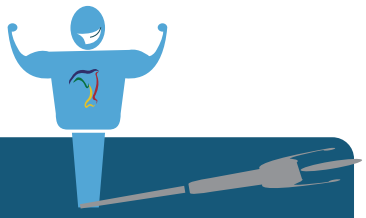
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

# ¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

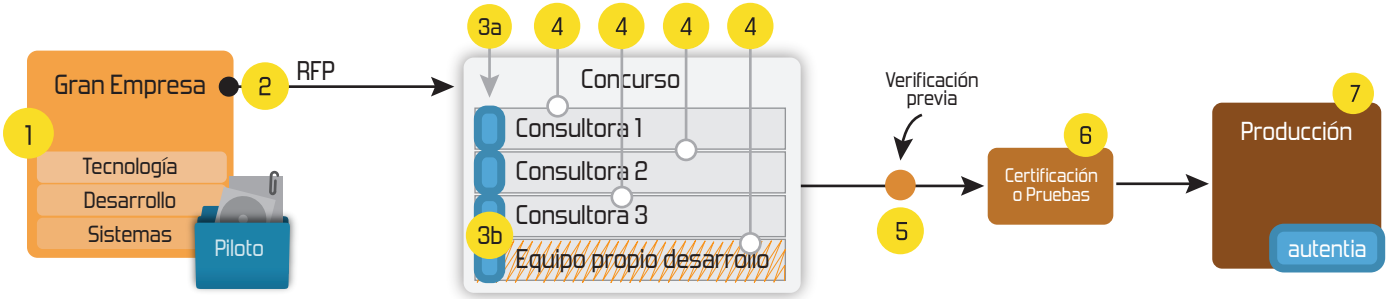
## 1. Desarrollo de componentes y proyectos a medida



## 2. Auditoría de código y recomendaciones de mejora

## 3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.




## 4. Cursos de formación (impartidos por desarrolladores en activo)

- Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery
- Gestor portales (Liferay)
- Gestor de contenidos (Alfresco)
- Aplicaciones híbridas
- Control de autenticación y acceso (Spring Security)
- UDDI
- Web Services
- Rest Services
- Social SSO
- SSO (Cas)
- JPA-Hibernate, MyBatis
- Motor de búsqueda empresarial (Solr)
- ETL (Talend)
- Dirección de Proyectos Informáticos.
- Metodologías ágiles
- Patrones de diseño
- TDD
- Tareas programadas (Quartz)
- Gestor documental (Alfresco)
- Inversión de control (Spring)
- BPM (jBPM o Bonita)
- Generación de informes (JasperReport)
- ESB (Open ESB)

Compartimos nuestro conocimiento en:
**www.adictosaltrabajo.com**

Para más información visítenos en:
**www.autentia.com**

Alberto Moratilla Ocaña

Consultor tecnológico de desarrollo de proyectos informáticos.

Ingeniero en Informática, especialidad en Ingeniería del Software. DEA en doctorado en Información, Documentación y Conocimiento. MBA en empresas de Nuevas Tecnologías. CQF.

Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación

Somos expertos en Java/J2EE

Ver todos los tutoriales del autor



**Fecha de publicación del tutorial: 2015-04-16** Tutorial visitado 98 veces Descargar en PDF

## 0. Índice de contenidos.



## 1. Introducción

Una de las características más relevante del lenguaje de programación Java es el hecho de que utiliza una máquina virtual para la ejecución de los programas, haciendo ésta de intermediaria entre la máquina real (HW + SO) y los programas desarrollados por los programadores.

Esto supuso una gran ventaja a la hora de poder distribuir un mismo código ya compilado en diferentes tipos de máquinas físicas con diferentes sistemas operativos. Basta con compilar el código fuente de alto nivel a código intermedio (que es código máquina para la JVM), también llamado Bytecode, para que pudiera ser empleado en máquinas virtuales que corren sobre máquinas físicas.

En este tutorial vamos a ver una introducción al ByteCode de Java. Para ello veremos brevemente la arquitectura clásica de una máquina virtual de Java, y también veremos cómo inspeccionar el código generado (ByteCode) para algunos ejemplos sencillos, entendiendo lo que hacen cada una de las instrucciones de bajo nivel.

## 2. Entorno

Para realizar este tutorial se ha empleado el siguiente entorno de desarrollo:

- Hardware: Mac Book Pro 15" Intel Core i7 2,8 GHz, 16 GB RAM.
- Sistema Operativo: Mac OS X Yosemite.
- Netbeans 8.0.1.
- Java(TM) SE Runtime Environment (build 1.8.0_31-b13)

## 3. Código Máquina y Java ByteCode

Si tomamos como referencias otros lenguajes clásicos, existentes en el momento de la creación de Java, como por ejemplo C o Pascal, podremos ver que su código se compila a código máquina. Esta característica lo hace dependiente de la arquitectura sobre la que se va a ejecutar, tanto por hardware (está diseñador para ser ejecutado por unos chips adecuados), como por el kernel del sistema operativo (ya que utilizará seguramente librerías y facilidades provista por éste).

Si tienes formación en computadores o eres un amante del "bajo nivel", sabrás que ese código máquina es una secuencia de órdenes que utilizan las partes físicas del procesador directamente: registros, punteros de memoria, operaciones básicas del conjunto de instrucciones del procesador. Es el conocido (y complicado) lenguaje ensamblador, donde podemos especificar unas instrucciones básicas de bajo nivel que se convierten directamente a código binario, que son operaciones que realiza directamente la CPU del ordenador. Si eres de la vieja escuela seguro que has hecho alguna rutina a bajo nivel para manipular el teclado o la tarjeta gráfica.

En Java, sin embargo, se sigue una arquitectura diferente. Se pasa de un modelo en el que el código de alto nivel es compilado a código máquina directamente, a un modelo en el que el código Java de alto nivel es compilado a código intermedio, que es interpretado por una máquina virtual (JVM) que encapsula el funcionamiento de la máquina física real subyacente

El código binario de Java para esta máquina virtual recibe el nombre de Bytecode. Este código binario de Java es transformado a código binario de la máquina real por la máquina virtua a través de su compilador JIT (Just in Time) en el momento de la ejecución. Por tanto, si tenemos un programa Java ya compilado, podrá ser ejecutado en una máquina virtual, con independencia de la máquina real (Hardware + SO) en la que está instalada esa máquina virtual.

Se relega entonces el problema no a disponer de una versión compilada de nuestro programa para cada plataforma destino (como sucede en C o Pascal por ejemplo), sino a tener una *sola versión compilada de nuestro programa Java en ByteCode* y

disponer de una máquina virtual para cada plataforma (principio WORA: Write Once Run Anywhere -escribe una vez, ejecuta en cualquier lugar-). Como podrás suponer *las máquina virtuales son creadas por terceros*, asi que en realidad nos están quitando trabajo (tedioso) que realizar: no sólo compilar para cada plataforma, sino asegurarnos que nuestro código respeta las peculiaridades de la plataforma destino. Además podemos beneficiarnos de mejoras en las máquinas virtuales fácilmente.

Por tanto Java = ***Java API + JVM***

**¿Por qué es importante conocer la transformación a ByteCode?**

Debemos tener muy claro que el ordenador siempre va a ejecutar el código máquina, porque es lo que entiende el procesador o los otros chips que van a realizar el trabajo de verdad, y que existe un compilador o intérprente que transforma nuestro lenguaje favorito a código máquina.

¿Qué quiere decir esto? Pues que por muy bien que escribamos nuestro código de alto nivel (en este caso Java), estamos en manos de lo que el compilador o intérprete le envíe al procesador. Por tanto las discusiones de si es más óptima una forma de hacer un bucle u otra se tienen que dirimir en esta fase: *¿qué se le está enviando al procesador realmente?*. Es por ello que las sucesivas mejoras en los compiladores permiten la transformación en un código máquina más óptimo (menos gasto de memoria, mejor reutilización de ésta, menor tamaño, más velicidad, etcétera.).

Al final es importante ser conscientes de que nuestros programas son transformados a instrucciones el procesador, y debemos, si el caso lo requiere, intentar cambiar nuestro código de alto nivel para ayudar al compilador a pasarle a la CPU un código mejor para ella.

## 4. Arquitectura básica de la JVM

*(Nota del autor: la arquitectura de la JVM cambia ente versiones. La aquí descrita se refiere a la versión 7 y anteriores. En la JVM8 hay cambios en la organización de la memoria. No obstante, nos servirá para comprender los mecanismos fundamentales).*

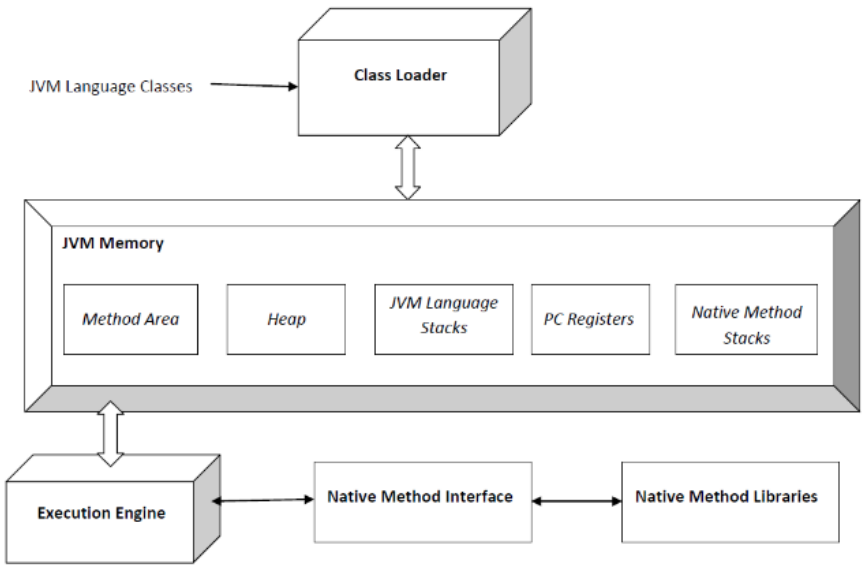
Antes de comenzar a analizar el código máquina de la JVM, es necesario conocer algunos conceptos de la arquitectura que utiliza internamente. Recuerda que, aunque se trata de una máquina virtual, no deja de ser una máquina en la que se procesan las instrucciones, de forma bastante simular a cualquier otro sistema físico existente. Por tanto habrá espacios de memoria, una pila contador, de instrucciones, etcétera. Puedes encontrar una referencia de la especificación de la máquina virtual de java versión SE 8 en este enlace: https://docs.oracle.com/javase/specs/jvms/se8/html/ y para anteriores en https://docs.oracle.com/javase/specs/index.html.

Los principios en los que se basa la máquina virtual de Java podrían serian los siguientes (link):

- *Basado en Stack (pila)*: al contrario que el modelo utilizado por los x86 o ARM, que es en base a registro. No obstante, existen diferentes tipos de JVM, y por ejemplo, la JVM de Google - Dalvik - está basada en registros, ya que permite disponer de un código más reducido, a cambio de ser más dependiente del hardware subyacente.
- *Referencias simbólicas* para las clases y las interfaces (no así a los tipos primitivos -int, double, boolean...-) en vez de referencias a posiciones físicas de memoria que lo harían dependiente de la plataforma.
- *Uso de un recolector de basura o Garbage Collector*, que elimina las instancias que no son utilizadas, en vez de ser necesario ocuparse de eliminarlas explícitamente.
- *Independencia de la plataforma subyacente* (HW+SO) al definir los tipos primitivos de una forma clara y única.
- *Network byte order*, en las clases, siendo así independiente del tipo little endian de los x86 o big endian de los RISC.

La máquina virtual de Java dispone de un motor de ejecución que comienza con el hilo principal ejecutando el método *main* que como sabemos es el obligatorio que debe tener toda aplicación Java (public void main). Para ello tiene que pasar por el ClassLoader, que se encarga de cargar las clases que se van a utilizar.

Cada aplicación Java se ejecuta en una máquina virtual propia y puede disponer de varios hilos: desde hilos propios para su funcionamiento como el que lanza el garbage collector (hilo daemon) o realizan tareas periódicas como control de eventos, a otros hilos lanzados por el programador (hilos non-daemon) desde la llamada al método *run*




"JvmSpec7" by Michelle Ridomi - Own work. Licensed under CC BY-SA 4.0 via Wikimedia Commons.

A partir del comienzo de la ejecución por el hilo principal de la aplicación, comienza a usar las estructuras de subsistemas, zonas de memorias e instrucciones de las que dispone. Vamos a ver, sin entrar en profunfidad, cómo funciona para comprender las operaciones del código máquina que veremos en los ejemplos.

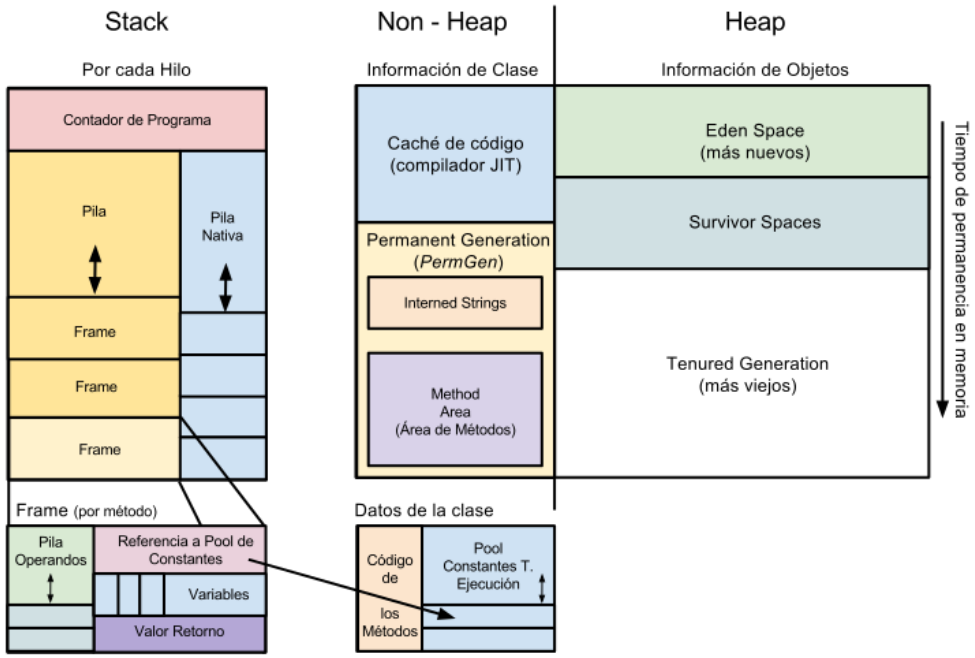
La memoria de la máquina virtual se podría dividir en dos grandes partes: las estructuras que existen para cada hilo y las estructuras compartidas para todos los hilos. Vamos a verlas en más detalle.

**Últimos Tutoriales del Autor**

» Imprimiendo documentos Office y PDF existentes con Java en entorno Windows. Batch & Print

» Experimenta con tu código en Eclipse utilizando Scrapbooks

» Introducción a Polymer




Basado en el esquema de: http://blog.jamesdbloom.com/JVMInternals.html

**Estructura por cada hilo**

- **Stack (Pila)**: que es el entorno que se crea por cada uno de los hilos que corren en la máquina. Podemos encontrar en él:
  - **Contador de programa**: que es el típico puntero para llevar el control de la ejecución del código. Esto es, indica qué instrucción se está ejecutando.
  - **Pila de frames (Stack)**: una estructura de tipo LIFO para poder almacenar (apilar mejor dicho) los *Frames* que surgen en la ejecución del código por parte del hilo. La pila nativa (Native Stack) es similar pero para llamadas nativas (JNI).
  - **Frames**: cada vez que el hilo ejecuta un método se crea un frame nuevo, y cuando se sale se elimina de la pila. Por esta razón no se comparte información entre hilos dentro de un método: porque esta información está en el frame que está en el stack por cada hilo. Contiene a su vez:
    - *Valor de retorno* del método.
    - *Array de Variables locales* para el método y el hilo. Pueden ser o bien tipos primitivos o bien referencias a instancias de clases.
    - *Pila de operandos*: donde se empujan (push) valores, y se realizan operaciones, cogiendo (pop) el resultado. Esta parte es la que centrará nuestra atención a la hora de explicar las operaciones en ByteCode
    - *Referencia a las constantes en tiempo de ejecución* de la clase en la que está el método. Veremos que es fundamental para hacer la invocación a la ejecución de métodos, constantes de la clase, etcétera.

**Estructura común para todos los hilos**

- **Non Heap (no montón)**: donde se almacena el código fuente de la aplicación y las constantes en tiempo de ejecución (incluidos los Strings). Es donde se almacena la información relativa a las clases. Se compone de:
  - *Code Cache* (caché de código): que es el lugar en el que se compilan y almacenan los métodos compilados a código nativo de la máquina por el compilador JIT (Just in time) para que pueda ser entendido por el procesador. Se puede encontrar una buena referencia aquí
  - *Permanent Generation* (el famoso PermGen): que contiene el área de métodos (código de las clases, tanto campos como métodos y su código incluyendo constantes) y las cadenas de texto utilizadas en la aplicación. A partir de JVM8 este espacio ha sido cambiado.
- **Heap (montón):** el resto de memoria donde se almacenan los objetos y sus valores (es decir, las instancias reales de las clases). Es donde actúa el Garbage Collector de la JVM, eliminando las referencias a los datos que no se utilizan, y clasificando en diferentes áreas los datos según su tiempo de supervivencia en el sistema.

  De poco tiempo a permanante: *Young Generation* (Eden space, Survivor Spaces) que son las instancias recién creadas o que tienen poco tiempo de supervivencia; *Tenured (Old) Generation*, que han aguantado más tiempo porque son referenciadas en más lugares y más frecuentemente. Este modelo puede cambiar dependiendo de la implementación y de la versión de la máquina virtual empleada.

Una vez que hemos visto la estructura general de la JVM, vamos a ver e interpretar algunos ejemplos de código en Java.

## 5. Inspeccionando el Bytecode de Java

Para ver el código ByteCode de una clase de Java, tenemos dos modos:

- A través de un editor que tegan vista Hexadecimal para ver el código binario e intentar ver las equivalencias con el juego de instrucciones de la JVM.
- La otra opción es mucho más sencilla para los humanos: usando la utilidad "javap" que muestra las instrucciones equivalentes de ese bytecode. Así no tenemos que hacer el esfuerzo de convertir el Hexadecimal a las instrucciones equivalentes de JVM.

**Primer Ejemplo: clase con un atributo**

Vamos a verlo con un ejemplo sencillo: creamos una clase llamada BCtest con el siguiente contenido:

```
1  package com.adictosAlTrabajo.bc;
2
3  public class BCtest {
4      int valor = 42;
5  }
```

Es decir, una clase que tiene un campo de tipo int (primitivo) con el valor 42.

Para ver el bytecode vamos al lugar donde está ubicada la clase compilada (el .class) ejecutamos la siguiente instrucción desde nuestra shell (tenemos que tener java/bin en el path) para que decompile la clase y nos muestre el código:

```
javap -c BCTest.class
```

El resultado es el siguiente:

```
Compiled from "BCTest.java"
public class com.adictosAlTrabajo.bc.BCTest
{
    int valor;

    public com.adictosAlTrabajo.bc.BCTest();
    Code:
    0: aload_0
    1: invokespecial #1                  // Method java/lang/Object."<init>":()V
    4: aload_0
    5: bipush        42
    7: putfield      #2                  // Field valor:I
    10: return
}
```

En primer lugar vemos la declaración de la clase y del campo que contiene la misma, *int valor*. Posteriormente, se pasa a definir el constructor de la clase, cuyo código es el que vamos a analizar (más que nada porque no hay otra cosa):

- La primera instrucción, aload_0, indica que está guardando en el array de variables locales el valor que está en la posición 0 de la tabla de variables locales a la pila de operandos. Este valor es la referencia a la propia instancia, es decir *this*. Veremos que hay operaciones de tipo load, indicando la primera letra el timpo de datos, como por ejemplo iload para enteros o aload para objetos.
- La segunda instrucción, invokespecial #1, representa una llamada aun método de inicialización o método privado o de una superclase. En este caso es el constructor de la superclase de nuestro objeto, que es *Object*. El #1 representa la posición del pool de consantes.
- Se vuelve a realizar la llamada para definir *this* después de la llamada del constructor.
- La cuarta instrucción, bipush, indica que se carga en la pila de operandos el valor 42. El prefijo "bi" indica que es un entero expresado en byte. Ahora que está en la pila de operados es posible que podamos hacer algo con ello.
- El siguiente paso, putfield #2 lo que hace es asignar el valor existentes en la cabeza de la pila de operandos al campo #2 del pool de constantes de la clase, que en este caso se refiere al atributo de clase "valor".
- Finalmente se termina la ejecución con un return, que devuelve el valor que esté en la cima de la pila, que en este caso será la referencia a <>this</>

Por tanto tenemos un par de operaciones básicas en este ejemplo (puedes consultar todas aquí para la JVM8):

- *xload*: que se encarga de cargar un valor del array de variables locales a la cabeza de la pila de operados. Con el prefijo x se indica el tipo de datos con el que se opera.
- *xpush*: que carga valores inmediatos (los que no se leen de variables) a la pila de operandos. Con el prefijo x se indica el tipo de datos con el que se opera.
- *invokespecial*: para la llamada de métodos privados, constructor, o de métodos de la superclase de la clase. En nuestro caso es el constructor.

Los números que aparecen en la parte izquierda antecediendo a cada operación y que no son consecutivos, indican el lugar en el que comienzan las instrucciones. Y efectivamente, no van consecutivas porque algunas instrucciones ocupan más bytes que otras. En nuestro ejemplo:

- 1 byte: aload_0
- 2 bytes: bipush 42
- 3 bytes: invokespecial #1; putfield #2

| aload_0 | invokespecial #1 | aload_0 | bipush 42 |
|---|---|---|---|
| 1 | 3 | 1 | 2 |

Por ello el primer aload_0 comienzan en el 0 (primera posición), y el segundo comienza en la 4 (1 byte del primer aload y 3 bytes del invokespecial).

En resumen, lo que hace nuestro código de ejemplo es:

1. se carga el puntero a this en el array de variables locales;
2. se llama al método que está en la posición 1 del pool de constantes (referencias) de la clase;
3. posteriormente se carga en la pila el valor 42 y se asigna al campo que está indicando en la posición 2 del pool de constantes.

**Conversión de ByteCode a Código Binario**

Como ya hemos dicho, el Bytecode es código binario interpretado por la máquina virtual de Java. Es decir, que como en un lenguaje ensamblador cualquiera, existe una relación directa entre las instrucciones máquina -virtual- que hemos visto antes y el código binario que generan.

Si tomas el fichero .class de este ejemplo y haces un dump con un editor hexadecimal, verás algo como lo siguiente:

```
0000: CA FE BA BE 00 00 00 34 00 14 0A 00 04 00 10 09   .......4........
0010: 00 03 00 11 07 00 12 07 00 13 01 00 05 76 61 6C   .............val
0020: 6F 72 01 00 01 49 01 00 06 3C 69 6E 69 74 3E 01   or...I...<init>.
0030: 00 03 28 29 56 01 00 04 43 6F 64 65 01 00 0F 4C   ..()V...Code...L
0040: 69 6E 65 4E 75 6D 62 65 72 54 61 62 6C 65 01 00   ineNumberTable..
0050: 12 4C 6F 63 61 6C 56 61 72 69 61 62 6C 65 54 61   .LocalVariableTa
0060: 62 6C 65 01 00 04 74 68 69 73 01 00 20 4C 63 6F   ble...this.. Lco
0070: 6D 2F 61 64 69 63 74 6F 73 41 6C 54 72 61 62 61   m/adictosAlTraba
0080: 6A 6F 2F 62 63 2F 42 43 54 65 73 74 3B 01 00 0A   jo/bc/BCTest;...
0090: 53 6F 75 72 63 65 46 69 6C 65 01 00 0B 42 43 54   SourceFile...BCT
00A0: 65 73 74 2E 6A 61 76 61 0C 00 07 00 08 0C 00 05   est.java........
00B0: 00 06 01 00 1E 63 6F 6D 2F 61 64 69 63 74 6F 73   .....com/adictos
00C0: 41 6C 54 72 61 62 61 6A 6F 2F 62 63 2F 42 43 54   AlTrabajo/bc/BCT
00D0: 65 73 74 01 00 10 6A 61 76 61 2F 6C 61 6E 67 2F   est...java/lang/
00E0: 4F 62 6A 65 63 74 00 21 00 03 00 04 00 00 00 01   Object.!........
00F0: 00 00 00 05 00 06 00 00 00 01 00 01 00 07 00 08   ................

0100: 00 01 00 09 00 00 00 39 00 02 00 01 00 00 00 0B   .......9........
0110: 2A B7 00 01 2A 10 2A B5 00 02 B1 00 00 00 02 00   *...*.*.........
0120: 0A 00 00 00 0A 00 02 00 00 00 03 00 04 00 04 00   ................
0130: 0B 00 00 00 0C 00 01 00 00 00 0B 00 0C 00 0D 00   ................
0140: 00 00 01 00 0E 00 00 00 02 00 0F                  ...........
```

Si te fijas, he marcado la parte del fichero en binario que corresponde a las instrucciones que hemos visto anteriormente. Puedes ver la equivalencia debajo de cada instrucción, en el apartado "Forms" en el siguiente enlace: https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html#jvms-6.5.invokespecial o aquí si quieres ver todas resumidas:http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings. No sé si será legal hacerlo, pero puedes editar un .class con un editor hexadecimal para cambiar su ByteCode... Quizá alguna vez te saque de un apuro si no tienes los ficheros .java de fuente :). Vamos instrucción por instrucción:

| Instrucción ByteCode | Equiv. Bytes |
|---|---|
| aload_0 | 2A |
| invokespecial #1 | B7 00 01 |
| aload_0 | 2A |
| bipush 42 | 10 2A |
| putfield #2 | B5 00 02 |
| return | B1 |

Ya puedes imaginar cómo cambiar un aload_0 por un aload_1 o cambiar la llamada al constructor del padre. Aunque claro, las consecuencias pueden ser "curiosas".

**Consulta a la Tabla de Constantes (Constant Pool) y el Array de Variables Locales**

Has podido ver que algunas instrucciones, como invokespecial y putfield hacen referencia a la tabla de constantes (constant pool), que contiene las referencias en tiempo de ejecución a las variables, clases, nombres, etcétera. Por tanto tenemos que saber qué es #1 o #2 realmente.

También en la primera instrucción, aload_0, hace referencia a la posición 0 del Array de Variables Locales, que contiene la referencia a *this*

Si queremos ver la tabla de constantes y el array de variables, tendremos que ejecutar con la opción -v (verbose) de javap, que nos dará más detalles:

```
javap -c -v BCTest.class
```

El resultado del comando es una información mucho más detallada:

```
Classfile /Users/amoratilla/NetBeansProjects/bc/build/classes/com/adictosAlTrabajo/bc/
Last modified 06-abr-2015; size 331 bytes
MD5 checksum f453b473706e2e9cb8a9cc5e0d87be6b
Compiled from "BCTest.java"
public class com.adictosAlTrabajo.bc.BCTest
minor version: 0
major version: 52
flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
#1 = Methodref          #4.#16         // java/lang/Object."<init>":()V
#2 = Fieldref           #3.#17         // com/adictosAlTrabajo/bc/BCTest.valor:I
#3 = Class              #18            // com/adictosAlTrabajo/bc/BCTest
#4 = Class              #19            // java/lang/Object
#5 = Utf8               valor
#6 = Utf8               I
#7 = Utf8               <init>
#8 = Utf8               ()V
#9 = Utf8               Code
#10 = Utf8               LineNumberTable
#11 = Utf8               LocalVariableTable
#12 = Utf8               this
#13 = Utf8               Lcom/adictosAlTrabajo/bc/BCTest;
#14 = Utf8               SourceFile
#15 = Utf8               BCTest.java
#16 = NameAndType        #7:#8          // "<init>":()V
#17 = NameAndType        #5:#6          // valor:I
#18 = Utf8               com/adictosAlTrabajo/bc/BCTest
#19 = Utf8               java/lang/Object
{
    int valor;
    descriptor: I
    flags:

    public com.adictosAlTrabajo.bc.BCTest();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
    stack=2, locals=1, args_size=1
    0: aload_0
    1: invokespecial #1                  // Method java/lang/Object."<init>":()V
    4: aload_0
    5: bipush        42
    7: putfield      #2                  // Field valor:I
    10: return
    LineNumberTable:
    line 3: 0
    line 4: 4
    LocalVariableTable:
    Start  Length  Slot  Name   Signature
    0      11     0  this   Lcom/adictosAlTrabajo/bc/BCTest;
}
SourceFile: "BCTest.java"
```

Podemos ver cómo, al comienzo, debajo de *Constant pool*, tenemos el listado de constantes. En la posición #1 tenemos la referencia al método constructor por defecto de la clase y en la posición #2 tenemos la referencia al campo valor.

```
    Constant pool:
#1 = Methodref          #4.#16         // java/lang/Object."<init>":()V
#2 = Fieldref           #3.#17         // com/adictosAlTrabajo/bc/BCTest.valor:I
#3 = Class              #18            // com/adictosAlTrabajo/bc/BCTest
#4 = Class              #19            // java/lang/Object
#5 = Utf8               valor
#6 = Utf8               I
#7 = Utf8               <init>
#8 = Utf8               ()V
#9 = Utf8               Code
#10 = Utf8               LineNumberTable
#11 = Utf8               LocalVariableTable
#12 = Utf8               this
#13 = Utf8               Lcom/adictosAlTrabajo/bc/BCTest;
#14 = Utf8               SourceFile
#15 = Utf8               BCTest.java
#16 = NameAndType        #7:#8          // "<init>":()V
#17 = NameAndType        #5:#6          // valor:I
#18 = Utf8               com/adictosAlTrabajo/bc/BCTest
#19 = Utf8               java/lang/Object
```

Del mismo modo, al final del todo tenemos la *LocalVariableTable*. En este ejemplo sencillo tenemos un array de variables con una sola posición, que es la referencia al propio objeto de la clase, *this*. Si lo piensas, el tributo de clase *valor* no es una variable, sino una referencia a un atributo de la clase, que como hemos visto va ubicado en el pool de constantes.

```
LocalVariableTable:
Start  Length  Slot  Name   Signature
0      11      0  this   Lcom/adictosAlTrabajo/bc/BCTest;
```

**Segundo ejemplo: con un método de clase**

Vamos con otro ejemplo algo más completo. Tenemos la clase:

```
package com.adictosAlTrabajo.bc;

public class BCTest {
    int valor = 42;

    public int suma(int a, int b){
        return a+b;
    }
}
```

El resultado de la ejecución de comando es:

```
Compiled from "BCTest.java"
public class com.adictosAlTrabajo.bc.BCTest{
    int valor;

    public com.adictosAlTrabajo.bc.BCTest();
    Code:
    0: aload_0
    1: invokespecial #1                  // Method java/lang/Object."<init>":()V
    4: aload_0
    5: bipush        42
    7: putfield      #2                  // Field valor:I
    10: return

    public int suma(int, int);
    Code:
    0: iload_1
    1: iload_2
    2: iadd
    3: ireturn
}
```

Podemos ver que la inicialización es la misma (el constructor), pero se ha añadido el método suma(), que podemos ver en segundo lugar. Recuerda que se crea un frame nuevo en la pila del hilo por cada llamada a un método. En este caso no estamos llamándolo, solo lo hemos añadido, y como tal se refleja en el código sin la llamada.

Vamos a analizar el contenido de *public int suma(int, int)* por separado:

```
public int suma(int, int);
Code:
0: iload_1
1: iload_2
2: iadd
3: ireturn
```

- La primera instrucción, que es iload_1, podemos deducir por su construcción, que se encarga de empujar a la pila de operandos el valor que está situado en la posición 1 del array de variables, y que además ese valor es un entero. Es evidente que ese valor es el del primer parámetro de llamada a la función. Esta instrucción es especial, porque es inmediata y no necesita parámetro (al ser un número muy cercano y utilizado). Si fuese referido a la posición 15 por ejemplo, sería necesario emplear iload que sí tiene parámetro.
- La segunda instrucción, hace lo mismo que la primera, pero en esta ocasión es valor es el 2, que será el segundo parámetro. Recuerda que el valor que está en la posición 0 del array de variables es la referencia a la propia clase, el *this*.
- La tercera instrucción realiza la operación de suma, iadd, que es una operación que implícitamente toma los dos valores que están en la cabeza de la pila, los suma, los elimina y deja como resultado el valor de la operación.
- Finalmente, la última instrucción, ireturn, lo que hace es devolver como salida el valor que está en la cabeza de la pila de operandos, que en este caso será el resultado de la instrucción iadd.

En este ejemplo no se hace ninguna invocación al método suma. En el siguiente paso vamos a ver cómo lo llamamos desde el constructor.

**Tercer ejemplo: llamada a un método**

En este ejemplo vamos a ver el Bytecode de una invocación al método *suma* que hemos definido anteriormente. Por simplicidad se va a realizar desde el constructor con un parámetro directo y con el valor.

El código es el siguiente:

```
package com.adictosAlTrabajo.bc;

public class BCTest {
    int valor = 42;

    public BCTest(){
        int resultado;
        resultado = suma(5,valor);
    }

    public int suma(int a, int b){
        return a+b;
    }
}
```

El resultado de la compilación es el siguiente:

```
Compiled from "BCTest.java"
public class com.adictosAlTrabajo.bc.BCTest {
    int valor;

    public com.adictosAlTrabajo.bc.BCTest();
    Code:
    0: aload_0
```

```
    1: invokespecial #1                  // Method java/lang/Object."<init>":()V
    4: aload_0
    5: bipush        42
    7: putfield      #2                  // Field valor:I
    10: aload_0
    11: iconst_5
    12: aload_0
    13: getfield      #2                  // Field valor:I
    16: invokevirtual #3                  // Method suma:(II)I
    19: istore_1
    20: return

    public int suma(int, int);
    Code:
    0: iload_1
    1: iload_2
    2: iadd
    3: ireturn
}
```

Como ves, si comparas con el primer ejemplo, la parte del constructor ha crecido bastante porque ahora además de definir el contenido de la variable *int valor=42* (ya que es el constructor), también hace la llamada al método suma. Vamos a centrarnos únicamente en el constructor, y concretamente en la parte nueva.

El código nuevo es:

```
(código del primer ejemplo...)
10: aload_0
11: iconst_5
12: aload_0
13: getfield      #2                  // Field valor:I
16: invokevirtual #3                  // Method suma:(II)I
19: istore_1
20: return
```

Vamos instrucción por instrucción:

- aload_0 como vimos antes, lo que hace es cargar en la cabeza de la pila el valor que está en la posición 0 del array de variables, que sabemos que es la referencia a *this*.
- iconst_5, que se encarga de empujar una constante a la pila, en este caso 5. Esta instrucción sólo funciona con valores del -1 al 5. Si por ejemplo fuese un 6, sería un bipush (como en la carga del valor 42). Esto se hace por optimización de código ya que no se usa operandos. También depende de la versión de la JVM. Como puedes imaginar, se introduce el 5 en la pila porque es el primer parámetro que se utilizará para la llamada a suma.
- aload_0, nuevamente carga la referencia a this en la pila.
- getfield #2, que se encarga de tomar el valor almacenado en el pool de constantes en la posición #2 y empujarlo a la pila. Por tanto tomará el valor 42 que es el que hemos cargado en esa constante en la instrucción de la posición 7: putfield #2.
- invokevirtual #3, que se encarga de hacer una llamada a un método de la clase. Exactamente llama al que está referenciado en el pool de constantes en la posición #3. Si hacemos un javap -c -v podremos ver que el valor es la referencia del método suma:

```
#3 = Methodref          #4.#24         // com/adictosAlTrabajo/bc/BCTest.suma:(II)
```

- istore_1 que almacena el valor entero, que es la salida de la llamada suma (public int) y la almacena en la posición 1 del array de variables.
- Simplemente nos queda hacer un return, que como se trata del constructor, no devuelve nada.

**Cuarto Ejemplo: con una bifurcación (if)**

Vamos a ver cómo se comporta nuestro código si introducimos una instrucción de bifurcación *if*. Hemos eliminado el código anterior y nos queda un código como el siguiente:

```
package com.adictosAlTrabajo.bc;

public class BCTest {
    public int bifurcacion(int a, int b)
    {
        if(a<b)
            return 1;
        else
            return 2;
    }
}
```

El resultado de la compilación a Bytecode es:

```
Compiled from "BCTest.java"
public class com.adictosAlTrabajo.bc.BCTest {
    public com.adictosAlTrabajo.bc.BCTest();
    Code:
    0: aload_0
    1: invokespecial #1                  // Method java/lang/Object."<init>":()V
    4: return

    public int bifurcacion(int, int);
    Code:
    0: iload_1
    1: iload_2
    2: if_icmpge     7
    5: iconst_1
    6: ireturn
    7: iconst_2
    8: ireturn
}
```

Vamos a obviar la parte del constructor que ya nos la sabemos, y vamos a centrarnos en el método *public int bifurcacion* que es el que contiene la instrucción if que vamos a analizar. Si sabes ensamblador te sonará bastante.

- iload_1, carga en la pila de operandos el valor indicado en la posición 1 del array de variables (operador inmediato).
- iload_2, carga en la pila de operandos el valor indicado en la posición 2 del array de variables.
- if_icmpge 7, que es el típico salto de comparación de ensamblador con enteros: "if int comparator greather or equal", es decir, si el segundo valor es mayor o igual (b es mayor o igual que a) entonces salta a la instrucción que está en el byte 7 del código. Si no lo es, continúa, y elimina los valores que compara de la pila de operandos. Por tanto hay una bifurcación con dos opciones:
    - *Opción 1*: no se cumple que *a<b*
        - iconst_1, se ocupa de cargar el valor entero 1 en la pila.

        - ireturn para devolver el valor 1 como salida del método.
    - *Opción 2*: sí se cumple que *a<b*
        - iconst_2, se ocupa de cargar el valor entero 2 en la pila.
        - ireturn para devolver el valor 1 como salida del método.

Como era de esperar, el sistema de bifurcación en el ByteCode de la JVM es similar al mecanismo empleado por otros sistemas de bajo nivel, como por ejemplo el ensamblador del x86.

**Quinto Ejemplo: bucle**

En este ejemplo vamos a ver un bucle for sencillo que cuenta del 1 al 5. Como ya sabrás seguramente, un bucle de tipo for de compone de:

- Contador que cambia en cada iteración.
- Comprobación (bifurcación) de la condición de parada.

El código del ejemplo:

```
1  package com.adictosAlTrabajo.bc;
2
3  public class BCTest {
4      public void bucle() {
5          int[] vector = new int[5];
6
7          for (int i = 0; i < 5; i++) {
8              vector[i] = i + 2;
9          }
10     }
11 }
```

El resultado de la compilación en Bytecode es el siguiente:

```
1  Compiled from "BCTest.java"
2  public class com.adictosAlTrabajo.bc.BCTest {
3      public com.adictosAlTrabajo.bc.BCTest();
4      Code:
5      0: aload_0
6      1: invokespecial #1                  // Method java/lang/Object."<init>":()V
7      4: return
8
9      public void bucle();
10     Code:
11     0: iconst_5
12     1: newarray       int
13     3: astore_1
14     4: iconst_0
15     5: istore_2
16     6: iload_2
17     7: iconst_5
18     8: if_icmpge     23
19     11: aload_1
20     12: iload_2
21     13: iload_2
22     14: iconst_2
23     15: iadd
24     16: iastore
25     17: iinc          2, 1
26     20: goto          6
27     23: return
28 }
```

Nuevamente nos centramos en el método *public void bucle()* que es el que nos interesa porque contiene el bucle. La secuencia de instrucciones es:

- *Comienzo del bucle:*
    - iconst_5, empuja el valor número 5, que es un entero, a la cabeza de la pila.
    - newarray int, genera un array de enteros con la extensión indicada en la cabeza de la pila, es decir, 5.
    - astore_1, almacena la referencia del array de 5 elementos en el array de variables locales al método en la posición 1, que por tanto almacenará la referencia de la variable *vector*
    - iconst_0, carga la constante 0 en la pila, que es el valor con el que inicalizamos la variable del contador del bucle, que es i.
    - istore_2, almacena en las variables locales, en la posición 2, el valor 0 que hemos puesto anteriormente en la cabeza de la pila. Por tanto en la posición 2 estará almacenada la variable *i*.
    - iload_2, carga en la pila la información que acabamos de salvar, es decir, el valor del contador i
    - iconst_5, introduce en la pila la constante 5, que es la condición que se evalúa para ver si termina el bucle en cada vuelta.
    - if_icmpge 23, es la comparación de enteros que compara los dos últimos valores introducidos en la pila. Esto es, compara el valor cargado en la pila de la variable en posición 2 (i) y el valor de la constante empujada a la pila, que es el 5. Se trata de una comparación de enteros, de modo que si i es mayor o igual a 5, entonces salta a la instrucción de la posición del byte 23 (que es la salida del bucle).
- *Contenido del bucle:*
    - aload_1, carga en la pila la referencia al objeto (por eso comienza la instrucción por a) que está en la posición 1 del array de variables del método. En este caso se trata de la referencia al array de ints que hemos definido al comienzo.
    - iload_2, carga el valor que está en la posición 2 del array de variables del método en la pila. Es el valor de i.
    - iload_2, carga de nuevo el valor de i, ya que lo vamos a utilizar para hacer la suma dentro del bucle y también para referenciar la posición. No es que se haya vuelto loco el compilador :)
    - iconst_2, carga el valor 2 en la pila. Se trata del número 2, que es el número que vamos a sumar al valor de i para asignar a una posición i del vector.
    - iadd, realiza la operación de suma de los dos últimos valores, que es el 2 y el valor de i.
    - iastore,almacena el valor del resultado de la suma en el vector indicado en la pila (segundo valor de la pila), y en la posición indicada por el tercer valor de la pila.
- *Final del bucle:*
    - iinc 2, 1 incrementa el valor que está en la posición del array de variables en la posición 2 en 1 unidad, que es el aumento del contador del bucle.
    - goto 6, redirige el flujo nuevamente a la instrucción que está en la posición 6 del código, por tanto, es el lugar en el que se van a cargar en la pila de operandos los valores para hacer la comparación de salida del bucle.

## 6. Conclusiones

En este tutorial hemos visto la importancia de conocer el bytecode al que se transforman nuestras instrucciones de Java. Java no es más que un lenguaje de alto nivel, que es transformado a un lenguaje de nivel intermedio, bytecode, para ser intepretado por la Java Virtual Machine (JVM). La máquina virtual de Java permite abstraer al programador de la dificultad de adaptar su código para cada tipo de máquina en la que lo pretenda ejecutar, relegando este inconveniente a la utilización de diferentes

máquinas virtuales que son las que realmente se adaptan a las particularidades de cada sistema.

Conocer el bytecode también nos ayudará a ser mejores programadores. Nos permite comprobar cómo se llevan a cabo en realidad mecanismos que sólo vemos sus efectos, como por ejemplo la llamada al constructor por defecto del padre o cuándo se realiza la asignación de variables. También podemos comprobar qué estructuras son realmente más óptimas a la hora de generar código máquina, que es lo que realmente se ejecuta en nuestros procesadores.

### 7. Referencias


- https://docs.oracle.com/javase/specs/jls/se8/html/index.html
- http://viralpatel.net/blogs/java-virtual-machine-an-inside-story/
- http://blog.jamesdbloom.com/JVMInternals.html
- http://blog.jamesdbloom.com/JavaCodeToByteCode_PartOne.html
- https://www.artima.com/insidejvm/ed2/jvm.html
- http://www.programering.com/a/MzM3QzNwATA.html
- http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals/
- http://www.javacodegeeks.com/2013/12/mastering-java-bytecode.html


## A continuación puedes evaluarlo:

Regístrate para evaluarlo

## Por favor, vota +1 o compártelo si te pareció interesante



Anímate y coméntanos lo que pienses sobre este **TUTORIAL:**

**» Registrate** y accede a esta y otras ventajas **«**