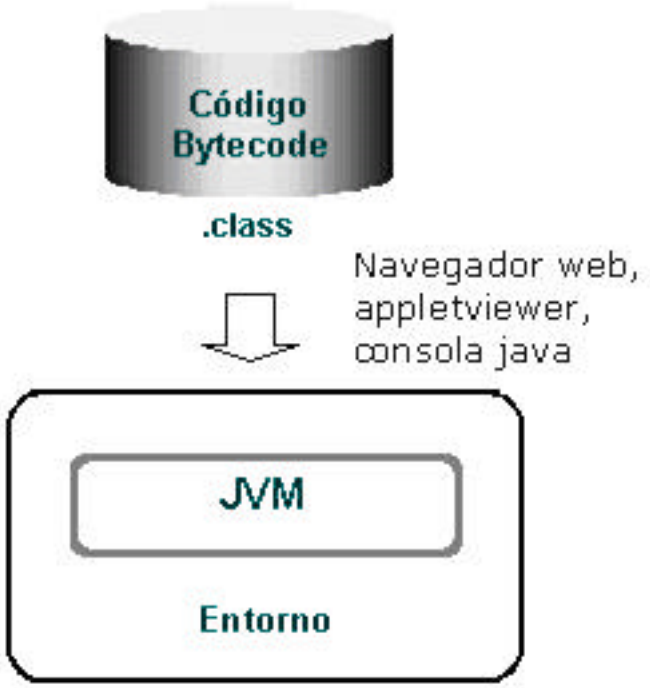

Figura 3.2: Ejecución de un programa Java.

Los java *bytecodes* permiten el ya conocido "write once, run anywhere" (compila una sola vez y ejecútalo donde quieras). Podemos compilar nuestros programas a *bytecodes* en cualquier plataforma que tenga el compilador Java. Los *bytecodes* luego pueden ejecutarse en cualquier implementación de la máquina virtual de Java (JVM). Esto significa que mientras el ordenador tenga un JVM, el mismo programa escrito en Java puede ejecutarse en Windows, *Solaris*, *iMac*, *Linux*, etc.

3.3. Principales características del lenguaje Java.

Las principales que han llevado al lenguaje Java a la extensión y profundidad de mercado de la cual disponen son los que se muestran a continuación:

3.3.1. Orientado a objetos.

La primera característica, orientado a objetos ("OO"), se refiere a un método de programación y al diseño del lenguaje. Aunque hay muchas interpretaciones para OO, una primera idea es diseñar el software de forma que los distintos tipos de datos que use estén unidos a sus operaciones. Así, los datos y el código (funciones o métodos) se combinan en entidades llamadas objetos. Un objeto puede verse como un paquete que contiene el "comportamiento" (el código) y el "estado" (datos). El principio es separar aquello que cambia de las cosas que permanecen inalterables. Frecuentemente, cambiar una estructura de datos implica un cambio en el código que opera sobre los mismos, o viceversa. Esta separación en objetos coherentes e independientes ofrece una base más estable para el diseño de un sistema software.

El objetivo es hacer que grandes proyectos sean fáciles de gestionar y manejar, mejorando como consecuencia su calidad y reduciendo el número de proyectos fallidos. Otra de las grandes promesas de la programación orientada a objetos es la creación de entidades más genéricas (objetos) que permitan la reutilización del software entre proyectos, una de las premisas fundamentales de la Ingeniería del Software. Un objeto genérico "cliente", por ejemplo, debería en teoría tener el mismo conjunto de comportamiento en diferentes proyectos, sobre todo cuando estos coinciden en cierta medida, algo que suele suceder en las grandes organizaciones. En este sentido, los objetos podrían verse como piezas reutilizables que pueden emplearse en múltiples proyectos distintos, posibilitando así a la industria del software a construir proyectos de envergadura empleando componentes ya existentes y de comprobada calidad; conduciendo esto finalmente a una reducción drástica del tiempo de desarrollo. Podemos usar como ejemplo de objeto el aluminio.

Una vez definidos datos (peso, maleabilidad, etc.), y su "comportamiento" (soldar dos piezas, etc.), el objeto "aluminio" puede ser reutilizado en el campo de la construcción, del automóvil, de la aviación, etc.

La reutilización del software ha experimentado resultados dispares, encontrando dos dificultades principales: el diseño de objetos realmente genéricos es pobremente comprendido, y falta una metodología para la amplia comunicación de oportunidades de reutilización. Algunas comunidades de "código abierto" (open source) quieren ayudar en este problema dando medios a los desarrolladores para diseminar la información sobre el uso y versatilidad de objetos reutilizables y librerías de objetos.

3.3.2. Independencia de la plataforma.

La segunda característica, la independencia de la plataforma, significa que programas escritos en el lenguaje Java pueden ejecutarse igualmente en cualquier tipo de hardware. Es lo que significa ser capaz de escribir un programa una vez y que pueda

ejecutarse en cualquier dispositivo, tal como reza el axioma de Java, ‘’’write once, run everywhere’’’.

Para ello, se compila el código fuente escrito en lenguaje Java, para generar un código conocido como “bytecode” (específicamente Java bytecode) instrucciones máquinas simplificadas específicas de la plataforma Java. Esta pieza está “a medio camino” entre el código fuente y el código máquina que entiende el dispositivo destino. El *bytecode* es ejecutado entonces en la máquina virtual (VM), un programa escrito en código nativo de la plataforma destino (que es el que entiende su hardware), que interpreta y ejecuta el código. Además, se suministran librerías adicionales para acceder a las características de cada dispositivo (como los gráficos, ejecución mediante hebras o *threads*, la interfaz de red) de forma unificada. Se debe tener presente que, aunque hay una etapa explícita de compilación, el *bytecode* generado es interpretado o convertido a instrucciones máquina del código nativo por el compilador JIT (Just In Time).

Hay implementaciones del compilador de Java que convierten el código fuente directamente en código objeto nativo, como GCJ. Esto elimina la etapa intermedia donde se genera el *bytecode*, pero la salida de este tipo de compiladores sólo puede ejecutarse en un tipo de arquitectura.

La licencia sobre Java de Sun insiste que todas las implementaciones sean “compatibles”. Esto dio lugar a una disputa legal entre Microsoft y Sun, cuando éste último alegó que la implementación de Microsoft no daba soporte a las interfaces RMI y JNI además de haber añadido características ‘’dependientes’’ de su plataforma. Sun demandó a Microsoft y ganó por daños y perjuicios (unos 20 millones de dólares) así como una orden judicial forzando la aceptación de la licencia de Sun. Como respuesta, Microsoft no ofrece Java con su versión de sistema operativo, y en recientes versiones de Windows, su navegador Internet Explorer no admite la ejecución de *applets* sin un conector (o *plugin*) aparte. Sin embargo, Sun y otras fuentes ofrecen versiones gratuitas para distintas versiones de Windows.

Las primeras implementaciones del lenguaje usaban una máquina virtual interpretada para conseguir la portabilidad. Sin embargo, el resultado eran programas que se ejecutaban comparativamente más lentos que aquellos escritos en C o C++. Esto hizo que Java se ganase una reputación de lento en rendimiento. Las implementaciones recientes de la JVM dan lugar a programas que se ejecutan considerablemente más rápido que las versiones antiguas, empleando diversas técnicas.

La primera de estas técnicas es simplemente compilar directamente en código nativo como hacen los compiladores tradicionales, eliminando la etapa del *bytecode*. Esto da lugar a un gran rendimiento en la ejecución, pero tapa el camino a la portabilidad. Otra técnica, conocida como compilación JIT (*Just In Time*, o ‘’’compilación al vuelo’’’), convierte el bytecode a código nativo cuando se ejecuta la aplicación. Otras máquinas virtuales más sofisticadas usan una ‘’’recompilación dinámica’’’ en la que la VM es capaz de analizar el comportamiento del programa en ejecución y recompila y optimiza las partes críticas.

La recompilación dinámica puede lograr mayor grado de optimización que la compilación tradicional (o estática), ya que puede basar su trabajo en el conocimiento que de primera mano tiene sobre el entorno de ejecución y el conjunto de clases

cargadas en memoria. La compilación *JIT* y la recompilación dinámica permiten a los programas Java aprovechar la velocidad de ejecución del código nativo sin por ello perder la ventaja de la portabilidad:

La portabilidad es técnicamente difícil de lograr, y el éxito de Java en ese campo ha sido dispar. Aunque es de hecho posible escribir programas para la plataforma Java que actúen de forma correcta en múltiples plataformas de distinta arquitectura, el gran número de estas con pequeños errores o inconsistencias llevan a que a veces se parodie el eslogan de Sun, "Write once, run everywhere" como "Write once, debug everywhere" (o “Escríbelo una vez, ejecútalo en todas partes” por “Escríbelo una vez, depúralo en todas partes”)

El concepto de independencia de la plataforma de Java cuenta, sin embargo, con un gran éxito en las aplicaciones en el entorno del servidor, como los Servicios Web, los Servlets, los Java Beans, así como en sistemas empotrados basados en OSGi, usando entornos Java empotrados.

3.3.3. Recolector de basura.

Un argumento en contra de lenguajes como C++ es que los programadores se encuentran con la carga añadida de tener que administrar la memoria de forma manual. En C++, el desarrollador debe asignar memoria en una zona conocida como *heap* (montículo) para crear cualquier objeto, y posteriormente desalojar el espacio asignado cuando desea borrarlo. Un olvido a la hora de desalojar memoria previamente solicitada, o si no lo hace en el instante oportuno, puede llevar a una *fuga de memoria*, ya que el sistema operativo piensa que esa zona de memoria está siendo usada por una aplicación cuando en realidad no es así. Así, un programa mal diseñado podría consumir una cantidad desproporcionada de memoria. Además, si una misma región de memoria es desalojada dos veces el programa puede volverse inestable y llevar a un eventual *cuelgue*.

En Java, este problema potencial es evitado en gran medida por el recolector automático de basura (o *automatic garbage collector*). El programador determina cuándo se crean los objetos y el entorno en tiempo de ejecución de Java (Java runtime) es el responsable de gestionar el ciclo de vida de los objetos. El programa, u otros objetos pueden tener localizado un objeto mediante una referencia a éste (que, desde un punto de vista de bajo nivel es una dirección de memoria). Cuando no quedan referencias a un objeto, el recolector de basura de Java borra el objeto, liberando así la memoria que ocupaba previniendo posibles fugas (ejemplo: un objeto creado y únicamente usado dentro de un método sólo tiene entidad dentro de éste; al salir del método el objeto es eliminado). Aún así, es posible que se produzcan fugas de memoria si el código almacena referencias a objetos que ya no son necesarios—es decir, pueden aún ocurrir, pero en un nivel conceptual superior. En definitiva, el recolector de basura de Java permite una fácil creación y eliminación de objetos, mayor seguridad y frecuentemente más rápida que en C++.

La recolección de basura de Java es un proceso prácticamente invisible al desarrollador. Es decir, el programador no tiene conciencia de cuándo la recolección de basura tendrá lugar, ya que ésta no tiene necesariamente que guardar relación con las

acciones que realiza el código fuente. Debe tenerse en cuenta que la memoria es sólo uno de los muchos recursos que deben ser gestionados.

3.4. Plataforma Java.

Con plataforma nos referimos al ambiente de hardware y software en donde el programa se ejecuta, por ejemplo, plataformas como *Linux*, *Solaris*, *Windows 2003* y *MacOS*. En casi todos los casos las plataformas son descritas como la combinación del sistema operativo y el hardware. La plataforma Java se diferencia de estas plataformas, es que es una plataforma sólo de software y se ejecuta sobre las otras plataformas de hardware.

La plataforma Java tiene 2 componentes:

- La máquina virtual de Java (JVM)
- El Java API (Application Programming Interface)

Ya hemos visto algo de la máquina virtual de Java (JVM); es la base de la plataforma Java y es llevada a diferentes plataformas de hardware.

El Java API es una gran colección de componentes de software que proporcionan muchas utilidades para el programador, por ejemplo, los API's para las interfases gráficas. Los API's de Java están agrupados en librerías de ciertas Clases e interfaces, estas librerías son conocidas como paquetes.

El siguiente gráfico describe un programa que se está ejecutando sobre la plataforma Java. Como vemos, el Java API y la máquina virtual aíslan al programa del hardware:

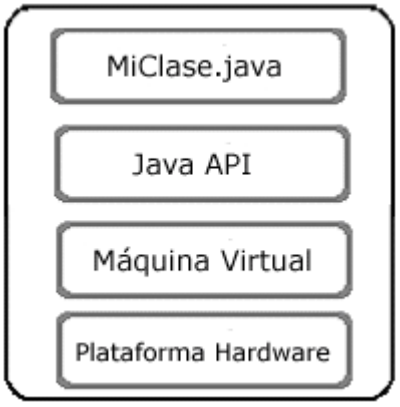

Figura 3.3: Programa que se ejecuta sobre la plataforma Java.

3.5. Tecnologías Java.

Actualmente Sun Microsystems ha agrupado la tecnología Java en tres tecnologías claramente diferenciadas, cada una de ellas adaptada a un área específica de la industria:

• *Java 2 Platform, Enterprise Edition* (J2EE) pensada para servir las necesidades que puedan tener las empresas que quieran ofrecer servicios a sus clientes, proveedores y empleados.

• *Java 2 Platform, Standard Edition* (J2SE) pensada para satisfacer las necesidades de usuarios y programadores en sus equipos personales y estaciones de trabajo.

• *Java 2 Micro Edition* (J2ME) enfocada tanto para productores de dispositivos portátiles de consumo como para quienes proporcionan servicios de información disponibles para estos dispositivos.

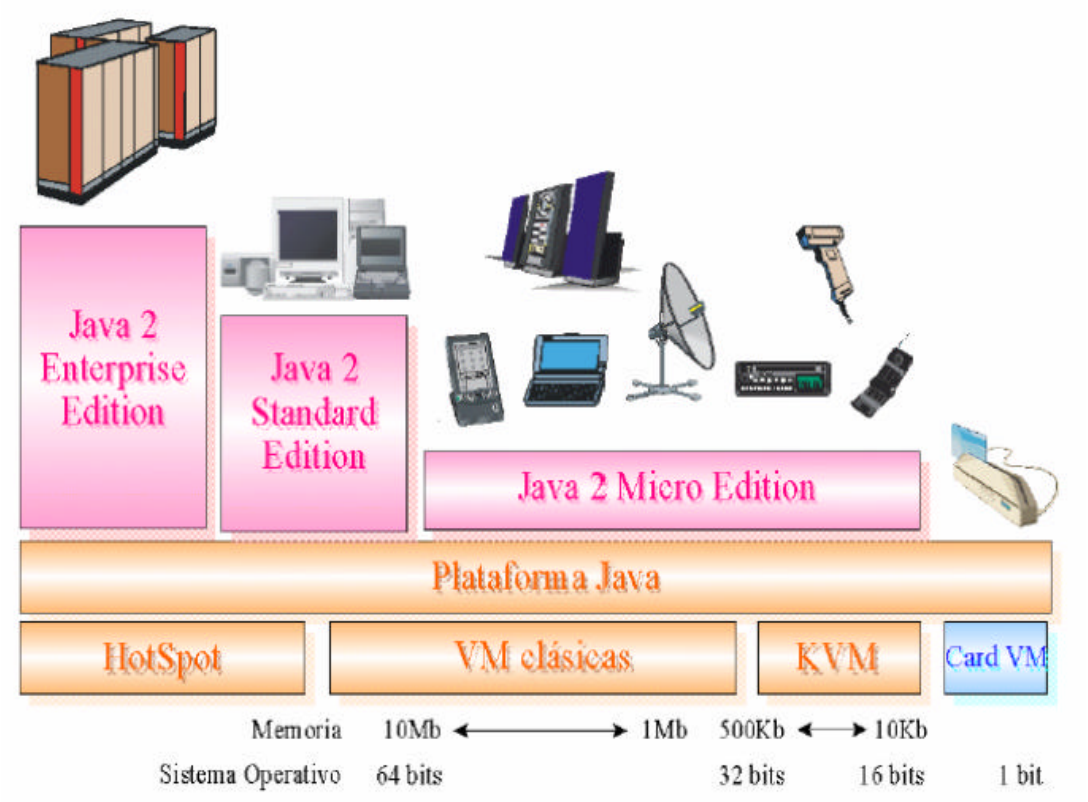

Figura 3.4: Tecnologías Java disponibles actualmente.

Cada una de estas plataformas define en su interior un conjunto de tecnologías que pueden ser utilizadas con un producto en particular:

• *Java Virtual Machine* que encuadra en su interior un amplio rango de equipos de computación.
• Librerías y APIs especializadas para cada tipo de dispositivo.
• Herramientas para desarrollo y configuración de equipos.

3.6. Maquina virtual Java.

La primera **máquina virtual de Java** (JVM o *Java Virtual Machine*), desarrollada por Sun Microsystems, es una **máquina virtual** que ejecuta el código resultante de la compilación de un programa escrito mediante el lenguaje de programación Java, conocido como bytecode de Java. Aunque compiladores de otros lenguajes pueden dar lugar a bytecode ejecutable sobre la JVM.

La JVM es un componente crucial de la plataforma Java. La disponibilidad de JVMs para gran cantidad de tipos de hardware y plataformas software hacen que Java pueda funcionar tanto como un software intermediario entre aplicaciones (middleware), y como una plataforma en sí misma. De ahí la expresión "Write once, run anywhere" (escribir una vez, ejecutar en cualquier parte).

A partir de J2SE 5.0, los cambios en la especificación de la JVM han sido desarrollados bajo el organismo Java Community Process en forma de la JSR (Java Specification Request) 924. Desde el año 2006, los cambios en la especificación para recoger los cambios propuestos al formato de ficheros de clase (JSR 202[1]) se están realizando como parte del mantenimiento de la JSR 924. La especificación de la JVM se encuentra publicada en forma de libro, conocido como el "libro azul". El prefacio reza así:

> "Pretendemos que esta especificación documente suficientemente la Máquina Virtual de Java para hacer posibles implementaciones compatibles de entornos de confianza. Sun ofrece mecanismos para verificar una correcta operación de las implementaciones de la Máquina Virtual de Java."

Kaffe es un ejemplo de una de estas implementaciones "limpias" de Java. Sun mantiene el control sobre la marca registrada "Java", que usa para certificar aquellas implementaciones que se son perfectamente compatibles con las especificaciones de Sun.

3.6.1. Entorno de ejecución.

Un programa escrito en Java, debe pasar por un proceso que se llama compilación. Si este término es generalmente usado para el mecanismo encargado de convertir código fuente a código nativo de la plataforma sobre la que pretende ejecutarse, en Java, la compilación produce un código intermedio (el bytecode) destinado a ser ejecutado por la máquina virtual de Java. Este bytecode toma la forma de un fichero con extensión ".class". Todo programa en Java está compuesto por una o varias clases, y cada clase estará en un archivo diferente, tanto antes como después de la compilación.

Para facilitar la distribución de grandes aplicaciones, las clases que lo forman pueden ser empaquetadas juntas, tras su compilación, en un único archivo (con extensión ".jar"). Este archivo puede entonces ser ejecutado por la JVM de dos formas distintas. La forma original era mediante un proceso conocido como interpretación, por el cual se lleva a cabo la emulación del conjunto de instrucciones de la JVM, y las

instrucciones se ejecutan secuencialmente. Este proceso, como suele serlo generalmente la ejecución de programas mediante la interpretación suele ser lento, por lo que más modernamente se lleva a cabo un proceso conocido como compilación JIT (Just In Time, o "al momento"). Esta compilación se aplica sobre el bytecode generado tras la primera compilación al inicio de la aplicación, y da como resultado una ejecución mucho más eficiente, aunque con la penalización en tiempo que conlleva esta segunda compilación. Un ejemplo de este tipo de compilador es el HotSpot de Sun.

La arquitectura básica que presenta la maquina virtual Java y su interacción con el compilador queda representada en la siguiente figura:

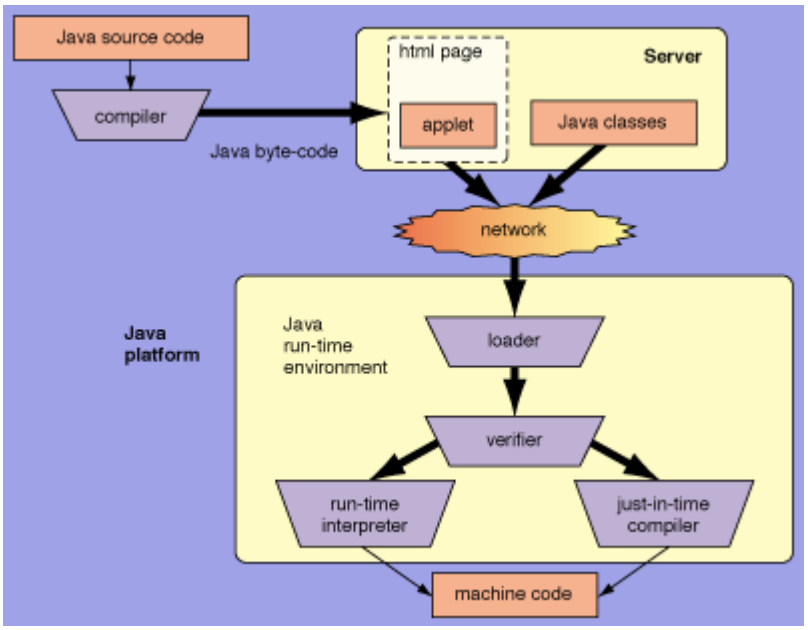

Figura 3.5: Estructura simplificada interna de la maquina virtual.

Donde como se puede observar la plataforma dispone básicamente de tres módulos:

- *Loader*: para cargar las clases en memoria.
- *Verifier*: verifica las clases que se han cargado.
- *Interprete*: ejecuta los bytecodes.
- *JIT compiler*: compila e interpreta los bytecodes en tiempo de ejecución.

Una descripción mas detallada de cada uno de estos componentes se realizará a lo largo del proyecto puesto que tal es el objetivo principal del proyecto. Además de estos componentes principales hay otros componentes auxiliares que se encuentran recogidos en la siguiente figura y que también serán estudiados en profundidad mas adelante:

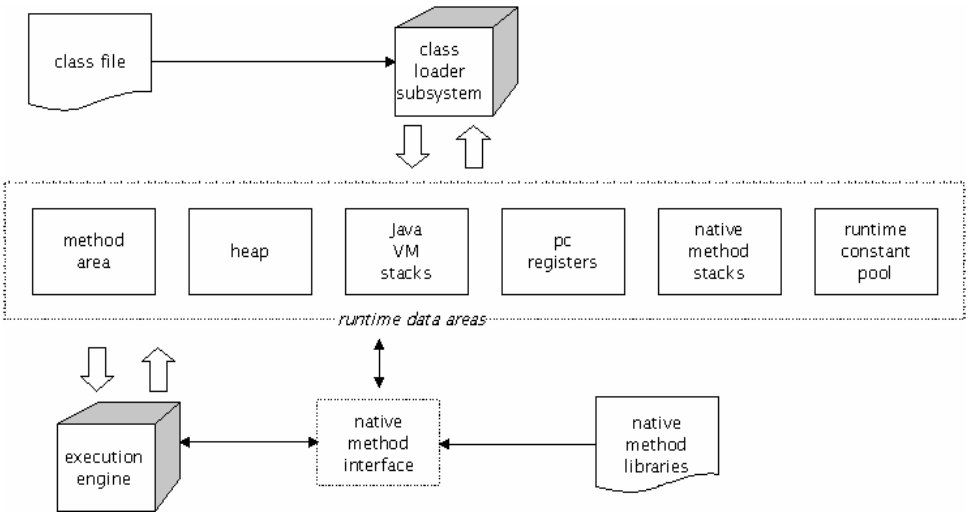

Figura 3.6: Estructura detallada de la maquina virtual.

3.6.2. La maquina virtual y los bytecodes Java.

La máquina virtual de Java "verifica" todo el bytecode antes de su ejecución. Esto significa que sólo una parte de éste puede dar lugar a código ejecutable. Por ejemplo, una instrucción JUMP (de salto) sólo puede dirigirse a una instrucción dentro de la misma función. Por esta razón, el hecho de que la JVM sea una arquitectura basada en pila no implica una penalización en tiempo cuando se emula sobre una arquitectura basada en registros, al usar un compilador JIT. La verificación de código asegura que no se pueda acceder a zonas de memoria restringidas. Por tanto, la protección de memoria la lleva a cabo la propia JVM sin necesidad de una unidad explícita para la gestión de memoria o MMU (*Memory Management Unit*), siendo útil en sistemas que no cuentan con este tipo de hardware.

La máquina virtual de Java (JVM) soporta instrucciones para los siguientes grupos de tareas:

- Carga y almacenamiento
- Aritméticas
- Conversión de tipos
- Creación de objetos y su manipulación
- Gestión de la pila (push / pop, meter / sacar)
- Saltos
- Llamada a métodos y salida de estos
- Generación de excepciones

El objetivo es la compatibilidad. Cada sistema operativo y arquitectura en que tenga que ejecutarse una aplicación Java debe tener su propia implementación de la JVM. Estas máquinas virtuales interpretan el bytecode de igual forma desde el punto de vista semántico, pero cada implementación puede ser distinta. Más complicado que la emulación del bytecode es la implementación completa y eficiente de la API de Java, en cada plataforma destino.