# Arquitectura de la Máquina Virtual Java

**Mtro. en C. Rolando Menchaca Méndez**

**Dr. Félix García Carballeira**

**Palabras Clave** *:* Java, Máquina Virtual Java, Seguridad en Java, Generación de Código Justo en el Momento.

## Resumen

*En el presente artículo se describe la Arquitectura de la Máquina Virtual Java, que es parte medular de toda la tecnología Java. Se hace énfasis en sus componentes principales, como el procesador virtual Java, que se encarga de ejecutar los códigos de operación (bytecodes) generados por los compiladores Java, el verificador de código, que junto con el cargador de clases y el administrador de seguridad, se encargan de implementar los mecanismos empleados para proporcionar seguridad a los usuarios de Java. Además, a lo largo de la discusión, se hace un análisis de las principales características que han hecho posible el éxito de Java en nuestros días.*

[English]

## Artículo

### 1. Introducción

Cuando una persona desarrolla una aplicación en un lenguaje como C o C++, el archivo binario que genera el compilador y que contiene el código que implementa dicha aplicación, se puede ejecutar únicamente sobre la plataforma sobre la cual fue desarrollada, debido a que dicho código es especifico a esa plataforma.

La plataforma Java se encuentra por encima de otras plataformas. El código que generan sus compiladores no es específico de una maquina física en particular, sino de una máquina virtual. Aún cuando existen múltiples implantaciones de la Máquina Virtual Java, cada una específica de la plataforma sobre la cual subyace, existe una única especificación de la máquina virtual, que proporciona una vista independiente del hardware y del sistema operativo sobre el que se esté trabajando. De esta manera un programador en Java "escribe su programa una vez, y lo ejecuta donde sea"[1] .

Es precisamente la máquina virtual Java la clave de la independencia de los programas Java, sobre el sistema operativo y el hardware en que se ejecutan, ya que es la encargada de proporcionar la vista de un nivel de abstracción superior, donde además de la independencia de la plataforma antes mencionada, presenta un lenguaje de programación simple, orientado a objetos, con verificación estricta de tipos de datos, múltiples hilos, con ligado dinámico y con recolección automática de basura.

A continuación, en la sección 2, describiremos brevemente la historia de Java. En la sección 3 analizaremos las ventajas y desventajas de los lenguajes que utilizan máquinas virtuales, contra los lenguajes completamente interpretados. En la sección 4 definiremos las propiedades que posee el lenguaje Java y en el 5, la forma en que está estructurada para lograr estas propiedades.

Finalmente, en las secciones 6 y 7 se describen las principales desventajas de Java, contra lenguajes totalmente compilados como C.

## 2. El nacimiento de Java

Java es un lenguaje de programación de Sun Microsystems originalmente llamado "Oak", que fue concebido bajo la dirección de James Gosling y Bill Joy, quienes pertenecían a una subsidiaria de Sun, conocida como "*FirstPerson Inc*". Oak nació para programar pequeños dispositivos electrodomésticos, como los asistentes personales digitales PADs (*Personal Digital Assistants*) y un poco más adelante se utilizó para ejecutar aplicaciones para televisores. Ninguno de estos productos tuvo éxito comercial. Gosling y Joy se quedaron con una tecnología robusta, eficiente, orientada a objetos, independiente de la arquitectura, pero hasta ese momento, sin ninguna utilidad práctica.

No pasó mucho tiempo, cuando en Sun se dieron cuenta de que todas estas características cubrían a la perfección las necesidades de las aplicaciones de Internet. De esta manera, con unos cuantos retoques, Oak se convirtió en Java.

Aunado a todas las características que posee Java (modelo de objetos dinámico, sistema estricto de tipos, paquetes, hilos, excepciones, etcétera), cuando Nestcape Inc. anunció su incorporación dentro de su navegador (*Netscape Navigator*), el nivel de interés sobre el lenguaje creció dramáticamente, debido al número importante de personas que utilizan WWW diariamente. Todo lo anterior se ha conjugado para lograr el éxito actual de Java, siendo el actor principal su máquina virtual, tema de este artículo.

## 3. Máquina Virtual vs Lenguajes Interpretados

El concepto de máquina virtual es antiguo. Fue usado por IBM en 1959 para describir uno de los primeros sistemas operativos que existieron en la historia de la computación, el *VM*. En 1970, el ambiente de programación de *SmallTalk* llevó la idea a un nuevo nivel y construyó una máquina virtual para soportar abstracciones orientadas a objetos de alto nivel, sobre las máquinas subyacentes.

Como ya hemos esbozado, las máquinas virtuales tienen varias ventajas importantes. La primera es que presentan un medio excelente para alcanzar la portabilidad. Otra de las ventajas importantes, es que introduce otro nivel de abstracción y de protección, entre la computadora y el software que ejecuta sobre ella. Esto cobra particular importancia en un ambiente donde el código que ejecutamos proviene de algún lugar del mundo y es escrito por alguna "buena" persona.

### 3.1. Lenguajes Totalmente Interpretados

Es posible decir que los lenguajes totalmente interpretados, como *Tcl* y *JavaScript*, también poseen las cualidades de ser altamente portables y seguros, pero existe una diferencia importante entre este tipo de lenguajes y los basados en una máquina virtual: la eficiencia.

Para ejecutar un programa escrito en un lenguaje completamente interpretado, el intérprete debe realizar el análisis léxico y sintáctico en el momento de estar ejecutando el programa, lo que provoca una sobrecarga muy considerable en la ejecución del mismo. De hecho, en algunas pruebas informales *Tcl* puede ser hasta 200 veces más lento que C [1].

### 3.2. Lenguajes Compilados de Código Intermedio

Los lenguajes basados en una máquina virtual, comúnmente son más rápidos que los totalmente interpretados, debido a que utilizan una arquitectura de código intermedio. La idea es dividir la tarea de ejecutar un programa en dos partes. En la primera, se realiza el análisis léxico y sintáctico del programa fuente, para generar el programa en instrucciones del procesador virtual (código intermedio) y en el segundo paso, se itera sobre el código intermedio para obtener la ejecución final del programa.

Los lenguajes compilados de código intermedio, pueden llegar a ser un orden de magnitud más rápido que los lenguajes completamente interpretados, pero, por consiguiente, un orden de magnitud más lentos que lenguajes optimizados como C o C++ [1].

## 4. Propiedades del Lenguaje Java

Se dice que el código Java es **portable** , debido a que es posible ejecutar el mismo archivo de clase (*.class*), sobre una amplia variedad de arquitecturas de hardware y de software, sin ninguna modificación.

Java es un lenguaje **dinámico**, debido a que las clases son cargadas en el momento en que son necesitadas (dinámicamente), ya sea del sistema de archivos local o desde algún sitio de la red mediante algún protocolo *URL*.

Java tiene la capacidad de aumentar su sistema de tipos de datos dinámicamente o en tiempo de ejecución. Este "enlace tardío" (*late-binding*) significa que los programas sólo crecen al tamaño estrictamente necesario, aumentando así la **eficiencia** del uso de los recursos. Java hace menos suposiciones sobre las implantaciones de las estructuras de datos, que los lenguajes estáticos de "enlace temprano" o en tiempo de compilación (*early-binding*) como C o C++.

Debido a que Java nació en la era post-Internet, fue diseñado con la idea de la **seguridad** y la fiabilidad, por lo que se le integraron varias capas de seguridad para evitar[2] que programas maliciosos pudiesen causar daños en los sistemas, sobre los que ejecuta la implantación de la Máquina Virtual Java.

## 5. La Máquina Virtual Java (MVJ)

La Máquina Virtual Java es el núcleo del lenguaje de programación Java. De hecho, es imposible ejecutar un programa Java sin ejecutar alguna implantación de la MVJ. En la MVJ se encuentra el motor que en realidad ejecuta el programa Java y es la clave de muchas de las características principales de Java, como la portabilidad, la eficiencia y la seguridad.

Siempre que se corre un programa Java, las instrucciones que lo componen no son ejecutadas directamente por el hardware sobre el que subyace, sino que son pasadas a un elemento de software intermedio, que es el encargado de que las instrucciones sean ejecutadas por el hardware. Es decir, el código Java no se ejecuta directamente sobre un procesador físico, sino sobre un **procesador virtual Java**, precisamente el software intermedio del que habíamos hablado anteriormente.

La representación de los códigos de instrucción Java (*bytecode*) es simbólica, en el sentido de que los desplazamientos e índices dentro de los métodos no son constantes, sino que son cadenas de caracteres o nombres simbólicos. Estos nombres son resueltos la primera vez que se ejecuta el método, es decir, el nombre simbólico se busca dentro del archivo de clase (*.class*) y se determina el valor numérico del desplazamiento. Este valor es guardado para aumentar la velocidad de futuros accesos. Gracias a esto, es posible introducir un nuevo método o sobreescribir uno existente en tiempo de ejecución, sin afectar o romper la estructura del código.

En la figura 1 puede observarse la capa de software que implementa a la máquina virtual Java. Esta capa de software oculta los detalles inherentes a la plataforma, a las aplicaciones Java que se ejecuten sobre ella. Debido a que la plataforma Java fue diseñada pensando en que se implementaría sobre una amplia gama de sistemas operativos y de procesadores, se incluyeron dos capas de software para aumentar su portabilidad. La primera dependiente de la plataforma es llamada*adaptador* , mientras que la segunda, que es independiente de la plataforma, se le llama *interfaz de portabilidad*. De esta manera, la única parte que se tiene que escribir para una plataforma nueva, es el adaptador. El sistema operativo proporciona los servicios de manejo de ventanas, red, sistema de archivos, etcétera.

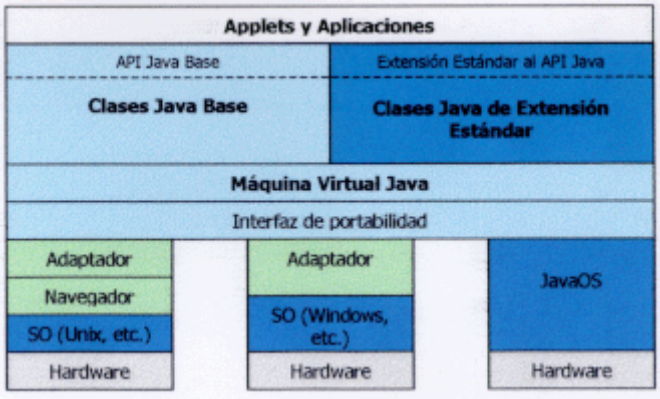


**Figura 1. La Máquina Virtual Implementada para una variedad de plataformas.**

### 5.1. La plataforma Java (Sistema en tiempo de ejecución)

Sun utiliza el término "Máquina Virtual Java", para referirse a la especificación abstracta de una máquina de software para ejecutar programas Java. La especificación de esta máquina virtual, define elementos como el formato de los archivos de clases de Java (*.class*), así como la semántica de cada una de las instrucciones que componen el conjunto de instrucciones de la máquina virtual. A las implantaciones de esta especificación se les conocen como "Sistemas en Tiempo de Ejecución Java". En la figura 2 se puede observar los componentes típicos de un sistema de tiempo de ejecución. Ejemplos de Sistemas de tiempo de ejecución son el Navegador de Nestcape, el Explorador de Microsoft y el programa Java (incluido en el JDK). Un sistema de tiempo de ejecución incluye típicamente:

- Motor de ejecución. El procesador virtual que se encarga de ejecutar el código (bytecode), generado por algún compilador de Java o por algún ensamblador[3] del procesador virtual Java.

- Manejador de memoria. Encargado de obtener memoria para las nuevas instancias de objetos, arreglos, etcétera, y realizar tareas de recolección de basura.

- Manejador de errores y excepciones. Encargado de generar, lanzar y atrapar excepciones.

- Soporte de métodos nativos. Encargado de llamar métodos de C++ o funciones de C, desde métodos Java y viceversa.

- Interfaz multihilos. Encargada de proporcionar el soporte para hilos y monitores.

- Cargador de clases. Su función es cargar dinámicamente las clases Java a partir de los archivos de clase (.class).

- Administrador de seguridad. Se encarga de asegurar que las clases cargadas sean seguras, así como controlar el acceso a los recursos del sistema.

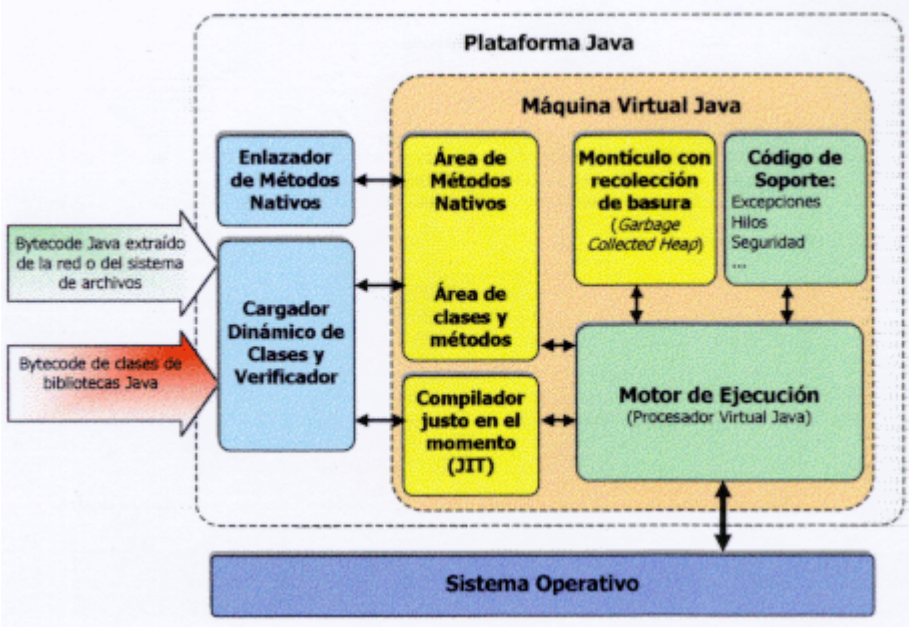


**Figura 2. Arquitectura del Sistema de Tiempo de Ejecución Java.**

Adicionalmente, existe un conjunto de clases Java estándar, fuertemente ligadas a la implantación de cada MVJ en particular. Ejemplos de esto los tenemos en las clases encargadas de funciones, como los accesos a los recursos de la red, manejar el sistema de ventanas, los hilos y el sistema de archivos local. Todos estos elementos en conjunto actúan como una interfaz de alto nivel, para acceder a los recursos del sistema operativo. Es esta interfaz la clave de la portabilidad de los programas Java, debido a que independientemente del hardware o sistema operativo sobre el que se esté trabajando, la máquina virtual Java oculta todas estas diferencias.

A continuación describiremos con mayor detalle cada uno de estos elementos.

### 5.2. Motor de Ejecución

Es la entidad de hardware o software, que ejecuta las instrucciones contenidas en los códigos de operación (*bytecodes*) que implementan los métodos Java. En las versiones iniciales de Sun, el motor de ejecución consistía de un interprete de códigos de operación. En las versiones más avanzadas de nuestros días, se utiliza la tecnología de "generación de código justo en el momento" (*Just-in-Time code generation*), en dónde las instrucciones que implementan a los métodos, se convierten en código nativo que se ejecuta directamente en la máquina sobre la que se subyace. El código nativo se genera únicamente la primera vez que se ejecuta el código de operación Java, por lo que se logra un aumento considerable en el rendimiento de los programas.

### 5.3. El Conjunto de Instrucciones del Procesador Virtual

Muchas de las instrucciones del procesador virtual Java, son muy similares a las que se pueden encontrar para los procesadores comunes y corrientes, como los Intel, es decir, incluyen los grupos de instrucciones típicos como los aritméticos, los de control de flujo, de acceso a memoria, a la pila, etcétera.

Una de las características más significativas del conjunto de instrucciones del procesador virtual Java, es que están basadas en la pila y utilizan "posiciones de memoria" numeradas, en lugar de registros. Esto es hasta cierto punto lógico, debido a que la máquina virtual está pensada para correr sobre sistemas con procesadores sustancialmente diferentes. Es difícil hacer suposiciones sobre el número o tipo de registros que estos pudiesen tener. Esta característica de estar basada en operaciones sobre la pila, impone una desventaja a los programas escritos en Java, contra los lenguajes completamente compilados, como C o C++, debido a que los compiladores de estos pueden generar código optimizado para la plataforma particular sobre la que se esté trabajando, haciendo uso de los registros, etcétera.

Varias de las instrucciones que componen el conjunto de instrucciones del procesador virtual de Java, son bastante más complejas que las que se pueden encontrar en procesadores comunes. Ejemplo de ello, tenemos las casi 20 instrucciones para realizar operaciones, tales como invocar métodos de objetos, obtener y establecer sus propiedades o generar y referenciar nuevos objetos. Es evidente que operaciones de este estilo son de una complejidad considerable y la proyección a sus respectivas instrucciones, sobre el conjunto de instrucciones del procesador de la máquina, implicará a varias decenas de esas instrucciones.

### 5.4. El Verificador de Java

Como hemos mencionado anteriormente, una de las principales razones para utilizar una máquina virtual, es agregar elementos de seguridad a nuestro sistema, por lo que si un intérprete falla o se comporta de manera aleatoria, debido a código mal formado, es un problema muy serio. La solución trivial a este problema sería incluir código encargado de capturar errores y verificar que el código sea correcto. Es evidente que la principal desventaja de esta solución, es que volveremos a caer en un sistema sumamente seguro, pero altamente ineficiente.

Los diseñadores de Java tomaron otro camino. Cuando estaban diseñando el conjunto de instrucciones para la máquina virtual de Java, tenían dos metas en mente. La primera era que el conjunto de instrucciones fuera similar a las instrucciones que se pueden encontrar en los procesadores reales. La segunda era construir un conjunto de instrucciones que fuera fácilmente verificable.

En Java, justo después de que se obtiene una clase del sistema de archivos o de Internet, la máquina virtual puede ejecutar un verificador que se encargue precisamente de constatar que la estructura del archivo de clase es correcta. El verificador se asegura que el archivo tenga el número mágico (0xCAFEBABE) y que todos los registros que contiene el archivo tengan la longitud correcta, pero aún más importante, comprueba que todos los códigos de operación sean seguros de ejecutar. Es importante notar que Java no necesita que el verificador se ejecute sobre el archivo de clase, sino que es activado por el sistema en tiempo de ejecución y sólo sobre clases que el mismo sistema decida. Por lo común, las clases verificadas son las provenientes de Internet.

Aún en nuestros días, los cargadores de clases comerciales tienen varios defectos, por lo que la construcción de mejores verificadores sigue siendo un problema abierto. Por ejemplo, Karsten Sohr, en septiembre de 1999 encontró que el cargador de Microsoft tiene problemas con los tipos de datos, entre los bloques de excepciones, lo que puede provocar forzamientos de conversiones de tipos arbitrarios, comprometiendo la seguridad del sistema, debido a que de esta manera es posible acceder a recursos que debieran estar restringidos.

### 5.5. Administrador de Memoria

Java utiliza un modelo de memoria conocido como "administración automática del almacenamiento" (*automatic storage management*), en el que el sistema en tiempo de ejecución de Java mantiene un seguimiento de los objetos. En el momento que no están siendo referenciados por alguien, automáticamente se libera la memoria asociada con ellos. Existen muchas maneras de implementar recolectores de basura, entre ellas tenemos:

- *Contabilizar referencias*. La máquina virtual Java asocia un contador a cada instancia de un objeto, donde se refleja el número de referencias hacia él. Cuando este contador es 0, la memoria asociada al objeto es susceptible de ser liberada. Aún cuando este algoritmo es muy sencillo y de bajo costo (en términos computacionales), presenta problemas con estructuras de datos circulares.

- *Marcar e intercambiar (Mark-and-Sweep)*. Este es el esquema más común para implementar el manejo de almacenamiento automático. Consiste en almacenar los objetos en un montículo (heap) de un tamaño considerable y marcar periódicamente (generalmente mediante un bit en un campo que se utiliza para este fin) los objetos que no tengan ninguna referencia hacia ellos. Adicionalmente existe un montón alterno, donde los objetos que no han sido marcados, son movidos periódicamente. Una vez en el montículo alterno, el recolector de basura se encarga de actualizar las referencias de los objetos a sus nuevas localidades. De esta manera se genera un nuevo montículo, que contiene únicamente objetos que están siendo utilizados.

Existen muchos otros algoritmos para implementar sistemas que cuenten con recolección de basura. En [6] se puede encontrar una panorámica bastante completa del estado del arte actual a ese respecto.

### 5.6. Administrador de Errores y Excepciones

Las excepciones son la manera como Java indica que ha ocurrido algo "extraño"[4] durante la ejecución de un programa Java. Comúnmente las excepciones son generadas y lanzadas por el sistema, cuando uno de estos eventos ocurre. De la misma manera, los métodos tienen la capacidad de lanzar excepciones, utilizando la instrucción de la MVJ, *athrow*.

Todas las excepciones en Java son instancias de la clase *java.lang.Throwable* o de alguna otra que la especialice. Las clases *java.lang.Exception* y *java.lang.Error*, heredan directamente de java.lang.Throwable. La primera se utiliza para mostrar eventos, de los cuales es posible recuperarse, como la lectura del fin de archivo o la falla de la red, mientras que la segunda se utiliza para indicar situaciones de las cuales no es posible recuperarse, como un acceso indebido a la memoria.

Cuando se genera una excepción, el sistema de tiempo de ejecución de Java, y en particular el manejador (*handler*) de errores y excepciones, busca un manejador para esa excepción, comenzando por el método que la originó y después hacia abajo en la pila de llamadas. Cuando se encuentra un manejador, éste atrapa la excepción y se ejecuta el código asociado con dicho manejador. Lo que ocurre después depende del código del manejador, pero en general, puede suceder que:

- Se utilice un goto para continuar con la ejecución del método original
- Su utilice un return para salir del método
- Se utilice athrow para lanzar otra excepción

En el caso que no se encuentre un manejador para alguna excepción previamente lanzada, se ejecuta el manejador del sistema, cuya acción típica es imprimir un mensaje de error y terminar la ejecución del programa.

Como ya mencionamos, para generar una excepción se utiliza la instrucción *athrow*, que toma un elemento de la pila. Dicho elemento debe ser la referencia a un objeto que herede de la clase *java.lang.Throwable*.

### 5.7. Soporte para Métodos Nativos

Las clases en Java pueden contener métodos que no estén implementados por códigos de operación (*bytecode*) Java, sino por algún otro lenguaje compilado en código nativo y almacenado en bibliotecas de enlace dinámico, como las DLL de Windows o las bibliotecas compartidas SO de Solaris.

El sistema de tiempo de ejecución incluye el código necesario para cargar dinámicamente y ejecutar el código nativo que implementa estos métodos. Una vez que se enlaza el módulo que contiene el código que implementa dicho método, el procesador virtual atrapa las llamadas a éste y se encarga de invocarlo. Este proceso incluye la modificación de los argumentos de la llamada, para adecuarlos al formato que requiere el código nativo, así como transferirle el control de la ejecución. Cuando el código nativo termina, el módulo de soporte para métodos nativos se encarga de recuperar lo resultados y de adecuarlos al formato de la máquina virtual Java.

De manera análoga, el módulo de soporte para código nativo se encarga de canalizar una llamada a un método escrito en Java, hecha desde un procedimiento o método nativo. Para mayor información sobre la interfaz de métodos nativos de Java(*Java native interface*), se puede consultar [10].

### 5.8. Interfaz de Hilos

Java es un lenguaje que permite la ejecución concurrente de varios hilos de ejecución, es decir, el sistema de tiempo de ejecución de Java tiene la posibilidad de crear más de un procesador virtual Java, donde ejecutar diferentes flujos de instrucciones, cada uno con su propia pila y su propio estado local. Los procesadores virtuales pueden ser simulados por software o implementados mediante llamadas al sistema operativo, sobre el cual subyace.

En el conjunto de instrucciones de la máquina virtual Java, sólo existen dos directamente relacionadas con los hilos, *monitorenter* y *monitorexit*, que sirven para definir secciones de código, que deben ejecutarse en exclusión mutua. El resto del soporte de los hilos se realiza atrapando llamadas a los métodos pertenecientes a la clase java.lang.Thread. En [8] puede encontrarse una descripción más detallada sobre la interfaz de hilos de Java, así como un análisis de sus características más importantes.

### 5.9. Cargador de Clases

Los programas Java están completamente estructurados en clases. Por lo tanto, una función muy importante del sistema en tiempo de ejecución, es cargar, enlazar e inicializar clases dinámicamente, de forma que sea posible instalar componentes de software en tiempo de ejecución. El proceso de cargado de las clases se realiza sobre demanda, hasta el último momento posible.

La Máquina Virtual Java utiliza dos mecanismos para cargar las clases. El primero consiste en un cargador de clases del sistema, cuya función es cargar todas las clases estándar de Java, así como la clase cuyo nombre es estrada vía la línea de comandos. De manera adicional, existe un segundo mecanismo para cargar clases dentro del sistema, utilizando una instancia de la clase *java.lang.ClassLoader* o alguna otra definida por el usuario, que especialice a la anterior. Es importante hacer notar que el cargador de clases es uno de los recursos que debe proteger el administrador de seguridad. No debe permitir, por ejemplo, que los *applets* no confiables creen sus propios cargadores debido a que puede ser un punto por el que pueden romperse las restricciones de seguridad.

Los cargadores especializados por los programadores, pueden definir la localización remota de donde se cargarán las clases o asignar atributos de seguridad apropiados para sus aplicaciones particulares. Finalmente, se puede usar a los cargadores para proporcionar espacios de nombres separados a diferentes componentes de una aplicación.

### 5.10. Arquitectura de Seguridad en Java

Java utiliza una serie de mecanismos de seguridad, con el fin de dificultar la escritura de programas maliciosos que pudiesen afectar la integridad de las aplicaciones y los datos de los usuarios. Cada sistema en tiempo de ejecución Java tiene la capacidad de definir sus propias políticas de seguridad, mediante la implantación de un "administrador de seguridad" (*security manager*), cuya función es proteger al sistema de tiempo de ejecución, definiendo el ámbito de cada programa Java en cuanto a las capacidades de acceder a ciertos recursos, etcétera.

El modelo de seguridad original proporcionado por la plataforma Java, es conocido como la "caja de arena" [4] (*sandbox*), que consiste en proporcionar un ambiente de ejecución muy restrictivo para código no confiable que haya sido obtenido de la red. Como se muestra en la figura 3 , la esencia del modelo de la caja de arena, es que el código obtenido del sistema de archivo local es por naturaleza confiable. Se le permite el acceso a los recursos del sistema, como el mismo sistema de archivos o los puertos de comunicación. Mientras, el código obtenido de la red se considera no confiable. Por lo tanto, tiene acceso únicamente a los recursos que se encuentran accesibles desde la caja de arena.

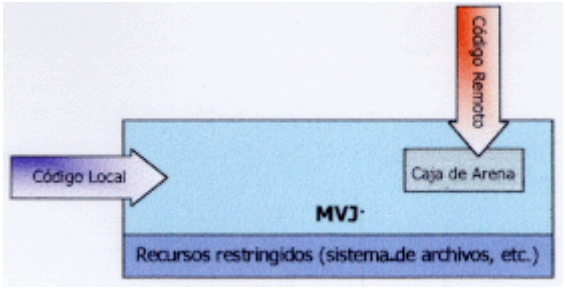


**Figura 3. Modelo de seguridad del JDK 1.0.**

Como hemos mencionado, la máquina implementa otros mecanismos de seguridad, desde el nivel de lenguaje de programación, como la verificación estricta de tipos de datos, manejo automático de la memoria, recolección automática de basura, verificación de los límites de las cadenas y arreglos, etcétera. Todo con el fin de obtener, de una manera relativamente fácil, código "seguro".

En segunda instancia, los compiladores y los verificadores de código intentan asegurar que sólo se ejecuten códigos de ejecución (*bytecodes*) Java, con la estructura correcta y no maliciosos. Asimismo, analizamos cómo con el cargador de clases se pueden definir espacios de nombres locales, lo que ayuda a garantizar que un *applet* no confiable pueda interferir con el funcionamiento de otros programas.

Finalmente, el acceso a los recursos importantes del sistema, es administrado entre el sistema de tiempo de ejecución y el administrador de seguridad (*Security Manager*), que es implementado por la clase *java.lang.SecurityManager*, que permite a las implantaciones incorporar políticas de seguridad. De esta manera, es posible para las aplicaciones determinar si una operación es insegura o contraviene las políticas de seguridad, antes de ejecutarla.

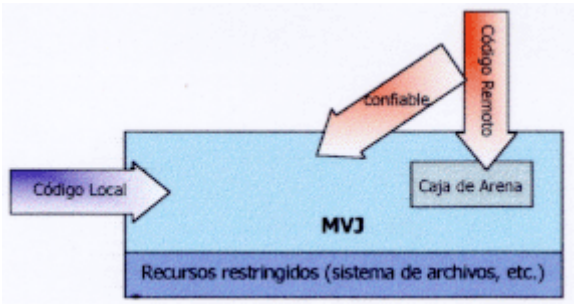


**Figura 4. Modelo de seguridad del JDK 1.1.**

El JDK 1.1 introduce el concepto de "*apple*t firmado" (*signed applet*), en el que los *applets* que poseen una firma digital correcta, son considerados como confiables. Por lo tanto, reciben los mismos privilegios que el código obtenido del sistema de archivos. Los *applets* firmados, junto con la firma, se envían en un archivo de formato JAR (*Java Archive*). En este modelo de applets sin firma, continúan corriendo en la caja de arena. En la figura se puede observar el modelo de seguridad del JDK 1.1.

Finalmente, como se muestra en la figura 5, en la arquitectura de la plataforma de seguridad de Java 2 se introdujeron diferentes niveles de restricción, se eliminó la idea de que el código proveniente del sistema de archivo local siempre es confiable, etcétera. Para mayor información sobre los mecanismos de seguridad de Java consultar [3].

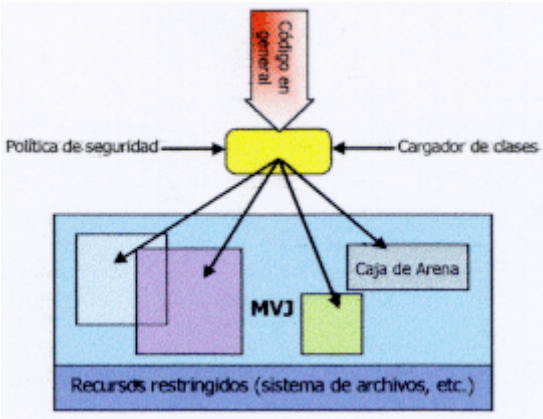


**Figura 5. Modelo de seguridad de Java 2.**

**6. Desventajas de las Máquinas Virtuales**

Una de las razones por que las máquinas virtuales no son la panacea de la computación, es que agregan gran complejidad al sistema en tiempo de ejecución. Por ejemplo, la MVJ espera que la computadora sobre la que subyace, soporte el estándar de IEEE para los números de punto flotante de 32 y 64 bits, así como enteros largos de 64 bits. La mayoría de las plataformas lo hacen pero hay algunas que no, lo que implica trabajo extra.

La principal desventaja de los lenguajes basados en máquina virtual, es que efectivamente son más lentos que los lenguajes completamente compilados, debido a la sobrecarga que genera tener una capa de software intermedia entre la aplicación y el hardware de la computadora. Esta desventaja no es demasiado crítica. Cabe recordar que no hace mucho tiempo sucedió una historia similar entre C y los lenguajes ensambladores. La mayoría de los programadores de ese tiempo (programadores de ensamblador), se resistían al cambio, argumentando que el uso de un lenguaje como C impondría demasiada sobrecarga a sus aplicaciones. Es por esto que no sería raro que, en un futuro próximo, la historia se repita.

### 6.1 Deficiencias de la MVJ

- Conjunto de instrucciones no ortogonal. Un lenguaje de programación es ortogonal, si tiene el mismo número de instrucciones asociadas a cada uno de los tipos de datos.

- Difícil de extender. Debido a que se utiliza un byte para codificar el código de operación de las instrucciones del procesador virtual Java (de ahí el nombre de bytecode), es difícil agregar nuevas instrucciones.

- No posee un árbol de análisis sintáctico. El código intermedio, usado en la MVJ, es simple y plano, es decir no incluye información acerca de la estructura del método original. En un lenguaje completamente compilado, esta información juega un papel muy importante en la optimización, debido a que permite trabajar con el control de dependencias y de flujo.

## 7. Java Realmente Rápido

Aún con un verificador, un conjunto de instrucciones muy bien diseñado y un procesador virtual escrito en C, no se ha logrado obtener un rendimiento realmente alto. Existen varias formas de acelerar la ejecución de las aplicaciones Java, como por ejemplo mejorar los algoritmos para la recolección de basura, asignación de la memoria, ejecución de métodos, etcétera. Además, existen los chips, que tienen la capacidad de ejecutar directamente el código del procesador virtual Java.

El código Java, compilado justo en el momento, tampoco puede competir con el código completamente compilado de C, debido a que los compiladores son diseñados para obtener todas las ventajas posibles de la arquitectura sobre la que subyace, mientras que Java JIT tiene que cargar con las restricciones que impone el diseño del conjunto de instrucciones del procesador virtual Java, que, como ya habíamos mencionado, están basadas en el uso de la pila.

## 8. Unas Palabras Finales

Aún cuando Java proviene de una idea relativamente antigua, ha venido a revolucionar la computación de nuestros días, debido a que integra tres características de suma importancia: la seguridad, la escalabilidad y la portabilidad.

---

[1] Eslogan que emplea Sun: "write once, run anywhere".

[2] Es importante tener en cuenta que es imposible hablar de trminos absolutos en cuanto a seguridad se refiere. Los mecanismos implementados por la mquina virtual Java nicamente mejoran dicha seguridad, pero no presentan una solucin definitiva al problema

[3] As como los procesadores fsicos tienen asociado un lenguaje ensamblador, el procesador virtual Java posee el propio. Ejemplo es Jasmin, que puede encontrarse en http://cat.nyu.edu/meyer/jasmin/resources.html.

[4] Las excepciones pueden indicar eventos de muy diversa ndole, desde errores por accesos indebidos a la memoria, fallos en la red o alcanzar el fin de archivo en una lectura.

**Bibliografía**


[1] Jon Meyer . Troy Downing . (1997) ***Java Virtual Machine***. O'Reilly.

[2] Douglas Kramer. (1996) ***The Java Platform***. JavaSoft .

[3] Lindholm, Tim . Lindholm Yellin. Frank . ***The JavaTM Virtual Machine Specification***. Sun Microsystems, Inc.. Second Edition http://www.javasoft.com/docs/books/vmspec.

[4] http://java.sun.com/products/jdk/1.3/docs/guide/security.

[5] Venners, Bill. ***Inside the Java Virtual Machine***. McGraw-Hill.

[6] Jones, Richard, Wiley, John and Sons. ***Algorithms for Automatic Dynamic Memory Management***. Garbage Collection.

[7] Qiaoyun Li. ***Java Virtual Machine-Present and Near Future***. Sony Corporate.

[8] Menchaca, Méndez Rolando . (1999) ***Los Hilos de la Máquina Virtual Java. Soluciones Avanzadas***. [7(6)].

[9] http://www.kaffe.org.

[10] http://java.sun.com/products/jdk/1.3/docs/guide/jni.