UNIVERSIDAD DE OVIEDO

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES E INGENIEROS INFORMÁTICOS DE GIJÓN

ÁREA DE LENGUAJES Y SISTEMAS INFORMÁTICOS

PROYECTO FIN DE CARRERA Nº 1022036

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

DOCUMENTO Nº 4

MANUAL DE USUARIO

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 Nº 1022036

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

DOCUMENTO Nº 4

MANUAL DE USUARIO

JAVIER NOVAL ARANGO
JUNIO 2003
TUTOR: FRANCISCO ORTÍN SOLER

Índice general

Índice de figuras

Índice de cuadros

INTRODUCCIÓN

En este documento se explicará cómo instalar (capítulo 1) y hacer funcionar (capítulo 2) tanto el proyecto como el sistema (*nitrO*) en el que está basado. También se hará una introducción al lenguaje desarrollado, Java−−, y se presentarán su gramática y una serie de diagramas sintácticos que servirán de ayuda para comprenderla (capítulo 3). De la misma forma se mostrarán las clases básicas implementadas para el lenguaje, haciendo una descripción de los métodos de que disponen y los operadores que soportan (capítulo 4).

Capítulo 1

INSTALACIÓN

1.1 Requisitos previos

El prototipo nitrO está programado en Python (ref. [7]), por lo tanto es necesario disponer del intérprete de este lenguaje. Puede ser descargado de `http://www.python.org/`, tanto en forma de código fuente como de binarios precompilados para varias plataformas.

Al instalar el intérprete, dependiendo de la plataforma se puede elegir la lista de módulos adicionales, entre los cuales debe estar marcado el paquete Tk para la creación de GUIs, puesto que es necesario para el funcionamiento del prototipo.

Hay que advertir que aunque el sistema nitrO fue creado usando Python 2.1, en el código del proyecto se han empleado algunas características que hacen necesario disponer de Python 2.2 o superior. En estos momentos la versión 2.3 del intérprete aún no ha sido catalogada como estable por sus desarrolladores, sin embargo se ha probado a emplearla para la ejecución del sistema con éxito y sin encontrar más fallos que los ya presentes en la versión 2.2 (concretamente, un fallo en la implementación del interfaz con Tk puede hacer que se cierre la aplicación de forma inesperada), por lo que se puede utilizar si se desea.

1.2 Instalación del sistema nitrO

El sistema se incluye comprimido en el CD que acompaña al proyecto. Para su instalación se necesita:

- Descomprimirlo en un directorio de nuestra elección.
- Añadir el nombre del directorio en el que hayamos dejado los archivos a la variable de entorno *PYTHONPATH*.

Para ejecutar el sistema tenemos dos posibilidades, dependiendo de nuestra plataforma:

- **Windows**: suponiendo que la instalación del intérprete haya creado correctamente las asociaciones de las extensiones de los ficheros de Python con el intérprete, bastará hacer doble click sobre el fichero nitro.pyw; de no ser así, deberemos ejecutar dicho archivo con el intérprete pythonw.
- **UN*X**: para facilitar la ejecución se ha creado un pequeño *script*, start.sh, que lanza el intérprete de Python en segundo plano (de forma que no inutilice la terminal); el *script* presupone que el intérprete se llama python2.2 y se encuentra situado en el directorio /usr/bin, de no ser así habrá que modificarlo convenientemente.

La descripción detallada de cada uno de los archivos y directorios puede encontrarse dentro del archivo readme.txt que acompaña a la distribución del sistema.

El código fuente, diseño y distintas baterías de prueba pueden descargarse de la dirección http://www.di.uniovi.es/reflection/lab/prototypes.html

1.3 Instalación del proyecto

El código creado para el proyecto se entrega también en formato comprimido. Los ficheros que lo componen deben extraerse en el mismo directorio en el que se instaló el sistema nitrO.

Para comprobar su funcionamiento una vez descomprimido, puede ejecutarse alguno de los ejemplos desarrollados (se encuentran situados en el subdirectorio ejemplos).

Capítulo 2

EL ENTORNO NITRO

2.1 Metalenguaje del sistema

Una de las características del sistema es su independencia del lenguaje de programación. Cualquier aplicación utilizando cualquier lenguaje, puede ser ejecutada en nuestro sistema, interactuando dinámicamente con otras aplicaciones desarrolladas en otros lenguajes. Una aplicación podrá también construirse utilizando un conjunto de lenguajes.

Para que esta independencia del lenguaje sea posible en nuestro sistema de programación, es necesario idear un mecanismo de especificación de lenguajes que separe su descripción del sistema computacional. Para ello se ha descrito un lenguaje de especificación de lenguajes: un metalenguaje. Haciendo uso de éste, se puede describir un lenguaje en nuestro sistema de tres modos:

1. Especificación del lenguaje mediante un archivo de extensión ml. Siguiendo la gramática § 2.1.1, se describe el lenguaje y se almacena en el directorio del sistema con igual nombre que el lenguaje creado, y extensión ml.

2. Especificándolo en el archivo de aplicación y anteponiéndolo a su código (como veremos en § 2.2.2).

3. Modificando un lenguaje ya existente previamente a la ejecución de una aplicación (detallado en § 2.2.4).

2.1.1 Gramática del metalenguaje

La gramática de la figura 2.1 representa la especificación del metalenguaje utilizado en las especificaciones de lenguajes para nuestro sistema; los elementos en negrita representan componentes léxicos, para separarlos de los propios de la notación EBNF (ref. [5]).

La primera parte es la identificación del lenguaje. Este nombre ha de ser único para el lenguaje y deberá coincidir con el nombre del archivo (si se está creando un archivo ml). A continuación se especifican las descripciones léxica, sintáctica, y los tokens de escape y reconocimiento automático.

2.1.2 Descripción léxica

La descripción léxica se lleva a cabo dentro de la sección Scanner empleando reglas libres de contexto. Cada regla ha de estar precedida de una cadena de caracteres (entre comillas dobles) descriptiva de su significado. La notación en la que puede expresarse cada regla es en BNF (Backus Naur Form) (ref. [5]).

```
<startLang>     ::= LANGUAGE = ID <scan> <parser> <skip> <notskip>
<scan>          ::= SCANNER = { <scannerRules> }
<parser>        ::= PARSER = { <parserRules> }
<skip>          ::= SKIP = { <tokens> }
<notskip>       ::= NOTSKIP = { <tokens> }
<scannerRules>  ::= {<SR> ;}
<SR>            ::= STRING ID -> <rightSR> {<moreRightSR>}
<rightSR>       ::= {<rightSRItem>} <ruleCode>
<moreRightSR>   ::= <rightSR>
                  | λ
<rightSRItem>   ::= STRING
                  | ID
<parserRules>   ::= {PR ; }
<PR>            ::= STRING ID -> <rightPR> {<moreRightPR>}
<rightPR>       ::= <rightPRItems> <ruleCode>
<rightPRItems>  ::= ID
                  | _REIFY_
<ruleCode>      ::= CODE
                  | λ
<moreRightPR>   ::= <rightPR>
                  | λ
<tokens>        ::= {STRING ;}
```

Figura 2.1: Gramática del metalenguaje en nitrO

Los elementos no terminales son identificadores y los terminales cadenas de caracteres sensibles a mayúsculas/minúsculas. Toda producción ha de finalizar con un punto y coma, y la parte derecha de la producción se separa de la izquierda con la pareja de caracteres ->. Toda producción puede tener alternativas en su parte derecha, separadas con el carácter |.

Puesto que el tratamiento de estas producciones está basado en un algoritmo descendente con retroceso (backtracking), si dos partes derechas pueden ser válidas para la entrada analizada, la primera será analizada y la segunda ignorada. Es buen criterio a seguir si esto se produce, el ubicar con anterioridad aquellas reglas que posean una parte derecha de mayor longitud que aquellas con las que pueda tener conflictos. Por lo tanto, la producción al vacío (λ) deberá ubicarse como la última parte derecha de toda producción.

Las producciones pueden tener asociadas reglas semánticas para constituir definiciones dirigidas por sintaxis (ref. [1]). El modo en que éstas son codificadas se describe en § 2.1.5.

2.1.3 Descripción sintáctica

Las reglas sintácticas de nuestro metalenguaje se ubican en la sección Parser. El modo en el que éstas son representadas coincide con las empleadas en la descripción léxica. La única diferencia es que la parte derecha de una regla sintáctica no puede poseer símbolos gramaticales terminales—éstos deben estar previamente especificados en la parte léxica.

El símbolo gramatical no terminal, ubicado en la parte izquierda de la primera producción, representa el símbolo inicial de la gramática; no es necesario que éste posea un identificador determinado.

La separación entre reglas léxicas y sintácticas supone básicamente una agrupación conceptual. Además, como se comentará en § 2.1.4, en el reconocimiento de una producción léxica se ignora la detección de tokens de escape—no en el caso de las reglas sintácticas.

2.1.4 Tokens de escape y reconocimiento automático

Las dos secciones restantes (Skip y NotSkip) facilitan la eliminación y reconocimiento automático de componentes léxicos en la aplicación a analizar. Si queremos que el analizador léxico del procesador de lenguaje elimine automáticamente un conjunto de tokens, debemos especificar éstos dentro de la sección Skip. El analizador sintáctico nunca tendrá noción de ellos.

La sección NotSkip produce el efecto contrario que Skip. Si queremos reconocer automáticamente un conjunto de tokens, sin necesidad de especificar su aparición sintácticamente, podremos hacerlo ubicándolos en esta sección. Un ejemplo de este tipo de tokens puede ser el tabulador en el lenguaje Python (ref. [7]): el código ha de esta indentado (sangrado) mediante tabuladores para saber a qué estructura de control pertenece, pero, ¿cómo contemplar esto en su gramática?

Si un token NotSkip es reconocido automáticamente, su lexema aparecerá en el texto (atributo text del nodo—ver siguiente punto) asociado al no terminal de la parte izquierda de la producción analizada.

2.1.5 Especificación Semántica

Las producciones pueden especificar al final de ellas código Python representativo de su semántica. Este código ha de estar ubicado entre los pares de caracteres <# y #>. Al codificar esta semántica, se ha de tener en cuenta que Python utiliza los tabuladores como indentadores obligatorios y el salto de línea como separador de instrucciones.

La ejecución de una acción semántica posee el siguiente contexto:

- Todos los símbolos gramaticales del árbol tienen asociado un objeto. Éstos forman parte de una lista denominada nodes. El primer elemento (nodes[0]) es el objeto asociado al no terminal de la izquierda; el resto representan los nodos de la parte derecha, enumerados de izquierda a derecha.
- Todo nodo del árbol posee un atributo text que representa el código reconocido, eliminando los tokens Skip y habiendo reconocido automáticamente los NotSkip.
- La función global write visualiza en la ventana gráfica de la aplicación la cadena de caracteres pasada como parámetro.
- El objeto global nitrO, nos brinda todos los servicios de nuestro sistema computacional—ver § 2.3.3.

Puesto que todo nodo es un objeto Python, y este lenguaje posee reflectividad estructural, podemos asignarle cualquier atributo dinámicamente. Al no tener comprobación estática de tipos (ref. [2]), podemos crear en tiempo de ejecución nuevos atributos mediante el operador de asignación. Esto hace que la herramienta suponga un mecanismo de codificación de definiciones dirigidas por sintaxis (ref. [1]).

Una vez que el árbol sintáctico de una aplicación haya sido creado, se ejecutará únicamente el código asociado a la producción del símbolo inicial; ésta deberá encargarse de llamar al resto de las evaluaciones semánticas. La regla semántica de un nodo se ejecuta al pasarle a éste el mensaje execute. Por lo tanto, si queremos que se evalúe un nodo, debemos invocar su método execute.

Para ver ejemplos prácticos de la descripción de lenguajes, examínense los archivos de extensión ml de la distribución del prototipo.

2.1.6 Instrucción reify

Aunque nuestro sistema sea independiente del lenguaje de programación, su flexibilidad se centra en la utilización de una instrucción que todo lenguaje posee: la instrucción reify. El programador de lenguajes sólo debe ubicar el terminal _REIFY_ en aquella parte de la gramática donde pueda aparecer dicha instrucción.

La utilidad y funcionamiento de esta instrucción serán explicados en § 2.2.3.

2.2 Aplicaciones del sistema

2.2.1 Gramática de Aplicaciones

La codificación de una aplicación en nuestro sistema, indistintamente del lenguaje de programación seleccionado, ha de seguir la gramática EBNF de la figura 2.2.

<*startApp*>	::=	**APPLICATION = STRING LANGUAGE** <*langSpec*> <*langAlt*> **APPCODE**
<*langSpec*>	::=	**STRING**
	|	<*startLang*>
<*langAlt*>	::=	**+ CODE**
	|	λ

Figura 2.2: Gramática de una aplicación en nitrO

El identificador único de la aplicación es la cadena de caracteres entre comillas dobles que se asigna a la palabra reservada Application. La parte final de un archivo de aplicación siempre es el código de ésta (APPCODE), siguiendo la gramática del lenguaje utilizado.

A la hora de implementar una aplicación, o un módulo de aplicación, dentro de nuestro sistema, debemos tener en cuenta que el lenguaje a utilizar ha de haber sido especificado de alguna de las tres formas mencionadas en 2.1.

Si optamos por ubicar la especificación del lenguaje de programación en un archivo de extensión ml, éste deberá llamarse igual que el identificador del lenguaje y será asignado a la palabra reservada Language. El sistema buscará su especificación en el directorio del sistema, empleando el nombre de archivo especificado.

2.2.2 Aplicaciones autosuficientes

El segundo modo de especificar el lenguaje de programación a utilizar por una aplicación es incluyéndolo en la propia aplicación. Antes su codificación, puede especificarse el lenguaje a utilizar siguiendo la gramática descrita en 2.1.1. Una vez descrito éste, la aplicación se codificará e interpretará en base a esta descripción.

La oportunidad de crear aplicaciones que puedan describir su propia sintaxis y semántica, hacen que éstas sean autosuficientes y directamente ejecutables. En cualquier plataforma en la que nuestro sistema esté instalado, este tipo de aplicaciones puede ejecutarse e interactuar con el resto de programas existentes en el sistema. Un ejemplo práctico de su utilización, haciendo uso de un paquete de distribución, es el desarrollo de un sistema de agentes móviles (ref. [3]) independientes del lenguaje; las aplicaciones viajan por la red y se ejecutan en cualquier máquina con el lenguaje que ellas deseen, sin necesidad de que éste esté instalado en la plataforma de ejecución.

2.2.3 Reflectividad No Restrictiva

Cuando desarrollamos la especificación de un lenguaje para nuestro sistema, definimos sus aspectos léxicos, sintácticos y semánticos. Una vez descritos éstos, una aplicación se codifica, valida y ejecuta en base a esta especificación que se mantiene invariable a lo largo de su ciclo de vida. Uno de los objetivos principales de este prototipo es obtener un modo de flexibilizar la especificación de un lenguaje, desde sus propias aplicaciones.

Como mostrábamos en 2.1.6, todo lenguaje de nuestro sistema dispone de una instrucción Reify con tan solo ubicar el terminal _REIFY_ en aquella posición de la gramática donde pueda aparecer dicha sentencia. Esta instrucción está constituida por la palabra reservada reify, seguida de código Python entre las parejas de caracteres <# y #>.

El código asociado a una instrucción reify será evaluado en el espacio computacional del intérprete en lugar ejecutarse en el contexto de la aplicación. Se produce un salto real en la torre de teórica de intérpretes propuesta por Smith (ref. [6]). El resultado es que, en la codificación de aplicaciones, es posible especificar, aumentar o modificar las características del lenguaje de programación, como si nos encontrásemos diseñando éste.

El ejecutar un código en el contexto computacional de su intérprete nos permite:

- Conocer el estado del sistema (objetos, clases, variables...): introspección.
- Acceder y modificar la estructura de sus objetos: reflectividad estructural.
- Modificar y aumentar la semántica de su lenguaje de programación: reflectividad computacional o de comportamiento.
- Modificar la sintaxis del lenguaje por la propia aplicación: reflectividad de lenguaje.

Un breve ejemplo de las tres primeras características se muestra en el archivo musimApp1.na, entregado en el directorio de pruebas de la distribución del prototipo.

2.2.4 Reflectividad de Lenguaje

La tercera y última forma de especificar un lenguaje de programación en nuestro prototipo es mediante la modificación de un lenguaje existente, previamente definido por alguno de los dos mecanismos ya mencionados. Una vez identificado el lenguaje a utilizar, utilizando el lexema +, el programador puede ubicar código Python a ejecutar en el contexto de intérprete, de igual que la instrucción reify descrita en el punto anterior.

El código descrito se evaluará antes de la ejecución de la aplicación, significando una personalización del lenguaje de programación para la ejecución de una determinada aplicación: el lenguaje no se modifica para todo el sistema, sino que es amoldado a la aplicación concreta.

El modo en el que este código actúa para modificar la especificación de un lenguaje es accediendo al conjunto de reglas que describen el lenguaje y, por medio de la utilización de reflectividad estructural, modificar éstas para obtener la personalización del lenguaje. Un ejemplo de esta posibilidad está codificado en el archivo printApp.na del prototipo distribuido.

2.3 Interfaz Gráfico

Para facilitar la programación de lenguajes y aplicaciones en nuestro sistema, el prototipo dispone de un pequeño interfaz gráfico. Este interfaz está construido sobre el estándar TK (ref. [4]) independiente de la plataforma a utilizar.

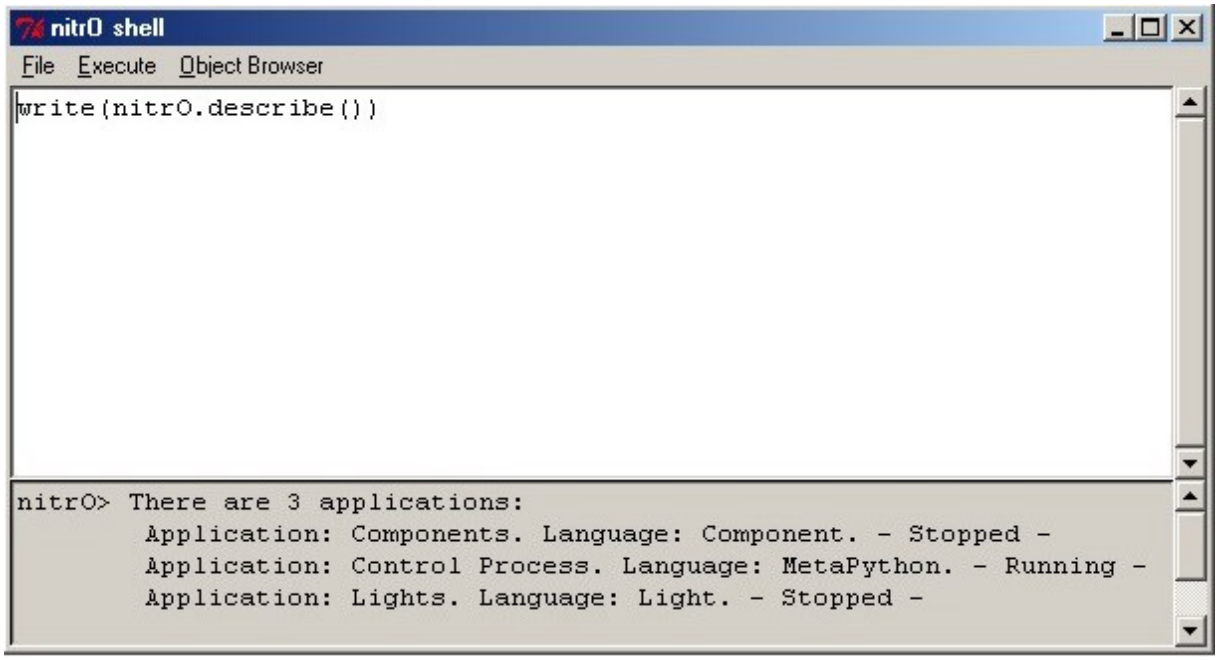


Figura 2.3: Ventana principal del prototipo.

2.3.1 Intérprete de Comandos

La ventana principal del sistema tiene el siguiente aspecto:
La ventana está dividida en tres partes:

1. Menú del sistema.
2. Editor de código del intérprete de comandos.
3. Ventana de salida, en la que se visualiza el resultado de ejecutar los comandos especificados en el editor.

El usuario del sistema describe los comandos deseados en el editor y evalúa éstos seleccionando la opción Execute del menú del sistema. La principal cuestión que puede preguntarse el usuario es ¿qué comandos poseo para acceder al sistema?

El código susceptible de ser ejecutado es cualquier programa Python; éste será evaluado en el contexto de ejecución de nuestro sistema. Además, este contexto tendrá dos elementos añadidos para acceder al sistema:

1. La función write, que recibe cualquier parámetro y lo muestra en la ventana de salida. Será utilizado cuando deseemos conocer algún valor. Por ejemplo, en la Figura A.1 se muestra la cadena de caracteres que describe el sistema.
2. El objeto nitrO, que nos da acceso a todos los elementos del sistema. Ofrece un conjunto de objetos (lenguajes y aplicaciones) así como un conjunto de métodos para trabajar sobre éstos. El conjunto de objetos y métodos existentes podrá consultarse dinámicamente, gracias al carácter introspectivo del sistema, mediante el Object Browser (§ 2.3.3).

2.3.2 Archivos Empleados

En la utilización nuestro sistema podemos diferenciar tres tipos de archivos:

1. Archivos de especificación de lenguajes de programación codificados mediante nuestro metalenguaje. Estos archivos poseen la extensión ml, siguen el metalenguaje descrito en § 2.1.1, y deben ubicarse en el directorio del sistema.

2. Archivos de aplicación. Poseen la extensión na (*nitrO application*) y describen aplicaciones del sistema en un determinado lenguaje de programación.

3. Archivos script o comandos. La extensión ns (*nitrO script*) identifica una secuencia de sentencias a ejecutar por el intérprete de comandos del sistema.

El tratamiento del último tipo de archivos se lleva a cabo mediante el menú de archivo (*File*) de nuestro prototipo.

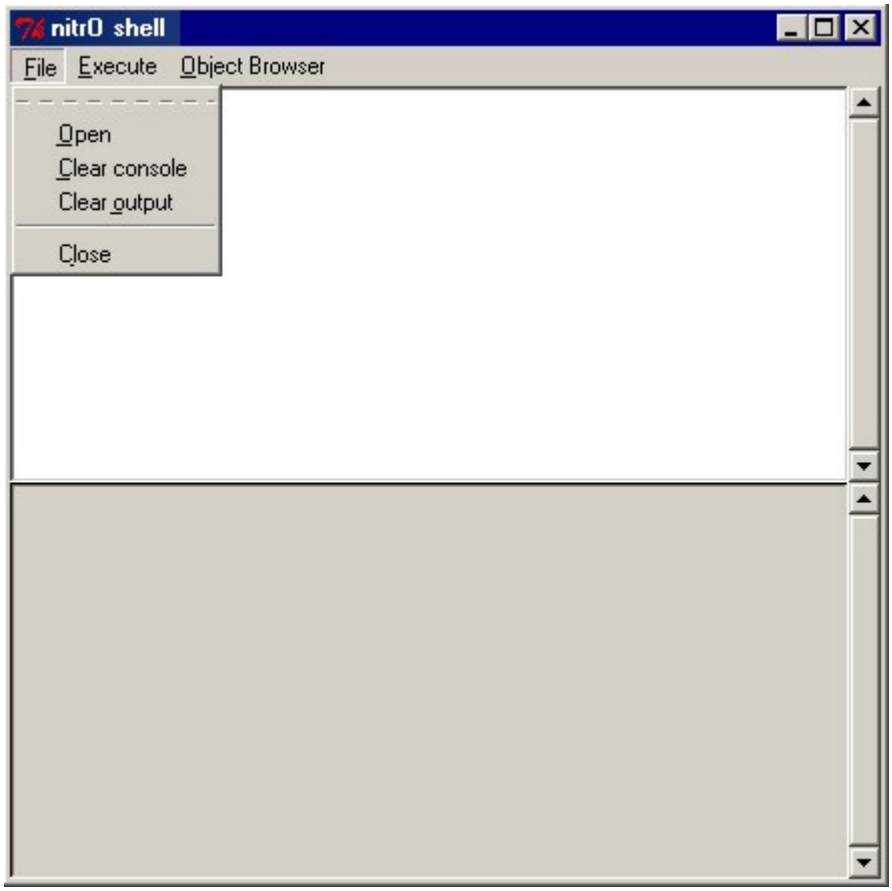


Figura 2.4: Menú archivo del prototipo.

Mediante este menú, pueden cargarse archivos de script, así como limpiar el editor y la ventana de salida y finalizar la ejecución del sistema.

2.3.3 Introspección del sistema

Todo el acceso al sistema se obtiene a través del objeto nitrO, que ofrece un conjunto de atributos y métodos descriptores del sistema. Sin embargo, para utilizar éstos, debemos conocer dinámicamente su existencia y funcionalidad. Para ello tenemos dos herramientas en el sistema:

1. Métodos describe: La mayor parte de los objetos del sistema implementan un método describe que nos muestra su descripción. Pasándoles este mensaje y escribiendo en la

consola el resultado de su invocación (mediante la función write) obtendremos información dinámica de su estado.

Un ejemplo de esta utilización se muestra en la Figura 2.4.

2. Object Browser: Esta última opción del menú nos muestra el conjunto de todos los atributos y métodos del objeto nitrO, y por lo tanto de todo el sistema. Haciendo uso de esta herramienta, el programador podrá conocer el conjunto de aplicaciones y lenguajes existentes, su estructura y sus mensajes, y podrá programar y modificar el sistema en función de la información consultada.

En la Figura 2.5 se muestra la información dinámica del sistema. A modo de ejemplo, comentaremos parte de ésta:

- Accediendo al atributo __class__ de nitrO, obtenemos todos los métodos de éste (execute, executeFile...) así como la descripción de la clase y cada uno de los mensajes, consultando sus atributos __doc__.
- El atributo apps es una lista de las aplicaciones existentes en el sistema (Component, ControlProcess y Lights).
- Cada aplicación posee un conjunto de atributos y métodos (su clase) que nos ofrecen información y funcionalidad de ésta. Uno de estos atributos es siempre la especificación de su lenguaje de programación mediante una estructura de objetos—atributo language.
- Por cada lenguaje de programación tenemos un conjunto de reglas léxicas y sintácticas libres de contexto (lexicalSpec y syntacticSpec), así como los componentes léxicos a descartar y a reconocer automáticamente (skipSpec y notSkipSpec).

Como se aprecia en este ejemplo, la información del sistema es elevada y compleja y, por tanto, una herramienta introspectiva como el Object Browser facilita el trabajo al programador.

2.3.4 Ejecución de aplicaciones

Cada vez que se ejecuta una aplicación en nuestro sistema, se crea una ventana gráfica para ésta. La utilización de la función write por esta aplicación, supone la visualización de la información pasada como parámetro en su ventana asociada.

En la figura anterior se muestra cómo el sistema está ejecutando 3 aplicaciones, cada una con su propia ventana de visualización. Una aplicación puede cerrarse cuando esté parada (al final de su título aparezca Stopped). El hecho de cerrar su ventana supone la eliminación de la aplicación asociada dentro del sistema.

Si deseamos finalizar una aplicación en ejecución, deberemos, desde el editor de comandos, enviar un mensaje stop al objeto nitrO, pasándole como parámetro el identificador de la aplicación. El título de ésta mostrará cuándo esté tratando de finalizar (*trying to stop*) y finalmente su estado de parada (*stopped*).

Cerrar la ventana principal del sistema supone cerrar el conjunto de aplicaciones existentes.

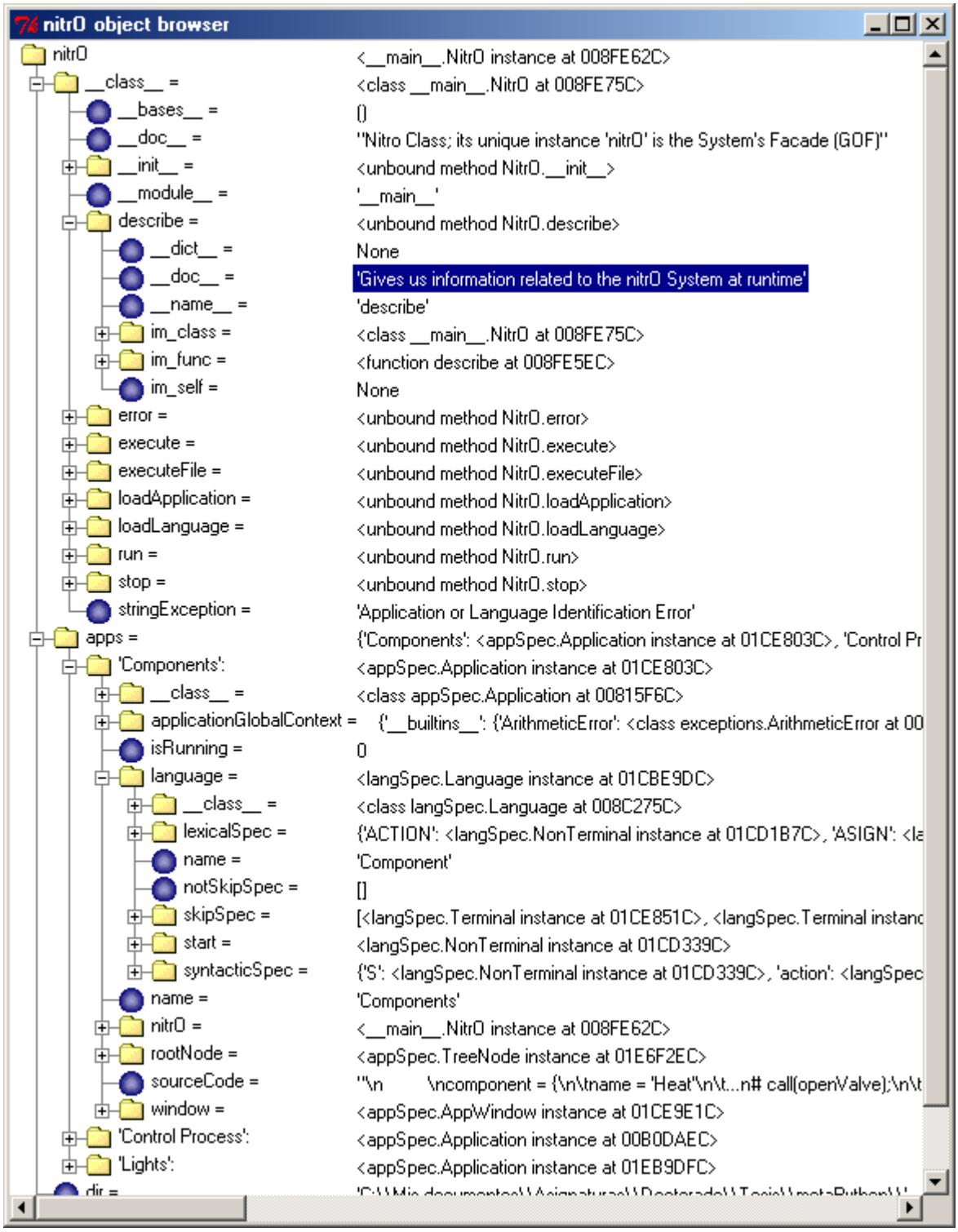


Figura 2.5: Estado del sistema en tiempo de ejecución, mostrado por el Object Browser.

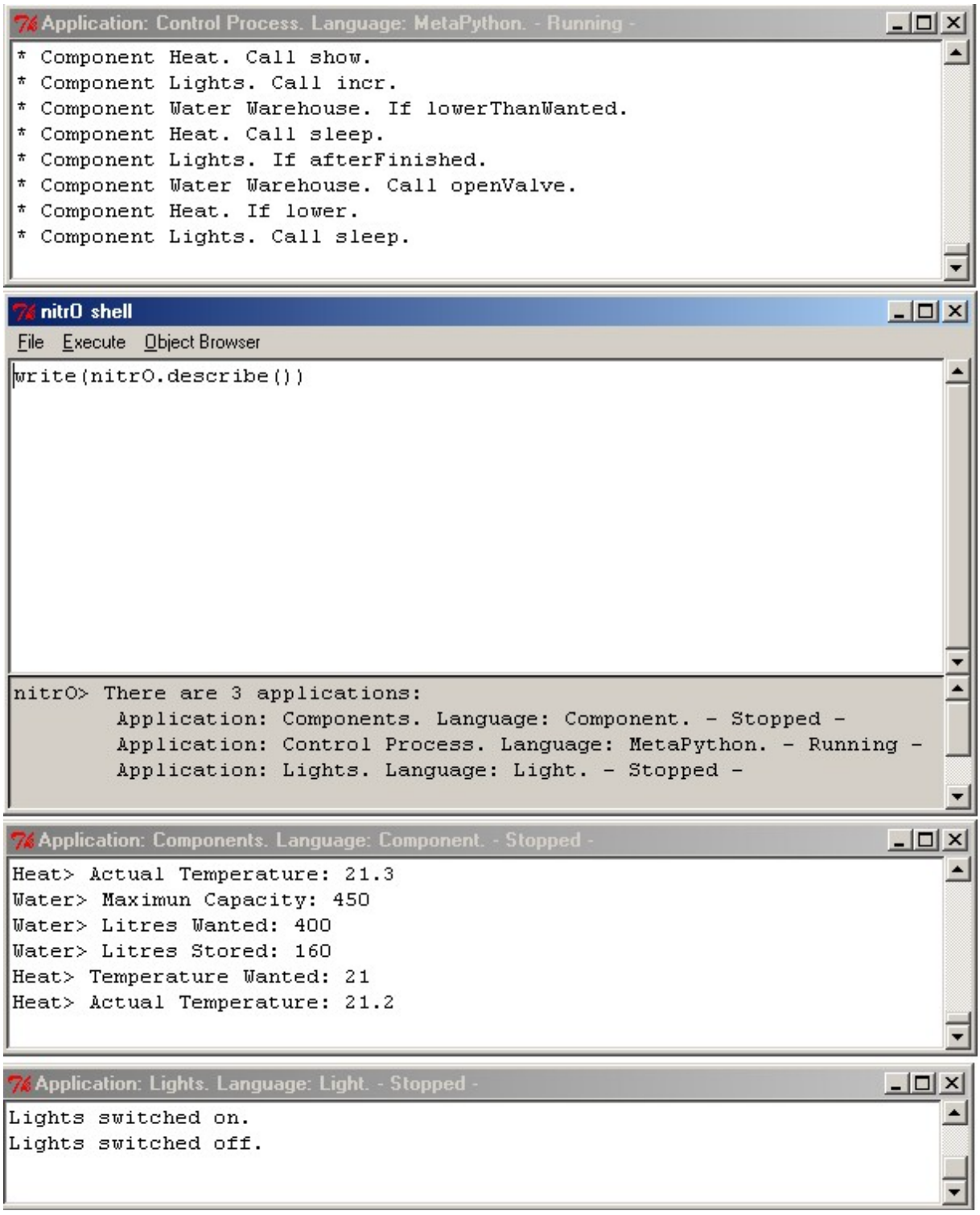


Figura 2.6: Ejecución de aplicaciones en el sistema.

Capítulo 3

EL LENGUAJE JAVA−−

3.1 Introducción al lenguaje

Java−−, como su nombre indica, está basado en una versión simplificada del lenguaje Java, con algunos cambios—el principal, que no existen tipos primitivos, sino que todo son objetos.

Algunas de las principales carencias de este lenguaje son:

- No soporta *interfaces*.
- No se pueden crear clases anidadas.
- El código del programa no se puede repartir entre varios archivos, y tampoco se pueden crear paquetes.
- No se soporta el control de acceso a los miembros de las clases—todos los atributos y métodos son públicos.
- No existen métodos ni atributos estáticos.

En las próximas secciones se mostrarán algunos ejemplos de uso del lenguaje, explicando sus capacidades y los puntos en los que difiere de Java. Se advierte que se supondrán conocimientos del lenguaje Java o alguno similar (C++, por ejempo).

3.1.1 *¡Hola mundo!*

La figura 3.1 muestra la versión Java−− del clásico programa *¡Hola mundo!*:

```
1 Application = "JavaApp"
2 Language = "Java"
3
4 {
5         Console cons = new Console();
6         cons.println('¡Hola mundo!');
7 }
```

Figura 3.1: El programa *¡Hola mundo!*

Procederemos a explicar el programa paso a paso:

1. En la línea 1 se le da nombre a la aplicación. Será válido cualquier nombre siempre que sea único dentro del conjunto de aplicaciones que se estén ejecutando en el sistema (de lo contrario se sobreescribiría la aplicación homónima al ejecutarlo).

2. La línea 2 le indica al sistema el nombre del lenguaje empleado, que debe ser “Java”.

3. El bloque de código que aparece en las líneas 4–7 es el cuerpo del programa principal, nuestro punto de entrada.

4. En la línea 5 se emplea el operador new para crear una nueva instancia de la clase Console y se asigna a la referencia llamada cons.

5. Por último, en la línea 6 escribe la cadena “¡Hola mundo!” por la salida estándar.

Como se puede comprobar, la mayor diferencia respecto a Java se produce por no soportar métodos estáticos, lo que impide usar una clase con un método main() como en el lenguaje original. En su lugar, al final de todo programa Java−− aparece un bloque de código situado fuera de cualquier clase, y ese bloque será nuestro programa principal.

3.1.2 Control de flujo

Se han incluido las sentencias *if-then-else*, *for* y *while* para controlar el flujo de ejecución del programa. Un ejemplo de su uso aparece en la figura 3.2, junto con algunas operaciones con enteros y cadenas.

```
1  Application = "JavaApp"
2  Language = "Java"
3
4  {
5          Console cons = new Console();
6          Integer a = 1, b = 2;
7
8          if(a < b)
9                  cons.println('Es menor');
10         else
11                 cons.println('No es menor');
12
13         for(Integer i = 0; i < 10; ++i)
14                 cons.println('i vale ' + i.toString());
15
16         Integer x = 3;
17         while(x > 0) {
18                 cons.println(x.toString());
19                 --x;
20         }
21 }
```

Figura 3.2: Ejemplo de control del flujo de ejecución del programa

Se puede observar que no difiere mucho de un programa Java equivalente, salvo en detalles menores como el empleo de la comilla simple para las cadenas de texto en lugar de la comilla doble. Sin embargo, hay una diferencia importante, y es la interpretación de los valores lógicos. En Java existe un tipo básico *bool* para representar el resultado de las operaciones lógicas. Sin embargo, Java−− carece de tipos primitivos, por lo que hubo que buscar otra forma de representar esos resultados. La elección fue emplear el mismo sistema que en Lisp: una referencia nula tiene valor lógico *falso*, mientras que una no nula tiene valor lógico *verdadero*. En cuanto a los operadores soportados, la clase Bool no define ningún método que implemente operadores, y los motivos se explicarán en la sección 3.1.3.

3.1.3 Operadores internos y externos

En el lenguaje Java−− existen dos tipos de operadores, que hemos llamado *internos* y *externos*. Los primeros reciben su nombre por estar implementados como métodos de las instancias (por lo que podrían ser modificables, aunque actualmente el lenguaje no lo permite—sin embargo la estructura necesaria ya está implementada). Los operadores externos, en cambio, trabajan sobre las referencias, y por tanto son ajenos a las instancias. La tabla 3.1 muestra qué operadores pertenecen a cada categoría.

	Internos	**Externos**
Aritméticos	+, -, *, /, %, ++, –	
Lógicos	<, <=, >, >=, ==, !=	&&, ||, !, is, ?:

Cuadro 3.1: Clasificación de operadores internos y externos

Como se puede observar en la tabla, todos los operadores aritméticos son internos (como es lógico, puesto que necesitan el valor asociado a la instancia), mientras que los lógicos están repartidos entre aquellos que emplean el valor (y por tanto son internos) y los que se limitan a consultar si sus argumentos son o no nulos, o si dos referencias apuntan a la misma instancia (operador is), por lo que son externos.

Tal y como muestra la tabla, todas las operaciones que tienen sentido con valores booleanos se corresponden con operadores externos, y por ese motivo la clase Bool no tiene ningún otro operador definido—simplemente no son necesarios.

3.1.4 Clases y métodos

La figura 3.3 muestra el siguiente ejemplo, en el que se introducen las clases y los métodos.

Nuevamente se comprueba que no hay grandes diferencias con lo que sería un programa Java equivalente. Este sencillo código también permite mostrar el funcionamiento de la herencia, los métodos virtuales y el acceso a los atributos. Se comprueba cómo un método accede únicamente a los atributos definidos en su clase o en sus clases base, aunque en clases derivadas se oculten con otros. También se ve cómo empleando super se puede acceder a la clase base.

3.1.5 Constructores

El ejemplo de la figura 3.4 muestra cómo se define un constructor en el lenguaje Java−−, como un método con el mismo nombre de la clase y sin tipo de retorno. También se muestra la sentencia *super* (no confundir con la expresión), que sirve para invocar al constructor de la clase padre y que, de aparecer, debe ser la primera sentencia en el cuerpo del contructor.

3.1.6 Sobrecarga de métodos

El lenguaje Java−− también soporta la sobrecarga de métodos, tal y como se muestra en la figura 3.5. Por supuesto, los constructores también pueden sobrecargarse, al no ser más que métodos con ciertos "privilegios" especiales.

3.1.7 Reflectividad

La figura 3.6 muestra cómo empleando la sentencia reify se puede emplear código Python dentro de un programa Java−−, y acceder así a toda la estructura interna del sistema. En el ejemplo también se puede observar la forma de acceder a una variable local desde el código Python.

```
1  Application = "JavaApp"
2  Language = "Java"
3
4  class A {
5          Integer i = 10;
6          Integer j = 20;
7
8          void metodo(Console cons) {
9                  cons.println('Estamos en la clase A');
10                 cons.println('func devuelve ' + func().toString());
11         }
12
13         Integer func() {
14                 return i + j;
15         }
16 }
17
18 class B extends A {
19         Integer i = 30;
20         Integer j = 40;
21
22         void metodo(Console cons) {
23                 cons.println('Estamos en la clase B');
24                 cons.println('func devuelve ' + func().toString());
25                 cons.println('Pero en el contexto de A devuelve '
26                         + super.func().toString());
27         }
28 }
29
30 {
31         Console cons = new Console();
32
33         A a = new A();
34         a.metodo(cons);
35
36         B b = new B();
37         b.metodo(cons);
38 }
```

Figura 3.3: Ejemplo del control del flujo de ejecución del programa

```
1  Application = "JavaApp"
2  Language = "Java"
3
4  class A {
5          Integer i;
6
7          A(Integer i) {
8                  this.i = i;
9          }
10
11         void metodo(Console cons) {
12                 cons.println('i = ' + i.toString());
13         }
14 }
15
16 class B extends A {
17         B() {
18                 super(5);
19         }
20 }
21
22 {
23         Console cons = new Console();
24
25         A a = new A(3);
26         a.metodo(cons);
27
28         B b = new B();
29         b.metodo(cons);
30 }
```

Figura 3.4: Ejemplo del uso de constructores

```
1  Application = "JavaApp"
2  Language = "Java"
3
4  class A {
5          void metodo(Console cons, Integer i) {
6                  cons.println('i = ' + i.toString());
7          }
8
9          void metodo(Console cons, String s) {
10                 cons.println('s = ' + s);
11         }
12 }
13
14 {
15         Console cons = new Console();
16
17         A a = new A();
18         a.metodo(cons, 1);
19         a.metodo(cons, 'hola');
20 }
```

Figura 3.5: Ejemplo de la sobrecarga de métodos

```
1  Application = "JavaApp"
2  Language = "Java"
3
4  class A {
5          void metodo(Console cons) {
6                  reify
7  <#
8  print 'Hola desde Python'
9  cons = theInterpreter.getSymbolTable().getVar('cons').getInstance()
10 print 'La instancia de la consola:', cons
11 #>
12         }
13 }
14
15 {
16         Console cons = new Console();
17
18         A a = new A();
19         a.metodo();
20 }
```

Figura 3.6: Ejemplo de reflectividad

Al emplear reify debemos tener en cuenta que Python tiene en cuenta la indentación, aunque Java−− no lo haga. Por tanto, deberemos comenzar las sentencias en la primera columna (mientras no tengamos bloques, naturalmente), incluyendo el cierre de reify, o Python se quejará por esos espacios inesperados.

3.2 Referencia

3.2.1 Operadores

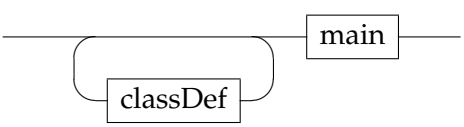
La tabla 3.2 muestra los operadores existentes en el lenguaje, así como su asociatividad. Se encuentran ordenados de mayor a menor prioridad.

Como se puede observar no se han implementado los operadores de asignación ampliados existentes en el lenguaje Java (+=, *=, ...). Sin embargo añadirlos no sería problemático, pues podrían convertirse internamente en los operadores normales (ya que a += b ≡ a = a + b, por ejemplo).

3.2.2 Diagramas sintácticos

S

bodyStatements

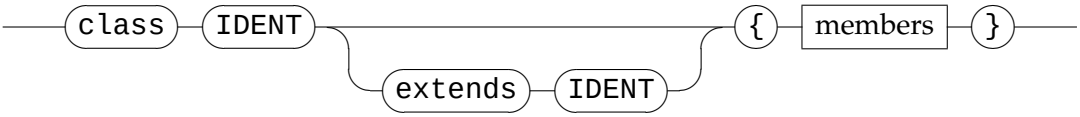


main

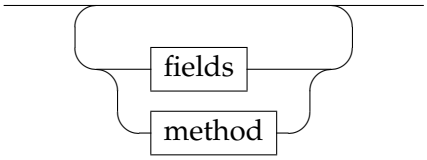
Símbolo	**Nombre o significado**	**Asociatividad**
()	Llamada a función	Izquierda a derecha
.	Miembro de una instancia	
++	Incremento	Derecha a izquierda
–	Decremento	
!	*NO* lógico	Derecha a izquierda
+	Más unario	
-	Menos unario	
(*tipo*)	Ahormado de tipos	
*	Multiplicación	Izquierda a derecha
/	División	
%	Resto	
+	Suma	Izquierda a derecha
-	Resta	
==	Igualdad	Izquierda a derecha
!=	Desigualdad	
is	Identidad	
<	Menor que	Izquierda a derecha
<=	Menor o igual que	
>	Mayor que	
>=	Mayor o igual que	
&&	*Y* lógico	Izquierda a derecha
||	*O* lógico	Izquierda a derecha
?:	Condicional	Derecha a izquierda
=	Asignación	Derecha a izquierda
,	Coma	Izquierda a derecha

Cuadro 3.2: Operadores del lenguaje Java−−

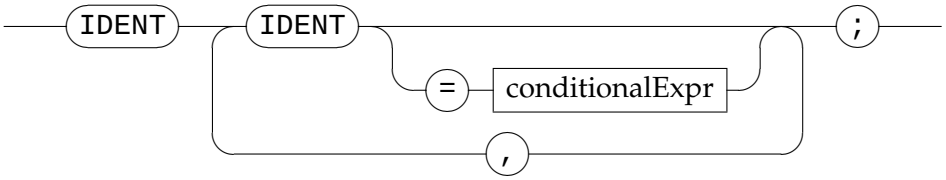
classDef



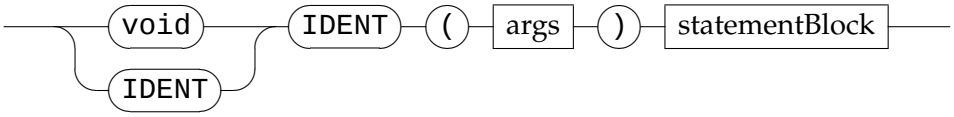
members



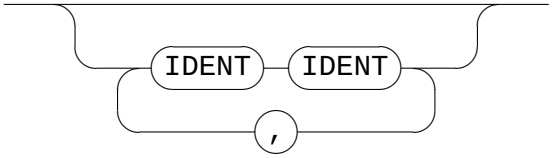
fields



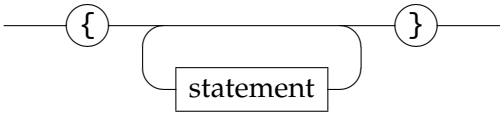
method



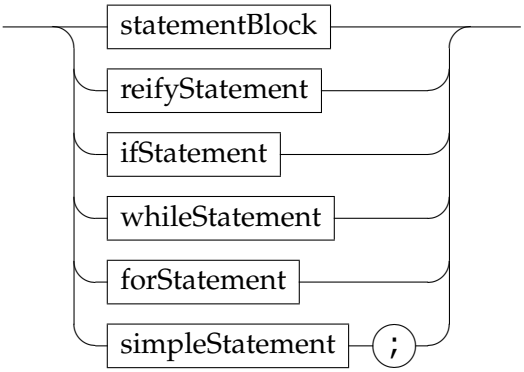
args



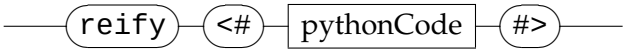
statementBlock



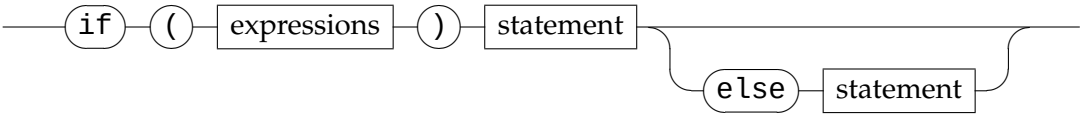
statement



reifyStatement

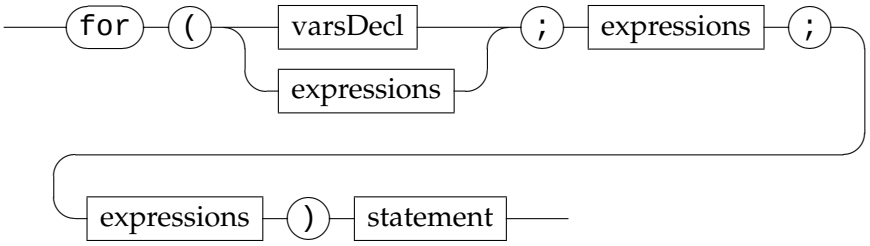


ifStatement

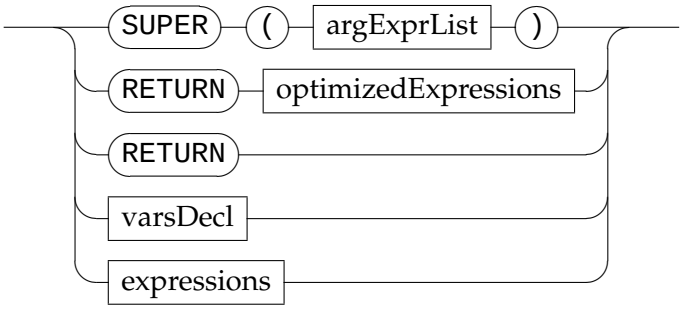


whileStatement

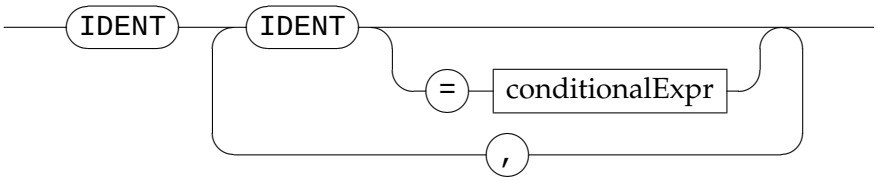
forStatement



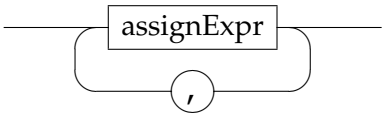
simpleStatement



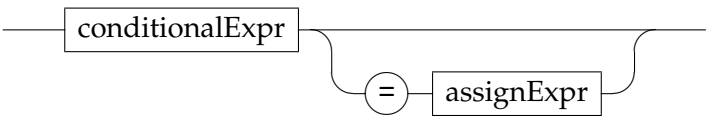
varsDecl



expressions

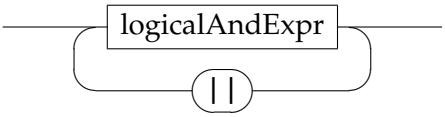


assignExpr

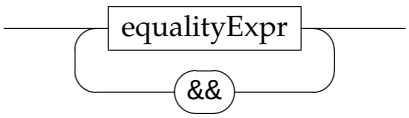


conditionalExpr

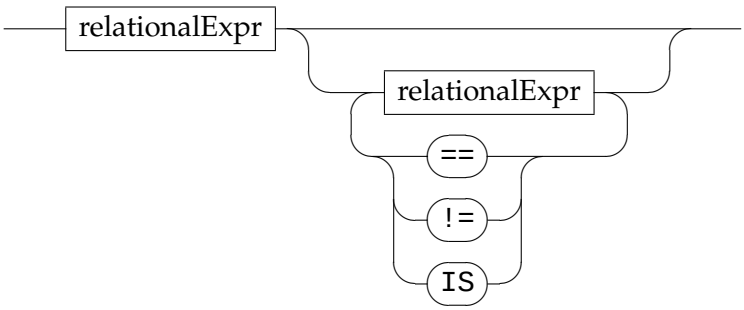
logicalOrExpr



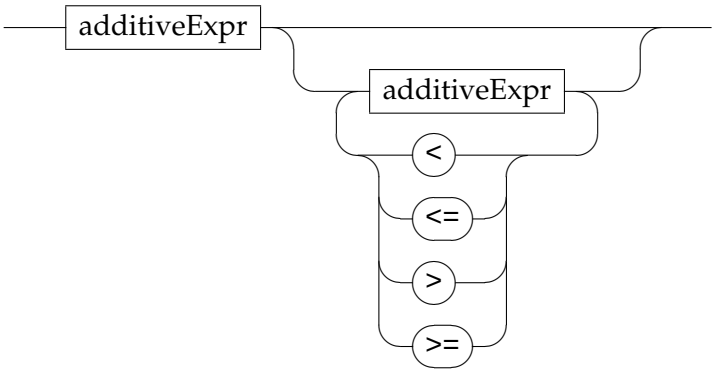
logicalAndExpr



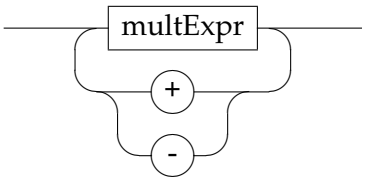
equalityExpr



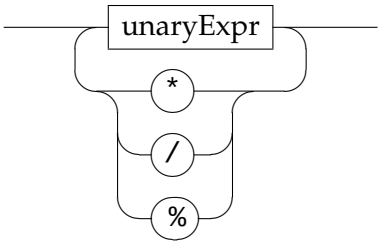
relationalExpr



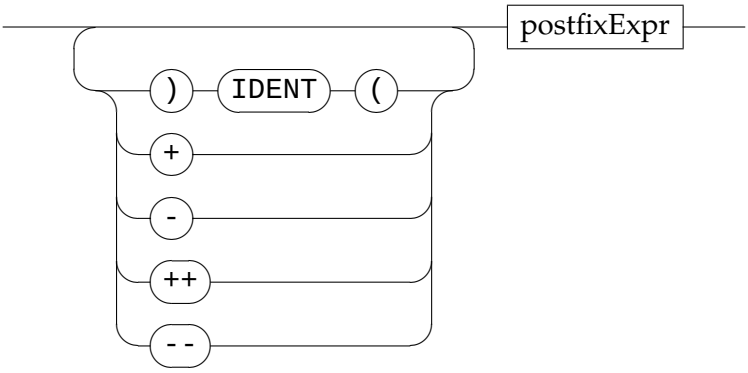
additiveExpr



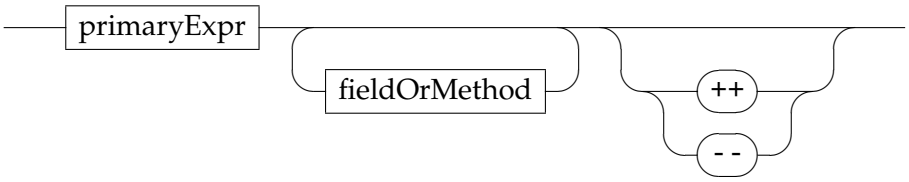
multExpr



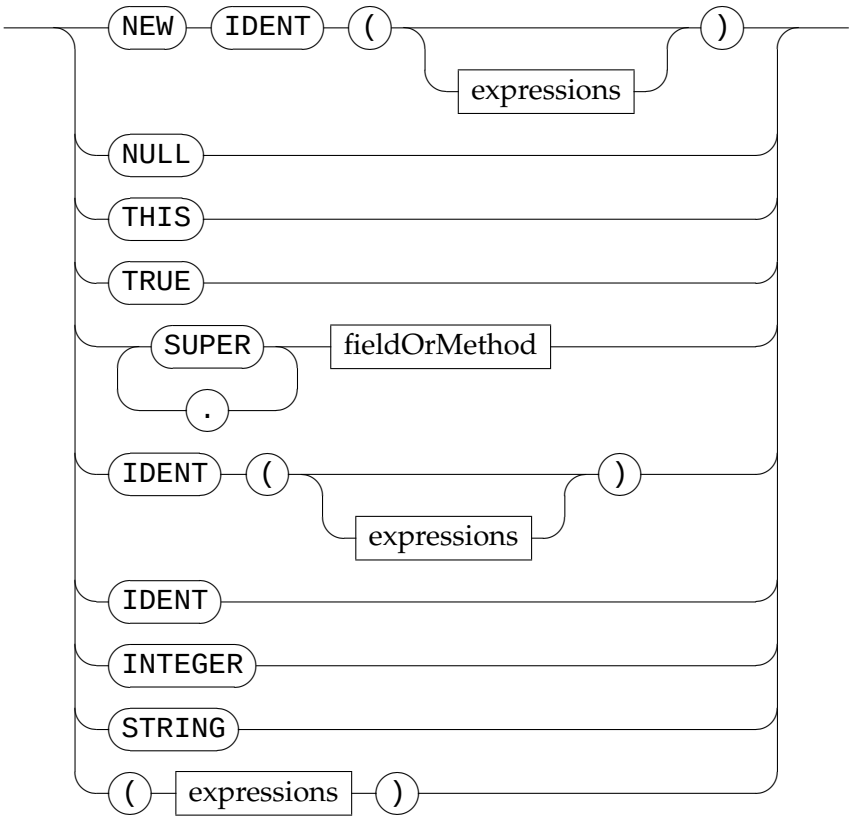
unaryExpr



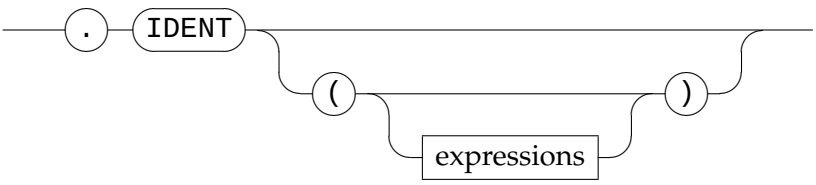
postfixExpr



primaryExpr



fieldOrMethod



Capítulo 4

API DEL LENGUAJE JAVA−−

4.1 Diagrama de clases

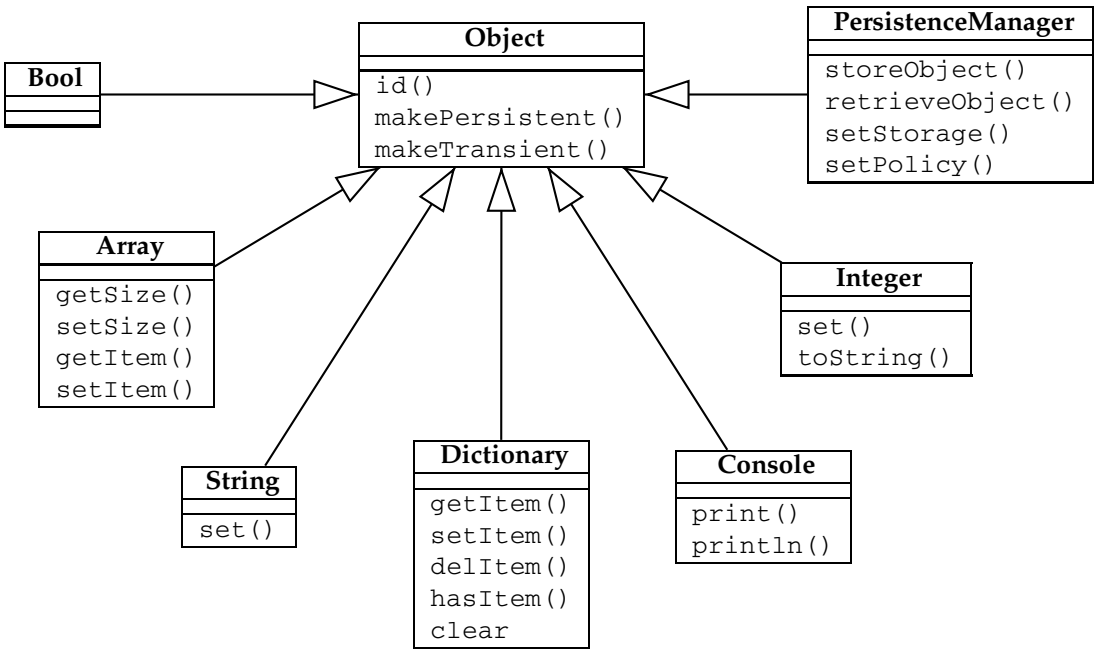


Figura 4.1: Diagrama de clases del API del lenguaje

En la figura 4.1 se muestra la jerarquía de las clases del API del lenguaje Java−−. Se puede observar en particular que al derivar todas las clases de Object, tanto las primitivas como todas las clases definidas por el usuario, automáticamente toda instancia se podrá hacer persistente sin más que invocar un método de la misma. Además, para los casos en que simplemente se quiere guardar un *snapshot* de un objeto, se ha creado una clase PersistenceManager que expone a los programas de usuario la funcionalidad necesaria para cargar y almacenar objetos de forma puntual.

4.2 Descripción de las clases

4.2.1 Clase Object

Clase base de todos los objetos del lenguaje Java−−.

4.2.1.1 Método id()

- **Descripción:** Devuelve el GUID del objeto.
- **Prototipo:** id()
- **Argumentos:** Ninguno.
- **Valor de retorno:** Una cadena con el GUID del objeto.

4.2.1.2 Método makePersistent()

- **Descripción:** Hace que el objeto y sus miembros sean persistentes.
- **Prototipo:** makePersistent()
- **Argumentos:** Ninguno.
- **Valor de retorno:** Ninguno.

4.2.1.3 Método makeTransient()

- **Descripción:** Hace que el objeto y sus miembros dejen de ser persistentes.
- **Prototipo:** makeTransient()
- **Argumentos:** Ninguno.
- **Valor de retorno:** Ninguno.

4.2.2 Clase Bool

Clase de la única instancia que se utiliza para representar el valor lógico *verdadero* en los resultados de todas las operaciones y métodos cuyo resultado sea de tipo booleano.

4.2.3 Clase Integer

Un entero.

4.2.3.1 Operaciones soportadas

- Con parámetro Integer: +, -, *, /, %, <, <=, >, >=, ==, !=

4.2.3.2 Método set()

- **Descripción:** Cambia el valor del entero.
- **Prototipo:** set(Integer newValue)
- **Argumentos:**

  newValue: Entero del que se copiará el nuevo valor.
- **Valor de retorno:** Ninguno.

4.2.3.3 Método toString()

- **Descripción:** Crea una representación textual del entero.
- **Prototipo:** toString()
- **Argumentos:** Ninguno.
- **Valor de retorno:** Una cadena con la representación textual del entero.

4.2.4 Clase String

Una cadena de texto.

4.2.4.1 Operaciones soportadas

- Con parámetro String: +

4.2.4.2 Método set()

- **Descripción:** Cambia el valor del entero.
- **Prototipo:** set(Integer newValue)
- **Argumentos:**

  newValue: Entero del que se copiará el nuevo valor.
- **Valor de retorno:** Ninguno.

4.2.5 Clase Console

Interfaz básico para escribir texto por la salida estándar (que por defecto se redirige a la ventana de la aplicación).

4.2.5.1 Método print()

- **Descripción:** Escribe una cadena por la salida estándar.
- **Prototipo:** print(String str)
- **Argumentos:**

  str: Cadena a escribir.
- **Valor de retorno:** Ninguno.

4.2.5.2 Método println()

- **Descripción:** Escribe una cadena por la salida estándar, añadiendo un salto de línea al final.
- **Prototipo:** println(String str)
- **Argumentos:**

  str: Cadena a escribir.
- **Valor de retorno:** Ninguno.

4.2.6 Clase Array

Vector de objetos.

4.2.6.1 Método getSize()

- **Descripción:** Devuelve el tamaño del array.
- **Prototipo:** getSize()
- **Argumentos:** Ninguno
- **Valor de retorno:** Un entero con el tamaño del array.

4.2.6.2 Método setSize()

- **Descripción:** Cambia el tamaño del array
- **Prototipo:** setSize(Integer newSize)
- **Argumentos:**

  newSize: Nuevo tamaño para el array.
- **Valor de retorno:** Ninguno.

4.2.6.3 Método getItem()

- **Descripción:** Devuelve uno de los objetos del array.
- **Prototipo:** getItem(Integer index)
- **Argumentos:**

  index: Posición del objeto dentro del array.
- **Valor de retorno:** El objeto pedido.

4.2.6.4 Método setItem()

- **Descripción:** Cambia uno de los objetos del array.
- **Prototipo:** setItem(Integer index, Object obj)
- **Argumentos:**

  index: Posición del objeto dentro del array.
  obj: El nuevo objeto.
- **Valor de retorno:** Ninguno.

4.2.7 Clase Dictionary

Un diccionario que mapea cadenas a objetos.

4.2.7.1 Método getItem()

- **Descripción:** Devuelve uno de los objetos del diccionario.
- **Prototipo:** getItem(String key)
- **Argumentos:**

  key: El nombre del objeto dentro del diccionario.
- **Valor de retorno:** El objeto pedido, o null si no existe un elemento con la clave dada.

4.2.7.2 Método setItem()

- **Descripción:** Inserta un objeto en el diccionario con un nombre dado, o lo sustituye si ya existía.
- **Prototipo:** setItem(String key, Object value)
- **Argumentos:**

  key: El nombre del objeto dentro del diccionario.
  value: El objeto a introducir.
- **Valor de retorno:** Ninguno.

4.2.7.3 Método delItem()

- **Descripción:** Elimina una entrada del diccionario.
- **Prototipo:** delItem(String key)
- **Argumentos:**

  key: El nombre del objeto dentro del diccionario.
- **Valor de retorno:** Ninguno.

4.2.7.4 Método hasItem()

- **Descripción:** Pregunta si el diccionario contiene algún objeto con un cierto nombre.
- **Prototipo:** hasItem(String key)
- **Argumentos:**

  key: El nombre del objeto dentro del diccionario.
- **Valor de retorno:** Un booleano indicando si el objeto está en el diccionario o no.

4.2.7.5 Método clear()

- **Descripción:** Vacía el diccionario.
- **Prototipo:** clear()
- **Argumentos:** Ninguno.
- **Valor de retorno:** Ninguno.

4.2.8 Clase PersistenceManager

Interfaz básico con el sistema de persistencia.

4.2.8.1 Método storeObject()

- **Descripción:** Guarda un objeto en el almacenamiento actual.
- **Prototipo:** storeObject(Object obj)
- **Argumentos:**

  obj: El objeto a guardar.
- **Valor de retorno:** Una cadena con el GUID del objeto almacenado.

4.2.8.2 Método retrieveObject()

- **Descripción:** Carga un objeto desde el almacenamiento actual.
- **Prototipo:** retrieveObject(String guid)
- **Argumentos:**

  guid: GUID del objeto a recuperar.
- **Valor de retorno:** El objeto cargado.

BIBLIOGRAFÍA

[1] A. V. Aho. *Compiladores: Principios, Técnicas y Herramientas*. Addison-Wesley Iberoamericana, 1990.

[2] Luca Cardelli. Type systems. En *Handbook of Computer Science and Engineering*, capítulo 103. CRC Press, 1997.

[3] Markus Straber Fritz Hohl, Joachin Baumann. Beyond Java Merging CORBA-based Mobile Agents and WWW. En *Joint W3C/OMG Workshop on Distributed Objects and Mobile Code*. Boston, Massachusetts (EE.UU.), junio 1996.

[4] Fredrik Lundh. An introduction to Tkinter. `http://www.pythonware.com/library/tkinter/introduction/`, enero 1999.

[5] Juan Manuel Cueva Lovelle. *Conceptos Básicos de Procesadores de Lenguaje*. Editorial Servitec, diciembre 1998.

[6] B. C. Smith. *Reflection and Semantics in a Procedural Language*. Tesis Doctoral, Massachusetts Institute of Technology, Cambridge (EE.UU.), 1982. MIT-LCS-TR-272.

[7] Guido van Rossum. Python reference manual (release 2.2.2). `http://www.python.org/doc/2.2.2/ref/ref.html`, octubre 2002.