JVM

J.A. Medina/ J.J. Sánchez
Ciencias de la Computación
Universidad de Alcalá

PRESENTACIÓN

Objetivo

- Conocer la arquitectura de la JVM, detalles de su funcionamiento y demás características más importantes.

Bibliografía

Gosling J., Joy B., Steele G. & Bracha G. "*The Java Language Specification*", Addison-Wesley, 2000 (disponible en formato electrónico en JavaSpec).

Lindholm T. & Yellin F., "*The Java Virtual Machine Specification*", Addison-Wesley, 1999 (disponible en formato electrónico en JVMSpec).

Engels B., "Inside the Java 2 Virtual Machine", McGraw-Hill Companies, Segunda Edición, 2000

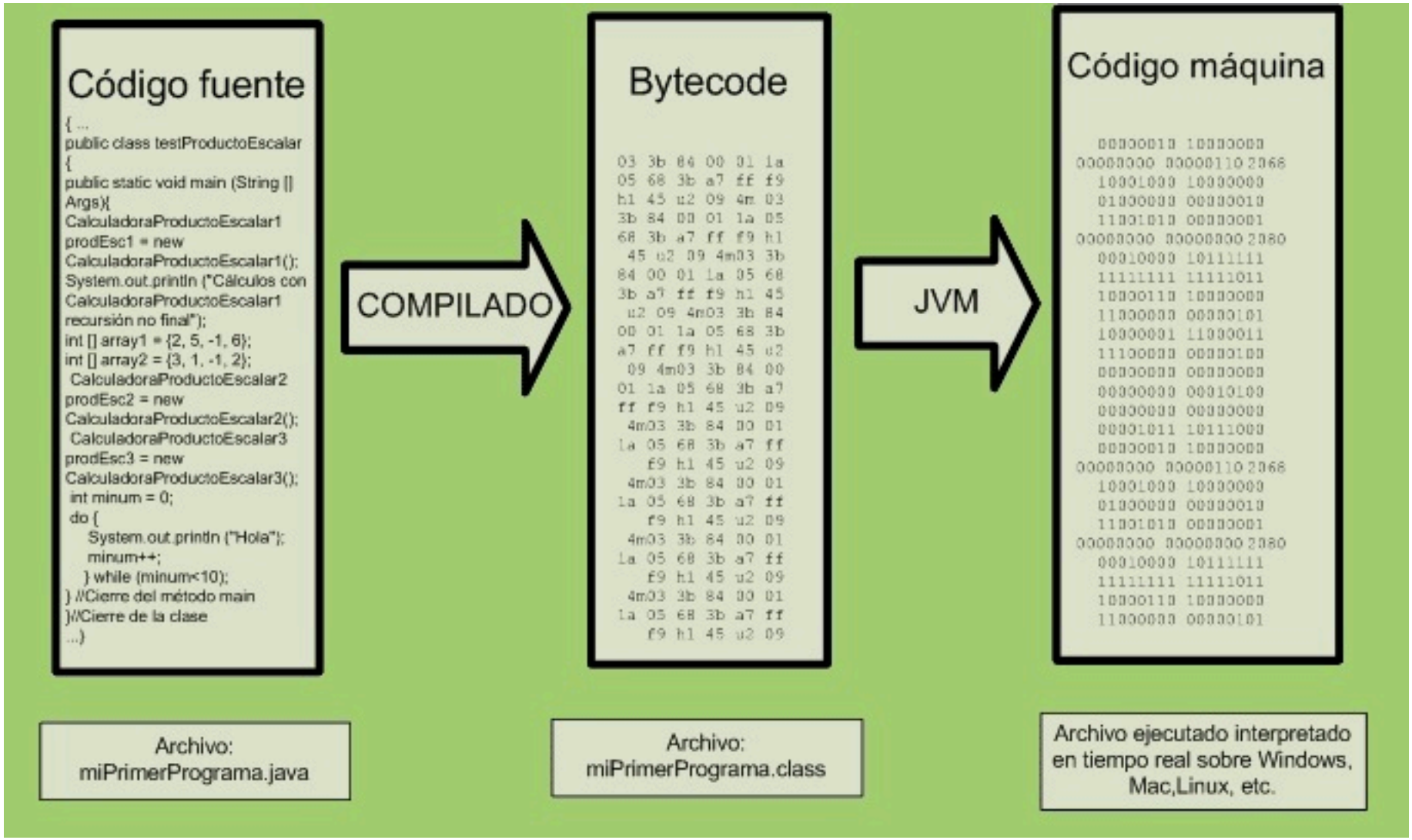
Taivalsaari A.,"Virtual Machine Design", Notas Seminario 2003 (disponible electrónicamente en "Virtual Machine Design)

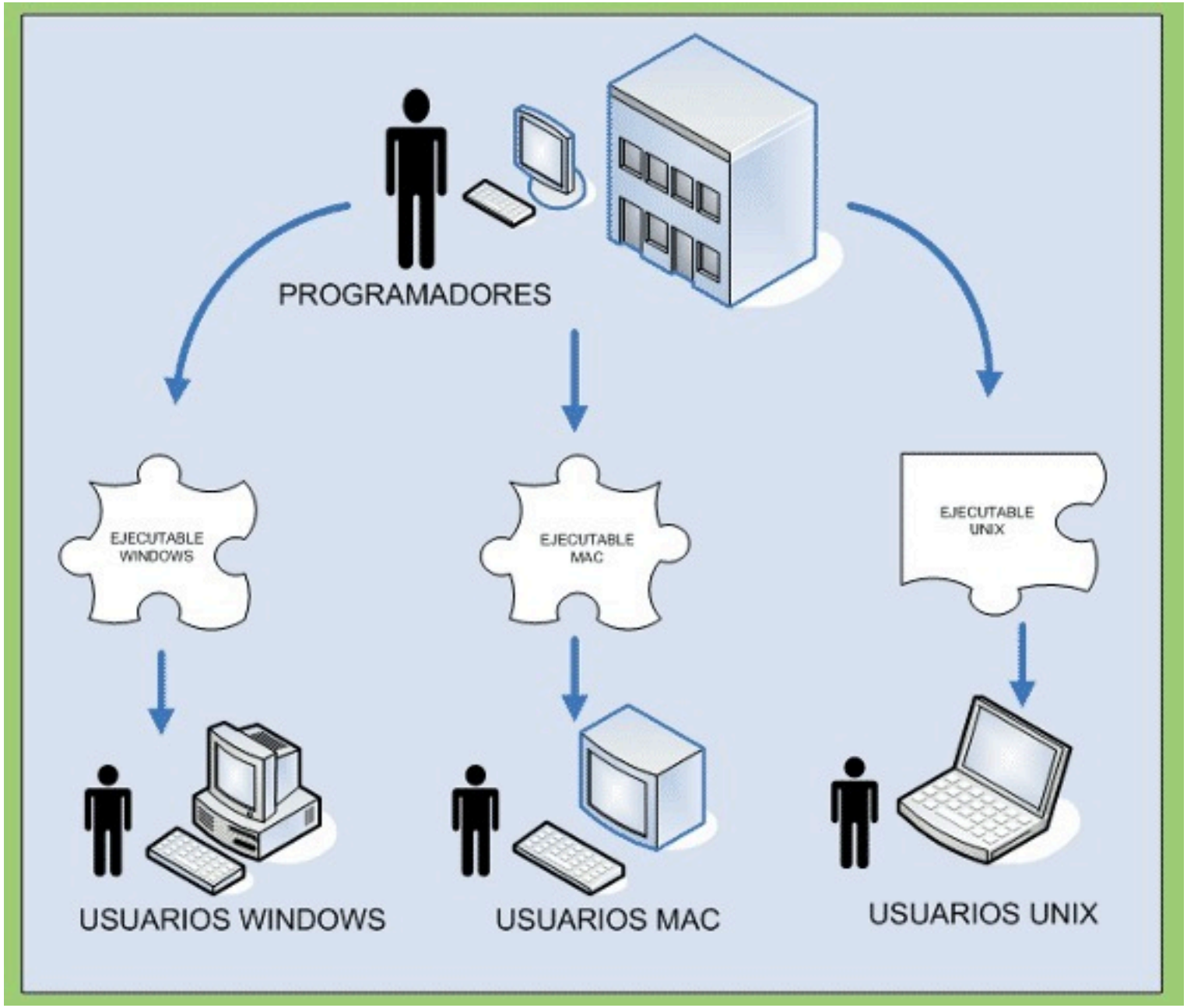
Java Virtual Machine (JVM)

INTRODUCCIÓN

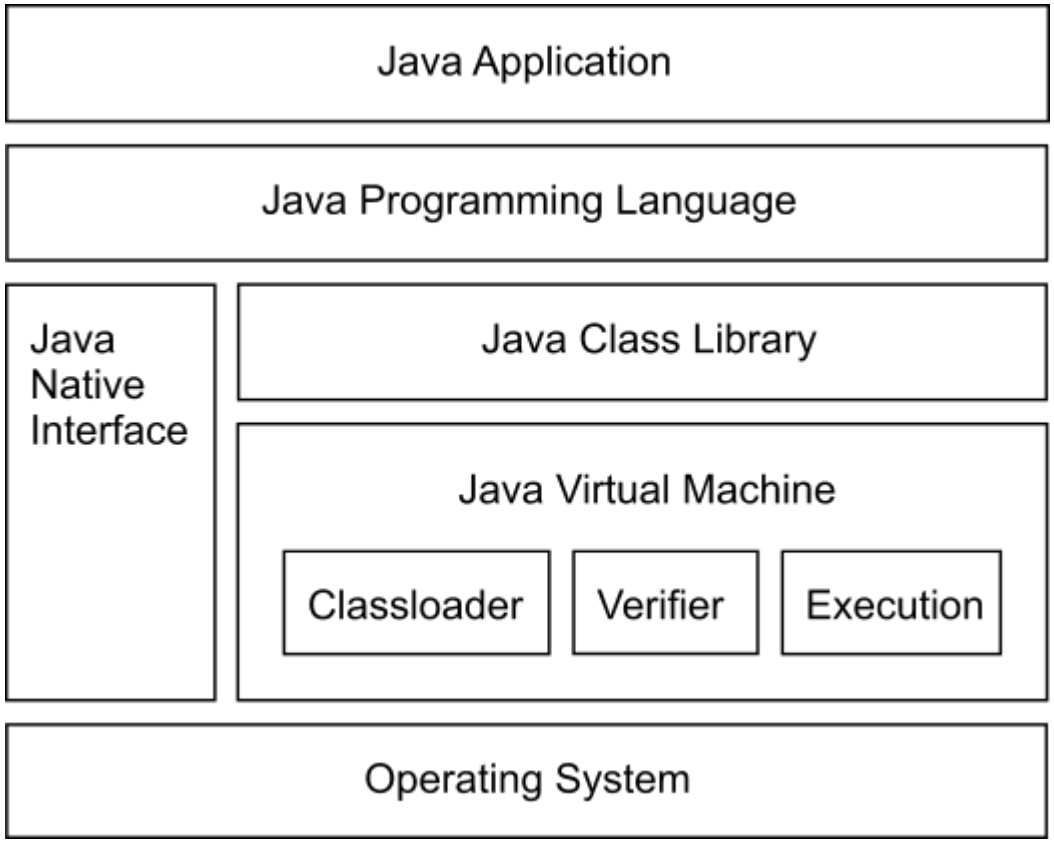
Tecnología Java

- El lenguaje de programación Java
- La librería (JDK)
- La máquina virtual de Java (JVM)
  - Un juego de instrucciones y su significado - los ***bytecodes***
  - Un formato binario – el ***class file format***
  - Un algoritmo para ***verificar*** los ficheros class


Código fuente
{ ...
public class testProductoEscalar
{
public static void main (String []
Args){
CalculadoraProductoEscalar1
prodEsc1 = new
CalculadoraProductoEscalar1();
System.out.println ("Cálculos con
CalculadoraProductoEscalar1
recursión no final");
int [] array1 = {2, 5, -1, 6};
int [] array2 = {3, 1, -1, 2};
CalculadoraProductoEscalar2
prodEsc2 = new
CalculadoraProductoEscalar2();
CalculadoraProductoEscalar3
prodEsc3 = new
CalculadoraProductoEscalar3();
int minum = 0;
do {
System.out.println ("Hola");
minum++;
} while (minum<10);
} //Cierre del método main
}//Cierre de la clase
...}
COMPILADO
Bytecode
03 3b 84 00 01 1a
05 68 3b a7 ff f9
h1 45 u2 09 4m 03
3b 84 00 01 1a 05
68 3b a7 ff f9 h1
45 u2 09 4m03 3b
84 00 01 1a 05 68
3b a7 ff f9 h1 45
u2 09 4m03 3b 84
00 01 1a 05 68 3b
a7 ff f9 h1 45 u2
09 4m03 3b 84 00
01 1a 05 68 3b a7
ff f9 h1 45 u2 09
4m03 3b 84 00 01
1a 05 68 3b a7 ff
f9 h1 45 u2 09
4m03 3b 84 00 01
1a 05 68 3b a7 ff
f9 h1 45 u2 09
4m03 3b 84 00 01
1a 05 68 3b a7 ff
f9 h1 45 u2 09
JVM
Código máquina
00000010 10000000
00000000 00000110 2068
10001000 10000000
01000000 00000010
11001010 00000001
00000000 00000000 2080
00010000 10111111
11111111 11111011
10000110 10000000
11000000 00000101
10000001 11000011
11100000 00000100
00000000 00000000
00000000 00010100
00000000 00000000
00001011 10111000
00000010 10000000
00000000 00000110 2068
10001000 10000000
01000000 00000010
11001010 00000001
00000000 00000000 2080
00010000 10111111
11111111 11111011
10000110 10000000
11000000 00000101
Archivo:
miPrimerPrograma.java
Archivo:
miPrimerPrograma.class
Archivo ejecutado interpretado
en tiempo real sobre Windows,
Mac,Linux, etc.


PROGRAMADORES
EJECUTABLE WINDOWS
EJECUTABLE MAC
EJECUTABLE UNIX
USUARIOS WINDOWS
USUARIOS MAC
USUARIOS UNIX

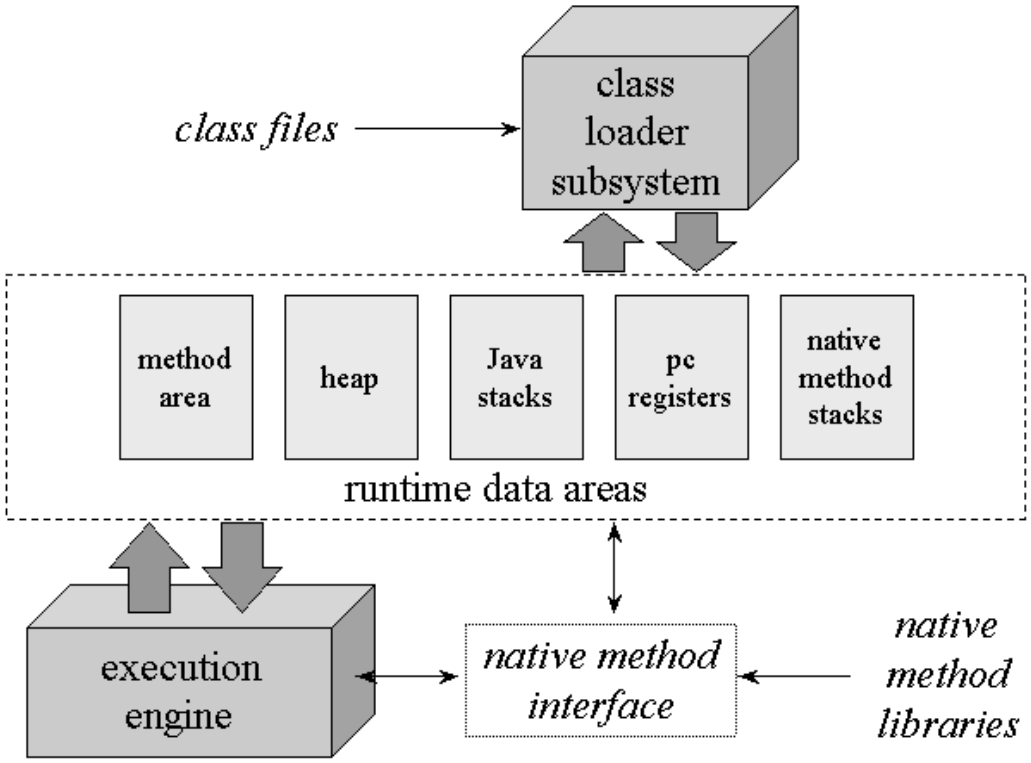
Sistema Java



La máquina virtual de Java

- Soporta arquitecturas CISC y RISC
- Máquina de pila (similar a Forth VM)
- Esperando un chip que implemente dicha arquitectura...

La JVM


class files
class loader subsystem
method area
heap
Java stacks
pc registers
native method stacks
runtime data areas
execution engine
native method interface
native method libraries

Registros

- PC – contador de programa
- OPTOP – puntero a la cima de la pila de operandos
- MARCO – puntero al actual entorno de ejecución
- VARS – puntero a la primera variable local del entorno actual

La pila de Java

- A medida que se crean hilos, cada uno recibe un contador de programa y una pila
- Se crea un marco (frame) en cada invocación a un método, cada uno contiene
  - Variables locales
  - Entorno de ejecución
  - Pila de operandos

Variables locales

- Array de variables de 32 bit
  - Tipos de más de 32 bit (double) usan celdas consecutivas
  - Apuntado por el registro vars
  - Se almacena y se extrae desde/hacia la pila de operandos

Entorno de ejecución

- Información sobre el estado actual de la pila de Java
  - Anterior método invocado
  - Puntero a las variables locales
  - Puntero al comienzo y al final de la pila de operandos

Pila de operandos

- 32 bit FIFO
- Tiene los argumentos de los opcodes
- Un subconjunto de la pila de Java
  - Área principal para el estado de la ejecución del bytecode

Montículo con Garbage Collected

- Memoria en la que las instancias se crean
- El intérprete monitoriza su uso y reclama la memoria que no está en uso
- La recolección de basura es automática

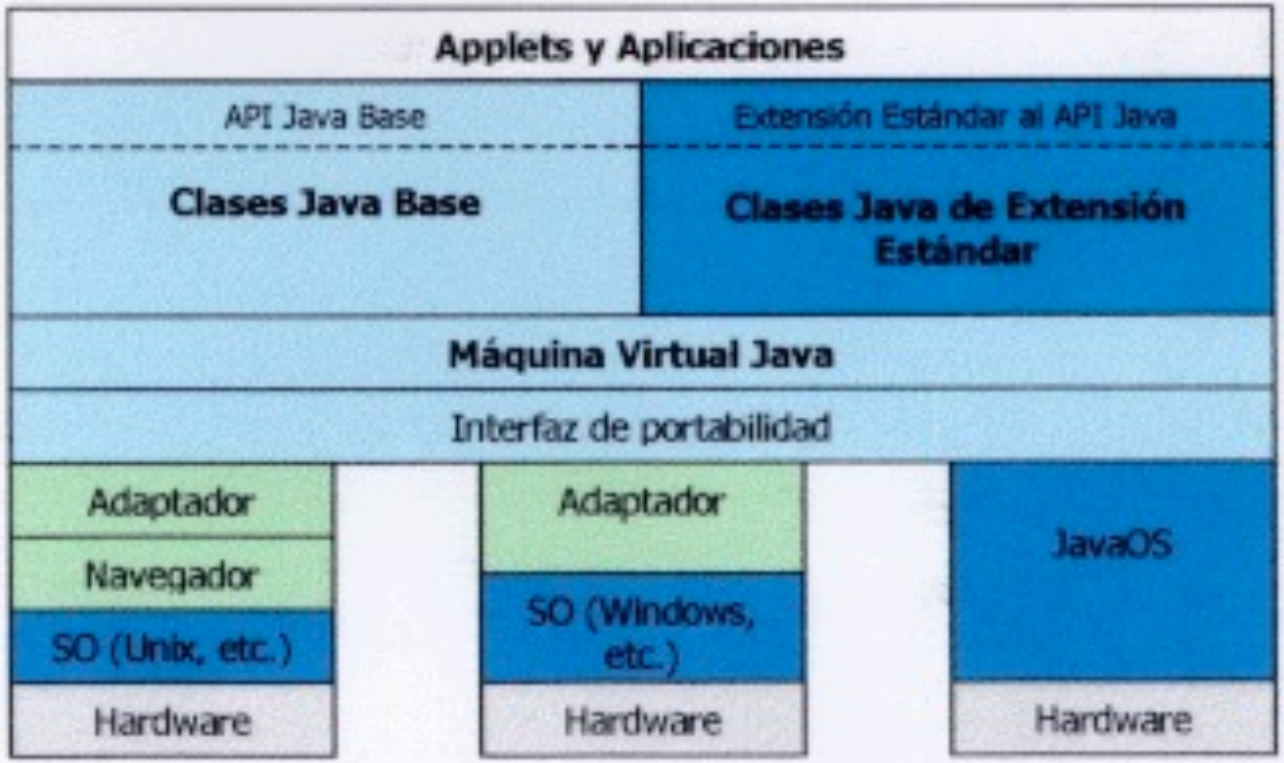
Área de memoria

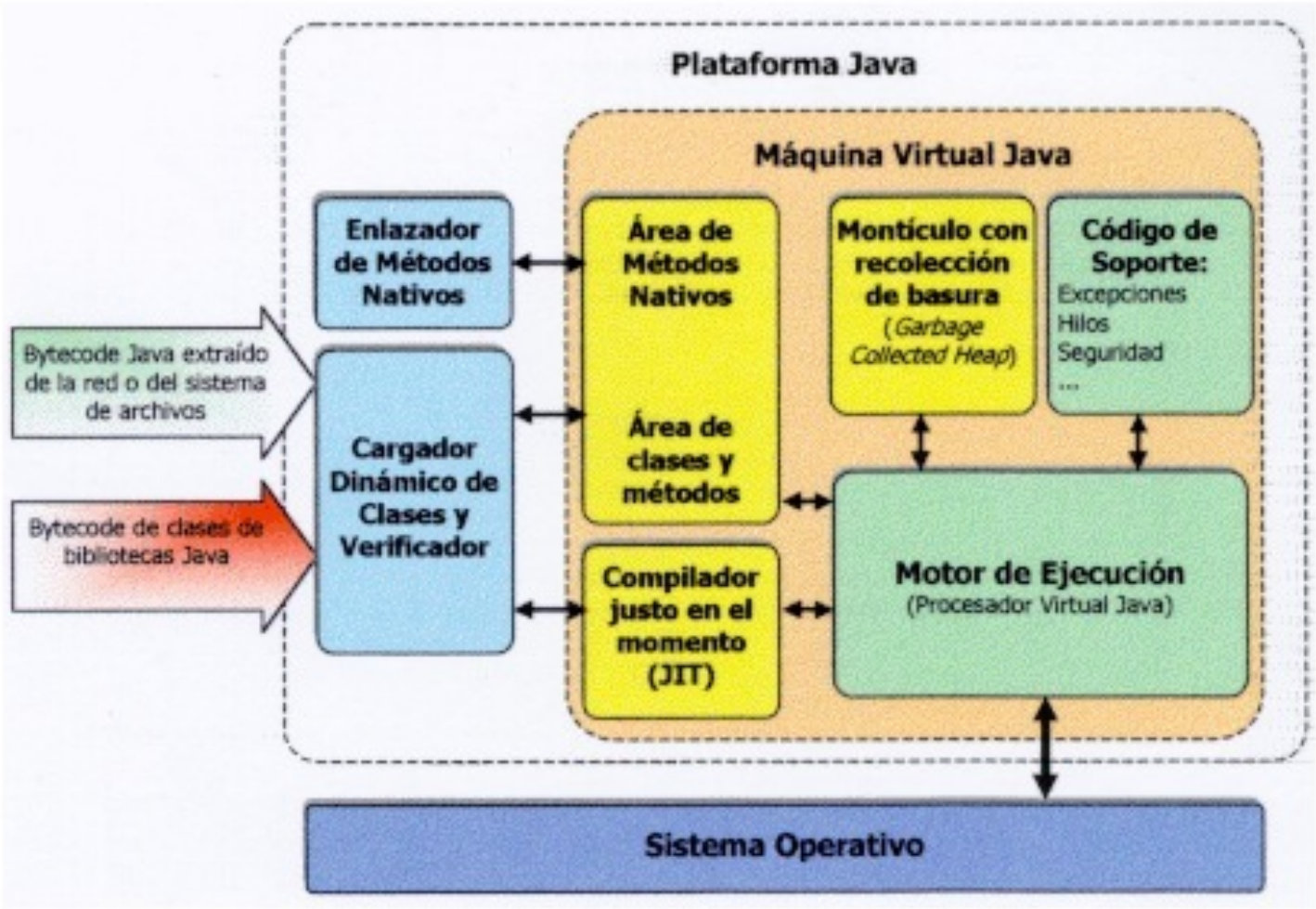
- Área de memoria- bytecodes de todos los métodos de Java
- Constant Pool – nombres de clases, nombres de métodos y atributos, cadenas

Limitaciones de la máquina virtual*

- 4 Gb de espacio interno debido al empleo de pilas de 32 bits
- Los métodos están limitados a 32 Kb normalmente debido al direccionamiento usado en los saltos (16 bit offset)
- 256 variables locales/pilas (campo de 8 bit)
- 32k entradas en el pool por método

* ¡¡¡Depende de la implementación!!!!


Applets y Aplicaciones
API Java Base
Extensión Estándar al API Java
Clases Java Base
Clases Java de Extensión Estándar
Máquina Virtual Java
Interfaz de portabilidad
Adaptador
Navegador
SO (Unix, etc.)
Hardware
Adaptador
SO (Windows, etc.)
Hardware
JavaOS
Hardware


Plataforma Java
Máquina Virtual Java
Enlazador de Métodos Nativos
Área de Métodos Nativos
Área de clases y métodos
Montículo con recolección de basura (Garbage Collected Heap)
Código de Soporte:
Excepciones
Hilos
Seguridad
...
Bytecode Java extraído de la red o del sistema de archivos
Bytecode de clases de bibliotecas Java
Cargador Dinámico de Clases y Verificador
Compilador justo en el momento (JIT)
Motor de Ejecución
(Procesador Virtual Java)
Sistema Operativo

CLASS LOADING

El subsistema de carga de clases

- El subsistema de carga de clases realiza tres funciones: carga, enlazado e inicialización
- El proceso de enlazado consiste en tres subtareas: verificación, preparación y resolución

Proceso de carga de clases

- La carga se refiere a la obtención del fichero de clase del tipo (clase), su parseo para obtener la información y almacenamiento de la información en el área de metodos
- Para cada tipo, la JVM almacena (entre otras) las siguientes informaciones:
  - El nombre completo de la clase
  - El nombre completo de la superclase y una lista de los interfaces directos que implementa
  - Si es una clase o un método
  - Los modificadores del tipo (public, abstract, final, etc)
  - Un pool de constantes para el tipo: constantes y referencias simbólicas
  - Información de los atributos: nombres y modificadores
  - Información de los métodos: nombre, tipo de retorno, número y tipo de los parámetros, modificadores, bytecodes, tamaño de la pila y tabla de excepciones

Proceso de carga de clases (y2)

- Al final del proceso se crea una instancia de java.lang.Class para el tipo cargado
- Sirve para informar al programador sobre el tipo (reflexión)

```
// algunos métodos de la clase Class
public String getName()
public Class getSupClass()
public boolean isInterface()
public Class[] getInterfaces()
public Method[] getMethods()
public Fields[] getFields()
public Constructor[] getConstructors()
```

- Sólo se crea una instancia de java.lang.Class por clase (tipo)

Enlazado: Verificación

- En la fase de enlazado hay tres subfases: verificación, preparación y resolución
- En la fase de verificación se asegura que la representación binaria de la clase es correcta:
  - Que se generó utilizando un compilador válido y que está bien formado
  - La clase puede ser una subclase válida en tiempo de compilación pero no en tiempo de ejecución
- Algunas cosas a verificar:
  - Que todo método tiene una signatura estructuralmente válida
  - Que toda instrucción obedece las reglas del lenguaje Java
  - Que todo salto se realiza al comienzo (y no en medio) de una instrucción (para evitar enmascaramiento de código -> virus)

Enlazado: Preparación

- Se reserva memoria para las variables de la clase (i.e `static`) y se inicializan con los valores por defecto
- Las atributos (no estáticos) no se inicializan en este momento

Enlazado: Resolución

- Se reemplazan las referencias simbólicas a tipos, atributos y métodos por sus referencias finales
- Las referencias simbólicas se resuelven directamente mediante una búsqueda en el área de métodos

Inicialización

- En esta fase, los atributos reciben los valores indicados por el programador
- Consiste de dos fases:
  - Inicializar su superclase directa
  - Ejecutar sus propias instrucciones de inicialización
- La primera clase inicializada es `Object`.
- Las variables estáticas son tratadas como constantes y toman su valor en la compilación

Instanciación

- Tras las etapas anteriores las clases están lista para su uso y pueden ser instanciadas
- Cuando una clase se instancia, se crea memoria para ella en el montículo
- La memoria se asigna de manera recursiva para todas las variables de su superclase y sus antecesores
- Tras esto se inicializan dichas variables a sus valores por defecto

Instanciación (y2)

- El constructor se ejecuta y se procesa de la siguiente manera
  - Asigna los argumentos para el constructor a sus variables parámetros
  - Si el constructor se ha llamado explícitamente por otro constructor de la clase, evalúa los argumentos y procesa el constructor invocando recursivamente
  - Inicializa las variables para la clase a sus valores adecuados
  - Ejecuta el resto del cuerpo del constructor
- Finalmente devuelve una referencia al nuevo objeto creado

BYTECODE

Juego de instrucciones

- Codificación "Big Endian" – bits de mayor orden en la dirección de memoria más baja
- Las instrucciones se alinean al byte por motivos de eficiencia
- En la actualidad maneja 160 opcodes
- Las instrucciones están muy relacionadas con las fuentes de Java (fácil decompilación)

Tipos de datos

`reference`	Puntero a un objeto
`int`	32-bit integer (signed)
`long`	64-bit integer (signed)
`float`	32-bit floating-point (IEEE 754-1985)
`double`	64-bit floating-point (IEEE 754-1985)

- No hay `boolean, char, byte, o short`
  - La pila contiene sólo datos de 32-bit y 64-bit
  - Instrucciones de conversión

Juego de instrucciones

- Los `bytecodes`
- Operaciones en la pila de operandos
- Longitud variable
  - Sencillas, p.e. `iadd`
  - Complejas, p.e. `new`
- Manejan referencias simbólicas

Tipos de instrucciones

- Aritméticas
- Almacenamiento/extracción
- Conversión de tipos
- Creación y manipulación de objetos
- Manipulación de la pila de operandos
- Control del flujo
- Invocación y retorno de métodos

Aritméticas

- Operan en los valores de la pila de operandos
- Devuelven el resultado a dicha pila
- Instrucciones para `int, long, float` y `double`
- No soportan `byte, short o char`

Aritméticas

- Suma: `iadd, ladd, fadd, dadd`
- Resta: `isub, lsub, fsub, dsub`
- Multiplicación: `imul, lmul, fmul, dmul`
- División: `idiv, ldiv, fdiv, ddiv`
- Resto: `irem, lrem, frem, drem`
- Negación: `ineg, lneg, fneg, dneg`
- Desplazamiento: `ishl, ishr, iushr, lshl, lshr, lushr`
- Bitwise OR: `ior, lor`
- Bitwise AND: `iand, land`
- Bitwise XOR: `ixor, lxor`
- Incremento: `iinc`
- Comparación: `dcmpg, dcmpl, fcmpg, fcmpl, lcmp`

Almacenamiento/extracción

- Extracción
  - Coloca un valor de una variable local en la pila
  - Coloca una constante en la pila
- Almacenamiento
  - Transfiere un valor de la pila a una variable local

Almacenamiento/extracción

- Extrae una variable local
  - `iload, iload_<n>, lload, lload_<n>, fload, fload_<n>, dload, dload_<n>, aload, aload_<n>`
- Almacena una variable local
  - `istore, istore_<n>, lstore, lstore_<n>, fstore, fstore_<n>, dstore, dstore_<n>, astore, astore_<n>`
- Extrae una constante
  - `bipush, sipush, ldc, ldc_w, ldc2_w, aconst_null, iconst_m1, iconst_<i>, lconst_<l>, fconst_<f>, dconst_<d>`

Ejemplo almacenamiento/extracción

```
int a, b, c;

a = 1;
b = 123;
c = a+b;
```

```
0:   iconst_1
1:   istore_0    // a
2:   bipush 123
4:   istore_1    // b
5:   iload_0     // a
6:   iload_1     // b
7:   iadd
8:   istore_2    // c
```

Instrucciones de Objetos

- Crear una nueva instancia o array
  - `new, newarray, anewarray, multianewarray`
- Acceso a atributos
  - `getfield, putfield, getstatic, putstatic`
- Craga/almacenamiento de Array
  - `baload, caload, saload, iaload, laload, faload, daload, aaload`
  - `bastore, castore, sastore, iastore, lastore, fastore, dastore, aastore`
- Longitud de un Array
  - `arraylength`
- Verificación de propiedades
  - `instanceof, checkcast`

Manipulación de la pila de operandos

- Manipulación directa
  - `pop, pop2`
  - `dup, dup2, dup_x1, dup2_x1, dup_x2, dup2_x2`
  - `swap`

Control de flujo

- Salto condicional
  - `ifeq, iflt, ifle, ifne, ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmplt, if_icmpgt, if_icmple, if_icmpge, if_acmpeq, if_acmpne.`
- Switch
  - `tableswitch, lookupswitch.`
- Salto incondicional
  - `goto, goto_w, jsr, jsr_w, ret.`

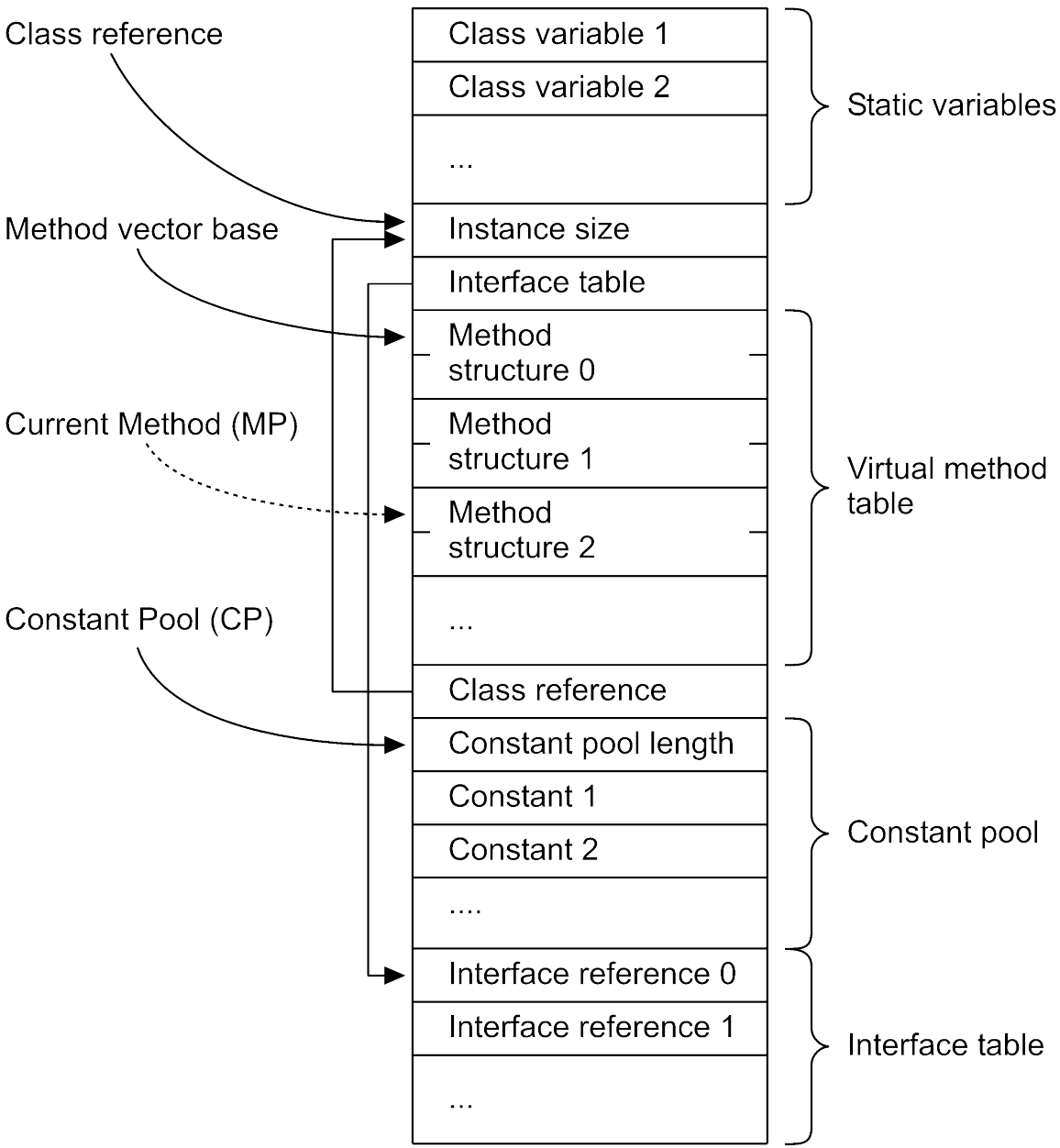
Invocación de métodos y retorno

- `invokevirtual`
  - Invoca un método
  - Es el más común
- `invokeinterface`
  - Invoca un método que es implementado por un interfaz
- `invokespecial`
  - Invoca un método que requiere un manejo especial
  - Un método de inicialización, un método privado o de una superclase
- `invokestatic`
  - Invoca un método estático

INVOCACIÓN DE MÉTODOS

Información de clases

- Tamaño de las instancias
- Variable estáticas
- Tabla de métodos
- Tabla de interfaces
- Pool de constantes
- Referencia a superclase

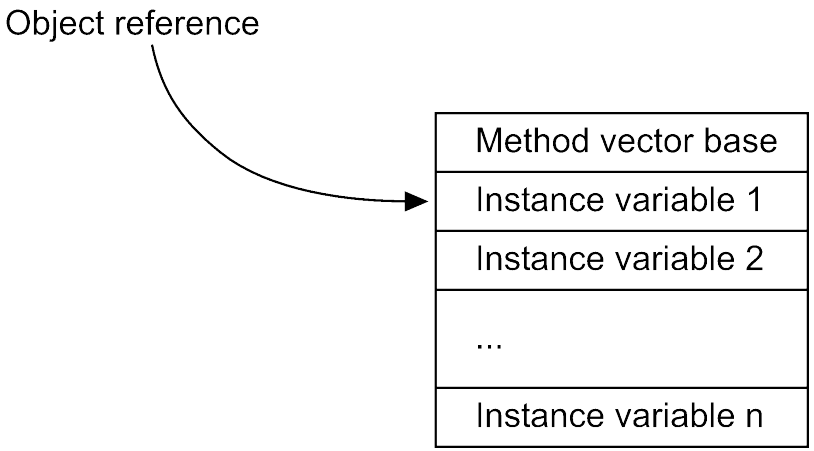

Class reference
Method vector base
Current Method (MP)
Constant Pool (CP)
Class variable 1
Class variable 2
...
Instance size
Interface table
Method structure 0
Method structure 1
Method structure 2
...
Class reference
Constant pool length
Constant 1
Constant 2
....
Interface reference 0
Interface reference 1
...
Static variables
Virtual method table
Constant pool
Interface table

Estructura de los métodos

Start address	Method length	
Constant pool	Local count	Arg. count

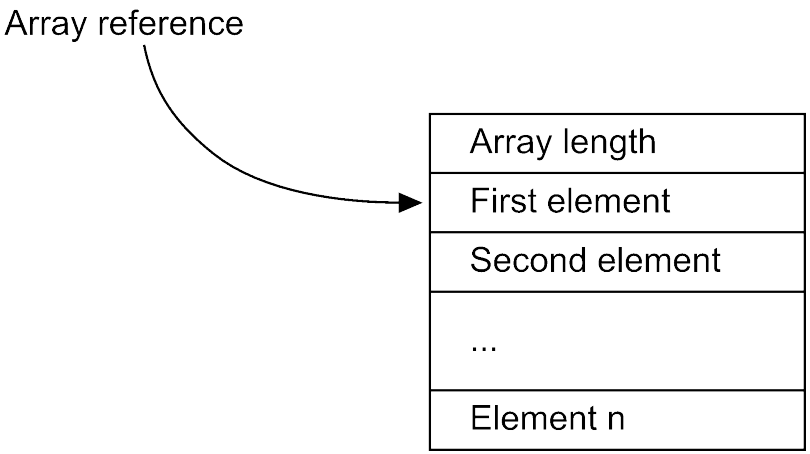
- Información
  - Dirección
  - Longitud
  - Puntero al pool de constantes de la clase
  - Número de argumentos y variables locales

Formato de los objetos



- Puntero directo
- Manejador
- Puntero a la información de la clase

Formato de los Arrays



- Puntero directo
- Manejador
- Longitud

Pool de constantes

- Contiene:
  - Constantes simples (p.e. 123, 0.345)
  - Cadenas
  - Referencias a clases
  - Referencias a atributos
  - Referencias a métodos
- Todas las referencias son simbólicas en el fichero class
- Se convierten a punteros directos en la carga

Estructuras de datos en tiempo de ejecución

- PC – contador de programa
- Pila de operandos
  - SP – puntero de pila
  - VP – puntero de variables
- MP – puntero de métodos
  - Referencias a las estructuras de los métodos
- CP – pool de constantes
  - Pool de constantes actual

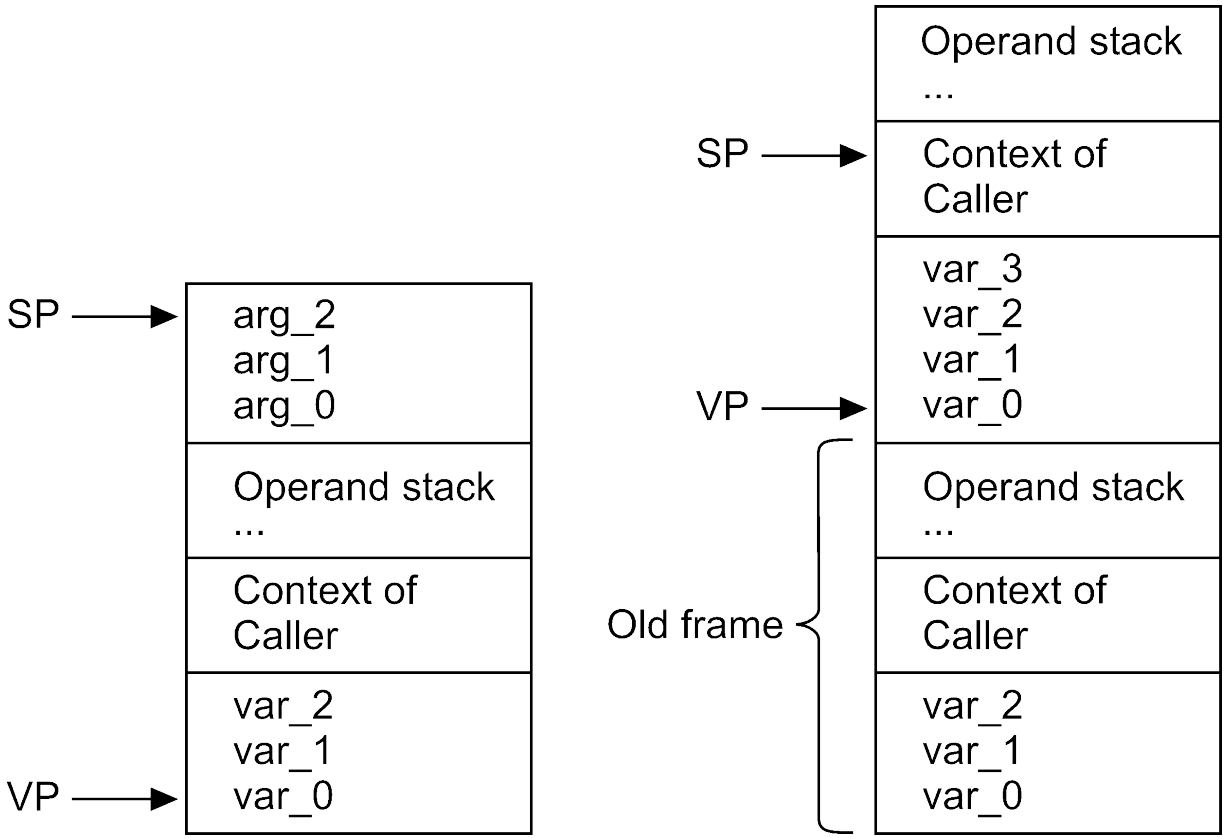
Paso de parámetros (vía pila)

```
   int val = foo(1, 2);
    ...
    public int foo(int a, int b) {
        int c = 1;
        return a+b+c;
    }

The invocation sequence:
    aload_0             // Push the object reference
    iconst_1            // and the parameter onto the
    iconst_2            // operand stack.
    invokevirtual   #2  // Invoke method foo:(II)I.
    istore_1            // Store the result in val.

public int foo(int,int):
    iconst_1            // The constant is stored in a method
    istore_3            // local variable (at position 3).
    iload_1             // Arguments are accessed as locals
    iload_2             // and pushed onto the operand stack.
    iadd                // Operation on the operand stack.
    iload_3             // Push c onto the operand stack.
    iadd
    ireturn             // Return value is on top of stack.a
```

Pila al invocar métodos



GARBAGE COLLECTION

Historia del GC

- Las técnicas de “Garbage Collection” comenzaron a usarse en los 60 con LISP, Smalltalk, Eiffel, Haskell, ML, Scheme y Modula-3
- Se popularizaron en los 90 con Java (y luego C#)
- La implementación del GC en la JVM ha mejorado sensiblemente en las pasadas décadas

Beneficios/Inconvenientes

- Beneficios
  - Aumenta la fiabilidad
  - Separación de gestión de memoria del diseño del programa
  - Tiempo de depuración inferior
  - Minimizan el desperdicio de memoria

    Java programs do NOT have memory leaks; "unintentional object retention" is more accurate
- Inconvenientes
  - Longitud de las pausas de GC
  - Utilización de CPU/Memoria

Opciones de GC en la JVM

- Sun 1.3 JDK incluye 3 estrategias de GC
- 1.4 JDK incluye 6 y una docena de opciones de línea de comando
- En función de la aplicación deberemos escoger la estrategia:
  - Pequeñas pausas más frecuentes
  - Mayores pausas menos frecuentes

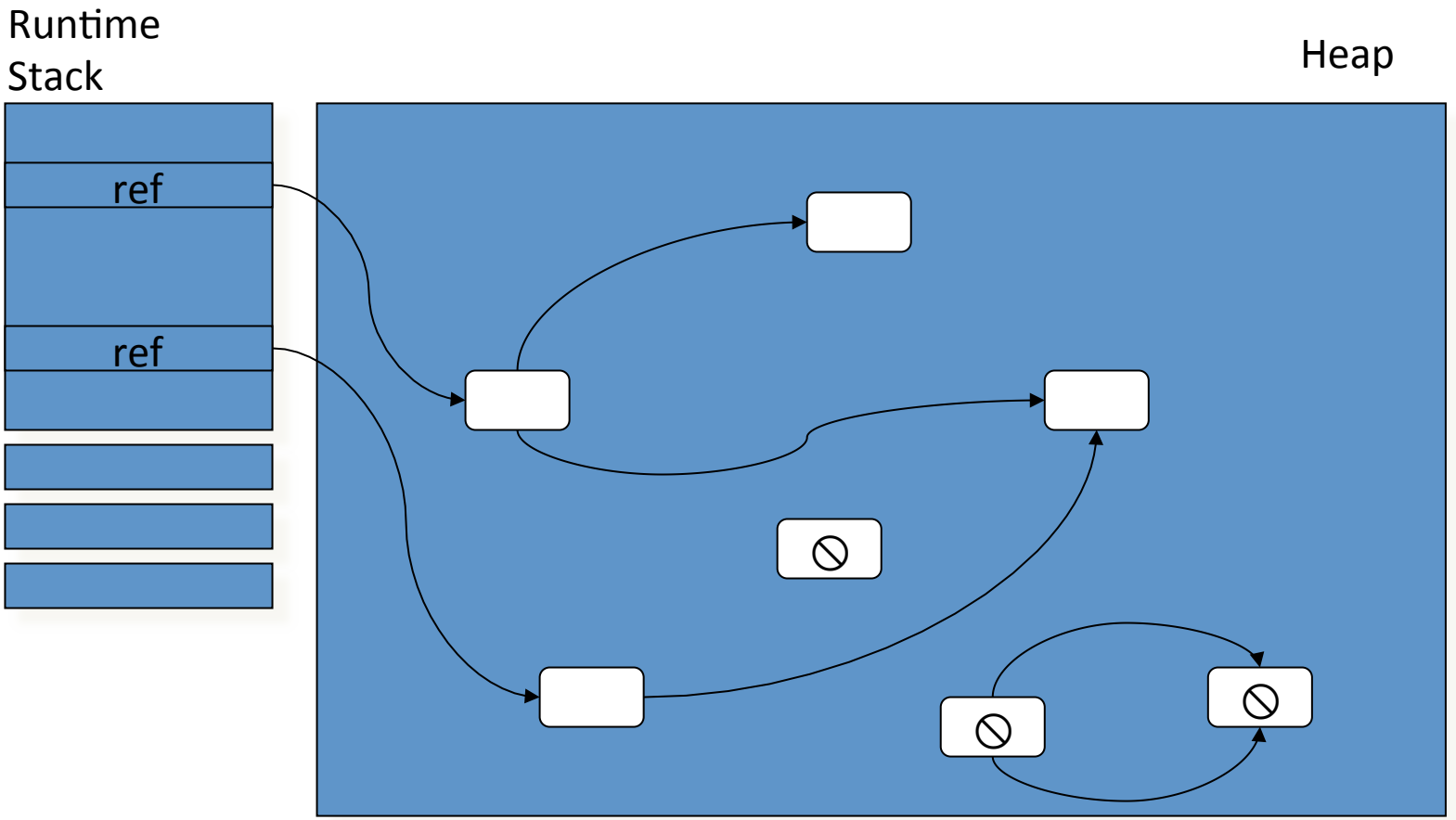
Fases GC

- GC tiene dos fases:
  - Detección
  - Reclamación
- Los pasos a seguir incluyen
  - Mark-Sweep, Mark-Compact, Copying…

Accesibilidad

- Raíces – referencias a un objeto en una variable estática o local de la “frame” activa
- Objetos raíz – directamente alcanzables desde las raíces
- Objetos Vivos – objetos transitivamente alcanzables desde las raíces
- Objetos basura – resto objetos

Ejemplo accesibilidad



Algoritmos de GC

- Dos aproximaciones:
  - Conteo de referencias – mantiene una cuenta del número de referencias a un objeto; si la cuenta es cero es un objeto basura
  - Trazado – recorre el grafo de los objetos empezando por las raíces, marcando los objetos; al terminar los objetos no marcados son basura

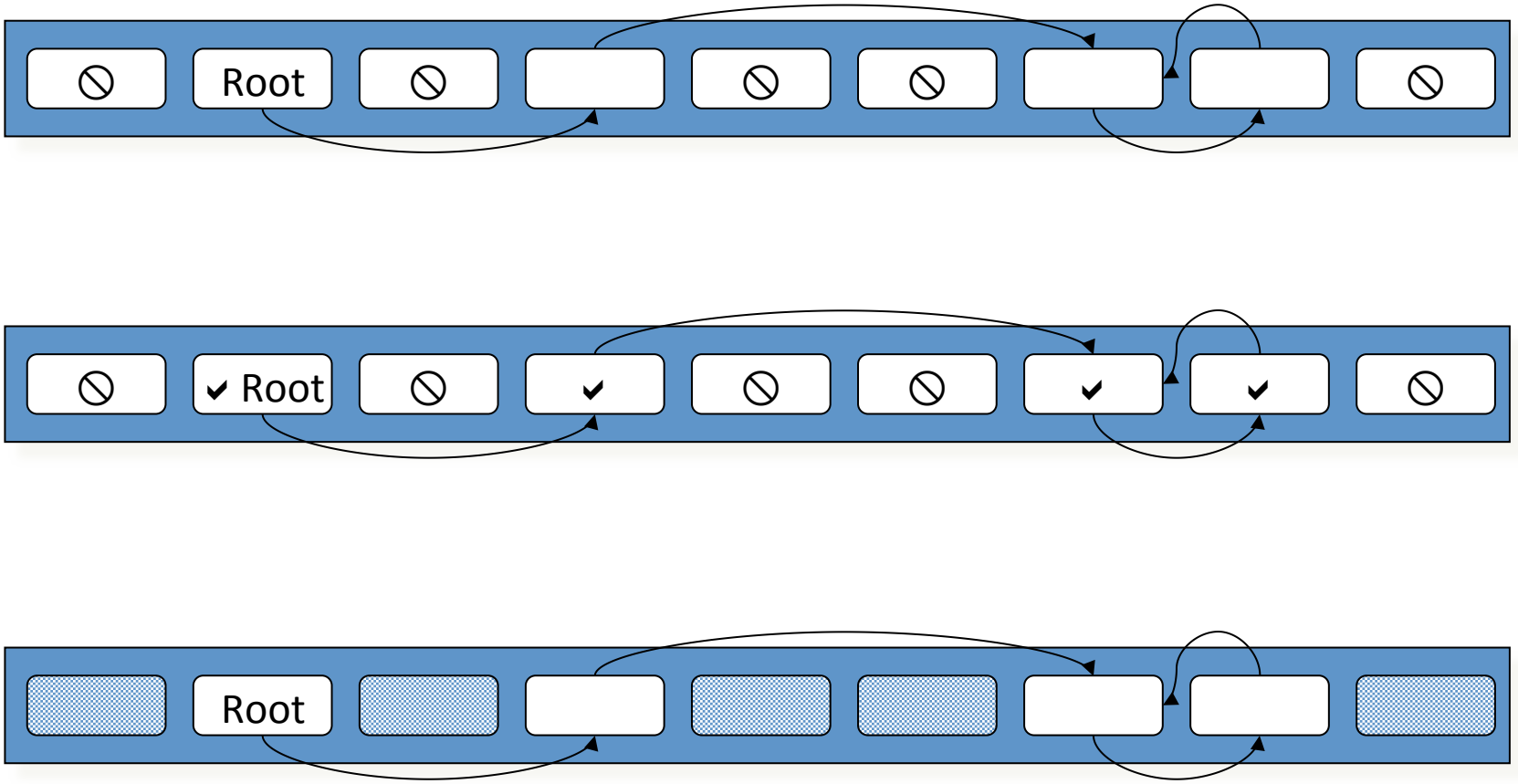
Conteo de referencias

- Ventajas:
  - Puede ejecutarse en ciclos breves
- Desventajas:
  - Problemas con los ciclos
  - Sobrecarga al incrementar/decrementar los contadores
- Está “demodé”

Trazado

- El algoritmo básico de trazado se llama *mark & sweep*
  - *mark phase* –> recorre el grafo de referencias marcando los objetos
  - *sweep phase* –> los objetos no marcados son finalized/freed

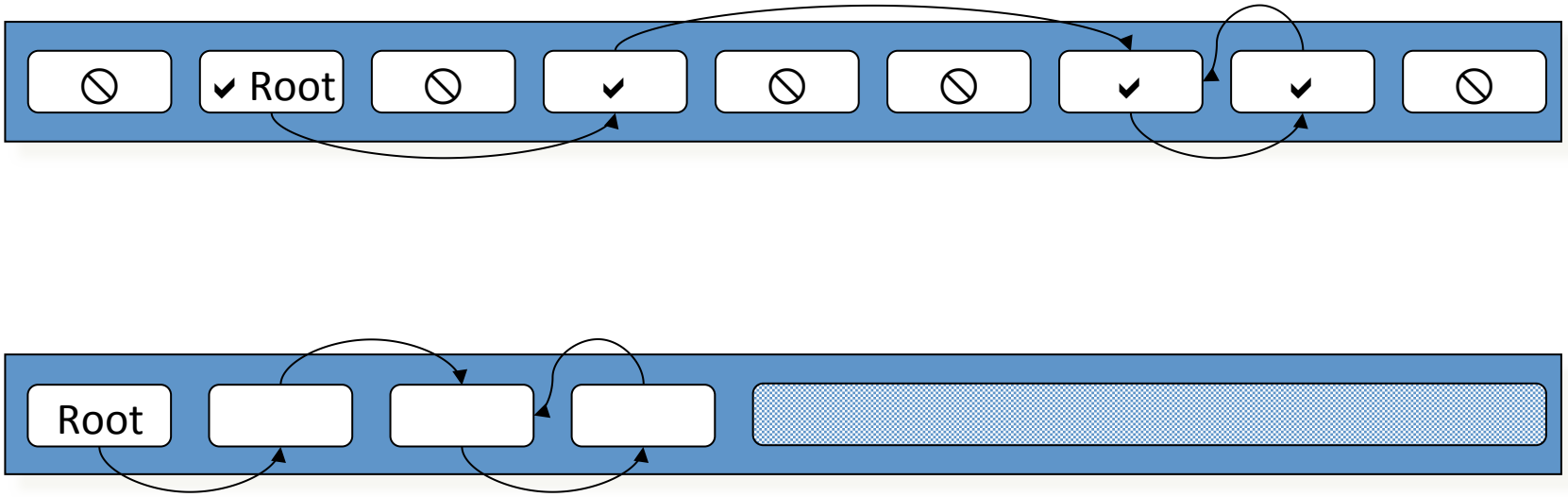
Trazado



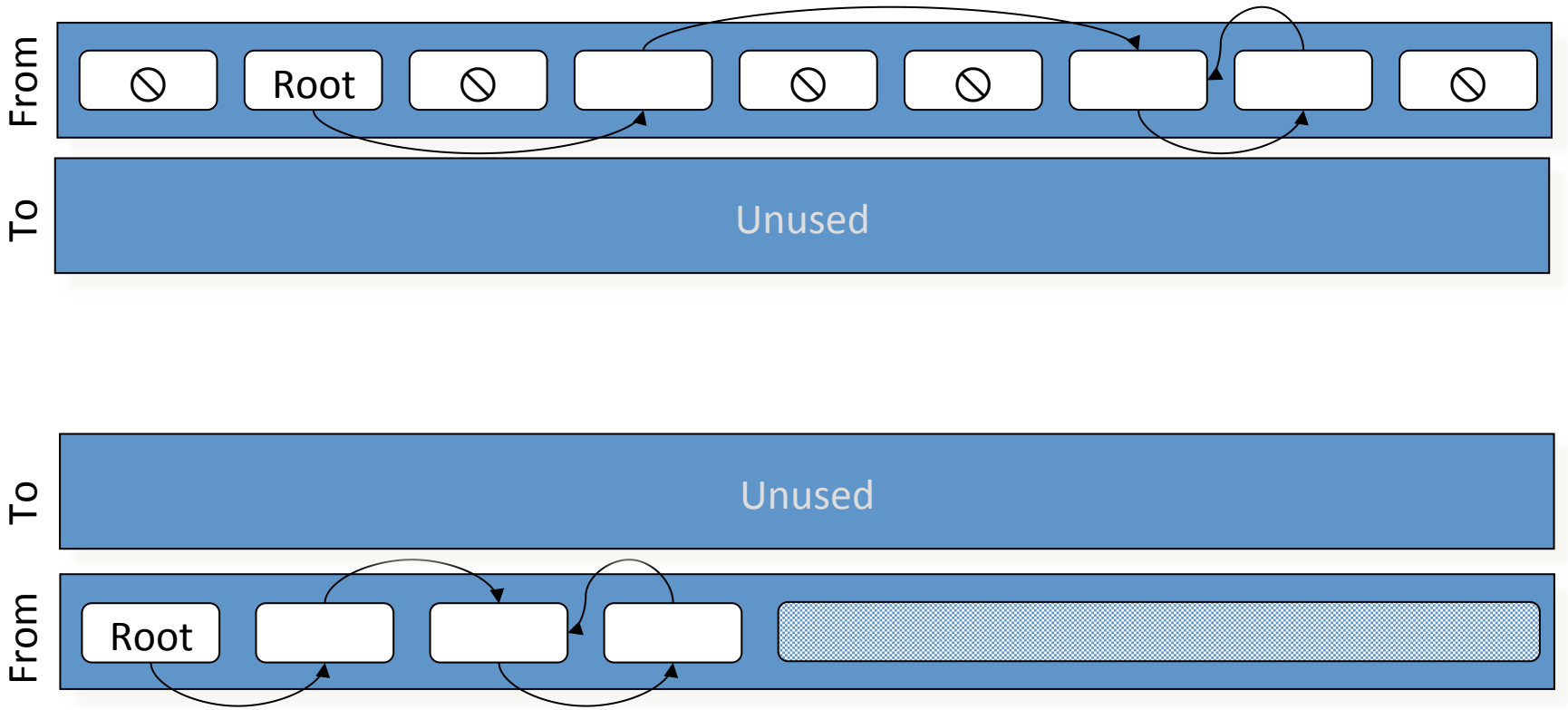
Mark-Sweep

- Mayor problema: la fragmentación de la memoria
- Dos estrategias:
  - *Mark-Compact Collector* – después de marcar mueve objetos vivos a áreas contiguas de memoria
  - *Copy Collector* – mueve objetos vivos a un nuevo área

Mark-Compact



Copy Collector



Características

- Mark-Compact es proporcional al tamaño del montículo
- Copy es proporcional al tamaño de los objetos vivos

Copiado

- Ventajas:
  - Muy rápido…si el conteo del objeto es bajo
  - No fragmenta
  - Rápida asignación
- Desventajas:
  - Dobla el espacio necesario y no es práctico para montículos grandes

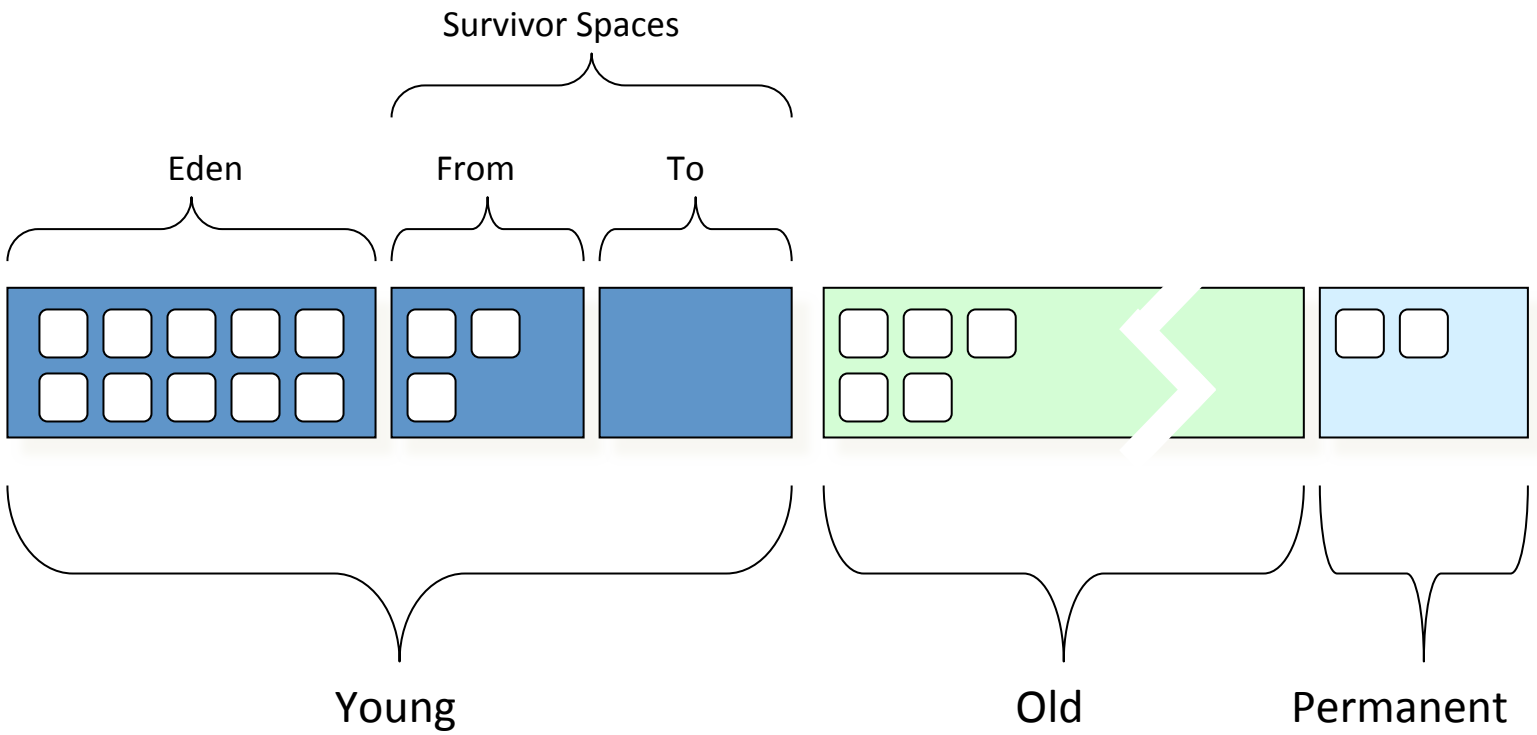
Observaciones generales

- Dos observaciones muy importantes:
  - La mayoría de los objetos mueren jóvenes
  - Hay pocas referencias de los objetos antiguos a los nuevos
- Se conocen como las hipótesis generacionales

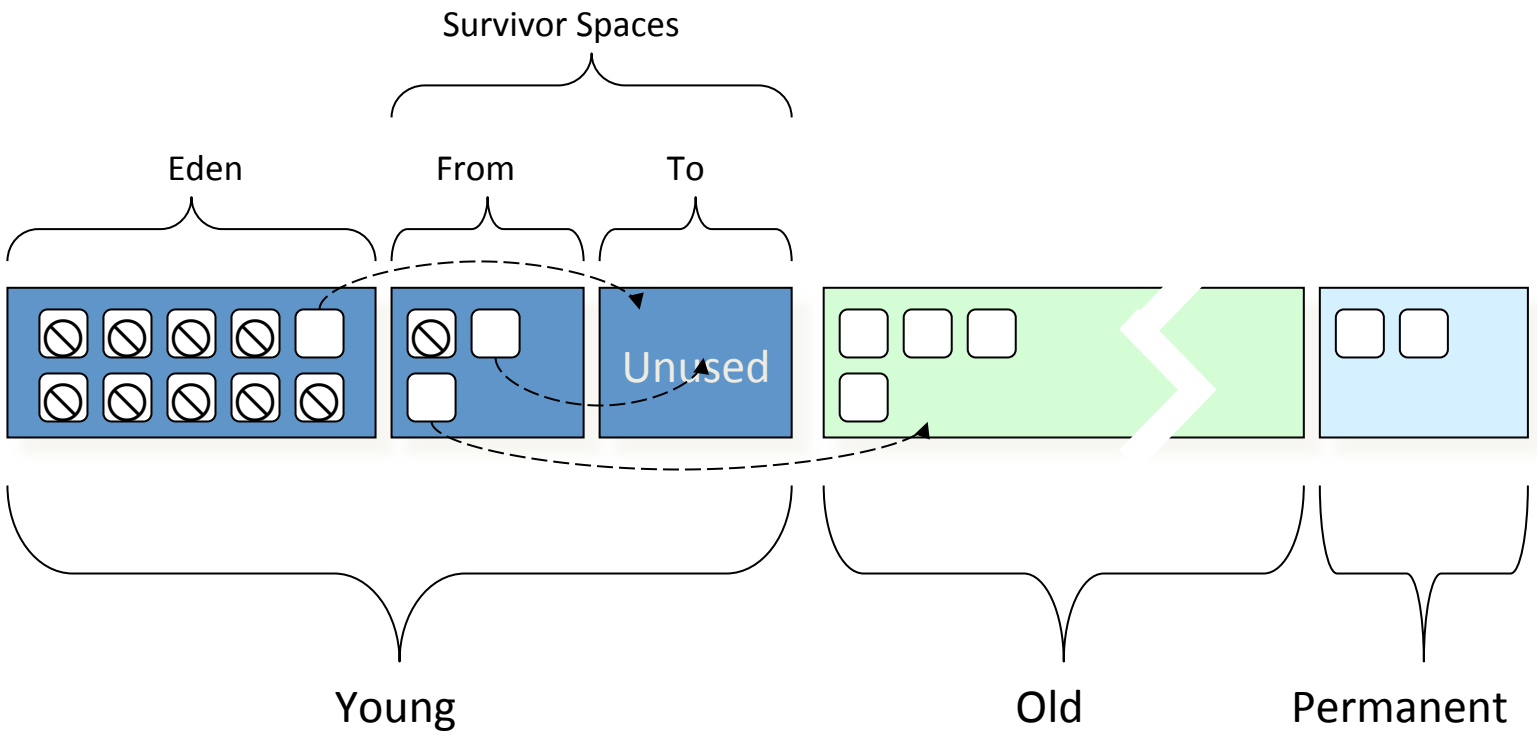
Generaciones

- El montículo se parte en generaciones, una joven y otra vieja
  - Generación joven – todos los objetos se crean aquí. La mayoría de la actividad de GC ocurre en este área (pero es rápida).
  - Vieja generación – objetos de larga duración, los jóvenes se pasan a este área a medida que envejecen. Poca actividad de GC (pero es más lenta)

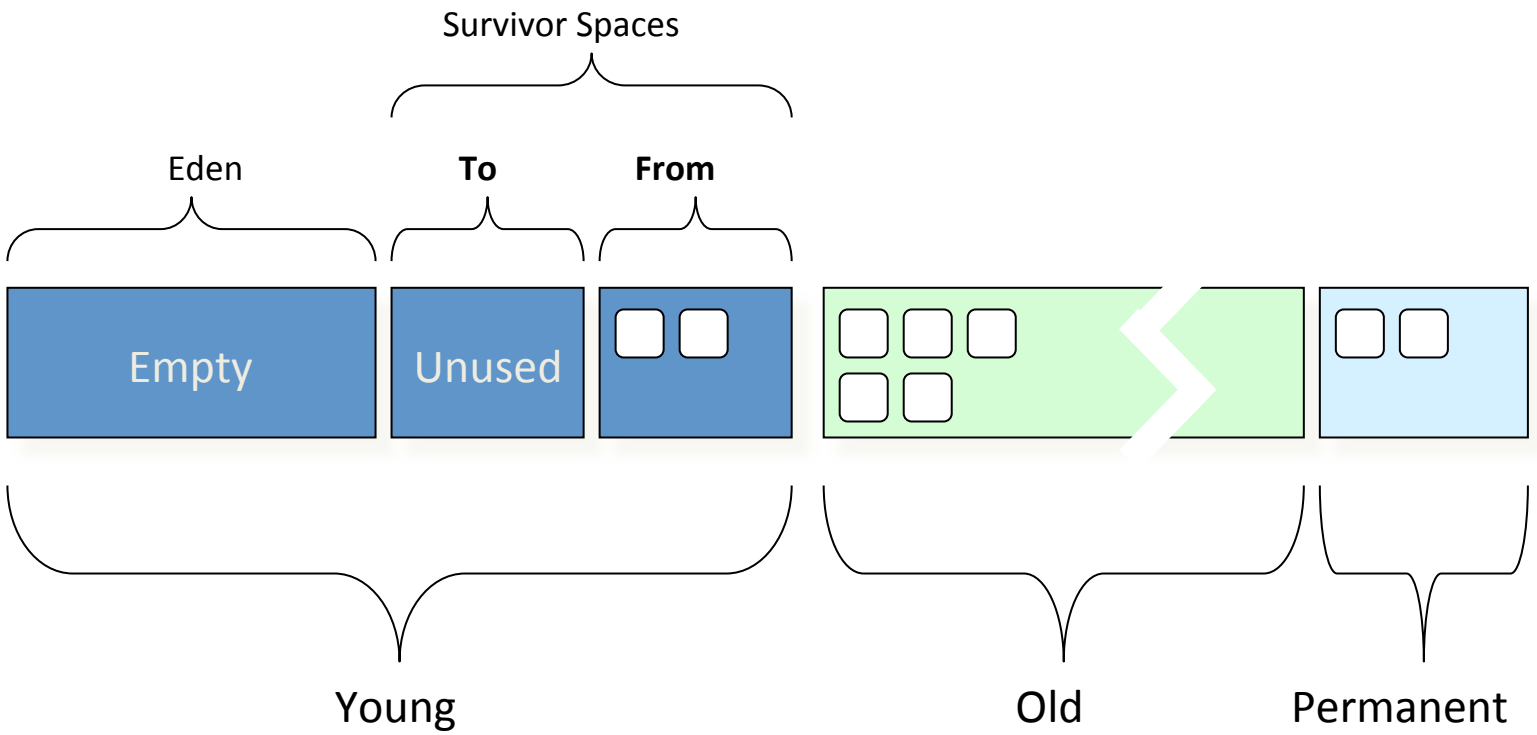
Estructura del montículo



Antes de un GC menor



Después de un GC menor



Detalles adicionales…

- Garbage Collection in Java
  http://www.cs.usm.maine.edu/talks/05/printezis.pdf
- A brief history of garbage collection
  http://www-128.ibm.com/developerworks/java/library/j-jtp10283/
- Garbage collection in the HotSpot JVM
  http://www-128.ibm.com/developerworks/java/library/j-jtp11253/
- Diagnosing a GC problem
  http://java.sun.com/docs/hotspot/gc1.4.2/example.html