



UNIVERSIDAD DE OVIEDO

**ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES E INGENIEROS INFORMÁTICOS DE
GIJÓN**

ÁREA DE LENGUAJES Y SISTEMAS INFORMÁTICOS

PROYECTO FIN DE CARRERA N° 1022036

**DESARROLLO DE UN SISTEMA DE PERSISTENCIA IMPLÍCITA
BASADO EN PROGRAMACIÓN ORIENTADA A ASPECTOS**

DOCUMENTO N° 3

MANUAL TÉCNICO



**JAVIER NOVAL ARANGO
JUNIO 2003**



UNIVERSIDAD DE OVIEDO

**ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES E INGENIEROS INFORMÁTICOS DE
GIJÓN**

ÁREA DE LENGUAJES Y SISTEMAS INFORMÁTICOS

PROYECTO FIN DE CARRERA N° 1022036

**DESARROLLO DE UN SISTEMA DE PERSISTENCIA IMPLÍCITA
BASADO EN PROGRAMACIÓN ORIENTADA A ASPECTOS**

DOCUMENTO N° 3

MANUAL TÉCNICO



JAVIER NOVAL ARANGO

JUNIO 2003

TUTOR: FRANCISCO ORTÍN SOLER

Índice general

1. DECISIONES DE IMPLEMENTACIÓN	1
1.1. Estructura para los campos	1
1.2. Estructura para los métodos	2
2. CLASES Y FUNCIONES	4
2.1. Fichero <i>guidgen.py</i>	4
2.1.1. Funciones	4
2.2. Fichero <i>Java.ml</i>	4
2.2.1. Clase <i>FalseFile</i>	4
2.2.2. Clase <i>SymbolTable</i>	5
2.2.3. Funciones	8
2.3. Fichero <i>interpreter.py</i>	8
2.3.1. Clase <i>RuntimeSymbolTable</i>	8
2.3.2. Clase <i>Interpreter</i>	10
2.4. Fichero <i>objs.py</i>	19
2.4.1. Clase <i>JClass</i>	19
2.4.2. Clase <i>JNullClass</i>	26
2.4.3. Clase <i>JMethodGroup</i>	26
2.4.4. Clase <i>JMethod</i>	28
2.4.5. Clase <i>JReturn</i>	32
2.4.6. Clase <i>UserCodeWrapper</i>	33
2.4.7. Clase <i>JConstructor</i>	33
2.4.8. Clase <i>JField</i>	35
2.4.9. Clase <i>JInstance</i>	36
2.4.10. Clase <i>JRef</i>	41
2.4.11. Funciones	43
2.5. Fichero <i>persistence.py</i>	52
2.5.1. Clase <i>InstanceTable</i>	52
2.5.2. Clase <i>Manager</i>	53
2.5.3. Clase <i>Storage</i>	58
2.5.4. Clase <i>SimpleStorage</i>	59
2.5.5. Clase <i>DBMStorage</i>	61
2.5.6. Clase <i>BSDDBStorage</i>	63
2.5.7. Clase <i>StoragePolicy</i>	65
2.5.8. Clase <i>SimplePolicy</i>	66
2.5.9. Clase <i>TimedPolicy</i>	67
3. GRAMÁTICA DEL LENGUAJE JAVA--	69

A. CÓDIGO FUENTE	72
A.1. Java.ml	72
A.2. interpreter.py	98
A.3. objs.py	102
A.4. persistence.py	119
A.5. guidgen.py	125

Índice de figuras

1.1. Clases para el ejemplo de implementación de los campos	1
1.2. Los campos en una instancia de la clase ClaseC	2
1.3. Clases para el ejemplo de implementación de los métodos	3
1.4. Implementación de los métodos	3

Capítulo 1

DECISIONES DE IMPLEMENTACIÓN

1.1 Estructura para los campos

La estructura de datos elegida para la implementación de los campos de las instancias está basada en los diccionarios de Python. En primer lugar, cada instancia tendrá un diccionario indexado por clases de Java—, teniendo entradas para su clase y todas sus clases base. Para cada una de esas entradas tendrá otro diccionario, esta vez indexado por el nombre del atributo y que contendrá referencias en las que finalmente se almacenarán los valores de los atributos. Por cada clase aparecerán todos los atributos accesibles desde ella.

```
1  class ClaseA {
2      Integer A = new Integer ();
3
4      void metodo1 () {
5          ++A;
6      }
7  }
8
9  class ClaseB {
10     Integer B = new Integer ();
11
12     void metodo2 () {
13         ++A;
14         ++B;
15     }
16 }
17
18 class ClaseC {
19     Integer A = new Integer ();
20
21     void metodo3 () {
22         ++A;
23         ++B;
24     }
25 }
```

Figura 1.1: Clases para el ejemplo de implementación de los campos

Como ejemplo, dado el código de la figura 1.1, se obtendrá en memoria la estructura de datos que se muestra en la figura 1.2 para cualquier nueva instancia de la clase `ClaseC`. Podemos observar cómo en todo momento se tiene acceso a los campos que son visibles desde cualquier clase de la jerarquía, de forma que cuando el método `metodo1` cambie el valor del atributo `A`, podrá acceder al definido en la clase `ClaseA`, al igual que lo haría el método `metodo2`. Sin

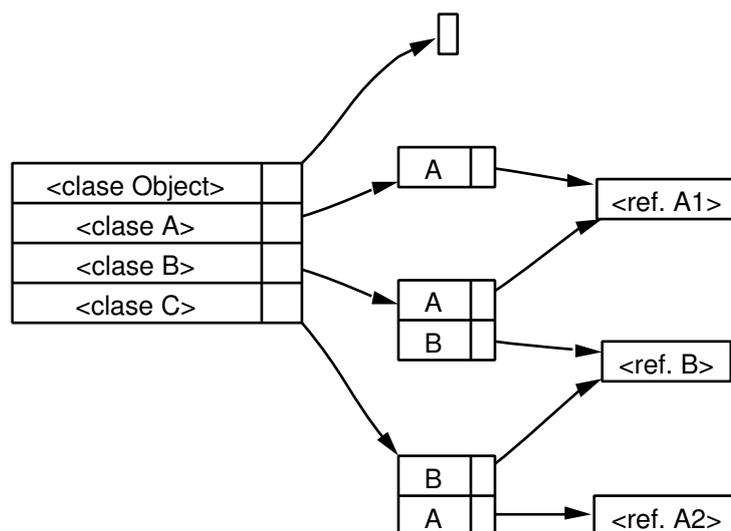


Figura 1.2: Los campos en una instancia de la clase ClaseC

embargo, `metodo3` accederá al atributo `A` definido en `ClaseC`.

La estructura elegida facilita el poder asegurar que cada método sólo ve aquellos campos que debe ver. Ya que al entrar en un método se introducen en la tabla de símbolos los campos a los que puede acceder, para que puedan emplearse sin necesidad de prefijarlos con `this.`, sólo tendremos que introducir aquellos campos que aparezcan en la entrada de la clase en que se definió el método. Además, también simplifica la búsqueda de miembros al usar la forma `referencia.campo`, puesto que nos basta mirar en la entrada de la clase a la que pertenezca la referencia.

Otra alternativa hubiese sido que para cada clase sólo se guardasen aquellos campos que se definieron en ella. Esta opción se desechó por la pérdida de rendimiento que suponía el tener que ir recorriendo la jerarquía para buscar un cierto campo, rendimiento que con la estructura adoptada sólo se pierde cuando se crean o destruyen instancias, aunque a costa de un mayor consumo de memoria. Por otra parte, al no elegir la estructura más simple se nos dificultaría la tarea de definir nuevos campos en tiempo de ejecución (como se puede hacer en el propio Python); sin embargo este hecho no se tuvo en consideración, ya que estamos modelando un lenguaje similar a Java y que por tanto tendría una estructura estática.

1.2 Estructura para los métodos

En el caso de los métodos de las clases se optó por una estructura similar a la de los campos. En cada clase tendremos un diccionario de *grupos de métodos* (conjuntos de métodos con el mismo nombre pero con parámetros diferentes en número y/o tipo) indexados por el nombre base de los mismos. Cada grupo tendrá internamente otra tabla con todos sus métodos, usando como clave en esta ocasión el *mangled name*¹ de cada método. Además cada clase tendrá otra tabla con todos los métodos que son aplicables a sus instancias indexados por su *mangled name*, ya que en muchas ocasiones éste nos será conocido y de esa forma podemos simplificar los accesos.

Si creamos la estructura de clases de la figura 1.3, la estructura que se nos crea en memoria es la que se muestra en la figura 1.4 (se muestran únicamente las tablas con los grupos de métodos – faltaría la tabla de métodos de cada clase).

¹Nombre creado a partir del nombre base del método y de los nombres de los tipos de argumentos que recibe.

```

1 class ClaseA {
2     void metodo1() {}
3     void metodo1(Integer i) {}
4 }
5
6 class ClaseB {
7     void metodo1(String s) {}
8     void metodo2(Integer i) {}
9 }

```

Figura 1.3: Clases para el ejemplo de implementación de los métodos

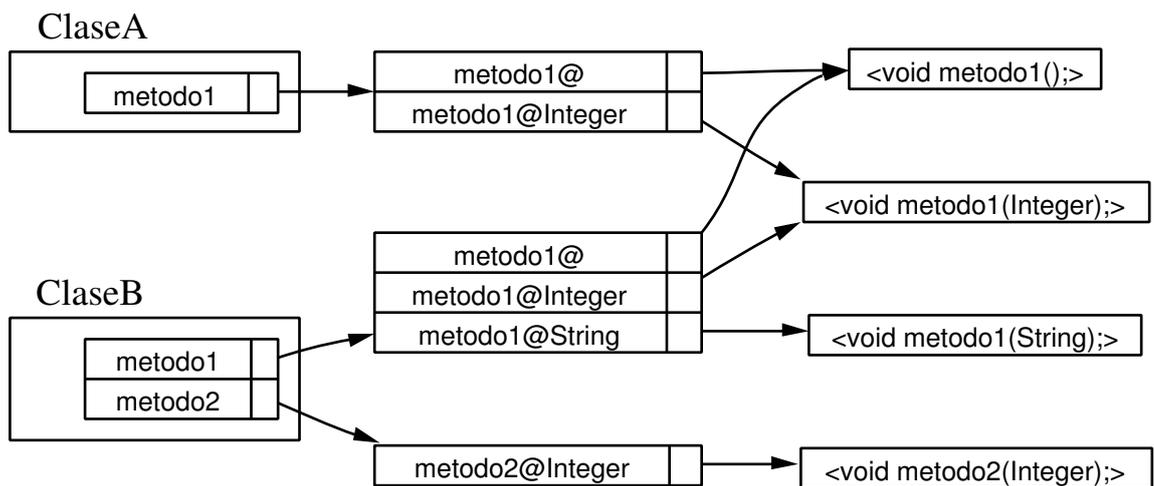


Figura 1.4: Implementación de los métodos

Capítulo 2

CLASES Y FUNCIONES

2.1 Fichero *guidgen.py*

2.1.1 Funciones

2.1.1.1 Función `guid()`

- **Descripción:** Genera una cadena que contiene un identificador único (*GUID*) con la siguiente estructura: *ip.uid.pid.tid.timestamp.random*, siendo:
 - ip:** Dirección IP de la máquina.
 - uid:** Identificador del usuario, o el carácter '1' si no está disponible
 - pid:** Identificador del proceso, o el carácter '1' si no está disponible
 - tid:** Identificador del hilo actual, o el carácter '1' si no está disponible
 - timestamp:** Segundos transcurridos desde la medianoche del 1 de Enero de 1970, con 3 decimales si el reloj del sistema soporta valores reales.
 - random:** Un entero aleatorio en el rango [0,32767].
- **Prototipo:** `guid()`
- **Argumentos:** Sin argumentos.
- **Valor de retorno:** Una cadena conteniendo el GUID.

2.2 Fichero *Java.ml*

2.2.1 Clase `FalseFile`

Simula un fichero para poder sustituir la salida estándar y hacer que lo que se escriba en ella se muestre en la ventana de la aplicación.

2.2.1.1 Atributos

- **`__write`:** La función de *nitrO* para escribir en la ventana de la aplicación.

2.2.1.2 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self, write)`
- **Argumentos:**
 - `self`: La propia instancia.
 - `write`: La función de *nitrO* para escribir en la ventana de la aplicación.
- **Valor de retorno:** Ninguno.

2.2.1.3 Método `write()`

- **Descripción:** Escribe una cadena.
- **Prototipo:** `write(self, str)`
- **Argumentos:**
 - `self`: La propia instancia.
 - `str`: La cadena a escribir.
- **Valor de retorno:** Ninguno.

2.2.2 Clase `SymbolTable`

Tabla de símbolos para el análisis semántico.

2.2.2.1 Atributos

- **`currentClass`:** Clase actual.
- **`currentMethod`:** Método actual.
- **`currentBlockVars`:** Variables definidas en el bloque actual.
- **`vars`:** Variables accesibles desde el bloque actual.
- **`blockStack`:** Pila de bloques.
- **`varStack`:** Pila de variables.

2.2.2.2 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self)`
- **Argumentos:**
 - `self`: La propia instancia.
- **Valor de retorno:** Ninguno.

2.2.2.3 Método classEnter()

- **Descripción:** Apila la clase en la que entramos y ajusta las variables visibles.
- **Prototipo:** classEnter(self, klass)
- **Argumentos:**
 - self:** La propia instancia.
 - klass:** La clase en la que entramos.
- **Valor de retorno:** Ninguno.

2.2.2.4 Método classExit()

- **Descripción:** Desapila la clase de la que salimos y ajusta las variables visibles.
- **Prototipo:** classExit(self)
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Ninguno.

2.2.2.5 Método methodEnter()

- **Descripción:** Apila el método en la que entramos y ajusta las variables visibles.
- **Prototipo:** methodEnter(self, method, args)
- **Argumentos:**
 - self:** La propia instancia.
 - method:** El método en el que entramos.
 - args:** Argumentos del método: es una lista de pares (nombre, tipo).
- **Valor de retorno:** Ninguno.

2.2.2.6 Método methodExit()

- **Descripción:** Desapila el método del que salimos y ajusta las variables visibles.
- **Prototipo:** methodExit(self)
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Ninguno.

2.2.2.7 Método blockEnter()

- **Descripción:** Apila el bloque en la que entramos y ajusta las variables visibles.
 - **Prototipo:** blockEnter(self)
 - **Argumentos:**
 - self:** La propia instancia.
 - **Valor de retorno:** Ninguno.
-

2.2.2.8 Método `blockExit()`

- **Descripción:** Desapila el bloque del que salimos y ajusta las variables visibles.
- **Prototipo:** `blockExit(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Ninguno.

2.2.2.9 Método `setVar()`

- **Descripción:** Añade una variable en el bloque actual.
- **Prototipo:** `setVar(self, name, type)`
- **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre de la variable.
 - type:** Tipo de la variable.
- **Valor de retorno:** Ninguno.

2.2.2.10 Método `getVar()`

- **Descripción:** Devuelve el tipo de una variable accesible desde el bloque actual.
- **Prototipo:** `getVar(self, name)`
- **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre de la variable.
- **Valor de retorno:** El tipo de la variable.

2.2.2.11 Método `getCurrentClass()`

- **Descripción:** Devuelve la clase actual.
 - **Prototipo:** `getCurrentClass(self)`
 - **Argumentos:**
 - self:** La propia instancia.
 - **Valor de retorno:** La clase actual.
-

2.2.3 Funciones

2.2.3.1 Función `generate_invokeVirtual()`

- **Descripción:** Genera el código correspondiente a la invocación virtual de un método.
- **Prototipo:** `generate_invokeVirtual(tree, klass, name, args, argTypes)`
- **Argumentos:**
 - tree:** Código que nos lleva a obtener la referencia al objeto.
 - klass:** Clase en la que tenemos que buscar el método.
 - name:** Nombre del método.
 - args:** Expresiones para calcular los argumentos que se le pasarán al método.
 - argTypes:** Tipos de los argumentos, para elegir el método a llamar.
- **Valor de retorno:** Una tupla con el código necesario para la llamada al método y su tipo de retorno, en ese orden.

2.2.3.2 Función `getClass()`

- **Descripción:** Busca la clase con el nombre pedido en la tabla de clases.
- **Prototipo:** `getClass(name)`
- **Argumentos:**
 - name:** Nombre de la clase.
- **Valor de retorno:** La clase pedida.

2.3 Fichero *interpreter.py*

2.3.1 Clase `RuntimeSymbolTable`

Tabla de símbolos en tiempo de ejecución.

2.3.1.1 Atributos

- **blockVars:** Variables definidas en el bloque actual.
 - **blockStack:** Pila de bloques.
 - **vars:** Variables accesibles desde el bloque actual.
 - **varStack:** Pila de variables.
 - **currentClass:** Clase actual.
 - **classStack:** Pila de clases.
 - **currentInstance:** Instancia actual.
 - **instanceStack:** Pila de instancias.
-

2.3.1.2 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self)`
- **Argumentos:**
 - `self:` La propia instancia.
- **Valor de retorno:** Ninguno.

2.3.1.3 Método `getCurrentClass()`

- **Descripción:** Devuelve la clase actual.
- **Prototipo:** `getCurrentClass(self)`
- **Argumentos:**
 - `self:` La propia instancia.
- **Valor de retorno:** La clase actual.

2.3.1.4 Método `getCurrentInstance()`

- **Descripción:** Devuelve la instancia actual.
- **Prototipo:** `getCurrentInstance(self)`
- **Argumentos:**
 - `self:` La propia instancia.
- **Valor de retorno:** La instancia actual.

2.3.1.5 Método `methodEnter()`

- **Descripción:** Apila el método en la que entramos y ajusta las variables visibles.
- **Prototipo:** `methodEnter(self, instance, method, args)`
- **Argumentos:**
 - `self:` La propia instancia.
 - `instance:` La instancia sobre la que se ejecuta el método.
 - `method:` El método en el que entramos.
 - `args:` Argumentos del método: es una lista de pares (nombre, tipo).
- **Valor de retorno:** Ninguno.

2.3.1.6 Método `methodExit()`

- **Descripción:** Desapila el método del que salimos y ajusta las variables visibles.
 - **Prototipo:** `methodExit(self)`
 - **Argumentos:**
 - `self:` La propia instancia.
 - **Valor de retorno:** Ninguno.
-

2.3.1.7 Método `blockEnter()`

- **Descripción:** Apila el bloque en la que entramos y ajusta las variables visibles.
- **Prototipo:** `blockEnter(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Ninguno.

2.3.1.8 Método `blockExit()`

- **Descripción:** Desapila el bloque del que salimos y ajusta las variables visibles.
- **Prototipo:** `blockExit(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Ninguno.

2.3.1.9 Método `setVar()`

- **Descripción:** Añade una variable en el bloque actual.
- **Prototipo:** `setVar(self, name, ref)`
- **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre de la variable.
 - ref:** Referencia contenida en la variable.
- **Valor de retorno:** Ninguno.

2.3.1.10 Método `getVar()`

- **Descripción:** Devuelve la referencia contenida en una variable accesible desde el bloque actual.
- **Prototipo:** `getVar(self, name)`
- **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre de la variable.
- **Valor de retorno:** El tipo de la variable.

2.3.2 Clase `Interpreter`

Intérprete de código de usuario.

2.3.2.1 Atributos

- **application:** La aplicación.
- **symbolTable:** La tabla de símbolos en tiempo de ejecución.
- **classes:** Clases conocidas.
- **manager:** El gestor de persistencia.

2.3.2.2 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self, application, classes =)`
- **Argumentos:**
 - self:** La propia instancia.
 - application:** La aplicación.
 - classes:** Conjunto inicial de clases.
- **Valor de retorno:** Ninguno.

2.3.2.3 Método `terminate()`

- **Descripción:** Se llama al terminar la ejecución del programa para que se encargue de finalizar todos sus componentes.
- **Prototipo:** `terminate(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Ninguno.

2.3.2.4 Método `getSymbolTable()`

- **Descripción:** Devuelve la tabla de símbolos en tiempo de ejecución
- **Prototipo:** `getSymbolTable(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** La tabla de símbolos.

2.3.2.5 Método `getClassByName()`

- **Descripción:** Obtiene una clase dado su nombre.
 - **Prototipo:** `getClassByName(self, name)`
 - **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre de la clase.
 - **Valor de retorno:** La clase pedida.
-

2.3.2.6 Método `getPersistenceManager()`

- **Descripción:** Devuelve el gestor de persistencia.
- **Prototipo:** `getPersistenceManager(self)`
- **Argumentos:**
 - `self`: La propia instancia.
- **Valor de retorno:** El gestor de persistencia.

2.3.2.7 Método `getApplication()`

- **Descripción:** Devuelve la aplicación.
- **Prototipo:** `getApplication(self)`
- **Argumentos:**
 - `self`: La propia instancia.
- **Valor de retorno:** La aplicación.

2.3.2.8 Método `parse()`

- **Descripción:** Evalúa código de usuario.
- **Prototipo:** `parse(self, l)`
- **Argumentos:**
 - `self`: La propia instancia.
 - `l`: Supuesta secuencia con código de usuario.
- **Valor de retorno:** El resultado de ejecutar el código, o bien `l` si no es código de usuario.

2.3.2.9 Método `visit_main()`

- **Descripción:** Visitor: bloque principal.
- **Prototipo:** `visit_main(self, block)`
- **Argumentos:**
 - `self`: La propia instancia.
 - `block`: Bloque de código con el programa principal.
- **Valor de retorno:** Ninguno.

2.3.2.10 Método `visit_block()`

- **Descripción:** Visitor: un bloque de código.
 - **Prototipo:** `visit_block(self, *statements)`
 - **Argumentos:**
 - `self`: La propia instancia.
 - `statements`: Cero o más sentencias que componen el bloque.
 - **Valor de retorno:** Ninguno.
-

2.3.2.11 Método `visit_decl()`

- **Descripción:** Visitor: una declaración de variables.
- **Prototipo:** `visit_decl(self, klass, *vars)`
- **Argumentos:**
 - self:** La propia instancia.
 - klass:** El tipo de las variables.
 - vars:** Lista con al menos un nombre de variable.
- **Valor de retorno:** Ninguno.

2.3.2.12 Método `visit_return()`

- **Descripción:** Visitor: la expresión `'return expr;'`.
- **Prototipo:** `visit_return(self, exprs)`
- **Argumentos:**
 - self:** La propia instancia.
 - exprs:** La expresión (o expresiones separadas por comas) cuyo resultado se devolverá.
- **Valor de retorno:** Ninguno.

2.3.2.13 Método `visit_voidReturn()`

- **Descripción:** Visitor: la expresión `'return;'`.
- **Prototipo:** `visit_voidReturn(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Ninguno.

2.3.2.14 Método `visit_exprs()`

- **Descripción:** Visitor: una o más expresiones en una misma sentencia.
 - **Prototipo:** `visit_exprs(self, *exprs)`
 - **Argumentos:**
 - self:** La propia instancia.
 - exprs:** Las expresiones a evaluar.
 - **Valor de retorno:** El resultado de evaluar la última expresión.
-

2.3.2.15 Método `visit_assign()`

- **Descripción:** Visitor: una asignación.
- **Prototipo:** `visit_assign(self, lval, rval)`
- **Argumentos:**
 - self:** La propia instancia.
 - lval:** Parte izquierda de la asignación.
 - rval:** Parte derecha de la asignación.
- **Valor de retorno:** La parte izquierda de la asignación.

2.3.2.16 Método `visit_is()`

- **Descripción:** Visitor: comprobación de la identidad de un objeto.
- **Prototipo:** `visit_is(self, lval, rval)`
- **Argumentos:**
 - self:** La propia instancia.
 - lval:** Parte izquierda del operador.
 - rval:** Parte derecha del operador.
- **Valor de retorno:** Una referencia booleana verdadera o falsa.

2.3.2.17 Método `visit_land()`

- **Descripción:** Visitor: *and* lógico.
- **Prototipo:** `visit_land(self, lval, rval)`
- **Argumentos:**
 - self:** La propia instancia.
 - lval:** Parte izquierda del operador.
 - rval:** Parte derecha del operador.
- **Valor de retorno:** Una referencia booleana verdadera o falsa.

2.3.2.18 Método `visit_lor()`

- **Descripción:** Visitor: *or* lógico.
 - **Prototipo:** `visit_lor(self, lval, rval)`
 - **Argumentos:**
 - self:** La propia instancia.
 - lval:** Parte izquierda del operador.
 - rval:** Parte derecha del operador.
 - **Valor de retorno:** Una referencia booleana verdadera o falsa.
-

2.3.2.19 Método visit_Inot()

- **Descripción:** Visitor: *not* lógico.
- **Prototipo:** visit_Inot(self, val)
- **Argumentos:**
 - self:** La propia instancia.
 - val:** La expresión a negar.
- **Valor de retorno:** Una referencia booleana verdadera o falsa.

2.3.2.20 Método visit_new()

- **Descripción:** Visitor: creación de objetos.
- **Prototipo:** visit_new(self, klass, ctor, ctor_args)
- **Argumentos:**
 - self:** La propia instancia.
 - klass:** Clase de la que crear la instancia.
 - ctor:** El constructor a invocar.
 - ctor_args:** Lista de expresiones cuyos valores se pasarán como argumentos al constructor.
- **Valor de retorno:** El nuevo objeto creado.

2.3.2.21 Método visit_null()

- **Descripción:** Visitor: la referencia nula.
- **Prototipo:** visit_null(self)
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** La referencia nula.

2.3.2.22 Método visit_this()

- **Descripción:** Visitor: la referencia a la instancia sobre la que se ejecuta el método actual.
 - **Prototipo:** visit_this(self)
 - **Argumentos:**
 - self:** La propia instancia.
 - **Valor de retorno:** La instancia actual.
-

2.3.2.23 Método visit_true()

- **Descripción:** Visitor: el valor lógico verdadero.
- **Prototipo:** visit_true(self)
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** La instancia verdadera.

2.3.2.24 Método visit_ident()

- **Descripción:** Visitor: una variable accesible desde el método actual.
- **Prototipo:** visit_ident(self, ident)
- **Argumentos:**
 - self:** La propia instancia.
 - ident:** Nombre de la variable.
- **Valor de retorno:** La referencia a esa variable.

2.3.2.25 Método visit_int()

- **Descripción:** Visitor: un entero.
- **Prototipo:** visit_int(self, i)
- **Argumentos:**
 - self:** La propia instancia.
 - i:** El entero.
- **Valor de retorno:** El entero.

2.3.2.26 Método visit_str()

- **Descripción:** Visitor: una cadena.
 - **Prototipo:** visit_str(self, s)
 - **Argumentos:**
 - self:** La propia instancia.
 - s:** La cadena.
 - **Valor de retorno:** La cadena.
-

2.3.2.27 Método `visit_invokevirtual()`

- **Descripción:** Visitor: invocación de un método virtual.
- **Prototipo:** `visit_invokevirtual(self, ref, mangledName, args)`
- **Argumentos:**
 - self:** La propia instancia.
 - ref:** Referencia sobre la que se invoca el método.
 - mangledName:** *Mangled name* del método.
 - args:** Argumentos para la invocación.
- **Valor de retorno:** El valor devuelto por ese método.

2.3.2.28 Método `visit_invokespecial()`

- **Descripción:** Visitor: invocación especial de un método (resuelto en tiempo de compilación).
- **Prototipo:** `visit_invokespecial(self, ref, method, args)`
- **Argumentos:**
 - self:** La propia instancia.
 - ref:** Referencia sobre la que se invoca el método.
 - method:** El método.
 - args:** Argumentos para la invocación.
- **Valor de retorno:** El valor devuelto por ese método.

2.3.2.29 Método `visit_field()`

- **Descripción:** Visitor: acceso a un campo de una instancia.
- **Prototipo:** `visit_field(self, ref, klass, name)`
- **Argumentos:**
 - self:** La propia instancia.
 - ref:** Referencia de la que accedemos al campo.
 - klass:** Clase que creó el campo.
 - name:** Nombre del campo.
- **Valor de retorno:** El campo accedido.

2.3.2.30 Método `visit_if()`

- **Descripción:** Visitor: sentencia `if`.
 - **Prototipo:** `visit_if(self, cond, ifTrue, ifFalse)`
 - **Argumentos:**
 - self:** La propia instancia.
 - cond:** La condición.
-

ifTrue: Código a ejecutar si *cond* es verdadera.

ifFalse: Código a ejecutar si *cond* es falsa.

- **Valor de retorno:** Ninguno.

2.3.2.31 Método `visit_while()`

- **Descripción:** Visitor: sentencia `while`.

- **Prototipo:** `visit_while(self, cond, code)`

- **Argumentos:**

self: La propia instancia.

cond: La condición.

code: Código a ejecutar mientras *cond* sea verdadera.

- **Valor de retorno:** Ninguno.

2.3.2.32 Método `visit_for()`

- **Descripción:** Visitor: sentencia `for`.

- **Prototipo:** `visit_for(self, init, cond, step, code)`

- **Argumentos:**

self: La propia instancia.

init: Expresión de inicialización del bucle.

cond: La condición.

step: Expresión a ejecutar entre iteraciones.

code: Código a ejecutar mientras *cond* sea verdadera.

- **Valor de retorno:** Ninguno.

2.3.2.33 Método `visit_reify()`

- **Descripción:** Visitor: sentencia `reify`.

- **Prototipo:** `visit_reify(self, reify_code)`

- **Argumentos:**

self: La propia instancia.

reify_code: Código a ejecutar.

- **Valor de retorno:** Ninguno.
-

2.3.2.34 Método `visit_upcast()`

- **Descripción:** Visitor: *cast* de un objeto hacia niveles más generales de su jerarquía.
- **Prototipo:** `visit_upcast(self, klass, ref)`
- **Argumentos:**
 - self:** La propia instancia.
 - klass:** Nuevo tipo.
 - ref:** La referencia original.
- **Valor de retorno:** Una referencia con el nuevo tipo.

2.3.2.35 Método `visit_downcast()`

- **Descripción:** Visitor: *cast* de un objeto hacia niveles más específicos de su jerarquía.
- **Prototipo:** `visit_downcast(self, klass, ref)`
- **Argumentos:**
 - self:** La propia instancia.
 - klass:** Nuevo tipo.
 - ref:** La referencia original.
- **Valor de retorno:** Una referencia con el nuevo tipo.

2.4 Fichero `objs.py`

2.4.1 Clase `JClass`

Representa una clase en el programa Java--, ya sea primitiva o de usuario.

2.4.1.1 Atributos

- **__name:** Nombre de la clase.
 - **__parent:** Clase padre.
 - **__isA:** Diccionario con la clase actual y todas sus clases base.
 - **__fields:** Diccionario con todos los campos de la clase y de sus clases base.
 - **__ownFields:** Diccionario con los campos definidos en esta clase.
 - **__methods:** Diccionario con todos los métodos de la clase y de sus clases base.
 - **__ownMethods:** Diccionario con los métodos definidos en esta clase.
 - **__methodGroups:** Grupos de métodos.
 - **__constructors:** Diccionario con los constructores definidos en esta clase.
-

2.4.1.2 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self, name, parent = None)`
- **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre de la clase para el programa Java--.
 - parent:** Clase Java-- padre.
- **Valor de retorno:** Ninguno.

2.4.1.3 Método `__repr__()`

- **Descripción:** Define la forma en que se imprimen los objetos de esta clase.
- **Prototipo:** `__repr__(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Cadena con la representación del objeto lista para su impresión.

2.4.1.4 Método `isNullClass()`

- **Descripción:** Indica si es la clase de la instancia nula.
- **Prototipo:** `isNullClass(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Un booleano puesto a True.

2.4.1.5 Método `getName()`

- **Descripción:** Devuelve el nombre de la clase.
- **Prototipo:** `getName(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** El nombre de la clase.

2.4.1.6 Método `getParent()`

- **Descripción:** Devuelve la clase padre, de haberla.
 - **Prototipo:** `getParent(self)`
 - **Argumentos:**
 - self:** La propia instancia.
 - **Valor de retorno:** La clase padre.
-

2.4.1.7 Método `getFields()`

- **Descripción:** Devuelve con los campos de la clase.
- **Prototipo:** `getFields(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Un diccionario con los campos.

2.4.1.8 Método `getField()`

- **Descripción:** Devuelve un campo determinado.
- **Prototipo:** `getField(self, name)`
- **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre del campo a devolver.
- **Valor de retorno:** El campo pedido.

2.4.1.9 Método `updateFromParent()`

- **Descripción:** Actualiza los métodos y campos con los heredados de la clase padre.
- **Prototipo:** `updateFromParent(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Ninguno.

2.4.1.10 Método `isA()`

- **Descripción:** Nos dice si una clase dada es uno de nuestros ancestros.
 - **Prototipo:** `isA(self, klass)`
 - **Argumentos:**
 - self:** La propia instancia.
 - klass:** La clase por la que preguntamos.
 - **Valor de retorno:** El valor `True` o `False` según la clase dada sea uno de nuestros ancestros o no.
-

2.4.1.11 Método `addFields()`

- **Descripción:** Añade una serie de campos a la clase.
- **Prototipo:** `addFields(self, fields)`
- **Argumentos:**
 - self:** La propia instancia.
 - fields:** Una lista con los campos a añadir.
- **Valor de retorno:** Ninguno.

2.4.1.12 Método `addField()`

- **Descripción:** Añade un campo a la clase.
- **Prototipo:** `addField(self, field)`
- **Argumentos:**
 - self:** La propia instancia.
 - field:** El campo a añadir.
- **Valor de retorno:** Ninguno.

2.4.1.13 Método `addMethods()`

- **Descripción:** Añade una serie de métodos a la clase.
- **Prototipo:** `addMethods(self, methods)`
- **Argumentos:**
 - self:** La propia instancia.
 - fields:** Una lista con los métodos a añadir.
- **Valor de retorno:** Ninguno.

2.4.1.14 Método `addMethod()`

- **Descripción:** Añade un método a la clase.
 - **Prototipo:** `addMethod(self, method)`
 - **Argumentos:**
 - self:** La propia instancia.
 - field:** El método a añadir.
 - **Valor de retorno:** Ninguno.
-

2.4.1.15 Método addConstructors()

- **Descripción:** Añade una serie de constructores a la clase.
- **Prototipo:** addConstructors(self, constructors)
- **Argumentos:**
 - self:** La propia instancia.
 - fields:** Una lista con los constructores a añadir.
- **Valor de retorno:** Ninguno.

2.4.1.16 Método addConstructor()

- **Descripción:** Añade un constructor a la clase.
- **Prototipo:** addConstructor(self, constructor)
- **Argumentos:**
 - self:** La propia instancia.
 - field:** El constructor a añadir.
- **Valor de retorno:** Ninguno.

2.4.1.17 Método addDefaultConstructor()

- **Descripción:** Añade el constructor por defecto.
- **Prototipo:** addDefaultConstructor(self)
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Ninguno.

2.4.1.18 Método getMethods()

- **Descripción:** Devuelve los métodos más aptos para invocarse con los tipos dados para los argumentos.
 - **Prototipo:** getMethods(self, name, argTypes)
 - **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre de los posibles métodos.
 - argTypes:** Tipos de los argumentos que deben aceptar.
 - **Valor de retorno:** Una lista con los métodos.
-

2.4.1.19 Método `getMethod()`

- **Descripción:** Devuelve el método más apto para invocarse con los tipos dados para los argumentos.
- **Prototipo:** `getMethod(self, name, argTypes)`
- **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre del método.
 - argTypes:** Tipos de los argumentos que debe aceptar.
- **Valor de retorno:** El método.

2.4.1.20 Método `getMethodByMangledName()`

- **Descripción:** Devuelve un método dado su *mangled name*, esto es, nombre derivado del nombre del método y de los tipos de sus argumentos.
- **Prototipo:** `getMethodByMangledName(self, mangledName)`
- **Argumentos:**
 - self:** La propia instancia.
 - mangledName:** Nombre transformado del método.
- **Valor de retorno:** El método pedido.

2.4.1.21 Método `getConstructors()`

- **Descripción:** Devuelve los constructores más aptos para invocarse con los tipos dados para los argumentos.
- **Prototipo:** `getConstructors(self, argTypes)`
- **Argumentos:**
 - self:** La propia instancia.
 - argTypes:** Tipos de los argumentos que deben aceptar.
- **Valor de retorno:** Una lista con los constructores.

2.4.1.22 Método `getConstructor()`

- **Descripción:** Devuelve el constructor más apto para invocarse con los tipos dados para los argumentos.
 - **Prototipo:** `getConstructor(self, argTypes)`
 - **Argumentos:**
 - self:** La propia instancia.
 - argTypes:** Tipos de los argumentos que debe aceptar.
 - **Valor de retorno:** El constructor.
-

2.4.1.23 Método `getDefaultConstructor()`

- **Descripción:** Devuelve el constructor por defecto (es decir, el que no recibe argumentos).
- **Prototipo:** `getDefaultConstructor(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** El constructor.

2.4.1.24 Método `distance()`

- **Descripción:** Devuelve la distancia entre la clase actual y una clase ancestro suya, definida como el número de niveles en la jerarquía que hay entre ellas.
- **Prototipo:** `distance(self, ancestro)`
- **Argumentos:**
 - self:** La propia instancia.
 - ancestro:** La supuesta clase base.
- **Valor de retorno:** La distancia entre la clase actual y la dada, o -1 si la clase dada no es una clase base de la actual.

2.4.1.25 Método `newInstance()`

- **Descripción:** Crea una nueva instancia de la clase.
- **Prototipo:** `newInstance(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** La nueva instancia.

2.4.1.26 Método `createFields()`

- **Descripción:** Crea los miembros para una instancia a partir de los campos.
 - **Prototipo:** `createFields(self, owner)`
 - **Argumentos:**
 - self:** La propia instancia.
 - owner:** Instancia que contendrá los campos.
 - **Valor de retorno:** Un diccionario con los miembros.
-

2.4.1.27 Método `__createFields()`

- **Descripción:** Método auxiliar que realiza efectivamente la creación de miembros.
- **Prototipo:** `__createFields(self, fields, owner)`
- **Argumentos:**
 - self:** La propia instancia.
 - fields:** Diccionario donde vamos creando los campos.
 - owner:** Instancia que contendrá los campos.
- **Valor de retorno:** Un diccionario con los miembros.

2.4.2 Clase `JNullClass`

La clase de la instancia nula.

2.4.2.1 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Ninguno.

2.4.2.2 Método `isA()`

- **Descripción:** Nos dice si una clase dada es uno de nuestros ancestros.
- **Prototipo:** `isA(self, klass)`
- **Argumentos:**
 - self:** La propia instancia.
 - klass:** La clase por la que preguntamos.
- **Valor de retorno:** Un booleano puesto a True.

2.4.2.3 Método `isNullClass()`

- **Descripción:** Indica si es la clase de la instancia nula.
- **Prototipo:** `isNullClass(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Un booleano puesto a True.

2.4.3 Clase `JMethodGroup`

Un grupo de métodos con el mismo nombre pero con argumentos de tipos diferentes.

2.4.3.1 Atributos

- **__name:** Nombre base de los métodos del grupo.
- **__methods:** Todos los métodos del grupo.

2.4.3.2 Atributos

- **__name:** El nombre base del método.
- **__methods:** Los propios métodos.

2.4.3.3 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self)`
- **Argumentos:**
`self:` La propia instancia.
- **Valor de retorno:** Ninguno.

2.4.3.4 Método `__repr__()`

- **Descripción:** Define la forma en que se imprimen los objetos de esta clase.
- **Prototipo:** `__repr__(self)`
- **Argumentos:**
`self:` La propia instancia.
- **Valor de retorno:** Cadena con la representación del objeto lista para su impresión.

2.4.3.5 Método `addMethod()`

- **Descripción:** Añade un método al grupo.
- **Prototipo:** `addMethod(self, method)`
- **Argumentos:**
`self:` La propia instancia.
`field:` El método a añadir.
- **Valor de retorno:** Ninguno.

2.4.3.6 Método `getMethodByMangledName()`

- **Descripción:** Devuelve un método dado su *mangled name*, esto es, nombre derivado del nombre del método y de los tipos de sus argumentos.
 - **Prototipo:** `getMethodByMangledName(self, mangledName)`
 - **Argumentos:**
`self:` La propia instancia.
`mangledName:` Nombre transformado del método.
 - **Valor de retorno:** El método pedido.
-

2.4.3.7 Método `match()`

- **Descripción:** Devuelve los métodos del grupo que mejor se adapten para ser invocados con argumentos de los tipos dados.
- **Prototipo:** `match(self, argTypes)`
- **Argumentos:**
 - `self`: La propia instancia.
 - `argTypes`: Tipos de los argumentos que deben aceptar.
- **Valor de retorno:** Una lista con los métodos.

2.4.3.8 Método `isEmpty()`

- **Descripción:** Indica si el grupo está vacío.
- **Prototipo:** `isEmpty(self)`
- **Argumentos:**
 - `self`: La propia instancia.
- **Valor de retorno:** Un booleano que dice si el grupo tiene algún método.

2.4.3.9 Método `clone()`

- **Descripción:** Crea un grupo nuevo con los mismos métodos.
- **Prototipo:** `clone(self)`
- **Argumentos:**
 - `self`: La propia instancia.
- **Valor de retorno:** El nuevo grupo.

2.4.4 Clase `JMethod`

Un método en el lenguaje Java—.

2.4.4.1 Atributos

- `__name`: Nombre del método.
 - `__returnType`: Tipo de retorno.
 - `__args`: Argumentos que recibe, en forma de lista de pares (nombre, tipo).
 - `__argTypes`: Lista con los tipos de los argumentos.
 - `__argTypeNames`: Lista con los nombres de los tipos de los argumentos.
 - `__argNames`: Lista con los nombres de los argumentos.
 - `__mangledName`: Nombre transformado del método.
 - `__code`: Código a ejecutar cuando sea invocado.
-

2.4.4.2 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self, name, returnType, args, code = None)`
- **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre del método.
 - returnType:** Tipo de su valor de retorno.
 - args:** Argumentos que recibe, en forma de lista de pares (nombre, tipo).
 - code:** Código a ejecutar cuando sea invocado.
- **Valor de retorno:** Ninguno.

2.4.4.3 Método `mangle()`

- **Descripción:** Método estático que calcula el *mangled name* de un método.
- **Prototipo:** `mangle(self, typeNames)`
- **Argumentos:**
 - self:** La propia instancia.
 - typeNames:** Nombres de los tipos de los argumentos.
- **Valor de retorno:** El *mangled name* del método.

2.4.4.4 Método `__repr__()`

- **Descripción:** Define la forma en que se imprimen los objetos de esta clase.
- **Prototipo:** `__repr__(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Cadena con la representación del objeto lista para su impresión.

2.4.4.5 Método `getName()`

- **Descripción:** Devuelve el nombre del método.
 - **Prototipo:** `getName(self)`
 - **Argumentos:**
 - self:** La propia instancia.
 - **Valor de retorno:** Una cadena con el nombre del método.
-

2.4.4.6 Método `getMangledName()`

- **Descripción:** Devuelve el *mangled name* del método.
- **Prototipo:** `getMangledName(self)`
- **Argumentos:**
 - `self`: La propia instancia.
- **Valor de retorno:** Una cadena con el *mangled name* del método.

2.4.4.7 Método `getClass()`

- **Descripción:** Devuelve la clase a la que pertenece el método.
- **Prototipo:** `getClass(self)`
- **Argumentos:**
 - `self`: La propia instancia.
- **Valor de retorno:** La clase.

2.4.4.8 Método `setClass()`

- **Descripción:** Cambia la clase a la que pertenece el método.
- **Prototipo:** `setClass(self, klass)`
- **Argumentos:**
 - `self`: La propia instancia.
 - `klass`: La nueva clase.
- **Valor de retorno:** Ninguno.

2.4.4.9 Método `getReturnType()`

- **Descripción:** Devuelve el tipo de retorno del método.
- **Prototipo:** `getReturnType(self)`
- **Argumentos:**
 - `self`: La propia instancia.
- **Valor de retorno:** El tipo de retorno.

2.4.4.10 Método `getArgs()`

- **Descripción:** Devuelve los argumentos del método.
 - **Prototipo:** `getArgs(self)`
 - **Argumentos:**
 - `self`: La propia instancia.
 - **Valor de retorno:** Una lista con los argumentos del método en forma de lista de pares (nombre, tipo).
-

2.4.4.11 Método `getSuperMethod()`

- **Descripción:** Devuelve el método que sobrescribimos.
- **Prototipo:** `getSuperMethod(self)`
- **Argumentos:**
 - `self`: La propia instancia.
- **Valor de retorno:** El método.

2.4.4.12 Método `setSuperMethod()`

- **Descripción:** Cambia el método que sobrescribimos.
- **Prototipo:** `setSuperMethod(self, super)`
- **Argumentos:**
 - `self`: La propia instancia.
 - `super`: El método que sobrescribimos.
- **Valor de retorno:** Ninguno.

2.4.4.13 Método `getArgTypes()`

- **Descripción:** Devuelve los tipos de los argumentos.
- **Prototipo:** `getArgTypes(self)`
- **Argumentos:**
 - `self`: La propia instancia.
- **Valor de retorno:** Una lista con los tipos.

2.4.4.14 Método `getArgTypeNames()`

- **Descripción:** Devuelve los nombres de los argumentos.
- **Prototipo:** `getArgTypeNames(self)`
- **Argumentos:**
 - `self`: La propia instancia.
- **Valor de retorno:** Una lista con los nombres.

2.4.4.15 Método `distance()`

- **Descripción:** Calcula la distancia de los tipos de los argumentos que recibimos respecto a aquellos que se indicaron al declarar el método.
 - **Prototipo:** `distance(self, args)`
 - **Argumentos:**
 - `self`: La propia instancia.
 - `args`: Tipos de los argumentos que recibimos.
 - **Valor de retorno:** La distancia, o -1 si no es apto.
-

2.4.4.16 Método `setCode()`

- **Descripción:** Indica qué código se ejecutará al invocarse el método.
- **Prototipo:** `setCode(self, code)`
- **Argumentos:**
 - self:** La propia instancia.
 - code:** El código a ejecutar.
- **Valor de retorno:** Ninguno.

2.4.4.17 Método `invoke()`

- **Descripción:** Ejecuta el código del método.
- **Prototipo:** `invoke(self, interpreter, instance, args)`
- **Argumentos:**
 - self:** La propia instancia.
 - intepreter:** El intérprete.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Argumentos que nos pasan.
- **Valor de retorno:** Lo que devuelva el código del método.

2.4.5 Clase `JReturn`

Una clase para simplificar la ejecución de los return: se lanzan instancias suyas como excepciones y se capturan en los métodos.

2.4.5.1 Atributos

- **`__value`:** Valor que devolvemos.

2.4.5.2 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self, value)`
- **Argumentos:**
 - self:** La propia instancia.
 - value:** Valor que devolvemos.
- **Valor de retorno:** Ninguno.

2.4.5.3 Método `getValue()`

- **Descripción:** Devuelve el valor que se retorna.
- **Prototipo:** `getValue(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** El valor asignado.

2.4.6 Clase `UserCodeWrapper`

Un envoltorio para el código del usuario, que permite ejecutarlo como si se tratase de cualquier otra función.

2.4.6.1 Atributos

- **`__code`:** Código a ejecutar.

2.4.6.2 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self, code)`
- **Argumentos:**
 - self:** La propia instancia.
 - code:** Código a ejecutar.
- **Valor de retorno:** Ninguno.

2.4.6.3 Método `__call__()`

- **Descripción:** Ejecuta el código de usuario.
- **Prototipo:** `__call__(self, interpreter, instance, args)`
- **Argumentos:**
 - self:** La propia instancia.
 - intepreter:** El intérprete.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Argumentos que nos pasan.
- **Valor de retorno:** El valor devuelto por el código de usuario.

2.4.7 Clase `JConstructor`

Un constructor en el lenguaje Java—.

2.4.7.1 Atributos

- **`__hasSuperStatement`:** Indica si en el constructor se usó la sentencia `super` para llamar al constructor adecuado de la clase base.
-

2.4.7.2 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self, klass, args, code = None, hasSuperStatement = False)`
- **Argumentos:**
 - self:** La propia instancia.
 - klass:** Clase a la que pertenece el constructor.
 - args:** Argumentos que debemos recibir, en forma de lista de pares (nombre, tipo).
 - code:** Código a ejecutar cuando se invoque el constructor.
 - hasSuperStatement:** Indica si en el constructor se usó la sentencia `super` para llamar al constructor adecuado de la clase base.
- **Valor de retorno:** Ninguno.

2.4.7.3 Método `__repr__()`

- **Descripción:** Define la forma en que se imprimen los objetos de esta clase.
- **Prototipo:** `__repr__(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Cadena con la representación del objeto lista para su impresión.

2.4.7.4 Método `setHasSuperStatement()`

- **Descripción:** Le dice al constructor si se ha empleado la construcción `super` para invocar al constructor de la clase padre.
- **Prototipo:** `setHasSuperStatement(self, hasSuperStatement)`
- **Argumentos:**
 - self:** La propia instancia.
 - hasSuperStatement:** Booleano indicando si tiene la construcción `super` o no.
- **Valor de retorno:** Ninguno.

2.4.7.5 Método `invoke()`

- **Descripción:** Ejecuta el código del constructor.
 - **Prototipo:** `invoke(self, interpreter, instance, args)`
 - **Argumentos:**
 - self:** La propia instancia.
 - intepreter:** El intérprete.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Argumentos que nos pasan.
 - **Valor de retorno:** Ninguno.
-

2.4.8 Clase JField

Un campo de una clase del lenguaje Java—.

2.4.8.1 Atributos

- **__name:** Nombre del campo.
- **__klass:** Tipo del campo.
- **__init:** Valor inicial.

2.4.8.2 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self, name, klass, init)`
- **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre del campo.
 - klass:** Tipo del campo.
 - init:** Valor inicial.
- **Valor de retorno:** Ninguno.

2.4.8.3 Método `getName()`

- **Descripción:** Devuelve el nombre del campo.
- **Prototipo:** `getName(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Una cadena con el nombre del campo.

2.4.8.4 Método `getClass()`

- **Descripción:** Devuelve la clase del campo.
- **Prototipo:** `getClass(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** La clase del campo.

2.4.8.5 Método `getInit()`

- **Descripción:** Devuelve el valor inicial del campo.
 - **Prototipo:** `getInit(self)`
 - **Argumentos:**
 - self:** La propia instancia.
 - **Valor de retorno:** El valor inicial del campo.
-

2.4.9 Clase JInstance

Una instancia del lenguaje Java—.

2.4.9.1 Atributos

- **klass:** Tipo de la instancia.
- **value:** Valor primitivo asociado.
- **guid:** GUID de la instancia.
- **storing:** Si la instancia se está almacenando.
- **modified:** Si la instancia está modificada.
- **nonStorable:** Si la instancia no se debe almacenar.
- **persistent:** Si la instancia es persistente.
- **fields:** Diccionario con los campos de la instancia.

2.4.9.2 Atributos

- **value:** Si la instancia corresponde a un tipo primitivo con un valor asociado (por ejemplo, un entero o una cadena), se usa este miembro para almacenar dicho valor.

2.4.9.3 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self, klass, guid = None)`
- **Argumentos:**
 - self:** La propia instancia.
 - klass:** El tipo de la instancia.
 - guid:** El GUID de la instancia.
- **Valor de retorno:** Ninguno.

2.4.9.4 Método `getID()`

- **Descripción:** Devuelve el GUID de la instancia.
 - **Prototipo:** `getID(self)`
 - **Argumentos:**
 - self:** La propia instancia.
 - **Valor de retorno:** Una cadena con el GUID de la instancia.
-

2.4.9.5 Método `isNonStorable()`

- **Descripción:** Indica si la instancia se puede almacenar o no.
- **Prototipo:** `isNonStorable(self)`
- **Argumentos:**
 - `self:` La propia instancia.
- **Valor de retorno:** Un booleano.

2.4.9.6 Método `setNonStorable()`

- **Descripción:** Le dice a la instancia que no es almacenable.
- **Prototipo:** `setNonStorable(self)`
- **Argumentos:**
 - `self:` La propia instancia.
- **Valor de retorno:** Ninguno.

2.4.9.7 Método `__getstate__()`

- **Descripción:** Prepara una representación del estado interno para almacenarse mediante el módulo *pickle*.
- **Prototipo:** `__getstate__(self)`
- **Argumentos:**
 - `self:` La propia instancia.
- **Valor de retorno:** Diccionario con los campos a guardar.

2.4.9.8 Método `__setstate__()`

- **Descripción:** Cambia el estado interno de la instancia a partir del recuperado por el módulo *pickle*.
- **Prototipo:** `__setstate__(self)`
- **Argumentos:**
 - `self:` La propia instancia.
 - `dict:` Diccionario a partir del que crear el estado interno.
- **Valor de retorno:** Ninguno.

2.4.9.9 Método `isA()`

- **Descripción:** Indica si la instancia es de la clase dada o de alguna de sus descendientes.
 - **Prototipo:** `isA(self, klass)`
 - **Argumentos:**
 - `self:` La propia instancia.
 - `klass:` Clase de la que preguntamos si somos de su tipo.
 - **Valor de retorno:** Un booleano.
-

2.4.9.10 Método getClass()

- **Descripción:** La clase de la instancia.
- **Prototipo:** getClass(self)
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** La clase.

2.4.9.11 Método getField()

- **Descripción:** Devuelve un campo determinado.
- **Prototipo:** getField(self, klass, field)
- **Argumentos:**
 - self:** La propia instancia.
 - klass:** Clase que creó el campo (o una de sus clases base).
 - field:** Nombre del campo.
- **Valor de retorno:** El campo pedido.

2.4.9.12 Método setField()

- **Descripción:** Cambia el valor contenido en un campo.
- **Prototipo:** setField(self, klass, field, ref)
- **Argumentos:**
 - self:** La propia instancia.
 - klass:** Clase que creó el campo (o una de sus clases base).
 - field:** Nombre del campo.
 - ref:** Valor a asignar al campo.
- **Valor de retorno:** Ninguno.

2.4.9.13 Método getFields()

- **Descripción:** Devuelve todos los campos de la instancia pertenecientes a una cierta clase.
 - **Prototipo:** getFields(self, klass)
 - **Argumentos:**
 - self:** La propia instancia.
 - klass:** La clase.
 - **Valor de retorno:** Un diccionario con los campos.
-

2.4.9.14 Método `getValue()`

- **Descripción:** Devuelve el valor interno asociado a la instancia, de haberlo.
- **Prototipo:** `getValue(self)`
- **Argumentos:**
 - `self:` La propia instancia.
- **Valor de retorno:** El valor asociado.

2.4.9.15 Método `setValue()`

- **Descripción:** Cambia el valor interno asociado a la instancia.
- **Prototipo:** `setValue(self, value)`
- **Argumentos:**
 - `self:` La propia instancia.
 - `value:` El nuevo valor interno.
- **Valor de retorno:** El valor.

2.4.9.16 Método `isNull()`

- **Descripción:** Indica si es la instancia nula.
- **Prototipo:** `isNull(self)`
- **Argumentos:**
 - `self:` La propia instancia.
- **Valor de retorno:** Un booleano.

2.4.9.17 Método `isModified()`

- **Descripción:** Indica si se ha modificado la instancia desde la última vez que se cargó o almacenó.
- **Prototipo:** `isModified(self)`
- **Argumentos:**
 - `self:` La propia instancia.
- **Valor de retorno:** Un booleano.

2.4.9.18 Método `setModified()`

- **Descripción:** Marca la instancia como modificada o no modificada.
 - **Prototipo:** `setModified(self, modified)`
 - **Argumentos:**
 - `self:` La propia instancia.
 - `modified:` Booleano que indica si la instancia se modificó o no.
 - **Valor de retorno:** Ninguno.
-

2.4.9.19 Método `isPersistent()`

- **Descripción:** Indica si la instancia es persistente.
- **Prototipo:** `isPersistent(self)`
- **Argumentos:**
 - `self:` La propia instancia.
- **Valor de retorno:** Un booleano.

2.4.9.20 Método `makePersistent()`

- **Descripción:** Hace que la instancia y sus campos sean persistentes.
- **Prototipo:** `makePersistent(self)`
- **Argumentos:**
 - `self:` La propia instancia.
- **Valor de retorno:** Ninguno.

2.4.9.21 Método `makeTransient()`

- **Descripción:** Hace que la instancia y sus campos dejen de ser persistentes.
- **Prototipo:** `makeTransient(self, manager)`
- **Argumentos:**
 - `self:` La propia instancia.
 - `manager:` El gestor de persistencia.
- **Valor de retorno:** Ninguno.

2.4.9.22 Método `store()`

- **Descripción:** Guarda una instancia en un almacenamiento.
- **Prototipo:** `store(self, storage)`
- **Argumentos:**
 - `self:` La propia instancia.
 - `storage:` El almacenamiento en el que nos debemos guardar.
- **Valor de retorno:** Ninguno.

2.4.9.23 Método `getRefs()`

- **Descripción:** Devuelve todos los campos de la instancia, sin importar en qué clase se crearon.
 - **Prototipo:** `getRefs(self)`
 - **Argumentos:**
 - `self:` La propia instancia.
 - **Valor de retorno:** Un diccionario con los campos.
-

2.4.9.24 Método `restore()`

- **Descripción:** Carga una instancia de un almacenamiento.
- **Prototipo:** `restore(self, interpreter, storage)`
- **Argumentos:**
 - self:** La propia instancia.
 - interpreter:** El intérprete.
 - storage:** El almacenamiento del que recuperar nuestros campos.
- **Valor de retorno:** Ninguno.

2.4.10 Clase `JRef`

Una referencia del lenguaje Java—.

2.4.10.1 Atributos

- **klass:** Tipo de la referencia.
- **owner:** Instancia de la que somos un miembro.
- **instance:** Instancia que contenemos.

2.4.10.2 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self, klass, instance, owner = None)`
- **Argumentos:**
 - self:** La propia instancia.
 - klass:** Tipo de la referencia.
 - instance:** Instancia que contenemos.
 - owner:** Instancia de la que somos un miembro.
- **Valor de retorno:** Ninguno.

2.4.10.3 Método `getClass()`

- **Descripción:** Devuelve la clase de la referencia.
 - **Prototipo:** `getClass(self)`
 - **Argumentos:**
 - self:** La propia instancia.
 - **Valor de retorno:** La clase.
-

2.4.10.4 Método getInstance()

- **Descripción:** Devuelve la instancia contenida en la referencia.
- **Prototipo:** getInstance(self)
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** La instancia.

2.4.10.5 Método isNull()

- **Descripción:** Indica si la instancia está vacía, esto es, almacenamos la instancia nula.
- **Prototipo:** isNull(self)
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Un booleano.

2.4.10.6 Método setOwner()

- **Descripción:** Marca la referencia como campo de la instancia dada.
- **Prototipo:** setOwner(self, owner)
- **Argumentos:**
 - self:** La propia instancia.
 - owner:** Nueva instancia contenedora.
- **Valor de retorno:** Ninguno.

2.4.10.7 Método setInstance()

- **Descripción:** Cambia la instancia que almacenamos.
- **Prototipo:** setInstance(self, instance)
- **Argumentos:**
 - self:** La propia instancia.
 - instance:** Nueva instancia que contendremos.
- **Valor de retorno:** Ninguno.

2.4.10.8 Método setInstanceFrom()

- **Descripción:** Hace que contengamos la misma instancia que otra referencia.
 - **Prototipo:** setInstanceFrom(self, ref)
 - **Argumentos:**
 - self:** La propia instancia.
 - ref:** Referencia de la que copiaremos la instancia contenida.
 - **Valor de retorno:** Ninguno.
-

2.4.10.9 Método `__getstate__()`

- **Descripción:** Prepara una representación del estado interno para almacenarse mediante el módulo *pickle*.
- **Prototipo:** `__getstate__(self)`
- **Argumentos:**
`self`: La propia instancia.
- **Valor de retorno:** Diccionario con los campos a guardar.

2.4.10.10 Método `__setstate__()`

- **Descripción:** Cambia el estado interno de la instancia a partir del recuperado por el módulo *pickle*.
- **Prototipo:** `__setstate__(self, dict)`
- **Argumentos:**
`self`: La propia instancia.
`dict`: Diccionario a partir del que crear el estado interno.
- **Valor de retorno:** Ninguno.

2.4.10.11 Método `restore()`

- **Descripción:** Termina la restauración del estado interno de la referencia.
- **Prototipo:** `restore(self, interpreter, storage)`
- **Argumentos:**
`self`: La propia instancia.
`interpreter`: El intérprete.
`storage`: El almacenamiento del que recuperar la instancia a la que apuntábamos.
- **Valor de retorno:** Ninguno.

2.4.11 Funciones

2.4.11.1 Función `makeBool()`

- **Descripción:** Función auxiliar para simplificar el código cuando hace falta devolver un valor booleano al programa del usuario; devuelve la instancia que representa al valor lógico verdadero o la instancia nula (valor lógico falso) según el parámetro sea verdadero o falso, respectivamente.
 - **Prototipo:** `makeBool(test)`
 - **Argumentos:**
`test`: Determina si se devuelve la instancia que representa al valor lógico verdadero o falso.
 - **Valor de retorno:** La instancia correspondiente.
-

2.4.11.2 Función `makeSimple()`

- **Descripción:** Crea una instancia de un tipo básico y le asigna el valor dado.
- **Prototipo:** `makeSimple(klass, value)`
- **Argumentos:**
 - klass:** Clase de la que crear la instancia.
 - value:** Valor a asignar.
- **Valor de retorno:** Una nueva instancia de la clase solicitada y con el valor indicado asignado.

2.4.11.3 Función `makeBinaryAOp()`

- **Descripción:** Crea el método correspondiente a un cierto operador aritmético binario.
- **Prototipo:** `makeBinaryAOp(klass, op)`
- **Argumentos:**
 - klass:** Tipo del resultado.
 - op:** Función que corresponde al operador (generalmente una de las del módulo *operator* de Python).
- **Valor de retorno:** Una función que ejecuta las acciones correspondientes al operador, apta para ser añadida a una clase como uno de sus métodos.

2.4.11.4 Función `makeBinaryLOp()`

- **Descripción:** Crea el método correspondiente a un cierto operador aritmético binario.
- **Prototipo:** `makeBinaryLOp(op)`
- **Argumentos:**
 - op:** Función que corresponde al operador (generalmente una de las del módulo *operator* de Python).
- **Valor de retorno:** Una función que ejecuta las acciones correspondientes al operador, apta para ser añadida a una clase como uno de sus métodos.

2.4.11.5 Función `objectMethod_id()`

- **Descripción:** Método *id()* de la clase de usuario *Object*: devuelve el GUID del objeto.
 - **Prototipo:** `objectMethod_id(interpreter, method, instance, args)`
 - **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - method:** El propio método.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
 - **Valor de retorno:** Una instancia de la clase de usuario *String* con el GUID del objeto.
-

2.4.11.6 Función `objectMethod_makePersistent()`

- **Descripción:** Método `makePersistent()` de la clase de usuario *Object*: hace al objeto persistente.
- **Prototipo:** `objectMethod_makePersistent(interpreter, method, instance, args)`
- **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - method:** El propio método.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
- **Valor de retorno:** La instancia *void*.

2.4.11.7 Función `objectMethod_makeTransient()`

- **Descripción:** Método `makeTransient()` de la clase de usuario *Object*: hace que el objeto deje de ser persistente.
- **Prototipo:** `objectMethod_makeTransient(interpreter, method, instance, args)`
- **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - method:** El propio método.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
- **Valor de retorno:** La instancia *void*.

2.4.11.8 Función `integerMethod_preinc()`

- **Descripción:** Operador de preincremento de la clase de usuario *Integer*.
 - **Prototipo:** `integerMethod_preinc(interpreter, method, instance, args)`
 - **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - method:** El propio método.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
 - **Valor de retorno:** La instancia que se le pasó, con su valor incrementado en una unidad.
-

2.4.11.9 Función `integerMethod_postinc()`

- **Descripción:** Operador de postincremento de la clase de usuario *Integer*.
- **Prototipo:** `integerMethod_postinc(interpreter, method, instance, args)`
- **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - method:** El propio método.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
- **Valor de retorno:** Una nueva instancia de la clase de usuario *Integer* cuyo valor es igual al valor original de la instancia (antes de incrementarse éste en una unidad).

2.4.11.10 Función `integerMethod_predec()`

- **Descripción:** Operador de predecremento de la clase de usuario *Integer*.
- **Prototipo:** `integerMethod_predec(interpreter, method, instance, args)`
- **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - method:** El propio método.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
- **Valor de retorno:** La instancia que se le pasó, con su valor decrementado en una unidad.

2.4.11.11 Función `integerMethod_postdec()`

- **Descripción:** Operador de postdecremento de la clase de usuario *Integer*.
 - **Prototipo:** `integerMethod_postdec(interpreter, method, instance, args)`
 - **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - method:** El propio método.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
 - **Valor de retorno:** Una nueva instancia de la clase de usuario *Integer* cuyo valor es igual al valor original de la instancia (antes de decrementarse en una unidad).
-

2.4.11.12 Función `integerMethod_set()`

- **Descripción:** Método `set()` de la clase de usuario *Integer*: asigna el valor del entero que se le pasa a la instancia.
- **Prototipo:** `integerMethod_set(interpreter, method, instance, args)`
- **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - method:** El propio método.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
- **Valor de retorno:** La instancia *void*.

2.4.11.13 Función `integerMethod_toString()`

- **Descripción:** Método `toString()` de la clase de usuario *Integer*.
- **Prototipo:** `integerMethod_toString(interpreter, method, instance, args)`
- **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - method:** El propio método.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
- **Valor de retorno:** Una instancia de la clase de usuario *String* con la representación textual del valor de la instancia.

2.4.11.14 Función `integerCtor_default()`

- **Descripción:** Constructor por defecto de la clase de usuario *Integer*: asigna el valor 0 a la instancia.
 - **Prototipo:** `integerCtor_default(interpreter, ctor, instance, args)`
 - **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - ctor:** El propio constructor.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
 - **Valor de retorno:** La instancia *void*.
-

2.4.11.15 Función `integerCtor_int()`

- **Descripción:** Constructor de la clase de usuario *Integer*: asigna a la instancia el valor del entero que se le pasa.
- **Prototipo:** `integerCtor_int(interpreter, ctor, instance, args)`
- **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - ctor:** El propio constructor.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
- **Valor de retorno:** La instancia *void*.

2.4.11.16 Función `stringMethod_set()`

- **Descripción:** Método `set()` de la clase de usuario *String*: asigna el valor de la cadena que se le pasa a la instancia.
- **Prototipo:** `stringMethod_set(interpreter, method, instance, args)`
- **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - method:** El propio método.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
- **Valor de retorno:** La instancia *void*.

2.4.11.17 Función `stringCtor_default()`

- **Descripción:** Constructor por defecto de la clase de usuario *String*: asigna la cadena vacía a la instancia.
 - **Prototipo:** `stringCtor_default(interpreter, ctor, instance, args)`
 - **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - ctor:** El propio constructor.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
 - **Valor de retorno:** La instancia *void*.
-

2.4.11.18 Función `stringCtor_string()`

- **Descripción:** Constructor de la clase de usuario *String*: asigna a la instancia el valor de la cadena que se le pasa.
- **Prototipo:** `stringCtor_string(interpreter, ctor, instance, args)`
- **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - ctor:** El propio constructor.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
- **Valor de retorno:** La instancia *void*.

2.4.11.19 Función `consoleMethod_print()`

- **Descripción:** Método *print()* de la clase de usuario *Console*: escribe una cadena de texto por pantalla.
- **Prototipo:** `consoleMethod_print(interpreter, method, instance, args)`
- **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - method:** El propio método.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
- **Valor de retorno:** La instancia *void*.

2.4.11.20 Función `consoleMethod_println()`

- **Descripción:** Método *println()* de la clase de usuario *Console*: escribe una cadena de texto por pantalla seguida de un salto de línea.
 - **Prototipo:** `consoleMethod_println(interpreter, method, instance, args)`
 - **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - method:** El propio método.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
 - **Valor de retorno:** La instancia *void*.
-

2.4.11.21 Función `arrayMethod_getSize()`

- **Descripción:** Método `getSize()` de la clase de usuario *Array*: devuelve el tamaño actual del array.
- **Prototipo:** `arrayMethod_getSize(interpreter, method, instance, args)`
- **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - method:** El propio método.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
- **Valor de retorno:** Una nueva instancia de la clase de usuario *Integer* con el tamaño actual del array como valor.

2.4.11.22 Función `arrayMethod_setSize()`

- **Descripción:** Método `setSize()` de la clase de usuario *Array*: cambia el tamaño actual del array.
- **Prototipo:** `arrayMethod_setSize(interpreter, method, instance, args)`
- **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - method:** El propio método.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
- **Valor de retorno:** La instancia *void*.

2.4.11.23 Función `arrayMethod_getItem()`

- **Descripción:** Método `getItem()` de la clase de usuario *Array*: devuelve uno de los elementos del array.
 - **Prototipo:** `arrayMethod_getItem(interpreter, method, instance, args)`
 - **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - method:** El propio método.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
 - **Valor de retorno:** La instancia que ocupaba la posición indicada en el array.
-

2.4.11.24 Función `arrayMethod_setItem()`

- **Descripción:** Método `setItem()` de la clase de usuario *Array*: cambia uno de los elementos del array.
- **Prototipo:** `arrayMethod_setItem(interpreter, method, instance, args)`
- **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - method:** El propio método.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
- **Valor de retorno:** La instancia *void*.

2.4.11.25 Función `arrayCtor_default()`

- **Descripción:** Constructor por defecto de la clase de usuario *Array*: crea un array con capacidad inicial para un elemento.
- **Prototipo:** `arrayCtor_default(interpreter, ctor, instance, args)`
- **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - ctor:** El propio constructor.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
- **Valor de retorno:** La instancia *void*.

2.4.11.26 Función `arrayCtor_int()`

- **Descripción:** Constructor de la clase de usuario *Array*: crea un array con capacidad inicial para tantos elementos como se le indique.
 - **Prototipo:** `arrayCtor_int(interpreter, ctor, instance, args)`
 - **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - ctor:** El propio constructor.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
 - **Valor de retorno:** La instancia *void*.
-

2.4.11.27 Función `managerMethod_storeObject()`

- **Descripción:** Método `storeObject()` de la clase de usuario `PersistenceManager`: almacena un objeto en el sistema de almacenamiento seleccionado actualmente.
- **Prototipo:** `managerMethod_storeObject(interpreter, method, instance, args)`
- **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - method:** El propio método.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
- **Valor de retorno:** Una nueva instancia de la clase de usuario `String` con el GUID de la instancia almacenada.

2.4.11.28 Función `managerMethod_retrieveObject()`

- **Descripción:** Método `retrieveObject()` de la clase de usuario `PersistenceManager`: recupera un objeto del sistema de almacenamiento seleccionado actualmente.
- **Prototipo:** `managerMethod_retrieveObject(interpreter, method, instance, args)`
- **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - method:** El propio método.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
- **Valor de retorno:** La instancia solicitada.

2.4.11.29 Función `managerCtor_default()`

- **Descripción:** Constructor por defecto de la clase de usuario `PersistenceManager`
- **Prototipo:** `managerCtor_default(interpreter, ctor, instance, args)`
- **Argumentos:**
 - interpreter:** El intérprete del código de usuario.
 - ctor:** El propio constructor.
 - instance:** Instancia sobre la que se ejecuta el método.
 - args:** Lista con las referencias que se pasaron como argumentos.
- **Valor de retorno:** La instancia `void`.

2.5 Fichero `persistence.py`

2.5.1 Clase `InstanceTable`

Recuerda todas las instancias cargadas o almacenadas mientras sigan activas en memoria.

2.5.1.1 Atributos

- **instances:** Tabla de referencias débiles a las instancias que se almacenan o recuperan.

2.5.1.2 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self)`
- **Argumentos:**
 - `self:` La propia instancia.
- **Valor de retorno:** Ninguno.

2.5.1.3 Método `addInstance()`

- **Descripción:** Añade una instancia a la tabla.
- **Prototipo:** `addInstance(self, inst)`
- **Argumentos:**
 - `self:` La propia instancia.
 - `inst:` La instancia a añadir.
- **Valor de retorno:** Ninguno.

2.5.1.4 Método `getInstance()`

- **Descripción:** Recupera una instancia de la tabla.
- **Prototipo:** `getInstance(self, id)`
- **Argumentos:**
 - `self:` La propia instancia.
 - `id:` El GUID de la instancia a recuperar.
- **Valor de retorno:** La instancia pedida.

2.5.1.5 Método `dellInstance()`

- **Descripción:** Elimina una instancia de la tabla.
- **Prototipo:** `dellInstance(self, inst)`
- **Argumentos:**
 - `self:` La propia instancia.
 - `inst:` La instancia a eliminar.
- **Valor de retorno:** Ninguno.

2.5.2 Clase `Manager`

Sirve como punto único de acceso a la funcionalidad de persistencia.

2.5.2.1 Atributos

- **interpreter:** El intérprete.
- **application:** La aplicación.
- **instanceTable:** La tabla de instancias.
- **storages:** Tabla con todos los almacenamientos registrados.
- **policias:** Tabla con todas las políticas registradas.
- **storage:** Almacenamiento actual.
- **policy:** Política actual.

2.5.2.2 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self, interpreter, application)`
- **Argumentos:**
 - self:** La propia instancia.
 - interpreter:** El intérprete
 - application:** La aplicación.
- **Valor de retorno:** Ninguno.

2.5.2.3 Método `getInstanceTable()`

- **Descripción:** Devuelve la tabla de instancias.
- **Prototipo:** `getInstanceTable(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** La tabla de instancias.

2.5.2.4 Método `registerStorage()`

- **Descripción:** Registra un almacenamiento para su posterior uso.
 - **Prototipo:** `registerStorage(self, name, storage)`
 - **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre con el que lo registramos.
 - storage:** El almacenamiento.
 - **Valor de retorno:** Ninguno.
-

2.5.2.5 Método unregisterStorage()

- **Descripción:** Elimina un almacenamiento de la lista de almacenamientos registrados.
- **Prototipo:** unregisterStorage(self, name)
- **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre del almacenamiento.
- **Valor de retorno:** El almacenamiento eliminado.

2.5.2.6 Método getStorage()

- **Descripción:** Devuelve el almacenamiento de nombre dado.
- **Prototipo:** getStorage(self, name)
- **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre del almacenamiento.
- **Valor de retorno:** El almacenamiento pedido.

2.5.2.7 Método setStorage()

- **Descripción:** Elige el almacenamiento a usar.
- **Prototipo:** setStorage(self, name)
- **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre del almacenamiento.
- **Valor de retorno:** Ninguno.

2.5.2.8 Método registerPolicy()

- **Descripción:** Registra una política para su posterior uso.
- **Prototipo:** registerPolicy(self, name, policy)
- **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre con el que la registramos.
 - storage:** La política.
- **Valor de retorno:** Ninguno.

2.5.2.9 Método unregisterPolicy()

- **Descripción:** Elimina una política de la lista de políticas registradas.
- **Prototipo:** unregisterPolicy(self, name)
- **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre de la política.
- **Valor de retorno:** La política eliminada.

2.5.2.10 Método getPolicy()

- **Descripción:** Devuelve la política de nombre dado.
- **Prototipo:** getPolicy(self, name)
- **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre de la política.
- **Valor de retorno:** La política pedida.

2.5.2.11 Método setPolicy()

- **Descripción:** Elige la política a usar.
- **Prototipo:** setPolicy(self, name)
- **Argumentos:**
 - self:** La propia instancia.
 - name:** Nombre de la política.
- **Valor de retorno:** Ninguno.

2.5.2.12 Método retrieveObject()

- **Descripción:** Carga una instancia desde el almacenamiento actual.
 - **Prototipo:** retrieveObject(self, id)
 - **Argumentos:**
 - self:** La propia instancia.
 - id:** GUID de la instancia a recuperar.
 - **Valor de retorno:** La instancia cargada.
-

2.5.2.13 Método `storeObject()`

- **Descripción:** Guarda una instancia en el almacenamiento actual.
- **Prototipo:** `storeObject(self, instance)`
- **Argumentos:**
 - self:** La propia instancia.
 - instance:** Instancia a almacenar.
- **Valor de retorno:** El GUID de la instancia.

2.5.2.14 Método `makePersistent()`

- **Descripción:** Hace que una instancia y sus miembros sean persistentes.
- **Prototipo:** `makePersistent(self, instance)`
- **Argumentos:**
 - self:** La propia instancia.
 - instance:** Instancia a hacer persistente.
- **Valor de retorno:** Ninguno.

2.5.2.15 Método `makeTransient()`

- **Descripción:** Hace que una instancia y sus miembros dejen de ser persistentes.
- **Prototipo:** `makeTransient(self, instance)`
- **Argumentos:**
 - self:** La propia instancia.
 - instance:** Instancia que dejará de ser persistente.
- **Valor de retorno:** Ninguno.

2.5.2.16 Método `commit()`

- **Descripción:** Marca el final de una transacción.
 - **Prototipo:** `commit(self)`
 - **Argumentos:**
 - self:** La propia instancia.
 - **Valor de retorno:** Ninguno.
-

2.5.2.17 Método `terminate()`

- **Descripción:** Se llama al terminar el programa, se encarga de hacer que los almacenamientos y políticas finalicen.
- **Prototipo:** `terminate(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Ninguno.

2.5.2.18 Método `notifyModified()`

- **Descripción:** Marca una instancia como modificada, para que la política sepa qué debe guardar.
- **Prototipo:** `notifyModified(self, instance, modified)`
- **Argumentos:**
 - self:** La propia instancia.
 - instance:** La instancia a marcar.
 - modified:** Si se modificó o no.
- **Valor de retorno:** Ninguno.

2.5.3 Clase `Storage`

Un almacenamiento genérico.

2.5.3.1 Atributos

- **interpreter:** El intérprete.
- **manager:** El gestor de persistencia.
- **instanceTable:** La tabla de instancias.

2.5.3.2 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self, interpreter)`
- **Argumentos:**
 - self:** La propia instancia.
 - interpreter:** El intérprete.
- **Valor de retorno:** Ninguno.

2.5.3.3 Método setManager()

- **Descripción:** Le indica al almacenamiento cuál es el gestor de persistencia con el que está registrado.
- **Prototipo:** setManager(self, manager)
- **Argumentos:**
 - self:** La propia instancia.
 - manager:** El gestor de persistencia.
- **Valor de retorno:** Ninguno.

2.5.3.4 Método storeObject()

- **Descripción:** Interfaz con cada almacenamiento concreto para guardar una instancia.
- **Prototipo:** storeObject(self, inst)
- **Argumentos:**
 - self:** La propia instancia.
 - inst:** La instancia a guardar.
- **Valor de retorno:** Ninguno.

2.5.3.5 Método retrieveObject()

- **Descripción:** Interfaz con cada almacenamiento concreto para cargar una instancia.
- **Prototipo:** retrieveObject(self, id)
- **Argumentos:**
 - self:** La propia instancia.
 - id:** El GUID de la instancia a cargar.
- **Valor de retorno:** La instancia cargada.

2.5.4 Clase SimpleStorage

Un almacenamiento sencillo: consiste en una tabla en memoria con las instancias almacenadas. Dicha tabla se carga o guarda entera en disco cada vez.

2.5.4.1 Atributos

- **instances:** Tabla con las instancias a guardar.
 - **appName:** Nombre de la aplicación.
 - **fileName:** Nombre del fichero en que se guardarán las instancias.
-

2.5.4.2 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self, interpreter, appName)`
- **Argumentos:**
 - self:** La propia instancia.
 - interpreter:** El intérprete.
 - appName:** Nombre de la aplicación.
- **Valor de retorno:** Ninguno.

2.5.4.3 Método `has_key()`

- **Descripción:** Pregunta si en el almacenamiento hay guardada una instancia con un cierto GUID.
- **Prototipo:** `has_key(self, id)`
- **Argumentos:**
 - self:** La propia instancia.
 - id:** El GUID de la instancia a cargar.
- **Valor de retorno:** Un booleano que indica si la instancia está almacenada o no.

2.5.4.4 Método `doRetrieveObject()`

- **Descripción:** Carga una instancia.
- **Prototipo:** `doRetrieveObject(self, id)`
- **Argumentos:**
 - self:** La propia instancia.
 - id:** El GUID de la instancia a cargar.
- **Valor de retorno:** La instancia.

2.5.4.5 Método `doStoreObject()`

- **Descripción:** Almacena una instancia.
 - **Prototipo:** `doStoreObject(self, id, inst)`
 - **Argumentos:**
 - self:** La propia instancia.
 - id:** El GUID de la instancia a cargar.
 - inst:** La instancia a almacenar.
 - **Valor de retorno:** Ninguno.
-

2.5.4.6 Método `commit()`

- **Descripción:** Marca el final de una transacción.
- **Prototipo:** `commit(self)`
- **Argumentos:**
 - `self`: La propia instancia.
- **Valor de retorno:** Ninguno.

2.5.4.7 Método `terminate()`

- **Descripción:** Se debe llamar cuando no se vaya a usar más un almacenamiento.
- **Prototipo:** `terminate(self)`
- **Argumentos:**
 - `self`: La propia instancia.
- **Valor de retorno:** Ninguno.

2.5.4.8 Método `dump()`

- **Descripción:** Vuelca la tabla de instancias almacenadas al disco.
- **Prototipo:** `dump(self, filename)`
- **Argumentos:**
 - `self`: La propia instancia.
 - `filename`: Nombre del fichero en donde se almacenará.
- **Valor de retorno:** Ninguno.

2.5.4.9 Método `load()`

- **Descripción:** Carga la tabla de instancias almacenadas desde el disco.
- **Prototipo:** `load(self, filename)`
- **Argumentos:**
 - `self`: La propia instancia.
 - `filename`: Nombre del fichero desde donde se cargará.
- **Valor de retorno:** Ninguno.

2.5.5 Clase `DBMStorage`

Almacenamiento basado en la funcionalidad del módulo `dbm` de Python.

2.5.5.1 Atributos

- **appName:** Nombre de la aplicación.
 - **fileName:** Nombre del fichero en que se guardarán las instancias.
 - **dbm:** Base de datos DBM que usamos para guardar las instancias.
-

2.5.5.2 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self, interpreter, appName)`
- **Argumentos:**
 - self:** La propia instancia.
 - interpreter:** El intérprete.
 - appName:** Nombre de la aplicación.
- **Valor de retorno:** Ninguno.

2.5.5.3 Método `has_key()`

- **Descripción:** Pregunta si en el almacenamiento hay guardada una instancia con un cierto GUID.
- **Prototipo:** `has_key(self, id)`
- **Argumentos:**
 - self:** La propia instancia.
 - id:** El GUID de la instancia a cargar.
- **Valor de retorno:** Un booleano que indica si la instancia está almacenada o no.

2.5.5.4 Método `doRetrieveObject()`

- **Descripción:** Carga una instancia.
- **Prototipo:** `doRetrieveObject(self, id)`
- **Argumentos:**
 - self:** La propia instancia.
 - id:** El GUID de la instancia a cargar.
- **Valor de retorno:** La instancia.

2.5.5.5 Método `doStoreObject()`

- **Descripción:** Almacena una instancia.
 - **Prototipo:** `doStoreObject(self, id, inst)`
 - **Argumentos:**
 - self:** La propia instancia.
 - id:** El GUID de la instancia a cargar.
 - inst:** La instancia a almacenar.
 - **Valor de retorno:** Ninguno.
-

2.5.5.6 Método `commit()`

- **Descripción:** Marca el final de una transacción.
- **Prototipo:** `commit(self)`
- **Argumentos:**
 - `self:` La propia instancia.
- **Valor de retorno:** Ninguno.

2.5.5.7 Método `terminate()`

- **Descripción:** Se debe llamar cuando no se vaya a usar más un almacenamiento.
- **Prototipo:** `terminate(self)`
- **Argumentos:**
 - `self:` La propia instancia.
- **Valor de retorno:** Ninguno.

2.5.6 Clase `BSDDBStorage`

Almacenamiento basado en la funcionalidad del módulo `bsddb` de Python.

2.5.6.1 Atributos

- **appName:** Nombre de la aplicación.
- **fileName:** Nombre del fichero en que se guardarán las instancias.
- **db:** Base de datos BSD que usamos para guardar las instancias.

2.5.6.2 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self, interpreter, appName)`
- **Argumentos:**
 - `self:` La propia instancia.
 - `interpreter:` El intérprete.
 - `appName:` Nombre de la aplicación.
- **Valor de retorno:** Ninguno.

2.5.6.3 Método `has_key()`

- **Descripción:** Pregunta si en el almacenamiento hay guardada una instancia con un cierto GUID.
- **Prototipo:** `has_key(self, id)`
- **Argumentos:**
 - self:** La propia instancia.
 - id:** El GUID de la instancia a cargar.
- **Valor de retorno:** Un booleano que indica si la instancia está almacenada o no.

2.5.6.4 Método `doRetrieveObject()`

- **Descripción:** Carga una instancia.
- **Prototipo:** `doRetrieveObject(self, id)`
- **Argumentos:**
 - self:** La propia instancia.
 - id:** El GUID de la instancia a cargar.
- **Valor de retorno:** La instancia.

2.5.6.5 Método `doStoreObject()`

- **Descripción:** Almacena una instancia.
- **Prototipo:** `doStoreObject(self, id, inst)`
- **Argumentos:**
 - self:** La propia instancia.
 - id:** El GUID de la instancia a cargar.
 - inst:** La instancia a almacenar.
- **Valor de retorno:** Ninguno.

2.5.6.6 Método `commit()`

- **Descripción:** Marca el final de una transacción.
 - **Prototipo:** `commit(self)`
 - **Argumentos:**
 - self:** La propia instancia.
 - **Valor de retorno:** Ninguno.
-

2.5.6.7 Método terminate()

- **Descripción:** Se debe llamar cuando no se vaya a usar más un almacenamiento.
- **Prototipo:** terminate(self)
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Ninguno.

2.5.7 Clase StoragePolicy

Una política genérica.

2.5.7.1 Atributos

- **modifiedInstances:** Tabla de instancias modificadas y que se deben almacenar.
- **manager:** El gestor de persistencia.

2.5.7.2 Método __init__()

- **Descripción:** Constructor.
- **Prototipo:** __init__(self)
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Ninguno.

2.5.7.3 Método setManager()

- **Descripción:** Le indica la política cuál es el gestor de persistencia con el que está registrada.
- **Prototipo:** setManager(self, manager)
- **Argumentos:**
 - self:** La propia instancia.
 - manager:** El gestor de persistencia.
- **Valor de retorno:** Ninguno.

2.5.7.4 Método addInstance()

- **Descripción:** Añade una instancia a la lista de instancias modificadas.
 - **Prototipo:** addInstance(self, instance)
 - **Argumentos:**
 - self:** La propia instancia.
 - instance:** La instancia a añadir.
 - **Valor de retorno:** Ninguno.
-

2.5.7.5 Método `deleteInstance()`

- **Descripción:** Elimina una instancia de la lista de instancias modificadas.
- **Prototipo:** `deleteInstance(self, instance)`
- **Argumentos:**
 - self:** La propia instancia.
 - instance:** La instancia a eliminar.
- **Valor de retorno:** Ninguno.

2.5.7.6 Método `storeInstances()`

- **Descripción:** Almacena todas las instancias que se hayan marcado como modificadas, y avisa al gestor para que el almacenamiento haga *commit* a su vez.
- **Prototipo:** `storeInstances(self)`
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Ninguno.

2.5.8 Clase `SimplePolicy`

Una política sencilla: cada n modificaciones, efectúa un *commit*.

2.5.8.1 Atributos

- **maxChanges:** Número máximo de cambios antes de efectuar un *commit*.
- **numChanges:** Número de cambios que se han efectuado desde el último *commit*.

2.5.8.2 Método `__init__()`

- **Descripción:** Constructor.
- **Prototipo:** `__init__(self, maxChanges = 1)`
- **Argumentos:**
 - self:** La propia instancia.
 - maxChanges:** Número máximo de cambios antes de un *commit*.
- **Valor de retorno:** Ninguno.

2.5.8.3 Método `notifyModified()`

- **Descripción:** Marca una instancia como modificada.
 - **Prototipo:** `notifyModified(self, instance, modified)`
 - **Argumentos:**
 - self:** La propia instancia.
-

instance: La instancia a marcar.

modified: Si se marca como modificada o no.

- **Valor de retorno:** Ninguno.

2.5.8.4 Método `commit()`

- **Descripción:** Marca el final de una transacción.

- **Prototipo:** `commit(self)`

- **Argumentos:**

self: La propia instancia.

- **Valor de retorno:** Ninguno.

2.5.8.5 Método `terminate()`

- **Descripción:** Se debe llamar cuando no se vaya a usar más una política.

- **Prototipo:** `terminate(self)`

- **Argumentos:**

self: La propia instancia.

- **Valor de retorno:** Ninguno.

2.5.9 Clase `TimedPolicy`

Política que espera n segundos desde que se efectúa un cambio antes de hacer el `commit`, dando tiempo para que se puedan hacer más modificaciones.

2.5.9.1 Atributos

- **seconds:** Número de segundos que esperaremos desde el primer cambio hasta que efectuemos un `commit`.
- **timer:** El temporizador que usaremos para contar el tiempo, o `None` si no se activó.
- **lock:** El cerrojo que usaremos para evitar problemas de concurrencia al acceder al almacenamiento.

2.5.9.2 Método `__init__()`

- **Descripción:** Constructor.

- **Prototipo:** `__init__(self)`

- **Argumentos:**

self: La propia instancia.

- **Valor de retorno:** Ninguno.
-

2.5.9.3 Método notifyModified()

- **Descripción:** Marca una instancia como modificada.
- **Prototipo:** notifyModified(self, instance, modified)
- **Argumentos:**
 - self:** La propia instancia.
 - instance:** La instancia a marcar.
 - modified:** Si se marca como modificada o no.
- **Valor de retorno:** Ninguno.

2.5.9.4 Método commit()

- **Descripción:** Marca el final de una transacción.
- **Prototipo:** commit(self)
- **Argumentos:**
 - self:** La propia instancia.
- **Valor de retorno:** Ninguno.

2.5.9.5 Método terminate()

- **Descripción:** Se debe llamar cuando no se vaya a usar más una política.
 - **Prototipo:** terminate(self)
 - **Argumentos:**
 - self:** La propia instancia.
 - **Valor de retorno:** Ninguno.
-

Capítulo 3

GRAMÁTICA DEL LENGUAJE JAVA— —

La siguiente gramática es la finalmente implementada para la interpretación del lenguaje Java—. No es idéntica a la presentada en el diseño por las limitaciones de la herramienta, al no permitir ésta partes repetitivas de forma automática, por lo que hubo que modificar la gramática hasta obtener una equivalente y que nitrO aceptase.

```
<S> ::= <bodyStatements>
<bodyStatements> ::= <classDef> <bodyStatements>
| <main>
<main> ::= <statementBlock>
<classDef> ::= CLASS IDENT <extends> LCURLY <members> RCURLY
<extends> ::= EXTENDS IDENT
| λ
<members> ::= IDENT LPAREN <args> RPAREN <statementBlock> <members>
| VOID IDENT LPAREN <args> RPAREN <statementBlock>
| <members>
| IDENT IDENT LPAREN <args> RPAREN <statementBlock>
| <members>
| IDENT IDENT ASSIGN <conditionalExpr> <moreFieldDecls>
| <members>
| IDENT IDENT <moreFieldDecls> <members>
| λ
<args> ::= IDENT IDENT <moreArgs>
| λ
<moreArgs> ::= COMMA IDENT IDENT <moreArgs>
| λ
<moreFieldDecls> ::= COMMA IDENT ASSIGN <conditionalExpr> <moreFieldDecls>
| COMMA IDENT <moreFieldDecls>
| SEMI
<statements> ::= <statement> <statements>
| λ
<statement> ::= <statementBlock>
| _REIFY_
| IF LPAREN <optimizedExpressions> RPAREN <statement>
| <elseBlock>
| WHILE LPAREN <optimizedExpressions> RPAREN <statement>
| FOR LPAREN <forInit> SEMI <optimizedExpressions> SEMI
| <optimizedExpressions> RPAREN <statement>
| <simpleStatement> SEMI
```

<code><elseBlock></code>	::=	ELSE <code><statement></code> λ
<code><forInit></code>	::=	IDENT IDENT ASSIGN <code><assignExpr></code> <code><moreVarDecls></code> IDENT IDENT <code><moreVarDecls></code> <code><optimizedExpressions></code> λ
<code><statementBlock></code>	::=	LCURLY <code><statements></code> RCURLY
<code><simpleStatement></code>	::=	SUPER LPAREN <code><argExprList></code> RPAREN RETURN <code><optimizedExpressions></code> RETURN IDENT IDENT ASSIGN <code><assignExpr></code> <code><moreVarDecls></code> IDENT IDENT <code><moreVarDecls></code> <code><optimizedExpressions></code> λ
<code><moreVarDecls></code>	::=	COMMA IDENT ASSIGN <code><assignExpr></code> <code><moreVarDecls></code> COMMA IDENT <code><moreVarDecls></code> λ
<code><optimizedExpressions></code>	::=	<code><expressions></code>
<code><expressions></code>	::=	<code><assignExpr></code> <code><moreExpressions></code>
<code><moreExpressions></code>	::=	COMMA <code><assignExpr></code> <code><moreExpressions></code> λ
<code><assignExpr></code>	::=	<code><conditionalExpr></code> <code><moreAssignExpr></code>
<code><moreAssignExpr></code>	::=	ASSIGN <code><assignExpr></code> λ
<code><conditionalExpr></code>	::=	<code><logicalOrExpr></code> <code><moreConditionalExpr></code>
<code><moreConditionalExpr></code>	::=	QUESTION <code><assignExpr></code> COLON <code><conditionalExpr></code> λ
<code><logicalOrExpr></code>	::=	<code><logicalAndExpr></code> <code><moreLogicalOrExpr></code>
<code><moreLogicalOrExpr></code>	::=	LOR <code>logicalAndExpr</code> λ
<code><logicalAndExpr></code>	::=	<code><equalityExpr></code> <code><moreLogicalAndExpr></code>
<code><moreLogicalAndExpr></code>	::=	LAND <code><equalityExpr></code> λ
<code><<equalityExpr>></code>	::=	<code><relationalExpr></code> <code><moreEqualityExpr></code>
<code><moreEqualityExpr></code>	::=	EQUAL <code><relationalExpr></code> NOTEQUAL <code><relationalExpr></code> IS <code><relationalExpr></code> λ
<code><relationalExpr></code>	::=	<code><additiveExpr></code> <code><moreRelationalExpr></code>
<code><moreRelationalExpr></code>	::=	LT <code><additiveExpr></code> LTE <code><additiveExpr></code> GT <code><additiveExpr></code> GTE <code><additiveExpr></code> λ
<code><additiveExpr></code>	::=	<code><multExpr></code> <code><moreAdditiveExpr></code>
<code><moreAdditiveExpr></code>	::=	PLUS <code><multExpr></code> <code><moreAdditiveExpr></code> MINUS <code><multExpr></code> <code><moreAdditiveExpr></code> λ
<code><multExpr></code>	::=	<code><unaryExpr></code> <code><moreMultExpr></code>
<code><moreMultExpr></code>	::=	MUL <code><unaryExpr></code> <code><moreMultExpr></code> DIV <code><unaryExpr></code> <code><moreMultExpr></code>

		MOD <unaryExpr> <moreMultExpr>
		λ
<unaryExpr>	::=	LPAREN IDENT RPAREN <unaryExpr>
		PLUS <unaryExpr>
		MINUS <unaryExpr>
		LNOT <unaryExpr>
		INC <unaryExpr>
		DEC <unaryExpr>
		<postfixExpr>
<postfixExpr>	::=	<primaryExpr> <postfixSuffixes>
<postfixSuffixes>	::=	DOT IDENT LPAREN <argExprList> RPAREN <postfixSuffixes>
		DOT IDENT <postfixSuffixes>
		INC
		DEC
		λ
<argExprList>	::=	<expressions>
		λ
<primaryExpr>	::=	NEW IDENT LPAREN <argExprList> RPAREN
		NULL
		THIS
		TRUE
		SUPER DOT <superAccess>
		IDENT LPAREN <argExprList> RPAREN
		IDENT
		INTEGER
		STRING
		LPAREN <optimizedExpressions> RPAREN
<superAccess>	::=	SUPER DOT <superAccess>
		IDENT LPAREN <argExprList> RPAREN
		IDENT

Apéndice A

CÓDIGO FUENTE

A.1 Java.ml

```
1 Language = Java
2
3 Scanner = {
4   "Dígito"
5   digit -> "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
6
7   "Carácter válido para un identificador"
8   char ->
9       "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i"
10      | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r"
11      | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
12      | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I"
13      | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R"
14      | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
15      | "_"
16      ;
17
18   "Carácter o dígito"
19   charOrDigit -> char | digit ;
20
21   "Cero o más caracteres o dígitos"
22   moreCharsOrDigits -> charOrDigit moreCharsOrDigits | ;
23
24   "Cero o más dígitos"
25   moreDigits -> digit moreDigits | ;
26
27   "Identificador"
28   IDENT -> char moreCharsOrDigits ;
29
30   "Entero"
31   INTEGER -> digit moreDigits ;
32
33   "Carácter válido para las cadenas de texto"
34   stringChar ->
35       char
36       | digit
37       | "ñ" | "Ñ"
38       | "á" | "é" | "í" | "ó" | "ú"
39       | "Á" | "É" | "Í" | "Ó" | "Ú"
40       | "." | "," | ":" | ";" | "=" | "#" | "$" | "%" | "&" | "|"
41       | "(" | ")" | "{" | "}" | "[" | "]" | "+" | "-" | "*" | "/"
42       | "<" | ">" | "¡" | "!" | "¿" | "?" | "@" | "_" | " "
43       ;
44 }
```

```
45      "Secuencia de caracteres de una cadena"
46      stringChars -> stringChar stringChars | ;
47
48      "Cadena"
49      STRING -> "" stringChars "" ;
50
51      "Punto y coma"
52      SEMI -> ";" ;
53
54      "Paréntesis izquierdo"
55      LPAREN -> "(" ;
56
57      "Paréntesis derecho"
58      RPAREN -> ")" ;
59
60      "Llave izquierda"
61      LCURLY -> "{" ;
62
63      "Llave derecha"
64      RCURLY -> "}" ;
65
66      "Más"
67      PLUS -> "+" ;
68
69      "Menos"
70      MINUS -> "-" ;
71
72      "Por"
73      MUL -> "*" ;
74
75      "Entre"
76      DIV -> "/" ;
77
78      "Módulo"
79      MOD -> "%" ;
80
81      "Asignación"
82      ASSIGN -> "=" ;
83
84      "Igualdad"
85      EQUAL -> "==" ;
86
87      "Desigualdad"
88      NOTEQUAL -> "!=" ;
89
90      "Identidad"
91      IS -> "is" ;
92
93      "Menor"
94      LT -> "<" ;
95
96      "Menor o igual"
97      LTE -> "<=" ;
98
99      "Mayor"
100     GT -> ">" ;
101
102     "Mayor o igual"
103     GTE -> ">=" ;
104
105     "Y lógico"
106     LAND -> "&&" ;
107
```

```
108      "O lógico"
109      LOR -> "||" ;
110
111      "Negación"
112      LNOT -> "!" ;
113
114      "Punto"
115      DOT -> "." ;
116
117      "Coma"
118      COMMA -> "," ;
119
120      "Interrogante"
121      QUESTION -> "?" ;
122
123      "Dos puntos"
124      COLON -> ":" ;
125
126      "Incremento"
127      INC -> "++" ;
128
129      "Decremento"
130      DEC -> "--" ;
131
132      "Palabra clave new"
133      NEW -> "new" ;
134
135      "Palabra clave true"
136      TRUE -> "true" ;
137
138      "Palabra clave false"
139      FALSE -> "false" ;
140
141      "Palabra clave if"
142      IF -> "if" ;
143
144      "Palabra clave else"
145      ELSE -> "else" ;
146
147      "Palabra clave while"
148      WHILE -> "while" ;
149
150      "Palabra clave for"
151      FOR -> "for" ;
152
153      "Palabra clave class"
154      CLASS -> "class" ;
155
156      "Palabra clave extends"
157      EXTENDS -> "extends" ;
158
159      "Palabra clave return"
160      RETURN -> "return" ;
161
162      "Palabra clave this"
163      THIS -> "this" ;
164
165      "Palabra clave true"
166      TRUE -> "true" ;
167
168      "Palabra clave super"
169      SUPER -> "super" ;
170
```

```

171     "Palabra clave null"
172     NULL -> "null" ;
173
174     "Palabra clave void"
175     VOID -> "void" ;
176 }
177
178 Parser = {
179     "Regla inicial"
180     S -> bodyStatements
181 <## S -> bodyStatements
182 try:
183     # Redirigimos la salida a la ventana
184
185     import sys
186
187     class FalseFile(object):
188         def __init__(self , write): self.__write = write
189         def write(self , str): self.__write(str)
190
191     sys.stdout = FalseFile(write)
192     sys.stderr = sys.stdout
193
194     #-----
195
196     # Importaciones varias
197
198     import traceback , interpreter
199     from objs import *
200     from pprint import pprint
201
202     #-----
203
204     # Funciones para automatizar partes de la generación de código
205     # y evitar el copypaste
206
207     global generate_invokeVirtual
208
209     def generate_invokeVirtual(tree , klass , name , args , argTypes):
210         #if not nodes[0].lvalue:
211         #     raise 'El operando de la llamada a métodos ' \
212         #         + ' debe ser un lvalue '
213
214         method = klass.getMethod(name , argTypes)
215
216         mangledName = method.getMangledName()
217         resultTree = ( 'invokevirtual' , tree , mangledName , args)
218         resultType = method.getReturnType()
219
220         return resultTree , resultType
221
222     #-----
223
224     # Tabla de símbolos específica para el análisis semántico
225
226     class SymbolTable(object):
227         def __init__(self):
228             self.currentClass = None
229             self.currentMethod = None
230             self.currentBlockVars = {}
231             self.vars = {}
232             self.blockStack = []
233             self.varStack = []

```

```

234     def classEnter(self , klass):
235         self.currentClass = klass
236         self.blockEnter()
237     def classExit(self):
238         self.currentClass = None
239         self.blockExit()
240     def methodEnter(self , method , args):
241         self.currentMethod = method
242         self.blockEnter()
243         for name, klass in args: self.setVar(name, klass)
244     def methodExit(self):
245         self.blockExit()
246         self.currentMethod = None
247     def blockEnter(self):
248         self.blockStack.append(self.currentBlockVars)
249         self.currentBlockVars = {}
250         self.varStack.append(self.vars)
251         self.vars = self.vars.copy()
252     def blockExit(self):
253         self.currentBlockVars = self.blockStack.pop()
254         self.vars = self.varStack.pop()
255     def setVar(self , name, type):
256         if name in self.currentBlockVars:
257             raise 'La variable %s ya está definida' \
258                 + ' en el bloque actual' % name
259         self.currentBlockVars[name] = type
260         self.vars[name] = type
261     def getVar(self , name):
262         if name not in self.vars:
263             raise ('La variable %s no está definida' \
264                 + ' en el método %s') \
265                 % (name, self.currentMethod)
266         return self.vars[name]
267     def getCurrentClass(self): return self.currentClass
268
269     global symbolTable
270     symbolTable = SymbolTable()
271
272     global classes
273     classes = {} # Mapeado nombre -> clase
274
275     global getClass
276     def getClass(name):
277         try:
278             return classes[name]
279         except KeyError , e:
280             raise 'La clase %s no está definida' % name
281
282     for i in interpreter.primitiveClasses: classes[i.getName()] = i
283
284     print 'Precargando clases...'
285     nodes[1].classMap = {}
286     nodes[1].tree = ()
287     nodes[1].execute()
288
289     #-----
290
291     # Ordenamos las clases de acuerdo a su grafo de dependencias
292
293     print 'Ordenando clases...'
294
295     global classMap
296     classMap = nodes[1].classMap

```

```

297     classList = []
298     userClasses = classMap.keys()
299     global allClasses # Para que no se queje luego la función del filter
300     allClasses = classes.keys() # Las de usuario las metemos luego poco a poco
301     def classFilterFunc(name):
302         parent = classMap[name][1]
303         return parent in allClasses
304     while userClasses:
305         tmp = filter(classFilterFunc , userClasses)
306         for c in tmp: userClasses.remove(c)
307         allClasses += tmp
308         classList += tmp
309
310     #-----
311
312     # Creamos las clases
313
314     print 'Creando clases...'
315     for c in classList:
316         parent = classMap[c][1]
317         try:
318             parentObj = getClass(parent)
319         except KeyError, e:
320             raise 'La clase base %s no está definida' % parent
321         classes[c] = JClass(c, parentObj)
322
323     #-----
324
325     # Cogemos los campos y métodos de cada clase
326
327     print 'Precargando miembros de las clases...'
328
329     global classFields , classMethods , classConstructors
330     classFields = {}
331     classMethods = {}
332     classConstructors = {}
333
334     for c in classList:
335         klass = classMap[c]
336         symbolTable.classEnter(classes[c])
337         members = klass[2]
338         members.fields = {}
339         members.methods = []
340         members.constructors = []
341         members.execute()
342         classFields[c] = members.fields
343         classMethods[c] = members.methods
344         classConstructors[c] = members.constructors
345
346     #-----
347
348     # Creamos los campos y métodos de cada clase y se los añadimos
349
350     global currentReturnType
351     global parsingConstructors , firstStatement , hasSuperStatement
352
353     global theInterpreter
354     theInterpreter = interpreter.Interpreter(nodes[0].application , classes)
355     theInterpreter.write = write
356
357     print 'Creando miembros de las clases...'
358
359     pendingMethods = {}

```

```

360     pendingConstructors = {}
361
362     for c in classList:
363         klass = classes[c]
364         symbolTable.classEnter(klass)
365
366         klass.updateFromParent()
367
368         for f in classFields[c].values():
369             field = JField(f[0], f[1], theInterpreter.parse(f[2]))
370             klass.addField(field)
371
372     pm = pendingMethods[klass] = []
373     for m in classMethods[c]:
374         name = m[0]
375         currentReturnType = classes[m[1]]
376         args = [(x[0], classes[x[1]]) for x in m[2]]
377         codeNode = m[3]
378         method = JMethod(name, currentReturnType, args)
379         klass.addMethod(method)
380         pm.append((method, currentReturnType, args, codeNode))
381
382     pc = pendingConstructors[klass] = []
383     for m in classConstructors[c]:
384         name = m[0]
385         args = [(x[0], classes[x[1]]) for x in m[1]]
386         codeNode = m[2]
387         constructor = JConstructor(klass, args)
388         klass.addConstructor(constructor)
389         pc.append((constructor, args, codeNode))
390
391     # Se añade el constructor por defecto si no tiene ninguno
392     klass.addDefaultConstructor()
393
394     symbolTable.classExit()
395
396     for klass in pendingMethods.keys():
397         symbolTable.classEnter(klass)
398
399         for f in klass.getFields().values():
400             symbolTable.setVar(f.getName(), f.getClass())
401
402     parsingConstructors = False
403     for m in pendingMethods[klass]:
404         method = m[0]
405         currentReturnType = m[1]
406         args = m[2]
407         codeNode = m[3]
408
409         symbolTable.methodEnter(method, args)
410         codeNode.execute()
411         symbolTable.methodExit()
412         code = UserCodeWrapper(codeNode.tree)
413         method.setCode(code)
414
415     parsingConstructors = True
416     currentReturnType = interpreter.voidClass
417     for c in pendingConstructors[klass]:
418         constructor = c[0]
419         args = c[1]
420         codeNode = c[2]
421
422         symbolTable.methodEnter(constructor, args)

```

```

423         firstStatement = True
424         hasSuperStatement = False
425         codeNode.execute()
426         symbolTable.methodExit()
427         code = UserCodeWrapper(codeNode.tree)
428         constructor.setCode(code)
429         constructor.setHasSuperStatement(hasSuperStatement)
430
431         symbolTable.classExit()
432
433         #-----
434
435         # Parseamos el main
436
437         print 'Precargando el programa principal...'
438         main = nodes[1].main
439         main.execute()
440         tree = main.tree
441
442         #pprint(tree)
443         print 'Ejecutando'
444         print '-' * 79
445
446         #-----
447
448         # A ejecutar
449
450         theInterpreter.parse(tree)
451         theInterpreter.terminate()
452
453     except SystemExit, e:
454         print e
455     except:
456         traceback.print_exc()
457 #>
458 ;
459
460     "Lo que puede ir en el cuerpo del programa"
461     bodyStatements ->
462         classDef bodyStatements
463 <## bodyStatements -> classDef bodyStatements
464     nodes[1].classMap = nodes[0].classMap
465     nodes[1].execute()
466     nodes[2].classMap = nodes[0].classMap
467     nodes[2].execute()
468     nodes[0].main = nodes[2].main
469 #>
470         | main
471 <## bodyStatements -> main
472     nodes[0].main = nodes[1]
473 #>
474         ;
475
476     "El programa principal (lo que sería el public static void main...)"
477     main ->
478         statementBlock
479 <## main -> statementBlock
480     nodes[1].execute()
481     nodes[0].tree = ('main', nodes[1].tree)
482 #>
483         ;
484
485     "Una clase"

```

```

486     classDef ->
487         CLASS IDENT extends LCURLY members RCURLY
488 <## classDef -> CLASS IDENT extends LCURLY members RCURLY
489 nodes[3].execute()
490
491 name = nodes[2].text
492 tmpClassMap = nodes[0].classMap
493 if (name in tmpClassMap) or (name in classes):
494     raise 'Ya existe una clase llamada %s' % name
495 tmpClassMap[name] = (name, nodes[3].parent, nodes[5])
496 #>
497     ;
498
499     "La clase padre, de haberla"
500     extends ->
501         EXTENDS IDENT
502 <## extends -> EXTENDS IDENT
503 nodes[0].parent = nodes[2].text
504 #>
505     |
506 <## extends ->
507 nodes[0].parent = 'Object' # En Java, Object es la "superclase cósmica"
508 #>
509     ;
510
511     "Los miembros"
512     members ->
513         IDENT LPAREN args RPAREN statementBlock members
514 <## members -> IDENT LPAREN args RPAREN statementBlock members
515 name = nodes[1].text
516 className = symbolTable.getCurrentClass().getName()
517 if name != className:
518     raise 'El constructor tiene que tener el mismo nombre que la clase (%s)' \
519         % className
520
521 nodes[3].execute()
522
523 constructors = nodes[0].constructors
524 constructors.append((name, nodes[3].args, nodes[5]))
525
526 nodes[6].methods = nodes[0].methods
527 nodes[6].fields = nodes[0].fields
528 nodes[6].constructors = constructors
529 nodes[6].execute()
530 #>
531     | VOID IDENT LPAREN args RPAREN statementBlock members
532 <## members -> VOID LPAREN args RPAREN statementBlock members
533 name = nodes[2].text
534 className = symbolTable.getCurrentClass().getName()
535 if name == className: raise 'El constructor no puede devolver un valor'
536
537 nodes[4].execute()
538
539 # Como tipo del método ponemos <void>, una pseudoclase que no derivará de
540 # Object; de esa forma, podemos hacer todas las comprobaciones de tipos sin
541 # necesidad de casos especiales (salvo para los nulos)
542 methods = nodes[0].methods
543 methods.append((name, '<void>', nodes[4].args, nodes[6]))
544
545 nodes[7].methods = methods
546 nodes[7].fields = nodes[0].fields
547 nodes[7].constructors = nodes[0].constructors
548 nodes[7].execute()

```

```

549 #>
550         | IDENT IDENT LPAREN args RPAREN statementBlock members
551 <## members -> IDENT IDENT LPAREN args RPAREN statementBlock members
552 name = nodes[2].text
553 className = symbolTable.getCurrentClass().getName()
554 if name == className: raise 'El constructor no puede devolver un valor'
555
556 nodes[4].execute()
557
558 methods = nodes[0].methods
559 methods.append((name, nodes[1].text, nodes[4].args, nodes[6]))
560
561 nodes[7].methods = methods
562 nodes[7].fields = nodes[0].fields
563 nodes[7].constructors = nodes[0].constructors
564 nodes[7].execute()
565 #>
566         | IDENT IDENT ASSIGN conditionalExpr moreFieldDecls members
567 <## members -> IDENT IDENT ASSIGN conditionalExpr moreFieldDecls members
568 fields = nodes[0].fields
569 name = nodes[2].text
570 klass = getClass(nodes[1].text)
571
572 if name in fields:
573     raise SystemExit('Ya existe un campo llamado %s' % name)
574
575 nodes[4].execute()
576 rtype = nodes[4].type
577 if not rtype.isA(klass):
578     raise 'No se puede asignar un %s a un %s' % (rtype, klass)
579 fields[name] = (name, klass, nodes[4].tree)
580
581 nodes[5].fields = fields
582 nodes[5].type = klass
583 nodes[5].execute()
584
585 nodes[6].fields = fields
586 nodes[6].methods = nodes[0].methods
587 nodes[6].constructors = nodes[0].constructors
588 nodes[6].execute()
589 #>
590         | IDENT IDENT moreFieldDecls members
591 <## members -> IDENT IDENT moreFieldDecls members
592 fields = nodes[0].fields
593 name = nodes[2].text
594 klass = getClass(nodes[1].text)
595
596 if name in fields:
597     raise SystemExit('Ya existe un campo llamado %s' % name)
598
599 fields[name] = (name, klass, ('null',))
600
601 nodes[3].fields = fields
602 nodes[3].type = klass
603 nodes[3].execute()
604
605 nodes[4].fields = fields
606 nodes[4].methods = nodes[0].methods
607 nodes[4].constructors = nodes[0].constructors
608 nodes[4].execute()
609 #>
610         |
611         ;

```

```

612
613     "Cero o más argumentos separados por comas"
614     args →
615         IDENT IDENT moreArgs
616 <#
617 nodes[3].args = ((nodes[2].text , nodes[1].text),)
618 nodes[3].execute()
619 nodes[0].args = nodes[3].args
620 #>
621         |
622 <#
623 nodes[0].args = ()
624 #>
625         ;
626
627     "Más argumentos separados por comas..."
628     moreArgs →
629         COMMA IDENT IDENT moreArgs
630 <#
631 args = nodes[0].args
632 name = nodes[3].text
633
634 duplicado = bool(filter(lambda i , name = name: i[0] == name, args))
635 if duplicado:
636     raise SystemExit('Nombre de parámetro duplicado: %s' % name)
637
638 nodes[4].args = args + ((name, nodes[2].text),)
639 nodes[4].execute()
640 nodes[0].args = nodes[4].args
641 #>
642         |
643         ;
644
645     "Más declaraciones de variables"
646     moreFieldDecls →
647         COMMA IDENT ASSIGN conditionalExpr moreFieldDecls
648 <#
649 fields = nodes[0].fields
650 name = nodes[2].text
651 klass = nodes[0].type
652
653 if name in fields:
654     raise SystemExit('Ya existe un campo llamado %s' % name)
655
656 nodes[4].execute()
657 rtype = nodes[4].type
658 if not rtype.isA(klass):
659     raise 'No se puede asignar un %s a un %s' % (rtype , klass)
660 fields[name] = (name, klass , nodes[4].tree)
661
662 nodes[5].fields = fields
663 nodes[5].type = klass
664 nodes[5].execute()
665 #>
666         | COMMA IDENT moreFieldDecls
667 <#
668 fields = nodes[0].fields
669 name = nodes[2].text
670 klass = nodes[0].type
671
672 if name in fields:
673     raise SystemExit('Ya existe un campo llamado %s' % name)
674

```

```

675 fields[name] = (name, klass, ( 'null' ,))
676
677 nodes[3].fields = fields
678 nodes[3].type = klass
679 nodes[3].execute()
680 #>
681         | SEMI
682         ;
683
684         "Cero o más sentencias"
685         statements ->
686             statement statements
687 <## statements -> statement statements
688 nodes[1].execute()
689 nodes[2].tree = (nodes[1].tree ,)
690 nodes[2].execute()
691 nodes[0].tree += nodes[2].tree
692 #>
693         |
694         ;
695
696         "Una sentencia"
697         statement ->
698             statementBlock
699 <## statement -> statementBlock
700 global firstStatement # SI NO SE MARCA COMO GLOBAL ASIGNA A UNA VARIABLE LOCAL
701 firstStatement = False
702 nodes[1].execute()
703 nodes[0].tree = nodes[1].tree
704 #>
705         | _REIFY_
706 <## statement -> _REIFY_
707 nodes[0].tree = ( 'reify' , nodes[1])
708 #>
709         | IF LPAREN optimizedExpressions RPAREN statement elseBlock
710 <## statement -> IF LPAREN optimizedExpressions RPAREN statement elseBlock
711 global firstStatement
712 firstStatement = False
713 nodes[3].execute()
714 nodes[5].execute()
715 nodes[6].execute()
716 nodes[0].tree = ( 'if' , nodes[3].tree , nodes[5].tree , nodes[6].tree)
717 #>
718         | WHILE LPAREN optimizedExpressions RPAREN statement
719 <## statement -> WHILE LPAREN optimizedExpressions RPAREN statement
720 global firstStatement
721 firstStatement = False
722 nodes[3].execute()
723 nodes[5].execute()
724 nodes[0].tree = ( 'while' , nodes[3].tree , nodes[5].tree)
725 #>
726         | FOR LPAREN forInit SEMI
727             optimizedExpressions SEMI
728             optimizedExpressions RPAREN statement
729 <## statement -> FOR LPAREN forInit SEMI
730 #         optimizedExpressions SEMI
731 #         optimizedExpressions RPAREN statement
732 global firstStatement
733 firstStatement = False
734 symbolTable.blockEnter()
735 nodes[3].execute()
736 nodes[5].execute()
737 nodes[7].execute()

```

```

738 nodes[9].execute()
739 symbolTable.blockExit()
740
741 nodes[0].tree = ( 'for' , nodes[3].tree , nodes[5].tree , nodes[7].tree ,
742                 nodes[9].tree )
743 #>
744         | simpleStatement SEMI
745 <## statements -> simpleStatement SEMI
746 nodes[1].execute()
747 nodes[0].tree = nodes[1].tree
748 global firstStatement
749 firstStatement = False
750 #>
751         ;
752
753         "El else del if"
754         elseBlock ->
755                 ELSE statement
756 <## elseBlock -> ELSE statement
757 nodes[2].execute()
758 nodes[0].tree = nodes[2].tree
759 #>
760         |
761 <## elseBlock ->
762 nodes[0].tree = (
763 #>
764         ;
765
766         "Posibles inicializaciones en el bucle del for"
767         forInIt ->
768                 IDENT IDENT ASSIGN assignExpr moreVarDecls
769 <## forInIt -> IDENT IDENT ASSIGN assignExpr moreVarDecls
770 className = nodes[1].text
771 klass = getClass(className)
772
773 nodes[4].execute()
774 rtype = nodes[4].type
775 if not rtype.isA(klass):
776         raise 'No se puede asignar un %s a un %s' % (rtype , klass)
777
778 nodes[5].tree = ((nodes[2].text , nodes[4].tree) ,)
779 nodes[5].type = klass
780 nodes[5].execute()
781
782 for (name, init) in nodes[5].tree:
783         symbolTable.setVar(name, klass)
784
785 nodes[0].tree = ( 'decl' , klass ) + nodes[5].tree
786 #>
787         | IDENT IDENT moreVarDecls
788 <## forInIt -> IDENT IDENT moreVarDecls
789 className = nodes[1].text
790 klass = getClass(className)
791
792 nodes[3].tree = ((nodes[2].text , ( 'null' , )) ,)
793 nodes[3].type = klass
794 nodes[3].execute()
795
796 for (name, init) in nodes[3].tree:
797         symbolTable.setVar(name, klass)
798
799 nodes[0].tree = ( 'decl' , klass ) + nodes[3].tree
800 #>

```

```

801         | optimizedExpressions
802 <## forInit -> optimizedExpressions
803 nodes[1].execute()
804 nodes[0].tree = nodes[1].tree
805 #>
806         |
807 <## forInit ->
808 nodes[0].tree = ()
809 #>
810         ;
811
812         "Bloque de sentencias"
813         statementBlock ->
814             LCURLY statements RCURLY
815 <## statementBlock -> LCURLY statements RCURLY
816 symbolTable.blockEnter()
817 nodes[2].tree = ()
818 nodes[2].execute()
819 nodes[0].tree = ( 'block' , ) + nodes[2].tree
820 symbolTable.blockExit()
821 #>
822         ;
823
824         "Una sentencia simple"
825         simpleStatement ->
826             SUPER LPAREN argExprList RPAREN
827 <## simpleStatement -> SUPER LPAREN argExprList RPAREN
828 if not parsingConstructors:
829     raise 'La construcción "super(args);" ' \
830         + 'sólo se puede usar en constructores '
831 if not firstStatement:
832     raise 'La construcción "super(args);" debe ser ' \
833         + ' la primera sentencia del constructor '
834
835 nodes[3].execute()
836
837 parent = symbolTable.getCurrentClass().getParent()
838 constructor = parent.getConstructor(nodes[3].types)
839
840 nodes[0].tree = ( 'invokespecial' , ( 'this' , ) , constructor , nodes[3].args )
841 nodes[0].type = constructor.getReturnType() # Obligatoriamente será <void>
842 nodes[0].lvalue = False
843
844 global hasSuperStatement
845 hasSuperStatement = True
846 #>
847         | RETURN optimizedExpressions
848 <## simpleStatement -> RETURN optimizedExpressions
849 nodes[2].execute()
850
851 klass = nodes[2].type
852 if not klass.isA(currentReturnType):
853     raise ( 'El tipo del return (%s) no es el del método (%s)' \
854         + ' ni un tipo derivado' ) \
855         % (klass , currentReturnType)
856
857 nodes[0].tree = ( 'return' , nodes[2].tree )
858 #>
859         | RETURN
860 <## simpleStatement -> RETURN
861 if getClass( '<void>' ) != currentReturnType:
862     raise 'El método necesita que se devuelva un valor '
863 nodes[0].tree = ( 'voidreturn' , )

```

```

864 #>
865         | IDENT IDENT ASSIGN assignExpr moreVarDecls
866 <## simpleStatement -> IDENT IDENT ASSIGN assignExpr moreVarDecls
867 className = nodes[1].text
868 klass = getClass(className)
869
870 nodes[4].execute()
871 rtype = nodes[4].type
872 if not rtype.isA(klass):
873     raise 'No se puede asignar un %s a un %s' % (rtype, klass)
874
875 nodes[5].tree = ((nodes[2].text, nodes[4].tree),)
876 nodes[5].type = klass
877 nodes[5].execute()
878
879 for (name, init) in nodes[5].tree:
880     symbolTable.setVar(name, klass)
881
882 nodes[0].tree = ('decl', klass) + nodes[5].tree
883 #>
884         | IDENT IDENT moreVarDecls
885 <## simpleStatement -> IDENT IDENT moreVarDecls
886 className = nodes[1].text
887 klass = getClass(className)
888
889 nodes[3].tree = ((nodes[2].text, ('null',)),)
890 nodes[3].type = klass
891 nodes[3].execute()
892
893 for (name, init) in nodes[3].tree:
894     symbolTable.setVar(name, klass)
895
896 nodes[0].tree = ('decl', klass) + nodes[3].tree
897 #>
898         | optimizedExpressions
899 <## simpleStatement -> optimizedExpressions
900 nodes[1].execute()
901 nodes[0].tree = nodes[1].tree
902 #>
903         |
904         ;
905
906         "Más declaraciones de variables"
907 moreVarDecls -> COMMA IDENT ASSIGN assignExpr moreVarDecls
908 <#
909 klass = nodes[0].type
910
911 nodes[4].execute()
912 rtype = nodes[4].type
913 if not rtype.isA(klass):
914     raise 'No se puede asignar un %s a un %s' % (rtype, klass)
915
916 nodes[5].tree = ((nodes[2].text, nodes[4].tree),)
917 nodes[5].type = klass
918 nodes[5].execute()
919
920 nodes[0].tree += nodes[5].tree
921 #>
922         | COMMA IDENT moreVarDecls
923 <#
924 nodes[3].tree = ((nodes[2].text, ('null',)),)
925 nodes[3].type = nodes[0].type
926 nodes[3].execute()

```

```

927 nodes[0].tree += nodes[3].tree
928 #>
929     |
930     ;
931
932     "Expresiones separadas por comas, pudiendo optimizar"
933     optimizedExpressions ->
934         expressions
935 <## optimizedExpressions -> expressions
936 nodes[1].execute()
937
938 if len(nodes[1].tree) > 1:
939     nodes[0].tree = ( 'exprs' ,) + nodes[1].tree
940 else:
941     nodes[0].tree = nodes[1].tree[0]
942 nodes[0].lvalue = nodes[1].lvalue
943 nodes[0].type = nodes[1].types[-1]
944 #>
945     ;
946
947     "Expresiones separadas por comas"
948     expressions ->
949         assignExpr moreExpressions
950 <#
951 nodes[1].execute()
952
953 nodes[2].lvalue = nodes[1].lvalue
954 nodes[2].execute()
955
956 nodes[0].tree = (nodes[1].tree ,) + nodes[2].tree
957 nodes[0].types = (nodes[1].type ,) + nodes[2].types
958 nodes[0].lvalue = nodes[2].lvalue
959 #>
960     ;
961
962     "Más expresiones separadas por comas"
963     moreExpressions ->
964         COMMA assignExpr moreExpressions
965 <#
966 nodes[2].execute()
967
968 nodes[3].lvalue = nodes[2].lvalue
969 nodes[3].execute()
970
971 nodes[0].tree = (nodes[2].tree ,) + nodes[3].tree
972 nodes[0].types = (nodes[2].type ,) + nodes[3].types
973 nodes[0].lvalue = nodes[3].lvalue
974 #>
975     |
976 <#
977 nodes[0].tree = ()
978 nodes[0].types = ()
979 #>
980     ;
981
982     "Una expresión de asignación"
983     assignExpr ->
984         conditionalExpr moreAssignExpr
985 <#
986 nodes[1].execute()
987
988 nodes[2].tree = nodes[1].tree
989 nodes[2].type = nodes[1].type

```

```

990 nodes[2].lvalue = nodes[1].lvalue
991 nodes[2].execute()
992
993 nodes[0].tree = nodes[2].tree
994 nodes[0].type = nodes[2].type
995 nodes[0].lvalue = nodes[2].lvalue
996 #>
997     ;
998
999     "Lado derecho de una asignación"
1000     moreAssignExpr ->
1001         ASSIGN assignExpr
1002 <#
1003 if not nodes[0].lvalue :
1004     raise 'No se puede asignar nada a algo que no sea un lvalue '
1005
1006 nodes[2].execute()
1007
1008 leftType = nodes[0].type
1009 rightType = nodes[2].type
1010 if not rightType.isA(leftType):
1011     raise 'No se puede asignar un %s a un %s' % (rightType , leftType)
1012
1013 nodes[0].tree = ( 'assign' , nodes[0].tree , nodes[2].tree )
1014 nodes[0].type = nodes[2].type
1015 #nodes[0].lvalue = True # No hace falta , ya comprobamos que era cierto
1016 #>
1017     |
1018     ;
1019
1020     "Expresión condicional"
1021     conditionalExpr ->
1022         logicalOrExpr moreConditionalExpr
1023 <#
1024 nodes[1].execute()
1025
1026 nodes[2].tree = nodes[1].tree
1027 nodes[2].type = nodes[1].type
1028 nodes[2].lvalue = nodes[1].lvalue
1029 nodes[2].execute()
1030
1031 nodes[0].tree = nodes[2].tree
1032 nodes[0].type = nodes[2].type
1033 nodes[0].lvalue = nodes[2].lvalue
1034 #>
1035     ;
1036
1037     "La segunda parte de la expr. condicional (el operador ternario)"
1038     moreConditionalExpr ->
1039         QUESTION assignExpr COLON conditionalExpr
1040 <#
1041 nodes[2].execute()
1042 nodes[4].execute()
1043 nodes[0].tree = ( 'if' , nodes[0].tree , nodes[2].tree , nodes[4].tree )
1044 nodes[0].lvalue = False
1045
1046 left = nodes[2].type
1047 right = nodes[4].type
1048
1049 # Devolvemos como tipo de la expresión el más genérico
1050 if left.isA(right):
1051     nodes[0].type = right
1052 elif right.isA(left):

```

```

1053         nodes[0].type = left
1054     else:
1055         raise 'Los tipos de las expresiones then-else del operador ternario' \
1056             ' no son compatibles'
1057     #>
1058         |
1059         ;
1060
1061         "OR lógico"
1062         logicalOrExpr ->
1063             logicalAndExpr moreLogicalOrExpr
1064     <#
1065     nodes[1].execute()
1066
1067     nodes[2].tree = nodes[1].tree
1068     nodes[2].type = nodes[1].type
1069     nodes[2].lvalue = nodes[1].lvalue
1070     nodes[2].execute()
1071
1072     nodes[0].tree = nodes[2].tree
1073     nodes[0].type = nodes[2].type
1074     nodes[0].lvalue = nodes[2].lvalue
1075     #>
1076         ;
1077
1078         "Más ORs lógicos"
1079         moreLogicalOrExpr ->
1080             LOR logicalAndExpr
1081     <#
1082     nodes[2].execute()
1083
1084     nodes[0].tree = ( 'lor' , nodes[0].tree , nodes[2].tree )
1085     #>
1086         |
1087         ;
1088
1089         "AND lógico"
1090         logicalAndExpr ->
1091             equalityExpr moreLogicalAndExpr
1092     <#
1093     nodes[1].execute()
1094
1095     nodes[2].tree = nodes[1].tree
1096     nodes[2].type = nodes[1].type
1097     nodes[2].lvalue = nodes[1].lvalue
1098     nodes[2].execute()
1099
1100     nodes[0].tree = nodes[2].tree
1101     nodes[0].type = nodes[2].type
1102     nodes[0].lvalue = nodes[2].lvalue
1103     #>
1104         ;
1105
1106         "Más ANDs lógicos"
1107         moreLogicalAndExpr ->
1108             LAND equalityExpr
1109     <#
1110     nodes[2].execute()
1111
1112     nodes[0].tree = ( 'land' , nodes[0].tree , nodes[2].tree )
1113     #>
1114         |
1115         ;

```

```

1116
1117     "Igualdad"
1118     equalityExpr ->
1119         relationalExpr moreEqualityExpr
1120 <#
1121 nodes[1].execute()
1122
1123 nodes[2].tree = nodes[1].tree
1124 nodes[2].type = nodes[1].type
1125 nodes[2].lvalue = nodes[1].lvalue
1126 nodes[2].execute()
1127
1128 nodes[0].tree = nodes[2].tree
1129 nodes[0].type = nodes[2].type
1130 nodes[0].lvalue = nodes[2].lvalue
1131 #>
1132     ;
1133
1134     "Más igualdades"
1135     moreEqualityExpr ->
1136         EQUAL relationalExpr
1137 <#
1138 nodes[2].execute()
1139
1140 tree , klass = generate_invokeVirtual(nodes[0].tree , nodes[0].type ,
1141     'operator==', [nodes[2].tree] , [nodes[2].type])
1142
1143 nodes[0].tree = tree
1144 nodes[0].type = klass
1145 nodes[0].lvalue = False
1146 #>
1147     | NOTEQUAL relationalExpr
1148 <#
1149 nodes[2].execute()
1150
1151 tree , klass = generate_invokeVirtual(nodes[0].tree , nodes[0].type ,
1152     'operator!=', [nodes[2].tree] , [nodes[2].type])
1153
1154 nodes[0].tree = tree
1155 nodes[0].type = klass
1156 nodes[0].lvalue = False
1157 #>
1158     | IS relationalExpr
1159 <#
1160 nodes[2].execute()
1161
1162 nodes[0].tree = ( 'is ' , nodes[0].tree , nodes[2].tree )
1163 #>
1164     |
1165     ;
1166
1167     "Relación"
1168     relationalExpr ->
1169         additiveExpr moreRelationalExpr
1170 <#
1171 nodes[1].execute()
1172
1173 nodes[2].tree = nodes[1].tree
1174 nodes[2].type = nodes[1].type
1175 nodes[2].lvalue = nodes[1].lvalue
1176 nodes[2].execute()
1177
1178 nodes[0].tree = nodes[2].tree

```

```

1179 nodes[0].type = nodes[2].type
1180 nodes[0].lvalue = nodes[2].lvalue
1181 #>
1182     ;
1183
1184     "Más relaciones"
1185     moreRelationalExpr ->
1186         LT additiveExpr
1187 <#
1188 nodes[2].execute()
1189
1190 tree , klass = generate_invokeVirtual(nodes[0].tree , nodes[0].type ,
1191     'operator<' , [nodes[2].tree] , [nodes[2].type])
1192
1193 nodes[0].tree = tree
1194 nodes[0].type = klass
1195 nodes[0].lvalue = False
1196 #>
1197     | LTE additiveExpr
1198 <#
1199 nodes[2].execute()
1200
1201 tree , klass = generate_invokeVirtual(nodes[0].tree , nodes[0].type ,
1202     'operator<=' , [nodes[2].tree] , [nodes[2].type])
1203
1204 nodes[0].tree = tree
1205 nodes[0].type = klass
1206 nodes[0].lvalue = False
1207 #>
1208     | GT additiveExpr
1209 <#
1210 nodes[2].execute()
1211
1212 tree , klass = generate_invokeVirtual(nodes[0].tree , nodes[0].type ,
1213     'operator>' , [nodes[2].tree] , [nodes[2].type])
1214
1215 nodes[0].tree = tree
1216 nodes[0].type = klass
1217 nodes[0].lvalue = False
1218 #>
1219     | GTE additiveExpr
1220 <#
1221 nodes[2].execute()
1222
1223 tree , klass = generate_invokeVirtual(nodes[0].tree , nodes[0].type ,
1224     'operator>=' , [nodes[2].tree] , [nodes[2].type])
1225
1226 nodes[0].tree = tree
1227 nodes[0].type = klass
1228 nodes[0].lvalue = False
1229 #>
1230     |
1231     ;
1232
1233     "Suma"
1234     additiveExpr ->
1235         multExpr moreAdditiveExpr
1236 <#
1237 nodes[1].execute()
1238
1239 nodes[2].tree = nodes[1].tree
1240 nodes[2].type = nodes[1].type
1241 nodes[2].lvalue = nodes[1].lvalue

```

```
1242 nodes[2].execute()
1243
1244 nodes[0].tree = nodes[2].tree
1245 nodes[0].type = nodes[2].type
1246 nodes[0].lvalue = nodes[2].lvalue
1247 #>
1248         ;
1249
1250         "Más sumas"
1251         moreAdditiveExpr ->
1252             PLUS multExpr moreAdditiveExpr
1253 <#
1254 nodes[2].execute()
1255
1256 tree , klass = generate_invokeVirtual(nodes[0].tree , nodes[0].type ,
1257     'operator+' , [nodes[2].tree] , [nodes[2].type])
1258
1259 nodes[3].tree = tree
1260 nodes[3].type = klass
1261 nodes[3].lvalue = False
1262 nodes[3].execute()
1263
1264 nodes[0].tree = nodes[3].tree
1265 nodes[0].type = nodes[3].type
1266 nodes[0].lvalue = nodes[3].lvalue
1267 #>
1268         | MINUS multExpr moreAdditiveExpr
1269 <#
1270 nodes[2].execute()
1271
1272 tree , klass = generate_invokeVirtual(nodes[0].tree , nodes[0].type ,
1273     'operator-' , [nodes[2].tree] , [nodes[2].type])
1274
1275 nodes[3].tree = tree
1276 nodes[3].type = klass
1277 nodes[3].lvalue = False
1278 nodes[3].execute()
1279
1280 nodes[0].tree = nodes[3].tree
1281 nodes[0].type = nodes[3].type
1282 nodes[0].lvalue = nodes[3].lvalue
1283 #>
1284         |
1285         ;
1286
1287         "Multiplicación"
1288         multExpr ->
1289             unaryExpr moreMultExpr
1290 <#
1291 nodes[1].execute()
1292
1293 nodes[2].tree = nodes[1].tree
1294 nodes[2].type = nodes[1].type
1295 nodes[2].lvalue = nodes[1].lvalue
1296 nodes[2].execute()
1297
1298 nodes[0].tree = nodes[2].tree
1299 nodes[0].type = nodes[2].type
1300 nodes[0].lvalue = nodes[2].lvalue
1301 #>
1302         ;
1303
1304         "Más multiplicaciones"
```

```

1305         moreMultExpr ->
1306             MUL unaryExpr moreMultExpr
1307 <#
1308 nodes[2].execute()
1309
1310 tree , klass = generate_invokeVirtual(nodes[0].tree , nodes[0].type ,
1311     'operator*' , [nodes[2].tree] , [nodes[2].type])
1312
1313 nodes[3].tree = tree
1314 nodes[3].type = klass
1315 nodes[3].lvalue = False
1316 nodes[3].execute()
1317
1318 nodes[0].tree = nodes[3].tree
1319 nodes[0].type = nodes[3].type
1320 nodes[0].lvalue = nodes[3].lvalue
1321 #>
1322         | DIV unaryExpr moreMultExpr
1323 <#
1324 nodes[2].execute()
1325
1326 tree , klass = generate_invokeVirtual(nodes[0].tree , nodes[0].type ,
1327     'operator/' , [nodes[2].tree] , [nodes[2].type])
1328
1329 nodes[3].tree = tree
1330 nodes[3].type = klass
1331 nodes[3].lvalue = False
1332 nodes[3].execute()
1333
1334 nodes[0].tree = nodes[3].tree
1335 nodes[0].type = nodes[3].type
1336 nodes[0].lvalue = nodes[3].lvalue
1337 #>
1338         | MOD unaryExpr moreMultExpr
1339 <#
1340 nodes[2].execute()
1341
1342 tree , klass = generate_invokeVirtual(nodes[0].tree , nodes[0].type ,
1343     'operator%' , [nodes[2].tree] , [nodes[2].type])
1344
1345 nodes[3].tree = tree
1346 nodes[3].type = klass
1347 nodes[3].lvalue = False
1348 nodes[3].execute()
1349
1350 nodes[0].tree = nodes[3].tree
1351 nodes[0].type = nodes[3].type
1352 nodes[0].lvalue = nodes[3].lvalue
1353 #>
1354         |
1355         ;
1356
1357     "Expresión unaria"
1358     unaryExpr ->
1359         LPAREN IDENT RPAREN unaryExpr
1360 <#
1361     className = nodes[2].text
1362     klass = getClass(className)
1363
1364     nodes[4].execute()
1365
1366     exprType = nodes[4].type
1367     if exprType.isA(klass):

```

```

1368         # Es un upcast, no hacen falta comprobaciones adicionales
1369         nodes[0].tree = nodes[4].tree
1370     elif klass.isA(exprType):
1371         # Es un downcast, sólo se puede resolver en ejecución
1372         nodes[0].tree = ( 'downcast' , klass , nodes[4].tree )
1373     else:
1374         raise 'Los tipos del cast y de la expresión no están relacionados'
1375
1376     nodes[0].type = klass
1377     nodes[0].lvalue = nodes[4].lvalue
1378     #>
1379         | PLUS postfixExpr
1380 <#
1381     nodes[2].execute()
1382
1383     nodes[0].tree = nodes[2].tree
1384     nodes[0].type = nodes[2].type
1385     nodes[0].lvalue = False
1386     #>
1387         | MINUS postfixExpr
1388 <#
1389     nodes[2].execute()
1390
1391     nodes[0].tree = ( 'neg' , nodes[1].tree )
1392     nodes[0].type = nodes[2].type
1393     nodes[0].lvalue = False
1394     #>
1395         | LNOT postfixExpr
1396 <#
1397     nodes[2].execute()
1398
1399     nodes[0].tree = ( 'Inot' , nodes[2].tree )
1400     nodes[0].type = getClass( 'Bool' )
1401     nodes[0].lvalue = False
1402     #>
1403         | INC unaryExpr
1404 <#
1405     nodes[2].execute()
1406
1407     tree , klass = generate_invokeVirtual(nodes[2].tree , nodes[2].type ,
1408         'operator++' , [], [])
1409     nodes[0].tree = tree
1410     nodes[0].type = klass
1411     nodes[0].lvalue = nodes[2].lvalue
1412     #>
1413         | DEC unaryExpr
1414 <#
1415     nodes[2].execute()
1416
1417     tree , klass = generate_invokeVirtual(nodes[2].tree , nodes[2].type ,
1418         'operator--' , [], [])
1419     nodes[0].tree = tree
1420     nodes[0].type = klass
1421     nodes[0].lvalue = nodes[2].lvalue
1422     #>
1423         | postfixExpr
1424 <#
1425     nodes[1].execute()
1426
1427     nodes[0].tree = nodes[1].tree
1428     nodes[0].type = nodes[1].type
1429     nodes[0].lvalue = nodes[1].lvalue
1430     #>

```

```

1431         ;
1432
1433         "Más expresiones unarias"
1434         postfixExpr ->
1435             primaryExpr postfixSuffixes
1436     <#
1437     nodes[1].execute()
1438
1439     nodes[2].tree = nodes[1].tree
1440     nodes[2].type = nodes[1].type
1441     nodes[2].lvalue = nodes[1].lvalue
1442     nodes[2].execute()
1443
1444     nodes[0].tree = nodes[2].tree
1445     nodes[0].type = nodes[2].type
1446     nodes[0].lvalue = nodes[2].lvalue
1447     #>
1448         ;
1449
1450         "Sufijos"
1451         postfixSuffixes ->
1452             DOT IDENT LPAREN argExprList RPAREN postfixSuffixes
1453     <#
1454     nodes[4].execute()
1455
1456     tree , klass = generate_invokeVirtual(nodes[0].tree , nodes[0].type ,
1457         nodes[2].text , nodes[4].args , nodes[4].types)
1458
1459     nodes[6].tree = tree
1460     nodes[6].type = klass
1461     nodes[6].lvalue = False
1462     nodes[6].execute()
1463
1464     nodes[0].tree = nodes[6].tree
1465     nodes[0].type = nodes[6].type
1466     nodes[0].lvalue = nodes[6].lvalue
1467     #>
1468         | DOT IDENT postfixSuffixes
1469     <#
1470     leftType = nodes[0].type
1471     name = nodes[2].text
1472     field = leftType.getField(name) # Para causar un error si el campo no existe
1473
1474     nodes[3].tree = ('field' , nodes[0].tree , nodes[0].type , nodes[2].text)
1475     nodes[3].type = field.getClass()
1476     nodes[3].lvalue = True
1477     nodes[3].execute()
1478
1479     nodes[0].tree = nodes[3].tree
1480     nodes[0].type = nodes[3].type
1481     nodes[0].lvalue = nodes[3].lvalue
1482     #>
1483         | INC
1484     <#
1485     if not nodes[0].lvalue:
1486         raise 'El operando del operador de postincremento debe ser un lvalue'
1487
1488     tree , klass = generate_invokeVirtual(nodes[0].tree , nodes[0].type ,
1489         'operator++' , [('int' , 0)] , [getClass('Integer')])
1490     nodes[0].tree = tree
1491     nodes[0].type = klass
1492     nodes[0].lvalue = False
1493     #>

```

```

1494         | DEC
1495 <#
1496 if not nodes[0].lvalue:
1497     raise 'El operando del operador de postincremento debe ser un lvalue'
1498
1499 tree, klass = generate_invokeVirtual(nodes[0].tree, nodes[0].type,
1500     'operator—', [( 'int', 0)], [getClass( 'Integer')])
1501 nodes[0].tree = tree
1502 nodes[0].type = klass
1503 nodes[0].lvalue = False
1504 #>
1505         |
1506         ;
1507
1508     "Argumentos de la llamada a un método"
1509     argExprList ->
1510         expressions
1511 <#
1512 nodes[1].execute()
1513 nodes[0].args = nodes[1].tree
1514 nodes[0].types = nodes[1].types
1515 #>
1516         |
1517 <#
1518 nodes[0].args = ()
1519 nodes[0].types = ()
1520 #>
1521         ;
1522
1523     "Expresión primaria"
1524     primaryExpr ->
1525         NEW IDENT LPAREN argExprList RPAREN
1526 <#
1527 nodes[4].execute()
1528 klass = getClass(nodes[2].text)
1529 ctor = klass.getConstructor(nodes[4].types)
1530 nodes[0].tree = ( 'new', klass, ctor, nodes[4].args)
1531 nodes[0].type = klass
1532 nodes[0].lvalue = False
1533 #>
1534         | NULL
1535 <#
1536 nodes[0].tree = ( 'null',)
1537 nodes[0].type = getClass( '<null>' )
1538 nodes[0].lvalue = False
1539 #>
1540         | THIS
1541 <#
1542 nodes[0].tree = ( 'this',)
1543 nodes[0].type = symbolTable.getCurrentClass()
1544 nodes[0].lvalue = False
1545 #>
1546         | TRUE
1547 <#
1548 nodes[0].tree = ( 'true',)
1549 nodes[0].type = getClass( 'Bool' )
1550 nodes[0].lvalue = False
1551 #>
1552         | SUPER DOT superAccess
1553 <#
1554 parent = symbolTable.getCurrentClass().getParent()
1555 if not parent:
1556     raise 'Esta clase no tiene clases base'

```

```

1557 nodes[3].type = parent
1558 nodes[3].execute()
1559
1560 nodes[0].tree = nodes[3].tree
1561 nodes[0].type = nodes[3].type
1562 nodes[0].lvalue = nodes[3].lvalue
1563 #>
1564         | IDENT LPAREN argExprList RPAREN
1565 <#
1566 nodes[3].execute()
1567
1568 tree , klass = generate_invokeVirtual(( 'this' ,), symbolTable.getCurrentClass() ,
1569         nodes[1].text , nodes[3].args , nodes[3].types)
1570
1571 nodes[0].tree = tree
1572 nodes[0].type = klass
1573 nodes[0].lvalue = False
1574
1575 #>
1576         | IDENT
1577 <#
1578 name = nodes[1].text
1579 nodes[0].tree = ( 'ident' , name)
1580 nodes[0].type = symbolTable.getVar(name)
1581 nodes[0].lvalue = True
1582 #>
1583         | INTEGER
1584 <#
1585 nodes[0].tree = ( 'int' , int(nodes[1].text))
1586 nodes[0].type = getClass( 'Integer' )
1587 nodes[0].lvalue = False
1588 #>
1589         | STRING
1590 <#
1591 nodes[0].tree = ( 'str' , nodes[1].text[1:-1])
1592 nodes[0].type = getClass( 'String' )
1593 nodes[0].lvalue = False
1594 #>
1595         | LPAREN optimizedExpressions RPAREN
1596 <#
1597 nodes[2].execute()
1598 nodes[0].tree = nodes[2].tree
1599 nodes[0].type = nodes[2].type
1600 nodes[0].lvalue = nodes[2].lvalue
1601 #>
1602         ;
1603
1604         "Acceso a un miembro de nuestra clase padre"
1605         superAccess ->
1606             SUPER DOT superAccess
1607 <#
1608 parent = symbolTable.getCurrentClass().getParent()
1609 if not parent:
1610     raise 'Esta clase no tiene clases base'
1611 nodes[3].type = parent
1612 nodes[3].execute()
1613
1614 nodes[0].tree = nodes[3].tree
1615 nodes[0].type = nodes[3].type
1616 nodes[0].lvalue = nodes[3].lvalue
1617 #>
1618         | IDENT LPAREN argExprList RPAREN
1619 <#

```

```

1620 nodes[3].execute()
1621
1622 method = nodes[0].type.getMethod(nodes[1].text , nodes[3].types)
1623
1624 nodes[0].tree = ( 'invokespecial' , ( 'this' , ) , method , nodes[3].args)
1625 nodes[0].type = method.getReturnType()
1626 nodes[0].lvalue = False
1627
1628 #>
1629         | IDENT
1630 <#
1631 leftType = nodes[0].type
1632 name = nodes[2].text
1633 field = leftType.getField(name) # Ya causa él un error si el campo no existe
1634
1635 nodes[0].tree = ( 'field' , nodes[0].tree , nodes[0].type , nodes[2].text)
1636 nodes[0].type = field.getClass()
1637 nodes[0].lvalue = True
1638 #>
1639         ;
1640 }
1641
1642 Skip = {
1643     " ";
1644     "\t";
1645     "\n";
1646 }
1647
1648 NotSkip = {
1649 }

```

A.2 interpreter.py

```

1  # -*- coding: ISO-8859-1 -*-
2
3  import operator , types , persistence
4  from objs import *
5
6  #-----
7
8  # La tabla de símbolos
9
10 class RuntimeSymbolTable(object):
11     def __init__(self):
12         self.blockVars = {}
13         self.blockStack = []
14         self.vars = {}
15         self.varStack = []
16         self.currentClass = None
17         self.classStack = []
18         self.currentInstance = None
19         self.instanceStack = []
20
21     def getCurrentClass(self): return self.currentClass
22     def getCurrentInstance(self): return self.currentInstance
23
24     def methodEnter(self , instance , method , args):
25         self.classStack.append(self.currentClass)
26         self.currentClass = method.getClass()
27
28         self.blockStack.append(self.blockVars)

```

```

29         self.blockVars = {}
30
31         self.varStack.append(self.vars)
32         self.vars = {}
33
34         self.instanceStack.append(self.currentInstance)
35         self.currentInstance = instance
36
37         # Metemos las variables miembro de la instancia
38         for name, ref in instance.getFields(self.currentClass).items():
39             self.setVar(name, ref)
40         # Ahora los argumentos que le pasamos al método
41         for (name, klass), ref in zip(method.getArgs(), args):
42             self.setVar(name, JRef(klass, ref.getInstance()))
43         # Y por último el this
44         self.setVar('this', JRef(self.currentClass, instance))
45
46     def methodExit(self):
47         self.vars = self.varStack.pop()
48         self.blockVars = self.blockStack.pop()
49         self.currentClass = self.classStack.pop()
50         self.currentInstance = self.instanceStack.pop()
51
52     def blockEnter(self):
53         self.blockStack.append(self.blockVars)
54         self.blockVars = {}
55         self.varStack.append(self.vars)
56         self.vars = self.vars.copy()
57
58     def blockExit(self):
59         self.blockVars = self.blockStack.pop()
60         self.vars = self.varStack.pop()
61
62     def setVar(self, name, ref):
63         self.blockVars[name] = ref
64         self.vars[name] = ref
65
66     def getVar(self, name):
67         return self.vars[name]
68
69 #
70
71 # El intérprete
72
73 class Interpreter(object):
74     def __init__(self, application, classes = {}):
75         self.application = application
76         self.symbolTable = RuntimeSymbolTable()
77         self.classes = classes
78         self.manager = persistence.Manager(self, application)
79
80     def terminate(self):
81         self.manager.terminate()
82
83     def getSymbolTable(self): return self.symbolTable
84     def getClassByName(self, name): return self.classes[name]
85     def getPersistenceManager(self): return self.manager
86     def getApplication(self): return self.application
87
88     def parse(self, l):
89         if l and operator.isSequenceType(l) \
90             and not isinstance(l, types.StringTypes):
91             f = getattr(self, 'visit_' + l[0])

```

```
92         return apply(f, l[1:])
93     else:
94         return l
95
96     def visit_main(self, block):
97         self.parse(block)
98
99     def visit_block(self, *statements):
100         for s in statements: self.parse(s)
101
102     def visit_decl(self, klass, *vars):
103         for name, init in vars:
104             ref = self.parse(init)
105             inst = ref.getInstance()
106             self.symbolTable.setVar(name, JRef(klass, inst))
107
108     def visit_return(self, exprs):
109         raise JReturn(self.parse(exprs).getInstance())
110
111     def visit_voidreturn(self):
112         raise JReturn(voidInstance)
113
114     def visit_exprs(self, *exprs):
115         e = [self.parse(expr) for expr in exprs]
116         return e[-1]
117
118     def visit_assign(self, lval, rval):
119         l = self.parse(lval)
120         r = self.parse(rval)
121         l.setInstanceFrom(r)
122         return l
123
124     def visit_is(self, lval, rval):
125         l = self.parse(lval).getInstance()
126         r = self.parse(rval).getInstance()
127         return JRef(boolClass, makeBool(l is r))
128
129     def visit_land(self, lval, rval):
130         l = self.parse(lval)
131         if l.isNull(): result = False
132         else: result = not self.parse(rval).isNull()
133         return JRef(boolClass, makeBool(result))
134
135     def visit_lor(self, lval, rval):
136         l = self.parse(lval)
137         if l.isNull(): result = not self.parse(rval).isNull()
138         else: result = True
139         return JRef(boolClass, makeBool(result))
140
141     def visit_lnot(self, val):
142         v = self.parse(val)
143         return JRef(boolClass, makeBool(v.isNull()))
144
145     def visit_new(self, klass, ctor, ctor_args):
146         inst = klass.newInstance()
147         a = [self.parse(arg) for arg in ctor_args]
148         ctor.invoke(self, inst, a)
149         r = JRef(klass, inst)
150         return r
151
152     def visit_null(self):
153         return JRef(nullClass, nullInstance)
154
```

```
155     def visit_this(self):
156         return JRef(self.symbolTable.getCurrentClass(),
157                    self.symbolTable.getCurrentInstance())
158
159     def visit_true(self):
160         return JRef(boolClass, boolInstance)
161
162     def visit_ident(self, ident):
163         r = self.symbolTable.getVar(ident)
164         return self.symbolTable.getVar(ident)
165
166     def visit_int(self, i):
167         ii = integerClass.newInstance()
168         ii.setValue(i)
169         return JRef(integerClass, ii)
170
171     def visit_str(self, s):
172         si = stringClass.newInstance()
173         si.setValue(s)
174         return JRef(stringClass, si)
175
176     def visit_invokevirtual(self, ref, mangledName, args):
177         r = self.parse(ref)
178         if r.isNull():
179             raise 'La instancia es nula'
180         inst = r.getInstance()
181         klass = inst.getClass()
182         method = klass.getMethodByMangledName(mangledName)
183         a = [self.parse(arg) for arg in args]
184         returnType = method.getReturnType()
185         result = method.invoke(self, inst, a)
186         #if not result: result = nullInstance
187         return JRef(returnType, result)
188
189     def visit_invokespecial(self, ref, method, args):
190         r = self.parse(ref)
191         if r.isNull():
192             raise 'La referencia es nula'
193         inst = r.getInstance()
194         a = [self.parse(arg) for arg in args]
195         returnType = method.getReturnType()
196         result = method.invoke(self, inst, a)
197         #if not result: result = nullInstance
198         return JRef(returnType, result)
199
200     def visit_field(self, ref, klass, name):
201         r = self.parse(ref)
202         if r.isNull(): raise 'La referencia es nula'
203         inst = r.getInstance()
204         return inst.getField(klass, name)
205
206     def visit_if(self, self, cond, ifTrue, ifFalse):
207         c = self.parse(cond)
208         if c.isNull(): return self.parse(ifFalse)
209         else: return self.parse(ifTrue)
210
211     def visit_while(self, self, cond, code):
212         while True:
213             c = self.parse(cond)
214             if c.isNull(): break
215             self.parse(code)
216
217     def visit_for(self, self, init, cond, step, code):
```

```

218         # Como en un for se pueden declarar variables locales en su
219         # sección de inicialización , tengo que hacer un blockEnter
220         self.symbolTable.blockEnter()
221         self.parse(init)
222         while True:
223             c = self.parse(cond)
224             if c.isNull(): break
225             self.parse(code)
226             self.parse(step)
227         self.symbolTable.blockExit()
228
229     def visit_reify(self, reify_code):
230         reify_code.execute()
231
232     def visit_upcast(self, klass, ref):
233         r = self.parse(ref)
234         return JRef(klass, r.getInstance())
235
236     def visit_downcast(self, klass, ref):
237         r = self.parse(ref)
238         inst = r.getInstance()
239         if inst.isA(klass):
240             return JRef(klass, inst)
241         else:
242             raise 'La instancia no es un %s' % klass

```

A.3 objs.py

```

1  # -*- coding: ISO-8859-1 -*-
2
3  import sys, operator, guidgen
4
5  class JClass(object):
6      def __init__(self, name, parent = None):
7          self.__name = name
8          self.__parent = parent
9          self.__isA = {name: self}
10         self.__fields = {}
11         self.__ownFields = {}
12         self.__methods = {}
13         self.__ownMethods = {}
14         self.__methodGroups = {}
15         self.__constructors = JMethodGroup(name)
16         if parent:
17             self.__isA.update(parent.__isA)
18             self.__fields.update(parent.__fields)
19             self.__methods.update(parent.__methods)
20             for name, group in parent.__methodGroups.items():
21                 self.__methodGroups[name] = group.clone()
22
23         def __repr__(self): return '<Java-- class %s>' % self.__name
24
25         def isNullClass(self): return False
26
27         def getName(self): return self.__name
28         def getParent(self): return self.__parent
29         def getFields(self): return self.__fields
30         def getField(self, name):
31             try:
32                 return self.__fields[name]
33             except KeyError, e:

```

```

34         raise 'La clase %s no tiene ningún campo llamado %s' \
35             %(self , name)
36
37     def updateFromParent(self):
38         parent = self.__parent
39         if parent:
40             self.__fields.update(parent.__fields)
41             self.__methods.update(parent.__methods)
42             for name, group in parent.__methodGroups.items():
43                 self.__methodGroups[name] = group.clone()
44
45     def isA(self , klass):
46         name = klass.getName()
47         return self.__isA.has_key(name)
48
49     def addFields(self , fields):
50         for f in fields: self.addField(f)
51
52     def addField(self , field):
53         name = field.getName()
54         if self.__ownFields.has_key(name):
55             raise 'Ya existe un campo llamado %s' % name
56         self.__fields[name] = field
57         self.__ownFields[name] = field
58
59     def addMethods(self , methods):
60         for m in methods: self.addMethod(m)
61
62     def addMethod(self , method):
63         method.setClass(self)
64         mangledName = method.getMangledName()
65         if self.__ownMethods.has_key(mangledName):
66             raise 'Redefinido el método "%s" de la clase %s' \
67                 %(method , self.getName())
68         try:
69             super = self.__methods[mangledName]
70             if method.getReturnType() != super.getReturnType():
71                 raise 'Si se sobreescribe un método, los' \
72                     + ' tipos de retorno deben coincidir'
73         except KeyError , e:
74             super = None
75         method.setSuperMethod(super)
76         self.__methods[mangledName] = method
77         self.__ownMethods[mangledName] = method
78         name = method.getName()
79         try:
80             group = self.__methodGroups[name]
81         except KeyError , e:
82             group = JMethodGroup(name)
83             self.__methodGroups[name] = group
84         group.addMethod(method)
85
86     def addConstructors(self , constructors):
87         for c in constructors: self.addConstructor(c)
88
89     def addConstructor(self , constructor):
90         constructor.setClass(self)
91         self.__constructors.addMethod(constructor)
92
93     # Añade el constructor por defecto si no se definió ningún constructor
94     def addDefaultConstructor(self):
95         def emptyCode(interpreter , self , instance , args): pass
96         if self.__constructors.isEmpty():

```

```

97         self.addConstructor(
98             JConstructor(self, (),
99                 emptyCode, False))
100
101     # Devuelve una lista con los métodos que se podrían lanzar si se llama
102     # al método name de la clase con los argumentos de los tipos indicados
103     # en argTypes (si la lista tiene longitud > 1 hay ambigüedad, y si es
104     # 0 es porque ninguno de los métodos es aplicable)
105     def getMethods(self, name, argTypes):
106         try:
107             return self.__methodGroups[name].match(argTypes)
108         except KeyError, e:
109             raise 'La clase %s no tiene ningún método llamado %s' \
110                 %(self, name)
111
112     # Devuelve el método que hay que llamar con los argumentos dados
113     def getMethod(self, name, argTypes):
114         methods = self.getMethods(name, argTypes)
115         numMethods = len(methods)
116         if numMethods < 1:
117             raise 'No se puede llamar al método %s con parámetros %s' \
118                 %(name, argTypes)
119         elif numMethods > 1:
120             candidates = "\n".join([str(i) for i in methods])
121             raise ("Ambigüedad en la llamada al método %s" \
122                 + " con parámetros %s\n" \
123                 + "Candidatos:\n%") \
124                 %(name, argTypes, candidates)
125         else:
126             return methods[0]
127
128     def getMethodByMangledName(self, mangledName):
129         try:
130             return self.__methods[mangledName]
131         except KeyError, e:
132             raise 'La clase %s no tiene ningún método' \
133                 + ' con mangled name %s' \
134                 %(self, mangledName)
135
136     def getConstructors(self, argTypes):
137         return self.__constructors.match(argTypes)
138
139     def getConstructor(self, argTypes):
140         constructors = self.getConstructors(argTypes)
141         numConstructors = len(constructors)
142         if numConstructors < 1:
143             raise ('No hay ningún constructor de %s que admita' \
144                 + ' los argumentos de tipos %s') \
145                 %(self.getName(), argTypes)
146         elif numConstructors > 1:
147             candidates = "\n".join([str(i) for i in constructors])
148             raise "Ambigüedad en la llamada al constructor de %s" \
149                 + " con parámetros %s\n" \
150                 + "Candidatos:\n%" \
151                 %(self.getName(), argTypes, candidates)
152         else:
153             return constructors[0]
154
155     def getDefaultConstructor(self):
156         return self.getConstructor(())
157
158     # Devuelve la distancia entre esta clase y una supuesta clase base
159     # suya, o -1 si ancestro no es una clase base de ésta

```

```

160     def distance(self , ancestor):
161         def calcDistance(fromClass , ancestor):
162             if ancestor.__name == fromClass.__name: return 0
163             else: return 1 + calcDistance(fromClass.__parent , ancestor)
164         if self.isNullClass(): return 0
165         elif not self.isA(ancestor): return -1
166         else: return calcDistance(self , ancestor)
167
168     #def invokeByMangledName(self , interpreter , mangledName , instance , args):
169     #    method = self.getMethodByMangledName(mangledName)
170     #    return m.invoke(interpreter , instance , args)
171
172     def newInstance(self , guid = None):
173         inst = JInstance(self , guid)
174         return inst
175
176     def createFields(self , owner):
177         return self.__createFields({}, owner)
178
179     def __createFields(self , fields , owner):
180         parent = self.__parent
181         if parent:
182             fields = parent.__createFields(fields , owner)
183             parentFields = fields[parent]
184             myFields = parentFields.copy()
185         else:
186             myFields = {}
187
188         # El siguiente bucle puede provocar que el programa entre
189         # en un bucle infinito y rompa (caso de tener p.ej. una
190         # clase A con un miembro de tipo B y una clase B con un
191         # miembro de tipo A: habría que comprobar previamente que
192         # no se dan casos de dependencia mutua como ése)
193         for name, f in self.__ownFields.items():
194             klass = f.getClass()
195             inst = f.getInit().getInstance()
196             myFields[name] = JRef(klass , inst , owner)
197
198         fields[self] = myFields
199         return fields
200
201 #-----
202
203 # Clase que representa al tipo nulo, es un caso especial por tener algunos
204 # comportamientos diferentes a las de una clase típica (concretamente el isA)
205 class JNullClass(JClass):
206     def __init__(self): JClass.__init__(self , '<null>')
207     def isA(self , klass): return True
208     def isNullClass(self): return True
209
210 #-----
211
212 # Grupo de métodos con el mismo nombre pero diferentes parámetros
213
214 class JMethodGroup(object):
215     def __init__(self , name):
216         self.__name = name
217         self.__methods = {}
218
219     def __repr__(self):
220         return '<Java-- method group %s: %s>' \
221             %(self.__name, self.__methods)
222

```

```

223     def addMethod(self , method):
224         self.__methods[method.getMangledName()] = method
225
226     def getMethodByMangledName(self , name): return self.__methods[name]
227
228     def match(self , argTypes):
229         # Calculamos la distancia de cada método
230         pairs = [(x.distance(argTypes) , x)
231                 for x in self.__methods.values()]
232         # Quitamos las distancias menores que 0 (métodos no aplicables)
233         pairs = [x for x in pairs if x[0] >= 0]
234         # Si no nos quedan métodos podemos cortar ya
235         if not pairs: return []
236         # Calculamos la distancia mínima
237         distances = [x[0] for x in pairs]
238         minDist = reduce(min , distances)
239         # Descartamos los métodos que no estén a la distancia mínima y
240         # los que nos queden son los que podrían servir
241         return [x[1] for x in pairs if x[0] == minDist]
242
243     def isEmpty(self): return not self.__methods
244
245     def clone(self):
246         theClone = JMethodGroup(self.__name)
247         theClone.__methods.update(self.__methods)
248         return theClone
249
250 #-----
251
252 class JMethod(object):
253     def __init__(self , name , returnType , args , code = None):
254         self.__name = name
255         self.__returnType = returnType
256         # args es una secuencia de pares (tipo , nombre),
257         self.__args = args
258         # Cogemos sólo los tipos
259         self.__argTypes = [x[1] for x in args]
260         # Ahora nos quedamos con los nombres de los tipos
261         self.__argTypeNames = [x.getName() for x in self.__argTypes]
262         # Y por último guardamos los nombres de los argumentos
263         self.__argNames = [x[0] for x in args]
264         self.__mangledName = JMethod.mangle(name , self.__argTypeNames)
265         self.__code = code
266
267     def mangle(name , typeNames): return '%@%' % (name , '&'.join(typeNames))
268     mangle = staticmethod(mangle)
269
270     def __repr__(self):
271         return '<Java--- method "%s %s.%s(%s)">' % (
272             self.__returnType.getName() , self.__klass.getName() ,
273             self.__name , ', '.join(self.__argTypeNames))
274
275     def getName(self): return self.__name
276     def getMangledName(self): return self.__mangledName
277     def getClass(self): return self.__klass
278     def setClass(self , klass): self.__klass = klass
279     def getReturnType(self): return self.__returnType
280     def getArgs(self): return self.__args
281     def getSuperMethod(self): return self.__superMethod
282     def setSuperMethod(self , super): self.__superMethod = super
283     def getArgTypes(self): return self.__argTypes
284     def getArgTypeNames(self): return self.__argTypeNames
285

```

```

286     # La distancia del método a los argumentos será el máximo de las
287     # distancias de cada argumento a los de la definición del método
288     def distance(self, args):
289         # Si la longitud de la lista de argumentos que nos pasan
290         # no coincide con la que tenemos que recibir, no valemos
291         if len(args) != len(self.__argTypes): return -1
292         # Si coinciden y args == [], entonces distancia cero
293         if not args: return 0
294         # Calculamos la distancia de cada argumento a los que
295         # realmente nos pasan...
296         distances = [x.distance(y)
297                     for x, y in zip(args, self.__argTypes)]
298         # ... luego calculamos el máximo y el mínimo...
299         maxDist = reduce(max, distances)
300         minDist = reduce(min, distances)
301         # ... y retornamos el máximo EXCEPTO si el mínimo es < 0
302         # (que sólo sucederá si algún parámetro no es convertible)
303         if minDist < 0: return -1
304         else: return maxDist
305
306     def setCode(self, code): self.__code = code
307
308     def invoke(self, interpreter, instance, args):
309         return self.__code(interpreter, self, instance, args)
310
311 #-----
312
313 # Para la ejecución de los return lanzamos una excepción de este tipo, y así
314 # nos ahorramos tener que hacer comprobaciones constantemente
315 class JReturn:
316     def __init__(self, value):
317         self.__value = value
318
319     def getValue(self):
320         return self.__value
321
322 # Envoltorio para poder llamar a código del usuario como si fuese una función
323 # primitiva
324 class UserCodeWrapper(object):
325     def __init__(self, code):
326         self.__code = code
327
328     def __call__(self, interpreter, method, instance, args):
329         symbolTable = interpreter.getSymbolTable()
330         symbolTable.methodEnter(instance, method, args)
331         try:
332             result = interpreter.parse(self.__code)
333             if result:
334                 result = result.getInstance()
335             else:
336                 result = voidInstance
337             #if self.getReturnType() != voidClass:
338             #    # Deberíamos alertar de que se terminó
339             #    # el método sin encontrar un return
340         except JReturn, r:
341             result = r.getValue()
342         symbolTable.methodExit()
343         return result
344
345 #-----
346
347 class JConstructor(JMethod):
348     def __init__(self, klass, args, code = None,

```

```

349         hasSuperStatement = False):
350         name = klass.getName()
351         JMethod.__init__(self, klass.getName(), voidInstance,
352             args, code)
353         self.__hasSuperStatement = hasSuperStatement
354
355     def __repr__(self):
356         return '<Java-- constructor "%s.%s(%s)">' % (
357             self.getClass().getName(), self.getName(),
358             ', '.join(self.getArgTypeNames()))
359
360     def setHasSuperStatement(self, hasSuperStatement):
361         self.__hasSuperStatement = hasSuperStatement
362
363     def invoke(self, interpreter, instance, args):
364         parent = self.getClass().getParent()
365         if parent and not self.__hasSuperStatement:
366             parentCtor = parent.getDefaultConstructor()
367             parentCtor.invoke(interpreter, instance, args)
368         JMethod.invoke(self, interpreter, instance, args)
369
370 # -----
371
372 class JField(object):
373     def __init__(self, name, klass, init):
374         self.__name = name
375         self.__klass = klass
376         self.__init = init
377
378     def getName(self): return self.__name
379     def getClass(self): return self.__klass
380     def getInit(self): return self.__init
381
382 # -----
383
384 class JInstance(object):
385     def __init__(self, klass, guid = None):
386         self.klass = klass
387         self.value = None
388
389         # Para la persistencia, cada instancia tendrá un GUID
390         if guid is None:
391             self.guid = guidgen.guid()
392         else:
393             self.guid = str(guid)
394
395         # No nos estamos guardando (para la recursividad en store())
396         self.storing = False
397
398         # Al crearnos estamos modificándonos
399         self.modified = True
400
401         # Por defecto somos almacenables y no persistentes
402         self.nonStorable = False
403         self.persistent = False
404
405         self.fields = klass.createFields(self)
406
407     def getID(self): return self.guid
408
409     def isNonStorable(self): return self.nonStorable
410     def setNonStorable(self): self.nonStorable = True
411

```

```

412     def __getstate__(self):
413         dict = self.__dict__.copy()
414         del dict['storing']
415         del dict['modified']
416         dict['klass'] = self.klass.getName()
417         fields = {}
418         for (klass, fieldDict) in self.fields.items():
419             fields[klass.getName()] = fieldDict
420         dict['fields'] = fields
421         if dict.has_key('manager'): del dict['manager']
422         return dict
423
424     def __setstate__(self, dict):
425         self.__dict__.update(dict)
426
427         self.storing = False
428         self.modified = False
429         self.not_ready = True
430
431     def isA(self, klass): return self.klass.isA(klass)
432     def getClass(self): return self.klass
433     def getField(self, klass, field): return self.fields[klass][field]
434     def setField(self, klass, field, ref):
435         self.fields[klass][field].setInstanceFrom(ref)
436     def getFields(self, klass): return self.fields[klass]
437     def getValue(self): return self.value
438     def setValue(self, value):
439         self.value = value
440         self.setModified()
441     def isNull(self): return self.klass.isNullClass()
442     def isModified(self): return self.modified
443     def setModified(self, modified = True):
444         self.modified = modified
445
446         # Si somos persistentes, debemos avisar al Persistence Manager
447         # de que nos hemos modificado
448         if self.isPersistent():
449             self.manager.notifyModified(self, modified)
450
451     def isPersistent(self): return self.persistent
452     def makePersistent(self, manager):
453         if not (self.isNonStorable() or self.isPersistent()):
454             self.manager = manager
455             refs = self.getRefs()
456             for ref in refs.values():
457                 inst = ref.getInstance()
458                 inst.makePersistent(manager)
459     def makeTransient(self):
460         if not self.isPersistent(): return
461         self.persistent = False
462         refs = self.getRefs()
463         for ref in refs.values():
464             inst = ref.getInstance()
465             inst.makeTransient()
466
467     def store(self, storage):
468         # Para cortar el proceso recursivo
469         if self.storing: return
470         self.storing = True
471         # Primero nuestros miembros
472         for (klass, fieldDict) in self.fields.items():
473             for ref in fieldDict.values():
474                 inst = ref.getInstance()

```

```

475         inst.store(storage)
476     # Si nos tenemos que guardar, nos guardamos
477     if not self.isNonStorable() and ( self.modified \
478         or not storage.has_key(self.getID()):
479         storage.storeObject(self)
480         self.modified = False
481     self.storing = False
482     return self.getID()
483
484     def getRefs(self):
485         refs = {}
486         for fieldDict in self.fields.values():
487             # Un simple update del diccionario no nos vale,
488             # porque queremos todas las referencias a miembros,
489             # y en alguna clase se pudo ocultar alguno de los de
490             # su clase base dándole el mismo nombre...
491             #refs.update(fieldDict)
492
493             for ref in fieldDict.values():
494                 refs[ref] = ref
495         return refs
496
497     def restore(self, interpreter, storage):
498         if hasattr(self, 'not_ready'):
499             # Restauramos la clase
500             self.klass = interpreter.getClassByName(self.klass)
501
502             # Restauramos los campos
503             fields = {}
504             for (klass, fieldDict) in self.fields.items():
505                 klass = interpreter.getClassByName(klass)
506                 fields[klass] = fieldDict
507             self.fields = fields
508
509             # Restauramos las referencias
510             refs = self.getRefs()
511             for ref in refs.values():
512                 ref.restore(interpreter, storage)
513
514             del self.not_ready
515
516 #
517
518     class JRef(object):
519         def __init__(self, klass, instance, owner = None):
520             self.klass = klass
521             self.owner = owner
522             self.setInstance(instance)
523
524         def getClass(self): return self.klass
525         def getInstance(self): return self.instance
526         def isNull(self): return self.instance.isNull()
527
528         def setOwner(self, owner):
529             self.owner = owner
530
531         def setInstance(self, instance):
532             self.instance = instance
533             if self.owner:
534                 self.owner.setModified(self)
535
536         def setInstanceFrom(self, ref):
537             self.setInstance(ref.getInstance())

```

```

538
539     def __getstate__(self):
540         dict = self.__dict__.copy()
541         dict[ 'klass' ] = self.klass.getName()
542         dict[ 'owner' ] = self.owner.getID()
543         dict[ 'instance' ] = self.instance.getID()
544         return dict
545
546     def __setstate__(self, dict):
547         self.__dict__.update(dict)
548         self.not_ready = True
549
550     def restore(self, interpreter, storage):
551         if hasattr(self, 'not_ready'):
552             self.klass = interpreter.getClassByName(self.klass)
553             self.owner = storage.retrieveObject(self.owner)
554             self.instance = storage.retrieveObject(self.instance)
555         del self.not_ready
556
557 #-----
558
559 nullClass = JNullClass()
560 nullInstance = nullClass.newInstance(guid = 0) # Similar al None de Python...
561 # Null y similares siempre serán las mismas para todo programa
562 nullInstance.setNonStorable()
563
564 voidClass = JClass( '<void>' )
565 voidInstance = voidClass.newInstance(guid = 1)
566 voidInstance.setNonStorable()
567
568 classClass = JClass( '<class>' )
569
570 boolClass = JClass( 'Bool' )
571 boolInstance = boolClass.newInstance(guid = 2) # Similar al T de Lisp
572 boolInstance.setNonStorable()
573
574 objectClass = JClass( 'Object' )
575 objectClass.addDefaultConstructor()
576
577 integerClass = JClass( 'Integer', objectClass )
578 stringClass = JClass( 'String', objectClass )
579 consoleClass = JClass( 'Console', objectClass )
580 arrayClass = JClass( 'Array', objectClass )
581 dictClass = JClass( 'Dictionary', objectClass )
582 managerClass = JClass( 'PersistenceManager', objectClass )
583
584 primitiveClasses = [
585     nullClass,
586     voidClass,
587     classClass,
588     boolClass,
589     objectClass,
590     integerClass,
591     stringClass,
592     consoleClass,
593     arrayClass,
594     dictClass,
595     managerClass
596 ]
597 nonStorableInstances = {
598     nullInstance.getID(): nullInstance,
599     voidInstance.getID(): voidInstance,
600     boolInstance.getID(): boolInstance

```

```

601 }
602
603 #-----
604
605 # Utilidad para crear booleanos rápidamente
606
607 def makeBool(test):
608     if test: bb = boolInstance
609     else: bb = nullInstance
610     return bb
611
612 # Para tipos simples como cadenas y enteros
613
614 def makeSimple(klass, value):
615     inst = klass.newInstance()
616     inst.setValue(value)
617     return inst
618
619 #-----
620
621 # Funciones auxiliares para crear operadores
622
623 def makeBinaryAOp(klass, op): # Operadores aritméticos binarios
624     def f(interpreter, method, instance, args):
625         arg = args[0].getInstance()
626         if arg.isNull(): raise 'El argumento del operador es nulo'
627         result = klass.newInstance()
628         result.setValue(op(instance.getValue(), arg.getValue()))
629         return result
630     return f
631
632 def makeBinaryLOp(op): # Operadores lógicos binarios
633     def f(interpreter, method, instance, args):
634         arg = args[0].getInstance()
635         if arg.isNull(): raise 'El argumento del operador es nulo'
636         return makeBool(op(instance.getValue(), arg.getValue()))
637     return f
638
639 #-----
640
641 # Métodos de la clase Object
642
643 objectClass.updateFromParent()
644
645 def objectMethod_id(interpreter, method, instance, args):
646     return makeSimple(stringClass, instance.getID())
647 def objectMethod_makePersistent(interpreter, method, instance, args):
648     manager = interpreter.getPersistenceManager()
649     manager.makePersistent(instance)
650     return voidInstance
651 def objectMethod_makeTransient(interpreter, method, instance, args):
652     manager = interpreter.getPersistenceManager()
653     manager.makeTransient(instance)
654     return voidInstance
655
656 objectClass.addMethods([
657     JMethod('id', stringClass, (), objectMethod_id),
658     JMethod('makePersistent', voidClass, (),
659             objectMethod_makePersistent),
660     JMethod('makeTransient', voidClass, (),
661             objectMethod_makeTransient)
662 ])
663

```

```

664 #-----
665
666 # Métodos de la clase Bool
667
668 boolClass.updateFromParent()
669
670 # Constructores
671 #def boolCtor_default(interpreter, ctor, instance, args):
672 #     raise 'No se permite construir instancias de la clase Bool'
673
674 #boolClass.addConstructors([
675 #     JConstructor(boolClass, (),
676 #                 boolCtor_default, False)
677 # ])
678
679 #-----
680
681 # Métodos de la clase Integer
682
683 integerClass.updateFromParent()
684
685 # Operadores aritméticos
686 integerMethod_sum = makeBinaryAOp(integerClass, operator.add)
687 integerMethod_sub = makeBinaryAOp(integerClass, operator.sub)
688 integerMethod_mul = makeBinaryAOp(integerClass, operator.mul)
689 integerMethod_div = makeBinaryAOp(integerClass, operator.div)
690 integerMethod_mod = makeBinaryAOp(integerClass, operator.mod)
691 def integerMethod_preinc(interpreter, method, instance, args):
692     instance.setValue(instance.getValue() + 1)
693     return instance
694 def integerMethod_postinc(interpreter, method, instance, args):
695     value = instance.getValue()
696     instance.setValue(value + 1)
697     return makeSimple(integerClass, value)
698 def integerMethod_predec(interpreter, method, instance, args):
699     instance.setValue(instance.getValue() - 1)
700     return instance
701 def integerMethod_postdec(interpreter, method, instance, args):
702     value = instance.getValue()
703     instance.setValue(value - 1)
704     return makeSimple(integerClass, value)
705
706 # Operadores de comparación
707 integerMethod_lt = makeBinaryLOp(operator.lt)
708 integerMethod_le = makeBinaryLOp(operator.le)
709 integerMethod_gt = makeBinaryLOp(operator.gt)
710 integerMethod_ge = makeBinaryLOp(operator.ge)
711 integerMethod_eq = makeBinaryLOp(operator.eq)
712 integerMethod_ne = makeBinaryLOp(operator.ne)
713
714 # Otros métodos
715 def integerMethod_set(interpreter, method, instance, args):
716     arg = args[0].getInstance()
717     if arg.isNull():
718         raise 'El parámetro de set() no puede ser nulo'
719     instance.setValue(arg.getValue())
720     return voidInstance()
721 def integerMethod_toString(interpreter, method, instance, args):
722     return makeSimple(stringClass, str(instance.getValue()))
723
724 integerClass.addMethods([
725     JMethod('operator+', integerClass,
726           (('x', integerClass),), integerMethod_sum),

```

```

727         JMethod( 'operator-' , integerClass ,
728             (( 'x' , integerClass ) , ) , integerMethod_sub ) ,
729         JMethod( 'operator*' , integerClass ,
730             (( 'x' , integerClass ) , ) , integerMethod_mul ) ,
731         JMethod( 'operator/' , integerClass ,
732             (( 'x' , integerClass ) , ) , integerMethod_div ) ,
733         JMethod( 'operator%' , integerClass ,
734             (( 'x' , integerClass ) , ) , integerMethod_mod ) ,
735         JMethod( 'operator++' , integerClass , () ,
736             integerMethod_preinc ) ,
737         JMethod( 'operator++' , integerClass ,
738             (( 'x' , integerClass ) , ) , integerMethod_postinc ) ,
739         JMethod( 'operator--' , integerClass , () ,
740             integerMethod_predec ) ,
741         JMethod( 'operator--' , integerClass ,
742             (( 'x' , integerClass ) , ) , integerMethod_postdec ) ,
743         JMethod( 'operator<' , boolClass ,
744             (( 'x' , integerClass ) , ) , integerMethod_lt ) ,
745         JMethod( 'operator<=' , boolClass ,
746             (( 'x' , integerClass ) , ) , integerMethod_le ) ,
747         JMethod( 'operator>' , boolClass ,
748             (( 'x' , integerClass ) , ) , integerMethod_gt ) ,
749         JMethod( 'operator>=' , boolClass ,
750             (( 'x' , integerClass ) , ) , integerMethod_ge ) ,
751         JMethod( 'operator==' , boolClass ,
752             (( 'x' , integerClass ) , ) , integerMethod_eq ) ,
753         JMethod( 'operator!=' , boolClass ,
754             (( 'x' , integerClass ) , ) , integerMethod_ne ) ,
755         JMethod( 'set' , voidClass ,
756             (( 'x' , integerClass ) , ) , integerMethod_set ) ,
757         JMethod( 'toString' , stringClass , () ,
758             integerMethod_toString )
759     )
760
761 # Constructores
762
763 def integerCtor_default( interpreter , ctor , instance , args ):
764     instance.setValue( 0 )
765     return voidInstance
766
767 def integerCtor_int( interpreter , ctor , instance , args ):
768     arg = args[ 0 ].getInstance()
769     if arg.isNull(): raise 'El argumento del constructor es nulo'
770     instance.setValue( arg.getValue() )
771     return voidInstance
772
773 integerClass.addConstructors( [
774     JConstructor( integerClass , () ,
775                 integerCtor_default , False ) ,
776     JConstructor( integerClass ,
777                 (( 'x' , integerClass ) , ) ,
778                 integerCtor_int , False )
779 ] )
780
781 #
782
783 # Métodos de la clase String
784
785 stringClass.updateFromParent()
786
787 # Operador de concatenación
788 stringMethod_sum = makeBinaryAOp( stringClass , operator.add )
789

```

```

790 # Otros métodos
791 def stringMethod_set(interpreter , method , instance , args):
792     arg = args[0].getInstance()
793     if arg.isNull():
794         raise 'El parámetro de set() no puede ser nulo'
795     instance.setValue(arg.getValue())
796     return voidInstance()
797
798 stringClass.addMethods([
799     JMethod('operator+', stringClass ,
800         (('x', stringClass),), stringMethod_sum),
801     JMethod('set', voidClass ,
802         (('x', stringClass),), stringMethod_set)
803 ])
804
805 # Constructores
806
807 def stringCtor_default(interpreter , ctor , instance , args):
808     instance.setValue('')
809     return voidInstance()
810
811 def stringCtor_str(interpreter , ctor , instance , args):
812     arg = args[0].getInstance()
813     if arg.isNull(): raise 'El argumento del constructor es nulo'
814     instance.setValue(arg.getValue())
815     return voidInstance()
816
817 stringClass.addConstructors([
818     JConstructor(stringClass , (),
819         stringCtor_default , False),
820     JConstructor(stringClass ,
821         (('x', stringClass),),
822         stringCtor_str , False)
823 ])
824
825 #-----
826
827 # Métodos de la clase Console
828
829 consoleClass.updateFromParent()
830
831 def consoleMethod_print(interpreter , method , instance , args):
832     arg = args[0].getInstance()
833     if arg.isNull():
834         s = '<null>'
835     else:
836         s = arg.getValue()
837     interpreter.write(s)
838     return voidInstance()
839
840 def consoleMethod_println(interpreter , method , instance , args):
841     arg = args[0].getInstance()
842     if arg.isNull():
843         s = '<null>'
844     else:
845         s = arg.getValue()
846     interpreter.write(s + "\n")
847     return voidInstance()
848
849 consoleClass.addMethods([
850     JMethod('print', voidClass , (('x', stringClass),),
851         consoleMethod_print),
852     JMethod('println', voidClass , (('x', stringClass),),

```

```

853         consoleMethod_println)
854     ])
855 consoleClass.addDefaultConstructor()
856
857 #-----
858
859 # Métodos de la clase Array
860
861 arrayClass.updateFromParent()
862
863 def arrayMethod_getSize(interpreter, method, instance, args):
864     return makeSimple(integerClass, instance.getValue()[1])
865
866 def arrayMethod_setSize(interpreter, method, instance, args):
867     arg = args[0].getInstance()
868     if arg.isNull():
869         raise 'El argumento de setSize() no puede ser nulo'
870     size = arg.getValue()
871     if size < 0:
872         raise 'El argumento de setSize() debe ser >= 0'
873     list = [JRef(objectClass, nullInstance) for i in range(size)]
874     oldlist, oldsize = instance.getValue()
875     minsize = min(size, oldsize)
876     list[0:minsize] = oldlist[0:minsize]
877     instance.setValue((list, size))
878     return voidInstance
879
880 def arrayMethod_getItem(interpreter, method, instance, args):
881     pos = args[0].getInstance()
882     if pos.isNull():
883         raise 'La posición para getItem() no puede ser nula'
884     pos = pos.getValue()
885     list, size = instance.getValue()
886     if (pos < 0) or (pos >= size):
887         raise ('Índice fuera de rango en getItem() ' \
888              + '{%d, válido rango [0..%d]}' \
889              % (pos, size - 1))
890     return list[pos].getInstance()
891
892 def arrayMethod_setItem(interpreter, method, instance, args):
893     pos = args[0].getInstance()
894     if pos.isNull():
895         raise 'La posición para setItem() no puede ser nula'
896     pos = pos.getValue()
897     list, size = instance.getValue()
898     if (pos < 0) or (pos >= size):
899         raise ('Índice fuera de rango en setItem() ' \
900              + '{%d, válido rango [0..%d]}' \
901              % (pos, size - 1))
902     item = args[1].getInstance()
903     list[pos].setInstance(item)
904     if instance.isPersistent():
905         self.manager.makePersistent(item)
906     instance.setModified()
907     return voidInstance
908
909 arrayClass.addMethods([
910     JMethod('getSize', integerClass, (),
911            arrayMethod_getSize),
912     JMethod('setSize', voidClass, (('size', integerClass),),
913            arrayMethod_setSize),
914     JMethod('getItem', objectClass, (('index', integerClass),),
915            arrayMethod_getItem),

```

```

916         JMethod( 'setItem', voidClass,
917                 (( 'index', integerClass), ( 'item', objectClass).),
918                 arrayMethod_setItem),
919     ])
920
921 # Constructores
922
923 def arrayCtor_default(interpreter, ctor, instance, args):
924     #size = 1
925     #list = [nullInstance]
926     size = 0
927     list = []
928     instance.setValue((list, size))
929     return voidInstance
930
931 def arrayCtor_int(interpreter, ctor, instance, args):
932     arg = args[0].getInstance()
933     if arg.isNull(): raise 'El argumento del constructor es nulo'
934     size = arg.getValue()
935     list = [JRef(objectClass, nullInstance) for i in range(size)]
936     instance.setValue((list, size))
937     return voidInstance
938
939 arrayClass.addConstructors([
940     JConstructor(arrayClass, (),
941                 arrayCtor_default, False),
942     JConstructor(arrayClass,
943                 (( 'size', integerClass).),
944                 arrayCtor_int, False)
945 ])
946
947 #-----
948
949 # Métodos de la clase Dictionary
950
951 dictClass.updateFromParent()
952
953 def dictMethod_getItem(interpreter, method, instance, args):
954     key = args[0].getInstance()
955     if key.isNull():
956         raise 'La clave en getItem() no puede ser nula'
957     dict = instance.getValue()
958     try:
959         return dict[key.getValue()].getInstance()
960     except:
961         return nullInstance # ¿sería mejor lanzar una excepción?
962
963 def dictMethod_setItem(interpreter, method, instance, args):
964     key = args[0].getInstance()
965     if key.isNull():
966         raise 'La clave en setItem() no puede ser nula'
967     value = args[1].getInstance()
968     dict = instance.getValue()
969     dict[key.getValue()] = JRef(objectClass, value)
970     if instance.isPersistent():
971         self.manager.makePersistent(value)
972     instance.setModified()
973     return voidInstance
974
975 def dictMethod_delItem(interpreter, method, instance, args):
976     key = args[0].getInstance()
977     if key.isNull():
978         raise 'La clave en delItem() no puede ser nula'

```

```

979         dict = instance.getValue()
980     try:
981         del dict[key.getValue()]
982     except:
983         pass # ¿sería mejor lanzar una excepción?
984     instance.setModified()
985     return nullInstance
986
987 def dictMethod_hasItem(interpreter, method, instance, args):
988     key = args[0].getInstance()
989     if key.isNull():
990         raise 'La clave en hasItem() no puede ser nula'
991     dict = instance.getValue()
992     return makeBool(key.getValue() in dict)
993
994 def dictMethod_clear(interpreter, method, instance, args):
995     instance.getValue().clear()
996     instance.setModified()
997     return voidInstance
998
999 dictClass.addMethods([
1000     JMethod('getItem', objectClass, (('key', stringClass),),
1001           dictMethod_getItem),
1002     JMethod('setItem', voidClass,
1003           (('key', stringClass), ('item', objectClass)),
1004           dictMethod_setItem),
1005     JMethod('hasItem', boolClass, (('key', stringClass),),
1006           dictMethod_hasItem),
1007     JMethod('delItem', voidClass, (('key', stringClass),),
1008           dictMethod_delItem),
1009     JMethod('clear', voidClass, (),
1010           dictMethod_clear)
1011 ])
1012
1013 # Constructores
1014
1015 def dictCtor_default(interpreter, ctor, instance, args):
1016     instance.setValue({})
1017     return voidInstance
1018
1019 dictClass.addConstructors([
1020     JConstructor(arrayClass, (),
1021                 dictCtor_default, False)
1022 ])
1023
1024 # -----
1025
1026 # Métodos de la clase PersistenceManager
1027
1028 managerClass.updateFromParent()
1029
1030 def managerMethod_storeObject(interpreter, method, instance, args):
1031     arg = args[0].getInstance()
1032     if arg.isNull():
1033         raise 'El argumento de storeObject no puede ser nulo'
1034     manager = instance.getValue()
1035     manager.storeObject(arg)
1036     return makeSimple(stringClass, instance.getID())
1037
1038 def managerMethod_retrieveObject(interpreter, method, instance, args):
1039     arg = args[0].getInstance()
1040     if arg.isNull():
1041         raise 'El argumento de retrieveObject no puede ser nulo'

```

```

1042         id = arg.getValue()
1043         manager = instance.getValue()
1044         try:
1045             inst = manager.retrieveObject(id)
1046             return inst
1047         except KeyError, e:
1048             return nullInstance # ¿Sería mejor lanzar una excepción?
1049
1050     def managerMethod_setStorage(interpreter, method, instance, args):
1051         arg = args[0].getInstance()
1052         if arg.isNull():
1053             raise 'El argumento de setStorage no puede ser nulo'
1054         manager = instance.getValue()
1055         name = arg.getValue()
1056         manager.setStorage(name)
1057         return voidInstance
1058
1059     def managerMethod_setPolicy(interpreter, method, instance, args):
1060         arg = args[0].getInstance()
1061         if arg.isNull():
1062             raise 'El argumento de setStorage no puede ser nulo'
1063         manager = instance.getValue()
1064         name = arg.getValue()
1065         manager.setPolicy(name)
1066         return voidInstance
1067
1068     managerClass.addMethods([
1069         JMethod('storeObject', stringClass, (('obj', objectClass),),
1070              managerMethod_storeObject),
1071         JMethod('retrieveObject', objectClass, (('id', stringClass),),
1072              managerMethod_retrieveObject),
1073         JMethod('setStorage', objectClass, (('name', stringClass),),
1074              managerMethod_retrieveObject),
1075         JMethod('setPolicy', objectClass, (('name', stringClass),),
1076              managerMethod_retrieveObject)
1077     ])
1078
1079     # Constructores
1080
1081     def managerCtor_default(interpreter, ctor, instance, args):
1082         instance.setValue(interpreter.getPersistenceManager())
1083         return voidInstance
1084
1085     managerClass.addConstructors([
1086         JConstructor(managerClass, (),
1087              managerCtor_default, False)
1088     ])

```

A.4 persistence.py

```

1  # -*- coding: ISO-8859-1 -*-
2
3  import pickle, weakref, objs
4  import anydbm
5  import bsddb
6  import threading
7
8  #=====
9
10 # La tabla de instancias
11

```

```

12 class InstanceTable:
13     def __init__(self):
14         self.instances = weakref.WeakValueDictionary()
15         # Al añadirlas, la búsqueda de instancias únicas siempre
16         # tendrá éxito, y como tienen otras referencias a ellas
17         # nunca desaparecerán del diccionario
18         self.instances.update(objs.nonStorableInstances)
19
20     def addInstance(self, inst):
21         self.instances[inst.getID()] = inst
22
23     def getInstance(self, id):
24         return self.instances[id]
25
26     def deleteInstance(self, inst):
27         del self.instances[inst.getID()]
28
29 =====
30
31 class Manager(object):
32     def __init__(self, interpreter, application):
33         self.interpreter = interpreter
34         self.application = application
35         self.instanceTable = InstanceTable()
36         self.storages = {}
37         self.policies = {}
38
39         # El almacenamiento por defecto
40         defaultStorage = SimpleStorage(interpreter, application.name)
41         defaultStorageName = 'default'
42         self.registerStorage(defaultStorageName, defaultStorage)
43         self.storage = defaultStorage
44
45         # La política por defecto
46         defaultPolicy = SimplePolicy()
47         defaultPolicyName = 'default'
48         self.registerPolicy(defaultPolicyName, defaultPolicy)
49         self.policy = defaultPolicy
50
51     def getInstanceTable(self): return self.instanceTable
52
53     def registerStorage(self, name, storage):
54         if self.storages.has_key(name):
55             raise ('Ya hay registrado un almacenamiento' \
56                  + ' llamado %s') % name
57         self.storages[name] = storage
58         storage.setManager(self)
59     def unregisterStorage(self, name):
60         storage = getStorage(name)
61         if self.storage is storage:
62             raise 'El almacenamiento %s está en uso' % name
63         del self.storages[name]
64         return storage
65     def getStorage(self, name):
66         try:
67             return self.storages[name]
68         except KeyError, e:
69             raise 'El almacenamiento %s no está definido' % name
70     def setStorage(self, name):
71         storage = self.getStorage(name)
72         self.storage.commit()
73         self.storage = storage
74

```

```

75     def registerPolicy(self , name, policy):
76         if self.policies.has_key(name):
77             raise ( 'Ya hay registrada una política ' \
78                   + ' llamada %s' ) % name
79         self.policies[name] = policy
80         policy.setManager(self)
81     def unregisterPolicy(self , name):
82         policy = self.getPolicy(name)
83         if self.policy is policy:
84             raise 'La política %s está en uso' % name
85         del self.policies[name]
86         return policy
87     def getPolicy(self , name):
88         try:
89             return self.policies[name]
90         except KeyError , e:
91             raise 'La política %s no está definida' % name
92     def setPolicy(self , name):
93         policy = self.getPolicy(name)
94         self.policy.commit()
95         self.policy = policy
96
97     def retrieveObject(self , id):
98         return self.storage.retrieveObject(id)
99
100    def storeObject(self , instance):
101        # Le pedimos al objeto que se guarde, así si sabe que no está
102        # modificado podrá abortar la operación y ahorrarnos tiempo
103        return instance.store(self.storage)
104
105    def makePersistent(self , instance):
106        instance.makePersistent(self)
107        self.storeObject(instance)
108
109    def makeTransient(self , instance):
110        instance.makeTransient()
111
112    def commit(self):
113        self.policy.commit()
114
115    def commitStorage(self):
116        self.storage.commit()
117
118    def terminate(self):
119        self.commit()
120        for s in self.storages.values(): s.terminate()
121        for p in self.policies.values(): p.terminate()
122
123    def notifyModified(self , instance , modified):
124        self.policy.notifyModified(instance , modified)
125
126    #=====
127
128    # Un medio de almacenamiento genérico + caché de objetos almacenados
129
130    class Storage(object):
131        def __init__(self , interpreter):
132            self.interpreter = interpreter
133
134        def setManager(self , manager):
135            self.manager = manager
136            self.instanceTable = manager.getInstanceTable()
137

```

```

138     def storeObject(self , instance):
139         # Guardamos la instancia
140         self.doStoreObject(instance.getID() , instance)
141
142         # Añadimos la instancia a la tabla
143         self.instanceTable.addInstance(instance)
144
145     def retrieveObject(self , id):
146         try:
147             # Intentamos coger la instancia de la tabla
148             instance = self.instanceTable.getInstance(id)
149         except KeyError:
150             # Recuperamos la instancia
151             instance = self.doRetrieveObject(id)
152
153             # Añadimos la instancia a la tabla
154             # IMPORTANTE: hay que añadirla ANTES de restaurar sus
155             # campos, o podemos tener referencias circulares que
156             # causarían una recursividad infinita...
157             self.instanceTable.addInstance(instance)
158
159             # Restaura la clase y demás propiedades
160             instance.restore(self.interpreter , self)
161
162         return instance
163
164 #-----
165
166 class SimpleStorage(Storage):
167     def __init__(self , interpreter , appName):
168         Storage.__init__(self , interpreter)
169         self.instances = {}
170         self.appName = interpreter.getApplication().name
171         fileName = '%s.dat' % (appName, self.__class__.__name__)
172         self.fileName = fileName
173         try:
174             self.load(fileName)
175         except: # Deberíamos filtrar excepciones...
176             print 'SimpleStorage: Error al cargar la tabla'
177
178     def has_key(self , id):
179         return self.instances.has_key(id)
180
181     def doRetrieveObject(self , id):
182         instRepr = self.instances[id]
183         instance = pickle.loads(instRepr)
184         return instance
185
186     def doStoreObject(self , id , instance):
187         instRepr = pickle.dumps(instance)
188         self.instances[id] = instRepr
189
190     def commit(self):
191         self.dump(self.fileName)
192
193     def terminate(self):
194         pass
195
196     def dump(self , filename):
197         f = open(filename , 'wb')
198         pickle.dump(self.instances , f)
199         f.close()
200

```

```

201     def load(self , filename):
202         f = open(filename , 'rb')
203         self.instances = pickle.load(f)
204         f.close()
205
206 #-----
207
208 class DBMStorage(Storage):
209     def __init__(self , interpreter , appName):
210         Storage.__init__(self , interpreter)
211         self.appName = interpreter.getApplication().name
212         fileName = '%.%.s.dat' % (appName, self.__class__.__name__)
213         self.fileName = fileName
214         self.dbm = anydbm.open(fileName , 'c')
215
216     def has_key(self , id):
217         return self.dbm.has_key(id)
218
219     def doRetrieveObject(self , id):
220         instRepr = self.dbm[id]
221         instance = pickle.loads(instRepr)
222         return instance
223
224     def doStoreObject(self , id , instance):
225         instRepr = pickle.dumps(instance)
226         self.dbm[id] = instRepr
227
228     def commit(self):
229         self.dbm.sync()
230
231     def terminate(self):
232         self.dbm.close()
233
234 #-----
235
236 class BSDDBStorage(Storage):
237     def __init__(self , interpreter , appName , format , *args):
238         Storage.__init__(self , interpreter)
239         funcs = { 'hash': bsddb.hashopen ,
240                 'bt': bsddb.btopen ,
241                 'rn': bsddb.rnopen}
242         try:
243             openFunc = funcs[format]
244         except KeyError , e:
245             raise 'El formato "%s" no se reconoce' % format
246         self.appName = interpreter.getApplication().name
247         fileName = '%.%.%.s.dat' \
248                 % (appName, self.__class__.__name__ , format)
249         self.fileName = fileName
250         openArgs = (fileName , 'c') + tuple(args)
251         self.db = openFunc(*openArgs)
252
253     def has_key(self , id):
254         return self.db.has_key(id)
255
256     def doRetrieveObject(self , id):
257         instRepr = self.db[id]
258         instance = pickle.loads(instRepr)
259         return instance
260
261     def doStoreObject(self , id , instance):
262         instRepr = pickle.dumps(instance)
263         self.db[id] = instRepr

```

```
264
265     def commit(self):
266         self.db.sync()
267
268     def terminate(self):
269         self.db.close()
270
271 #=====
272
273 # Política de almacenamiento
274
275 class StoragePolicy(object):
276     def __init__(self):
277         self.modifiedInstances = {}
278
279     def setManager(self, manager):
280         self.manager = manager
281
282     def addInstance(self, instance):
283         id = instance.getID()
284         self.modifiedInstances[id] = instance
285
286     def deleteInstance(self, instance):
287         id = instance.getID()
288         try:
289             del self.modifiedInstances[id]
290         except KeyError, e:
291             pass
292
293     def storeInstances(self):
294         for inst in self.modifiedInstances.values():
295             self.manager.storeObject(inst)
296         self.manager.commitStorage()
297         self.modifiedInstances = {}
298
299 #-----
300
301 class SimplePolicy(StoragePolicy):
302     def __init__(self, maxChanges = 1):
303         StoragePolicy.__init__(self)
304         self.maxChanges = maxChanges
305         self.numChanges = 0
306
307     def notifyModified(self, instance, modified):
308         if modified:
309             self.addInstance(instance)
310             self.numChanges += 1
311             if self.numChanges == self.maxChanges:
312                 self.commit()
313
314     def commit(self):
315         self.numChanges = 0
316         self.storeInstances()
317
318     def terminate(self):
319         pass
320
321 #-----
322
323 class TimedPolicy(StoragePolicy):
324     def __init__(self, seconds):
325         StoragePolicy.__init__(self)
326         self.seconds = seconds
```

```

327         self.timer = None
328         self.lock = threading.Lock()
329
330     def notifyModified(self, instance, modified):
331         if modified:
332             self.lock.acquire()
333             self.addInstance(instance)
334             if not self.timer:
335                 self.timer = threading.Timer(
336                     self.seconds, self.commit)
337             self.lock.release()
338
339     def commit(self):
340         self.lock.acquire()
341         if self.timer: self.timer.cancel()
342         self.storeInstances()
343         self.lock.release()
344
345     def terminate(self):
346         pass

```

A.5 guidgen.py

```

1  # -*- coding: ISO-8859-1 -*-
2
3  """Parte constante de un guid dentro de una misma sesión de nitro"""
4  _prefix = None
5
6  def guid():
7      """choose_boundary() modificado so produced GUIDs also include
8         the current thread ID"""
9
10     global _prefix
11
12     import time
13     import random
14
15     if _prefix is None:
16         import socket
17         import os
18
19         hostid = socket.gethostbyname(socket.gethostname())
20
21         try:
22             uid = 'os.getuid()'
23         except:
24             uid = '1'
25
26         try:
27             pid = 'os.getpid()'
28         except:
29             pid = '1'
30
31         _prefix = hostid + '.' + uid + '.' + pid
32
33     try:
34         import thread
35         tid = 'thread.get_ident()'
36     except:
37         tid = '1'
38     timestamp = '%.3f' % time.time()

```

```
39     seed = 'random.randint(0, 32767)'  
40     return _prefix + ',' + tid + ',' + timestamp + ',' + seed
```