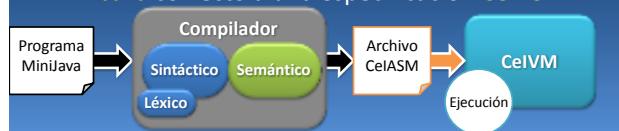


Generación de Código y Máquina Virtual de Compiladores e Intérpretes

Compiladores e Intérpretes 2014

Introducción

- La **Máquina Virtual de Compiladores e Intérpretes** (CeIVM) es un intérprete desarrollado por **Sebastián Escarza**
- Es capaz de **cargar, ensamblar y ejecutar** programas especificados en el lenguaje CelASM
- El compilador desarrollado **traducirá** un archivo **Minijava** correcto a una especificación **CelASM**



Introducción

- Gran parte de la **generación de código CelASM** para el archivo fuente se realizará como parte del **chequeo de sentencias** en el analizador **semántico**



Introducción

- En esta clase veremos:
 - Consideraciones de Implementación en Minijava
 - La estructura general de la CeIVM
 - Ejemplos del manejo de:
 - Expresiones y Asignaciones
 - Llamadas a métodos
 - Generación de Código a partir del AST en Expresiones

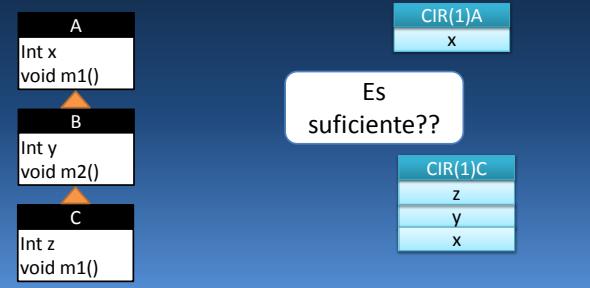
Consideraciones de Implementación en Minijava

Consideraciones: Datos Minijava

- En Minijava las **instancias** de las clases se **almacenan en el heap** (memoria dinámica).
- Estas **instancias se caracterizan** a través de una estructura llamada **Class Instance Record (CIR)**
- Es decir, un **CIR** es la forma de representar un objeto en ejecución
- Las **variables y parámetros** que sean **referencias** harán referencia a estos **CIRs**

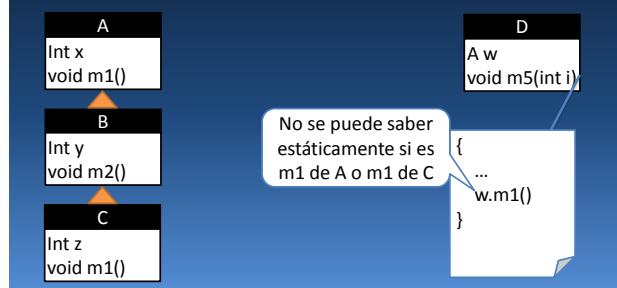
Consideraciones: Datos MiniJava

- Contenido de un CIR:



Consideraciones: Datos MiniJava

- Contenido de un CIR:



Consideraciones: Datos MiniJava

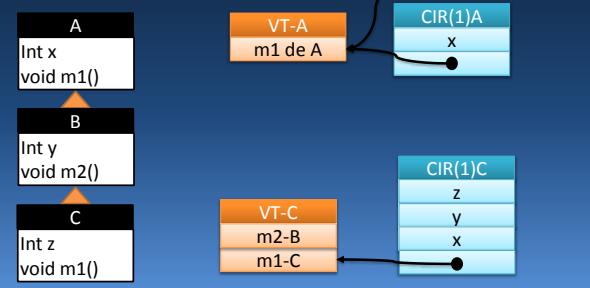
- En POO tenemos **vinculación dinámica** de código.
 - **No** siempre **podemos saber**, en tiempo de compilación, con qué **método** se corresponde una **llamada**.
 - Esta resolución debe hacerse en **ejecución** dependiendo del **objeto** sobre el cual se invoca el **método**.

Consideraciones: Datos MiniJava

- Para esto cada **CIR** va a tener una referencia a una **tabla** donde están las **direcciones de los métodos** que puede procesar.
 - Estas tablas son denominadas **Virtual Tables (VT)**
 - Por **cada clase** se creara una **VT** a la cual luego harán referencia las instancias (**CIR**) de tal clase.

Consideraciones: Datos MiniJava

- Contenido de un CIR:



Consideraciones: Datos MiniJava

- En RESUMEN:
 - Los **objetos en ejecución** se representan mediante **CIR**
 - Los **CIR** son dinámicos: se **crearan** a medida que se realiza la **computación** y sus **variables** de instancia podrán **cambiar de valores**.
 - Todo **CIR** contendrá una **referencia a la VT** correspondiente con su clase.
 - Las **VT** son **estáticas**, se crean antes de comenzar la ejecución.

Consideraciones: Registros de Activación en MiniJava

- La ejecución en MiniJava consiste en la **invocación a métodos y constructores**.
- Para manejar los **datos** que puede usar una **unidad en su activación** utilizamos los **registros de activación (RA)**.

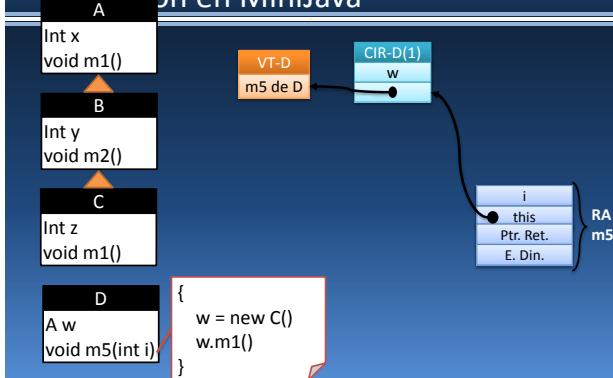
Consideraciones: Registros de Activación en MiniJava

- Por ejemplo, el **RA** de un método dinámico contendrá:
 - Valor de retorno
 - Parámetros
 - Variables Locales
 - Puntero de retorno
 - Enlace dinámico
 - Una referencia al objeto receptor (`this`)

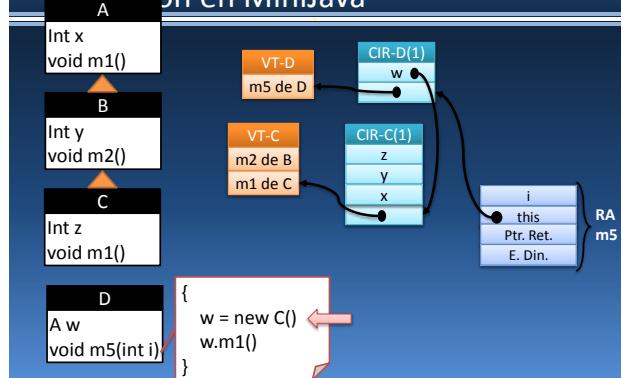
```
dynamic int m1(B p1, C p2){  
    varlocal int v1,v2;  
    ...  
}
```



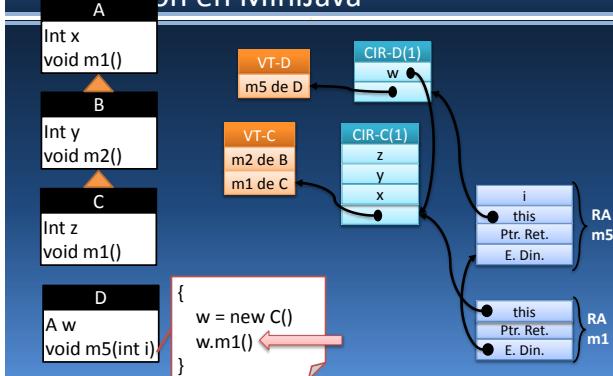
Consideraciones: Registros de Activación en MiniJava



Consideraciones: Registros de Activación en MiniJava



Consideraciones: Registros de Activación en MiniJava

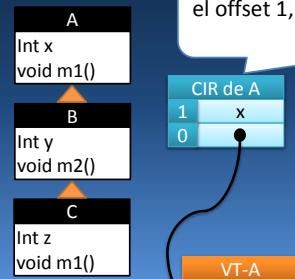


Consideraciones: CIRs, VTs, y RA

- Para poder acceder a las distintas **variables/parámetros** en ejecución es importante determinar en compilación cuál será su **offset** dentro de la estructura que los contiene
 - Offset de **Parámetros y Variables Locales** en el **RA**
 - Offset de **Variables de instancia** en el **CIR**
 - OJO! Si bien las variables de un ancestro no son visibles, éstas forman parte del CIR

Consideraciones: CIRs, VTs, y RA

- Por ejemplo:



La primera variable siempre empieza con el offset 1, ya que el 0 se reserva para el enlace a la VT

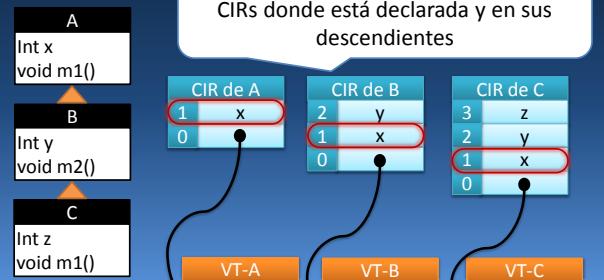
Consideraciones: CIRs, VTs, y RA

- Por ejemplo: Todas las variables del ancestro estarán en el CIR por más que no sean visibles (pueden ser usadas por los métodos heredados)



Consideraciones: CIRs, VTs, y RA

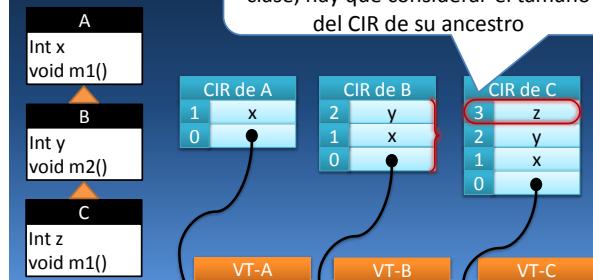
- Por ejemplo:



Una variable tiene el mismo offset en el CIRs donde está declarada y en sus descendientes

Consideraciones: CIRs, VTs, y RA

- Por ejemplo:



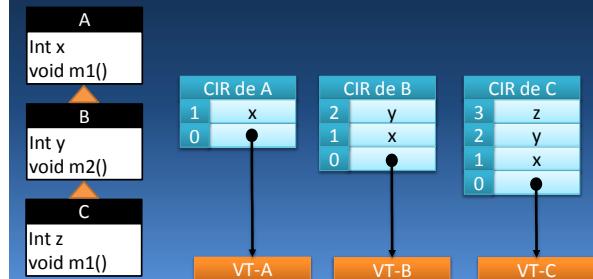
Por lo tanto, para determinar el offset de las variables declaradas en una clase, hay que considerar el tamaño del CIR de su ancestro

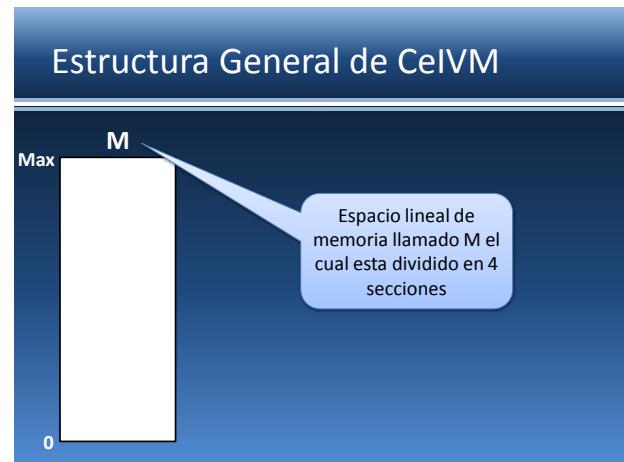
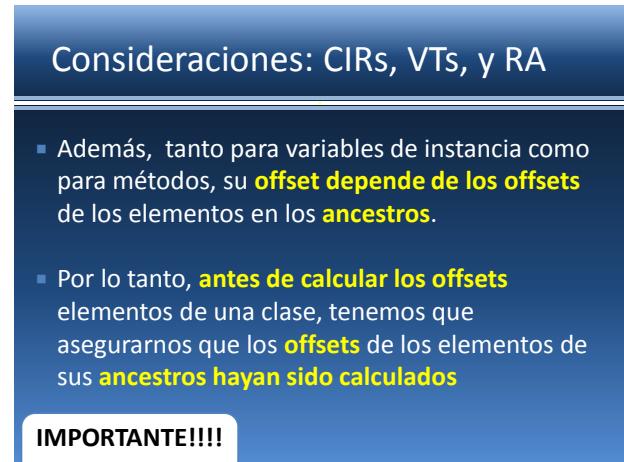
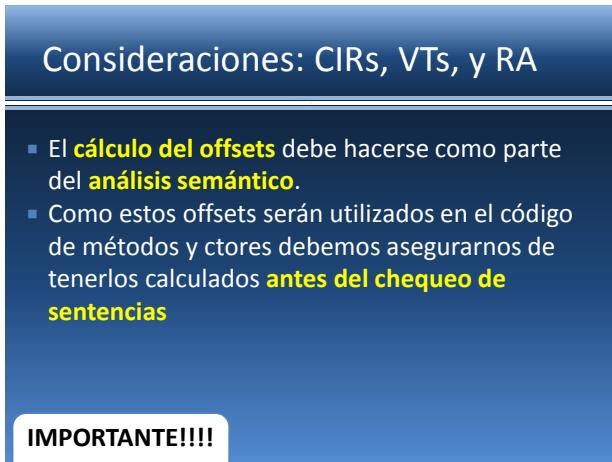
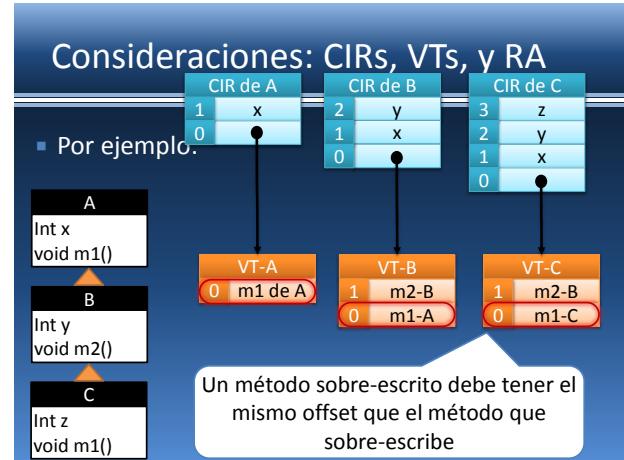
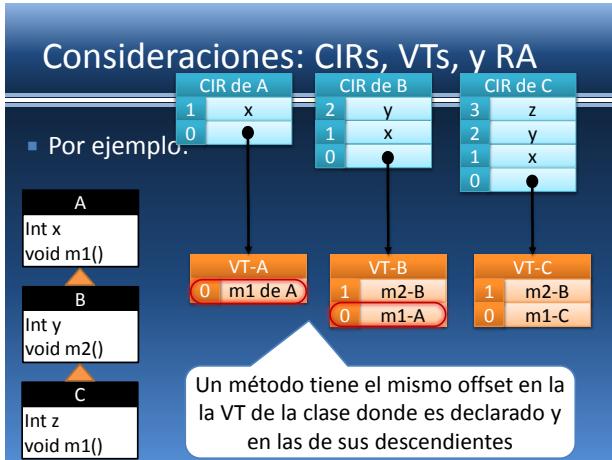
Consideraciones: CIRs, VTs, y RA

- Por otra parte, para poder acceder correctamente a la VT para obtener las **etiquetas de los métodos** a ejecutar, necesitamos conocer el **offset del método** dentro de la VT en tiempo de compilación

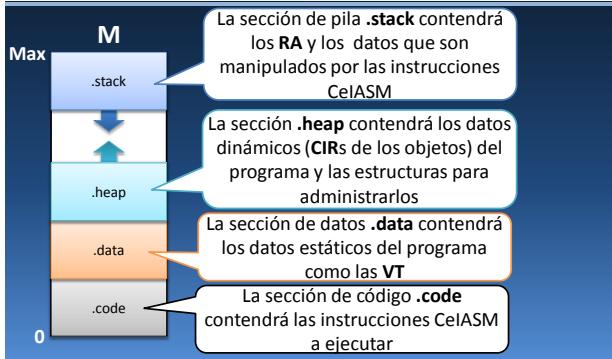
Consideraciones: CIRs, VTs, y RA

- Por ejemplo:

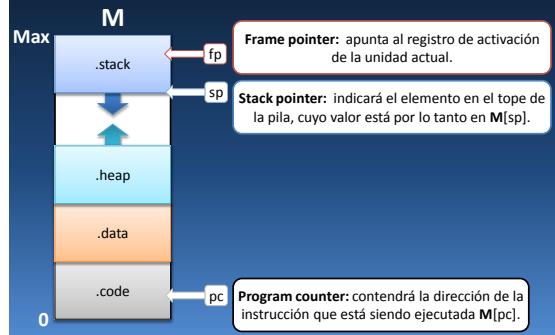




Estructura General de CelVM



Estructura General de CelVM



Estructura General de CelVM

- Una vez **cargado el programa** en la sección **.code** y los **datos estáticos** en **.data**, comienza la ejecución
- El funcionamiento de la máquina consiste en **ejecutar una a una las instrucciones** indicadas por el registro **pc** hasta:
 - Encontrar la instrucción de detención (**HALT**), o
 - Hasta la ocurrencia de algún **Error**.

Estructura General de CelVM

- El **formato general** de las instrucciones de CelASM es:
- label : NOMBREINST Arg1,Arg2,...,Argn
- Sólo podrá haber **una instrucción por línea**

Instrucciones y Ejemplos CelVM

Expresiones en la CelVM

- En la CelVM utilizaremos la **pila** (sección **.stack**) para ir evaluando las **expresiones**
- Dada una expresión de la forma **E1 + E2**
 - Primero se evaluarán los valores de **E1** y **E2**, los cuales quedarán **apilados**
 - Luego sobre estos valores calculados se **aplicará el operador +**.
- Las **expresiones** se resuelven siguiendo un **orden posfijo**

Expresiones en la CelVM

- Los **operadores**, al ser aplicados, toman los **operandos del tope** de la pila y los transforman en el resultado.
- Consideremos las siguientes instrucciones CelASM
 - PUSH k** : Apila la constante k en el tope de la pila
 - ADD** : Suma el tope y el tope-1 de la pila, almacena el resultado en el tope-1 y baja el tope
 - MUL** : Multiplica el tope y el tope-1 de la pila, almacena el resultado en el tope-1 y baja el tope

Expresiones en la CelVM

- Por Ejemplo $4 * 2 + 3 * 2$
- en Posfijo: 4 2 * 3 2 * +

PUSH 4

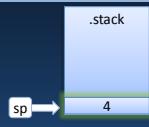


Expresiones en la CelVM

- Por Ejemplo $4 * 2 + 3 * 2$
- en Posfijo: 4 2 * 3 2 * +

PUSH 4 ←pc

PUSH 2



Expresiones en la CelVM

- Por Ejemplo $4 * 2 + 3 * 2$
- en Posfijo: 4 2 * 3 2 * +

PUSH 4

PUSH 2 ←pc

MUL



Expresiones en la CelVM

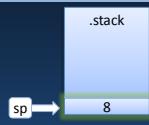
- Por Ejemplo $4 * 2 + 3 * 2$
- en Posfijo: 4 2 * 3 2 * +

PUSH 4

PUSH 2

MUL ←pc

PUSH 3



Expresiones en la CelVM

- Por Ejemplo $4 * 2 + 3 * 2$
- en Posfijo: 4 2 * 3 2 * +

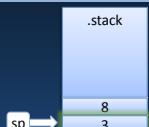
PUSH 4

PUSH 2

MUL

PUSH 3 ←pc

PUSH 2



Expresiones en la CelVM

- Por Ejemplo $4*2+3*2$
- en Posfijo: $4\ 2\ *\ 3\ 2\ *\ +$

PUSH 4
PUSH 2
MUL
PUSH 3
PUSH 2 
MUL



Expresiones en la CelVM

- Por Ejemplo $4*2+3*2$
- en Posfijo: $4\ 2\ *\ 3\ 2\ *\ +$

PUSH 4
PUSH 2
MUL
PUSH 3
PUSH 2
MUL 
ADD



Expresiones en la CelVM

- Por Ejemplo $4*2+3*2$
- en Posfijo: $4\ 2\ *\ 3\ 2\ *\ +$

PUSH 4
PUSH 2
MUL
PUSH 3
PUSH 2
MUL
ADD 



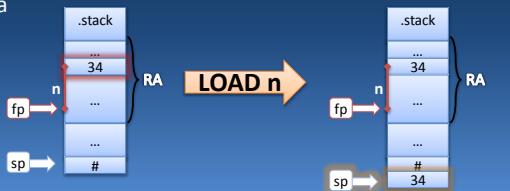
Asignaciones y acceso a Variables

- Las **variables**, en general, estarán asociadas a **registros de activación** o a **CIRs**
- En MiniJava tenemos dos **tipos de variables**:
 - **Variábeles de instancia**: tendrán un lugar en los **CIR** de los objetos a los que pertenecen
 - **Parámetros/Variables Locales**: tendrán un lugar en los **RA** de los métodos a los cuales pertenecen

Asignaciones y acceso a Variables

- Instrucciones para **usar y almacenar parámetros/variables locales** (Con **n** su desplazamiento en el RA actual).

LOAD n : apila el valor de un parámetro/variable local en la pila



Asignaciones y acceso a Variables

- Instrucciones para **usar y almacenar parámetros/variables locales** (Con **n** su desplazamiento en el RA actual).

STORE n : almacena el valor del tope de la pila en un parámetro/variable local. (**saca el valor** de la pila luego)



Asignaciones y acceso a Variables

- Instrucciones para **usar y almacenar variables de instancia** (Con **n** su desplazamiento en el CIR).

LOADREF n : apila el valor de una variable de instancia en la pila, en base a la referencia al objeto (CIR) del tope de la pila (lo reemplaza)



Asignaciones y acceso a Variables

- Instrucciones para **usar y almacenar variables de instancia** (Con **n** su desplazamiento en el CIR).

STOREREF n : almacena el valor del tope de la pila en una variable de instancia del objeto en tope-1 de la pila. (desapilando el valor y la referencia al objeto luego)



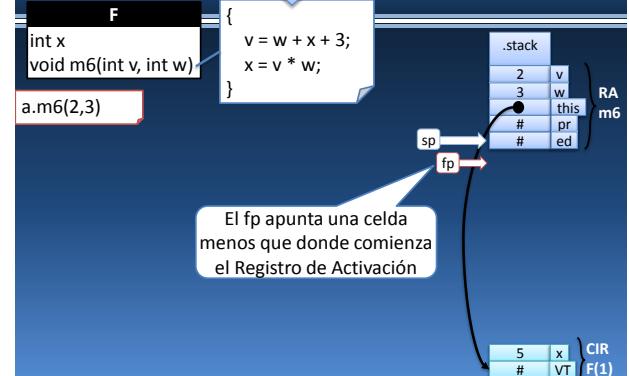
Asignaciones y acceso a Variables

- Además utilizaremos las siguientes instrucciones auxiliares:

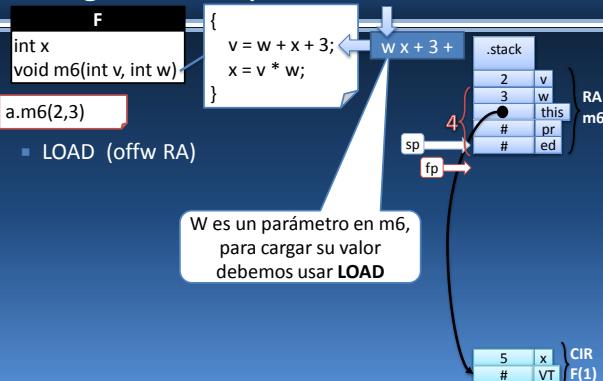
DUP : Duplica el tope de la pila

SWAP : Intercambia el tope con el tope-1

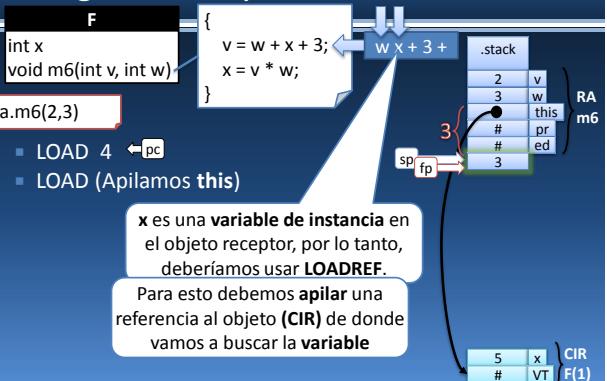
Asignaciones y acceso a Variables

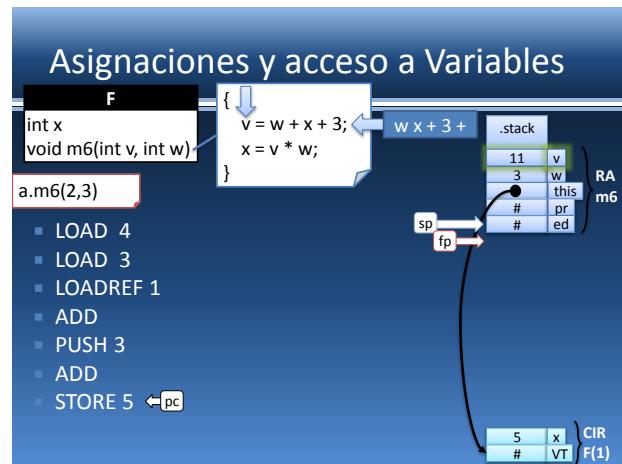
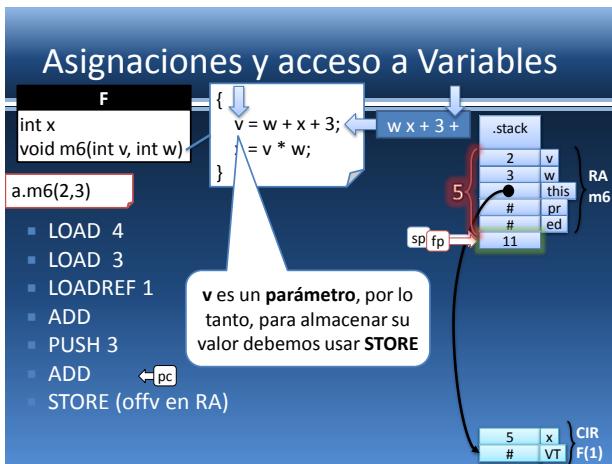
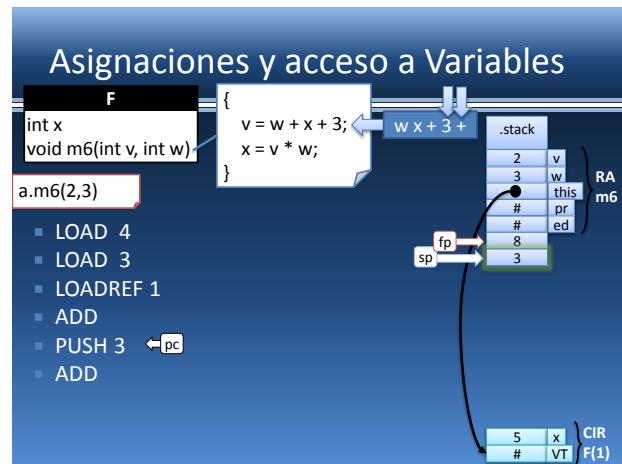
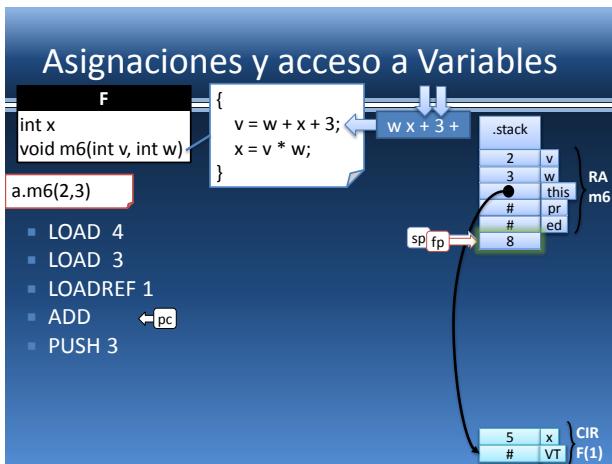
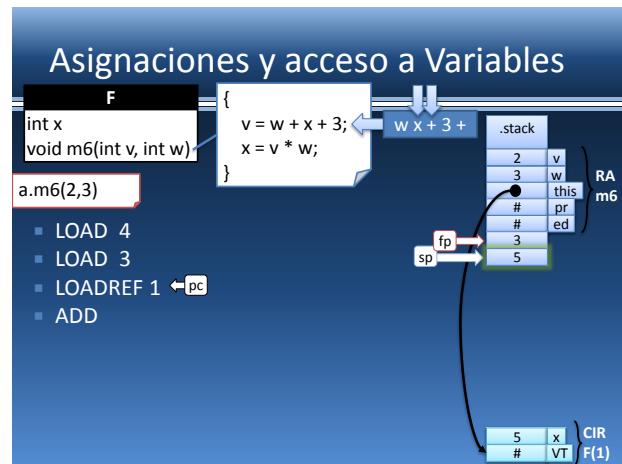
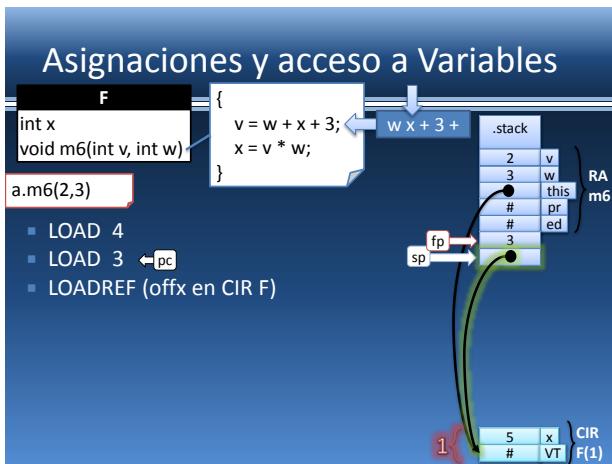


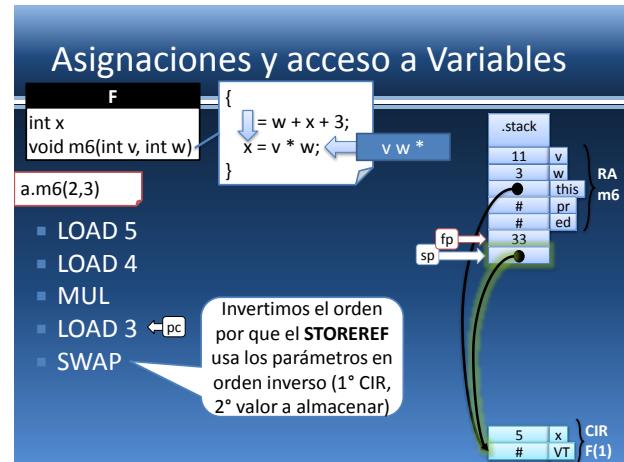
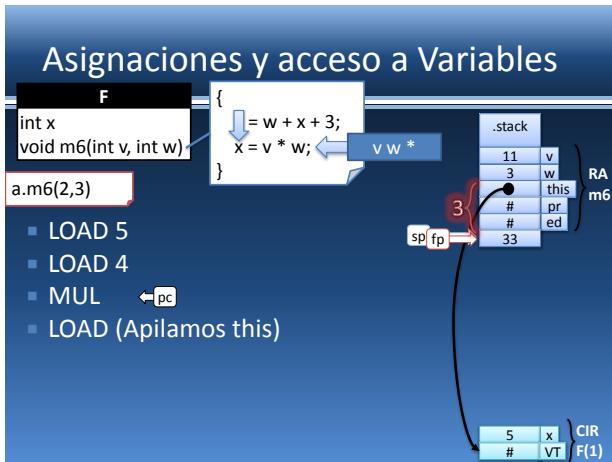
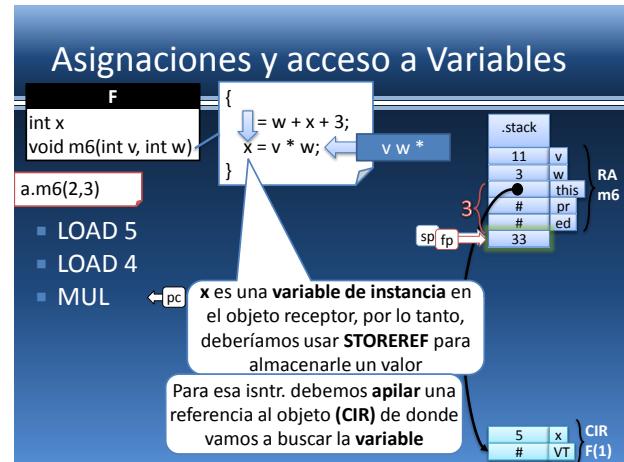
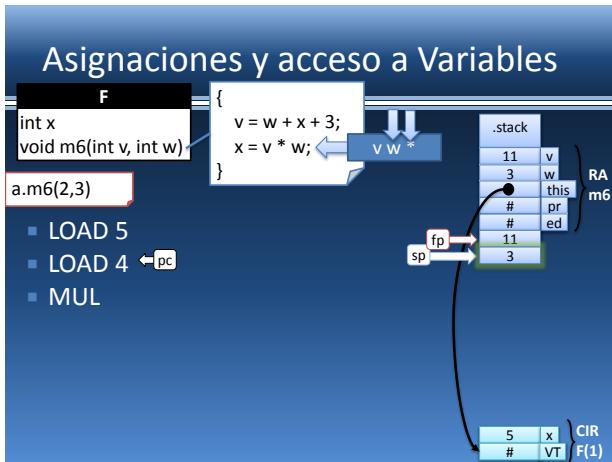
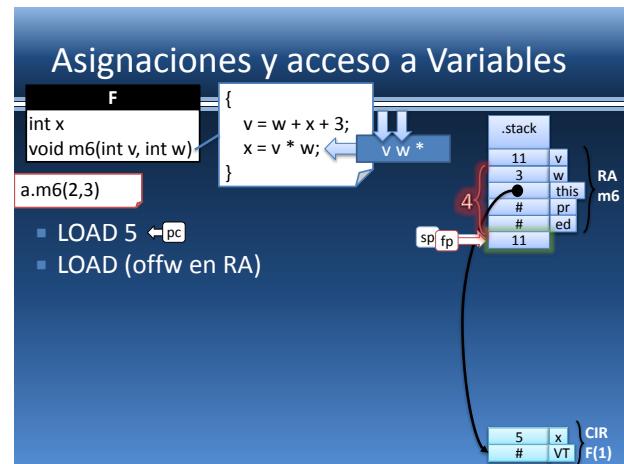
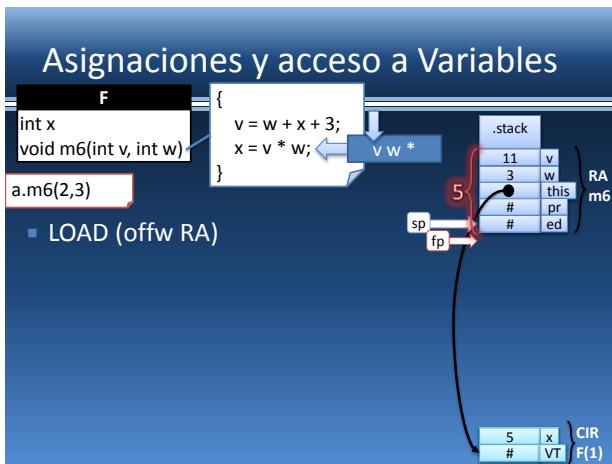
Asignaciones y acceso a Variables

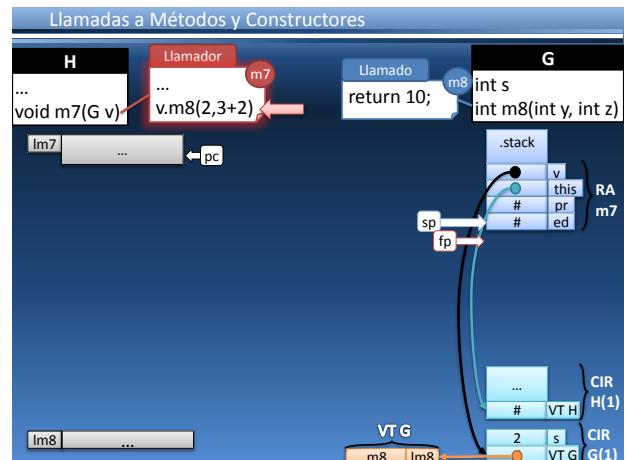
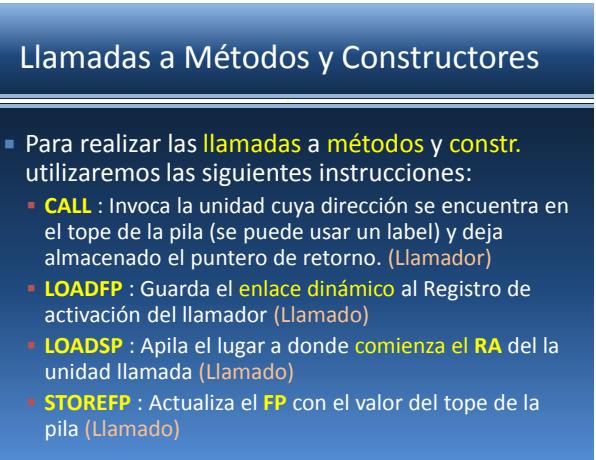
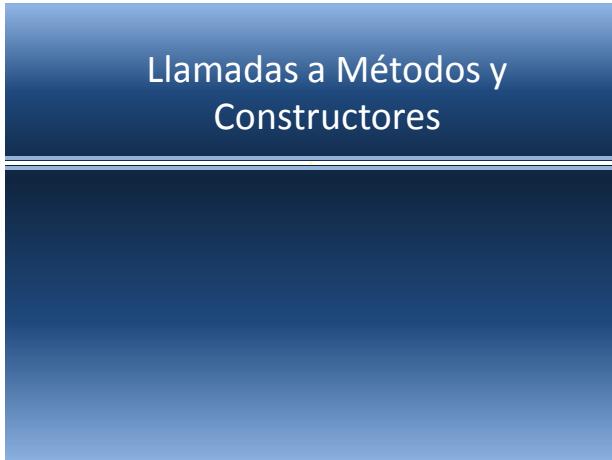
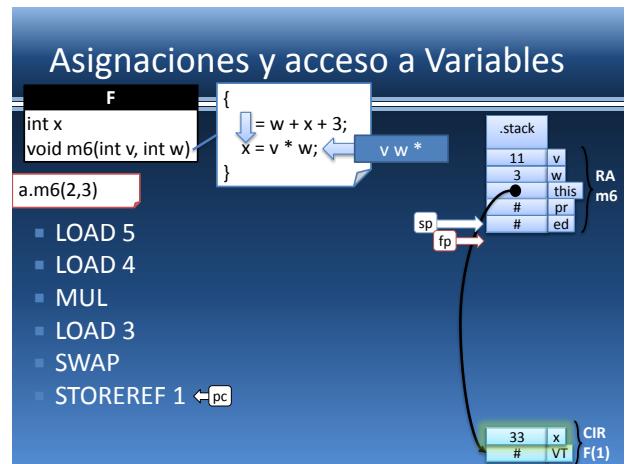
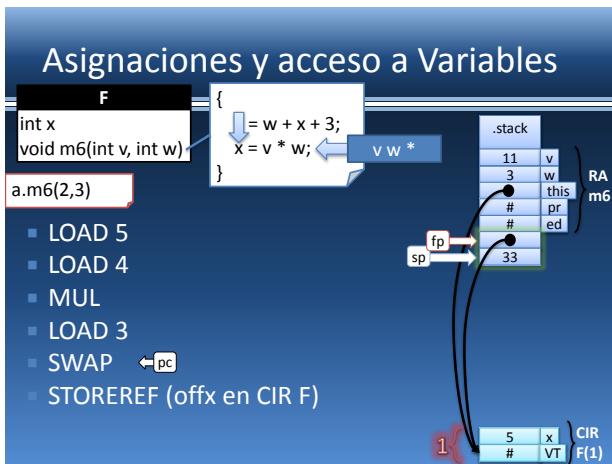


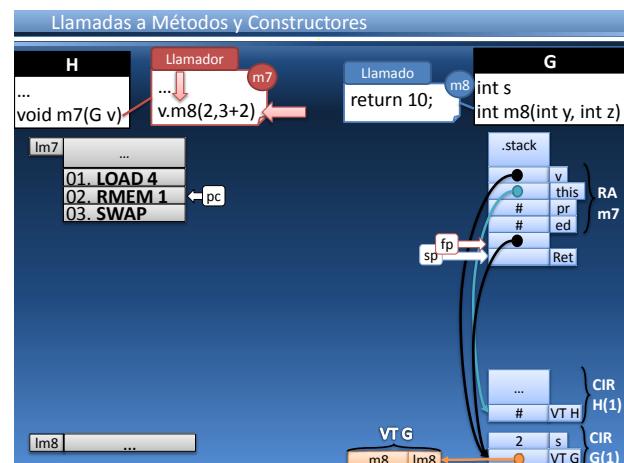
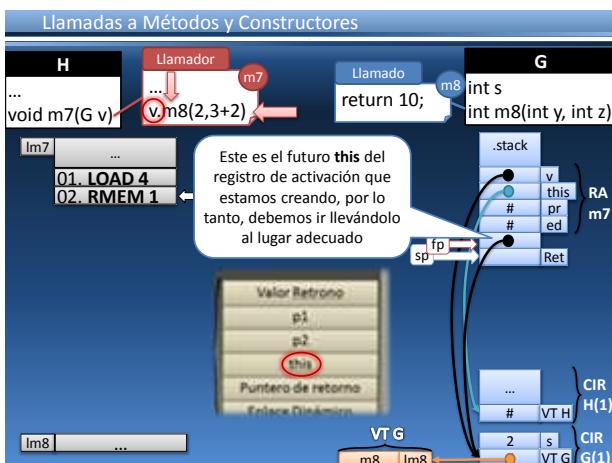
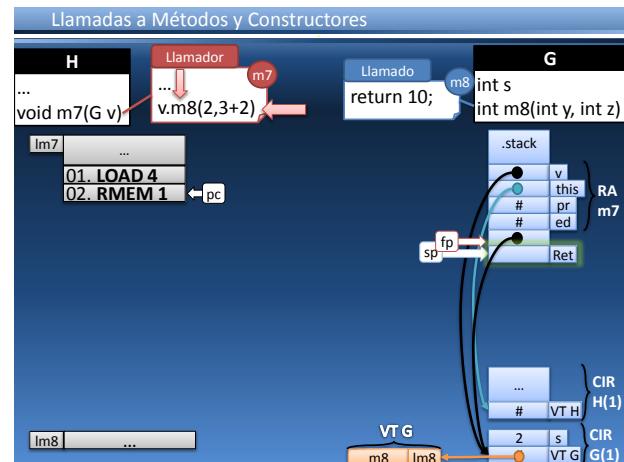
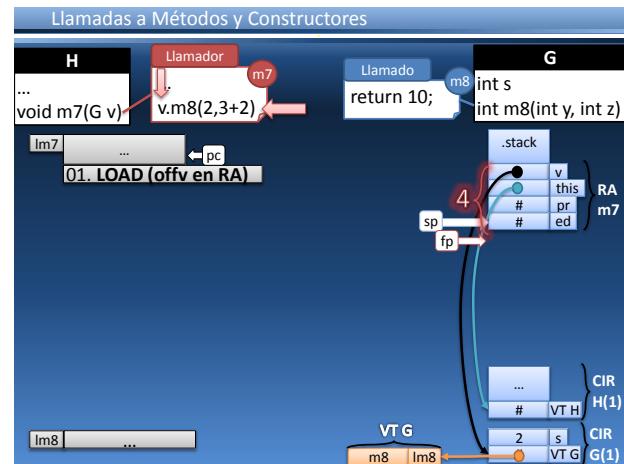
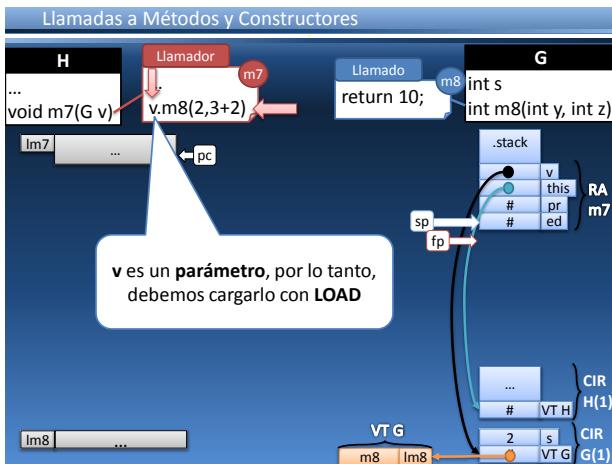
Asignaciones y acceso a Variables

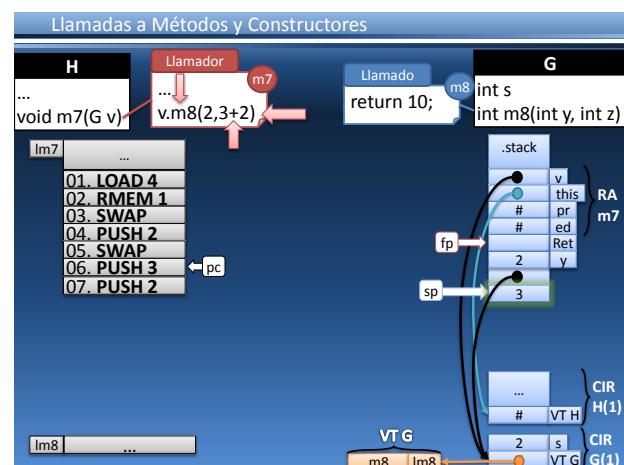
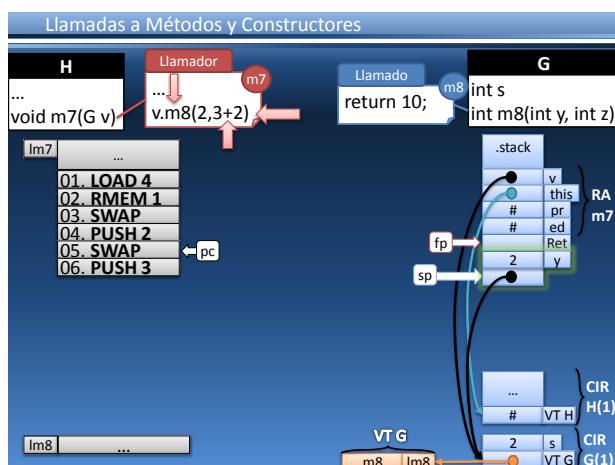
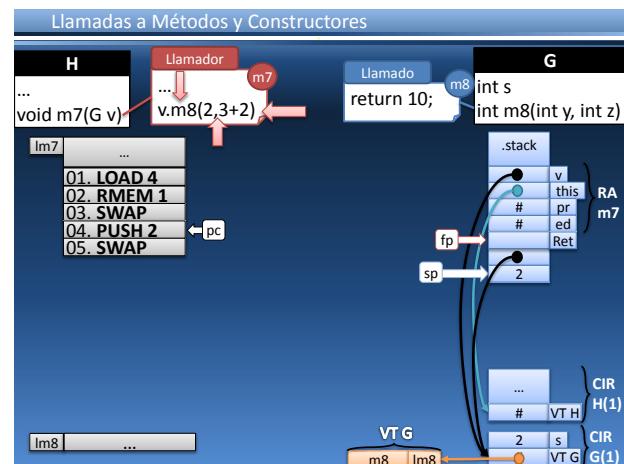
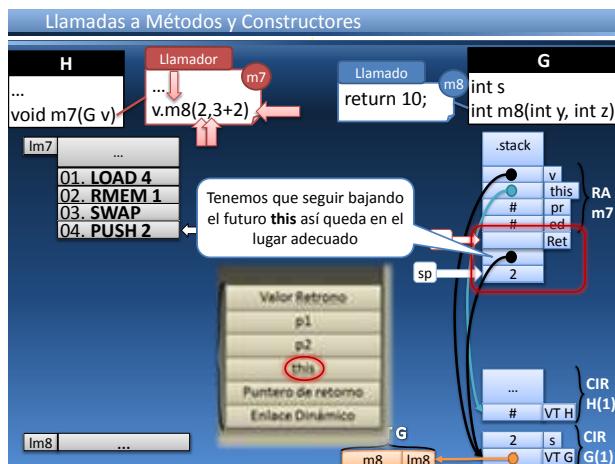
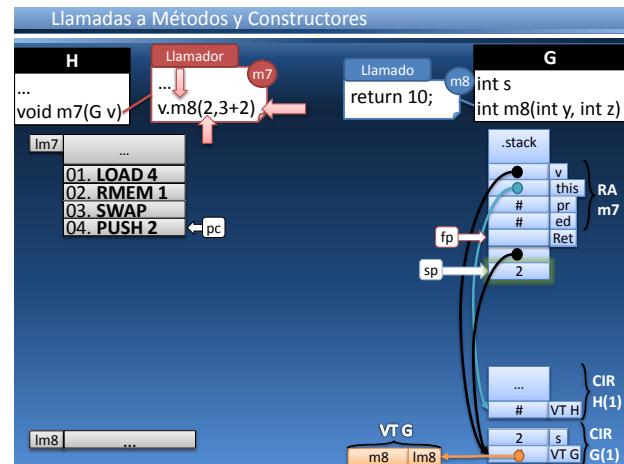
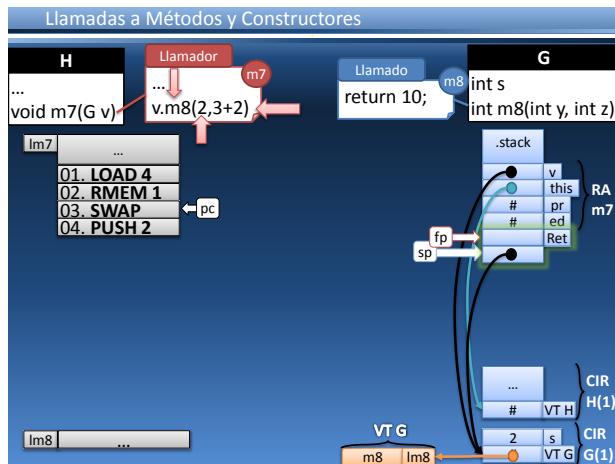


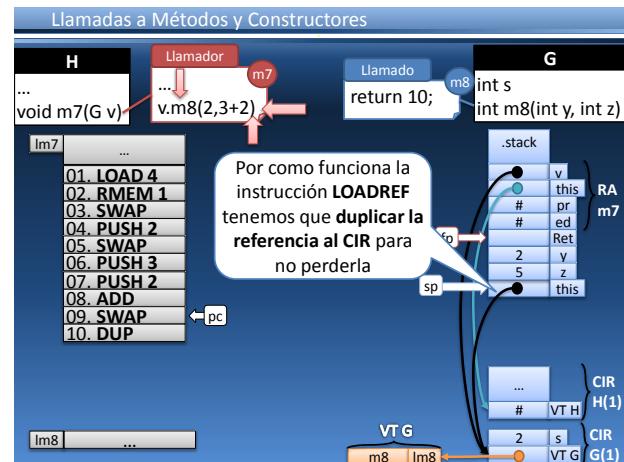
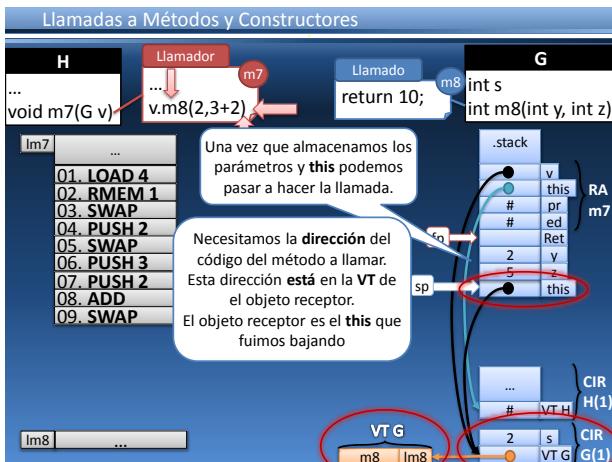
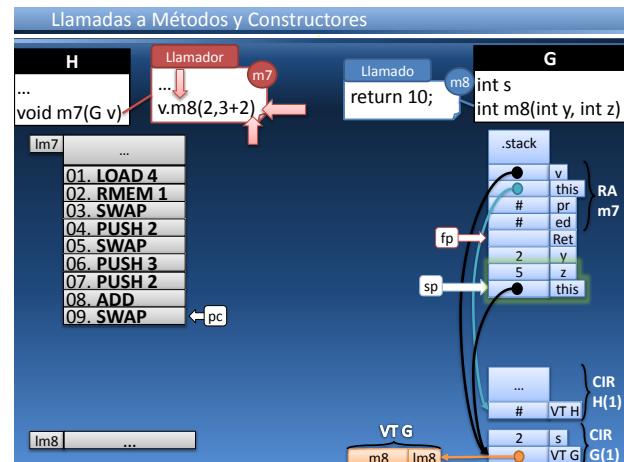
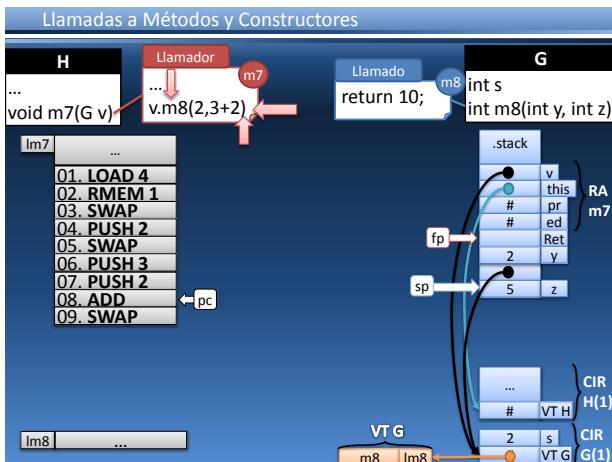
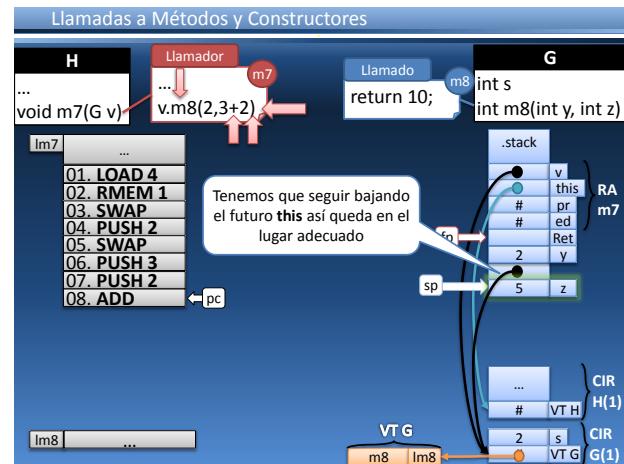
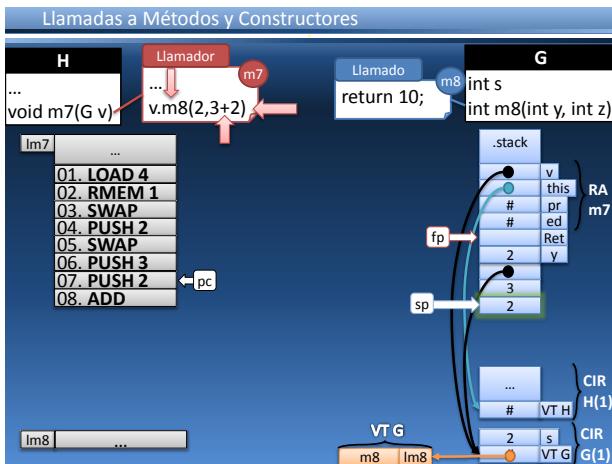


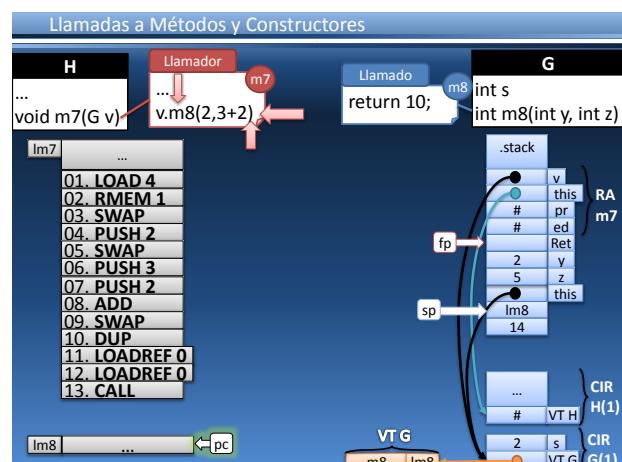
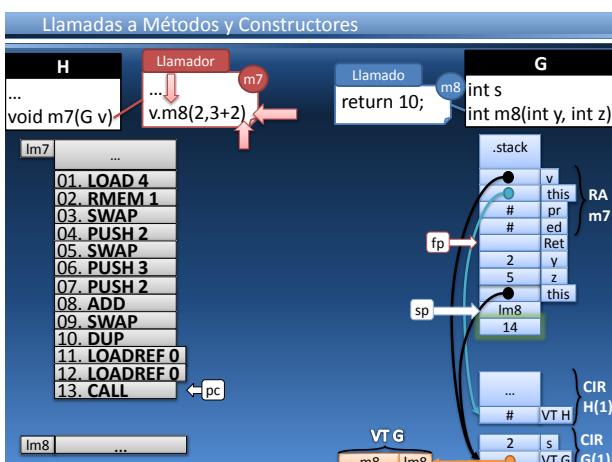
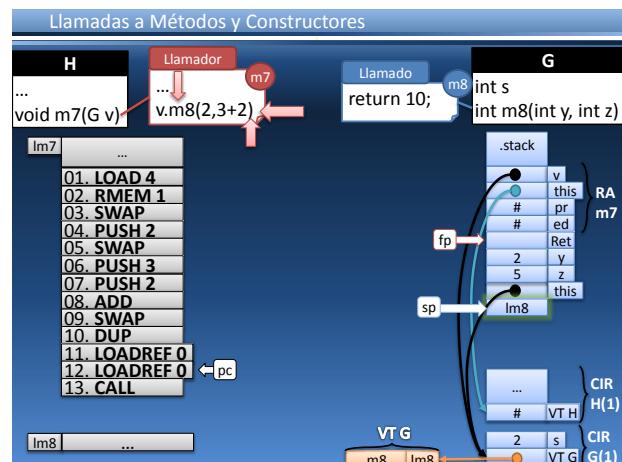
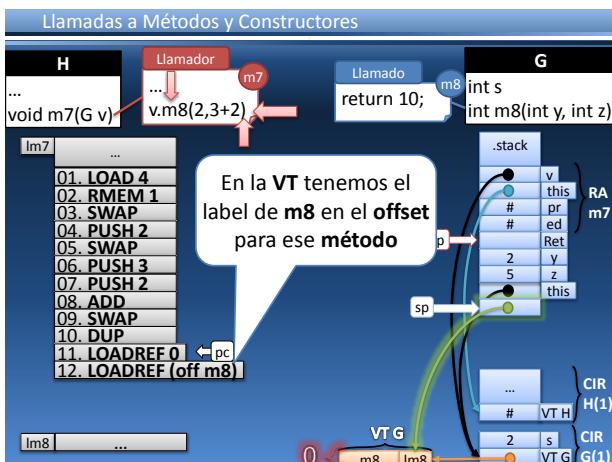
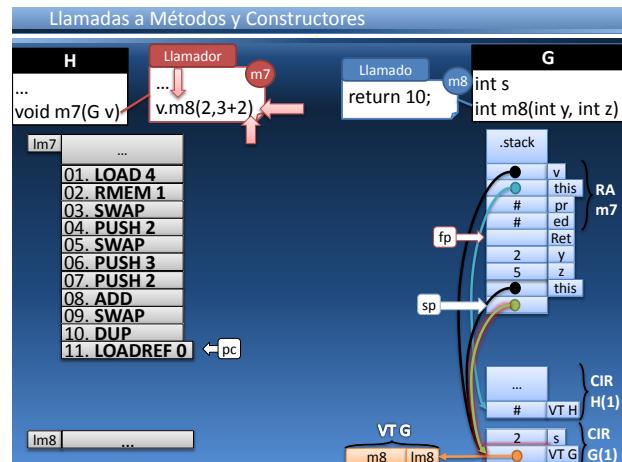


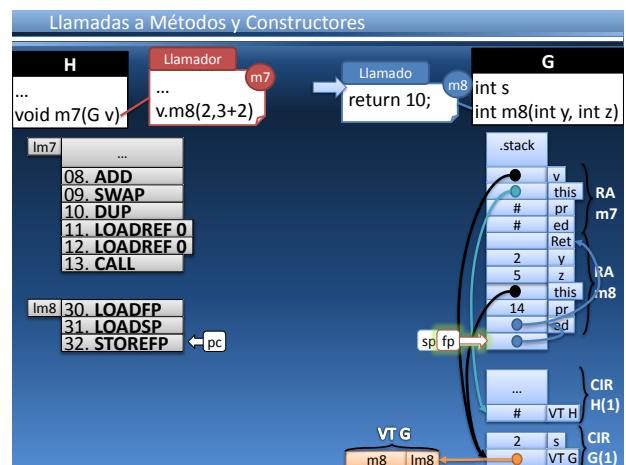
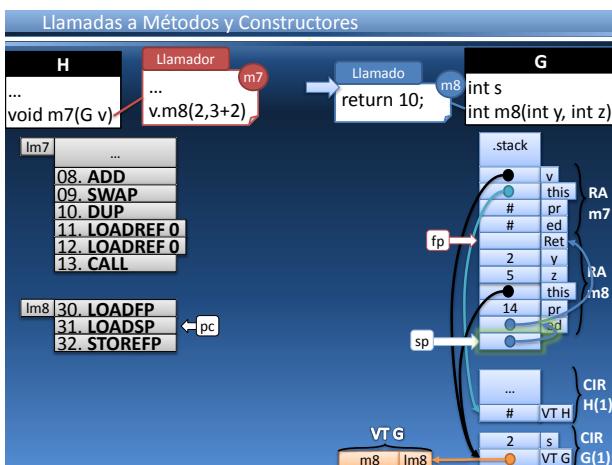
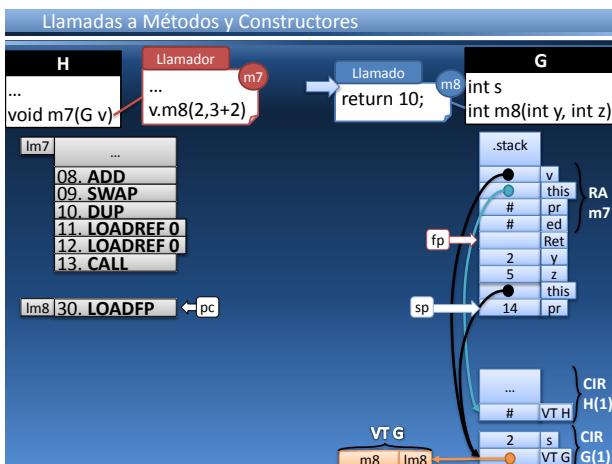
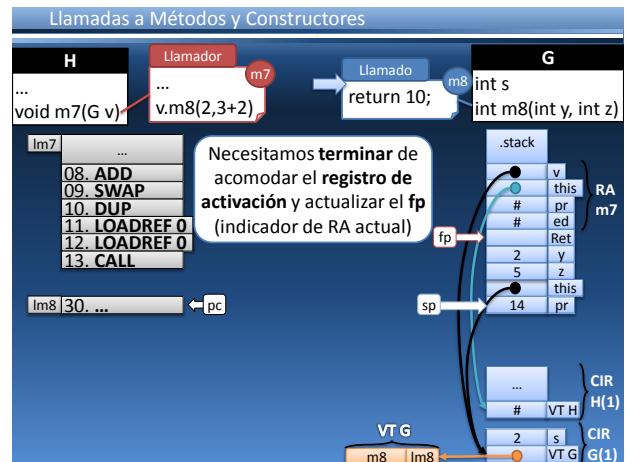
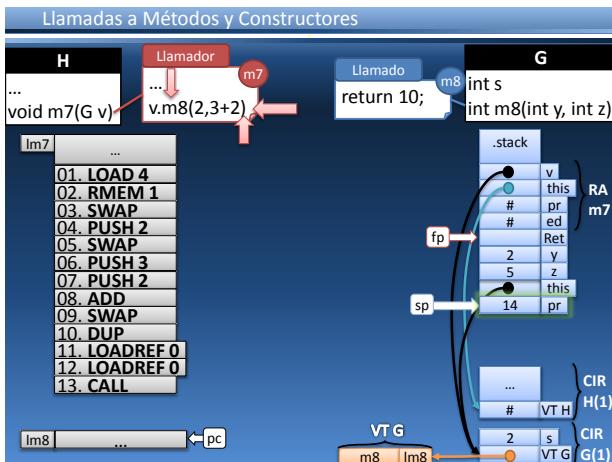


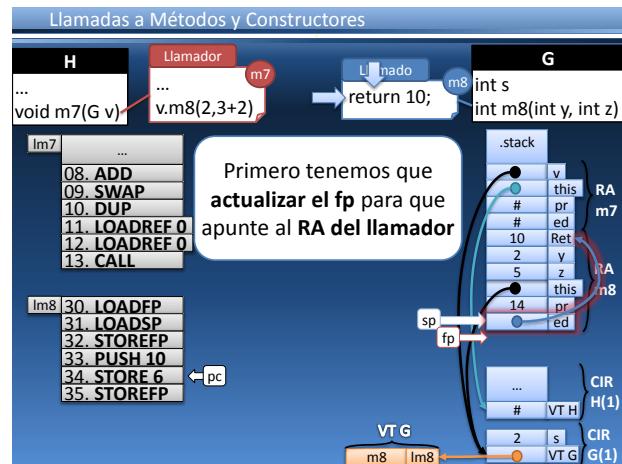
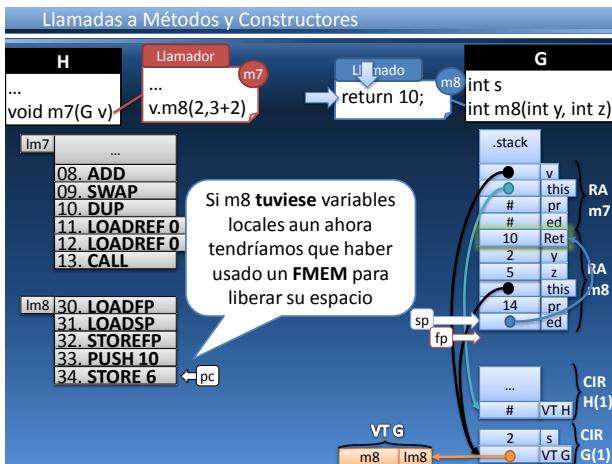
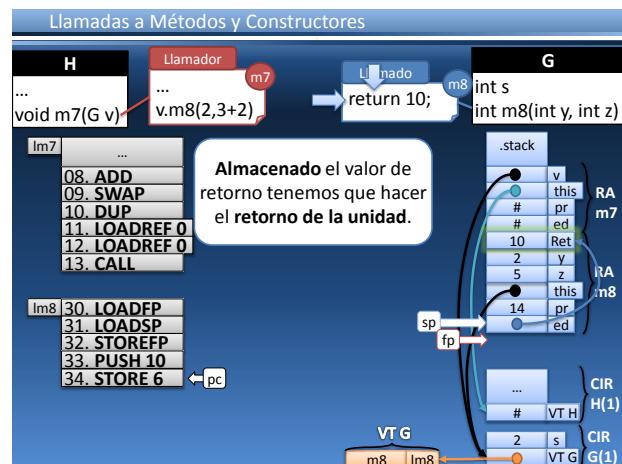
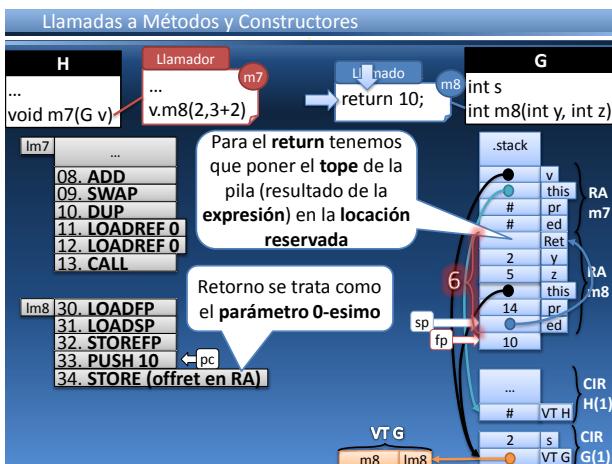
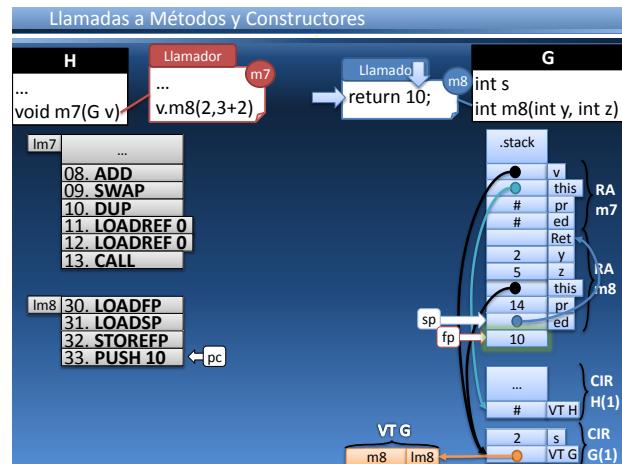
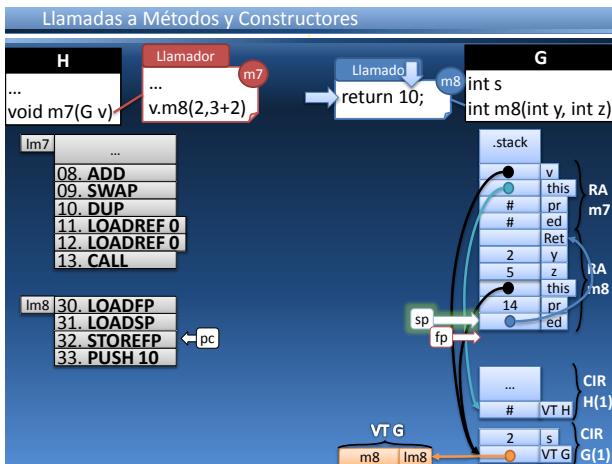


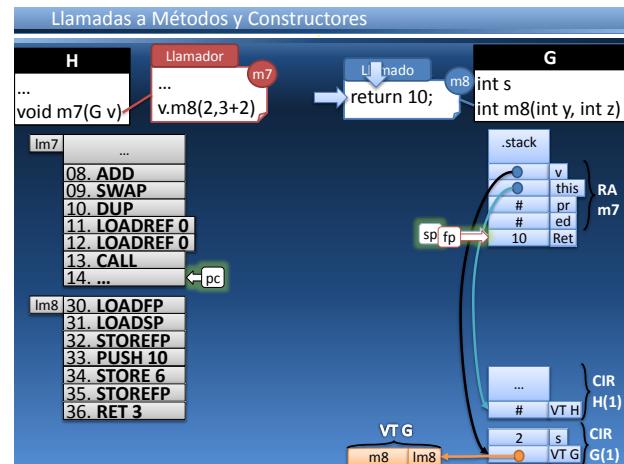
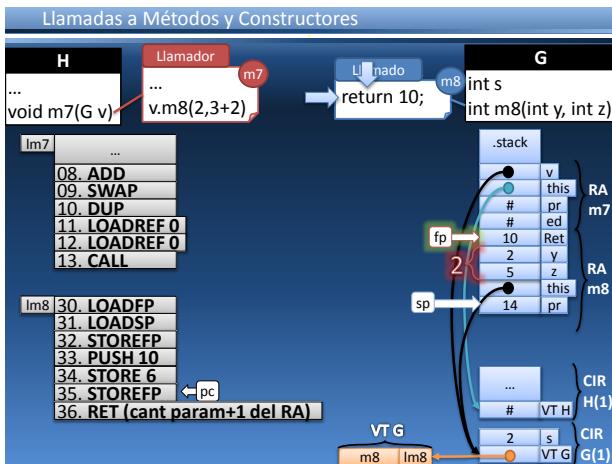






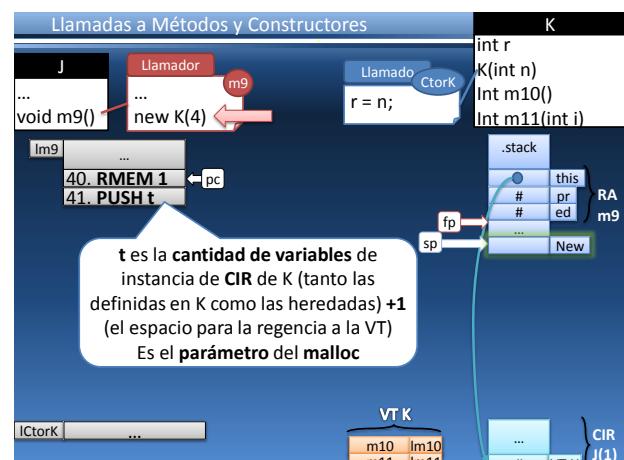
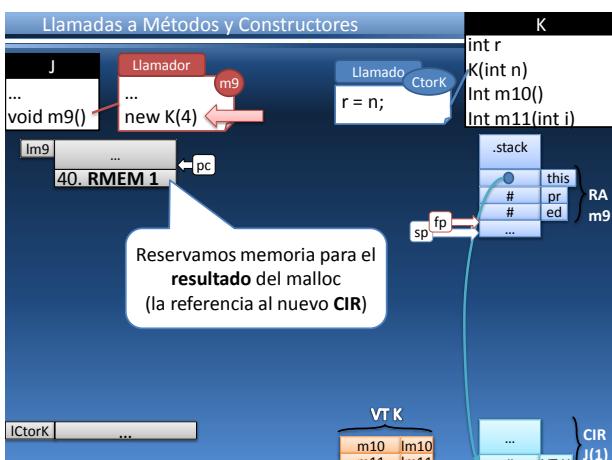
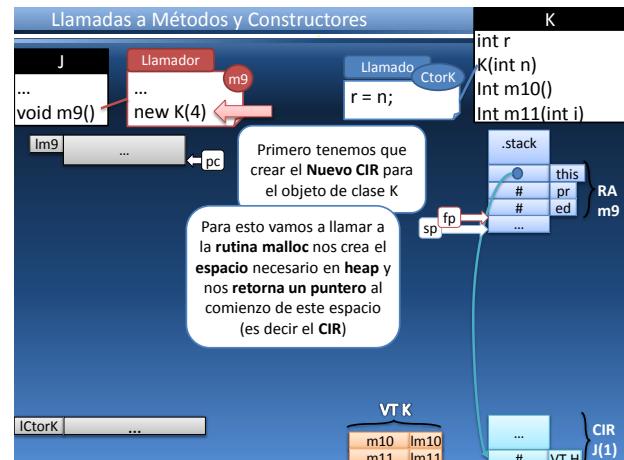


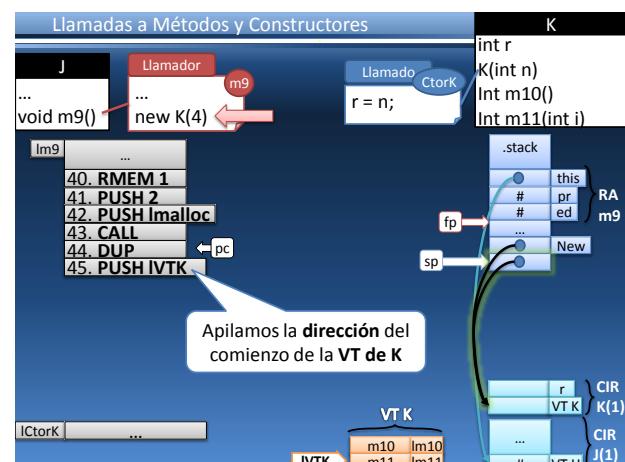
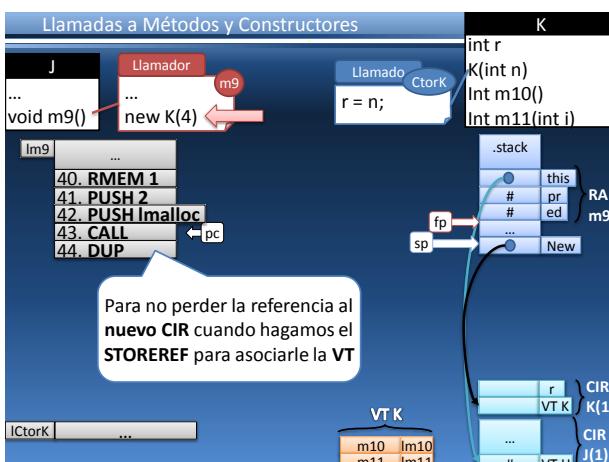
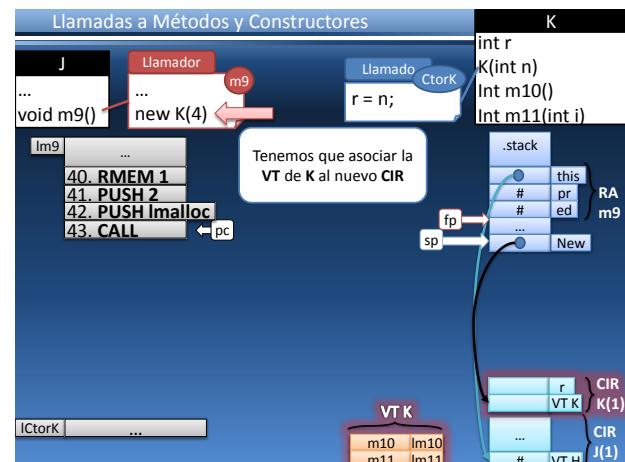
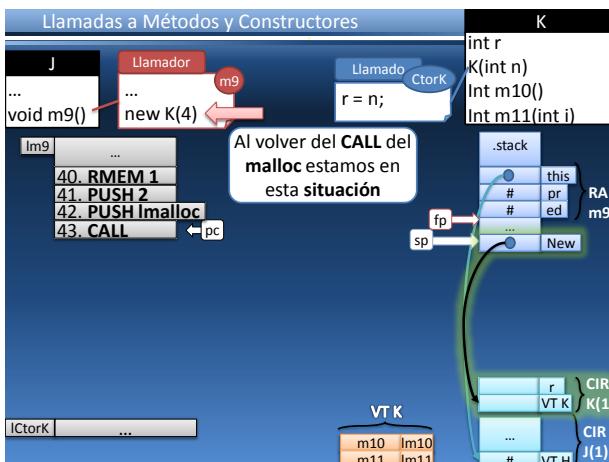
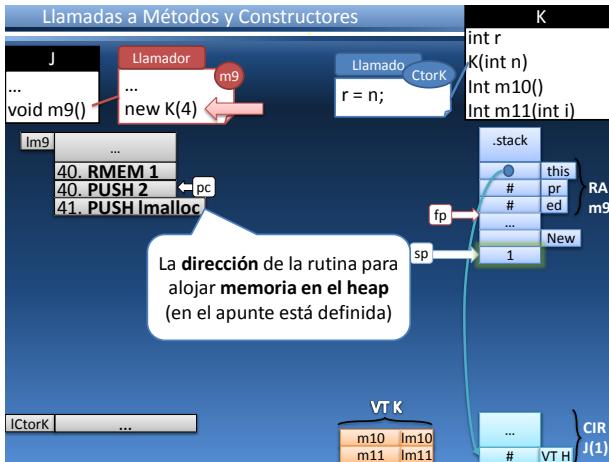


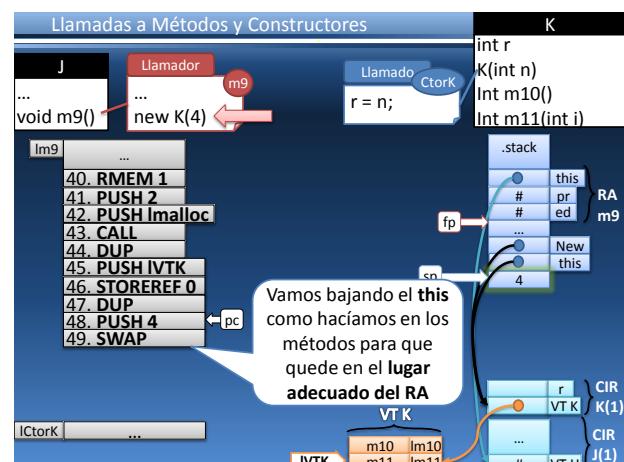
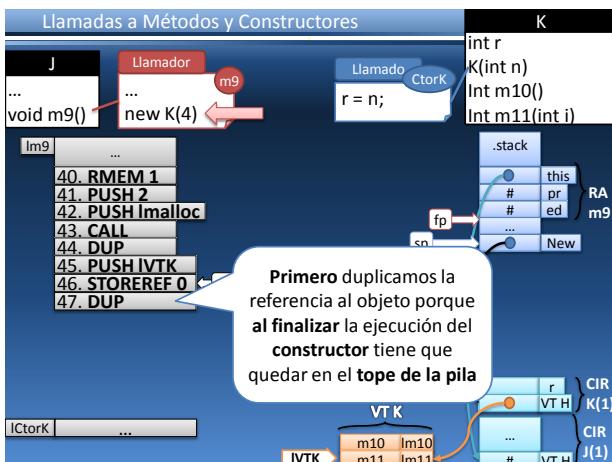
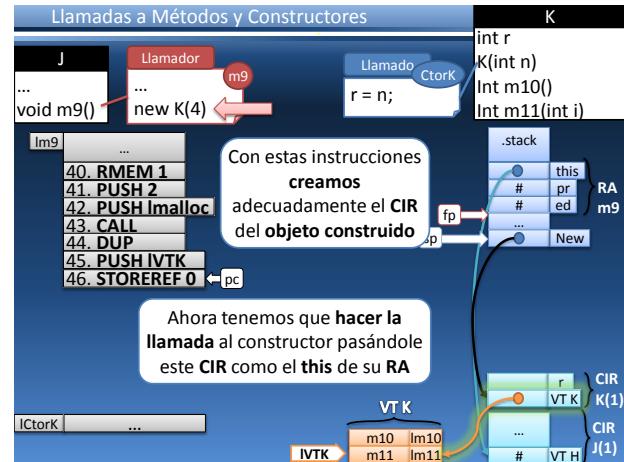
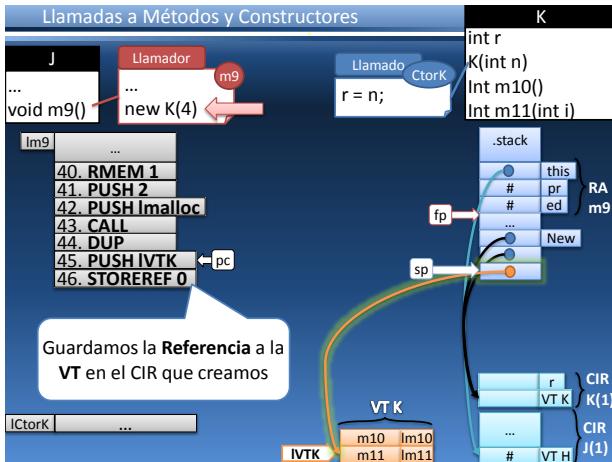


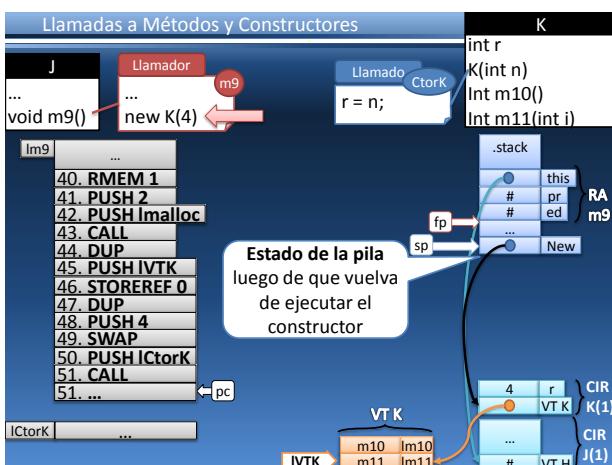
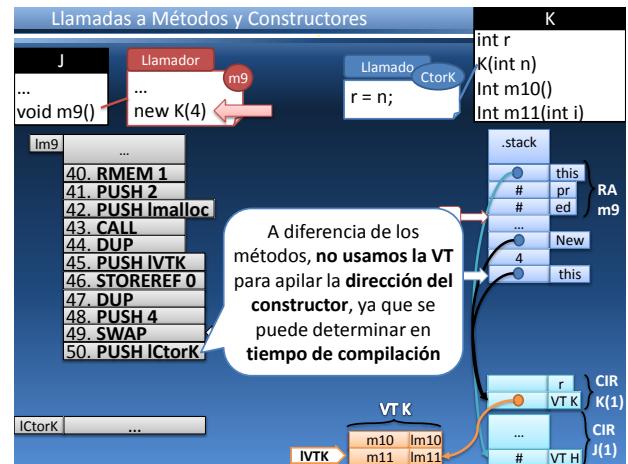
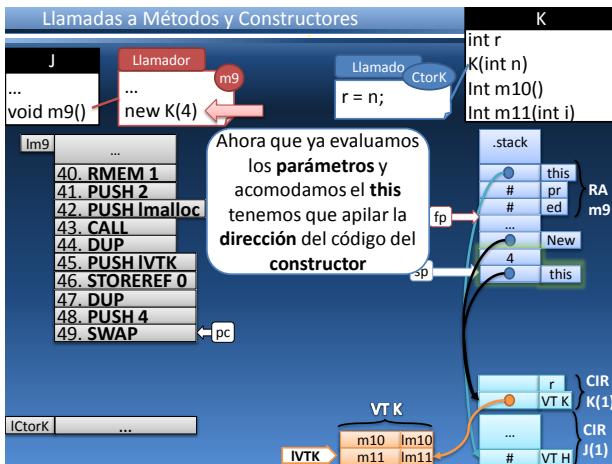
Constructores

- La **invocación a constructores** es muy similar a la de los **métodos**.
- Las principales **diferencias** son:
 - Antes de la llamada se **crea el objeto** asociado al constructor (este va a ser el **this** del RA del constructor).
 - Se le asigna la **VT** correspondiente
 - El **label** del código del constructor se determina en **tiempo de compilación**.









Estructura de traducción y datos estáticos

- Hasta ahora vimos qué instrucciones se utilizan para implementar expresiones, asignaciones y llamadas.
- En resumen, por cada clase generaremos código para:
 - Crear la VT de la clase, en la sección de DATOS estáticos (.data) de la memoria
 - Implementar todos los métodos y el constructor de la clase, en la sección de CÓDIGO (.code) de la memoria

Estructura de traducción y datos estáticos

- Para indicar qué parte de la memoria se está afectando utilizamos las **directivas**:
- .DATA** : las subsecuentes instrucciones afectarán la sección de datos
- .CODE** : las subsecuentes instrucciones afectarán la sección de código
- En particular, para **crear la VT** de una clase se utilizan las **instrucciones de definición de datos**
- DW n1, ..., nm** : inicializa m celdas de memoria con los valores n1,...,nm

Estructura de traducción y datos estáticos

```
class A{
    int x
    m1(int v){...}
    m2(){...}
    A(){...}
}
class B extends A{
    int y
    m2(){...}
    m3(){...}
    B(){...}
}
```

.DATA
 VT A: DW lm1A,lm2A
 .CODE
 lm1A ... (Código de m1 de A)
 lm2A ... (Código de m2 de A)
 ICtorA ... (Código de ctor A)

Estructura de traducción y datos estáticos

```
class A{
    int x
    m1(int v){...}
    m2(){...}
    A(){...}
}
Class B extends A{
    int y
    m2(){...}
    m3(){...}
    B(){...}
}
```

.DATA
 VT A: DW lm1A,lm2A
 .CODE
 lm1A ... (Código de m1 de A)
 lm2A ... (Código de m2 de A)
 ICtorA ... (Código de ctor A)
 .DATA
 VT B: DW lm1A, lm2B
 DW lm3B
 .CODE
 lm2B ... (Código de m2 de B)
 lm3B ... (Código de m3 de B)
 ICtorB ... (Código de Ctor B)

Generando Código CelVMASM

Generando Código

- El código (en su mayoría) lo vamos a generar como parte del **control de sentencias** en el analizador semántico
- Básicamente, idea es agregar **instrucciones de generación** a los métodos **check()** asociados a cada **nodo del AST**
- Aun así, es importante el **orden** en que se incluyen estas **instrucciones de generación** dentro de estos métodos

Generación de Código a partir del AST



