UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Desensamblador de código Java basado en la estructura del archivo CLASS

TESIS

Que para obtener el título de

Ingeniero en Computación

P R E S E N T A

Uriel Rodrigo Nava Velazco

DIRECTOR DE TESIS

Ing. Alejandro Mancilla Rosales

Ciudad Universitaria, Cd. Mx., 2004

Gracias mamá por hacer todo esto posible, por tus sacrificios, por tratar de hacerme un hombre de bien; gran parte de este trabajo te pertenece por todas las noches en vela que pasaste cerca de mi. También agradezco a Dios por obsequiarme a Nina, son mi más grande tesoro, las amo.

ÍNDICE

Índice.

- **Prefacio.**

- **Capítulo 1 Desensambladores y decompiladores.**

- **Capítulo 2 El lenguaje y la plataforma Java.**

- **Capítulo 3 La Máquina Virtual Java.**

- **Capítulo 4 El formato del archivo *.class*.**

- **Índice de tablas.**

PREFACIO

Prefacio.

El lenguaje de programación Java fue bien recibido por la comunidad de desarrolladores de software y por los proveedores de contenido para Internet. Usuarios de la red se beneficiaron de la seguridad de los programas Java y de la independencia de la plataforma. Los desarrolladores que crearon aplicaciones utilizando el lenguaje de programación Java también se beneficiaron desarrollando código una sola vez, sin la necesidad de hacer portables sus aplicaciones para cada software y plataforma.

Para algunos, Java sólo era conocido como una herramienta para crear *applets*. Sin embargo, el lenguaje de programación tiene un gran valor por trabajar en ambientes de redes distribuidas y es muy poderoso en el campo de aplicaciones de propósito general donde la independencia del hardware tiene poca importancia. La facilidad de programación y las características de seguridad ayudan a producir código rápidamente. Algunos errores comunes nunca ocurren con Java debido a características como el recolector de basura (*garbage collector*). Por todas las razones arriba mencionadas, el lenguaje de programación Java se ha vuelto muy popular y muchas organizaciones han adoptando ésta tecnología.

La necesidad de reutilización de código y el mantenimiento han ido aumentando es por ello que en los últimos años las técnicas de reingeniería que permiten obtener código fuente a partir de código compilado también han crecido. Los decompiladores son herramientas que aplican técnicas de reingeniería para generar código fuente a partir de código compilado, pero debido a la dificultad de desarrollar este tipo de herramientas, aunado a la escasa información, se ha ido avanzando lentamente en este campo.

En este trabajo de tesis se desarrolla un desensamblador de código Java, que bautizamos con el nombre de **Marago**. El propósito de éste desensamblador es mejorar la versión de **Sun Microsystems** -se enumeran las ventajas de **Marago** con respecto a **javap-**, además establecer las bases para desarrollar un decompilador que regenere código Java basado en la estructura del archivo *.class*.

Se presentan ejemplos de cómo se traduce código de alto nivel a código ensamblador para la Máquina Virtual Java utilizando el desensamblador **Marago**.

Hablaremos de los ofuscadores que son herramientas de software que tratan de evitar que un código pueda ser decompilado o desensamblado y finalmente proporcionamos el set de instrucciones completo de la Máquina Virtual Java.

CAPÍTULO

1

Capítulo 1 Desensambladores y decompiladores.

El lenguaje de programación Java ha tenido gran auge en los últimos años y su tendencia sigue a la alza; es un lenguaje de alto nivel, compilado e interpretado, razones que lo hacen vulnerable contra la ingeniería en reversa (decompiladores y desensambladores). En el presente capítulo se revisan conceptos sobre compiladores e intérpretes; se muestra un panorama general acerca de los decompiladores y desensambladores, la dificultad de implementación y los aspectos legales que los acompañan. En la última parte del capítulo mostraré cuales son las aplicaciones que surgen como respuesta a la ingeniería en reversa, los ofuscadores.

1.1 ¿Qué es un lenguaje de programación?.

El lenguaje, en su concepto más amplio, es un conjunto de símbolos y sonidos que se utilizan para expresar ideas y sentimientos, en particular; un lenguaje natural como el español es tan amplio y variado que podemos dar órdenes, expresar admiración, sentimientos, etc. Una clase especial de lenguaje más restringido en expresión es el que se utiliza para comunicarse con máquinas como las computadoras, llamados lenguajes de programación.[1]

Dentro de los lenguajes de programación encontramos los lenguajes de alto y bajo nivel. Los lenguajes de bajo nivel son cercanos a lenguajes de máquina y tienen una fuerte correspondencia con las operaciones implementadas en el hardware subyacente.

Los lenguajes de alto nivel son cercanos a los lenguajes usados por los humanos al expresar problemas y algoritmos, cada enunciado en un lenguaje de alto nivel puede ser equivalente a varios enunciados en un lenguaje de bajo nivel, los lenguajes de alto nivel tienen el propósito de facilitar la tarea de codificar un algoritmo, la ventaja clave de estos lenguajes es la abstracción, proceso mediante el cual se extraen las propiedades esenciales requeridas para la solución de un problema, mientras que se esconden los detalles de la implementación de la solución.

Conforme el nivel de abstracción se incrementa, el programador debe preocuparse cada vez menos del hardware en que se ejecuta el programa; los lenguajes de alto nivel se pueden dividir en dos grupos.

[1] Un lenguaje de programación es un lenguaje formal que sirve para escribir programas.

- Lenguajes imperativos o procedurales como : C, Pascal o Basic.
- Lenguajes declarativos como Prolog, Lisp o Scheme.

Los lenguajes imperativos son aquellos en los cuales la acción de un programa se define por una serie de operaciones ejecutadas en una secuencia definida por el programador. Para resolver un problema, el programador debe especificar una serie de pasos o enunciados que son ejecutados en secuencia.

Los lenguajes declarativos, por su parte, involucran la especificación del conjunto de reglas que definen la solución al problema; depende de la computadora o el intérprete determinar como alcanzar la solución a partir de las reglas dadas.

1.2 Compiladores e intérpretes.

Un compilador es un programa o grupo de programas que traducen un lenguaje de programación a otro –en este caso, el código fuente de un lenguaje de alto nivel es transformado a un lenguaje ensamblador o lenguaje objeto-. Un intérprete es un programa que traduce y ejecuta al mismo tiempo un programa escrito en lenguaje de alto nivel.

> "Un compilador es un software que lee un programa escrito en un lenguaje, el *lenguaje fuente*, y lo traduce a un programa equivalente en otro lenguaje, el *lenguaje objeto*".[2]

> "Un programa que traduce un lenguaje a otro, un código fuente de alto nivel es transformada a un lenguaje ensamblador".[3]

> "Un interprete en lugar de producir un programa objeto como resultado de una traducción, realiza las operaciones que implica el programa fuente".[4]

[2] Compiladores, Principios, técnicas y herramientas; Aho, V. Alfred; Ullman, Jeffrey; ed Pearson 1998 pag1.
[3] Compiler Design in C; Holub, A.I. ed Prentice Hall 1990. Pág. 1
[4] Compiladores, Principios, técnicas y herramientas; Aho, V. Alfred; Ullman, Jeffrey; ed Pearson 1998 pag3.

La diferencia entre un compilador y un intérprete es que el primero revisa que todo el código fuente esté correcto sintácticamente y semánticamente, si no existen errores, crea el código objeto; mientras que el intérprete lee una instrucción del código fuente y la ejecuta.
El proceso de compilación se divide en dos partes: análisis y síntesis. La parte del análisis separa el programa fuente en sus elementos componentes y crea una representación intermedia. La parte de la síntesis construye un programa objeto deseado a partir de la representación intermedia.

Conceptualmente, un compilador trabaja en *fases*, cada una, transforma el programa fuente de una representación en otra. El compilador, en general, como se muestra en la figura 1-1 se compone de las siguientes fases:

***Scanner* o analizador léxico.** Separa el programa fuente en *tokens* que son fragmentos de texto o unidades básicas como palabras reservadas, variables, símbolos aritméticos de relación, etc. Además quita comentarios, procesa directivas de control y almacena los *tokens* en la tabla de símbolos. Los analizadores léxicos toman como entrada una secuencia de expresiones regulares que describen a los *tokens* y producen un solo autómata finito que reconoce a cualquier *token*.

***Parser* o analizador sintáctico**. Verifica el correcto agrupamiento de los *tokens* en frases gramaticales, que se representan para su análisis en un *árbol de parsing.* El analizador sintáctico determina si una cadena de componentes léxicos puede ser generada por una gramática. La mayoría de los analizadores sintácticos están comprendidos en dos clases, llamadas analizadores ascendentes y analizadores descendentes. Estos términos hacen referencia al orden en que construyen los nodos del árbol de análisis sintáctico.

Analizador semántico. Verifica la congruencia de las expresiones o el significado del programa localizando errores. La fase de análisis semántico revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos de datos para la fase posterior de generación de código. En ella se utiliza la estructura jerárquica determinada por la fase de análisis sintáctico para identificar los operadores y operandos de expresiones y proposiciones.

La parte de la síntesis esta formada por:

Generador de código intermedio. Produce código para una máquina abstracta que sea fácil de simplificar y traducir. Aunque un programa fuente se puede traducir directamente al lenguaje objeto, algunas de las ventajas de utilizar una forma intermedia de código es que se puede crear un compilador para una máquina distinta, uniendo una etapa final para a nueva máquina y además se puede aplicar a la representación intermedia un optimador de código independiente de la máquina.

Optimizador de código. El código intermedio se analiza y transforma en código equivalente que es más eficiente. A menudo se puede lograr que el código directamente producido por los algoritmos de compilación se ejecute más rápidamente o que ocupe menos espacio o ambas cosas. Esta mejora se consigue mediante transformaciones de programas que se denominan **optimaciones**. Los compiladores que aplican transformaciones para mejorar el código se denominan **compiladores optimadores**.

Generador de código. La fase final del modelo del compilador produce código en ensamblador o código ejecutable. Toma como entrada una representación intermedia del programa fuente y produce como salida un programa objeto equivalente.

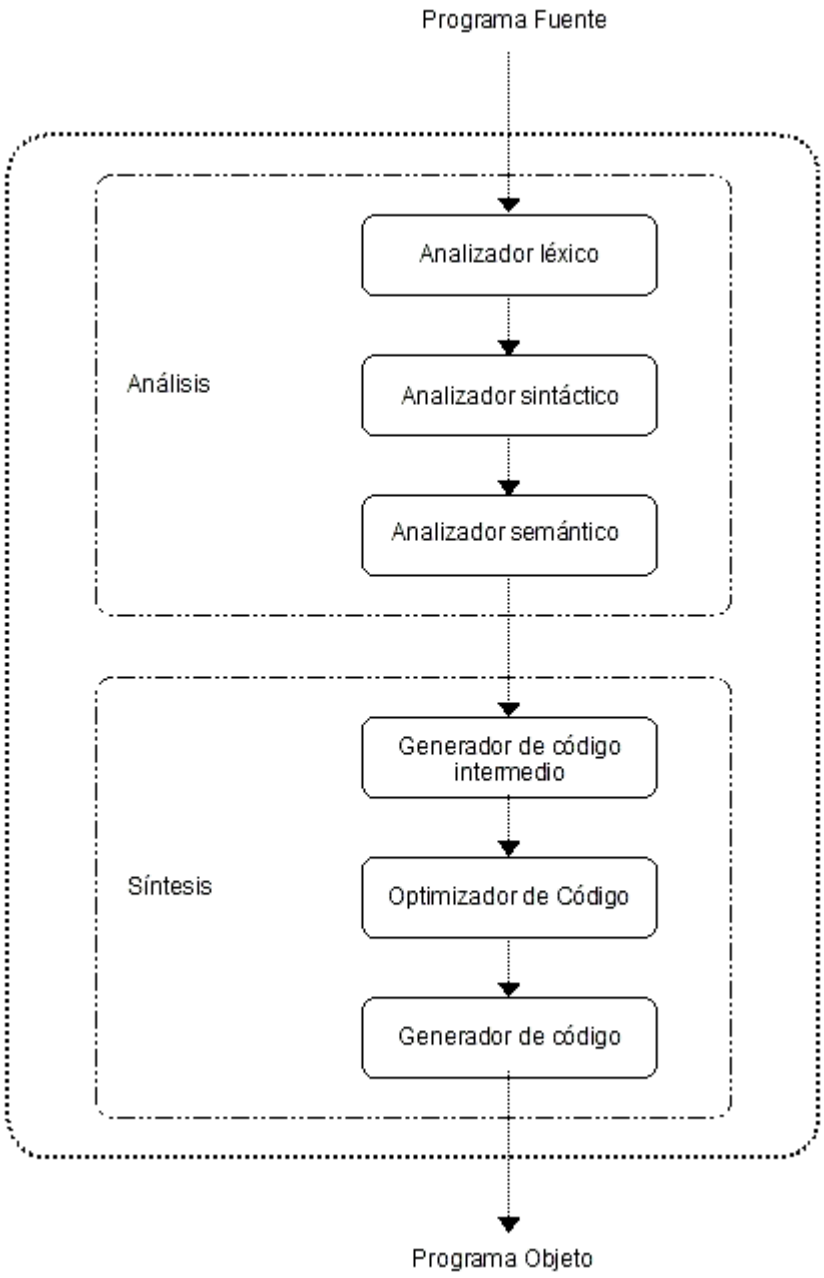

Figura 1-1 Fases de un compilador.

1.2.1 Analizador léxico.

El analizador léxico es la primera fase de un compilador, su principal función consiste en leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utiliza el analizador sintáctico para realizar su función, ver figura 1-2.

Como el analizador léxico es la parte del compilador que lee el texto fuente, también puede realizar ciertas funciones secundarias en la interfaz de usuario, como eliminar del programa fuente comentarios y espacios en blanco en forma de caracteres de espacio en blanco, caracteres TAB y de línea nueva. Otra función es relacionar los mensaje de error del compilador con el programa fuente. Por ejemplo, el analizador léxico puede tener localizado el número de caracteres de nueva línea detectados, de modo que se pueda asociar un número de línea nueva con un mensaje de error. En algunos compiladores el analizador léxico se encarga de hacer una copia del programa fuente en el que están marcados los mensajes de error; si el lenguaje fuente es la base de algunas funciones de preprocesamiento de macros, entonces, esas funciones del preprocesador también se pueden aplicar al hacer el análisis léxico.

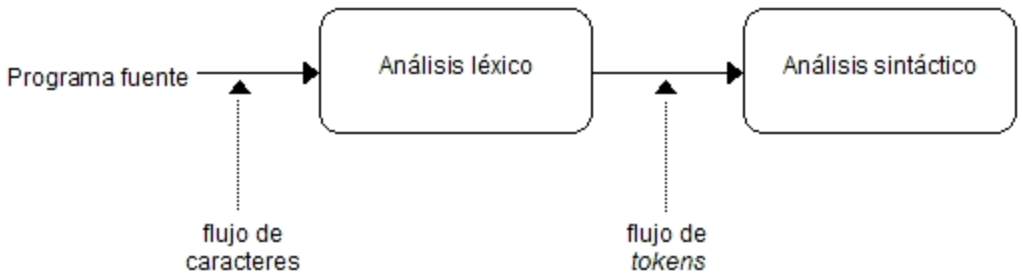

Figura 1-2 Analizador léxico.

En algunas ocasiones, los analizadores léxicos se dividen en una cascada de dos fases: la primera, llamada “examen”, y la segunda, “análisis léxico”. El examinador se encarga de realizar tareas sencillas, mientras que el analizador léxico es el que realiza las operaciones más complejas. Por ejemplo, un compilador de FORTRAN puede utilizar un examinador para eliminar espacios en blanco de la entrada.

1.2.1.1 Tokens.

Un *token* es una secuencia de caracteres que se pueden tratar como unidades en la gramática de un lenguaje de programación:

- identificadores (nombre de variables, funciones, etc).
- constantes (0.234, 3.712, etc).
- operadores (*, -, /, <. ==, etc).
- delimitadores (espacios en blanco, tabuladores, etc).
- palabras reservadas (`while`, `int`, `void`, etc).

1.2.2 Analizador sintáctico.

El analizador sintáctico obtiene una cadena de componentes léxicos o *tokens* del analizador léxico y comprueba si la cadena puede ser generada por la gramática del lenguaje fuente. El analizador sintáctico se encarga de informar de cualquier error de sintaxis de manera inteligible, también deberá recuperarse de los errores que ocurren frecuentemente para poder continuar con el procesamiento del resto de su entrada.

Los métodos universales de análisis sintáctico, como el algoritmo **Cocke –Younger – Kasami** y el de **Earley**[5], pueden analizar cualquier gramática, sin embargo, son demasiado ineficientes para usarlos en la producción de compiladores, los métodos empleados generalmente en los compiladores se clasifican como descendentes o ascendentes.

Como su nombre lo indica los **analizadores sintácticos descendentes** construyen árboles de análisis sintáctico desde arriba (la raíz) hasta abajo (las hojas), mientras que los **analizadores sintácticos ascendentes** comienzan en las hojas y suben hacia la raíz.

Un compilador debe comprobar si el programa fuente sigue las convenciones sintácticas y las semánticas del lenguaje fuente. Esta comprobación, llamada comprobación estática, garantiza la detección y comunicación de algunas clases de errores de programación, los ejemplos de comprobación estática incluyen:

- **Comprobación de tipos**. Un compilador debe informar de un error si se aplica un operador a un operando incompatible; por ejemplo, si se suman una variable tipo matriz y una variable de función.

- **Comprobación del flujo de control**. Las proposiciones que hacen que el flujo de control abandone una construcción deben tener algún lugar a donde transferir el flujo de control, Por

[5] Para una descripción detallada de los algoritmos consulte: Compiladores, Principios, técnicas y herramientas; Aho, V. Alfred; Ullman, Jeffrey; ed Pearson.

ejemplo, una proposición `break` en C hace que el control abandone la proposición `while`, `for` o `switch` más cercana, si dicha proposición que envuelve a la sentencia no existe, ocurre un error.

- **Comprobación de unicidad.** Hay situaciones en que se debe definir un objeto una vez exactamente. Por ejemplo, en Pascal, un identificador debe declararse de forma única, las etiquetas de una proposición `case` deben ser diferentes y no se pueden repetir los elementos de un tipo escalar.

- **Comprobaciones relacionadas con nombres.** En ocasiones, el mismo nombre debe aparecer dos o más veces, Por ejemplo, en Ada, un lazo o bloque puede tener un nombre que aparezca al principio y al final de la construcción. El compilador debe comprobar que se utilice el mismo nombre en ambos sitios.

1.2.3 Análisis semántico.

La fase de análisis semántico revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos de datos para la fase posterior de generación de código. En ella, se utiliza la estructura jerárquica determinada por la fase de análisis sintáctico para identificar los operadores permitidos por la especificación del lenguaje fuente. Por ejemplo, las definiciones de muchos lenguajes de programación requieren que el compilador indique un error cada vez que se use un número real como índice de una matriz. Sin embargo, la especificación del lenguaje puede permitir ciertas coerciones a los operandos, por ejemplo, cuando un operador aritmético binario se aplica a un número entero y a un número real, en este caso, el compilador puede necesitar convertir el número entero a real.

1.2.4 Generación de código.

En el modelo de análisis y síntesis de un compilador, la etapa inicial traduce un programa fuente a una representación intermedia, a partir de la cual, la etapa final genera el código objeto. Los detalles del lenguaje objeto se confinan en la etapa final, si esto es posible. En la fase final del modelo del compilador se toma como entrada una representación intermedia del programa fuente y produce como salida un programa objeto equivalente.

La entrada para el generador de código consta de la representación intermedia del programa fuente producida en la etapa inicial, junto con la información de la tabla de símbolos que se utiliza para determinar las direcciones durante la ejecución de los objetos de datos denotados por los nombres de la representación intermedia.

Se asume que antes de la generación del código, la etapa inicial, ha hecho los análisis léxicos, sintácticos y traducido el programa fuente a una representación intermedia razonablemente detallada, así que los valores de los nombres que aparecen en el lenguaje intermedio pueden ser representados por cantidades que la máquina objeto puede manejar directamente, también se supone

que ha tenido lugar la comprobación de tipos necesarios; de modo que los operadores de conversión de tipos ya se han insertado donde fue necesario. La salida del generador de código es el programa objeto. Al igual que el código intermedio, esta salida puede adoptar una variedad de formas.

1.3 Desensambladores y decompiladores.

Algunos compiladores producen código ensamblador, que se pasa a un ensamblador para su procesamiento, otros compiladores realizan el trabajo del ensamblador, produciendo código de máquina relocalizable que se puede pasar al editor de carga y enlace. El **código ensamblador** es una versión mnemotécnica del código de máquina, donde se usan nombres en lugar de código binarios para operaciones y también se usan nombres para las direcciones de memoria.

Un **desensamblador** es una herramienta de software que obtiene código ensamblador a partir de un código objeto, en particular, un desensamblador para el lenguaje Java obtiene código ensamblador para la Máquina Virtual Java a partir del archivo de *bytecode* o archivo *.class*

A continuación se muestra el programa `Suma.java`, que presenta el resultado de la suma de dos números enteros.

```
public class Suma
{
        public static void main (String[] args)
        {
               int a = 10;
               int b = 20;
               System.out.println( a + b );
        }
}
```

El código desensamblado que se muestra en la figura 1-3, fue obtenido con el desensamblador **Marago**. La versión que desarrollamos en esta tesis, es un desensamblador mejorado con respecto al que **Sun Microsystems** distribuye en su paquete de desarrollo (JDK), el desensamblador **javap**.

Marago no sólo desensambla el código *bytecode*, además genera el *constant_pool*, estructura de la que se hablará en el capítulo 4 ampliamente.

```
/*********************************/
//   National University of Mexico
//   School of Engineering
//
//   Disassembler by
//   Marago
//   Thesis
//   Uriel Nava
/*********************************/


import java.io.PrintStream;

public class Suma
{
 public Suma()
 {
  0 aload_0
  1 invokespecial #1 <Method: void Object()> <nameClass: Object>
  4 return
 }

 public static void main (String[] string0)
 {
  0 bipush 10
  2 istore_1
  3 bipush 20
  5 istore_2
  6 getstatic #2 <Field PrintStream out>
  9 iload_1
  10 iload_2
  11 iadd
  12 invokevirtual #3 <Method: void println(int int0)> <nameClass: PrintStream>
  15 return
 }
}
```

Figura 1-3 Archivo desensamblado por **Marago**.

El desensamblador **Marago** obtiene dos métodos; `public Suma()` y `public static void main(String[] string0)`, el primero, es el constructor de la clase, observe que el archivo fuente `suma.java` no declara ningún constructor; la Máquina Virtual Java lo genera implícitamente y el segundo método aparece porque es una aplicación, recuerde que los *applets* no contienen método `main`.

El método `main` del programa `Suma.java` declara dos valores de tipo entero `a` y `b`.

```
int a = 10;
int b = 20;
```

Se puede observar en la figura 1-3 el código desensamblado que declara esos valores enteros:

```
0 bipush 10
2 istore_1
3 bipush 20
5 istore_2
```

La instrucción `bipush` inserta en la pila (*java stack*) el valor entero 10 y la instrucción `istore_1` saca el valor de la pila y lo asigna a la variable uno, lo mismo sucede con `bipush 20` que inserta el valor entero 20 en la pila y la instrucción `istore_2` asigna el valor de la pila en la variable dos.

Finalmente el programa `Suma.java` muestra el resultado de la suma con la línea:

```
System.out.println( a + b );
```

Esta operación en el código desensamblado se hace en varios pasos.

```
9 iload_1
10 iload_2
11 iadd
12 invokevirtual #3 <Method: void println(int int0)>  <nameClass: PrintStream>
```

La instrucción `iload_1` copia el valor de la variable uno y la inserta en la pila y la instrucción `iload_2` copia el valor de la variable dos y lo inserta en la pila –después de esta instrucción se tiene dos valores en la pila-. La instrucción `iadd` saca los dos valores de la pila, los suma y el resultado lo vuelve a insertar. La instrucción `invokevirtual` llama al método `println` y muestra el resultado de la suma que se encuentra en la pila.

De esta forma, con el desensamblador podemos darnos cuenta qué esta ocurriendo con nuestro programa y saber de cierta forma cómo fue escrito el programa.

Un **decompilador** es una herramienta de software que intenta el proceso de compilación en reversa mediante la traducción de una entrada de programa binario puro a su equivalente en un programa en lenguaje de alto nivel. El programa de entrada no tiene información simbólica dentro de él y el lenguaje de alto nivel usado para compilar el programa binario no es necesariamente el mismo que el producido por el decompilador.

Aunque los decompiladores no han sido estudiados ampliamente y su literatura es escasa, existe una variedad de aplicaciones que pueden beneficiarse de ellos, incluyendo el mantenimiento de viejo código y recuperación de código perdido, también para depurar algún programa binario, migración de aplicaciones a nuevos ambientes de hardware, verificación de generación de código del compilador, y traducción de código escrito en un lenguaje obsoleto.

La estructura de un decompilador esta basada en la de un compilador, los principios y técnicas similares son usadas para ejecutar el análisis del programa.

El proceso de ingeniería en reversa y decompilación o desensamble de programas requiere tecnología sofisticada y debido a que los aspectos legales están fuertemente ligados a este tema; el desarrollo se vuelve más complicado, aunado a la dificultad implícita del desarrollo de un compilador que involucra el conocimiento de lenguajes de computación, arquitectura de computadoras, teoría de lenguajes, algoritmos y de ingeniería de software.

1.4 Aspectos legales.

Uno de los aspectos más importantes en el mundo del desarrollo de software es la propiedad intelectual y su alcance legal sobre la protección otorgada a los programas de computadora. La protección legal para el software envuelve muchos aspectos –firma que desarrolla el producto, derechos de autor, patentes, etc–, pero en particular, el aspecto que envuelve a la ingeniería en reversa es el de la propiedad intelectual.

Un software es una colección de elementos asociados con el desarrollo y operación de programas de computadora, pero no incluye la parte física de la computadora. Existen técnicamente varios elementos en un software de computadora los cuales forman parte del programa:

- Diseño de programas: diagramas de flujo, formas, reportes.
- Código fuente y objeto.
- Lenguaje de programación.
- Algoritmos.
- Manuales.
- Interfaz de usuario.
- Interfaz técnica.

Para propósitos de la ley y el derecho de autor en particular, no todos los elementos conciernen a ésta. Para conveniencia se dividen los elementos de un programa de computadora en tres partes –elementos protegidos, elementos sin proteger y elementos los cuales cambian su estado legal dependiendo del contexto en que se traten.

1.4.1 Elementos protegidos.

- **Algoritmos**. Considerado por la industria de la computación como un trabajo literario o artístico, es una parte muy valiosa que se encuentra detrás de un programa de computadora, un algoritmo envuelve un trabajo de investigación y de desarrollo. Una reproducción sin autorización de una parte substancial del algoritmo infringe los derechos de autor; sin embargo el alcance de la protección es limitada; “un programa de computadora desarrollado

en código fuente u objeto no infringe los derechos de autor del algoritmo, aun si el programa fue desarrollado utilizando un algoritmo sin consentimiento"[6].

- **Código fuente y objeto.** Son considerados elementos literales del programa y son definidos como " Un conjunto de enunciados o instrucciones que son usados directa o indirectamente en una computadora para obtener cierto resultado".

- **Interfaz de usuario.** La interfaz de usuario de un programa de computadora comprende todos los elementos utilizados para que el usuario interactúe con el programa; cualquier cambio en la apariencia de la interfaz infringe las leyes sobre los derechos de autor.

1.4.2 Elementos sin protección.

La función principal de un programa es un elemento que no tienen protección, es decir; las ideas y el funcionamiento básico de un programa así como el propósito de los métodos no están protegidos por la ley.

1.4.3 Elementos sin decisión.

Los comandos como los menús que permiten al usuario la funcionalidad del programa. En un procesador de texto, el menú que aparece en la parte superior con palabras como FILE EDIT VIEW INSERT HELP. Todos los elementos que se pueden acceder mediante menús o submenús o las tareas como copiar, cortar y pegar no están definidos claramente por la ley como elementos protegidos o no, depende del ámbito del programa.[7]

1.4.4 Ingeniería en reversa.

La **ingeniería en reversa** es el proceso mediante el cual las ideas y principios contenidos y expresados en un programa de computadora (cualquiera que sea su forma), son hechos transparentes y disponibles a través de operaciones como desensamblado y decompilación de programas. Ésta traducción del código objeto en otra forma de código podría infringir los derechos de autor de los programas de computadora, a menos que el dueño expresamente o implícitamente lo autoricen.

La ingeniería en reversa es importante en la industria de la computación, particularmente, con respecto al desarrollo de programas.

---

6 IP protection and reverse engineering of computer programs in the US and the EU. Charles R. McManis.

7 Software Copyright Protection: Can Europé Learn Form American Case Law, part 2, [2000] Derclaye Estelle, E.I.P.R., ed. Sweet & Maxwell.

Pero qué objeto tiene desarrollar un software que pueda desensamblar o decompilar programas, si al hacerlo se infringe la ley. Cuando se piensa desarrollar un desensamblador, decompilador o herramienta que permita obtener código fuente, no se hace pensando en desensamblar todos los programas que nos encontremos a nuestro paso, de hecho, en la mayoría de los casos es más sencillo desarrollar un programa de computadora desde cero, que tratar de modificar código ya escrito; en muchos casos decompilar programas, lejos de ayudarnos a entender la solución sólo complican más nuestros problemas debido a la dificultad de entender código que no hemos programado nosotros mismos.

Los fines para los que se desarrollan programas como los desensambladores y decompiladores pueden ser muy variados. Imaginemos que en una empresa que se dedica a desarrollar software ocurre una catástrofe – puede ser un virus, fallas de luz, hasta un terremoto- y pierde todos sus archivos fuente.

Sin duda sería una gran perdida para la empresa, ya que tendría que volver a programar todas sus aplicaciones, pero supongamos que tiene acceso a los códigos ejecutables; en este caso la utilización de un decompilador o desensamblador le sería de gran utilidad ya que le permitiría recuperar la mayoría de sus códigos fuente.

Otros aspectos como el mantenimiento, depuración, migración de código e inclusive para enseñar a programar involucran a la ingeniería en reversa.

1.5 Ofuscadores.

La idea detrás de la protección técnica del software es hacer económicamente impractica a la ingeniería en reversa. El objetivo es transformar una aplicación para que sea más difícil de entender, pero idéntica en su funcionalidad a la original. El desarrollo del software usa una utilería llamada ofuscador para proteger su aplicación.

Un **ofuscador** es un programa usado para transformar código objeto. La salida de un ofuscador es un programa objeto que es más difícil de entender pero funcionalmente es equivalente al original. Para lograr esto, los ofuscadores aplican transformaciones al programa objeto, como el ofuscador para Java, **Crema**.[8]

Los ofuscadores asumen que los programas original y ofuscado tienen el mismo comportamiento, pero en realidad los programas transformados son más lentos y más grandes que los originales, aunque el comportamiento de ambos programas debe ser idéntico

El código ofuscado no provee una aplicación con absoluta protección contra ataques maliciosos de ingeniería en reversa. El nivel de seguridad depende de la sofisticación de la transformación usada por el ofuscador y el poder, las herramientas disponibles y los recursos de la ingeniería en reversa.

[8] Hanpeter Van Vliet escribió el primer decompilador para Java llamado Mocha y también escribió el primer ofuscador llamado Crema, desafortunadamente falleció de cáncer a la edad de 34 años.

CAPÍTULO

2

Capítulo 2 El lenguaje y la plataforma Java.

Java es parte del conjunto integrado de sistemas que brinda soporte a la comunicación en el Web. El nombre de Java se refiere al lenguaje de programación pero también se refiere a un grupo de herramientas de software que permite crear e implantar contenido ejecutable utilizando el lenguaje de programación Java. En este capítulo se describen las partes que integran la plataforma Java y la forma en que trabaja el lenguaje de programación Java. Conocer a fondo la plataforma Java, la Máquina Virtual Java y el formato del archivo *.class* –que se describen en los siguientes capítulos- permitirán abordar el proceso de obtención de código Java a partir de *bytecode* de una manera más clara.

2.1 Orígenes de Java.

El lenguaje de programación Java fue diseñado para hacer frente a los retos de aplicaciones desarrolladas en ambientes de red, dentro de estos retos, destacan la seguridad en las aplicaciones, que éstas consuman el mínimo de recursos del sistema y puedan ejecutarse en cualquier hardware y plataforma de software.

El lenguaje de programación Java fue parte de un proyecto de investigación para desarrollar software avanzado para una amplia variedad de dispositivos de red y sistemas embebidos (*embedded systems*)[1]. La meta fue desarrollar una plataforma que operara en tiempo real, portable y distribuida. Cuando el proyecto comenzó, C++ fue el lenguaje seleccionado para desarrollar las herramientas, pero con el tiempo las dificultades encontradas en ese lenguaje de programación fueron creciendo al punto que era mucho más fácil crear una plataforma con un lenguaje totalmente nuevo. El diseño y la arquitectura fueron decisiones que se tomaron de una variedad de lenguajes como Eiffel, SmallTalk, C y Cedar/Mesa. El resultado fue un lenguaje plataforma que provee un ambiente ideal para desarrollar aplicaciones seguras, distribuidas y basadas en ambientes de red.

[1] Un sistema embebido, también conocido como sistema incorporado, es una combinación de hardware, software y eventualmente, componentes mecánicos diseñados para realizar una función específica.

En Diciembre de 1990, Patrick Naughton, un empleado de la empresa **Sun Microsystems**, reclutó a sus colegas **James Gosling** y Mike Sheridan para trabajar sobre un nuevo tema conocido como "**El proyecto verde"** que tenía como objetivo principal crear un lenguaje de programación accesible, fácil de aprender y de usar, que fuera universal y estuviera basado en un ambiente C++ ya que había mucha frustración por la complejidad y las limitaciones de los lenguajes de programación existentes.

En abril de 1991, el equipo decidió introducir un sistema de software con aplicación para consumidores *smart*[2]*;* como plataforma de lanzamiento para su proyecto. **James Gosling** escribió el compilador original y lo denominó "**Oak**" y con la ayuda de los otros miembros del equipo desarrollaron un decodificador que más tarde se convertiría en lenguaje Java[3].
Para 1992, el equipo ya había desarrollado un sistema prototipo conocido como "*7", (**Star Seven**); el sistema presentaba una interfaz con la forma de una casa y el control se llevaba a cabo mediante una pantalla sensible al tacto, vea figura 2-1. Posteriormente se aplico a otro sistema denominado **VOD** (*Video On Demand*) en el que se empleaba como interfaz para la televisión interactiva. Ninguno de estos proyectos se convirtió nunca en un sistema comercial, pero fueron desarrollados completamente en un Java primitivo. Por su parte, el presidente de la compañía **Sun Microsystems**, Scott McNealy, se dio cuenta en forma muy oportuna y estableció el Proyecto Verde como una subsidiaria de **Sun Microsystems**.

De 1993 a 1994, el equipo de Naughton se lanzó en busca de nuevas oportunidades en el mercado, mismas que se fueron dando mediante el sistema operativo base; sin embargo la subsidiaria fracasaría en su intento por ganar una oferta con Time – Warner lo que la llevaría a cancelar el proyecto verde.

Figura 2-1 Sistema *7 *Star Seven.*

Afortunadamente, el cese del Proyecto Verde coincidió con el nacimiento del fenómeno mundial Web. Al examinar las dinámicas de Internet, lo realizado por el ex equipo verde se adecuaba a este nuevo ambiente ya que cumplía con los mismos requerimientos de las *set-top box* OS que estaban diseñadas con un código de plataforma independiente pero sin dejar de ser pequeñas y confiables.

[2] La tecnología de Java puede desplegarse en sistemas incrustados como los aparatos electrónicos de consumo: dispositivos manuales, teléfonos y videocaseteras.
[3] En Estados Unidos al café también se le conoce como java; James Gosling quería darle a su lenguaje un nombre que transmitiera la idea de *energía*, como la cafeína.

Patrick Naugthon procedió a la construcción del lenguaje de programación Java que se accionaba con un navegador prototipo, más tarde se le fueron incorporando algunas mejoras y el navegador Hot Java fue dado a conocer al mundo en 1995.

Con el paso del tiempo el navegador Hot Java se convirtió en un concepto práctico dentro del lenguaje Java; demostró que podría proporcionar una forma segura de ejecutar código, con la ventaja de ser multiplataforma, es decir; que la aplicación puede ejecutarse en cualquier computadora sin importar el sistema operativo que la controla.

Una de las características más atractivas del Hot Java fue su soporte para los "*applets*", que son las partes del código Java que pueden ser cargadas mediante una red de trabajo para después ejecutarlo localmente y así lograr o alcanzar soluciones dinámicas en computación acordes al rápido crecimiento del ambiente Web.

Para dedicarse al desarrollo de productos basados en la tecnología Java, **Sun Microsystems** formó la empresa **Java Soft** en enero de 1996, de esta forma se dio continuidad al fortalecimiento del programa del lenguaje Java y así comenzar a trabajar con socios para crear aplicaciones, herramientas, sistemas de plataforma y servicios para aumentar las capacidades del lenguaje.

Durante ese mismo mes, **Java Soft** dio a conocer *Java Developmet Kit* (paquete de desarrollo Java, JDK) 1.0, una rudimentaria colección de componentes básicos para ayudar a los usuarios de software a construir aplicaciones de Java. Dicha colección incluía el compilador Java, un visualizador de *applets*, un *debugger* prototipo y una Máquina Virtual Java (*Java Virtual Machine*, JVM), necesarias para ejecutar aplicaciones basadas en Java, también incluía paquetería básica de gráficos, sonido, animación y trabajo en red.

Al mismo tiempo, **Netscape Comunications Inc**, mostró las ventajas de Java y rápidamente se asoció con **Java Soft** para explotar su nueva tecnología. No pasó mucho tiempo antes de que **Netscape Communications** decidiera apoyar a los Java *applets* en **Netscape Navigator 2.0**. Éste fue el factor clave que lanzó a Java a ser reconocido y famoso; y que a su vez forzó a otros vendedores para apoyar el soporte de *applets* en Java.

Para darnos una idea de la rápida aceptación que tiene Java en el mundo, tenemos el ejemplo de las conferencias "*Java Soft Java One*" en San Francisco, el primer *Java One* fue celebrado en abril de 1996 y atrajo a 5000 usuarios, un año después, en la segunda conferencia, *Java One* albergó a 10,000 usuarios, asistentes. **Java Soft** estima que el número actual de usuarios Java llega a 400 mil y sigue creciendo. Java también está ganando aceptación en el área empresarial, se ha estimado que actualmente las compañías de hoy que cuentan con más de 5000 empleados, una tercera parte están usando Java.

Figura 2-2 Duke es la mascota oficial de la plataforma Java.

2.2 El lenguaje de programación Java.

Java es un lenguaje de programación con el que los programadores pueden crear programas semejantes a los que se pueden desarrollar en C++, así como los *applets* que son una mini (*let*) aplicación (*app*) diseñada para ejecutarse dentro de un navegador. La compañía **Sun Microsystems** da una definición formal del lenguaje de programación Java:

> "Java es un lenguaje de programación de alto nivel, simple, orientado a objetos, distribuido, interpretado, sólido, seguro, de arquitectura neutral, portable, de alto desempeño, multihilos y dinámico"[4]

A continuación, se explica detalladamente cada una de las características del lenguaje de programación Java.

2.2.1 Simple.

Java ofrece toda la funcionalidad de un lenguaje potente, pero sin las características menos usadas y más confusas de éstos. C++ no es un lenguaje conveniente por razones de seguridad, pero C y C ++ son los lenguajes más difundidos, por ello Java se diseño para ser parecido a C++ y así facilitar un rápido y fácil aprendizaje.

Java elimina muchas de las características de otros lenguajes como C++, para mantener reducidas las especificaciones del lenguaje y añadir propiedades muy útiles como el recolector de basura[5] (reciclador de memoria dinámica). No es necesario preocuparse por liberar la memoria, el reciclador se encarga de ello y como es de baja prioridad, cuando entra en acción, permite liberar bloques de memoria muy grandes, lo que limita la fragmentación de la memoria. Java reduce en un 50% los errores más comunes de programación con los lenguajes C y C++ al eliminar muchas de las características de éstos entre las que destacan:

- Aritmética de apuntadores.
- Registros (*struct*).
- Definición de tipos (*typedef*).
- Macros (*#define*).
- Necesidad de liberar memoria (*free*).
- Java no ofrece soporte a herencia múltiple.
- Java tiene una clase `String` como parte del paquete java.lang. Esto difiere de los arreglos de caracteres de terminación nula, como se utiliza en C y en C++.

[4] Definición tomada de "*The Java Language Environment*"; Gosling, James y McGilton Henry; Mayo 1996.

[5] Para una descripción más detallada del recolector de basura, revise el capitulo 3 Máquina Virtual Java

Se puede aprender a programar en Java rápidamente una vez que se comprendan los conceptos básicos de la programación orientada a objetos. Java realiza un esfuerzo para no tener características sorprendentes. Hay muchos sistemas de programación que se enorgullecen de su versatilidad a la hora de proporcionar docenas de maneras diferentes de realizar lo mismo.

Si un lenguaje muestra lo suficiente las interioridades de la máquina, será libre de hacer casi cualquier cosa, de la manera que desee. Aunque es cierto que esto ofrece un rendimiento impresionante para el programador muy cuidadoso y experto, la libertad tiene un precio en cuanto a complejidad y compresión. En Java hay un número reducido de maneras de realizar una tarea dada.

2.2.2 Orientado a objetos.

Java implementa la tecnología básica de C++ con algunas mejoras y elimina algunas cosas para mantener el objetivo de la simplicidad del lenguaje. Java trabaja con sus datos como objetos y con interfaces a esos objetos. Soporta las tres característica propias del paradigma de la orientación a objetos:

- **Encapsulamiento** Implementa información oculta.
- **Polimorfismo** El mismo mensaje se envía a diferentes objetos, resultando en comportamientos que dependen de la naturaleza del objeto que recibió el mensaje.
- **Herencia** Pueden definirse nuevas clases y comportamientos (métodos) basados en clases existentes para obtener código reutilizable y organizado.

Las plantillas de objetos son llamadas como en C++, clases y sus copias, instancias. Estas instancias como en C++ necesitan ser construidas y destruidas en espacios de memoria.

2.2.3 Distribuido.

Java se ha construido con extensas capacidades de interconexión TCP/IP. Existen librerías de rutinas para acceder e interactuar con protocolos como *http* y *ftp*. Esto permite a los programadores acceder a la información a través de la red con tanta facilidad como a los archivos locales.

Java en sí no es distribuido, nos proporciona las librerías y herramientas para que los programas puedan ser distribuidos, es decir, que se ejecutan en varias máquinas, interactuando entre si.

2.2.4 Interpretado.

El intérprete Java (sistema *runtime*) puede ejecutar directamente el código objeto. Enlazar un programa normalmente consume menos recursos que compilarlo, por lo que los desarrolladores de Java pasan más tiempo desarrollando y menos compilando. No obstante, el compilador actual del JDK es bastante lento. Java es más lento que otros lenguajes de programación, como C++, ya que debe ser interpretado y no ejecutado como sucede en cualquier programa tradicional. Aunque este panorama está cambiando y Java sigue siendo un lenguaje interpretado, la situación se acerca mucho a la de programas compilados, sobre todo en lo que a la rapidez en la ejecución de código se refiere.

La Máquina Virtual Java es una definición estricta de una máquina virtual, por lo que un intérprete distinto debe estar disponible para cada arquitectura de hardware y sistema operativo en el cual se desea ejecutar la aplicación Java. Una vez que el intérprete Java y el sistema *runtime* estén disponibles en una plataforma dada, se pueden ejecutar aplicaciones Java sin importar donde han sido escritas, asumiendo que la aplicación fue desarrollada de manera portable.

2.2.5 Robusto.

Java realiza verificaciones en busca de problemas, tanto en tiempo de compilación como de ejecución. La comprobación de tipos en Java ayuda a detectar errores lo antes posible; en el ciclo de desarrollo, Java obliga a la declaración explicita de métodos, reduciendo así las posibilidades de error. Maneja la memoria para eliminar las preocupaciones por parte del programador sobre la liberación o corrupción de memoria, además se asegura del funcionamiento de la aplicación, realizando una verificación del *bytecode* que es el resultado de la compilación de un programa Java.

2.2.6 Seguro.

La seguridad en Java tiene dos facetas. En el lenguaje, características como los apuntadores o *casting* implícito que hace el compilador C y C++ se eliminan para prevenir el acceso ilegal a la memoria. Cuando se usa Java para crear un programa, se combinan las características del lenguaje con protecciones de sentido común aplicadas al propio programa.

El código de Java pasa muchas comprobaciones antes de ser ejecutado en una máquina. El código se pasa a través de un verificador de *bytecode* que comprueba el formato de los fragmentos de código y aplica un probador de teoremas para detectar fragmentos de código ilegal –código que falsea apuntadores, viola derechos de acceso sobre los objetos o intenta cambiar el tipo de clases de un objeto-, vea la figura 2-3 para entender la funcionalidad del cargador de clases[6] (*ClassLoader*).

[6] Aunque aquí se describe el papel que tiene el cargador de clases, en el capítulo siguiente se hará una descripción más detallada de sus funciones dentro de la Maquina Virtual Java.

El cargador de clases también ayuda a mantener la seguridad, separando el espacio de nombres del sistema de archivos locales del de los recursos procedentes de la red. Esto limita cualquier aplicación del tipo Caballo de Troya[7], ya que las clases se buscan primero entre las locales y luego entre las procedentes del exterior.

Las clases importadas de la red se almacenan en un espacio de nombres privado, asociado con el origen. Cuando una clase del espacio de nombres privados accede a otra clase, primero se busca en la clases predefinidas (del sistema local) y luego en el espacio de nombres de la clases que hace la referencia. Esto imposibilita que una clases suplan a una predefinida.

Dada la concepción del lenguaje y si todos los elementos se mantienen dentro del estándar marcado por **Sun Microsystems**, no hay peligro. Java imposibilita, también, abrir archivos de la maquina local (siempre que se realicen operaciones con archivos, éstas trabajan sobre el disco duro de la máquina donde parte el *applet*), no permite ejecutar ninguna aplicación nativa de una plataforma e impide que se utilicen otras computadoras como puentes, es decir, nadie puede utilizar una máquina para hacer peticiones o realizar operaciones con otras. Además, los intérpretes que incorporan los navegadores Web son más restrictivos.

Bajo esta condiciones y dentro de la filosofía que la única computadora segura es la que esta apagada se puede considerar que Java es un lenguaje seguro.

[7] Los virus tipo Caballo de Troya deben su nombre al libro de Homero “La Iliada” ya que este tipo de virus se disfrazan de aplicaciones inofensivas para poder atacar.

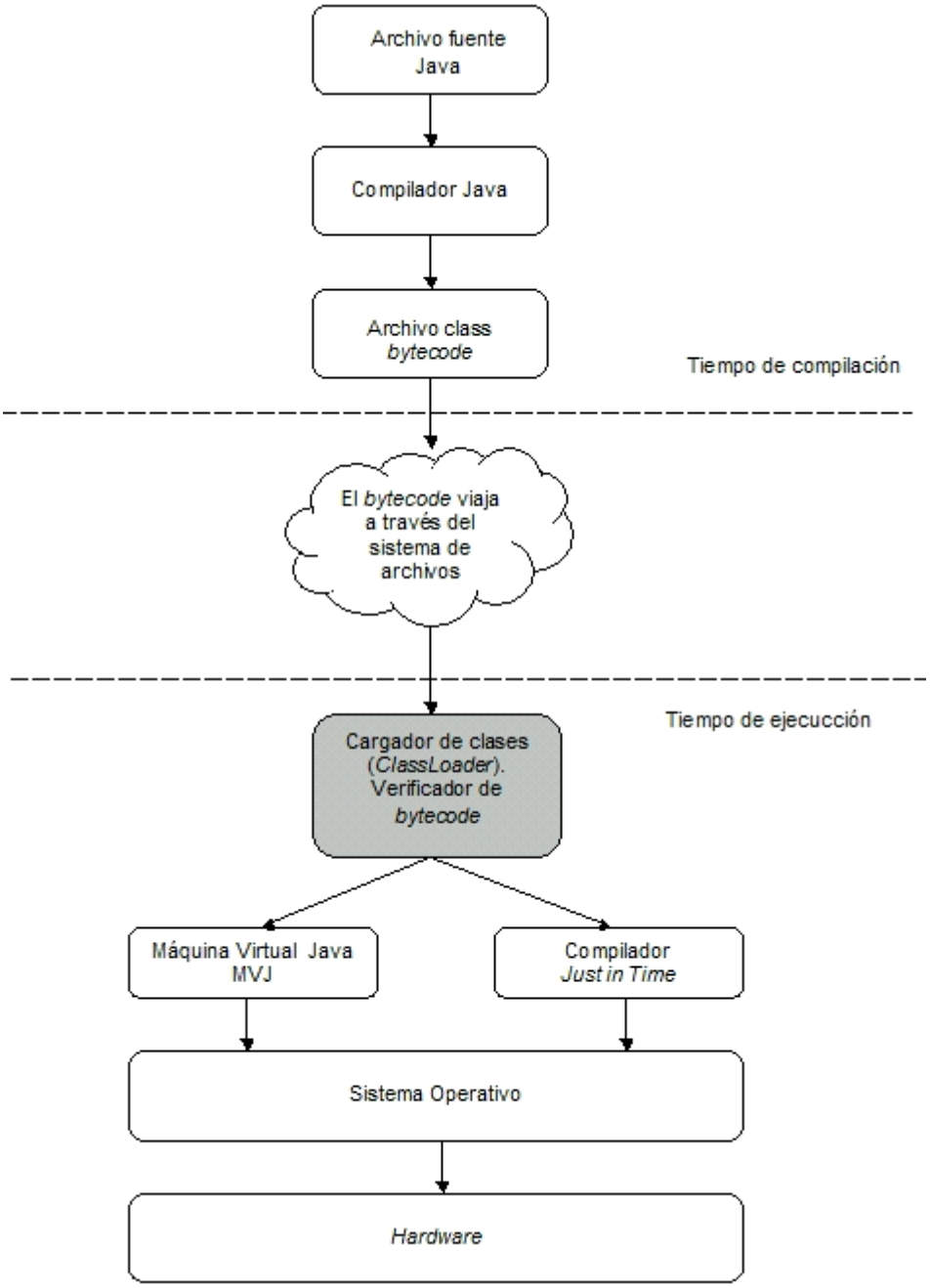

Figura 2-3 El cargador de clases verifica el archivo *.class.*

2.2.7 De arquitectura neutral.

Para establecer Java como parte integral de la red, el compilador Java compila su código a un archivo objeto de formato independiente de la arquitectura de la máquina en que se ejecutará. Cualquier máquina que tenga el sistema de ejecución (*runtime*) puede ejecutar ese código objeto.

La solución que el sistema Java adopta para resolver el problema de distribución binaria es un "formato de código binario" que es independiente de la arquitectura del hardware, interfaz del sistema operativo y sistema de ventanas.

El formato de este código binario independiente es una *arquitectura neutral*. Si la plataforma de ejecución de Java está disponible para un hardware y ambiente de software dado, una aplicación escrita en Java puede ser ejecutada en ese ambiente, sin necesidad de la plataforma donde fue desarrollada la aplicación.

2.2.8 Portable.

Más allá de la portabilidad básica para ser una arquitectura independiente, Java implementa otros estándares de portabilidad para facilitar el desarrollo. Un tipo entero en Java, `int`, siempre es un entero de 32 bits en complemento a dos. Un número real, `float`, siempre es un número de punto flotante de 32 bits, definido por el estándar IEEE 754. Además Java construye sus interfaces de usuario a través de un sistema abstracto de ventanas de forma que éstas puedan ser implantadas en un entorno Unix, Windows o MacOS.

2.2.9 De alto rendimiento.

Como los códigos de byte (*bytecodes*) son interpretados, el proceso de programación a veces no es tan rápido como la compilación y la ejecución directas en una plataforma de hardware en particular.

La compilación de Java incluye una opción para traducir los códigos de bytes (*bytecodes*) a código de máquina para la plataforma específica de hardware. Esto puede ofrecer la misma eficiencia que un proceso de compilación y cargado tradicional. De acuerdo con las pruebas de **Sun Microsystems**, el desempeño de esta traducción de código de byte a código de máquina es "casi indistinguible" de la compilación directa de programas en C o C++.

Sun Microsystems declara que un objeto se crea en aproximadamente 8.4 µs, para crear una nueva clase que contenga varios métodos se consume aproximadamente 11 µs, y para invocar un método de un objeto se requiere 1.7 µs.[8]

[8] Estos tiempos fueron tomados por Sun Microsystems y son resultado del promedio de muchas aplicaciones que se ejecutan en servidores de alto desempeño; tomado de "*The Java Language Environment*"; Gosling, James y McGilton Henry; Mayo 1996.

2.2.10 Multihilos.

Al ser multihilo o multitarea, Java permite realizar muchas actividades simultáneas en un programa. El término *multithreaded* es de difícil traducción aunque la forma más aceptada es la palabra **multitarea**. Las tareas a veces llamadas procesos ligeros o hilos de ejecución. son básicamente pequeños procesos o piezas independientes de un gran proceso. Al estar estas tareas construidas en el mismo lenguaje, son más fáciles de usar y más robustas que sus homólogas a C y C++.

El beneficio de ser multitarea consiste en un mejor rendimiento interactivo y mejor comportamiento en tiempo real. Aunque el comportamiento en tiempo real está limitado a las capacidades del sistema operativo, aun supera a los entornos de flujo único de programa (*single thread*) tanto en facilidad de desarrollo como en rendimiento.

2.2.11 Dinámico.

Java se beneficia todo lo posible de la tecnología orientada a objetos y no intenta conectar todos los módulos que comprende una aplicación hasta el mismo tiempo de ejecución, las librerías nuevas o actualizadas no paralizan la ejecución de las aplicaciones actuales siempre que mantengan la API anterior.

2.3 Compilación y ejecución de programas Java.

En la mayoría de los lenguajes de programación, se debe compilar o interpretar un programa que se ejecuta en una computadora. El lenguaje de programación Java pasa por dos fases: primero se compila y después se interpreta. Con el compilador, se traduce un programa (archivo *.java*) a un lenguaje intermedio llamado *bytecode* (archivo *.class*) –éste código es el que otorga la independencia de la plataforma-. El intérprete traduce y ejecuta cada instrucción de *bytecode* en la computadora.

El proceso de compilación ocurre una sola vez mientras que la interpretación ocurre cada ocasión que el programa es ejecutado. En la figura 2-4 se muestran las fases por las que pasa una aplicación antes de ser ejecutada.

Los código de bytes de Java (*bytecode*) son un conjunto de instrucciones, correspondientes a un lenguaje de máquina que no es específico de ningún procesador, sino de la Máquina Virtual Java.

Para las aplicaciones de internet (*applets*), la máquina virtual está incluida en el navegador y para las aplicaciones Java convencionales, puede venir con el sistema operativo, con el paquete Java o se puede obtener a través de internet.

Con este esquema se logra la portabilidad del lenguaje Java, pero conlleva un problema muy grande, la pérdida de velocidad en la ejecución del programa. Por esta razón, la solución fue diseñar un compilador que produjera un lenguaje que es interpretado a velocidades, si no iguales, sí

cercanas a la de los programas nativos (programas en código máquina propio de cada plataforma), logro conseguido mediante la Máquina Virtual Java. Con todo, las aplicaciones todavía adolecen de una falta de rendimiento apreciable.

Afortunadamente, el rendimiento con respecto a aplicaciones equivalentes escritas en código nativo ha ido disminuyendo debido a la utilización de **compiladores JIT** (*just in time*) o compiladores al instante. Un compilador JIT interacciona con la máquina virtual para convertir el código de bytes (*bytecode*) en código nativo. Como consecuencia, se mejora la velocidad durante la ejecución.

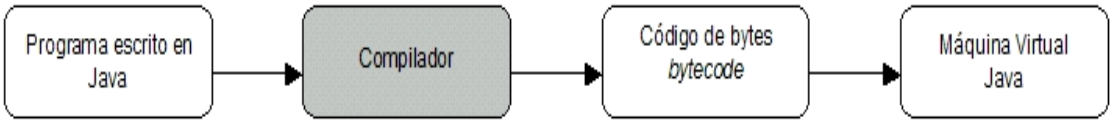

Figura 2-4 Fases de compilación y ejecución de una aplicación Java.

En la figura 2-5 se muestra un ejemplo de las fases de una aplicación Java, para fines demostrativos se trabajó con un programa llamado `EjemploCap.java`, el cual sólo muestra un mensaje en la pantalla. Al compilar el código fuente se genera el archivo `EjemploCap.class` que es un código de bytes o *bytecode,* finalmente la Máquina Virtual Java toma el archivo *.class,* lo interpreta y ejecuta las instrucciones.

```
public class EjemploCap
{
        public static void main(String[] args)
        {
                System.out.println("Ejemplo Capitulo 2");
        }
}
```

EjemploCap.java

```
ca fe ba be 00 03 00 2d 00 1d 0a 00 06 00 0f 09
00 10 00 11 08 00 12 0a 00 13 00 14 07 00 15 07
00 16 01 00 06 3c 69 6e 69 74 3e 01 00 03 28 29
56 01 00 04 43 6f 64 65 01 00 0f 4c 69 6e 65 4e
75 6d 62 65 72 54 61 62 6c 65 01 00 04 6d 61 69
6e 01 00 16 28 5b 4c 6a 61 76 61 2f 6c 61 6e 67
2f 53 74 72 69 6e 67 3b 29 56 01 00 0a 53 6f 75
72 63 65 46 69 6c 65 01 00 0f 45 6a 65 6d 70 6c
6f 43 61 70 2e 6a 61 76 61 0c 00 07 00 08 07 00
17 0c 00 18 00 19 01 00 12 45 6a 65 6d 70 6c 6f
20 43 61 70 69 74 75 6c 6f 20 32 07 00 1a 0c 00
1b 00 1c 01 00 0a 45 6a 65 6d 70 6c 6f 43 61 70
01 00 10 6a 61 76 61 2f 6c 61 6e 67 2f 4f 62 6a
65 63 74 01 00 10 6a 61 76 61 2f 6c 61 6e 67 2f
53 79 73 74 65 6d 01 00 03 6f 75 74 01 00 15 4c
6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72
65 61 6d 3b 01 00 13 6a 61 76 61 2f 69 6f 2f 50
72 69 6e 74 53 74 72 65 61 6d 01 00 07 70 72 69
6e 74 6c 6e 01 00 15 28 4c 6a 61 76 61 2f 6c 61
6e 67 2f 53 74 72 69 6e 67 3b 29 56 00 21 00 05
00 06 00 00 00 00 00 02 00 01 00 07 00 08 00 01
00 09 00 00 00 1d 00 01 00 01 00 00 00 05 2a b7
00 01 b1 00 00 00 01 00 0a 00 00 00 06 00 01 00
00 00 01 00 09 00 0b 00 0c 00 01 00 09 00 00 00
25 00 02 00 01 00 00 00 09 b2 00 02 12 03 b6 00
04 b1 00 00 00 01 00 0a 00 00 00 0a 00 02 00 00
00 05 00 08 00 06 00 01 00 0d 00 00 00 02 00 0e
```

EjemploCap.class

```
C:\Tesis\Marago>java EjemploCap
Ejemplo Capitulo 2
```

Figura 2-5 Fases por las que pasa la aplicación EjemploCap.java

2.4 La plataforma Java

En párrafos anteriores mencioné que Java no sólo es un lenguaje de programación con ciertas características; además, debido a que presenta un conjunto de herramientas de software que permiten crear e implantar contenido ejecutable utilizando el lenguaje Java se puede afirmar que Java también es una plataforma.

Una plataforma es el hardware o ambiente de software en donde se ejecuta un programa. Algunas plataformas conocidas son Windows 2000, Linux, Solaris y MacOS; la mayoría de éstas plataformas se describen como una combinación de sistema operativo y hardware. La plataforma Java difiere de éstos ya que es una plataforma basada en software y se ejecuta sobre plataformas basadas en hardware. La plataforma Java tiene dos componentes:

- La Máquina Virtual Java (*Java Virtual Machine*, JVM).
- La interfaz de programación de aplicaciones Java (*Application Program Interface*, API).

En capítulos posteriores se hablará ampliamente de la Máquina Virtual Java, por ahora sólo mencionaré que es una estructura de datos que funciona como una pila. La API de Java es una colección de componentes de software que proveen utilerías para desarrollar aplicaciones y está agrupada en librerías de clases que están relacionadas junto con sus interfaces; estas librerías se conocen como paquetes.

La figura 2-6 describe un programa que se ejecuta sobre la plataforma Java; la API de Java y la Máquina Virtual Java aíslan el programa del Hardware.

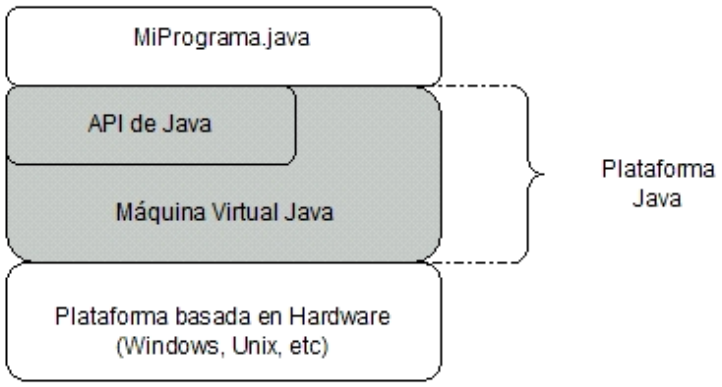

Figura 2-6 Esquema de la plataforma Java.

La mayoría de los programas escritos en Java son *applets* y aplicaciones; si navegamos a través de la red probablemente encontraremos muchos *applets*.

Un *applet* es un programa que se adhiere a ciertas convenciones que permiten que se ejecute dentro de un navegador, pero el lenguaje de programación Java no sólo sirve para crear programas que hagan más llamativo el tiempo que pasamos en la red, el propósito de Java es también crear poderosas herramientas y aplicaciones utilizando la API de Java.

Una aplicación, es un programa que se ejecuta directamente en la plataforma Java y para que se pueda soportar todo este tipo de programas, la API de Java lo hace a través de componentes que vienen en forma de paquetes y proveen un amplio rango de funcionalidad.

Cada implementación de Java viene con una Máquina Virtual y las API´s que están integradas en el paquete de desarrollo Java (*Java Development Kit*, JDK). La API de Java se divide en grupos para ayudar a desarrollar aplicaciones poderosas entre ellas están:

- **Esenciales**: Objetos, cadenas, hilos, números, datos de entrada y salida, propiedades del sistema, fechas y horas.
- **Applets**: El conjunto de convenciones usadas por los applets.
- **Redes**: URL´s, TCP Protocolo de control de transmisión (*Transmisión Control Protocol*), IP Protocolo de Internet (*Internet Protocol*) y *sockets*.
- **Internacionalización**: Ayuda para escribir programas que pueden ser localizados por usuarios de todo el mundo, los programas se pueden adaptar automáticamente a especificaciones locales y desplegarlas en el idioma apropiado.
- **Seguridad**: De alto y bajo nivel, incluye firmas digitales, manejo de llaves públicas y privadas y certificados.
- **Componentes de Software:** Conocidos como *JavaBeans*.
- **Serialización de Objetos:** Permite la persistencia de datos y comunicación vía RMI Invocación Remota de Métodos.
- **Java Database Conectivity (JDBC):** permite el acceso a bases de datos relaciones a través de enunciados SQL.

La siguiente figura describe el paquete de desarrollo Java[9].

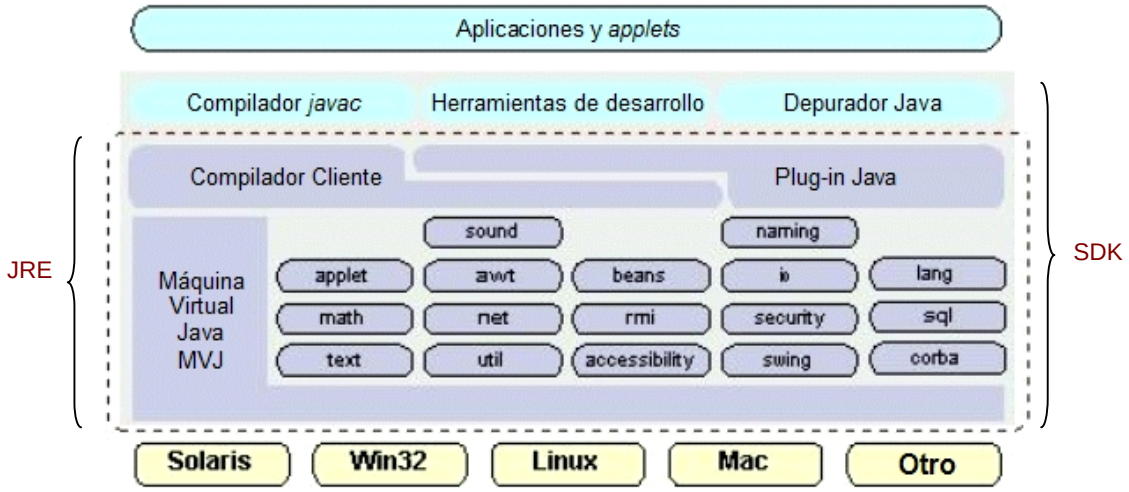

Figura 2-7 Paquete de desarrollo Java JDK.

Como se observa en la figura 2-7 se puede hablar de tres niveles.

- Las aplicaciones de usuario, que pueden ser applets, aplicaciones, etc.
- El paquete de desarrollo Java, que consta de un compilador, un depurador, la librería de clases y la máquina virtual Java.
- La plataforma, que es transparente tanto para el programador como para el usuario final.

[9] Java 2 *Runtime Environment* (JRE) consiste en la máquina virtual, clases núcleo de la plataforma Java y archivos de soporte. Java SDK incluye JRE y herramientas de desarrollo como el compilador y depuradores.

CAPÍTULO

3

Capítulo 3 La Máquina Virtual Java.

Para llevar a cabo la regeneración de código fuente a partir del formato del archivo *.class*, es importante conocer la relación existente entre el lenguaje de programación Java y la Máquina Virtual Java. En el presente capítulo se explica el papel que tiene una máquina virtual, los tipos de datos que utiliza y su funcionamiento, en particular, se describen las partes que constituyen la implementación de la Máquina Virtual Java y la forma en que representa sus datos y sus objetos.

3.1 ¿Qué es una máquina virtual?.

Cuando un programador utiliza 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 en la cual fue desarrollada, esto se debe a que el compilador genera código específico para el procesador sobre el cual está trabajando.

Una máquina virtual interpreta el código que genera un compilador y lo convierte en código nativo para el procesador sobre el cual trabaja. La plataforma Java se encuentra por encima de otras plataformas, es decir; el código que generan sus compiladores no es específico de un procesador en particular, sino de una máquina virtual. Aún cuando existen múltiples implementaciones de la Máquina Virtual Java (*Java Virtual Machine*, JVM)[1], cada una es específica de la plataforma sobre la cual se ejecuta, existe una única especificación de la máquina virtual, que proporciona una vista independiente del hardware y del sistema operativo sobre el que se está trabajando. De esta manera un programador en Java hablando de su código lo "**escribe una vez, y lo ejecuta donde sea**"[2].

Precisamente la JVM es la clave de la independencia de los programas Java, sobre el sistema operativo y el hardware en que se ejecutan, ya que se encarga de proporcionar una visión desde 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

[1] En adelante se utilizarán las iniciales de la Máquina Virtual Java o *Java Virtual Machine, JVM.*
[2] SUN Microsystems, la compañía creadora del lenguaje Java, emplea la frase "*Write once, run anywhere*".

estricta de tipos de datos, múltiples hilos, con ligado dinámico y con recolección automática de basura[3].

La JVM es el núcleo del lenguaje de programación Java. De hecho, es imposible ejecutar un programa Java sin ejecutar alguna implementación de la JVM. En la JVM 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 ejecuta un programa Java, las instrucciones que lo componen no son ejecutadas directamente por el procesador del sistema que se está utilizando, 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 antes mencionado.

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 en 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 3-1 puede observarse la capa de software que implementa a la JVM[4]. Esta capa de software oculta los detalles inherentes a la plataforma y 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.

[3] El concepto de recolector de basura se refiere a liberar memoria que ya no se ocupa, labor realizada por el *garbage collector* (*gc*).
[4] Para una descripción más detallada de las capas que componen el *paquete de desarrollo Java* JDK, consulte el capítulo 2.

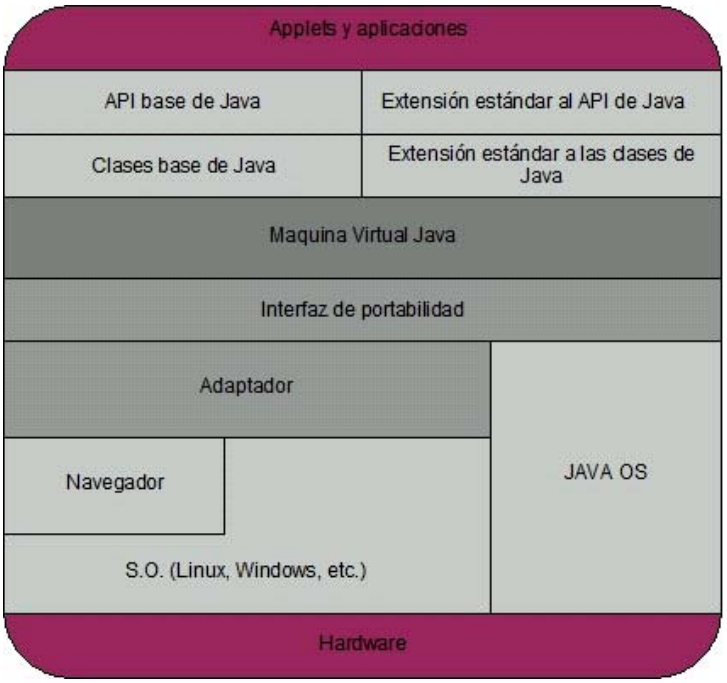

Figura 3-1 Capas de software que implementan a la Máquina Virtual Java.

3.2 Tipos de datos de la JVM.

Al igual que el lenguaje Java, la JVM opera con dos tipos de datos: *tipos primitivos* y *tipos por referencia*. Existen correspondientemente dos tipos de valores que pueden ser almacenados en variables, pasados como argumentos, regresados por los métodos y operados; los tipos primitivos almacenan *valores primitivos*, mientras que los *valores por referencia* se refieren a objetos, aunque no lo son. Los valores primitivos por el contrario no se refieren a algo, ellos mismos son los datos.

La JVM espera que todos los tipos de datos sean revisados en tiempo de compilación y no en tiempo de ejecución. En particular, los datos necesitan ser etiquetados o ser inspeccionados para determinar su tipo. El conjunto de instrucciones de la JVM distingue sus operandos utilizando instrucciones para operar sobre valores de tipos específicos; como ejemplo se tienen las instrucciones *iadd, ladd, fadd y dadd*[5] que suman dos valores numéricos, pero requieren operandos los cuales deben pertenecer a los tipos `int`, `long`, `float` y `double`, respectivamente.

La JVM contiene soporte explícito para los objetos. Un objeto es una instancia de una clase asignada dinámicamente o puede ser un arreglo. Una referencia a un objeto es considerada por la JVM como un tipo de dato por referencia; mientras que los valores de tipos de datos por referencia

[5] Para una descripción más detallada consulte el Apéndice A.

pueden ser pensados como apuntadores hacia los objetos, aunque las operaciones se ejecutan sobre los objetos y nunca sobre las direcciones de los objetos. Los objetos son siempre operados, pasados y probados como tipos de datos por referencia.

Los tipos de datos primitivos soportados por la JVM son de tipo numérico y *returnAddress*; los tipos numéricos a su vez están divididos en tipos numéricos enteros (`byte`, `short`, `int`, `long` y `char`) y en tipos numéricos flotantes (`float` y `double`). En el lenguaje Java, los tipos `byte`, `short` e `int` tienen el mismo *status*, pero en la JVM los tipos primitivos `byte`, `char` y `short` se conocen como tipos de almacenamiento; la JVM no posee soporte para manejarlos, así que son promovidos a valores de tipo `int`. Los tipos primitivos de punto flotante utilizan la representación numérica IEEE 754[6].

En el lenguaje Java existe el tipo de dato `boolean`, el cual sólo puede tener dos valores `true` o `false`. A nivel de la JVM no existe el tipo `boolean`, de hecho los tipos de datos booleanos son representados como enteros: 0 si es `false` o 1 si es `true`[7].

La JVM trabaja con otro tipo de dato primitivo que no está disponible para los programadores Java, el tipo *returnAddress* que es usado para implementar la cláusula `finally`[8] en los programas Java; este tipo de dato se utiliza en las instrucciones *jsr*, *ret* y *jsr_w* las cuales generan un valor que es guardado en una variable local utilizando *astore* para después usarlo con la instrucción *ret*[9].

Existen tres clases de tipos referenciados o de referencia: *clase*, *interfaz* y *arreglos*, las cuales son referenciados dinámicamente.

Una referencia puede tener valor nulo, es decir, una referencia sin objeto; lo cual se denota por la palabra reservada `null`. En la figura 3-2 se puede apreciar de forma gráfica la descripción de la familia de tipos de datos para la JVM.

[6] La norma IEEE 754 es un estándar para aritmética de punto flotante, ANSI/IEEE Std 754–1985 (IEEE, Nueva York)
[7] En algunas implementaciones de la máquina virtual, el valor de *true* no necesariamente es uno, basta con que sea diferente de cero.
[8] Uso de la cláusula *finally*, The Java Programming Language Third Edition; Gosling, James and Arnold, Ken.
[9] La instrucción *ret*, se utiliza para regresar de una subrutina.

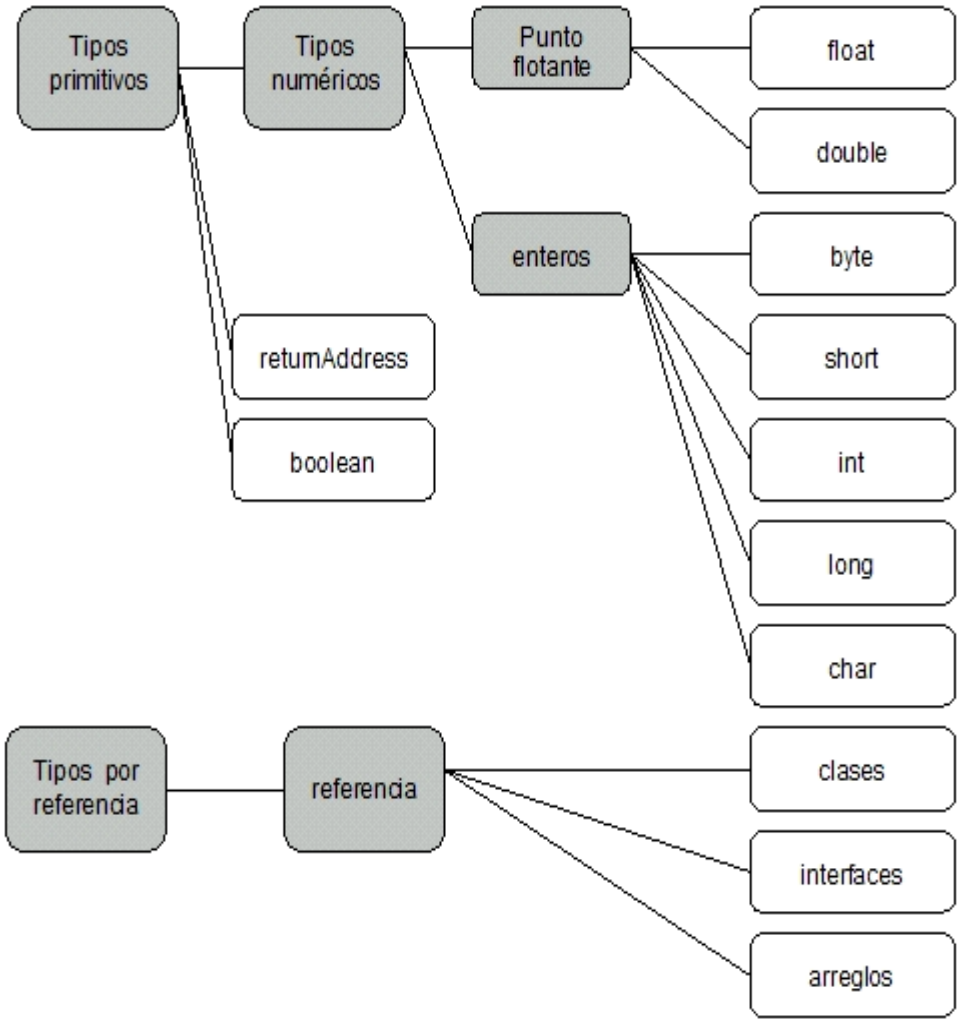

Figura 3-2 Tipos de datos para la JVM.

3.3 Palabras (words).

La unidad básica de tamaño para los datos que utiliza la JVM es una palabra (*word*). Ésta debe ser lo suficientemente grande para contener los valores de tipo `byte, char, short, int, float`, *por referencia* o *returnAddress;* dos palabras deben ser suficientes para contener los valores de tipo `long` o `double`.

Una palabra es generalmente del mismo tamaño que la utilizada por el sistema operativo en el que reside, es decir; en una plataforma de 32 bits, la palabra será de 32 bits; la selección del tamaño de la palabra dependerá de la plataforma específica, no es parte del diseño de la maquina virtual. En la tabla 3-1 se muestran los diferentes tipos de datos y su tamaño.

Tipo	Contiene	Valor por omisión	Tamaño [bit]	Valor Max | Valor Min
boolean	True o false	false	1	N.D[10]. | N.D.
char	Carácter Unicode	\u0000	16	\u0000 | \uFFFF
byte	entero con signo	0	8	(2e+8)/2 | -((2E+8)/2-1)
short	entero con signo	0	16	(2E+16)/2 | -((2E+16)/2-1)
Int	entero con signo	0	32	(2E+32)/2 | -((2E+32)/2-1)
long	entero con signo	0	64	(2E+64)/2 | -((2E+65)/2-1)
float	IEEE754 / punto flotante	0.0	32	(2E+32)/2 | -((2E+32)/2-1)
double	IEEE754 / punto flotante	0.0	64	(2E+64)/2 | -((2E+64)/2-1)

Tabla 3-1 Rango de valores para los tipos de datos de la JVM.

[10] No disponible.

3.4 Estructura de la JVM.

En la especificación de la máquina virtual, su comportamiento se describe en términos de subsistemas, áreas de memoria, tipos de datos e instrucciones. Estos componentes describen una estructura interna abstracta para la JVM[11]. El propósito de los componentes no es dictar una arquitectura interna para su implementación, sino proveer una forma que defina el comportamiento externo de la implementación. La especificación de cualquier máquina virtual requiere que su comportamiento este definido por sus componentes abstractos y sus interacciones.

En la figura 3-3 se muestra un diagrama de bloques que incluye un subsistema y áreas de memoria; cada JVM tiene un cargador de clases (*class loader*), mecanismo que carga las clases e interfaces a memoria; también posee un motor de ejecución responsable de ejecutar las instrucciones contenidas en los métodos de las clases.

Cuando la JVM ejecuta un programa, requiere almacenar varios datos, incluyendo *bytecode* y otro tipo de información que extrae del archivo *.class* como: instancias de objetos, parámetros de los métodos, valores de retorno, variables locales y resultados intermedios de cálculos. La JVM organiza la memoria que necesita y ejecuta un programa en varias áreas de datos en tiempo de ejecución.

Diferentes tipos de implementaciones de maquina virtual pueden tener diferentes configuraciones de memorias, algunas implementaciones poseen mucha memoria para trabajar y otros pueden tener muy poca memoria; la naturaleza abstracta de la especificación de las áreas de datos en tiempo de ejecución facilitan la implementación de la máquina virtual en una gran variedad de computadoras y dispositivos.

Algunas áreas de datos en tiempo de ejecución son compartidas por todos los hilos de la aplicación y otras son únicas para cada hilo. Cada instancia de la JVM posee un **área de métodos** *(method area)* y un **montículo** *(heap)*. Estas áreas son compartidas por todos los hilos que están siendo ejecutados dentro de la maquina virtual. Vea la figura 3-3 para una descripción gráfica de las áreas de memoria.

Cada aplicación o *applet* posee su propio **montículo** y **área de métodos** y cada hilo posee su propio registro o **contador de programa** *(program counter)* y **pila de programa** *(Java stack)* .

Cada **pila de programa** está a su vez dividida en **marcos de programa** *(frames)*, donde cada método posee su propio **marco de pila** (*stack frame).*

[11] En esencia la JVM se comporta como una pila.

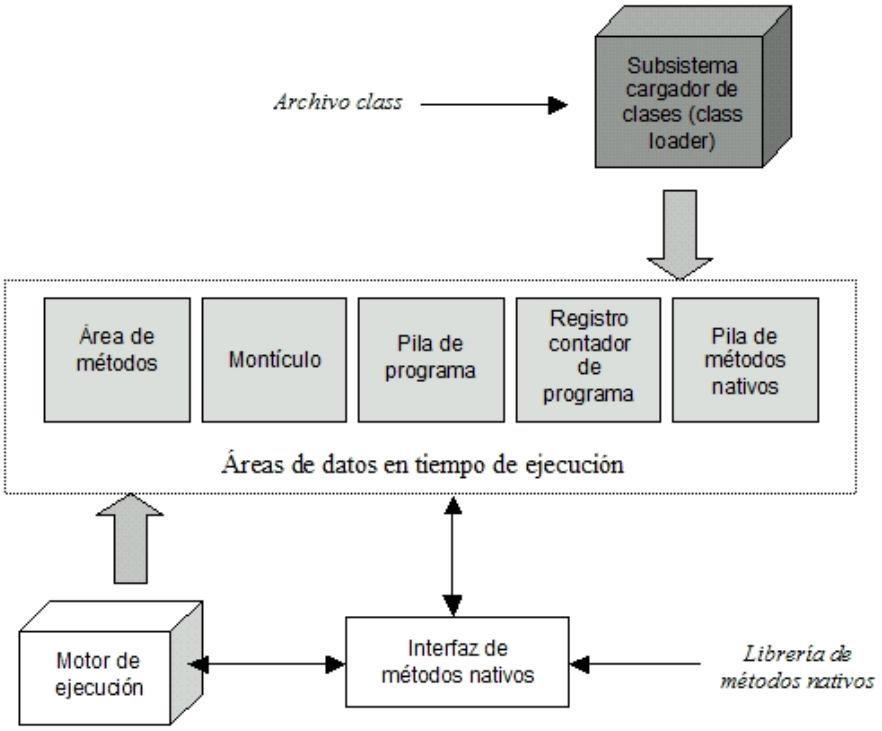

Figura 3-3 Estructura de la Máquina Virtual de Java.

3.4.1 Subsistema cargador de clases (Class Loader).

La parte de la implementación de la JVM que se encarga de encontrar y cargar las clases, es el subsistema cargador de clases (*class loader subsystem*). **La JVM contiene dos tipos de cargadores de clases**: un cargador de clases llamado *bootstrap* y otro definido por el usuario.

El cargador de clases *bootstrap* es parte de la implementación de la JVM y el cargador de clases definido por el usuario es parte de una aplicación java ejecutándose.

El subsistema cargador de clases (*class loader subsystem*) envuelve algunas otras partes de la maquina virtual y varias clases de la librería `java.lang`. El cargador de clases es responsable no sólo de localizar e importar los archivos binarios de las clases, también debe verificar que hayan sido correctamente importados de las clases, asignar e inicializar la memoria para las variables de las clases y asistir en la resolución de las referencias simbólicas. Estas actividades son ejecutadas en un estricto orden:

- **Carga**: encuentra e importa los datos binarios (archivos *.class*).
- **Enlace**: ejecuta verificación, preparación y resolución.
- **Verificación**: se asegura que las clases hayan sido correctamente importadas.
- **Preparación**: asigna memoria para las variables locales e inicializa los valores por omisión de las variables.
- **Resolución**: transforma las referencias simbólicas de las clases dentro de las referencias directas.
- **Inicialización**: invoca código java que inicialice las variables de las clases con sus valores apropiados.

La implementación de la JVM debe reconocer y cargar clases e interfaces almacenadas en archivos binarios que cumplan con el formato del archivo *.class*. Una implementación es libre de reconocer además otros tipos de archivos *.class* que no cumplan con la especificación.

Cada vez que se ejecuta una aplicación Java o un *applet;* se ejecuta también el cargador de clases (*class loader*), que se encarga de cargar las clases necesarias incluyendo las clases de la API de Java.

Dado un nombre de clase, el cargador de clases *bootstrap* debe producir los datos que se definieron dentro de la clase; para la versión de la JVM de **Sun Microsystems** 1.1, la implementación buscaba un directorio definido por el usuario almacenado en una variable de ambiente llamada **CLASSPATH**. El cargador busca dentro de cada directorio en el orden en que aparecen en el **CLASSPATH**, el nombre del subdirectorio es construido con base en el paquete de la clase. Por ejemplo si el cargador de clases *bootstrap* esta buscando la clase `java.lang.Object,` éste buscará el archivo `Object.class` en el subdirectorio java\lang del directorio definido en el **CLASSAPATH**.

En la versión de la JVM 1.2 (**Java 2**) el cargador de clases sólo busca en el directorio donde el API de Java fue instalado, para esta versión el cargador de clases no revisa el **CLASSPATH**.

En la figura 3-4 se muestra un diagrama de la JVM, que contiene un cargador de clases que carga los archivos *.class* tanto del programa como de la API de Java que se necesitan para que el *bytecode* sea ejecutado por el motor de ejecución.

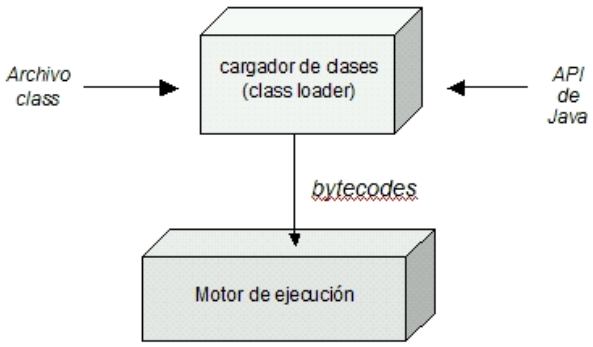

Figura 3-4 Esquema de la JVM interactuando con el cargador de clases (*class loader*).

3.4.2 El área de métodos (method area).

Dentro de la JVM, la información acerca de los tipos de datos cargados se almacena en un área lógica de memoria llamada área de métodos (*method area*). Cuando la JVM ejecuta una aplicación o *applet*, utiliza un cargador de clases para localizar el archivo *.class* apropiado. El cargador de clases lee el archivo *.class* –siempre de manera secuencial- y lo pasa a la JVM. Ésta, extrae información acerca de los tipos de datos que se declararon y almacena la información en el área de métodos (*method area*).

La memoria para variables estáticas declaradas en la clase también se reserva dentro del área de métodos. Para cada tipo de dato que se carga en el área de métodos, la JVM debe almacenar la siguiente información:

- Nombre calificado[12] de las variables declaradas.
- El nombre calificado de la clase padre de las variables (a menos que las variables pertenezcan a la clase `java.lang.Object`; ninguna variable perteneciente a esta clase posee una clase padre).
- Reconocer si el archivo *.class* es una clase o interfaz.
- Los modificadores de las variables(`public, abstract, final`, etc).
- Una lista ordenada de los nombres completamente calificados de cualquier interfaz padre directa.

[12] El término en inglés es *fully qualified name*, en español es más conocido como nombre calificado. Para una descripción detallada de lo nombres calificados vea el capítulo 4.

Dentro del archivo *.class* y de la JVM, los nombres de los tipos de datos son siempre almacenados como nombres calificados. En el código fuente Java, un nombre completamente calificado es el nombre del paquete del tipo de dato más un punto y el nombre del tipo de dato. Por ejemplo, el nombre calificado de la clase `Object` en el paquete `java.lang` es `java.lang.Object`. En el archivo *.class* los puntos son remplazados por diagonales de la forma **java/lang/Object**. Estos nombres son almacenados en el área de métodos.

Además de la información básica que para cada clase se carga en memoria, la JVM debe almacenar también:

- El ***constant_pool*** para la clase.
- Información de las variables de instancia.
- Información de los métodos.
- Todas las variables estáticas declaradas, excepto variables constantes.
- Una referencia a la clase `ClassLoader.`
- Una referencia a la clase `Class`.

3.4.2.1 Constant_pool.

Cada clase de Java o interfaz tiene asociado un *constant_pool* que es obtenido por la JVM para cada clase del archivo *.class*. El *constant_pool* es creado cuando una clase o interfaz es exitosamente cargada por la JVM -mediante el *class loader*- y se aloja en el área de métodos

El *constant_pool* tiene un papel similar a una tabla de símbolos encontrada en librerías y archivos ejecutables compartidos; representa una colección de todos los datos simbólicos descritos por la clase –esto incluye referencias simbólicas a campos, clases e interfaz y métodos usados internamente por la clase.

Las entradas al *constant_pool* están ordenadas; la primera constante en el archivo *.class* es el número uno, el segundo el dos y así sucesivamente. Estos números son significativos debido a que se hace referencia a la constante mediante su índice.

Las referencias a las constantes almacenadas utilizan 8 o 16 bits sin signo; así que el número máximo de entradas al *constant_pool* es de 65535. Cabe recalcar que no hay forma de que un método acceda a un *constant_pool* que pertenezca a otra clase; cada *constant_pool* es privado a su clase.

El *constant_pool* es usado para almacenar todas las constantes, (como ejemplo: números y cadenas `string`) que aparecen en el texto del método. Éstos valores constantes están disponibles dentro del método a través de instrucciones como: *ldc, ldc_w, ldc2_w*. En la figura 3-5 se muestra la apariencia del *constant_pool*, que se obtuvo con el desensamblador **Marago,** para el ejemplo del clásico programa “Hola Mundo”.

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

**********************************
Constant_pool
**********************************
1 CONSTANT_Methodref: 6,15
2 CONSTANT_Fieldref: 16,17
3 CONSTANT_String: 18
4 CONSTANT_Methodref: 19,20
5 CONSTANT_Class: 21
6 CONSTANT_Class: 22
7 CONSTANT_Utf8: <init>
8 CONSTANT_Utf8: ()V
9 CONSTANT_Utf8: Code
10 CONSTANT_Utf8: LineNumberTable
11 CONSTANT_Utf8: main
12 CONSTANT_Utf8: ([Ljava/lang/String;)V
13 CONSTANT_Utf8: SourceFile
14 CONSTANT_Utf8: HolaMundo.java
15 CONSTANT_NameAndType: 7,8
16 CONSTANT_Class: 23
17 CONSTANT_NameAndType: 24,25
18 CONSTANT_Utf8: Hola Mundo
19 CONSTANT_Class: 26
20 CONSTANT_NameAndType: 27,28
21 CONSTANT_Utf8: Paquete/HolaMundo
22 CONSTANT_Utf8: java/lang/Object
23 CONSTANT_Utf8: java/lang/System
24 CONSTANT_Utf8: out
25 CONSTANT_Utf8: Ljava/io/PrintStream;
26 CONSTANT_Utf8: java/io/PrintStream
27 CONSTANT_Utf8: println
28 CONSTANT_Utf8: (Ljava/lang/String;)V

**********************************
**********************************
```

Figura 3-5 Ejemplo del *constant_pool* para el programa Hola Mundo, obtenido por el desensamblador **Marago.**

3.4.2.2 Información de las variables de instancia.

La JVM debe almacenar información en el *área de métodos* para cada variable de instancia que es declarada, además el orden en que se declaran las variables en las clases o interfaces se mantiene en el área de métodos.

- Nombre de la variable de instancia.
- Tipo de dato de la variable de instancia.
- Modificador de la variable (`public, private, static`, etc).

3.4.2.3 Información de los métodos.

La JVM también debe almacenar información en el *área de métodos* para cada método que es declarado en una clase o interfaz, como ocurre con las variables de instancia, el orden en que son declarados también se conserva.

- El nombre del método.
- El tipo de regreso del método o `void.`
- El número y tipo de los parámetros del método.
- Los modificadores del método.

Además de la lista arriba mencionada, la siguiente información también es almacenada **para cada método que no es abstracto o nativo.**

- El *bytecode* del método.
- El tamaño de la pila de operandos y la sección de variables locales a los métodos.
- Una tabla de excepciones (*exception Table*).

3.4.2.4 Variables de clase.

Las variables estáticas (variables de clase) son compartidas por todas las instancias de una clase, pueden ser leídas o modificadas, incluso en ausencia de cualquier instancia, éstas variables están asociadas con la clase -no con las instancias de la clase- así que son parte lógica de los datos de la clase en el área de métodos (*method area*). Antes que la JVM pueda utilizar una clase, debe asignar memoria en el área de métodos para cada variable declarada estática (`static`) y que no sea `final`.
Las variables de clase declaradas como `final` (constantes) no son tratadas de la misma manera que las variables de clase que no son `final.` Cada clase que utilice una variable `final`, obtienen una copia de su valor constante en su propio *constant_pool*. Como parte del *constant_pool*, las variables `final` son almacenadas en el área de métodos, de la misma manera que las variables de clase que no lo son, pero considerando que las variables de clase no `final` son almacenadas como parte de los datos de la clase que las declara, en cambio las variables de clase declaradas `final` son almacenadas como parte de los datos de la clase que las usa.

3.4.2.5 Referencia a la clase `ClassLoader`.

Para cada clase que se carga, la JVM debe mantener la pista de cómo fue cargado, ya sea vía el cargador de clases *bootstrap* o por un cargador de clases definido por el usuario. Para las clases cargadas a través del cargador definido por el usuario, la JVM debe almacenar una referencia de la clase que cargó el cargador de clases definido por el usuario. Esta información es almacenada como parte de los datos de la clase en el área de métodos.

3.4.2.6 Referencia a la clase `Class`.

Una instancia de la clase `java.lang.Class` es creada por la JVM para cada clase que se carga. La JVM debe asociar la referencia a la instancia `Class` para los datos de la clase que fue cargada y que se encuentra en el área de métodos.

Dada una referencia al objeto `Class`, se puede obtener información acerca de los datos de la clase cargada, mediante los métodos declarados en la clase `Class`. Estos métodos dan acceso a la información almacenada en el área de métodos, a continuación se muestran algunos de los métodos declarados en la clase `Class`.

- `public String getName();`
- `public Class getSuperClass();`
- `public boolean isInterface();`
- `public Class[] getInterfaces();`
- `public ClassLoader getClassLoader();`

Estos métodos regresan información acerca de la clase cargada, `getName()` regresa el nombre completamente calificado de la clase, `getSuperClass()` regresa la instancia de la clase Padre[13], `isInterface()` determina si la clase especificada es una representación de una interfaz, `getInterfaces()` determina las interfaces implementadas por la clase o interfaz implementada por el objeto y `getClassLoader()` regresa el *class loader* de la clase, algunas implementaciones pueden usar `null` para representar el *class loader bootstrap*.

3.4.3 Pila de programa (Java stack).

Cada hilo en la JVM tiene una pila de programa (*Java Stack*) privado, creado al mismo tiempo que el hilo. La pila de programa esta a la vez dividida en cuadros (*frames*) y su función es equivalente a una pila convencional en el lenguaje C: mantener variables locales y resultados parciales y juega parte importante en la invocación del método y su retorno y sólo se puede operar con dos instrucciones *push* y *pop*.

[13] La capacidad de conocer detalles de otras clases (incluidas las nuestras) a través de una clase habilitada por Java, se conoce como reflexión.

El método que esta siendo ejecutado por un hilo, es el hilo del método actual. El cuadro de pila (*stack frame*) para el método actual es el cuadro actual. La clase que define el método actual es llamada clase actual, y el *constant_pool* de la clase actual es el *constant_pool* actual.

Si se ejecuta un método, la JVM mantiene la pista de la clase actual y del *constant_pool* actual, cuando la maquina virtual encuentra operaciones que se ejecutan sobre los datos almacenados en el cuadro de pila (*stack frame*), se ejecutan esas operaciones sobre el cuadro actual.

Cuando un hilo invoca un método Java, la JVM crea e introduce un nuevo cuadro dentro de la pila de programa del hilo actual; este nuevo cuadro se convierte en el cuadro actual, cuando el método se ejecuta, se usa el cuado para almacenar parámetros, variables locales, cálculos intermedios y otros datos.

Un método puede completarse de dos formas. Cuando se termina mediante la instrucción de retorno se dice que **terminó normalmente**, si se termina mediante el lanzamiento de una excepción se dice que **terminó abruptamente,** de cualquiera de las dos formas, la JVM borra y descarta el cuadro de pila del método, el cuadro para el método previo se convierte en el cuadro actual. Todos los datos en la pila de programa del hilo son privados, no hay forma de que un hilo altere o acceda a la pila de programa de otro hilo.

3.4.3.1 Cuadro de pila (stack frame)

El cuadro de pila posee tres partes; **variables locales, pila de operadores y cuadro de datos**, el tamaño de las variables locales y la pila de operadores, las cuales son medidas en palabras, dependen de las necesidades individuales de cada método. Estos tamaños son determinados en tiempo de compilación.

Cuando la JVM invoca un método, revisa los datos de la clase para determinar el número de palabras requeridas por el método en las variables locales y la pila de operadores. Se crea entonces, un cuadro de pila de tamaño apropiado para el método y se inserta en la pila de programa.

3.4.3.2 Variables locales.

La sección de variables locales para el cuadro de pila (*stack frame*), es organizada en un arreglo de cero palabras inicialmente. Las instrucciones que utilizan valores de la sección variables locales, utilizan un índice que comienza con el numero cero, los valores de `int`, `float`, *referencias a objetos y returnAddress* ocupan una entrada en el arreglo de variables locales. Los valores `byte`, `short` *y* `char` son convertidos a enteros antes de insertarlos en el arreglo de variables locales. Los valores `long` y *double* ocupan dos entradas consecutivas al arreglo.

La sección de variables locales contiene los parámetro de los métodos y a las variables locales, el compilador inserta primero los parámetros en el arreglo en el orden en que fueron declarados.

```
class EjemploVariablesLocales
{
	public static int metodoDeClase (int i, long l, float f, double d, char c,
	Object o)
	{
		return 0;
	}

	public int metodoDeInstancia (long l, int i, char c, Vector v)
	{
		return 0;
	}
}
```

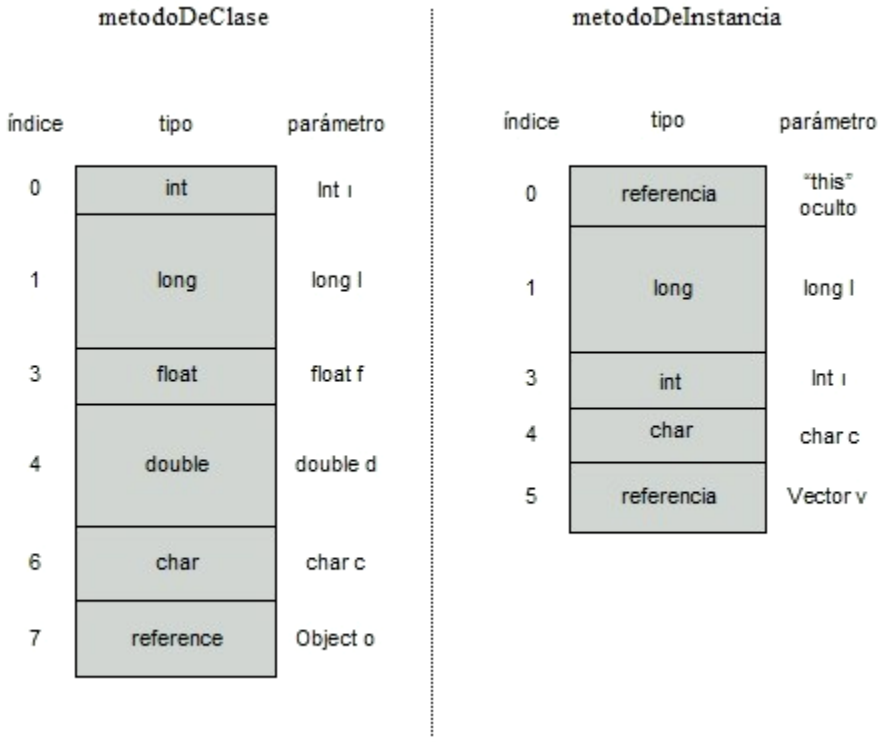

Figura 3-6 Comportamiento de la pila de variables locales, para dos, métodos.

En la figura 3-6 se muestra como actúa la pila de variables locales para dos métodos; note que el primer parámetro para el método `metodoDeInstancia()` es una referencia, aunque ésta no aparece en el código fuente. Esta referencia oculta es pasada para cada instancia de un método.

La instancia del método utiliza la referencia `this` para hacer referencia al objeto actual. Como puede ver, para el método `metodoDeClase(),` no se recibe una referencia oculta `this`, esto es

porque los métodos de clase, es decir; declarados `static`, no se refieren a objetos, no se puede acceder directamente a una variable de instancia desde un método de clase porque no hay instancias asociadas con la invocación del método.

3.4.3.3 Pila de operadores.

Al igual que la sección de *variables locales*, la *pila de operadores* es organizada como un arreglo de palabras, pero no se accede a él mediante índices, sino a través de operaciones *push* y *pop*. Un instrucción *push* introduce un valor dentro de la pila, mientras que una instrucción *pop* utiliza el valor y la retira.

La JVM almacena los mismos tipos de datos en la pila que en la sección de *variables locales*, la JVM es una arquitectura basada en una pila ya que las instrucciones toman sus operadores de la pila de operadores y no de registros. Las instrucciones pueden tomar sus operadores de otros lugares como el *opcode* que sigue inmediatamente después a la instrucción, (cada byte representa una instrucción) o del *constant_pool*.

El conjunto de instrucciones de la JVM mantienen su atención en la pila de operadores, la mayoría de ellos sacan el valor de la pila lo operan y el resultado lo introducen a la pila de operadores.

3.4.3.4 Cuadro de datos.

En adición a las *variables locales* y *pila de operadores*, la *pila de programa* incluye un cuadro donde almacena datos como el tipo de retorno de los métodos y lanzamiento de excepciones. Algunas instrucciones del *set* de instrucciones de la JVM se refieren a entradas al *constant_pool*, algunas otras sólo sacan valores constantes de tipo `int`, `long`, `double` o `String` del *constant_pool*, algunas otras instrucciones de refieren a instancias de clases o arreglos, acceso a campos o invocación de métodos.

Cuando la JVM se encuentra alguna de estas instrucciones que se refieren al *constant_pool*, utiliza el apuntador de cuadro de datos para acceder a la información en el *constant_pool*.

Cuando un método lanza una excepción, la JVM utiliza la tabla de excepciones a la que se refiere el cuadro de datos para determinar el tipo de excepción lanzada, si la JVM encuentra una igualdad entre la cláusula *catch* y la excepción del método, se transfiere el control al comienzo de la cláusula *catch*.

3.4.4 El montículo (heap).

Cuando una instancia de clase o arreglo es creado por una aplicación Java que está siendo ejecutada, la memoria para el nuevo objeto es asignada desde el montículo. Como sólo existe un montículo dentro de la JVM todos los hilos lo comparten, debido a que cada aplicación Java tiene

su instancia exclusiva de la JVM, existen montículos para cada aplicación Java, no hay forma que dos aplicaciones diferentes pueden compartir el área del montículo, sin embargo dos hilos diferentes de la misma aplicación sí pueden compartir dicha área. Esto último es la razón de que exista la sincronización cuando se utilizan *multi – hilos.*

La JVM posee una instrucción para asignar memoria en el montículo para un objeto nuevo – esto se realiza mediante la instrucción `new` – pero no posee una instrucción para liberar memoria.

No se puede liberar un objeto a través de código java. La JVM es responsable de la liberación de memoria ocupada por un objeto al cual no se le hará mas referencia en la aplicación. La JVM utiliza el recolector de basura para administrar el montículo.

3.4.4.1 Recolector de basura (garbage collector).

La función principal del recolector de basura (*garbage collector*) es reclamar la memoria utilizada por los objetos que ya no serán referenciados en la aplicación que se está ejecutando.

El recolector de basura no pertenece a la JVM, la especificación sólo requiere que un administrador se encargue del área de montículo. Por ejemplo, algunas implementaciones pueden simplemente tener una cantidad fija de memoria disponible dentro del montículo y lanzan la excepción `OutOfMemory` cuando el espacio se llena.

El recolector de basura utiliza varias técnicas para poder liberar la memoria, cuando una maquina virtual necesita memoria fresca para cargar clases, el recolector de basura recupera toda la memoria ocupada por objetos sin referencia y se encarga de encontrar y liberar clases sin referencia.

3.4.5 Registro contador de programa (program counter).

Cada hilo de un programa que está siendo ejecutado, posee su propio contador de programa (*program counter*) o "*registro cp*", el cual es creado cuando el hilo comienza. Cuando un hilo ejecuta un método Java, el *registro cp* contiene la dirección de la instrucción que está siendo ejecutada por el hilo. Una dirección puede ser un apuntador nativo o un *offset* desde el comienzo del *bytecode* del método, Si un hilo esta ejecutando un método nativo, el valor del *registro cp* está indefinido.

3.4.6 Pila de métodos nativos e interfaz de métodos nativos.

Además de las áreas definidas por la especificación de la máquina virtual y que han sido descritas anteriormente, un aplicación Java puede usar otras áreas de datos creadas para métodos nativos.

Cuando un hilo invoca un método nativo, entra a un nuevo mundo donde la estructura y las restricciones de seguridad de la JVM ya no existen. Un método nativo probablemente accede a las áreas de datos en tiempo de ejecución de la JVM (depende de la interfaz de métodos nativos), pero además puede usar registros dentro del procesador nativo, asignar memoria o utilizar algún tipo de pila.

Los métodos nativos son dependientes de la implementación, cuando un hilo invoca un método Java, la JVM crea un nuevo cuadro de pila *(stack frame)* y lo inserta en la pila de programa (*java stack)*, cuando un hilo invoca un método nativo, en vez de insertar un nuevo cuadro de pila en la *pila de programa*, la JVM enlaza dinámicamente con el método nativo. Otra forma de indicarlo es que la JVM agrega el código del método nativo a ella.

Si una implementación de una interfaz de método nativo utiliza un modelo como el de enlazamiento del lenguaje C, entonces la pila de método nativo funcionará como una pila C.

3.4.7 Motor de ejecución.

En el núcleo de la JVM se encuentra el *motor de ejecución*. En la especificación de la JVM, el comportamiento del motor de ejecución es definido en términos de su conjunto de instrucciones.

Para cada instrucción, la especificación describe en detalle lo que debe hacer la implementación cuando encuentra la instrucción que ejecuta el *bytecode*.[14]

[14] Las instrucciones están detalladas en el apéndice A.

CAPÍTULO

4

Capítulo 4 El formato del archivo *.class.*

El desensamblador que se desarrolla en esta tesis, **Marago**, es una versión mejorada con respecto a la que presenta **Sun Microsystems** en su JDK (J*ava Development Kit*), la herramienta **javap**. El proceso de ingeniería en reversa que realiza **Marago** está basado en la especificación del archivo *.class.* En este capitulo se describe el formato y las estructuras que conforman al archivo *.class*.

4.1 El archivo .class.

Un archivo *.class* es el resultado de compilar un archivo fuente Java; es un código intermedio llamado *bytecode* y representa al conjunto de instrucciones que ejecutará la Máquina Virtual Java (JVM), además, es la pieza que permite la portabilidad del lenguaje.

Debido a que la Máquina Virtual Java (JVM) se comporta como una pila[1] y debe interpretar el archivo *.class*, almacena más información de la que en realidad necesita; por lo que el proceso de ingeniería en reversa es exitoso en la mayoría de las ocasiones; sin embargo no siempre se puede obtener código fuente a partir del archivo *.class* –en la medida en que los ciclos `for`, `while`, `do` y sentencias `if` son más grandes o se presentan de forma anidada, es más difícil que el proceso de ingeniería en reversa sea exitoso, sin embargo se puede tener una aproximación al código-.

La estructura básica del archivo *.class* consta de una cadena de un byte (8 bits). El resto de las estructuras de datos se construyen mediante la concatenación de dos, cuatro o seis bytes de 8 bits consecutivos. Los datos con múltiples bytes son almacenados en orden *big - endian*, -donde los bytes más altos son almacenados primero-.

Este capítulo describe sus propios tipos de datos que representan al archivo *.class*. Los tipos **u1**, **u2** y **u4** representan uno, dos o cuatro bytes sin signo respectivamente.

El formato del archivo *.class* es presentado usando *pseudo – estructuras* escritas en una notación tipo C, para evitar confusiones con los campos de las clases e instancias de la Máquina Virtual de Java (JVM), el contenido de las estructuras serán referidas como *ítems*; sucesivamente los *ítems*

[1] La descripción y comportamiento de la Máquina Virtual Java se describe en el capítulo 3.

serán almacenados en el archivo *.class* de forma secuencial sin espacios o alineación. Un archivo *.class* presenta la siguiente estructura:

```
classFile
{
    u4                  magic;
    u2                  minor_version;
    u2                  major_version;
    u2                  constant_pool_count;
    cp_info             constant_pool[ constant_pool_count-1 ];
    u2                  access_flags;
    u2                  this_class;
    u2                  super_class;
    u2                  interfaces_count;
    u2                  interfaces [ interfaces_count ];
    u2                  fields_count;
    field_info          fields [ fields_count ];
    u2                  methods_count;
    method_info         methods [ methods_count ];
    u2                  attributes_count;
    attribute_info      attributes[ attributes_count ];
}
```

4.1.1 magic.

El *ítem magic* proporciona el número mágico que identifica el formato del archivo *.class*, su valor es **0xCAFEBABE.**

4.1.2 minor_version y major_version.

Los valores de los *ítems minor_version* y *major_version* son los números de versión menor y mayor del archivo *.class*. Juntos determinan la versión del formato del archivo *.class*. Si el archivo *.class* tiene una versión mayor **M** y una versión menor **m**, denotamos que la versión del archivo *.class* tiene el formato **M.m**. Así la versión del archivo *.class* puede ordenarse lexicográficamente, por ejemplo 1.5 < 2.0 <2.1.

Una implementación de la JVM soporta una versión del archivo *.class* **v** si y sólo si **v** se encuentra en el rango Mi.0 < v < Mj.m. Sólo **Sun Microsystems** puede especificar el rango de versiones.

Para las versiones de Java 1 la JVM normalmente soporta archivos *.class* con número *minor_version* igual a 3 y un número *major_version* igual a 45. Un cambio en el número mayor indica una incompatibilidad, lo que requiere una nueva JVM.

4.1.3 constant_pool_count.

El valor del *ítem constant_pool_count* debe ser más grande que cero. Proporciona el número de entradas en la tabla *constant_pool* del archivo *.class*, donde la entrada en el índice cero está incluida pero no está presente en la tabla *constant_pool*. Un índice *constant_pool* es considerado válido si es más grande que cero y menor que *constant_pool_count*.

4.1.4 constant_pool[].

El *ítem constant_pool* es un arreglo de longitud variable, su estructura representa varias cadenas, nombres de clases, nombres de campos y otras constantes que son referidas dentro de la estructura del archivo *.class* y sus subestructuras.

La primera entrada en el arreglo *constant_pool*, *constant_pool[0]*, está reservada para uso interno de la implementación de la JVM, esa entrada no está presente en el archivo *.class*, la primera entrada en el archivo *.class* es *constant_pool[1]*.

Cada entrada en el arreglo *constant_pool* desde uno hasta *constant_pool_count –1* es una estructura de longitud variable, donde su formato está indicado por su primer byte *tag*[2].

4.1.5 access_flags.

El valor del *ítem access_flags* es una máscara de modificadores utilizada con declaraciones de clases e interfaces. Los modificadores *access_flags* se muestran en la tabla 4.1.

Nombre de la bandera	Valor	Significado	Usado por
ACC_PUBLIC	0x0001	La clase o interfaz es pública, puede ser accesada desde afuera de su paquete.	Clase e interfaz
ACC_FINAL	0x0010	La clase es final, no permite herencia	Clase
ACC_SUPER	0x0020	Llama a los métodos de la superclase utilizando *invokespecial.*	Clase e interfaz
ACC_INTERFACE	0x0200	Es una interfaz.	Interfaz
ACC_ABSTRACT	0x0400	La clases o interfaz es abstracta, no puede ser instanciada	Clase e interfaz

Tabla 4-1 banderas de acceso para el *ítem access_flags.*

[2] Para cada entrada al *constant_pool* hay un byte etiqueta o byte *tag* que define que tipo de dato se almacenará

Una interfaz se distingue por su bandera *ACC_INTERFACE*, si *ACC_INTERFACE* no esta activada, el archivo *.class* es una clase, no una interfaz.

Las interfaces sólo pueden utilizar banderas indicadas en la tabla 4.1 como “Usado por Interfaz”, las clases sólo pueden usar banderas indicadas en la tabla 4.1 como “Usado por clase”. Una interfaz es implícitamente abstracta, por lo que su bandera *ACC_ABSTRACT* debe estar activada.

Una interfaz no puede ser `final`, ya que su implementación no está completamente definida, las banderas *ACC_FINAL* y *ACC_ABSTRACT* no pueden estar activadas al mismo tiempo para una clase, ya que su implementación no podría realizarse.

Todas las banderas sin usar incluyendo las no asignadas en la tabla 4.1 están reservadas para futuras versiones, por lo que deberán usarse en cero y serán ignoradas por la JVM.

4.1.6 this_class.

El valor del *ítem this_class* debe ser un índice válido en la tabla *constant_pool*. La entrada *constant_pool* en ese índice debe ser una estructura *CONSTANT_Class_info*, que representa el nombre de la clase o interfaz definida por el archivo *.class*, además si una clase tiene un paquete, el *ítem this_class* define el nombre completamente calificado de la clase con su paquete.

4.1.7 super_class.

Para una clase, el valor del *ítem super_class* debe ser cero o un índice válido para la tabla *constant_pool*. Si el valor de ítem *super_class* no es cero, el índice de entrada al *constant_pool* debe ser una estructura *CONSTANT_Class_info,* que representa la superclase de la clase definida por el archivo *.class*; es importante notar que ni la superclase o alguna de sus superclases pueden ser clases `final`.

Si el valor de *super_class* es cero entonces el archivo *.class* representa a la clase `java.lang.Object,` la única clase que no tiene una clase padre.

Para una interfaz, el valor de *super_class* debe ser siempre un índice válido para la tabla *constant_pool*, la entrada debe ser una estructura *CONSTANT_Class_info,* que representa a la clase `java.lang.Object.`

4.1.8 interfaces_count.

El valor del *ítem interfaces_count* representa el número de interfaces que implementa el archivo *.class.*

4.1.9 interfaces[].

Cada valor en el arreglo de *interfaces[]* debe ser un índice válido dentro de la tabla *constant_pool.* La entrada al *constant_pool* en cada valor *interfaces[i]*, donde $0 < i <$ *interfaces_count*, debe ser una estructura *CONSTANT_Class_info*, que representa una interfaz.

4.1.10 fields_count.

El valor del *ítem fields_count* representa el número de estructuras *field_info* en el arreglo *fields[]*. La estructura *field_info* representa todos los campos, variables de clase y variables de instancia, declaradas por la clase o la interfaz.

4.1.11 fields[].

Cada valor en el arreglo *fields[]* debe ser una estructura *field_info* de longitud variable. La estructura da una descripción de un campo en la clase o interfaz. El arreglo *fields[]* incluye únicamente los campos que son declarados por la clase o interfaz, no incluye *ítems* que representan campos que fueron heredados de su clase padre o interfaz padre.

4.1.12 methods_count.

El valor del *ítem methods_count* representa el número de estructuras *method_info* en el arreglo *methods[]*.

4.1.13 methods[].

Cada valor en el arreglo *methods[]* debe ser una estructura *method_info* de longitud variable. La estructura da una descripción completa del código de la JVM para un método en la clase o interfaz. Si el método no es nativo o abstracto, la JVM implementa las instrucciones del método.

La estructura *method_info* representa todos los métodos, tanto de instancia para clases como métodos estáticos declarados por la clase o la interfaz. El arreglo *methods[]* incluye únicamente aquellos métodos que son declarados explícitamente por la clase, no incluye métodos que fueron heredados de la clase padre o interfaz padre.

4.1.14 attributes_count.

El valor del *ítem attributes_count* representa el número de atributos en la tabla *attributes[]* de la clase.

4.1.15 attributes[].

Cada valor del arreglo *attributes[]* debe ser una estructura *attribute_info* de longitud variable. Los únicos atributos definidos por la especificación de la JVM que aparecen en el arreglo *attributes[]* o la estructura *classFile* son los atributos *SourceFile y Deprecated.*

La JVM ignora todos los atributos en el arreglo *attributes* de la estructura c*lassFile* que no reconoce, los atributos no definidos en la especificación no son permitidos y no tienen efecto en la semántica del archivo *.class*, sólo proveen información descriptiva adicional.

4.2 Estructura del constant_pool.

Todas las entradas en el arreglo *constant_pool* tiene el siguiente formato general:

```
cp_info
{
      u1   tag;
      u1   info[];
}
```

Cada *ítem* en el arreglo *constant_pool* debe comenzar con un *tag* (1 byte) indicando el tipo de entrada *cp_info*. El contenido del arreglo *info* varia con el valor del *tag*. Los *tag* válidos y sus valores están listados en la tabla 4.2.

Tipo de constante	Valor
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1

Tabla 4-2 Valores de las etiquetas (*tags*) para el *constant_pool.*

Cada *tag* está seguido por dos o más bytes que dan información acerca de la constante específica, el formato de la información adicional varía con el valor del *tag*.

4.2.1 CONSTANT_Class.

La estructura *CONSTANT_Class_info* es utilizada para representar una clase o una interfaz:

```
CONSTANT_Class_Info
{
     u1     tag;
     u2     name_index;
}
```

4.2.1.1 tag.

El *ítem tag* tiene el valor *CONSTANT_Class(7).*

4.2.1.2 name_index.

El valor del *ítem name_index* debe ser un índice válido dentro de la tabla *constant_pool*, el índice debe ser una estructura *CONSTANT_Utf8_info*, que representa un nombre de clase válido que ha sido convertido a la forma interna del archivo *.class*.

4.2.2 CONSTANT_Fieldref, CONSTANT_Methodref y CONSTANT_InterfaceMethodref.

Los campos, métodos e interfaces están representados por estructuras similares: *CONSTANT_Fieldref*, *CONSTANT_Methodref* y *CONSTANT_InterfaceMethodref*.

```
CONSTANT_Fieldref_info
{
     u1     tag;
     u2     class_index;
     u2     name_and_type_index;
}


CONSTANT_Methodref_info
{
     u1     tag;
     u2     class_index;
     u2     name_and_type_index;
}


CONSTANT_InterfaceMethodref_info
{
     u1     tag;
     u2     class_index;
     u2     name_and_type_index;
}
```

4.2.2.1 tag,

El *ítem tag* de la estructura *CONSTANT_Fieldref_info* tiene el valor *CONSTANT_Fieldref(9)*, para la estructura *CONSTANT_Methodref_info* el *tag* tiene el valor *CONSTANT_Methodref(10)*; finalmente para la estructura *CONSTANT_InterfaceMethodref_info* tiene el valor *CONSTANT_InterfaceMethodref(11)*.

4.2.2.2 class_index.

El valor del *ítem class_index* debe ser un índice válido en el arreglo *constant_pool*. El índice de entrada al *constant_pool* debe ser una estructura *CONSTANT_Class_Info*, que representa el tipo de clase o interfaz que contiene la declaración del campo o método.

El *ítem class_index* de la estructura *CONSTANT_Methodref_info* debe ser una clase y no una interfaz, el *ítem class_index* de la estructura *CONSTANT_InterfaceMethodref_info* debe ser una interfaz y el *ítem class_index* de la estructura *CONSTANT_Fieldref_info* puede ser una clase o interfaz.

4.2.2.3 name_and_type_index.

El valor del *ítem name_and_type_index* debe ser un índice válido dentro del arreglo *constant_pool*.

El índice de entrada al *constant_pool* debe ser una estructura *CONSTANT_NameAndType_info*, ésta entrada indica el nombre y descriptor del campo o método.

Si el nombre del método de *CONSTANT_Methodref_info* o *CONSTANT_InterfaceMethodref_info* comienza con *'<' ('u003c')*, entonces el nombre debe ser uno de los métodos internos especiales <*init*>[3] o <*clinit*>[4], en este caso el método no debe regresa algún valor.

4.2.3 CONSTANT_String.

La estructura *CONSTANT_String_Info* es usada para representar objetos del tipo `java.lang.String`.

```
CONSTANT_String_Info
{
    u1      tag;
    u2      string_index;
}
```

4.2.3.1 tag.

El *ítem tag* de la estructura *CONSTANT_String_Info* tiene el valor *CONSTANT_String_Info(8)*.

4.2.3.2 string_index.

El valor del *ítem string_index* debe ser un índice válido dentro del arreglo *constant_pool*. El índice de entrada al *constant_pool* debe ser una estructura *CONSTANT_Utf8_info*, que representa la secuencia de caracteres para que el objeto `java.lang.String` sea iniciado.

[3] El método <*init*> es el constructor de la clase y aparece siempre en el archivo *.class* aunque no se haya escrito uno en código fuente.
[4] El método <*clinit*> hace referencia al bloque estático.

4.2.4 CONSTANT_Integer y CONSTANT_Float.

Las estructuras *CONSTANT_Integer* y *CONSTANT_Float* representan constantes numéricas (`int` y `float`) de 4 bytes sin signo.

```
CONSTANT_Integer_info
{
    u1      tag;
    u4      bytes;
}


CONSTANT_Float_info
{

    u1      tag;
    u4      bytes;
}
```

4.2.4.1 tag.

El *ítem tag* de la estructura *CONSTANT_Integer_info* posee el valor *CONSTANT_Integer(3)* y para la estructura *CONSTANT_Float_info* posee el valor *CONSTANT_Float(4).*

4.2.4.2 bytes.

El *ítem bytes* de la estructura *CONSTANT_Integer_info* contiene el valor de la constante `int`. Los bytes del valor son almacenados en orden *big – endian* (Primero el byte mas alto).

El *ítem bytes* de la estructura *CONSTANT_Float_info* contiene el valor de la constante `float` en punto flotante siguiendo el formato IEEE 754. Los bytes son almacenados en orden *big – endian* y primero son convertidos a un argumento `int`, entonces:

- Si el argumento es 0x7f800000, el valor `float` será infinito positivo
- Si el argumento es 0xff800000, el valor `float` será infinito negativo
- Si el argumento está en el rango 0x7f800001 hasta 0x7fffffff o en el rango 0xff800001 hasta 0xffffffff, el valor `float` será NaN[5].
- En otros casos, sea *s*, *e* y *m* tres valores que pueden ser calculados por:

```
int s = ((bytes >>31)==0)? 1: -1;
int e = ((bytes >>23)&0xff);
```

[5] NaN *Not a Number*

```
int m = (e == 0)?
      (bytes & 0x7fffff)<<1:
      (bytes & 0x7fffff) | 0x800000;
```

entonces el valor `float` es igual al resultado de la expresión matemática:

$$s * m * 2^{e-150}$$

4.2.5 CONSTANT_Long y CONSTANT_Double.

CONSTANT_Long_info y *CONSTANT_Double_info* representan constantes numéricas de 8 bytes (`long` y `double`).

```
CONSTANT_Long_info
{
    u1      tag;
    u4      high_bytes;
    u4      low_bytes;
}
```

```
CONSTANT_Double_info
{
    u1      tag;
    u4      high_bytes;
    u4      low_bytes;
}
```

Todas las constantes de 8 bytes toman dos entradas en el arreglo *constant_pool* del archivo *.class*. Si la estructura *CONSTANT_Long_info* o *CONSTANT_Double_info* es un *ítem* con un índice *n* en la tabla *constant_pool*, entonces el siguiente *ítem* válido en el *pool* es localizado en el índice *n + 2*. El índice *n +1* debe ser considerado inválido y no debe ser usado.

4.2.5.1 tag.

El *ítem tag* de la estructura *CONSTANT_Long_info* posee el valor *CONSTANT_Long(5)* y para la estructura *CONSTANT_Double_info* el *tag* posee el valor *CONSTANT_Double(6).*

4.2.5.2 high_bytes, low_bytes.

Los *ítems* sin signo *high_bytes* y *low_bytes* de las estructura *CONSTANT_Long_info* contienen juntos el valor de la constante `long` *((long)high_bytes << 32 + low_bytes*, donde los bytes de cada uno son almacenados en orden *big – endian.*

Los ítems *high_bytes* y *low_bytes* de la estructura *CONSTANT_Double_info* contienen el valor `double` en formato IEEE 754, los bytes de cada *ítem* son almacenados en orden *big – endian*. Los ítems *high_bytes* y *low_bytes* son convertidos a argumentos `long,` entonces:

- Si el argumento es 0x7f80000000000000L, el valor `double` será infinito positivo.
- Si el argumento es 0xff80000000000000L, el valor `double` es infinito negativo.
- Si el argumento está en el rango 0x7ff0000000000001L hasta 0x7fffffffffffffffL o en el rango 0xfff0000000000001L hasta 0xffffffffffffffffL, el valor `double` será NaN.
- En cualquier otro caso, sea *s*, *e* y *m* tres valores que pueden ser calculados por:

```
int s = (( bits >>63) == 0)?1:-1;
int e = (int) ((bits >>52 )& 0x7ffL;
long m = (e == 0) ?
      (bits & 0xfffffffffffffL)<<1:
      (bits & 0xfffffffffffffL) | 0x10000000000000L;
```

Entonces el valor en punto flotante es:

$$s * m * 2^{e-1075}$$

4.2.6 CONSTANT_Name_And_Type.

La estructura *CONSTANT_Name_And_Type_info* es usada para representar un campo o método, sin indicar la clase o interfaz a la que pertenece.

```
CONSTANT_NameAndType_info
{
    u1      tag;
    u2      name_index;
    u2      descriptor_index;
}
```

4.2.6.1 tag.

El *ítem tag* de la estructura *CONSTANT_NameAndType_info* posee el valor *CONSTANT_NameAndType(12)*

4.2.6.2 Name_index.

El valor del *ítem name_index* debe ser un índice válido en el arreglo *constant_pool*. El índice de entrada al *constant_pool* debe ser una estructura *CONSTANT_Utf8_info*, que representa un nombre de método o campo Java válido, almacenado como un nombre simple (no completo) o como un identificador Java.

4.2.6.3 Descriptor_index.

El valor del *ítem descriptor_index* debe ser una entrada válida al *constant_pool*, el índice de entrada debe ser una estructura *CONSTANT_Utf8_info*, que representa un descriptor de campo Java válido o un descriptor de método.

4.2.7 CONSTANT_Utf8.

La estructura *CONSTANT_utf8_info* es usada para representar valores de constantes `String`. Las cadenas Utf8 son codificadas como secuencias de caracteres que contienen únicamente caracteres ASCII no nulos que son representados sólo por un byte por carácter, pero los caracteres de más de 16 bits también pueden ser representados. Todos los caracteres en el rango u0001 a u07ff son representados por un sólo byte:

0 bits 6 – 0

Los siete bits de datos en el byte dan el valor del carácter representado. El carácter nulo *'u0000'* y caracteres en el rango *'u0080'* a '*u07ff*' son representados por un par de bytes *x* y *y*.

x:110 bits 10 – 6
y: 10 bits 5 – 0

Los bytes representan el carácter con el valor ((x&0x1f)<<6 + (y& 0x3f).

Caracteres en el rango 'u0800' a 'uffff' son representados por tres bytes *x*, *y* y *.z*

x:1110 bits 15 –12
y: 10 bits 11 – 6
z: 10 bits 5 – 0

El carácter con el valor ((x& 0xf) <<12 + ((y & 0x3f)<<6 + (z&0x3f) es representado por éstos bytes.

Los bytes de caracteres multibytes son almacenados en el archivo *.class* en orden *big – endian*.

Hay dos diferencias entre este formato y el estándar UTF8. Primero, el byte nulo (byte) 0 es codificado utilizando un formato de dos bytes. Segundo, sólo los formatos *one – byte*, *two – bytes y three - bytes* son utilizados, la JVM no reconoce otros formatos.[6]

La estructura CONSTANT_Utf8_info es la siguiente:

```
CONSTANT_Utf8_info
{
    u1      tag;
    u2      length;
    u1      bytes[ length ];
}
```

4.2.7.1 tag.

El *ítem tag* de la estructura *CONSTANT_Utf8_info* posee el valor *CONSTANT_Utf8(1)*

4.2.7.2 length.

El valor del *ítem length* representa el número de bytes en el arreglo *bytes[]* (no la longitud de la cadena resultante). La cadena en la estructura *CONSTANT_Utf8_info* no termina en `null`

4.2.7.3 bytes [].

El arreglo *bytes* contiene los bytes de la cadena, los bytes que no pueden tener valor son (`byte`) 0 o (`byte`) 0xf0 – (`byte`)0xff.

[6] Para más información consulte : File System Safe UCS Transformation Format (FSS_UTF), X/Open Preliminary Specification (X/Open Company Ltd. Document Number P316).

4.3 Campos.

Cada campo es descrito por una estructura *field_info* de longitud variable. El formato de esta estructura es:

```
field_info
{
    u2      access_flags;
    u2      name_index;
    u2      descriptor_index;
    u2      attributes_count;
    attribute_info attributes[ attributes_count ];
}
```

4.3.1 access_flags.

El valor del *ítem access_flags* es una mascara de modificadores usados para describir los permisos de acceso y las propiedades de un campo. Los modificadores *access_flags* se muestran en la tabla 4.3.

Nombre de Bandera	Valor	Significado	Usado por
ACC_PUBLIC	0x0001	Es public; puede ser accesado desde otros paquetes	Cualquier campo
ACC_PRIVATE	0x0002	Es private; usado dentro de la definición de clase	Clases y Campos
ACC_PROTECTED	0x0004	Es protected; puede ser accesado desde sus subclases	Clases y Campos
ACC_STATIC	0x0008	Es static.	Cualquier campo
ACC_FINAL	0x0010	Es final; no puede ser sobrescrito o reasignado después de su inicialización	Cualquier campo
ACC_VOLATILE	0x0040	Es volatile; no puede ser atrapado	Clases y Campos
ACC_TRANSIENT	0x0080	Es transient; no puede ser escrito o leído por un manejador de objetos persistentes	Clases y Campos

Tabla 4-3 Banderas de acceso a los campos.

Los campos de las interfaces sólo pueden usar banderas indicadas en la tabla 4.3 como: “usadas por campos”, mientras que los campos de las clases pueden usar cualquier bandera de la tabla 4.3

Todos los bits sin utilizar del ítem *access_flags*, incluyendo aquellos no asignados en la tabla 4.3 son reservados para el futuro, siempre tienen el valor de cero y son ignorados por la JVM.

Los campos de clase pueden tener sólo una de las banderas *ACC_PUBLIC, ACC_PROTECTED o ACC_PRIVATE* activada; un campo de clase no puede tener las banderas *ACC_FINAL* y *ACC_VOLATILE* activadas.

Cada campo de una interfaz es implícitamente `static` y `final` y debe tener ambas banderas activadas *ACC_FINAL* y *ACC_STATIC*; cada campo de una interfaz es implícitamente `public` y deberá tener la bandera *ACC_PUBLIC* activada.

4.3.2 name_index.

El valor del *ítem name_index* deberá ser un índice válido en la tabla *constant_pool*. El índice de entrada al *constant_pool* es una estructura *CONSTANT_Utf8_info,* que representa un nombre de campo Java válido almacenado como un nombre simple, esto es, como un identificador Java.

4.3.3 descriptor_index

El valor del *ítem descriptor_index* deberá ser un índice válido en la tabla *constant_pool*, el índice de entrada *constant_pool* es estructura *CONSTANT_Utf8_info,* que representa un descriptor de campo Java válido.

4.3.4 attributes_count.

El valor del *ítem Attributes_count* indica el número de atributos adicionales de este campo.

4.3.5 attributes[].

Cada valor del arreglo *attributes[]* debe ser una estructura *attributes* de longitud variable. Un campo puede tener varios atributos asociados a él.

Los atributos definidos por la especificación que aparecen en el arreglo *attributes[]* de la estructura *field_info* son *ConstantValue*, *Synthetic* y *Deprecated*. La implementación de la JVM ignora el resto de atributos en la tabla *attributes[]* que no reconoce. Los atributos no definidos en la especificación no pueden afectar la semántica del archivo *.class,* sólo proveen información adicional.

4.4 Métodos.

Cada método incluyendo cada inicialización de métodos de instancia, es descrita por una estructura *method_info*, donde dos métodos en una misma clase no pueden tener el mismo nombre y descriptor. La estructura tiene el siguiente formato:

```
method_info
{
     u2           access_flags:
     u2           name_index:
     u2           descriptor_index;
     u2           attributes_count
     attribute_info attributes[ attributes_count ]
}
```

4.4.1 access_flags.

El valor del *ítem access_flags* es una máscara de modificadores usados para describir los permisos de acceso y las propiedades de un método o inicialización de instancia de método. La tabla 4.4 muestra los modificadores *access_flags.*

Nombre de bandera	Valor	Significado
ACC_PUBLIC	0x0001	Es public; puede ser accesado desde fuera de su paquete.
ACC_PRIVATE	0x0002	Es private; usado sólo dentro de su definición de clase.
ACC_PROTECTED	0x0004	Es protected puede ser accesado dentro de sus subclases.
ACC_STATIC	0x0008	Es static.
ACC_FINAL	0x0010	Es final; no puede ser sobrescrito.
ACC_SYNCHRONIZED	0x0020	Es synchronized; un monitor atrapa su implementación.
ACC_NATIVE	0x0100	Es native; implementa en java otro lenguaje.
ACC_ABSTRACT	0x0400	Es abstract; no esta provista la implementación
ACC_STRICT	0x0800	Método declarado strict

Tabla 4-4 Banderas de acceso para los métodos.

Los métodos de las clases pueden utilizar cualquier bandera, pero un método en específico sólo puede tener una de estas banderas activada a la vez: *ACC_PUBLIC*, *ACC_PROTECTED* o *ACC_PRIVATE.*

Si dicho método tiene su bandera *ACC_ABSTRACT* activada no puede tener activadas las siguientes banderas: ACC*_FINAL, ACC_NATIVE, ACC_PRIVATE, ACC_STATIC, ACC_STRICT o ACC_SYNCHRONIZED.*

Todos los métodos en las interfaces sólo tienen dos banderas activadas *ACC_ABSTRACT* y *ACC_PUBLIC*.

Todos los bits sin utilizar del ítem *access_flags*, incluyendo aquellos que no estén asignados en la tabla 4.4 son reservados para el futuro y tiene valor de cero, además son ignorados por la implementación de la JVM.

4.4.2 name_index.

El valor del *ítem name_index* deberá ser un índice válido en la tabla *constant_pool*, el índice de entrada al *constant_pool* es una estructura *CONSTANT_Utf8_Info*, que representa uno de los nombres especiales de los métodos internos <*init*> o <*clinit*> o un nombre de método Java válido almacenado como un nombre sencillo.

4.4.3 descriptor_index.

El valor del *ítem descriptor_index* es un índice válido en la tabla *constant_pool*. La entrada *constant_pool* es una estructura *CONSTANT_Utf8_info*, la cual representa un descriptor de método válido.

4.4.4 attributes_count.

El valor del *ítem attributes_count* indica el número de atributos adicionales del método.

4.4.5 attributes[].

Cada valor del arreglo *attributes[]* deberá ser una estructura de longitud variable de tipo *attribute*; un método puede tener cualquier numero de atributos asociados a el.

Los atributos definidos por la especificación que aparecen en el arreglo *attributes[]* de la estructura *method_info* son : *Code, Exceptions, Synthetic* y *Deprecated*.

Una implementación de la JVM ignora todos los atributos en el arreglo *attributes[]* de la estructura *method_info* que no puede reconocer. Los atributos no definidos en la especificación no pueden afectar la semántica del archivo *.class*, sólo proveen información adicional.

4.5 Atributos.

Los atributos son utilizados en las estructuras *classFile*, *field_info*, *method_info* y c*ode_attribute* del archivo *class*. Todos los atributos tienen la siguiente estructura en general:

```
attribute_info
{
     u2     attribute_name_index;
     u4     attibute_length;
     u1     info[ attribute_length ];
}
```

Para todos los atributos, *attribute_name_index* deberá ser un índice válido de 16 bits sin signo dentro del *constant_pool* en el archivo *.class*. La entrada al *constant_pool* en *attribute_name_index* es la estructura *CONSTANT_Utf8*, que representa el nombre del atributo. El valor del *ítem attribute_length* indica la longitud de la estructura *info[]*. La longitud no incluye los seis bytes iniciales que contienen los *ítems attribute_name_index* y *attribute_length.*

Ciertos atributos son predefinidos como parte de la especificación del archivo *.class.* Los atributos predefinidos son *SourceFile, ConstantValue, Code, Exceptions, InnerClasses, Synthetic, LineNumberTable, LocalVariableTable* y *Deprecated*. Los nombres de los atributos predefinidos están reservados.

De los atributos predefinidos, *Code*, *ConstantValue* y *Exceptions* deben ser reconocidos y leídos correctamente por un lector de archivos *.class* para que la JVM los interprete correctamente. Los atributos *InnerClasses* y *Synthetic* deben ser reconocidos y leídos correctamente para que la JVM implemente adecuadamente las librerías de la plataforma Java 2.

4.5.1 Definiendo y nombrando nuevos atributos.

Los compiladores para código fuente Java tienen permitido definir y emitir archivos *.class* que contengan nuevos atributos en el arreglo *attributes[]* de la estructura del archivo *.class.* La implementación de la JVM tiene permitido reconocer y usar los nuevos atributos encontrados en el arreglo *attributes[]* del archivo *class;* de cualquier manera, todos los atributos no definidos como parte de la especificación de la JVM no tendrán efecto en la semántica de la clase o interfaz. La JVM ignora atributos que no reconoce.

La JVM tiene prohibido lanzar excepciones o rehusarse a utilizar archivos *.class* simplemente porque existen nuevos atributos, obviamente las herramientas operativas del archivo *.class* pueden correr adecuadamente si no contiene todos los atributos necesarios.

Debido a que podemos definir atributos, sería fácil crear atributos que coincidieran en el nombre y la longitud, esto provocaría conflictos en la implementación para reconocer los atributos; debido a esto, **Sun Microsystems** acordó una convención para nombrar a los paquetes definido por la especificación del lenguaje Java. Un nuevo atributo definido por **Netscape**, por ejemplo, tendría el nombre **"COM.Nestcape.new-attribute"**.

4.5.2 El atributo SourceFile.

El atributo *SourceFile* es un atributo opcional de longitud fija en el arreglo *attribute[]* de la estructura c*lassFile* y es único; no puede haber más de un atributo *SourceFile.*

```
SourceFile
{
     u2     attribute_name_index;
     u4     attribute_length;
     u2     sourcefile_index;
}
```

4.5.2.1 attribute_name_index.

El valor del *ítem attribute_name_index* deberá ser un índice válido en la tabla *constant_pool*. El índice de entrada al *constant_pool* es una estructura *CONSTANT_Utf8_info* que representa a la cadena "SourceFile".

4.5.2.2 attribute_length.

El valor del *ítem attribute_length* de la estructura *SouceFile_attribute* debe ser 2.

4.5.2.3 sourcefile_index.

El valor del *item sourcefile_index* debe ser un índice válido dentro de la tabla *constant_pool* . El índice de entrada al *constant_pool* es una estructura *CONSTANT_Utf8_info* que representa la cadena del nombre del archivo fuente del cual, el archivo *.class* fue compilado.

El atributo *sourcefile_index* sólo representa el nombre del archivo fuente, nuca representa el nombre del directorio que contiene al archivo o la ruta del archivo.

4.5.3 El atributo ConstantValue

El atributo *ConstantValue*, es un atributo de longitud fija usado en el arreglo *attribute[]* de la estructura *field_info*, que representa el valor de un campo constante que debe ser (implícitamente o explícitamente) `static`; esto es, el bit *ACC_STATIC* (tabla 4.3) debe estar activado. El campo no se requiere para ser `final`.

No puede haber más de un atributo *ConstantValue* en la tabla dada por la estructura *field_info*. El campo constante representado por la estructura *field_info* es asignado al valor referenciado por su atributo *ConstantValue* como parte de su inicialización. Toda máquina virtual debe reconocer el atributo *ConstantValue*.

```
ConstantValue_attribute
{
     u2    attribute_name_index;
     u4    attribute_length;
     u2    constantValue_index;
}
```

4.5.3.1 attribute_name_index.

El valor del *ítem* debe ser un índice válido dentro de la tabla *constant_pool*. El índice de entrada al *constant_pool* es una estructura *CONSTANT_Utf8_info* que representa a la cadena "ConstantValue".

4.5.3.2 attribute_length.

El valor del *ítem attribute_length* de la estructura *ConstantValue_attribute* debe ser 2.

4.5.3.3 constantvalue_index.

El valor del *ítem constantvalue_index* debe ser un índice válido dentro de la tabla *constant_pool*. El índice de entrada al *constant_pool* debe ser el valor constante representado por su atributo. El índice de entrada en el *constant_pool* es de un tipo apropiado al campo, como se muestra en la tabla 4.5.

Tipo de Campo	Tipo de Entrada
long	CONSTANT_Long
float	CONSTANT_Float
double	CONSTANT_Double
int, short, char, byte, Boolean	CONSTANT_Integer
java.lang.String	CONSTANT_String

Tabla 4-5 Valores para el ítem *constanValue_index.*

4.5.4 El atributo Code.

El atributo *Code* es una estructura de longitud variable usada en la tabla *attribute* de la estructura *method_info*. *Code* contiene las instrucciones de la JVM e información auxiliar para un método Java, inicialización de método de instancia o inicialización de clase o interfaz. Cada implementación de una máquina virtual debe reconocer el atributo *Code*. Debe haber exactamente un atributo *Code* en cada estructura *method_info.* El atributo *Code* tiene el siguiente formato:

```
Code_attribute
{
     u2           attribute_name_index;
     u4           attribute_length;
     u2           max_stack;
     u2           max_locals;
     u4           code_length;
     u1           code[ code_length ];
     u2           exception_table_length;
     {
           u2     start_pc;
           u2     end_pc;
           u2     handler_pc;
           u2     catch_type;
     }            exception_table[ exception_table_length ];
     u2           attributes_count;
     attribute_info attributes [ attribute_count ];
}
```

4.5.4.1 attribute_name_index.

El valor del *ítem attribute_name_index* debe ser un índice válido en la tabla *constant_pool.* El índice de entrada al *constant_pool* es una estructura *CONSTANT_Utf8_info* que representa la cadena “Code”.

4.5.4.2 attribute_length.

El valor del *ítem* indica la longitud de los atributos, excluyendo los 6 bytes iniciales.

4.5.4.3 max_stack.

El valor del *ítem max_stack* representa el máximo número de palabras en la pila de operadores en cualquier punto durante la ejecución del método.

4.5.4.4 max_locals.

El valor del *ítem max_locals* representa el número de variables locales usadas por el método, incluyendo los parámetros pasados en la invocación del método. El índice de la primera variable local es 0. El índice más grande de una variable local para un valor de una palabra es *max_locals* – 1. El índice más grande de una variable local para un valor de dos palabras es *max_locals* – 2.

4.5.4.5 code_length.

El valor del *ítem code_length* representa el número de bytes en el arreglo *Code[]*para este método. El valor de *code_length* debe ser más grande que cero; el arreglo *Code[]*no debe estar vacío.

4.5.4.6 code[].

El arreglo *code[]*contiene los bytes actuales del código de la JVM que implementan al método.

4.5.4.7 exception_table_length.

El valor del *ítem exception_table_length* representa el número de entradas en el arreglo *exception_table[]*.

4.5.4.8 exception_table[].

Cada entrada en el arreglo *exception_table* describe una excepción manejada en el arreglo *code[]*. Cada entrada *exception_table* contiene los siguientes *ítems*.

- star_pc, end_pc

  El valor de los dos *ítems* indican el rango en el arreglo *code[]*donde el manejo de la excepción esta activo. El valor de *start_pc* debe ser un índice válido dentro del arreglo *code[]*. El valor de *end_pc* también debe ser un índice válido en el arreglo *code*, o debe ser igual a *code_length*, la longitud del arreglo *code[]*. El valor de *start_pc* debe ser menor que el valor *end_pc.*

  start_pc es inclusivo y *end_pc* es exclusivo; esto es, el manejador de excepciones debe estar activo mientras el *program counter* esta dentro del intervalo *[star_pc, end_pc).*

- handler_pc

  El valor del *ítem handler_pc* indica el comienzo del manejador de excepciones. El valor del *ítem* debe ser un índice válido en el arreglo *code[]*, debe ser el índice del *opcode* de una instrucción y debe ser menor que el valor del ítem *code_length.*

- catch_type

  Si el valor del *ítem catch_type* no es cero, debe ser un índice válido en la tabla *constant_pool*. La entrada *constant_pool* en el índice es una estructura *CONSTANT_Class_info* que representa una clase de excepciones que el manejador esta diseñado para atrapar. Esta clase debe ser la clase `Throwable` o una de sus subclases. El manejador de excepciones será llamado sólo si la excepción es una instancia de su clase o una de sus subclases.

  Si el valor de *catch_type* es cero, el manejador de excepciones es llamado para todas las excepciones, esto es usado para implementar *finally*.

4.5.4.9 attributes_count.

El valor del *ítem attributes_count* indica el número de atributos del atributo *code[]*.

4.5.4.10 attributes[].

Cada valor del arreglo *attributes* es una estructura *attribute* de longitud variable. El atributo *Code* puede tener cualquier número de atributos opcionales asociados a él. Actualmente los atributos *LineNumberTable* y *LocalVariableTable* contienen información de para desarrollo; son definidos y usados con el atributo *Code*.

4.5.5 El atributo Exceptions.

El atributo *Exceptions* es un atributo de longitud variable usado en el arreglo *attributes[]* de la estructura *method_info*. *Exceptions* indica que excepciones puede lanzar un método. Debe haber exactamente un atributo *Exception* en cada estructura *method_info*.

El atributo *Exceptions* tiene la siguiente forma:

```
exceptions_attribute
{
     u2    attribute_name_index;
     u4    attribute_length;
     u2    number_of_exceptions;
     u2    exception_index_table [ number_of_exceptions ];
}
```

4.5.5.1 attribute_name_index.

El valor del *ítem attribute_name_index* debe ser un índice válido dentro de la tabla *constant_pool*. El índice de entrada en el *constant_pool* es una estructura *CONSTANT_Utf8_info* que representa la cadena “Exception”.

4.5.5.2 attribute_length.

El valor del *ítem attribute_length* indica la longitud del atributo, excluyendo los 6 bytes iniciales.

4.5.5.3 number_of_exceptions.

El valor del *ítem number_of_exceptions* indica el número de entradas en la tabla *exception_index_table*.

4.5.5.4 exception_index_table[].

Cada valor que no sea cero en el arreglo *exception_index_table*, debe ser un índice válido en la tabla *constant_pool*. Para cada *ítem* del arreglo, si *exception_index_table[i]* != 0, donde 0 < i < *number_of_exceptions*, entonces la entrada *constant_pool* en el índice *exception_index_table[i*] es un estructura *CONSTANT_Class_info* que representa un tipo de clase que el método declara para lanzar.

Un método puede lanzar una excepción si al menos uno de los tres criterios sucede:

- La excepción es una instancia de *RuntimeException* o una de sus subclases.
- La excepción es una instancia de *Error* o una de sus subclases.
- La excepción es una instancia de una de las clases de excepciones especificada en la tabla *exception_index_table* o una de sus subclases.

4.5.6 El atributo LineNumberTable.

El atributo *LineNumberTable* es un atributo opcional de longitud variable en el arreglo *attributes[]* del atributo *Code*. Puede ser utilizado por desarrolladores para determinar que parte del código de la JVM corresponde a un número de línea dado en el archivo fuente original.

Si los atributos *LineNumberTable* son presentados en la tabla *attribute* de un atributo *Code* dado, entonces pueden aparecer en cualquier orden, además múltiples atributos *LineNumberTable* pueden representar una línea dada de código Java; esto es, no necesariamente el atributo *LineNumberTable* es uno a uno con las líneas del código fuente.

```
LineNumberTable_attribute
{
     u2     attribute_name_index;
     u4     attribute_length;
     u2     line_number_table_length;
     {
            u2     start_pc;
            u2     line_number;
     }      line_number_table[ line_number_table_length ];
}
```

4.5.6.1 attribute_name_index.

El valor del *ítem attribute_name_index* debe ser un índice válido dentro de la tabla *constant_pool*. El índice de entrada en el *constant_pool* es una estructura *CONSTANT_Utf8_info*, que representa a la cadena “LineNumberTable”.

4.5.6.2 attribute_length.

El valor del *ítem* indica la longitud del atributo, excluyendo los 6 bytes iniciales.

4.5.6.3 line_number_table_length.

El valor del *ítem line_number_table_length* indica el número de entradas en el arreglo *line_number_table*.

4.5.6.4 line_number_table[].

Cada entrada en el arreglo *line_number_table* indica que el número de línea en el archivo original cambia en el punto dado en el arreglo *code[]*. Cada entrada debe contener los siguientes ítems.

- start_pc.

  El valor de *star_pc* indica el índice dentro del arreglo *code[]*cuyo código para una nueva línea en el archivo fuente original comienza. El valor de *start_pc* debe ser menor que el valor del ítem *code_length* del atributo *Code*.

- line_number.

  El valor del *ítem* debe dar el correspondiente número de línea en el archivo fuente original.

4.5.7 El atributo LocalVariableTable.

El atributo *LocalVariableTable* es un atributo opcional de longitud variable del atributo *Code*. Puede ser usado por desarrolladores para determinar el valor de una variable local durante la ejecución de un método. Si los atributos *LocalVariableTable* son presentados en la tabla *attribute* del atributo *Code* dado, entonces aparecerán en cualquier orden. No hay más de un atributo *LocalVariableTable* por variable local en el atributo *Code*.

```
LocalVariableTable_attribute
{
     u2 attribute_name_index;
     u4 attribute_length;
     u2 local_variable_table_length;
     {
        u2    start_pc;
        u2    length;
        u2    name_index;
        u2    descriptor_index;
        u2    index;

     }  local_variable_table[ local_variable_table_length ];
}
```

4.5.7.1 attribute_name_index.

El valor del *ítem attribute_name_index* debe ser un índice válido dentro la tabla *constant_pool*. El índice de entrada en el *constant_pool* es una estructura *CONSTANT_Utf8_info* que representa a la cadena “LocalVariableTable”.

4.5.7.2 attribute_length.

El valor del *ítem* representa la longitud del atributo, excluyendo los 6 bytes iniciales.

4.5.7.3 local_variable_table_length.

El valor del *ítem local_variable_table_length* indica el número de entradas en el arreglo *local_variable_table*.

4.5.7.4 local_variable_table[].

Cada entrada en el arreglo *local_variable_table* indica un rango del arreglo *code[]* dentro del cual una variable local tienen valor. También indica el índice en las variables locales del cuadro actual donde la variable local puede ser encontrada. Cada entrada debe contener los siguientes *ítems*.

- start_pc, length.

  La variable local dada debe tener un valor en el índice dentro del arreglo *code[]* en el intervalo [*start_pc*, *start_pc* + *length*], esto es, entre *start_pc* y *start_pc* + *length* inclusivo. El valor de *start_pc* debe ser un índice válido dentro del arreglo *code[]*. El valor *start_pc* + *length* debe ser también un índice válido en el arreglo *code[]*.

- name_index, descriptor_index.

  El valor del *ítem name_index* debe ser un índice válido dentro de la tabla *constant_pool*. La entrada en el índice debe contener la estructura *CONSTANT_Utf8_info* que representa el nombre de la variable local almacenada en el archivo *.class*.

  El valor del *ítem descriptor_index* debe ser un índice válido dentro de la tabla *constant_pool*. La entrada al índice debe contener una estructura *CONSTANT_Utf8_info* que representa un descriptor válido para la variable local. Los descriptores de variables locales tienen la misma forma que los descriptores de campos.

- index.

  La variable local dada debe estar en el *index* de las variables locales de su método. Si la variable local en el *index* es de dos palabras (*double* o *long*) ésta ocupará *index* e *index* +1.

4.5.8 El atributo Synthetic.

El atributo *Synthetic* es un atributo de longitud fija dentro de la tabla de atributos de las estructuras *classFile*, *field_info* y *method_info*. Un miembro de clase que no aparece en el código fuente debe tener asignado un atributo *Synthetic*. El atributo *Synthetic* tiene la siguiente estructura.

```
Synthetic_attribute
{
     u2     attribute_name_index;
     u4     attribute_length;
}
```

4.5.8.1 attribute_name_index.

El valor del *ítem attribute_name_index* debe ser un índice válido dentro de la tabla *constant_pool*. El índice de entrada en el *constant_pool* es una estructura *CONSTANT_Utf8_info* que representa a la cadena “Synthetic”.

4.5.8.2 attribute_length.

El valor del *ítem attribute_length* es cero.

4.5.9 El atributo Deprecated.

El atributo *Deprecated* es un atributo opcional de longitud fija dentro de la tabla de atributos de las estructuras *classFile*, *field_info* y *method_info*. Una clase, interfaz, método o campo puede estar utilizando el atributo *Deprecated* que indica que la clase, interfaz, método o campo ha sido reemplazado. Un intérprete o alguna herramienta que lee el formato del archivo *.class*, como un compilador, puede utilizar este atributo para avisar al usuario que la clase, interfaz, método o campo será reemplazada. La presencia del atributo *Deprecated* no altera la semántica de la clase o interfaz.

El atributo *Deprecated* tiene el siguiente formato:

```
Deprecated_attribute
{
    u2 attribute_name_index;
    u4 attribute_length;
}
```

4.5.9.1 attribute_name_index.

El valor del *ítem attribute_name_index* debe ser un índice válido dentro de la tabla *constant_pool*. El índice de entrada al *constant_pool* es una estructura *CONSTANT_Utf8_info* que representa a la cadena “Deprecated”.

4.5.9.2 attribute_length .

El valor del *item attribute_length* es cero.

4.5.10 El atributo InnerClasses.

El atributo *InnerClasses* es un atributo de longitud variable dentro de la tabla de la estructura *classFile*. Si el *constant_pool* de una clase o interfaz se refiere a alguna clase o interfaz que no es miembro del paquete, la estructura *classFile* deberá tener exactamente un atributo *InnerClasses* en su tabla de atributos. El atributo *InnerClasses* tiene el siguiente formato:

```
InnerClasses_attribute
{
     u2 attribute_name_index;
     u4 attribute_length;
     u2 number_of_classes;
     {
            u2 inner_class_info_index;
            u2 outer_class_info_index;
            u2 inner_name_index;
            u2 inner_class_access_flags;
     }      classes[ number_of_classes ];
}
```

4.5.10.1 attribute_name_index.

El valor del *ítem attribute_name_index* debe ser un índice válido dentro de la tabla *constant_pool*. El índice de entrada al *constant_pool* es una estructura *CONSTANT_Utf8_info* que representa a la cadena "InnerClasses".

4.5.10.2 attribute_length.

El valor del *ítem attribute_length* representa la longitud del atributo, excluyendo los 6 bytes iniciales.

4.5.10.3 number_of_classes

El valor del *ítem number_of_classes* indica el número de entradas en el arreglo *classes[]*.

4.5.10.4 classes[].

Cada entrada *CONSTANT_Class_info* en la tabla *constant_pool* que representa una clase o interfaz que no pertenece al paquete deberá tener una entrada en el arreglo *classes[]*.

Si una clase tiene miembros que son clases o interfaces, su *constant_pool* (y por lo tanto tiene el atributo *InnerClasses*) deberá referirse a cada miembro, incluso si esos miembros no son mencionados por la clase. Estas reglas implican que una clase anidada o interfaz miembro tendrán información de su atributo *InnerClasses*. Cada entrada en el arreglo *classes[]* contiene los siguientes cuatro ítems:

- inner_class_info_index.

  El valor del ítem *inner_class_info_index* debe ser cero o un índice válido en la tabla *constant_pool*. El índice de entrada en el *constant_pool* es una estructura *CONSTANT_Class_info* que representa una clase, los *ítems* restantes en el arreglo son información acerca de la clase.

- outer_class_info_index.

  Si una clase no es un miembro, el valor del *ítem outer_class_info_index* debe ser cero. En otro caso el valor del *ítem* deberá ser un índice válido dentro de la tabla *constant_pool* y el índice de entrada deberá ser una estructura *CONSTANT_Class_info* que representa una clase o interfaz la cual es miembro.

- inner_name_index.

  Si la clase es anónima, el valor del ítem *inner_name_index* debe ser cero. En otro caso el valor del ítem *inner_name_index* debe ser un índice válido dentro de la tabla *constant_pool*, y la entrada en ese índice deberá ser una estructura *CONSTANT_Utf8_info* que representa el nombre original de la clase *inner*.

- inner_class_access_flags.

  El valor del ítem *inner_class_access_flags* es una máscara usada para denotar los permisos de acceso y las propiedades de la clase o interfaz *inner*, los valores de la bandera se muestran en la tabla 4-6.

Nombre de la bandera	Valor	Significado
ACC_PUBLIC	0x0001	Marcado o implícitamente es *public*
ACC_PRIVATE	0x0002	Marcado como private.
ACC_PROTECTED	0x0004	Marcado como protected
ACC_STATIC	0x0008	Marcado o implícitamente es static
ACC_FINAL	0x0010	Marcado como final.
ACC_INTERFACE	0x0200	Es una interfaz.
ACC_ABSTRACT	0x0400	Marcado o implícitamente es abstract

Tabla 4-6 Permisos de acceso a las clases anidadas.

Todos los bits del *ítem inner_class_access_flags* no asignados en la tabla 4-6, son reservados para el futuro, deberán estar en cero en los archivos *.class* generados y serán ignorados por la implementación de la JVM.

4.6 Nombres calificados.

Los nombres de clases e interfaces que aparecen en las estructuras del archivo *.class*, son representadas en forma de nombres calificados. Estos nombres son representados como una estructura *CONSTANT_Utf8_info*.

Por razones históricas la sintaxis de los nombres de clases e interfaces que aparecen en las estructuras del archivo *.class* difieren de la sintaxis que se utiliza en el lenguaje de programación Java. En su forma interna, el carácter **(“.”)** que normalmente separa los identificadores para una variable o un método es remplazado por la diagonal **(“/”)**. Por ejemplo, el nombre calificado de la

clase `Thread` es `java.lang.Thread`. En la forma interna, una referencia al nombre de la clase `Thread` es implementada utilizando la estructura `CONSTANT_Utf8_info` y representa la cadena "**java/lang/Thread**".

4.6.1 Descriptores.

Un descriptor es una cadena que representa el tipo de un método o un campo. Los descriptores están representados dentro del formato del archivo *.class* utilizando codificación UTF-8.

4.6.1.1 Descriptores de campos.

Un descriptor de campo representa un tipo de clase, instancia o variable local. A continuación se muestra la tabla 4-7 que contiene los descriptores de campos.

Descriptor	Tipo de dato	Interpretación
B	byte	byte con signo
C	char	carácter unicode
D	double	punto flotante de doble precisión
F	float	punto flotante de precisión simple
I	int	entero
J	long	entero largo
L<classname>;	referencia	instancia de una clase <classname>
S	short	short con signo
Z	boolean	valor booleano
[	referencia	arreglo de una dimensión

Tabla 4-7 Descriptores de campos.

<classname> representa el nombre completamente calificado de una clase o interfaz. Por ejemplo, el descriptor de una variable de instancia de tipo `int` es `I`. El descriptor de una variable de instancia de tipo `Object` es `Ljava/lang/Object;`. El descriptor de una variable de instancia que es un arreglo multidimensional de tipo double `double d[][][]` es `[[[D`.

4.6.1.2 Descriptores de métodos.

Un descriptor de método representa los parámetros que el método toma y devuelve, se representa bajo la siguiente gramática:

MethodDescriptor : *(ParameterDescriptor*)ReturnDescriptor*

donde:

ParameterDescriptor representa un parámetro pasado al método, el símbolo * significa que pueden existir cero o más posibles parámetros pasados al método.

ReturnDescriptor representa el tipo de valor devuelto por el método, si el método no regresa ningún valor, *ReturnDescriptor* tendrá entonces el valor `V`, que representa la palabra clave `void`. Un descriptor de método es válido si y sólo si representa los parámetros del método con una longitud total de 255 o menor, donde la longitud se calcula sumando el número de parámetros individuales, siendo que un parámetro de tipo `long` o `double` contribuye con dos unidades y cualquier otro parámetro contribuye con una unidad.

Por ejemplo, el descriptor de método para el método:

```
Object miMetodo(int i, double d, Vector v)
```

Es

```
(IDLjava/util/Vector;)Ljava/lang/Object;
```

Note que la forma de nombres completamente calificados son utilizados para describir los nombres de las clases `Object` y `Vector`.

CAPÍTULO

5

Capítulo 5 El desensamblador Marago.

En capítulos previos se habló de la tecnología Java, el proceso de compilación y el papel que tiene la Máquina Virtual Java. Además, se describió a detalle el formato del archivo *.class;* con esta información se tienen los conocimientos necesarios para generar código Java a partir de código de *bytecode*. En este capítulo se explica el funcionamiento del desensamblador **Marago** y la forma en que genera código Java a partir de la especificación del archivo *.class*.

5.1 Desensamblador mejorado.

Sun Microsystems distribuye el paquete de desarrollo JDK (*Java Development Kit*) que contiene algunas herramientas para crear y construir aplicaciones:

- **javac** es el compilador para el lenguaje de programación Java.
- **java** es el intérprete de aplicaciones.
- **javadoc** es el generador de documentación.
- **appletviewer** ejecuta y depura applets sin un navegador.
- **jar** es un manejador de archivos *.jar*.
- **jdb** es el depurador Java.
- **javah** se utiliza para escribir métodos nativos y genera librerías del lenguaje C.
- **extcheck** es una utilidad para detectar conflictos de archivos *.jar*.
- **javap** que es un desensamblador de archivos *.class*.

Nos enfocamos en la herramienta **javap** que desensambla archivos *.class;* el resultado depende de las opciones utilizadas. Aunque puede resultar muy útil, es una herramienta limitada, ya que no genera código que pueda compilarse o interpretarse, además la información que presenta es escasa y confusa.

Marago es una herramienta que a partir de la especificación del formato del archivo *.class* desensambla código Java y lo presenta al usuario de una manera sencilla y fácil de leer, además el código puede ser interpretado por Jazmín[1].

5.1.1 ¿Por qué Marago?.

Cuando **James Gosling** bautizó su lenguaje, lo hizo pensando en un nombre que fuera sinónimo de energía, -en Estados Unidos el café también recibe el nombre de java-. Ya que la cafeína es una sustancia altamente energética, se le bautizó al nuevo lenguaje con el nombre de Java.

Todo lo relacionado con el lenguaje de programación Java tiene tradicionalmente un nombre relativo al café. El primer intento de un decompilador hecho por **Hanpeter Van Vliet** se llamó **Mocha** haciendo alusión al café sabor moka, el primer ofuscador de código Java –también programado por Van Vliet- se llamó **Crema**.

El nombre del desensamblador que se desarrolla en esta tesis recibe el nombre de **Marago** que es un tipo de café que se cultiva en la zona de los Altos de Chiapas, México.

5.1.2 Ventajas de Marago con respecto a javap.

Cuando **Sun Microsystems** desarrolló su paquete SDK (*Standard Development Kit*), programó **javap** con la idea de tener una herramienta que ayude a crear y construir aplicaciones a los desarrolladores y permita ver la firma de las clases, métodos y variables, es decir, saber que métodos posee una clase, tipo de argumentos que reciben, tipo de dato que devuelven y los permisos de acceso a las variables.

Marago es una herramienta desarrollada con Java y no sólo está pensada para desensamblar código Java, sino que sea una base para la construcción de un decompilador. Algunas de las ventajas de Marago con respecto a javap son:

- *Presenta información del constant_pool*. El *constant_pool* es la tabla de símbolos que utiliza la JVM, **javap** no presenta información del *constant_pool* a pesar de que muchas instrucciones (*opcodes*) hacen referencia a él.

- *Presenta código en formato similar a un archivo .java.* **javap** despliega la información de las clases y los métodos en su formato de nombres completamente calificados, lo cual hace difícil la comprensión del programa.

[1] Yasmín es un ensamblador Java de código abierto, el desensamblador Marago genera código que Jasmin puede ensamblar. Para detalles acerca de Yasmín consulte "Java Virtual Machine", Meyer Jon y Downing Troy, ed. O´reilly.

- *Es la base para la construcción de un decompilador*. **javap** sólo es una herramienta que permite desensamblar código. **Marago** está pensado para trabajar como un decompilador.

- *Utiliza interfaz gráfica*. j**avap** sólo funciona en línea de comandos, **Marago** posee una interfaz gráfica que permite desensamblar código de manera sencilla.

- *Portabilidad.* **javap** no está programado en Java, por lo que es diferente para cada arquitectura, **Marago** al ser desarrollado enteramente en Java es independiente de la arquitectura en que se ejecuta.

En el ejemplo siguiente se muestra un programa que suma dos números y debajo de él se compara el código obtenido por los dos desensambladores.

```
class SumaDosNumeros
{
   public static void main(String[] args)
   {
        int a = 3;
        int b = 4;
        int result = a + b;
        System.out.print( "La suma de a y b es: " +  result );
   }
}
```

El programa `SumaDosNumeros` realiza la adición de dos números de tipo `int` y el resultado lo presenta en pantalla. Primero desensamblaremos el programa con **javap**, sin utilizar ninguna opción y obtenemos lo siguiente:

```
C:\Tesis\Marago Improved>javap SumaDosNumeros
Compiled from SumaDosNumeros.java
class SumaDosNumeros extends java.lang.Object {
    SumaDosNumeros();
    public static void main(java.lang.String[]);
}
```

Ahora utilizaremos la opción `-c` que permite ver el código completo:

```
C:\Tesis\Marago Improved>javap -c SumaDosNumeros
Compiled from SumaDosNumeros.java
class SumaDosNumeros extends java.lang.Object {
    SumaDosNumeros();
    public static void main(java.lang.String[]);
}
```

```
Method SumaDosNumeros()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

Method void main(java.lang.String[])
   0 iconst_3
   1 istore_1
   2 iconst_4
   3 istore_2
   4 iload_1
   5 iload_2
   6 iadd
   7 istore_3
   8 getstatic #2 <Field java.io.PrintStream out>
  11 new #3 <Class java.lang.StringBuffer>
  14 dup
  15 invokespecial #4 <Method java.lang.StringBuffer()>
  18 ldc #5 <String "La suma de a y b es: ">
  20 invokevirtual #6 <Method java.lang.StringBuffer append(java.lang.String)>
  23 iload_3
  24 invokevirtual #7 <Method java.lang.StringBuffer append(int)>
  27 invokevirtual #8 <Method java.lang.String toString()>
  30 invokevirtual #9 <Method void print(java.lang.String)>
  33 return
```

Marago funciona con una interfaz gráfica o desde la línea de comandos, para poder ejecutarlo se debe utilizar la siguiente instrucción:

```
$/root/pruebas/java –cp /foo/Marago.jar Marago /foo/archivo.class [-fp]
```

La opción `–cp` agrega al **CLASSPATH** el archivo Tesis.jar que contiene todas las clases necesarias para desensamblar, si se agrega a la variable de ambiente la dirección del archivo Marago.jar, la opción `–cp` no es necesaria.

Marago utiliza dos banderas `–f` sobrescribe un archivo .café desensamblado, `-p` crea el *constant_pool* dentro del archivo *.cafe*.

```
/**********************************/
//  National University of Mexico
//     School of Engineering
//
//        Disassembler by
//           Marago
//           Thesis
//         Uriel Nava
/**********************************/

/******  Inicio del Constant Pool   ******/
1 CONSTANT_Methodref : 11,20
2 CONSTANT_Fieldref : 21,22
3 CONSTANT_Class : 23
4 CONSTANT_Methodref : 3,20
5 CONSTANT_String : 24
6 CONSTANT_Methodref : 3,25
7 CONSTANT_Methodref : 3,26
8 CONSTANT_Methodref : 3,27
9 CONSTANT_Methodref : 28,29
10 CONSTANT_Class : 30
11 CONSTANT_Class : 31
12 CONSTANT_Utf8 : <init>
13 CONSTANT_Utf8 : ()V
14 CONSTANT_Utf8 : Code
15 CONSTANT_Utf8 : LineNumberTable
16 CONSTANT_Utf8 : main
17 CONSTANT_Utf8 : ([Ljava/lang/String;)V
18 CONSTANT_Utf8 : SourceFile
19 CONSTANT_Utf8 : SumaDosNumeros.java
20 CONSTANT_NameAndType : 12,13
21 CONSTANT_Class : 32
22 CONSTANT_NameAndType : 33,34
23 CONSTANT_Utf8 : java/lang/StringBuffer
24 CONSTANT_Utf8 : La suma de a y b es:
25 CONSTANT_NameAndType : 35,36
26 CONSTANT_NameAndType : 35,37
27 CONSTANT_NameAndType : 38,39
28 CONSTANT_Class : 40
29 CONSTANT_NameAndType : 41,42
30 CONSTANT_Utf8 : SumaDosNumeros
31 CONSTANT_Utf8 : java/lang/Object
32 CONSTANT_Utf8 : java/lang/System
33 CONSTANT_Utf8 : out
34 CONSTANT_Utf8 : Ljava/io/PrintStream;
35 CONSTANT_Utf8 : append
36 CONSTANT_Utf8 : (Ljava/lang/String;)Ljava/lang/StringBuffer;
37 CONSTANT_Utf8 : (I)Ljava/lang/StringBuffer;
38 CONSTANT_Utf8 : toString
39 CONSTANT_Utf8 : ()Ljava/lang/String;
40 CONSTANT_Utf8 : java/io/PrintStream
41 CONSTANT_Utf8 : print
42 CONSTANT_Utf8 : (Ljava/lang/String;)V
/***** Fin del Constant Pool ******/

import java.io.PrintStream;

class SumaDosNumeros
{
 SumaDosNumeros()
 {
  0 aload_0
```

```
 1 invokespecial #1 <Method: void Object()> <nameClass: Object>
 4 return
}

public static void main (String[] string0)
{
 0 iconst_3
 1 istore_1
 2 iconst_4
 3 istore_2
 4 iload_1
 5 iload_2
 6 iadd
 7 istore_3
 8 getstatic #2 <Field PrintStream out>
 11 new #3 <Class StringBuffer>
 14 dup
 15 invokespecial #4 <Method: void StringBuffer()> <nameClass: StringBuffer>
 18 ldc #5 <String "La suma de a y b es: ">
 20 invokevirtual #6 <Method: StringBuffer append(String string0)> <nameClass:
StringBuffer>
 23 iload_3
 24 invokevirtual #7 <Method: StringBuffer append(int int0)> <nameClass:
StringBuffer>
 27 invokevirtual #8 <Method: String toString()> <nameClass: StringBuffer>
 30 invokevirtual #9 <Method: void print(String string0)> <nameClass:
PrintStream>
 33 return
}
}
```

Más adelante se explicará a detalle el código generado, por ahora sólo se muestra el código con el propósito de comparar **Marago** con **javap**.

5.2 Construcción del desensamblador.

Para obtener código fuente Java existen varias formas de lograrlo, una de ellas se llama reflexión (*reflection*) que es la capacidad de conocer detalles de otras clases, incluidas las nuestras.

Los paquetes `java.lang.class` y `java.lang.reflect`[2] ofrecen clases e interfaces para obtener información relativa a las clases y los objetos, permite acceso a la información de campos, métodos y constructores de clases.

Otra forma de obtener código Java es a través de la especificación del archivo *.class*. **Marago** utiliza este método, ya que permite mayor control y flexibilidad –*reflection* no permite conocer que hay dentro de cada método-.

[2] Para más detalles del paquete java.lang.reflect puede consultar la documentación del API en www.java.sun.com.

En las siguientes páginas se explicará como construir un desensamblador a partir de la especificación del archivo *.class*, el cual podemos dividirlo en los siguientes elementos constitutivos:

- *Magic number*
- *Minor and Major version numbers*
- *ConstantPool Count*
- *ConstantPool*
- *Access Flags*
- *This class*
- *Super class*
- *Interfaces Count*
- *Interfaces*
- *Methods Count*
- *Methods*
- *Attributes Count*
- *Attributes*

Para hacer más sencillo el análisis, se creó una estructura *classFile*[3]*;* pero antes de comenzar a llenar esa estructura, **Marago** realiza algunas comprobaciones[4] sobre el archivo *.class;* en la figura 5-1 se presenta un algoritmo general de la forma en que trabaja el desensamblador.

[3] La estructura classFile se describe en el capítulo 4 "El formato del archivo .class".
[4] Antes de comenzar a desensamblar el archivo .class, Marago revisa si existe la referencia al archivo, si puede sobrescribirlo y si la sintaxis de llamada es correcta.

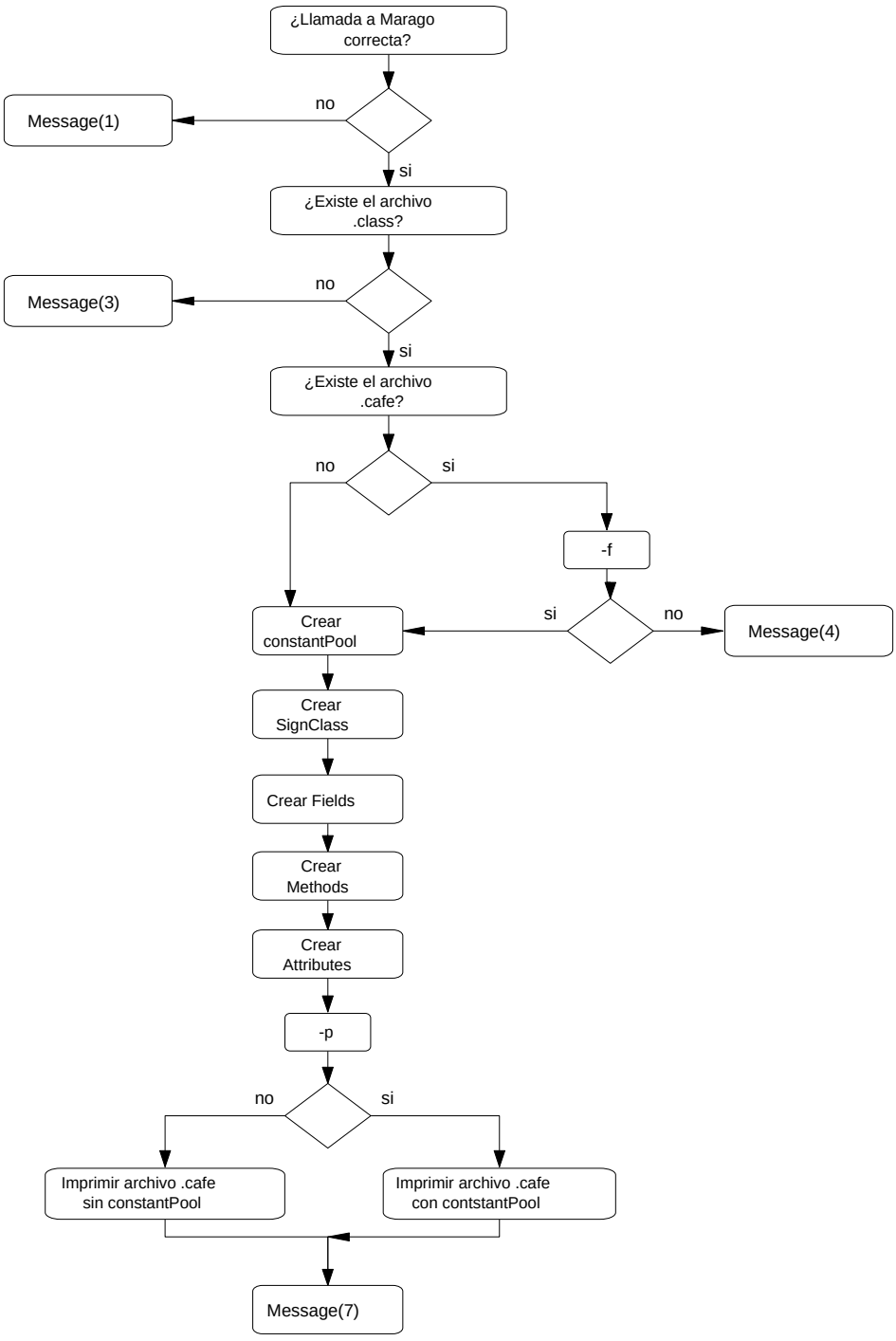

Figura 5-1 El diagrama de flujo presenta la forma en que trabaja **Marago**, antes de crear la estructura *classFile* realiza algunas comprobaciones sobre el archivo *.class*

Una vez realizadas las comprobaciones, **Marago** crea la estructura *classFile*. Encontrar los *ítems magic*, *minor_version* y *major_version* es sencillo, se localizan al comienzo del archivo *.class* –en la figura 5-2 puede encontrarlos fácilmente-.

El *ítem magic* en hexadecimal es construido por los primeros cuatro bytes, 0xCAFEBABE e indican a la JVM que está recibiendo un archivo *.class*. *minor_version* y *major_version* son los siguientes cuatro bytes 0x0003 y 0x002D respectivamente.

La figura 5-2 es el resultado de compilar la clase `HolaMundo`, este pequeño programa muestra en pantalla un saludo, el código se muestra a continuación.

```
public class HolaMundo
{
    final static String Saludo = "Hola Mundo";

    public static void main(String[] args)
    {
        System.out.print( Saludo );
    }
}
```

Para ver el contenido del archivo `HolaMundo.class` puede utilizar cualquier editor hexacedimal, para esta tesis se utilizó el editor **HexEditor** que es distribuido de manera gratuita.

```
ca fe ba be 00 03 00 2d 00 20 0a 00 06 00 12 09 00 13 00 14 08 00 15 0a 00 16 00
17 07 00 18 07 00 19 01 00 06 53 61 6c 75 64 6f 01 00 12 4c 6a 61 76 61 2f 6c 61
6e 67 2f 53 74 72 69 6e 67 3b 01 00 0d 43 6f 6e 73 74 61 6e 74 56 61 6c 75 65 01
00 06 3c 69 6e 69 74 3e 01 00 03 28 29 56 01 00 04 43 6f 64 65 01 00 0f 4c 69 6e
65 4e 75 6d 62 65 72 54 61 62 6c 65 01 00 04 6d 61 69 6e 01 00 16 28 5b 4c 6a 61
76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 29 56 01 00 0a 53 6f 75 72 63 65 46
69 6c 65 01 00 0e 48 6f 6c 61 4d 75 6e 64 6f 2e 6a 61 76 61 0c 00 0a 00 0b 07 00
1a 0c 00 1b 00 1c 01 00 0a 48 6f 6c 61 20 4d 75 6e 64 6f 07 00 1d 0c 00 1e 00 1f
01 00 09 48 6f 6c 61 4d 75 6e 64 6f 01 00 10 6a 61 76 61 2f 6c 61 6e 67 2f 4f 62
6a 65 63 74 01 00 10 6a 61 76 61 2f 6c 61 6e 67 2f 53 79 73 74 65 6d 01 00 03 6f
75 74 01 00 15 4c 6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d 3b 01
00 13 6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d 01 00 05 70 72 69
6e 74 01 00 15 28 4c 6a 61 76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 29 56 00
21 00 05 00 06 00 00 00 01 00 18 00 07 00 08 00 01 00 09 00 00 00 02 00 03 00 02
00 01 00 0a 00 0b 00 01 00 0c 00 00 00 1d 00 01 00 01 00 00 00 05 2a b7 00 01 b1
00 00 00 01 00 0d 00 00 00 06 00 01 00 00 00 01 00 09 00 0e 00 0f 00 01 00 0c 00
00 00 25 00 02 00 01 00 00 00 09 b2 00 02 12 03 b6 00 04 b1 00 00 00 01 00 0d 00
00 00 0a 00 02 00 00 00 07 00 08 00 08 00 01 00 10 00 00 00 02 00 11
```

Figura 5-2 Archivo hexadecimal de la clase HolaMundo.

5.2.1 Reconstruyendo el constant_pool.

Después de realizar algunas verificaciones y leer los *ítems magic*, *minor_version* y *major_version*, **Marago** debe crear el *constant_pool*. Todas las clases o interfaces poseen un *constant_pool*, que hace el papel de una tabla de símbolos.

De acuerdo con la estructura *classFile*, encontramos dos bytes que representan al *constant_pool_count*, que es el hexadecimal 0x0020 o el entero 32 –esto indica que el *constant_pool* para la clase `HolaMundo` tiene 31 índices-.

Los siguientes *ítems* en la estructura *classFile* construyen el *constant_pool*, que es un arreglo de longitud variable; muchos de los elementos son referencias simbólicas a otros miembros del *constant_pool*. Por ejemplo, una estructura *CONSTANT_String_info* apunta a una estructura *CONSTANT_Utf8_info* –esto se explico a detalle en el capítulo anterior-. La figura 5-4 es el código para el `PoolParser` , el cual construye un *constant_pool*.


```
ca fe ba be 00 03 00 2d 00 20 0a 00 06 00 12 09 00 13 00 14 08 00 15 0a 00 16 00
17 07 00 18 07 00 19 01 00 06 53 61 6c 75 64 6f 01 00 12 4c 6a 61 76 61 2f 6c 61
6e 67 2f 53 74 72 69 6e 67 3b 01 00 0d 43 6f 6e 73 74 61 6e 74 56 61 6c 75 65 01
00 06 3c 69 6e 69 74 3e 01 00 03 28 29 56 01 00 04 43 6f 64 65 01 00 0f 4c 69 6e
65 4e 75 6d 62 65 72 54 61 62 6c 65 01 00 04 6d 61 69 6e 01 00 16 28 5b 4c 6a 61
76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 29 56 01 00 0a 53 6f 75 72 63 65 46
69 6c 65 01 00 0e 48 6f 6c 61 4d 75 6e 64 6f 2e 6a 61 76 61 0c 00 0a 00 0b 07 00
1a 0c 00 1b 00 1c 01 00 0a 48 6f 6c 61 20 4d 75 6e 64 6f 07 00 1d 0c 00 1e 00 1f
01 00 09 48 6f 6c 61 4d 75 6e 64 6f 01 00 10 6a 61 76 61 2f 6c 61 6e 67 2f 4f 62
6a 65 63 74 01 00 10 6a 61 76 61 2f 6c 61 6e 67 2f 53 79 73 74 65 6d 01 00 03 6f
75 74 01 00 15 4c 6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d 3b 01
00 13 6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d 01 00 05 70 72 69
6e 74 01 00 15 28 4c 6a 61 76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 29 56 00
21 00 05 00 06 00 00 00 01 00 18 00 07 00 08 00 01 00 09 00 00 00 02 00 03 00 02
00 01 00 0a 00 0b 00 01 00 0c 00 00 00 1d 00 01 00 01 00 00 00 05 2a b7 00 01 b1
00 00 00 01 00 0d 00 00 00 06 00 01 00 00 00 01 00 09 00 0e 00 0f 00 01 00 0c 00
00 00 25 00 02 00 01 00 00 00 09 b2 00 02 12 03 b6 00 04 b1 00 00 00 01 00 0d 00
00 00 0a 00 02 00 00 00 07 00 08 00 08 00 01 00 10 00 00 00 02 00 11
```

Figura 5-3 *constant _pool* en el archivo hexadecimal.

```
import java.io.*;
import java.util.*;

public class PoolParser
{
 public static void main(String[] args)
 {
  try
  {
    File  fileInput = new File (args[0]);
    RandomAccessFile rafPool = new RandomAccessFile (fileInput, "r");
    PoolParser cP = new PoolParser (rafPool);
  }
  catch ( IOException e ) { e.printStackTrace(); }
  catch ( ArrayIndexOutOfBoundsException f  ) { f.printStackTrace(); }
 }

 public PoolParser(RandomAccessFile rafPool) throws IOException
 {
   String[][] constantPool = null;
   int constantPoolCount = 0;
   int jumpPool = 8;
   int tag, strSize;
   char car;

   rafPool.seek(jumpPool);
   constantPoolCount = rafPool.readShort();
   constantPool = new String[constantPoolCount][3];
   StringBuffer sbString = new StringBuffer();

   for(int i = 1; i < constantPoolCount ; i++)
   {
     tag =rafPool.readByte();
     switch(tag)
     {
       case 1:  constantPool[i][0] = "CONSTANT_Utf8";
                sbString.setLength(0);
                strSize = rafPool.readShort();
                while (strSize > 0)
                {
                  car = (char)(rafPool.readByte());
                  sbString.append(car);
                  strSize--;
                }
                constantPool[i][1] = sbString.toString();
                break;

       case 3:  constantPool[i][0] = "CONSTANT_Integer";
                constantPool[i][1] = Integer.toString(rafPool.readInt());
                beak;

       case 4:  constantPool[i][0] = "CONSTANT_Float";
                constantPool[i][1] =  Float.toString(rafPool.readFloat());
                break;

       case 5:  constantPool[i][0] = "CONSTANT_Long";
                constantPool[i][1] = Long.toString(rafPool.readLong());
                i++;
                break;

       case 6:  constantPool[i][0] = "CONSTANT_Double";
                constantPool[i][1] = Double.toString(rafPool.readDouble());
```

```
                i++;
                break;

        case 7:  constantPool[i][0] = "CONSTANT_Class";
                 constantPool[i][1] = Integer.toString(rafPool.readShort());
                 break;

        case 8:  constantPool[i][0] = "CONSTANT_String";
                 constantPool[i][1] = Integer.toString(rafPool.readShort());
                 break;

        case 9:  constantPool[i][0] = "CONSTANT_Fieldref";
                 constantPool[i][1] = Integer.toString(rafPool.readShort());
                 constantPool[i][2] = Integer.toString(rafPool.readShort());
                 break;

        case 10:  constantPool[i][0] = "CONSTANT_Methodref";
                  constantPool[i][1] = Integer.toString(rafPool.readShort());
                  constantPool[i][2] = Integer.toString(rafPool.readShort());
                  break;

        case 11:  constantPool[i][0] = "CONSTANT_InterfaceMethodRef";
                  constantPool[i][1] = Integer.toString(rafPool.readShort());
                  constantPool[i][2] = Integer.toString(rafPool.readShort());
                  break;

        case 12:  constantPool[i][0] = "CONSTANT_NameAndType";
                  constantPool[i][1] = Integer.toString(rafPool.readShort());
                  constantPool[i][2] = Integer.toString(rafPool.readShort());
                  break;

        default:  System.out.println ("Error de Parsing");
                  break;
               }
      }

      System.out.print("El Constant Pool tiene:"+constantPoolCount+"entradas");
      System.out.print( "\n"+ "/******   Constant Pool   ******/" + "\n");
      for( int i = 1; i < constantPoolCount; i++)
      {
        System.out.print(i + " ");
        for (int j = 0; j < constantPool[i].length; j++)
        {
          if( constantPool[i][j] != null )
          {
            if (j == 0)  System.out.print(constantPool[i][j] + ":" );
            else
            {
              System.out.print(constantPool[i][j]);
              if (j<(constantPool[i].length -1) && constantPool[i][j+1]!= null)
                 System.out.print(",");
            }
          }
        }
        System.out.print( "\n" );
      }
      System.out.print("/***** Constant Pool ******/" + "\n\n");
    }
}
```

Figura 5-4 Parser para crear el *constant_pool*

La clase `PoolParser` crea el *constant_pool*, no toma en cuenta *magic*, *minor_version* y *major_versión*, ya que para nuestro propósito (desensamblar código), no tienen importancia.

Marago lee el *ítem constant_pool_count* que le indica el número de entradas. Recuerde que el *constant_pool* es de tamaño variable, donde la entrada 0 se cuenta, pero no se utiliza. Los índices del *constant_pool* son muy importantes, ya que la JVM hace referencia al *constant_pool* a través de ellos.

```
/******  Inicio del Constant Pool   ******/
1 CONSTANT_Methodref : 6,18
2 CONSTANT_Fieldref : 19,20
3 CONSTANT_String : 21
4 CONSTANT_Methodref : 22,23
5 CONSTANT_Class : 24
6 CONSTANT_Class : 25
7 CONSTANT_Utf8 : Saludo
8 CONSTANT_Utf8 : Ljava/lang/String;
9 CONSTANT_Utf8 : ConstantValue
10 CONSTANT_Utf8 : <init>
11 CONSTANT_Utf8 : ()V
12 CONSTANT_Utf8 : Code
13 CONSTANT_Utf8 : LineNumberTable
14 CONSTANT_Utf8 : main
15 CONSTANT_Utf8 : ([Ljava/lang/String;)V
16 CONSTANT_Utf8 : SourceFile
17 CONSTANT_Utf8 : HolaMundo.java
18 CONSTANT_NameAndType : 10,11
19 CONSTANT_Class : 26
20 CONSTANT_NameAndType : 27,28
21 CONSTANT_Utf8 : Hola Mundo
22 CONSTANT_Class : 29
23 CONSTANT_NameAndType : 30,31
24 CONSTANT_Utf8 : HolaMundo
25 CONSTANT_Utf8 : java/lang/Object
26 CONSTANT_Utf8 : java/lang/System
27 CONSTANT_Utf8 : out
28 CONSTANT_Utf8 : Ljava/io/PrintStream;
29 CONSTANT_Utf8 : java/io/PrintStream
30 CONSTANT_Utf8 : print
31 CONSTANT_Utf8 : (Ljava/lang/String;)V
/***** Fin del Constant Pool ******/
```

Figura 5-5 *constant_pool* de la clase HolaMundo.

5.2.2 Reconstruyendo la firma de la clase.

Siguiendo con la estructura *classFile*, toca ahora leer los *ítems access_flags*, *this_class*, *super_class*, *interfaces_count* e *interfaces[]*.

La clase `SignClass` se encarga de construir la firma de la clase, así como su paquete – si existe- e importa las librerías que necesita la clase para que pueda ser interpretada.

Los dos bytes que siguen al *constant_pool* forman el *ítem access_flags* que es el hexadecimal 0x0021, de la tabla 4-1 se sabe que el acceso a la clase es `public`. El *ítem this_class* 0x0005 hace referencia al índice 5 del *constant_pool.*

El índice `5` en el *constant_pool* es una estructura *CONSTANT_Class*, la estructura hace referencia al índice `24` que es una estructura *CONSTANT_Utf8*, la cual contiene el nombre de la clase actual "HolaMundo". Si la clase perteneciera a algún paquete (*package*), el *ítem this_class* contendría el paquete y el nombre de la clase.

```
5 CONSTANT_Class : 24
24 CONSTANT_Utf8 : HolaMundo
```

El *ítem super_class* son los siguientes dos bytes 0x0006 y hace referencia al índice 6 del *constant_pool*.

El índice `6` del *constant_pool* es una estructura *CONSTANT_Class* que apunta al índice `25` del *constant_pool*, que es una estructura *CONSTANT_Utf8* la cual contiene el nombre de la superclase en el formato de nombre completamente calificado. Si la clase no tuviera una superclase *CONSTANT_Utf8* hace referencia a la clase `java.lang.Object` que es la superclase de todas las clases en Java.

```
6 CONSTANT_Class : 25
25 CONSTANT_Utf8 : java/lang/Object
```

interfaces_count 0x0000 indica cuantas interfaces son implementadas por la clase; si la clase no implementa ninguna interfaz, el valor del *ítem interfaces_count* debe ser cero. *interfaces[]* es un arreglo que contiene el nombre de las interfaces que son implementadas por la clase actual, si *interfaces_count* es cero, el arreglo *interfaces[]* no aparece en el archivo *.class.*

En nuestro ejemplo, la clase `HolaMundo` no implementa a ninguna interfaz, por lo cual el valor del *ítem interfaces_count* es cero 0x0000.

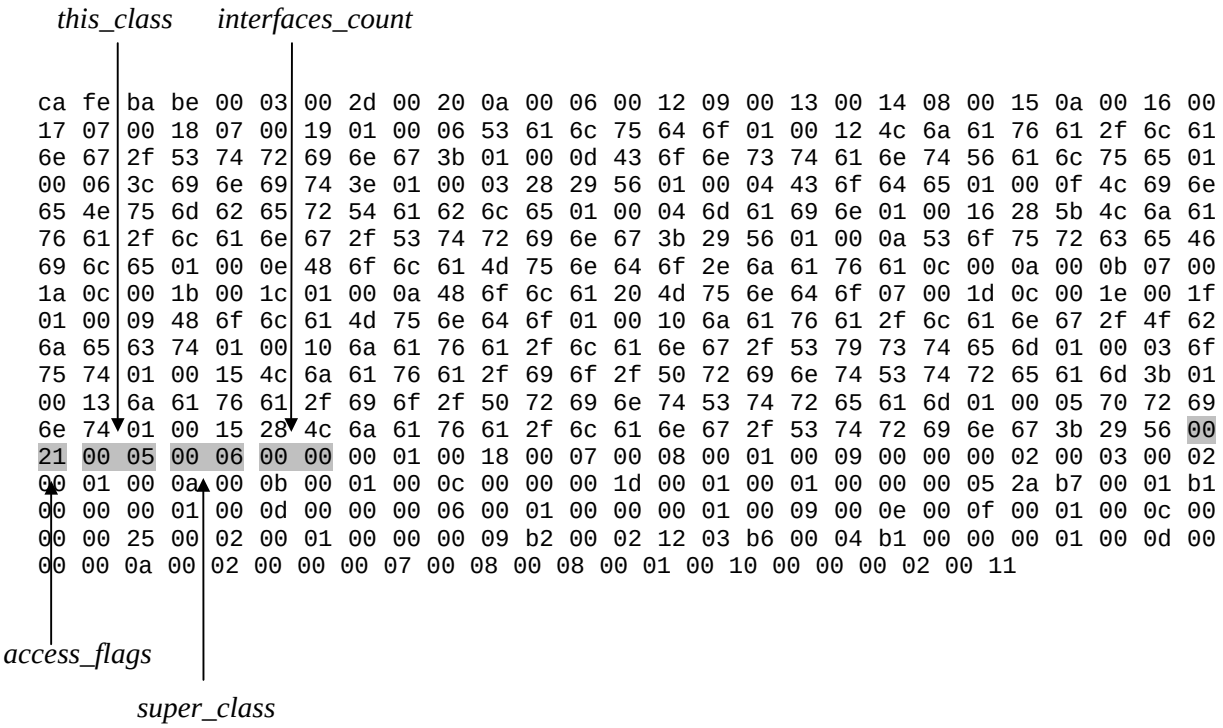

Figura 5-6 *ïtems* para la clase SignClass.

La clase `SignClass` cuenta con métodos que le ayudan a leer los *ítems access_flags*, *this_class*, *super_class*, *interfaces_count* e *interfaces[]*.

- `setFlagsClass()` Lee el *ítem access_flags* y obtiene el modificador de acceso a la clase.
- `setNameClass()` Lee el *ítem this_class* y obtiene el nombre de la clase y su paquete.
- `setSuperClass()` Lee el *ítem super_class* y obtiene la superclase.
- `setInterfacesClass()` Lee los *ítems interfaces_count* e *interfaces[]* y obtiene todas las interfaces que son implementadas por la clase.

Todos los métodos son estáticos y reciben como argumento la referencia del archivo *.class*. Para obtener el modificador de acceso a la clase se utiliza el *ítem access_flags* como clave de entrada a una estructura `hash`. Java en su paquete `java.util` tiene una clase llamada `Hashtable` que permite construir tablas `hash` de manera sencilla.

```
static Hashtable tableClass = new Hashtable();
```

`getFlags` obtiene el modificador de acceso a la clase utilizando el *ítem access_flags* como clave.

```
public static String getFlags (String key) throws IOException
{
  String strReturn = String.valueOf (tableClass.get(key));

  if (strReturn.equals("null")) Messages.message6();
  return strReturn;
}
```

Cuando un programa Java necesita una clase que no pertenece al paquete `java.lang` debe importarlo; el método `setImports` es utilizado por **Marago** para importar clases, que almacena en un `vector`.

```
public static void setImports(String strImport)
{
  String strTemp;

  ...

  if(strImport.indexOf('/')!=-
     1&&!strImport.substring(0,strImport.lastIndexOf("/")).equals(strTemp))
  {
    strImport = strImport.replace('/','.');
    if((!strImport.startsWith("java.lang")||!(strImport.lastIndexOf('.')
       ==9))&&SignClass.vecImports.indexOf("import "+ strImport.trim() +";" )==-
       1)
    SignClass.vecImports.addElement("import " + strImport.trim() + ";");
  }
}
```

Finalmente la clase `SignClass` construye la firma de la clase y la asigna a un `String strSignClass`.

```
strSignClass = strFlags + " " + strNameClass + " " + strSuperClass + " " +
               strInterfacesClass;
```

5.2.3 Reconstruyendo los Campos.

La clase `Fields` se encarga de obtener todas las variables globales de la clase, sean o no variables de instancia.

Los dos bytes siguientes en el archivo *.class*, forman el *ítem fields_count* que indica cuantas variables fueron declaradas en la clase[5]. Los bytes que le siguen al ítem *fields_count* forman estructuras *field_info* de tamaño variable que contienen información de las variables declaradas en la clase.

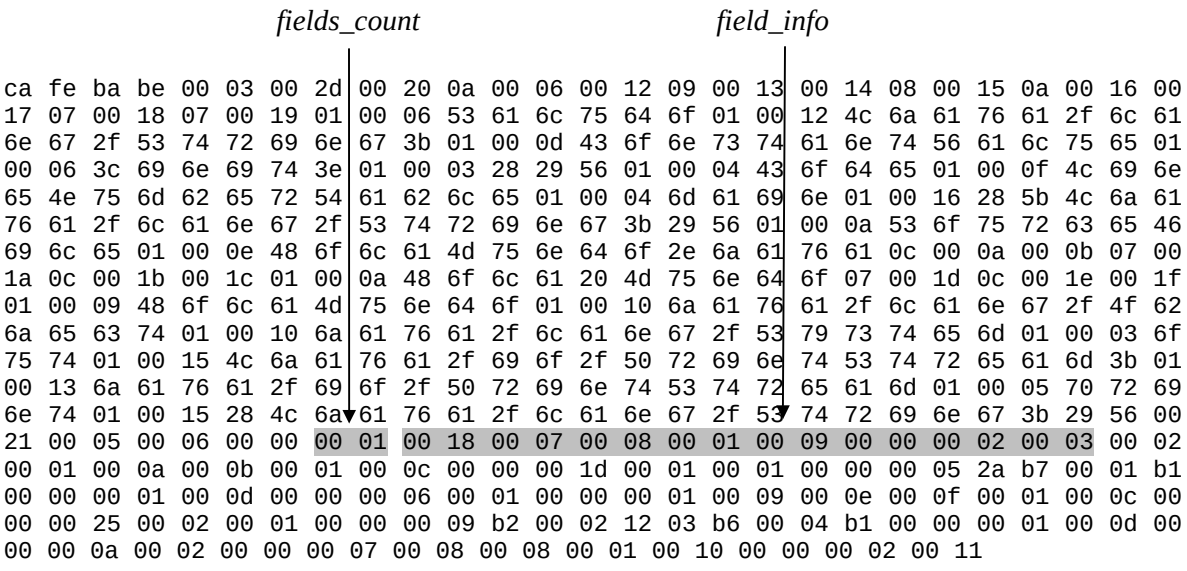

Figura 5-7 *ítems* para la clase Fields.

La clase `Fields` posee métodos que le ayudan a leer los *ítems fields_count* y *field_info* para obtener las variables globales.

- `countFields()` Lee el *item fields_count* y le permite saber cuantas variables globales y de instancia fueron declaradas.
- `setFields()` Construye la estructura *field_info* y obtiene el modificador de acceso, el tipo de dato y descriptor de cada variable declarada en la clase.
- `getFieldDescriptor()` Convierte el descriptor de cada variable en un tipo de dato Java válido.

[5] Las clases Inner declaran una variable que hace referencia a la clase Outer.

En nuestro ejemplo, la clase `HolaMundo`, el valor del *ítem fields_count* es 0x0001, ya que sólo declaramos una variable. La estructura *fields_info* se compone de los *ítems access_flags* 0x0018; con base en la tabla 4-3 sabemos que el modificador de la variable es `final static`. El *ítem name_index* 0x0007 hace referencia al índice `7` del *constant_pool*, que es una estructura *CONSTANT_Utf8* que representa el nombre de la variable.

```
7 CONSTANT_Utf8 : Saludo
```

El *ítem descriptor_index* 0x0008 hace referencia al índice 8 del *constant_pool* que es una estructura *CONSTANT_Utf8* que almacena el descriptor de la variable `Saludo`.

```
8 CONSTANT_Utf8 : Ljava/lang/String;
```

El *ítem attributes_count* 0x0001 enumera los atributos para la estructura *fields_info.* La estructura puede tener 3 atributos que son: *ConstantValue*, *Synthetic* o *Deprecate*, si apareciera algún otro atributo la JVM lo ignorará. La estructura *attributes* se menciona en el **capítulo 4**, el *ítem attribute_name_index* 0x0009 índica el índice del *constant_pool* y contiene el nombre del atributo.

```
9 CONSTANT_Utf8 : ConstantValue
```

Finalmente el *ítem constantValue_index* 0x0003 es un índice en el *constant_pool* que contiene una estructura *CONSTANT_String* que hace referencia a la estructura *CONSTANT_Utf8* la cual representa el valor de inicialización de la variable.

```
3 CONSTANT_String : 21
21 CONSTANT_Utf8 : Hola Mundo
```

Con todos estos datos sabemos que la clase `HolaMundo` declara una variable del tipo:

```
static final String Saludo = "Hola Mundo";
```

5.2.4 Reconstruyendo los Métodos.

La clase `Methods` se encarga de construir los métodos que fueron declarados en la clase, para ello debe leer los *ítems access_flags*, *name_index*, *descriptor_index*, *attributes_count* y *attributes[]*.

Siguiendo con nuestro ejemplo, el *ítem methods_count* indica cuantos métodos se declararon en la clase; en la figura 5-8 *methods_count* es 0x0002. Si revisamos el código de la clase `HolaMundo`

observamos que sólo existe el método `main`. ¿Si sólo declaramos un método, por qué aparecen en el archivo *.class* dos métodos?.

Uno de los métodos que aparece en la figura 5-8 se refiere al método `main`, el otro, se refiere al método <*init*>. Cuando compilamos un archivo Java, el compilador implícitamente crea el método constructor o método <*init*>. Un **constructor** es un procedimiento especial de una clase que es llamado automáticamente siempre que se crea un objeto de esa clase. Su función es iniciar el objeto.

Si no se declara un constructor en el archivo Java, el compilador crea uno que invoca al constructor de la clase padre.

Para el primer método, el *ítem access_flags* 0x0001 es el modificador de acceso al método, de la tabla 4-4 sabemos que es `public`. El siguiente *ítem* es *name_index* 0x000A y hace referencia al índice `10` del *constant_pool* e indica el nombre del método. El nombre del método es <*init*> o el constructor de la clase en este caso es `HolaMundo`.

```
10 CONSTANT_Utf8 : <init>
```

El *ítem descriptor_index* 0x000B es el descriptor del método constructor y hace referencia al índice `11` del *constant_pool*. Ahora sabemos que el método <*init*> no recibe argumento y tampoco devuelve un valor.

```
11 CONSTANT_Utf8 : ()V
```

El *ítem attributes_count* 0x0001 indica si el método tiene atributos; del capítulo anterior sabemos que un método puede tener los atributos *Code, Exceptions, Synthetic* y *Deprecated.* En esta ocasión se trata del atributo *Code* ya que *attribute_name_index* 0x000c es el índice `12` del *constant_pool* y representa el nombre del atributo.

```
12 CONSTANT_Utf8 : Code
```

El atributo *Code* es una estructura de longitud variable y contiene las instrucciones para la JVM e información auxiliar del método. En la figura 5-8 esta marcado en color verde el *ítem Code[]* que son las instrucciones que ejecutará la JVM, `2a b7 00 01 b1`.

Cada byte del *ítem Code[]* son instrucciones u *opcodes*, **Marago** toma este fragmento de *bytecode* y lo desensambla obteniendo las instrucciones para la JVM –esto se explica un poco más adelante-. Con la información obtenida podemos tener una primera aproximación del método constructor o método <*init*>.

```
public HolaMundo()
{
    2a b7 00 01 b1
}
```

El segundo método se obtiene de la misma forma en que obtuvimos el método constructor, en la figura 5-8 se muestra en color amarillo el *ítem Code[]* `b2 00 02 12 03 b6 00 04 b1.`

```
public static void main(String[] string0)
{
       b2 00 02 12 03 b6 00 04 b1
}
```

La clase `Methods` posee los siguientes métodos que le ayudan a construir la estructura *method_info:*

- `countMethods()` Obtiene el número de métodos que fueron declarados en el archivo *.class*.
- `setMethods()` Construye la estructura *Methods_info*.
- `puzzleMethods()` Crea la firma del método.
- `getMethDescriptor()` Obtiene los argumentos pasados al método.
- `getParams()` Obtiene el argumento que devuelve el método.

Las variables que se declaran dentro de cada método sólo existen cuando el método se está ejecutando, a esto se le llama **ámbito de las variables** o **alcance de las variables**; cuando el método termina el recolector de basura (*garbage collector, gc*) elimina las variables declaradas.

El método `nameGenerator` de la clase `Methods` genera el nombre de las variables y las almacena en un vector. La idea es sencilla, Marago genera el nombre de las variables concatenando el tipo de dato al que pertenece la variable y un contador.

```
public static String nameGenerator(String strNameVar)
{
  StringBuffer sbNameVar = new StringBuffer();
  int countVar = 0;
  boolean flagInsert = true;

  if(strNameVar.indexOf("[")!=-1)   strNameVar =
           strNameVar.substring(0,strNameVar.indexOf("[")).toLowerCase().trim();

  strNameVar = strNameVar.toLowerCase().trim();
  while(flagInsert)
  {
   sbNameVar.delete(0,sbNameVar.length());
   sbNameVar.append(strNameVar + Integer.toString(countVar));
   if(Methods.vecLocalVars.indexOf(sbNameVar.toString().trim())==-1&&
      Fields.vecNameFields.indexOf(sbNameVar.toString().trim())==-1)
   {
     Methods.vecLocalVars.addElement( sbNameVar.toString().trim() );
     flagInsert = false;
   }
```

```
   else    countVar++;
   }
  return sbNameVar.toString().trim();
}
```

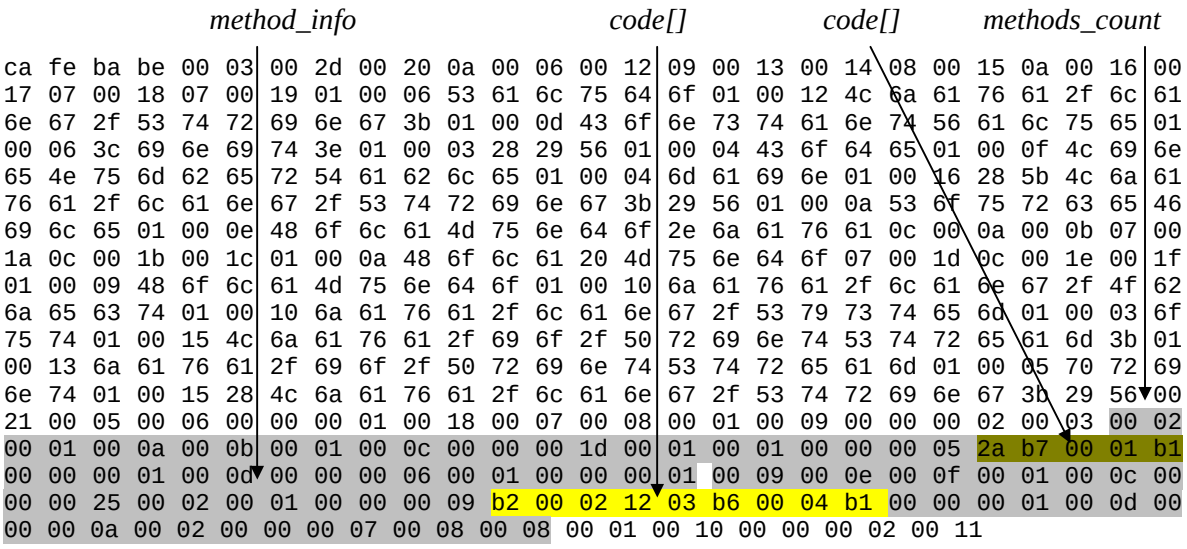

Figura 5-8 *ítems* que construyen los métodos dentro del archivo *.class*

5.2.5 Reconstruyendo los Atributos.

La última parte de la estructura *classFile* la forman *attributes_count* y *attributes[];* la clase `Attributes` se encarga de construir los atributos para *classFile*, *field_info*, *method_info* y c*ode_attribute.*.

De la figura 5-9, *attributes_count* tiene el valor 0x0001, por lo que solo tenemos un atributo. La estructura *attributes_info* se forma con *attribute_name_index* 0x0010, un índice del *constant_pool* que contiene el nombre del atributo.

```
16 CONSTANT_Utf8 : SourceFile
```

Sabemos ahora que el atributo es *SourceFile* que tiene una longitud fija y representa el nombre del archivo fuente. El *ítem sourcefile_index* 0x0011 es un índice del *constant_pool* que representa el nombre del archivo fuente.

```
17 CONSTANT_Utf8 : HolaMundo.java
```

La clase `Attributes` contiene métodos que construyen los atributos del archivo .class: *ConstantValue, Code, Exceptions, InnerClasses, Synthetic, SourceFile, LineNumberTable, LocalVariableTable y Deprecated*

- `InnerClassesAtt()` Construye el atributo *InnerClasses.*
- `SourceFileAtt()` Construye el atributo *SourceFile.*
- `GetFieldAtt()` Construye los atributos de la estructura *field_info.*
- `GetMethAtt()` Construye los atributos de la estructura *methods_info.*
- `CodeAtt()` Construye el atributo *Code.*
- `getCodeException()` Obtiene la tabla `try-catch-finally.`
- `getCodeAtt()` Obtiene los *opcodes* del método.
- `exceptionAtt()` Obtiene las excepciones lanzadas por el método.
- `constantValueAtt()` Construye el atributo *InnerClasses.*

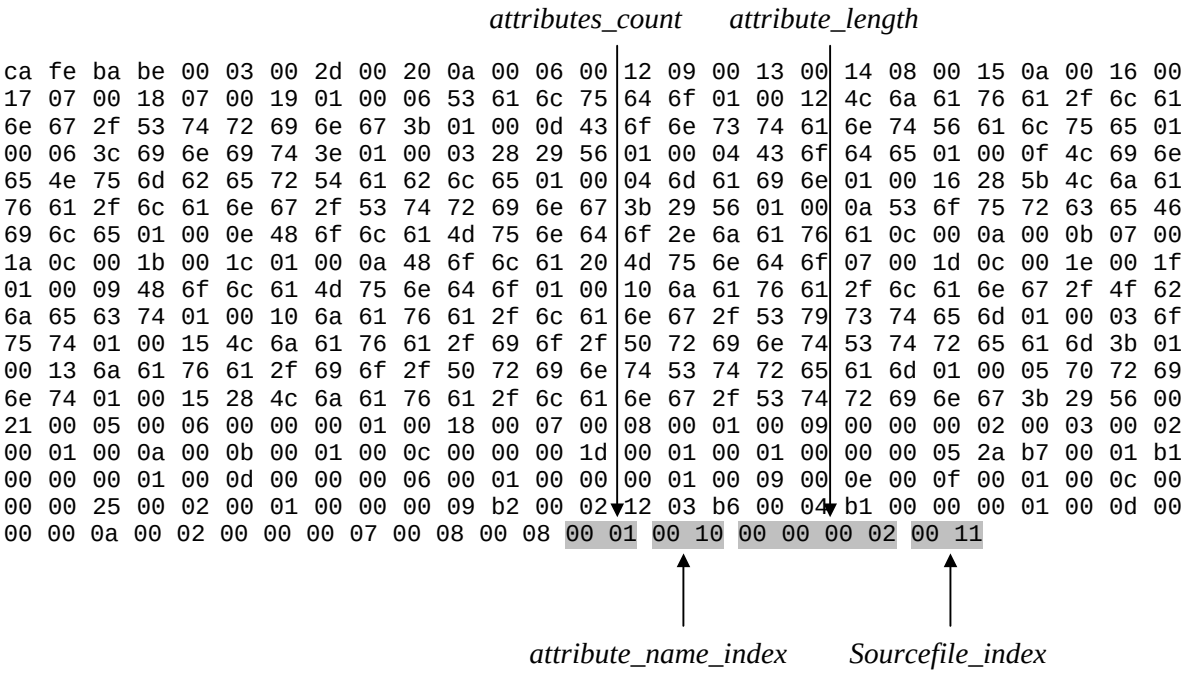

Figura 5-9 *ítems* para la clase Attributes.

5.2.6 Desensamblando el código.

Ya hemos leído el archivo *.class* y obtuvimos información valiosa. Nos falta darle orden a la información y obtendremos un código muy semejante en apariencia a un archivo fuente Java.

```
import java.io.PrintStream;

public class HolaMundo
{
   static final String Saludo = "Hola Mundo";

      public HolaMundo (  )
      {
          2a b7 00 01 b1
      }
      public static void main ( String[] string0 )
      {
          b2 00 02 12 03 b6 00 04 b1
      }
}
```

Observemos que nuestra clase está parcialmente decompilada, obtuvimos la firma de la clase, de los métodos y los paquetes que importa, pero aun nos falta obtener el código de cada método; ésta es la parte más complicada, ya que la JVM es una pila. Convertir código de pila a código de alto nivel no es asunto sencillo y queda fuera del alcance de esta tesis. Sin embargo, podemos desensamblar ese código y obtener los *opcodes* para cada método.

Los *opcodes* son instrucciones en lenguaje ensamblador que ejecuta la JVM, algunos *opcodes* utilizan un byte, otros realizan instrucciones más complejas –ya que deben hacer llamadas al *constant_pool* o realizar saltos, por lo que pueden ocupar varios bytes-. La clase `Converter` tiene el método `toByteCode()` que recibe los *opcodes* y los desensambla.

Para el método <*init*>, tenemos el código que queremos desensamblar `2a b7 00 01 b1` y obtenemos el siguiente resultado.

```
0 aload_0
1 invokespecial #1 <Method: void Object()> <nameClass: Object>
4 return
```

La instrucción **aload_0** corresponde al byte 0x2a(42) que inserta en la pila la referencia a un objeto. En el capítulo 3 mencionamos que los métodos de instancia (no `static`) deben pasar una referencia oculta a la pila de operadores; ésta referencia es la palabra `this.`

El siguiente *opcode* es **invokespecial** 0xb7(183) que invoca al constructor de la superclase. El *opcode* **invokespecial** es seguido por un entero de 16 bits sin signo, 0x0001, que es un índice en el *constant_pool.*

```
1 CONSTANT_Methodref : 6,18
6 CONSTANT_Class : 25
10 CONSTANT_Utf8 : <init>
11 CONSTANT_Utf8 : ()V
18 CONSTANT_NameAndType : 10,11
25 CONSTANT_Utf8 : java/lang/Object
```

El índice `1` en el *constant_pool* es una estructura *Constant_Methodref* que es una referencia a un método; el primer índice en *Constant_Methodref* es la clase a la que pertenece el método, `java.lang.Object`, el segundo índice se refiere al nombre y descriptor del método.

Del *constant_pool* sabemos que la instrucción **invokespecial** invoca al método <*init*> de la clase `Object`, es decir; llama al constructor de la superclase. El último *opcode* **return** 0xb1(177) regresa del método.

Conocemos que el constructor invoca al constructor de la superclase; en líneas arriba, mencionamos que cuando no definimos un constructor, la JVM implícitamente crea uno que invoca al constructor de la superclase. Podemos afirmar que en nuestra clase `HolaMundo` no se definió un constructor o esta vacío.

Para el método `main,` tenemos que el código a desensamblar es `b2 00 02 12 03 b6 00 04 b1.`

```
0 getstatic #2 <Field PrintStream out>
3 ldc #3 <String "Hola Mundo">
5 invokevirtual #4 <Method: void print(String string0)><nameClass: PrintStream>
8 return
```

El primer *opcode* es **getstatic** 0xb2(178) que representa la referencia a una variable`;` **getstatic** es seguido por un entero de 16 bits sin signo que es un índice al *constant_pool*.

```
2 CONSTANT_Fieldref : 19,20
19 CONSTANT_Class : 26
20 CONSTANT_NameAndType : 27,28
26 CONSTANT_Utf8 : java/lang/System
27 CONSTANT_Utf8 : out
28 CONSTANT_Utf8 : Ljava/ PrintStream;io/
```

Del *constant_pool*, podemos decir que **getstatic** obtiene la referencia de la variable `out` que pertenece a la clase `System`.

El siguiente *opcode* es **ldc** 0x12(18) que inserta en la pila un elemento del *constant_pool*, **ldc** es seguido por un entero de 8 bits sin signar, justamente el índice del *constant_pool*.

```
3 CONSTANT_String : 21
21 CONSTANT_Utf8 : Hola Mundo
```

El siguiente *opcode* es **invokevirtual** 0xb6 (182) que llama a un instancia de un método y es seguido por un entero de 16 bits sin signo, que es un índice en el *constant_pool.*

```
4 CONSTANT_Methodref : 22,23
22 CONSTANT_Class : 29
23 CONSTANT_NameAndType : 30,31
29 CONSTANT_Utf8 : java/io/PrintStream
30 CONSTANT_Utf8 : print
31 CONSTANT_Utf8 : (Ljava/lang/String;)V
```

invokevirtual llama al método `print`, de la clase `PrintStream` que recibe como argumento una cadena `String` y no devuelve valor. Finalmente el *opcode* **return** 0xb1(177) regresa del método.

Con esta información podemos asegurar que el método `main` invoca al método `print` para imprimir el mensaje **Hola Mundo**.

En la figura 5-10 presentamos el código que obtuvo el desensamblador **Marago** de la clase `HolaMundo`.

```
import java.io.PrintStream;

public class HolaMundo
{
  static final String Saludo = "Hola Mundo";

  public HolaMundo()
  {
     0 aload_0
     1 invokespecial #1 <Method: void Object()><nameClass: Object>
     4 return
  }

  public static void main(String[] string0)
  {
    0 getstatic #2 <Field PrintStream out>
    3 ldc #3 <String "Hola Mundo">
    5  invokevirtual  #4  <Method:  void  print(String  string0)><nameClass:
PrintStream>
    8 return
  }
}
```

Figura 5-10 Código obtenido por **Marago** desensamblado de la clase HolaMundo.

El código desensamblado por **Marago** nos da una idea muy cercana de cómo pudo haber sido programada la clase, pero lo más importante es que nos permite conocer a fondo lo que la JVM hará cuando interprete el archivo *.class*.

5.3 Entendiendo el código.

En las siguientes líneas veremos como se traduce código Java a código ensamblador, exponemos algunos ejemplos de código fuente Java y la forma en que la JVM los entiende. Encontraremos generalmente que una sentencia en código Java se traduce en varias instrucciones u *opcodes* en lenguaje ensamblador.

El conjunto de instrucciones de la JVM (*Instruction Set of JVM*) se distingue porque sus *opcodes* dependen del tipo de dato que se utiliza, la JVM frecuentemente toma ventaja de algunos operandos ya que los *opcodes* los utiliza de forma implícita. Por ejemplo, la instrucción iconst_0 inserta un int 0 en la pila, de esta forma ya no es necesario almacenar el operando para decodificarlo y operarlo; otra forma de almacenar un int 0 es a través de la instrucción bipush 0 que es correcta pero menos eficiente.

Los fragmentos de código expuestos líneas abajo tratan de ejemplificar de manera sencilla como interpreta código la JVM.

El método Suma declara 3 variables de tipo entero y les asigna un valor.

```
public void suma()
{
  int a, b, c;
  a = 1;
  b = 2;
  c = a + b;
}
```

Marago obtiene el código desensamblado.

```
public void suma()
{
  0 iconst_1     ; Inserta el int 1 en la pila.
  1 istore_1     ; Almacena el valor de la pila en la variable uno (a=1).
  2 iconst_2     ; Inserta el int 2 en la pila.
  3 istore_2     ; Almacena el valor de la pila en la variable dos (b=2).
  4 iload_1      ; Inserta la variable local uno en la pila.
  5 iload_2      ; Inserta la variable local dos en la pila.
  6 iadd         ; Suma los dos valores de la pila (a+b).
  7 istore_3     ; El resultado lo asigna a la variable tres (c=a+b).
  8 return       ; regresa del Método.
}
```

El ejemplo `suma()` utiliza la instrucción **iconst_<n>** para insertar en la pila (*java stack*) el valor que será asignado a la variable local 0. El *opcode* **istore_<*varnum*>** toma el valor de la pila y lo asigna a la variable local 0.

La instrucción **iload_<*varnum*>** inserta el valor de la variable local <*varnum*> en la pila, **iadd** suma las dos variables y el resultado se almacena en una tercer variable.

Repetiremos el ejemplo, sólo que ahora utilizaremos variables de tipo `short` en vez de variables enteras.

```
public void sumaShort()
{
  short a, b;
  a = 1;
  b = 2;
  short c = (short)(a + b);
}
```

El código desensamblado por **Marago** es:

```
public void sumaShort()
{
  0 iconst_1     ; Inserta el int 1 en la pila, los valores short son tratados
                 ; como enteros.
  1 istore_1
  2 iconst_2
  3 istore_2
  4 iload_1
  5 iload_2
  6 iadd
  7 i2s          ; Trunca el entero a un valor short (cast)
  8 istore_3
  9 return
}
```

La JVM trata a los números `short` y `bytes` como enteros, podemos ver en el código del método `sumaShort()`que al asignar los valores a las variables `a` y `b` utiliza las mismas instrucción que en el método `suma()`.

El método `incremento` es un ciclo que se repite hasta que la variable alcanza el valor diez.

```
void incremento()
{
  int var = 0;
  while ( var <= 10)
  {
    var++;
  }
}
```

El código desensamblado obtenido por **Marago** es:

```
void incremento ()
{
  0 iconst_0     ; Inserta el int 0 en la pila.
  1 istore_1     ; Almacena el valor de la pila en la variable uno (var=0).
  2 goto 8
  5 iinc 1 1     ; Incrementa en 1 a la variable local uno (var++).
  8 iload_1      ; Inserta el valor de la variable local uno en la pila.
  9 bipush 10    ; Inserta el int 10 en la pila.
  11 if_icmplt 5 ; Compara y brinca si se cumple la condición(var<=10).
  14 return
}
```

El método inserta en la pila el valor de la variable local cero (`var`) y el valor `10` con la instrucción **bipush**. La instrucción **if_icmplt** toma los dos valores y los compara. Si la condición se cumple (si es menor que), se realiza un salto a la dirección *pc* + *branchoffset* (5). Si la condición no se cumple, se continua con el siguiente *opcode*(14).

A continuación mostramos dos métodos con el propósito de ejemplificar la diferencia entre un método de clase y uno de instancia.

```
float metodoDeInstancia(float a, float b)
{
  return a + b;
}
static float metodoDeClase(float a, float b)
{
  return a + b;
}
```

Por convención, un método de instancia pasa una referencia a su instancia en la variable local 0. En el lenguaje de programación Java la instancia es accesible vía la palabra reservada `this`.

```
float metodoDeInstancia(float float0, float float1)
{
  0 fload_1      ; Inserta en la pila el valor de la variable local 1
  1 fload_2      ; Inserta en la pila el valor de la variable local 2
  2 fadd         ; Suma las variables float de la pila
  3 freturn      ; Devuelve el valor float
}
static float metodoDeClase(float float0, float float1)
{
  0 fload_0      ; Inserta en la pila el valor de la variable local 0
  1 fload_1      ; Inserta en la pila el valor de la variable local 1
  2 fadd         ; Suma las variables float de la pila
  3 freturn      ; Devuelve el valor float
}
```

Los métodos de clase (`static`) no tienen instancia, así que para ellos no es necesaria la referencia en la variable cero. Un método de clase comienza utilizando la variable local en el índice cero. Otro detalle interesante es cuando se suman dos valores, la JVM tiene *opcodes* para cada tipo de datos; si suma dos enteros es invocada la instrucción **iadd**, si son dos valores `float` se utiliza **fadd**, etc.

El siguiente ejemplo es el método `sumaArg(),` con él mostraremos cual es la forma en que la JVM utiliza los argumentos que son pasados a los métodos. Recuerde que los objetos son pasados por referencia, mientras que los tipos de dato primitivos son pasados por valor.

```
int sumaArg(int a)
{
  int b = 10;
  return a + b;
}
```

El método es muy sencillo, recibe un valor y devuelve un entero aumentado en diez unidades.

```
int sumaArg(int int0)
{
  0 bipush 10    ; Inserta en la pila el valor 10
  2 istore_2     ;  Almacena el valor en la variable local dos (b=10), la
                 ;  variable local uno es el argumento pasado al método.
  3 iload_1      ; Inserta en la pila el valor del argumento(a).
  4 iload_2
  5 iadd         ; Suma los dos valores.
  6 ireturn      ; Devuelve el entero.
}
```

A continuación mostramos el método `restaDouble()` que trabaja con números double, la JVM utiliza dos variables para almacenar números `double` y `long`.

```
double restaDouble()
{
  double i = 10;
  double j = 20;
  return j-i;
}
```

El método `restaDouble()` devuelve un valor `double` producto de una resta, el código desensamblado por **Marago** es el siguiente:

```
double restaDouble ()
{
  0 ldc2_w #2 <Double 10.0>   ; Inserta en la pila el valor double 10.0
  3 dstore_1                  ; Almacena el valor en las variables 1 y 2
                              ; (i=10).
  4 ldc2_w #4 <Double 20.0>   ; Inserta en la pila el valor double 20.0
  7 dstore_3                  ; Almacena el valor en las variables 3 y 4
                              ; (j=20).
  8 dload_3
  9 dload_1
  10 dsub                     ; Resta dos números double (j-i).
  11 dreturn                  ; Devuelve el valor double.
 }
```

Cuando se trabaja con números `double` y `long` JVM utiliza dos variables para almacenarlos, aunque en realidad sólo la primera variable contiene el dato, la segunda variable es `null`; JVM almacena así los números para ser congruente con el sistema operativo.

El método `decisión()` es una estructura de control, verifica el valor de la variable `b` y asigna un nuevo valor a la variable `a`.

```
void decision()
{
  boolean b = true;
  float a = 0;
  if( b )
  {
    a = 1e2F;
  }
}
```

El código desensamblado por **Marago** es:

```
void decision()
{
  0 iconst_1        ; Inserta en la pila el valor entero 1.
  1 istore_1        ; Almacena el valor de la pila en la variable local 1
  2 fconst_0
  3 fstore_2
  4 iload_1
  5 ifeq 1          ; Compara y brinca si la condición se cumple (if(b)).
  8 ldc #2 <Float 100.0>     ; Inserta en la pila el valor float 100.0
  10 fstore_2       ; Asigna el valor la variable local 2 ;(a=1e2F).
  11 return
}
```

La declaración de variables `boolean` es igual al de las variables de tipo entero, los tipos de datos `boolean y short` son tratados como datos `int`. El *opcode* **ifeq** toma el valor `int` de la pila, si la condición se cumple (si es igual a cero), realiza el salto a la dirección *pc + branchoffset;* si la condición no se cumple la ejecución continua con el siguiente *opcode.*

El siguiente fragmento de código es una estructura **if–else**, el método `decision2()` asigna un valor a la variable local `long` de acuerdo al valor del argumento que recibe.

```
long decision2(float var)
{
  long l;
  if( var < 1 )
  {  l = 10L;  }
  else
  {  l = 20L;  }
  return l;
}
```

El código desensamblado por **Marago** es:

```
long decision2(float float0)
{
  0 fload_1
  1 fconst_1
  2 fcmpg                     ; Compara dos números float
  3 ifge 13                   ; Brinca si es más mayor o igual que cero.
  6 ldc2_w #2 <Long 10>
  9 lstore_2
  10 goto 17
  13 ldc2_w #4 <Long 20>
  16 lstore_2
  17 lload_2
  18 lreturn
}
```

El *opcode* **fcmpg** toma dos números `float` de la pila y los compara utilizando reglas de IEEE754. Si los dos números son iguales se inserta en la pila el valor cero. Si el segundo es mayor que el primero se inserta el valor 1, si el primero es más grande que el segundo se inserta el valor -1. **ifge** toma el resultado de la comparación anterior y brinca si es mayor o igual que cero. Finalmente **lreturn** regresa el valor `long`. Recuerde que los datos de tipo `long` se almacenan en dos variables locales.

A continuación veremos como se traduce un bloque `for` a lenguaje ensamblador, el método `ejemploFor()` declara un bloque que itera 100 veces decrementando el valor de la variable local `k`.

```
void ejemploFor()
{
  int k = 100;
  for(int i = 0; i<100; i++)
  {
    k--;
  }
}
```

El código para la JVM es el siguiente:

```
void ejemploFor()
{
  0 bipush 100   ; Inserta el valor 100 en la pila
  2 istore_1     ; Almacena el valor de la pila en la variable 1 (int k=100).
  3 iconst_0
  4 istore_2     ; Asigna el valor 0 a la variable local dos (int i=0).
  5 goto 14
  8 iinc 1 –1    ; Decrementa a la variable local 1 en una unidad (k--).
  11 iinc 2 1    ; Incrementa a la variable local 2 en una unidad (i++).
  14 iload_2
  15 bipush 100
  17 if_icmplt 8 ; Compara y brinca si es menor que (i<100)
  20 return
}
```

El *opcode* **if_icmplt** toma dos valores de tipo `int` de la pila y los compara. Si la condición se cumple (menor que) se realiza un salto a la dirección *pc* + *branchoffset*, donde *pc* es la dirección del *opcode* actual en el *bytecode* y *branchoffset* es un entero de 16 bits con signo que sigue al *opcode;* **if_icmplt** es el encargado de revisar la condición del ciclo `for.`

Una de las estructuras más complicadas de construir para la JVM es `switch`, ya que debe calcular los saltos relativos para cada uno de los casos (`case`). El siguiente método utiliza una estructura switch para decidir que valor asigna a la variable local `res`.

```
float ejemploSwitch(int op)
{
  float res = 0;
  switch( op)
  {
    case 1:      res = 1;
             break;
    case 2:      res = 2;
             break;
    case 3:      res = 3;
             break;
    default:     res = 0;
             break;
    }
    return res;
}
```

El código desensamblado por Marago es el siguiente:

```
float ejemploSwitch(int int0)
{
  0 fconst_0
  1 fstore_2
  2 iload_1                   ; Inserta el valor del argumento en la pila.
  3 tableswitch               ; Ejecuta un salto calculado a alguna de las
                              ; sentencias case.
            case 1: 28        ; Si se cumple el caso uno brinca  al opcode 28
            case 2: 33        ; Si se cumple el caso dos brinca  al opcode 33
            case 3: 38        ; Si se cumple el caso tres brinca  al opcode 38
            default: 44       ; Si no se cumple algún caso uno brinca  al
                              ; opcode 44
  28 fconst_1
  29 fstore_2
  30 goto 46
  33 fconst_2
  34 fstore_2
  35 goto 46
  38 ldc #2 <Float 3.0>       ; Inserta el valor float 3.0 en la pila
  41 goto 46
  44 fconst_0
  45 fstore_2
  46 fload_2
  47 freturn
}
```

tableswitch toma un valor entero de la pila y ejecuta un salto a la dirección *pc + default_offset*, donde *pc* es la dirección del *opcode* **tableswitch** y *default_offset* son 32 bits con signo que siguen a continuación del opcode actual. La instrucción brinca a la dirección del *opcode* donde se cumple la igualdad.

Puede observar que para insertar el valor 1.0 ó 2.0 en la pila, la JVM utiliza las instrucciones **fconst_1** y **fconst_2** respectivamente. Debido a que no existe la instrucción **fconst_3** la JVM debe utilizar la instrucción **ldc** que inserta en la pila un elemento del *constant_pool*.

La instrucción **lookupswitch** también se utiliza en sentencias `switch`, cuando las sentencias `case` no están ordenadas se utiliza **lookupswitch** en vez de **tableswitch. lookupswitch** toma un valor `int` de la pila y busca en la tabla una entrada <*key*> que sea igual. Si la comparación es encontrada, brinca a la dirección correspondiente. Si no es encontrada, brinca a la dirección <*default_offset*>.

En muchos casos trabajamos con arreglos, Java los trata como objetos pero tiene instrucciones especiales para trabajar con ellos; el método `arreglo()` declara un arreglo y lo opera.

```
void arreglo()
{
  int[] var1= {1,2,3,4};
  int var2;
  var2 = var1[0] + var1[3];
}
```

El código obtenido por **Marago** es:

```
void arreglo()
{
  0 iconst_4     ; Inserta el int 4 en la pila.
  1 newarray int ; Crea un arreglo de tipo int y lo almacena en la
                 ; variable local 1.
  3 dup          ; Duplica la referencia del arreglo en la pila
  4 iconst_0     ; Inserta el int 0 en la pila.
  5 iconst_1     ; Inserta el int 1 en la pila.
  6 iastore      ; Almacena el valor 1 en el índice 0 del arreglo.
  7 dup
  8 iconst_1
  9 iconst_2
  10 iastore     ; Almacena el valor 2 en el índice 1 del arreglo.
  11 dup
  12 iconst_2
  13 iconst_3
  14 iastore     ; Almacena el valor 3 en el índice 2 del arreglo.
  15 dup
  16 iconst_3
  17 iconst_4
  18 iastore     ; Almacena el valor 4 en el índice 3 del arreglo.
  19 astore_1
  20 aload_1     ; Inserta la referencia del arreglo en la pila.
  21 iconst_0    ; Inserta el int 0 en la pila.
  22 iaload      ; Inserta el elemento 0 de la variable local 1 en la pila.
  23 aload_1
  24 iconst_3
  25 iaload      ; Inserta el elemento 3 de la variable local 1 en la pila.

  26 iadd        ; Suma los dos enteros de la pila.
  27 istore_2    ; Asigna el resultado en la variable local 2.
  28 return
 }
```

La instrucción **newarray** crea arreglos de una dimensión para `boolean`, `int, char, float, double, byte, short,` o `long`, en este caso, crea un arreglo de una dimensión de tipo `int` y la referencia la almacena en la variable local 1.

iastore almacena un entero en un arreglo; toma de la pila el valor entero, el índice del arreglo y la referencia al arreglo, es por eso que aparece la instrucción **dup** que duplica la referencia del arreglo ya que **iastore** la necesita. **iaload** inserta en la pila el valor de un índice de un arreglo. Cuando se crean arreglos multidimensionales JVM utiliza la instrucción **multinewarray**.

Veremos ahora cómo crear instancias de clases, nuestro ejemplo `Objetos` crea un nuevo objeto y modifica su atributo.

```
public class Objetos
{
  String atributo;

  void creandoObjetos()
  {
    Objetos obj = new Objetos();
    obj.atributo = "azul";
  }
}
```

El código obtenido por **Marago** es:

```
public class Objetos
{
  String atributo;

  public Objetos()
  {
    0 aload_0
    1 invokespecial #1 <Method: void Object()><nameClass: Object>
    4 return
  }

  void creandoObjetos()
  {
    0 new #2 <Class Objetos>
    3 dup
    4 invokespecial #3 <Method: void Objetos()><nameClass: Objetos>
    7 astore_1
    8 aload_1
    9 ldc #4 <String "azul">
    11 putfield #5 <Field String atributo>
    14 return
  }
}
```

En ejemplos anteriores explicamos el método <*init*> o constructor, debido a que nuestra clase no definió ningún constructor, es de esperarse que el constructor desensamblado invoque al constructor de la superclase.

Dentro del método `creandoObjetos()`, el primer *opcode* **new** crea un nuevo objeto e inserta la referencia en la pila, en este caso, crea un objeto que pertenece a la clase `Objetos`; el *opcode*

invokespecial llama al método `Objetos()` que es el constructor de la clase y es el encargado de inicializar el objeto; con la instrucción **astore_1** guardamos la referencia del objeto creado en la variable local 1. **aload_1** inserta la referencia del objeto en la pila para que **putfield** modifique su atributo.

El manejo de excepciones es importante en Java. Una excepción es un evento inesperado que el interprete no puede manejar, para ello, el lenguaje de programación Java incorpora un manejador de excepciones a través de bloques `try-catch-finally.`

Manejar excepciones proporciona una manera limpia de verificar errores, en nuestro siguiente ejemplo veremos como la JVM maneja las excepciones.

```
public class Excepciones
{
  public static void main(String[] args)
  {
    try
    {
      int number = Integer.parseInt(args[0]);
      System.out.print(number);
    }

    catch (NumberFormatException e )
    {
       e.printStackTrace();
    }
  }
}
```

El código obtenido por **Marago** es el siguiente:

```
public class Excepciones
{

  public Excepciones()
  {
    0 aload_0
    1 invokespecial #1 <Method: void Object()> <nameClass: Object>
    4 return
  }

  public static void main(String[] string0)
  {
    0 aload_0
    1 iconst_0
    2 aaload
    3 invokestatic #2 <Method: int parseInt(String string0)><nameClass: Integer>
    6 istore_1
    7 getstatic #3 <Field PrintStream out>
    10 iload_1
    11 invokevirtual #4 <Method: void print(int int0)><nameClass: PrintStream>
    14 goto 22
    17 astore_1
```

```
  18 aload_1
  19 invokevirtual #6 <Method: void printStackTrace()><nameClass: Throwable>
  22 return

  ************************************************
  *************** Exception table ****************
  ************************************************
       from        to       target        type
        0          14        17     NumberFormatException
  ************************************************
 }
}
```

Cuando se definen bloques `try-catch` o `try-catch-finally` **Marago** presenta una tabla de excepciones al final del método. La tabla tiene 4 campos **from, to, target** y type.

- **from** indica a partir de que *opcode* comienza en bloque `try`.
- **to** indica en que *opcode* termina el bloque `try`.
- **target** indica donde se localiza el bloque `catch`. Si algún *opcode* dentro del intervalo **from-to** lanza una excepción, el siguiente *opcode* será el que indique **target**.
- **type** indica que excepción lanza el bloque `try`.

5.4 Clases Inner

Existen 4 tipos de clases anidadas o *inner clases*: *top level class (*clases no `static` que son miembro de la clase*)*, *member class* (clases `static`), clases definidas dentro de método y clases anónimas. **Marago** puede desensamblar las *inner clases* a excepción de las clases definidas dentro de un método, esto es porque no dejan ningún rastro al compilarse. A continuación presentamos uns ejemplo de *inner clases* con el fin de mostrar como trabaja **Marago**.

```
class Outer
{
  Outer()
  {
    System.out.print("Soy la clase Outer");
  }

  class Inner
  {
    Inner()
    {
      System.out.print("Soy la clase Inner");
    }
  }
}
```

El código obtenido por **Marago** es el siguiente:

```
import java.io.PrintStream;

class Outer
{
  Outer ()
  {
    0 aload_0
    1 invokespecial #1 <Method: void Object()><nameClass: Object>
    4 getstatic #2 <Field PrintStream out>
    7 ldc #3 <String "Soy la clase Outer">
    9   invokevirtual   #4   <Method:   void   print(String   string0)><nameClass:
PrintStream>
    12 return
  }
  class Outer$Inner
  {
    private final Outer this$0;

    Outer$Inner(Outer outer0)
    {
      0 aload_0
      1 invokespecial #1 <Method: void Object()> <nameClass: Object>
      4 aload_0
      5 aload_1
      6 putfield #2 <Field Outer this$0>
      9 getstatic #3 <Field PrintStream out>
      12 ldc #4 <String "Soy la clase Inner">
      14  invokevirtual  #5  <Method:  void  print(String  string0)><nameClass:
PrintStream>
      17 return
    }
  }
}
```

CONCLUSIONES

Conclusiones.

Cuando decidimos desarrollar este tema de tesis, no sabíamos –mi director, Alejandro Mancilla y yo- con certeza la dificultad con la que nos íbamos a topar, básicamente debido a la falta de documentación y es que los decompiladores se consideran un tema tabú ya que involucra aspectos legales, propiedad intelectual, ética, etc.

Cuando alguien busca en internet información sobre los decompiladores, seguramente encontrará muchas páginas que hablan de ellos, sin embargo la mayoría de la información no sirve o simplemente es publicidad. Los decompiladores y desensambladores de código son un tema muy popular, pero la mayoría de la personas hablan de ellos como unas herramientas mágicas que pueden solucionar todos los problemas de los programadores. Es por esto, quizá, que llamen tanto la atención.

Aunado a la dificultad de desarrollar decompiladores y desensambladores, los esfuerzos por parte de desarrolladores en el campo de la reingeniería han sido tratados de eliminar por las grandes compañías; un ejemplo es el decompilador **Mocha,** aun en su versión beta, fue comprado por la compañía **Borland International**, de esta manera, los avances conseguidos por Hanpeter Van Vliet se perdieron. El proyecto del decompilador **Mocha** y el ofuscador **Crema** no prosiguieron debido a la muerte de Van Vliet a la edad de 34 años, el 31 de Diciembre de 1996.

Cuando se comenzó la investigación para la tesis, contacté a Godfrey Nolan, un irlandés que en enero del 2002 iba a publicar un libro sobre decompiladores de código Java con la editorial **Appres**. Sin embargo, cerca de salir publicado su libro, algunas empresas presionaron a la editorial para detener su publicación. Después de un año en enero del 2003 el libro no ha visto la luz.

Todos los obstáculos encontrados fueron una motivación más para crear una herramienta que permita generar código Java a partir de la especificación del archivo *.class*, además de establecer las bases para el desarrollo de un decompilador hecho totalmente en Java y quizá lo más importante fue tener una fuente de información formal y fidedigna sobre este tema.

En este trabajo de tesis se estableció la importancia de la tecnología Java, no sólo como lenguaje de programación de alto nivel, sino también como una plataforma basada en software.

Se definieron las características del lenguaje Java: simple, orientado a objetos, distribuido, interpretado, sólido, seguro, de arquitectura neutral, portable, multihilos y dinámico que permiten crear aplicaciones de alto desempeño.

Un precio que debemos pagar por las características del lenguaje Java es su lentitud para compilarse e interpretarse. Sin embargo, es un problema que se está tratando de corregir con la creación de compiladores JIT (*just in time*) o compiladores al instante, que convierten código de *bytecode* en código nativo a velocidades muy cercanas a las de los lenguajes compilados. Cuando el intérprete Java detecta que un método se utilizará varias veces, el compilador JIT compila sólo ese método y lo convierte en código nativo para que se utilice posteriormente.

Originalmente se planteó crear un decompilador, pero al avanzar la investigación sobre el tema, se observó que convertir código de pila a código de alto nivel es bastante complicado, una opción era convertir el código de pila en código de tres direcciones, que es una representación intermedia, para después convertirlo en código de alto nivel, pero el sólo hecho de implementar un algoritmo de este tipo merece ser tratado en un tema de tesis aparte, por lo que se decidió limitar la tesis a un desensamblador.

El desensamblador Marago que se desarrolló en esta tesis, mejoró a **javap,** la versión que **Sun Microsystems** presenta en el paquete de desarrollo JDK. **Marago** presenta el contenido del *constant_pool,* **javap** no; el formato en que aparece el código obtenido por **Marago** es fácil de leer, ya que convierte los nombres calificados de las clases de java; **Marago** es una herramienta que puede crecer ya que originalmente se planteó que fuera un decompilador, además utiliza una interfaz gráfica que facilita su uso y genera código para la Máquina Virtual **Yasmin**.

javap no está programado en Java, por lo que es diferente para cada arquitectura, **Marago** al ser desarrollado enteramente en Java es independiente de la arquitectura en que se ejecuta.

Marago es una aplicación que regenera código java a partir de la especificación del formato del archivo *.class*. y auque no decompilamos por completo el código Java, se establecieron las bases para desarrollar un decompilador y se espera que a corto plazo se implementen los algoritmos que permitan convertir código de pila en código de alto nivel.

APÉNDICE

A

Apéndice A Instrucciones de la Máquina Virtual Java.

Una instrucción de la Máquina Virtual Java consiste en un *opcode* de un byte que especifica la operación a ejecutar, seguido de cero o varios operandos que proveen los argumentos o datos que son utilizados por la operación. Algunas instrucciones no tienen operandos y constan sólo de un *opcode*. A continuación se presenta el conjunto de instrucciones de la Máquina Virtual Java (*Instruction Set*).

nop

Opcode.
0x00 (0) nop ; Hace nada.

No tiene efecto. Los compiladores algunas veces generan **nop** para depurar, probar o cumplir con propósitos de tiempo.

Antes	Después
...	...

aconst_null

Opcode
0x01 (1) aconst_null ; Inserta un valor `null`.

Inserta en la pila una referencia `null`. Los campos, variables arreglos y objetos tienen `null` como valor inicial.

Antes	Después
...	null
	...

iconst_m1

Opcode
0x02 (2) iconst_m1 ; Inserta el entero -1.

Inserta en la pila el entero –1.

Antes	Después
...	-1
	...

iconst_<n>

Opcode
0x03 (3) iconst_0 ; Inserta el entero 0.
0x04 (4) iconst_1 ; Inserta el entero 1.
0x05 (5) iconst_2 ; Inserta el entero 2.
0x06 (6) iconst_3 ; Inserta el entero 3.
0x07 (7) iconst_4 ; Inserta el entero 4.
0x08 (8) iconst_5 ; Inserta el entero 5.

Representa la serie de *opcodes* iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, e iconst_5. Son usados para insertar constantes enteras desde cero hasta cinco en la pila.

Antes	Después
...	<n>
	...

lconst_<l>

Opcode
0x09 (9) lconst_0 ; Inserta `long` 0.
0x0A (10) lconst_1 ; Inserta `long` 1.

Representa los *opcodes* lconst_0 y lconst_1, que son usados para insertar en la pila las constantes `long` 0 y 1 .

Antes	Después
...	<l> word 1
	<l> word 2
	...

fconst_<f>

Opcode
0x0B (11) fconst_0 ; Inserta el número `float` 0.0.
0x0C (12) fconst_1 ; Inserta el número `float` 1.0.
0x0D (13) fconst_2 ; Inserta el número `float` 2.0.

Representa la serie de *opcodes* fconst_0, fconst_1 y fconst_2 que son usados para insertar en la pila los números flotantes 0.0, 1.0 y 2.0 respectivamente.

Antes	Después
...	<f>
	...

dconst_<d>

Opcode
0x0E (14) dconst_0 ; Inserta el número `double` 0.0.
0x0F (15) dconst_1 ; Inserta el número `double` 1.0
Inserta en la pila el número <*d*> en formato `double`, donde <*d*> puede ser 0 ó 1; aunque existe otra instrucción que realiza la misma tarea, ld2_w 0.0; dconst_<*d*> es más eficiente

Antes	Después
...	<d> word 1
	<d> word 2
	...

bipush

Opcode
0x10 (16) bipush <*n*> ; Inserta en la pila un entero de un byte con signo.

Inserta un entero de un byte con signo en la pila donde –128 <= n <= 127.

Antes	Después
...	<n>
	...

sipush

Opcode
0x11 (17) sipush <*n*> ; Inserta en la pila un entero de dos bytes con signo.

Inserta un entero de dos bytes con signo en la pila donde –32768 <= n <= 32767.

Antes	Después
...	<n>
	...

ldc

Opcode
0x12 (18) ldc <*ítem*> ; Inserta en la pila un elemento del *constant_pool*.

El *opcode* ldc es seguido por un entero de 8 bits sin signo, éste entero es un índice en el *constant_pool* y puede representar las estructuras *CONSTANT_Integer, CONSTANT_Float* o *CONSTANT_String* (si el caso es un *CONSTANT_String*, los datos identifican una estructura *CONSTANT_Utf8* que representa una cadena).

Antes	Después
...	<ítem>
	...

ldc_w

Opcode
0x13 (19) ldc_w <*ítem*> ; Inserta en la pila un elemento del *constant_pool*.

Es la misma instrucción que ldc, sólo que en formato extendido (**wide**), permite índices de dos bytes sin signo, se utiliza cuando el *constant_pool* es muy grande.

ldc2_w

Opcode
0x14 (20) ldc2_w <*ítem*> ; Inserta en la pila un número `long` o `double`.

El *opcode* ldc2_w es seguido por un entero sin signo de dos bytes. Éste entero es el índice del *constant_pool* y representa las estructuras *CONSTANT_Double* o *CONSTANT_Long* las cuales contienen <*ítem*>.

Antes	Después
...	<ítem> word 1
	<item> word 2
	...

iload

Opcode
0x15 (21) iload <*varnum*> ; Inserta en la pila el valor de la variable local <*varnum*>.

Inserta el valor entero de la variable local en la pila. La instrucción puede aparecer en dos formas: en la primera forma, <*varnum*>, está en el un rango de 0 a 0xFF. En la segunda forma (**wide**), <*varnum*> va desde 0 a 0xFFFF. El valor de la variable <*varnum*> es obtenido e insertado en la pila.

Antes	Después
...	Int
	...

lload

Opcode
0x16 (22) lload <*varnum*> ; Inserta en la pila el valor `long` de la variable local <*varnum*> y <*varnum*>+1.

Inserta el valor `long` de la variable local en la pila. La instrucción puede aparecer en dos formas: en la primera forma, <*varnum*>, está en el rango desde 0 a 0xFF. En la segunda forma (**wide**), <*varnum*> va desde 0 a 0xFFFE. El valor de la variable <*varnum*> es obtenido e insertado en la pila.

Antes	Después
...	long word 1
	long word 2
	...

fload

Opcode
0x17 (23) fload <*varnum*> ; Inserta en la pila el valor `float` de la variable local.

Inserta el valor `float` de la variable local <*varnum*>en la pila. La instrucción puede aparecer en dos formas: en la primera forma, <*varnum*>, está en el rango desde 0 a 0xFF. En la segunda forma (**wide**), <*varnum*> va desde 0 a 0xFFFF. El valor de la variable <*varnum*> es obtenido e insertado en la pila.

Antes	Después
...	float
	...

dload

Opcode
0x18 (24) dload <*varnum*> ; Inserta en la pila el valor `double` de la variable local <*varnum*> y <*varnum*>+1 .

Inserta el valor `double` de la variable local <*varnum*> y <*varnum*>+1 en la pila. La instrucción puede aparecer en dos formas: en la primera, <*varnum*>, tienen un rango desde 0 a 0xFF. En la segunda (**wide**), <*varnum*> va desde 0 a 0xFFFE.

Antes	Después
...	double word 1
	double word 2
	...

aload

Opcode
0x19 (25) aload <*varnum*> ; Inserta en la pila la referencia a un objeto que se encuentra en la variable local <*varnum*>.

Regresa la referencia a un objeto que se encuentra en la variable local <*varnum*> y la inserta en la pila. La instrucción toma el parámetro <*varnum*>, un entero sin signo que indica que variable local debe obtener. La instrucción puede aparecer en dos formas: en la primera forma, <*varnum*>, está en el rango desde 0 a 0xFF. En la segunda forma (**wide**), <*varnum*> va desde 0 a 0xFFFF.

Antes	Después
...	Objeto
	...

iload_<n>

Opcode

0x1A (26)	iload_0	; Inserta en la pila el valor de la variable local 0.
0x1B (27)	iload_1	; Inserta en la pila el valor de la variable local 1.
0x1C (28)	iload_2	; Inserta en la pila el valor de la variable local 2.
0x1D (29)	iload_3	; Inserta en la pila el valor de la variable local 3.

Representa la serie de *opcodes* iload_0, iload_1, iload_2 e iload_3 que obtienen el valor entero de la variable local 0, 1, 2 ó 3 y lo inserta en la pila.

Antes	Después
...	Int
	...

lload_<n>

Opcode

0x1E (30)	lload_0	; Inserta en la pila el valor `long` de la variable local 0 y 1.
0x1F (31)	lload_1	; Inserta en la pila el valor `long` de la variable local 1 y 2.
0x20 (32)	lload_2	; Inserta en la pila el valor `long` de la variable local 2 y 3.
0x21 (33)	lload_3	; Inserta en la pila el valor `long` de la variable local 3 y 4.

Regresa el valor `long` de la variable local <*n*> y <*n*>+1 y lo inserta en la pila.

Antes	Después
...	long word 1
	long word 2
	...

fload_<n>

Opcode

0x22 (34)	fload_0	; Inserta en la pila el valor `float` de la variable local 0.
0x23 (35)	fload_1	; Inserta en la pila el valor `float` de la variable local 1.
0x24 (36)	fload_2	; Inserta en la pila el valor `float` de la variable local 2.
0x25 (37)	fload_3	; Inserta en la pila el valor `float` de la variable local 3.

Descripción.
Representa la serie de *opcodes* fload_0, fload_1, fload_2 y fload_3 que insertan en la pila el valor `float` de la variable 0, 1, 2 o 3.

Antes	Después
...	float
	...

dload_<n>

Opcode

0x26 (38)	dload_0	; Inserta en la pila el valor `double` de la variable local 0 y 1.
0x27 (39)	dload_1	; Inserta en la pila el valor `double` de la variable local 1 y 2.

0x28 (40) dload_2 ; Inserta en la pila el valor `double` de la variable local 2 y 3.
0x29 (41) dload_3 ; Inserta en la pila el valor `double` de la variable local 3 y 4.

Obtiene el valor `double` de la variable local <*n*> y <*n*>+1 y lo inserta en la pila.

Antes	Después
...	double word 1
	double word 2
	...

aload_<n>

Opcode
0x2A (42) aload_0 ; Inserta en la pila la referencia a un objeto de la variable local 0.
0x2B (43) aload_1 ; Inserta en la pila la referencia a un objeto de la variable local 1.
0x2C (44) aload_2 ; Inserta en la pila la referencia a un objeto de la variable local 2.
0x2D (45) aload_3 ; Inserta en la pila la referencia a un objeto de la variable local 3.

Obtiene la referencia a un objeto de la variable local <*n*> y la inserta en la pila.

Antes	Después
...	objeto
	...

iaload

Opcode
0x2E (46) iaload ; Obtiene un valor de un arreglo de enteros.

Obtiene el valor de un arreglo de enteros y lo inserta en la pila. *arrayref* es una referencia a un arreglo de enteros e *index* es un entero.

Antes	Después
index	int
arrayref	...
...	

laload

Opcode
0x2F (47) laload ; Obtiene un valor `long` de un arreglo.

Obtiene un valor `long` de un arreglo de números `long` y lo inserta en la pila. *arrayref* es una referencia a un arreglo de enteros e *index* es un entero.

Antes	Después
index	long word 1
arrayref	long word 2
...	...

faload

Opcode
0x30 (48) faload ; Obtiene un valor `float` de un arreglo

Obtiene el valor `float` de un arreglo de números `float` y lo inserta en la pila. *arrayref* es una referencia a un arreglo de enteros e *index* es un entero.

Antes	Después
index	float
arrayref	...
...	

daload

Opcode
0x31 (49) daload ; Obtiene un valor `double` de un arreglo.

Obtiene el valor `double` de un arreglo de números `double` y lo inserta en la pila. *arrayref* es una referencia a un arreglo de enteros e *index* es un entero.

Antes	Después
index	double word 1
arrayref	double word 2
...	...

aaload

Opcode
0x32 (50) daload ; Obtiene la referencia a un objeto dentro de un arreglo.

Obtiene la referencia a un objeto dentro de un arreglo de objetos y lo inserta en la pila. *arrayref* es una referencia a un arreglo de enteros e *index* es un entero.

Antes	Después
index	objeto
arrayref	...
...	

baload

Opcode
0x33 (51) baload ; Obtiene un valor `byte` o `boolean` de un arreglo.

Obtiene un `byte` o un `boolean` de un arreglo y lo inserta en la pila. *arrayref* es una referencia a un arreglo de enteros e *index* es un entero.

Antes	Después
index	valor
arrayref	...
...	

caload

Opcode
0x34 (52) caload ; Obtiene un carácter de un arreglo de caracteres.

Obtiene un carácter de un arreglo de caracteres y lo inserta en la pila. *arrayref* es una referencia a un arreglo de enteros e *index* es un entero.

Antes	Después
index	carácter
arrayref	...
...	

saload

Opcode
0x35 (53) saload ; Obtiene un `short` de un arreglo de números `short`.

Obtiene un valor `short` de un arreglo y lo inserta en la pila. *arrayref* es una referencia a un arreglo de enteros e *index* es un entero.

Antes	Después
index	short
arrayref	...
...	

istore

Opcode
0x36 (54) istore <*varnum*> ; asigna un valor entero a una variable local.

Toma un valor entero de la pila y lo asigna a la variable local <*varnum*>. Puede presentarse en dos formas. En la primera <*varnum*> es un entero sin signo en el rango de 0 a 0xFF. En la segunda (**wide**), <*varnum*> se encuentra en el rango de 0 a 0xFFFF.

Antes	Después
int	...
...	

lstore

Opcode
0x37 (55) lstore <*varnum*> ; Asigna un valor `long` a una variable local.

Toma un valor `long` de la pila y lo asigna a la variable local <*varnum*>. Puede presentarse en dos formas. En la primera, <*varnum*> es un entero sin signo en el rango de 0 a 0xFF. En la segunda (**wide**), <*varnum*> se encuentra en el rango de 0 a 0xFFFE.

Antes	Después
`long word 1`	...
`long word 2`	
...	

fstore

Opcode
0x38 (56) fstore <*varnum*> ; Asigna un valor `float` a una variable local.

Toma un valor `float` de la pila y lo asigna a la variable local <*varnum*>. Puede presentarse en dos formas. En la primera <*varnum*> es un entero sin signo en el rango de 0 a 0xFF. En la segunda (**wide**), <*varnum*> se encuentra en el rango de 0 a 0xFFFF.

Antes	Después
`float`	...
...	

dstore

Opcode
0x39 (57) dstore <*varnum*> ; Asigna un valor `double` a una variable local.

Toma un valor `double` de la pila y lo asigna a la variable local <*varnum*>. Puede presentarse en dos formas. En la primera, <*varnum*> es un entero sin signo en el rango de 0 a 0xFF. En la segunda (**wide**), <*varnum*> se encuentra en el rango de 0 a 0xFFFE.

Antes	Después
`double word 1`	...
`double word 2`	
...	

astore

Opcode
0x3A (58) astore <*varnum*> ; Almacena la referencia a un objeto en una variable local.

Toma la referencia a un objeto y la asigna a la variable local <*varnum*>. Puede presentarse en dos formas. En la primera, <*varnum*> es un entero sin signo en el rango de 0 a 0xFF. En la segunda (**wide**), <*varnum*> se encuentra en el rango de 0 a 0xFFFF.

Antes	Después
objeto	...
...	

istore_<n>

Opcode
0x3B (59) istore_<0> ; Asigna un valor entero a la variable local 0.
0x3C (60) istore_<1> ; Asigna un valor entero a la variable local 1.
0x3D (61) istore_<2> ; Asigna un valor entero a la variable local 2.
0x3E (62) istore_<3> ; Asigna un valor entero a la variable local 3.

Toma un valor entero de la pila y lo asigna a la variable local <*n*>.

Antes	Después
int	...
...	

lstore_<n>

Opcode
0x3F (63) lstore_<0> ; Asigna un valor long a las variables locales 0 y 1.
0x40 (64) lstore_<1> ; Asigna un valor long a las variables locales 1 y 2.
0x41 (65) lstore_<2> ; Asigna un valor long a las variables locales 2 y 3.
0x42 (66) lstore_<3> ; Asigna un valor long a las variables locales 3 y 4.

Asigna el valor `long` de la pila a las variables locales <*n*> y <*n*>+1.

Antes	Después
long word 1	...
long word 2	
...	

fstore_<n>

Opcode
0x43 (67) fstore_<0> ; Asigna un valor `float` a la variable local 0.
0x44 (68) fstore_<1> ; Asigna un valor `float` a la variable local 1.
0x45 (69) fstore_<2> ; Asigna un valor `float` a la variable local 2.
0x46 (70) fstore_<3> ; Asigna un valor `float` a la variable local 3.

Asigna el valor `float` de la pila a la variable local <*n*>.

Antes	Después
float	...
...	

dstore_<n>

Opcode
0x47 (71) dstore_<0> ; Asigna un valor `double` a las variables locales 0 y 1.
0x48 (72) dstore_<1> ; Asigna un valor `double` a las variables locales 1 y 2.
0x49 (73) dstore_<2> ; Asigna un valor `double` a las variables locales 2 y 3.
0x4A (74) dstore_<3> ; Asigna un valor `double` a las variables locales 3 y 4.

Asigna el valor `double` de la pila a las variables locales <*n*> y <*n*> +1.

Antes	Después
`double word 1`	`...`
`double word 2`	
`...`	

astore_<n>

Opcode
0x4B (75) astore_<0> ; Asigna una referencia a un objeto a la variable local 0.
0x4C (76) astore_<1> ; Asigna una referencia a un objeto a la variable local 1.
0x4D (77) astore_<2> ; Asigna una referencia a un objeto a la variable local 2.
0x4E (78) astore_<3> ; Asigna una referencia a un objeto a la variable local 3.

Asigna una referencia a un objeto o a un arreglo a la variable local <*n*>.

Antes	Después
`objeto`	`...`
`...`	

iastore

Opcode
0x4F (79) iastore ; Almacena un entero en un arreglo.

Toma un número entero de la pila y lo almacena en un arreglo de enteros. *arrayref* es una referencia a un arreglo de enteros. *index* es un `int`, *value* es el valor `int` que será almacenado en el arreglo.

Antes	Después
`value`	`...`
`index`	
`arrayref`	
`...`	

lastore

Opcode
0x50 (80) lastore ; Almacena un `long` en un arreglo.

Toma un número `long` de la pila y lo almacena dentro de un arreglo de números `long`. *arrayref* es una referencia a un arreglo de números `long`. *index* es un `int`, *value* es el valor `long` que será almacenado en el arreglo.

Antes	Después
value word 1	...
value word 2	
Index	
Arrayref	
...	

fastore

Opcode
0x51 (81) fastore ; Almacena un `float` en un arreglo.

Toma un número `float` de la pila y lo almacena en un arreglo de números `float`. *arrayref* es una referencia a un arreglo de números `float`. *index* es un `int`, *value* es el valor `float` que será almacenado en el arreglo.

Antes	Después
value	...
index	
arrayref	
...	

dastore

Opcode
0x52 (82) dastore ; Almacena un `double` en un arreglo.

Toma un número double de la pila y lo almacena en arreglo de números `double`. *arrayref* es una referencia a un arreglo de números `double`. *index* es un `int`, *value* es el valor `double` que será almacenado en el arreglo.

Antes	Después
value word 1	...
value word 2	
index	
arrayref	
...	

aastore

Opcode
0x53 (83) aastore ; Almacena la referencia a un objeto en un arreglo.

Almacena la referencia a un objeto en un arreglo de objetos. *arrayref* es una referencia a un arreglo de objetos. *index* es un `int`, *value* es la referencia al objeto que será almacenada en el arreglo.

Antes	Después
value	...
index	
arrayref	
...	

bastore

Opcode
0x54 (84) bastore ; Almacena un valor `byte` o `boolean` en un arreglo.

Toma un `int` de 32 bits de la pila, lo trunca a un byte de 8 bits con signo y lo almacena en un arreglo de bytes. *arrayref* es una referencia a un arreglo de `bytes`. *index* es un `int`, *value* es el valor `int` que será almacenado en el arreglo. También es utilizado con valores boléanos.

Antes	Después
value	...
index	
arrayref	
...	

castore

Opcode
0x55 (85) castore ; Almacena un carácter en un arreglo.

Toma un int de 32 bits de la pila, lo trunca a un valor de 16 bits sin signo y lo almacena en un arreglo de caracteres. *arrayref* es una referencia a un arreglo de caracteres U*nicode* de 16 bits. *index* es un `int`, *value* es el valor `int` que será almacenado en el arreglo.

Antes	Después
value	...
index	
arrayref	
...	

sastore

Opcode
0x56 (86) sastore ; Almacena un `short` en un arreglo.

Toma un entero de la pila, lo trunca a 16 bits con signo y lo almacena en un arreglo de `shorts`. *arrayref* es una referencia a un arreglo de `shorts` de 16 bits con signo. *index* es un `int`, *value* es el valor `int` que será almacenado en el arreglo.

Antes	Después
value	...
index	
arrayref	
...	

pop

Opcode
0x57 (87) pop ; Elimina el *ítem* de la pila.

Elimina el *ítem* de la pila.

Antes	Después
ítem	...
...	

pop2

Opcode
0x58 (88) pop ; Elimina dos *ítems* de la pila.

Elimina dos *ítems* de la pila. Se utiliza para eliminar números `long` o `double`.

Antes	Después
ítem1	...
ítem2	
...	

dup

Opcode
0x59 (89) dup ; Duplica el *ítem* en la pila.

Retira el *ítem* de la pila y lo inserta dos veces. La instrucción no puede ser utilizada para duplicar números `long` o `double`.

Antes	Después
ítem	ítem
...	ítem
	...

dup_x1

Opcode
0x5A (90) dup_x1 ; Duplica el *ítem* en el tope de la pila y lo inserta debajo un segundo *ítem*.

Duplica el *ítem* en el tope de la pila y lo inserta debajo de un segundo *ítem*. Ambos *ítems* deben ser del tamaño de una sola palabra.

Antes	Después
ítem1	ítem1
ítem2	ítem2
...	ítem1
	...

dup_x2

Opcode
0x5B (91) dup_x2 ; Duplica el *ítem* en el tope de la pila y lo inserta debajo un tercer *ítem*.

Duplica el *ítem* en el tope de la pila y lo inserta debajo de un tercer *ítem*. Todos los *ítems* deben ser del tamaño de una sola palabra.

Antes	Después
ítem1	ítem1
ítem2	ítem2
ítem3	ítem3
...	ítem1
	...

dup2

Opcode
0x5C (92) dup2 ; Duplica dos *ítems* en el tope de la pila.

Duplica dos *ítems* en el tope de la pila y los inserta en el mismo orden. Se puede utilizar está instrucción para duplicar *ítems* de una sola palabra o para duplicar *ítems* de dos palabras como `long` y `double`.

Antes	Después
ítem1	ítem1
ítem2	ítem2
...	ítem1
	ítem2
	...

dup2_x1

Opcode
0x5D (93) dup2_x1 ; Duplica dos *ítems* y los inserta debajo de un tercer *ítem*.

Duplica dos palabras en el tope de la pila y los inserta debajo de un tercer *ítem* que debe tener longitud de una palabra.

Antes	Después
ítem1	ítem1
ítem2	ítem2
ítem3	ítem3
...	ítem2
	ítem2
	...

dup2_x2

Opcode
0x5E (94) dup2_x2 ; Duplica dos *ítems* y los inserta debajo de un cuarto *ítem.*

Duplica dos palabras en el tope de la pila y los inserta debajo de un cuarto *ítem;* alternativamente el tercero o cuarto *ítem* puede ser de longitud de una palabra.

Antes	Después
ítem1	ítem1
ítem2	ítem2
ítem3	ítem3
ítem4	ítem4
...	ítem1
	ítem2
	...

swap

Opcode
0x5F (95) swap ; Cambia dos *ítems* en el tope de la pila.

Intercambia dos *ítems* en el tope de la pila que no pertenecen a un valor `long` o `double`.

Antes	Después
ítem1	ítem2
ítem2	ítem1
...	...

iadd

Opcode
0x60 (96) iadd ; Suma dos enteros.

Toma dos enteros de la pila, los suma e inserta el resultado. Un desbordamiento produce un resultado donde los bits de bajo orden son correctos, pero el bit de signo no.

Antes	Después
int	resultado
int	...
...	

ladd

Opcode
0x61 (97) ladd ; Suma dos números `long`.

Toma dos números `long` de la pila, los suma e inserta el resultado. Un desbordamiento produce un resultado donde los bits de bajo orden son correctos, pero el bit de signo no.

Antes	Después
long1 word 1	resultado word 1
long1 word 2	resultado word 2
long2 word 1	...
long2 word 2	
...	

fadd

Opcode
0x62 (98) fadd ; Suma dos números `float`.

Toma dos números `float` de la pila, los suma e inserta el resultado. La suma se hace bajo normas de IEEE.

Antes	Después
float	resultado
float	...
...	

dadd

Opcode
0x63 (99) dadd ; Suma dos números `double`.

Toma dos números `double` de la pila, los suma e inserta el resultado. La suma es calculada utilizando la norma IEEE 754.

Antes	Después
double1 word 1	resultado word 1
double1 word 2	resultado word 2
double2 word 1	...
double2 word 2	
...	

isub

Opcode
0x64 (100) isub ; Resta dos enteros.

Toma dos enteros de la pila, los resta (`int`2 – `int`1) e inserta el resultado.

Antes	Después
int1	resultado
int2	...
...	

lsub

Opcode
0x65 (101) lsub ; Resta dos números `long`.

Toma dos números `long` de la pila, los resta (`long`2 – `long`1) e inserta el resultado.

Antes	Después
long1 word 1	resultado word 1
long1 word 2	resultado word 2
long2 word 1	...
long2 word 2	
...	

fsub

Opcode
0x66 (102) fsub ; Resta dos números `float`.

Toma dos números `float` de la pila, los resta (`float`2 – `float`1) e inserta el resultado en la pila. La resta se hace siguiendo las normas de IEEE754.

Antes	Después
float1	resultado
float2	...
...	

dsub

Opcode
0x67 (103) dsub ; Resta dos números `double`.

Toma dos números `double` de la pila, los resta (`double`2 – `double`1) e inserta el resultado. La resta se realiza siguiendo las normas de IEEE754.

Antes	Después
double1 word 1	resultado word 1
double1 word 2	resultado word 2
double2 word 1	...
double2 word 2	
...	

imul

Opcode
0x68 (104) imul ; Multiplica dos enteros.

Toma dos enteros de la pila, los multiplica e inserta el resultado en ella. Si hay desbordamiento, imul produce un resultado donde los bits de bajo orden son correctos, pero el bit de signo puede ser incorrecto.

Antes	Después
int	resultado
int	...
...	

lmul

Opcode
0x69 (105) lmul ; Multiplica dos números `long`.

Toma dos números `long` de la pila, los multiplica e inserta el resultado. Si hay desbordamiento, lmul produce un resultado donde los bits de bajo orden son correctos, pero el bit de signo puede no serlo.

Antes	Después
long1 word 1	resultado word 1
long1 word 2	resultado word 2
long2 word 1	...
long2 word 2	
...	

fmul

Opcode
0x6A (106) fmul ; Multiplica dos números `float`.

Toma dos números `float` de la pila, los multiplica e inserta el resultado. La operación sigue las reglas de la norma IEEE754 para aritmética de punto flotante.

Antes	Después
float	resultado
float	...
...	

dmul

Opcode
0x6B (107) dmul ; Multiplica dos números double.

Toma dos números `double` de la pila, los multiplica e inserta el resultado. La operación sigue las reglas de la norma IEEE754 para aritmética de punto flotante.

Antes	Después
double1 word 1	resultado word 1
double1 word 2	resultado word 2
double2 word 1	...
double2 word 2	
...	

idiv

Opcode
0x6C (108) idiv ; Divide dos enteros.

Toma dos números enteros de la pila, los divide (el segundo, `int2` entre el primero, `int1`); por ejemplo, `int2` / `int1`. El resultado obtenido es truncado al entero más cercano a cero y lo inserta en la pila.

Antes	Después
int1	resultado
int2	...
...	

ldiv

Opcode
0x6D (109) ldiv ; Divide dos números `long`.

Toma dos números `long` de la pila, los divide (`long`2 / `long`1) . El resultado es redondeado al entero más cercano a cero y puesto en la pila.

Antes	Después
long1 word 1	resultado word 1
long1 word 2	resultado word 2
long2 word 1	...
long2 word 2	
...	

fdiv

Opcode
0x6E (110) fdiv ; Divide dos números `float`.

Toma dos números `float` de la pila, los divide (`float`2 / `float`1) e inserta el resultado en la pila. La operación sigue las reglas de la norma IEEE754 para aritmética de punto flotante.

Antes	Después
float1	resultado
float2	...
...	

ddiv

Opcode
0x6F (111) ddiv ; Divide dos números `double`.

Toma dos números `double` de la pila, los divide (`double`2 / `double`1) e inserta el resultado en la pila. La división entre cero dará un resultado infinito. La operación sigue las reglas de la norma IEEE754 para aritmética de punto flotante.

Antes	Después
double1 word 1	resultado word 1
double1 word 2	resultado word 2
double2 word 1	...
double2 word 2	
...	

irem

Opcode
0x70 (112) irem ; Regresa el residuo del cociente de dos números enteros.

Toma dos números enteros de la pila, los divide (`int2` / `int1`), calcula el residuo del cociente y lo inserta en la pila. Esto es usado por el operador % en Java. El residuo (*remainder*) se puede calcular de la siguiente forma:

```
remainder = int2-(int2/int1)*int1
```

Antes	Después
int1	resultado
int2	...
...	

lrem

Opcode
0x71 (113) lrem ; Regresa el residuo del cociente de dos números `long`.

Toma dos números `long` de la pila, los divide (`long2` /`long1`), calcula el residuo y lo inserta en la pila. Esto es usado por el operador % en Java.

Antes	Después
long1 word 1	resultado word 1
long1 word 2	resultado word 2
long2 word 1	...
long2 word 2	
...	

frem

Opcode
0x72 (114) frem ; Regresa el residuo del cociente de dos números `float`.

Toma dos números `float` de la pila, los divide (`float2` /`float1`), calcula el residuo y lo inserta en la pila. El residuo (*remainder*) se calculo como:

```
remainder = float2-(int)(float2/float1)*float1
```

Antes	Después
float1	resultado
float2	...
...	

drem

Opcode
0x73 (115) drem ; Regresa el residuo del cociente de dos números `double`.

Toma dos números `double` de la pila, los divide (`double2` / `double1`), calcula el residuo y lo inserta en la pila. El residuo (*remainder*) se calculo como:

```
remainder = double2-(int)(double2/double1)*double1
```

Antes	Después
double1 word 1	resultado word 1
double1 word 2	resultado word 2
double2 word 1	...
double2 word 2	
...	

ineg

Opcode
0x74 (116) ineg ; multiplica por –1 un entero.

Toma el número entero de la pila y lo multiplica por –1, el resultado lo inserta en la pila.

Antes	Después
int	-int
...	

lneg

Opcode
0x75 (117) lneg ; multiplica por –1 un `long`.
Toma un número `long` de la pila y lo multiplica por –1, el resultado lo inserta en la pila. Esto es lo mismo que (~long)+1

Antes	Después
long1 word 1	resultado word 1
long1 word 2	resultado word 2
...	...

fneg

Opcode
0x76 (118) fneg ; multiplica por –1 un `float`.

Toma el número `float` de la pila y lo multiplica por –1, el resultado lo inserta en la pila. La operación sigue las reglas e la norma IEEE754 para aritmética de punto flotante, IEEE tiene dos ceros +0.0 y –0.0 **fneg** aplicado a +0.0 es –0.0.

Antes	Después
float	-float
...	...

dneg

Opcode
0x77 (119) dneg ; multiplica por –1 un `double`.

Toma el número `double` de la pila y lo multiplica por –1, el resultado lo inserta en la pila. La operación sigue las reglas e la norma IEEE754 para aritmética de punto flotante, IEEE tiene dos ceros +0.0 y –0.0 **dneg** aplicado a +0.0 es –0.0.

Antes	Después
double1 word 1	resultado word 1
double1 word 2	resultado word 2
...	...

ishl

Opcode
0x78 (120) ishl ; Realiza el corrimiento a la izquierda de un entero.

Toma dos enteros de la pila. Desplaza `int2` a la izquierda tantas veces como lo indiquen los primeros 5 bits de `int1`. El resultado es puesto en la pila. Esto es lo mismo que la expresión $x*2^s$, donde **x** es el `int2` y **s** es `int1`.

Antes	Después
int1	resultado
int2	...
...	

lshl

Opcode
0x79 (121) lshl ; Realiza el corrimiento a la izquierda de un `long`.

Toma un `long` y un entero de la pila. Desplaza `long` a la izquierda tantas veces como lo indiquen los primeros 6 bits de `int`. El resultado es un número `long` que es puesto en la pila. Esto es lo mismo que la expresión $x*2^s$, donde **x** es el `int2` y **s** es `int1`.

Antes	Después
int	resultado word 1
long word 1	resultado word 2
long word 2	...
...	

ishr

Opcode
0x7A (122) ishr ; Realiza el corrimiento a la derecha de un entero.

Toma dos enteros de la pila. Desplaza `int1` a la derecha tantas veces como lo indiquen los primeros 5 bits de `int2`. El resultado es puesto en la pila. Esto es lo mismo que la expresión $x/2^s$, donde **x** es `int2` y **s** es `int1`. `int1` es desplazado aritméticamente preservando el signo.

Antes	Después
int1	resultado
int2	...
...	

lshr

Opcode
0x7B (123) lshr ; Realiza el corrimiento a la derecha de un `long`.

Toma un `long` y un entero de la pila. Desplaza `long` a la derecha tantas veces como lo indiquen los primeros 6 bits de `int`. El resultado es un número `long` que es puesto en la pila. Esto es lo mismo que la expresión $x/2^s$, donde **x** es el `int2` y **s** es `int1`.

Antes	Después
int	resultado word 1
long word 1	resultado word 2
long word 2	...
...	

iushr

Opcode
0x7C (124) iushr ; Realiza el corrimiento lógico de un entero a la derecha.

Toma dos enteros de la pila. Desplaza `int2` a la derecha tantas veces como lo indican los primeros 5 bits de `int1`. El resultado es puesto en la pila. `int1` es desplazado lógicamente (ignora el bit de signo; útil para valores sin signo).

Antes	Después
int1	resultado
int2	...
...	

lushr

Opcode
0x7D (125) lshr ; Realiza el corrimiento lógico de un `long` a la derecha.

Toma dos números de la pila. Desplaza `long` a la derecha tantas veces como lo indican los primeros 6 bits de `int`. El resultado es puesto en la pila. `long` es desplazado lógicamente (ignora el bit de signo; útil para valores sin signo).

Antes	Después
int	resultado word 1
long word 1	resultado word 2
long word 2	...
...	

iand

Opcode
0x7E (126) iand ; Realiza la operación lógica and entre dos enteros.

Calcula la operación lógica and entre dos enteros, el resultado es puesto en la pila.

Antes	Después
int1	resultado
int2	...
...	

land

Opcode
0x7F (127) land ; Realiza la operación lógica and entre dos números `long`.

Calcula la operación lógica and entre dos números `long`, el resultado es puesto en la pila.

Antes	Después
long1 word 1	resultado word 1
long1 word 2	resultado word 2
long2 word 1	...
long2 word 2	
...	

ior

Opcode
0x80 (128) ior ; Realiza la operación lógica or entre dos enteros.

Calcula la operación lógica or entre dos enteros, el resultado es puesto en la pila.

Antes	Después
int1	resultado
int2	...
...	

lor

Opcode
0x81 (129) lor ; Realiza la operación lógica or entre dos números `long`.

Calcula la operación lógica or entre dos números `long`, el resultado es puesto en la pila.

Antes	Después
long1 word 1	resultado word 1
long1 word 2	resultado word 2
long2 word 1	...
long2 word 2	
...	

ixor

Opcode
0x82 (130) ixor ; Realiza la operación lógica xor entre dos enteros.

Calcula la operación lógica xor entre dos enteros, el resultado es puesto en la pila.

Antes	Después
int1	resultado
int2	...
...	

lxor

Opcode
0x83 (131) lxor ; Realiza la operación lógica xor entre dos números `long`.

Calcula la operación lógica **xor** entre dos números `long`, el resultado es puesto en la pila.

Antes	Después
long1 word 1	resultado word 1
long1 word 2	resultado word 2
long2 word 1	...
long2 word 2	
...	

iinc

Opcode
0x84 (132) iinc <*varnum*> <*n*> ; Incrementa la variable local entera.

Incrementa la variable local de tipo entera <*varnum*> en <*n*>. La instrucción iinc toma dos parámetros: <*varnum*> es un entero sin signo que indica que variable local será incrementada. <*n*> es un `int` que incrementa o decrementa a la variable. La instrucción se presenta en dos formas. En la primera, <*varnum*> es un entero sin signo entre el rango desde 0 hasta 0xFF. En la segunda (**wide**), <*varnum*> es un entero sin signo entre el rango desde 0 hasta 0xFFFF.

Antes	Después
...	...

i2l

Opcode
0x85 (133) i2l ; Convierte un entero a un `long`.

Toma un entero de la pila, lo convierte a un número `long` y lo inserta en la pila.

Antes	Después
int	long word 1
...	long word 2
	...

i2f

Opcode
0x86 (134) i2f ; Convierte un entero a un `float`.

Toma un entero de la pila, realiza el *cast* para convertir el número entero en un número `float` y lo inserta en la pila. En esta operación se puede perder precisión ya que un `float` tiene 24 bits significativos, comparado con los 32 bits que tiene un `int`, así que los bits menos significativos se pierden. De cualquier forma la magnitud del resultado se preserva. El redondeo se aplica utilizando la norma IEEE754.

Antes	Después
`int`	`float`
`...`	`...`

i2d

Opcode
0x87 (135) i2d ; Convierte un entero en un `double`.

Toma un entero de la pila, realiza el *cast* y lo convierte en un número `double`. La conversión es exacta ya que los números `double` tienen la misma precisión que los `int`.

Antes	Después
`int`	`double word 1`
`...`	`double word 2`
	`...`

l2i

Opcode
0x88 (136) l2i ; Convierte un `long` en un `int.`

Toma un número `long` de la pila, ignora los 32 bits más significantes y pone los 32 bits menos significantes en la pila como un entero. Esto puede ocasionar que la magnitud o el signo cambie de valor.

Antes	Después
`long word 1`	`int`
`long word 2`	`...`
`...`	

l2f

Opcode
0x89 (137) l2f ; Convierte un `long` en un `float`.

Toma un número `long` de la pila, realiza el cast, lo convierte a un número `float` y lo inserta en la pila. Esto puede causar pérdida de precisión.

Antes	Después
long word 1	float
long word 2	...
...	

l2d

Opcode
0x8A (138) l2d ; Convierte un `long` en un `double`.

Toma un número `long` de la pila, realiza el *cast*, lo convierte en un número `double` y lo inserta en la pila. Esto puede ocasionar pérdida de precisión. El redondeo se hace bajo la norma IEEE754.

Antes	Después
long word 1	double word 1
long word 2	double word 2
...	...

f2i

Opcode
0x8B (139) f2i ; Convierte un `float` en un `int`.

Toma un número `float` de la pila, realiza el cast a un entero de 32 bits y lo inserta en la pila. Ésta conversión se realiza de acuerdo con las especificaciones de IEEE754. La parte fraccionaria se pierde ya que se redondea a cero, así 3.14 se convierte en 3.

Antes	Después
int	float
...	...

f2l

Opcode
0x8C (140) f2l ; Convierte un `float` en un `long`.

Realiza el *cast* de un valor `float` a un número `long` de 64 bits. f2i quita el número `float` de la pila, lo convierte a un `long` y lo inserta en ella. La parte fraccionaria se pierde ya que se redondea a cero.

Antes	Después
float	long word 1
...	long word 2
	...

f2d

Opcode
0x8D (141) f2d ; Convierte un `float` en un `double`.

Toma el valor `float` de la pila, realiza el cast a un número `double` y lo inserta en ella. Ésta conversión se realiza de acuerdo a las especificaciones de la IEEE754, no hay pérdida de precisión.

Antes	Después
`float`	`double word 1`
`...`	`double word 2`
	`...`

d2i

Opcode
0x8E (142) d2i ; Convierte un `double` en un `int`.

Toma un número `double` de la pila, realiza el *cast* a un número `int` de 32 bits e inserta el resultado en ella. El redondeo se realiza bajo las especificaciones de IEEE754, la parte fraccionaria se pierde porque se redondea a cero.

Antes	Después
`float`	`double word 1`
`...`	`double word 2`
	`...`

d2l

Opcode
0x8F (143) d2l ; Convierte un `double` en un `long`.

Toma un número `double` de la pila, realiza el *cast* a un `long` de 64 bits e inserta el resultado en la pila. El redondeo se realiza bajo las especificaciones de IEEE754, la parte fraccionaria se pierde porque se redondea a cero.

Antes	Después
`double word 1`	`long word 1`
`double word 2`	`long word 2`
`...`	`...`

d2f

Opcode
0x90 (144) d2f ; Convierte un `double` en un `float`.

Toma un número `double` de la pila, realiza el cast a un `float` e inserta el resultado en la pila. El redondeo se realiza bajo las especificaciones de IEEE754 el signo del número se conserva.

Antes	Después
double word 1	float
double word 2	...
...	

i2b

Opcode
0x91 (145) i2b ; Convierte un `int` en un `byte`.

Convierte un entero en un `byte` con signo. Un entero de 32 bits es retirado de la pila, los 24 bits más significativos son descartados (puestos a cero), el resultado es insertado en la pila. Esta conversión puede causar cambio de signo.

Antes	Después
int	byte
...	...

i2c

Opcode
0x92 (146) i2c ; Convierte un entero en un carácter.

Toma un entero de 32 bits y lo convierte en un `char` de 16 bits sin signo. i2c es utilizado en Java cuando se realiza un *cast* explicito entre un `int` y un `char`, ya que i2c produce un `char` sin signo, cualquier signo en el valor original se pierde.

Antes	Después
int	char
...	...

i2s

Opcode
0x93 (147) i2s ; Convierte un `int` en un `short`.

Convierte un entero en un `short` con signo. Un `int` de 32 bits es tomado de la pila, los 16 bits más significativos son descartados y el resultado es puesto en la pila.

Antes	Después
int	short
...	...

lcmp

Opcode
0x94 (148) lcmp ; Compara dos números `long`.

Toma dos números `long` de la pila y los compara. Si son iguales, se inserta un cero en la pila. Si `long2` es más grande que `long1`, se inserta 1. Si `long1` es más grande que `long2`, se inserta –1.

Antes	Después
`long1 word 1`	`resultado`
`long1 word 2`	`...`
`long2 word 1`	
`long2 word 2`	
`...`	

fcmp<op>

Opcode
0x95 (149) fcmpl ; Compara dos números `float` (-1 si es **NaN**).
0x96 (150) fcmpg ; Compara dos números `float` (1 si es **NaN**).

Toma dos números `float` de la pila y los compara, utilizando reglas de IEEE754. Si los dos números son iguales, se inserta en la pila el valor cero. Si `float2` es más grande que `float1`, se inserta el valor 1. Si `float1` es más grande que `float2`, se inserta el valor -1. Si alguno de los números es **NaN**, se inserta el valor 1 (**fcmpl** inserta –1). +0.0 y –0.0 son tratados igual.

Antes	Después
`float1`	`resultado`
`float2`	`...`
`...`	

dcmp<op>

Opcode
0x97 (151) dcmpl ; Compara dos números `double` (-1 si es **NaN**).
0x98 (152) dcmpg ; Compara dos números `double` (1 si es **NaN**).

Toma dos números `double` de la pila y los compara, utilizando reglas de IEEE754. Si los dos números son iguales, se inserta en la pila el valor cero. Si `double2` es más grande que `double1`, se inserta el valor 1. Si `double1` es más grande que `double2`, se inserta el valor -1. Si alguno de los números es **NaN**, se inserta el valor 1 (**dcmpl** inserta –1). +0.0 y –0.0 son tratados igual.

Antes	Después
`double1 word 1`	`resultado`
`double1 word 2`	`...`
`double2 word 1`	
`double2 word 2`	
`...`	

if<cond>

Opcode

0x99 (153)	ifeq <*branchoffset*>	; Brinca si *value* es **igual** a cero.
0x9A (154)	ifne <*branchoffset*>	; Brinca si *value* es **diferente** a cero.
0x9B (155)	iflt <*branchoffset*>	; Brinca si *value* es **menor** a cero.
0x9C (156)	ifge <*branchoffset*>	; Brinca si *value* es **mayor o igual** a cero.
0x9D (157)	ifgt <*branchoffset*>	; Brinca si *value* es **mayor a cero** a cero.
0x9E (158)	ifle <*branchoffset*>	; Brinca si *value* es **menor o igual** a cero.

Toma un valor `int` de la pila, si la condición se cumple, realiza el salto a la dirección *pc* + *branchoffset*; donde *pc* es la dirección del *opcode* actual y *branchoffset* es un entero de 16 bits con signo que sigue al *opcode*. Si la condición no se cumple, la ejecución continua con el siguiente *opcode*.

Antes	Después
`value`	`...`
`...`	

if_icmp<cond>

Opcode

0x9F (159)	if_icmpeq <*branchoffset*>	; Brinca si *value2* y *value1* **son iguales**.
0xA0 (160)	if_icmpne <*branchoffset*>	; Brinca si *value2* y *value1* **no son iguales**.
0xA1 (161)	if_icmplt <*branchoffset*>	; Brinca si v*alue2* **es menor que** *value1*.
0xA2 (162)	if_icmpge <*branchoffset*>	; Brinca si *value2* **es más grande o igual que** *value1*.
0xA3 (163)	if_icmpgt <*branchoffset*>	; Brinca si *value2* **es más grande que** *value1*
0xA4 (164)	if_icmple <*branchoffset*>	; Brinca si value2 **es menor o igual que** *value1*.

Toma dos valores de tipo `int` de la pila y los compara. Si la condición se cumple, se realiza un salto a la dirección *pc* + *branchoffset*, donde *pc* es la dirección del *opcode* actual en el *bytecode* y *branchoffset* es un entero de 16 bits con signo que sigue al *opcode*. Si la condición no se cumple, se continua con el siguiente *opcode*.

Antes	Después
`value1`	`...`
`value2`	
`...`	

if_acmp<cond>

Opcode

0xA5 (165)	if_acmpeq <*branchoffset*>	; Brinca si dos referencias a objetos **son iguales**.
0xA6 (166)	if_acmpne <*branchoffset*>	; Brinca si dos referencias a objetos **no son iguales**.

Brinca si la condición se cumple. Toma dos referencias a objetos de la pila y los compara; Dos referencias son iguales si se refieren al mismo objeto. Si la condición se cumple, realiza un salto a la dirección *pc* + *branchoffset*, donde *pc* es la dirección del *opcode* actual y *branchoffset* es un entero de 16 bits con signo que sigue al *opcode*. Si la condición no se cumple, se continua con el siguiente *opcode*.

Antes	Después
`referencia1`	`...`
`referencia2`	
`...`	

goto

Opcode
0xA7 (167) goto <*branchoffset*> ; Realiza un salto.

Siempre que se presenta el *opcode* realiza un salto a la dirección *pc* + *branchoffset*, donde *pc* es la dirección del *opcode* actual y *branchoffset* es un entero de 16 bits con signo que sigue al *opcode*.

Antes	Después
`...`	`...`

jsr

Opcode
0xA8 (168) jsr ; Brinca a subrutina.

Llama a una subrutina local definida dentro del cuerpo del método. Es usado para implementar la cláusula `finally`. **jsr** primero inserta la dirección *pc* + 3 e la pila, donde *pc* es la dirección del *opcode* actual.

La dirección *pc* +3 es la dirección del *opcode* que sigue inmediatamente a la instrucción **jsr** –ésta es utilizada por la instrucción **ret** para regresar de la subrutina—. A continuación salta a la dirección *pc* + *branchoffset*, donde *pc* es la dirección del *opcode* actual y *branchoffset* es un entero de 16 bits con signo que sigue al *opcode*.

Antes	Después
`...`	`dirección`
	`...`

ret

Opcode
0xA9 (169) ret <*varnum*> ; Regresa de una subrutina.

Regresa de una subrutina que fue invocada por jsr o jsr_w. Toma el parámetro <*varnum*> que es un entero sin signo que tiene la dirección de regreso de la subrutina y es el que sigue al *opcode*. La ejecución continua en la dirección <*varnum*>.

Antes	Después
`...`	`...`

tableswitch

Opcode
0xAA (170) tableswitch <*default_offset*> <*0-3 bytes*> <*low*><*high*><*low-high+1*>

Ejecuta un salto calculado. La tabla de saltos que utiliza **tableswitch** está dada después del *opcode*. La instrucción toma un valor entero de la pila. Si *val* es menor que <*low*> o más grande que <*high*>, ejecuta un salto a la dirección *pc + default_offset*, donde *pc* es la dirección del *opcode* **tableswitch** y *default_offset* aparece después del o*pcode* y son 32 bits con signo. Si el valor esta en el rango <*low*> a <*high*>, ejecuta un salto a la entrada *i*, donde *i* es igual a *val* - <*low*> y la primera entrada en la tabla es cero.

tableswitch es una instrucción de longitud variable, inmediatamente después del *opcode* **tableswitch**, entre 0 a 3 bytes de ceros son insertados como relleno, así que el parámetro *default_offset* comienza con un *offset* en el *Opcode* el cual es múltiplo de 4. A continuación, le siguen dos enteros de 32 bits con signo –*low* y *high*-, seguidos por l*ow-hig+1* enteros de 32 bits.

Antes	Después
`val`	`...`
`...`	

lookupswitch

Opcode
0xAB (171) lookupswitch <*default_offset*><*0-3bytes*><*n*><*key1*><*offset1*>..<*keyn*><*offsetn*>

Ejecuta una comparación y brinca, generalmente se utiliza para una sentencia `switch`. La tabla utilizada por **lookupswitch** aparece después del *opcode*. La instrucción trabaja como sigue. Toma un valor `int` de la pila, **lookupswitch** busca en la tabla una entrada <*key*> que sea igual. Si la comparación es encontrada, brinca a la dirección correspondiente. Si no es encontrada, brinca a la dirección <*default_offset*>.

lookupswitch es una instrucción de longitud variable. Después del *opcode*, le siguen 0 a 3 bytes de relleno, así que el parámetro <*default_offset*> es un *offset* múltiplo de 4. A continuación le sigue un entero *n* de 32 bits indicando el número de pares *key/offset* de la tabla. Esto es seguido por *n* pares de `int`. Para cada par, el primer `int` es el valor <`key`> que es comparado y el segundo `int` es el <*offset*> a donde brinca la instrucción.

Antes	Después
`val`	`...`
`...`	

ireturn

Opcode
0xAC (172) ireturn ; Regresa de un método con un resultado `long`.

El método actual debe tener un tipo de retorno `boolean`, `byte`, `short`, `char` o `int`. La instrucción toma un valor de tipo `int` de la pila y lo inserta en el *stack frame* para que lo tome quien invocó al método (invokevirtual, invokespecial, invokestatic o invokeinterface).

Antes	Después
`int`	`...`
`...`	

lreturn

Opcode
0xAD (173) lreturn ; Regresa de un método con un resultado `long`.

El método actual debe tener un tipo de retorno `long`. La instrucción toma un valor de tipo `long` de la pila y lo inserta en el *stack frame* para que lo tome quien invocó al método (invokevirtual, invokespecial, invokestatic o invokeinterface).

Antes	Después
`long word 1`	`...`
`long word 2`	
`...`	

freturn

Opcode
0xAE (174) freturn ; Regresa de un método con un resultado `float`.

El método actual debe tener un tipo de retorno `float`. La instrucción toma un valor de tipo `float` de la pila y lo inserta en el *stack frame* para que lo tome quien invocó al método (invokevirtual, invokespecial, invokestatic o invokeinterface).

Antes	Después
`float`	`...`
`...`	

dreturn

Opcode
0xAF (175) dreturn ; Regresa de un método con un resultado `double`.

El método actual debe tener un tipo de retorno `double`. La instrucción toma un valor de tipo `double` de la pila y lo inserta en el *stack frame* para que lo tome quien invocó al método (invokevirtual, invokespecial, invokestatic o invokeinterface).

Antes	Después
`double word 1`	`...`
`double word 2`	
`...`	

areturn

Opcode
0xB0 (176) areturn ; Regresa de un método con una referencia a un objeto.

La referencia al objeto debe ser compatible con el tipo de retorno del método actual. areturn toma de la pila una referencia a un objeto y la pone en el *stack frame* para que la tome quien invocó al método (invokevirtual, invokespecial, invokestatic o invokeinterface).

Antes	Después
objeto	...
...	

return

Opcode
0xB1 (177) return ; Regresa de un método.

Regresa de un método que tenga como tipo de retorno `void`. Todos los elementos de la pila son descartados. Si el método está marcado como **synchronized**, entonces una instrucción **monitorexit** es ejecutada.

Antes	Después
...	...

getstatic

Opcode
0xB2 (178) getstatic <*index*> ; Obtiene el valor de una variable `static`.

El *opcode* **getstatic** es seguido por un <*index*> que es un entero de 16 bits sin signo. Éste, es un índice al *constant_pool* de la clase actual. La entrada está etiquetada como *CONSTANT_Fieldref*, estructura que contiene dos valores; el primero, es un índice a una estructura *CONSTANT_Class* que contiene el nombre de la clase a la que pertenece la variable y el segundo, es una estructura *CONSTANT_NameAndType* que tienen el nombre de la variable y su tipo (*descriptor*).

Antes	Después
...	valor
	...

Antes	Después
...	valor 1
	valor 2
	...

Para variables **static** `double` o `long`.

putstatic

Opcode
0xB3 (179) putstatic <*index*> ; Asigna un valor a una variable `static`.

El *opcode* **putstatic** es seguido por un <*index*> que es un entero de 16 bits sin signo. Éste, es un índice al *constant_pool* de la clase actual. La entrada etiquetada como *CONSTANT_Fieldref, que* contiene dos valores; el primero, es un índice a una estructura *CONSTANT_Class* que contiene el nombre de la clase a la que pertenece la variable y el segundo, es una estructura *CONSTANT_NameAndType* que tienen el nombre de la variable y su tipo (*descriptor*).

Antes	Después
valor	...
...	

Antes	Después
valor 1	...
valor 2	
...	

Para variables **static** `double` o `long`.

getfield

Opcode
0xB4 (180) getfield <*index*> ; Obtiene el valor de una variable de campo (no **static**).

La instrucción toma una referencia de una variable, obtiene su valor y lo inserta en la pila; el valor puede ser de una palabra o dos palabras (como un valor `long` o `double`). El *opcode* es seguido por un <*index*> que es un entero de 16 bits sin signo. Éste, es un índice al *constant_pool* de la clase actual. La entrada es una estructura *CONSTANT_Fieldref* que contiene dos valores; el primero, es un índice a una estructura *CONSTANT_Class* que contiene el nombre de la clase a la que pertenece la variable y el segundo, es una estructura *CONSTANT_NameAndType* que tienen el nombre de la variable y su tipo (*descriptor*).

Antes	Después
referencia	valor
...	...

Antes	Después
referencia	valor 1
	valor 2
...	...

Para variables `double` o `long`.

putfield

Opcode
0xB5 (181) putfield <*index*> ; Asigna un valor a una variable de campo (no **static**).

Asigna un valor a una variable de campo, éste valor puede ser de una o dos palabras de extensión. El *opcode* **putfield** es seguido por un <*index*> que es un entero de 16 bits sin signo y es un índice en el *constant_pool* de la clase actual. Éste, es un índice a la estructura *CONSTANT_Fieldref* que contiene dos valores; el primero, es un índice a una estructura *CONSTANT_Class* que contiene el

nombre de la clase a la que pertenece la variable y el segundo, es una estructura *CONSTANT_NameAndType* que tienen el nombre de la variable y su tipo (*descriptor*).

Antes	Después
valor	...
referencia	
...	

Antes	Después
Valor 1	...
Valor 2	
referencia	
...	

Para variables `double` o `long`

invokevirtual

Opcode
0xB6 (182) invokevirtual <*method*> ; Llama a un instancia de un método.

Es utilizado para invocar todos los métodos excepto métodos de interfaces, métodos `static` y algunos métodos especiales. El *opcode* **invokevirtual** es seguido por un entero de 16 bits sin signo, <*method*>, que es un índice en el *constant_pool* de la clase actual.

La entrada está etiquetada como *CONSTANT_Methodref* que posee dos campos. El primero, apunta a una estructura *CONSTANT_Class* que contiene el nombre de la clase del método. El segundo, apunta a una estructura *CONSTANT_NameAndType* que contiene el nombre y descriptor del método. La instrucción toma de la pila los argumentos y la referencia del objeto que llama al método.

Antes	Después
arg1	[resultado]
arg2	...
...	
argn	
referencia	
...	

invokespecial

Opcode
0xB7 (183) invokespecial <*method*> ; Llama a un método de la superclase, privado o constructor.

Esta instrucción se utiliza para llamar métodos de la superclase, métodos privados o constructores –también llamados <*init*>-. El *opcode* **invokespecial** es seguido por un entero de 16 bits sin signo, <*method*>, que es un índice en el *constant_pool* de la clase actual.

La entrada está etiquetada como *CONSTANT_Methodref* que posee dos campos. El primero, apunta a una estructura *CONSTANT_Class* que contiene el nombre de la clase del método. El segundo,

apunta a una estructura *CONSTANT_NameAndType* que contiene el nombre y descriptor del método. La instrucción toma de la pila los argumentos y la referencia del objeto que llama al método.

Antes	Después
arg1	[resultado]
arg2	...
...	
argn	
referencia	
...	

invokestatic

Opcode
0xB8 (184) invokestatic < *method* > ; Llama a un método `static`.

Llama a un método static, también conocidos como métodos de clase. El *opcode* invokestatic es seguido por un entero de 16 bits sin signo, *<method>*, que es un índice en el *constant_pool* de la clase actual.

La entrada está etiquetada como *CONSTANT_Methodref* que posee dos campos. El primero, apunta a una estructura *CONSTANT_Class* que contiene el nombre de la clase del método. El segundo, apunta a una estructura *CONSTANT_NameAndType* que contiene el nombre y descriptor del método. La instrucción toma de la pila los argumentos.

Antes	Después
arg1	[resultado]
arg2	...
...	
argn	
...	

invokeinterface

Opcode
0xB9 (185) invokeinterface <*method*> ; Llama a un método de una interfaz.

Llama a un método declarado dentro de una interfaz Java. El *opcode* invokeinterface es seguido por un entero de 16 bits sin signo, *<method>*, que es un índice en el *constant_pool* de la clase actual.

La entrada está etiquetada como *CONSTANT_Methodref* que posee dos campos. El primero, apunta a una estructura *CONSTANT_Class* que contiene el nombre de la clase del método. El segundo, apunta a una estructura *CONSTANT_NameAndType* que contiene el nombre y descriptor del método. La instrucción toma de la pila los argumentos y la referencia del objeto que llama al método.

Antes	Después
arg1	[resultado]
arg2	...
...	
argn	
referencia	
...	

new

Opcode
0xBB (187) new <*class*> ; Crea un nuevo objeto.

Después del *opcode* new, le sigue un entero de 16 bits sin signar <*class*>, que es un índice en el *constant_pool* de la clase actual. La entrada está etiquetada como *CONSTANT_Class* que contiene el nombre de la clase del nuevo objeto.

Antes	Después
...	objeto
	...

newarray

Opcode
0xBC (188) newarray <*type*> ; Crea un arreglo para números o booleans.

Asigna arreglos de una dimensión para `boolean`, `int, char, float, double, byte, short,` o `long`. El *opcode* newarray es seguido por un byte sin signo que identifica el tipo de dato del arreglo. <*type*> puede tener los siguientes valores:

type	valor
boolean	4
char	5
float	6
double	7
byte	8
short	9
int	10
long	11

anewarray

Opcode
0xBD (189) anewarray <type> ; Crea un arreglo de objetos.

La instrucción crea un arreglo de objetos. El *opcode* es seguido por un entero de 16 bits sin signar <*type*>, que es un índice en el *constant_pool*. El índice debe apuntar a una estructura *CONSTANT_Class* que contiene el tipo de dato del arreglo.

Antes	Después
tamaño	objeto
...	...

arraylength

Opcode
0xBE (190) arraylength ; Obtiene la longitud de un arreglo.

Obtiene la longitud de un arreglo, si el arreglo es multidimensional, el *opcode* obtiene la longitud de la primera dimensión.

Antes	Después
objeto	tamaño
...	...

athrow

Opcode
0xBF (191) athrow ; Lanza una excepción o error.

El *opcode* es utilizado para lanzar una excepción, athrow toma de la pila la referencia del objeto que lanza la excepción. El objeto es una instancia de `Throwable` o una de sus subclases.

Antes	Después
objeto	n/a
...	

checkcast

Opcode
0xC0 (192) checkcast <*type*> ; Se asegura que un objeto o arreglo pertenezca a un tipo de dato.

El *opcode* es seguido por un entero de 16 bits sin signo que indica una entrada en el *constant_pool* La entrada apunta a la estructura *CONSTANT_Class* que contiene el tipo de dato al que debe pertenecer el objeto.

Antes	Después
objeto	tamaño
...	...

instanceof

Opcode
0xC1 (193) instanceof <*type*> ; Prueba si un objeto pertenece a algún tipo de dato.

La instrucción prueba que un objeto pertenezca a un tipo de dato. El *opcode* es seguido por un entero de 16 bits sin signo, <*type*>, que indica el tipo de dato al que debe pertenecer el objeto. Si se cumple la igualdad se inserta en la pila el entero 1, si no se cumple se inserta un cero. Si el objeto es `null`, siempre se inserta un cero.

Antes	Después
objeto	resultado
...	...

monitorenter

Opcode
0xC2 (194) monitorenter ; Entra a una región de código sincronizado.

Coordina el acceso a un objeto con múltiples *hilos* cuando es utilizado por un enunciado `synchronized` en Java.

Antes	Después
objeto	...
...	

monitorexit

Opcode
0xC3 (195) monitorenter ; Abandona una región de código sincronizado.

Libera el bloqueo que se produce tras una entrada a una región de código sincronizado.

Antes	Después
objeto	...
...	

wide

Opcode
0xC4 (196) wide <*instruction*> ; La siguiente instrucción usará un índice de 16 bits.

Aparece cuando el *constant_pool* es muy grande. **wide** es utilizado en conjunto con las instrucciones aload, dload, iload, fload, lload, astore, dsotore, istore, fstore, lstore, iinc y ret e incrementa su índice a 16 bits.

Antes	Después
...	...

multianewarray

Opcode
0xC5 (197) multianewarray <*type*><*n*> ; Crea arreglos multidimensionales.

El *opcode* crea arreglos multidimensionales. La instrucción está seguida por un entero de 16 bits sin signo que indica una entrada al *constant_pool*. El índice apunta a una estructura *CONSTANT_Class* que contiene el tipo de dato del arreglo. Después del índice, le sigue un byte sin signo ,<*n*>, que indica el número de dimensiones.

Antes	Después
tamañon	referencia
...	...
tamaño2	
tamaño1	
...	

ifnull

Opcode
0xC6 (198) ifnull <*offset*> ; Brinca si es nulo.

Toma la referencia a un objeto de la pila y prueba si es igual a `null`; si la condición se cumple, salta a la dirección *pc* + <*offset*>, donde *pc* es la dirección de la instrucción actual y <*offset*> es un entero de 16 bits con signo que sigue a la instrucción. Si la referencia no es `null`, la ejecución continua en el siguiente *opcode*.

Antes	Después
objeto	...
...	

ifnonnull

Opcode
0xC7 (199) ifnonnull <*offset*> ; Brinca si no es nulo.

Toma la referencia de un objeto de la pila y prueba si no es igual a `null`; si la condición se cumple, salta a la dirección *pc* + <*offset*>, donde *pc* es la dirección de la instrucción actual y <*offset*> es un entero de 16 bits con signo que sigue a la instrucción. Si la referencia no es `null`, la ejecución continua en el siguiente *opcode*.

Antes	Después
objeto	...
...	

goto_w

Opcode
0xC8 (200) goto_w <*offset*> ; Brinca a una dirección utilizando un *offset* extendido.

goto_w es idéntico a goto excepto que su <*offset*> es de 32 bits con signo.

Antes	Después
...	...

jsr_w

Opcode
0xC9 (201) jsr_w <*offset*> ; Brinca a una subrutina utilizando un *offset* extendido.

jsr_w es idéntico a jsr excepto que su <*offset*> es de 32 bits con signo.

Antes	Después
...	dirección
	...

breakpoint

Opcode
0xCA (202) RESERVADO

impdep1

Opcode
0xFE (254) RESERVADO

impdep2

Opcode
0xFF (255) RESERVADO

Notas

- Las direcciones son medidas en bytes desde el comienzo del *bytecode* (el byte 0 es la primera dirección en el *bytecode*).
- Existen algunos *opcodes* en formato rápido (*quick*) pero no son tratados en este apéndice.
- Si necesita una descripción mucho más detallada de cada uno de los *opcodes*, puede revisar *Java Virtual Machine*, Meyer Jon and Downing Troy, ed O´reilly.