Java: Un lenguaje de propósito general

Vicente Matellán Olivera José Centeno González
Pedro de las Heras Quirós Camino Fernández Llamas
Ignacio Aedo Cuevas Jesús M. González Barahona
Francisco J. Ballesteros Cámara
Departamento de Informática
Universidad Carlos III de Madrid
http://www.inf.uc3m.es

13 de octubre de 1998

Resumen

Java^{TM} se ha convertido en poco tiempo, gracias al empuje comercial de Sun Microsystems [1], en el paradigma de una nueva forma de diseñar, crear, distribuir y utilizar el software. En esencia Java es un lenguaje de programación orientado a objetos, interpretado, portable, con recolección de basura y soporte dentro del lenguaje para concurrencia, y que soporta la interacción entre objetos remotos.

Al amparo de estas ideas ha surgido todo un entorno de desarrollo y explotación de software, que Sun denomina el entorno Java (*Java environment*). Éste comprende el propio lenguaje Java, un sistema operativo (JavaOS), bibliotecas de clases, máquinas virtuales capaces de interpretar el código Java sobre distintas arquitecturas, etc. Además, Sun y otros proveedores han ido generando diferentes herramientas, API's, etc. que han dado lugar a un vocabulario propio con términos como JavaBeans, JDBC, AWT, applets, RMI, etc.

A lo largo del texto se encontrarán las definiciones de estos términos y la introducción a sus fundamentos, críticas y alabanzas a las ideas que se aportan, ejemplos de uso, referencias a la documentación y lugares en la red en los que se puede encontrar más información sobre los temas tratados. Se abordarán diversos aspectos de Java, desde la estructura básica del lenguaje, la historia de sus creadores y sus influencias,las bibliotecas gráficas, el acceso a bases de datos; hasta el soporte para realizar aplicaciones distribuidas o la concurrencia en Java.

Se ha estructurado este artículo en tres bloques principales y que pueden leerse por separado. En primer lugar, se describen los fundamentos del lenguaje, incluyendo el soporte gráfico; en segundo lugar, se presentan componentes más específicos como el acceso a bases de datos y los modelos de distribución y concurrencia y en tercer lugar, se analizan, desde una perspectiva crítica, algunos aspectos del diseño del lenguaje Java.

[1] Java es un producto de Sun Microsystems, en el resto del artículo se asumen los derechos de los legítimos propietarios de los productos y marcas comerciales que aparecen aunque no se indique explícitamente.

Índice General

Índice de Figuras

Índice de Tablas

1 Introducción

No está muy clara cual fue la idea que llevó al equipo de James Gosling a elegir este nombre para el lenguaje que estaban diseñando. Eso sí, después de comprobar que *Oak* (roble, en inglés), nombre elegido originalmente, ya estaba registrado para otro lenguaje de programación.

El nombre no es la única anécdota en el nacimiento de Java. Realmente desde la incorporación de James Gosling a Sun en 1984 hasta el anuncio oficial de Java el 23 de Mayo de 1995 en Sun World'95, toda una serie de acontecimientos no previstos llevaron al desarrollo de la que es la estrategia central de uno de los grandes de la informática.

En principio, el grupo de Gosling estaba dedicado a explorar las posibilidades de Sun en el mercado de la electrónica de consumo, un proyecto denominado *Green*. Este proyecto, originalmente basado en C++, tenía como objetivo crear una plataforma para el desarrollo de software independiente del procesador sobre el que se fuese a ejecutar, de forma que los fabricantes de electrodomésticos pudiesen desarrollar sus controladores independientemente del procesador que empleasen a la hora de fabricar el aparato, en general el más barato.

El proyecto comenzó extendiendo el compilador de C++, pero pronto se decidió que no era suficiente por lo que a mediados de 1991 se comenzó el diseño de *Oak*. En el otoño del 92 el equipo había terminado el sistema operativo del proyecto *Green* (GreenOS), la interfaz de usuario y había diseñado tres chips para realizar las pruebas, que permitían entre otras cosas ejecutar el código en un pequeño PDA (*Personal Digital Assistant*) denominado *7 (por la tecla del teléfono que había que pulsar para responder a las llamadas). Con ello se demostraban las principales cualidades perseguidas con el proyecto y que tan importantes han sido en el desarrollo de Java:

- Transportabilidad: capacidad para ejecutarse sobre *hardware* diferente.
- Reducido tamaño: capaz de ejecutarse en dispositivos reducidos como "*7".
- Eficiencia del código, a pesar de ser interpretado.
- Distribución: desde el *7 se podía controlar el resto de los dispositivos.

Mientras Sun decidía si la tecnología del proyecto Green era comercializable o no, una parte del grupo de ingenieros del proyecto *Green* pasó a una compañía denominada *FirstPerson Inc.*, cuyo objetivo era usar la tecnología desarrollada en el proyecto *Green* dentro del área de la televisión interactiva. Este intento no fructificó y en 1994 parte de los integrantes se incorporó a Sun Interactive. En ese momento el Web estaba explotando, por lo que la idea de realizar un navegador basado en Java parecía interesante. El resultado fue *WebRunner*, cuya demostración sí impresionó a los directivos de Sun, que vieron en él el comienzo de un nuevo mercado para herramientas, servidores, etc. Una página Web había dejado de ser una página de papel en versión electrónica para ser una forma genérica de proporcionar información de todo tipo, de forma global e independiente de la plataforma.

En Mayo de 1995 Sun anunció formalmente Java. La incorporación de esta tecnología en el navegador de Netscape Communications hizo que esta tecnología se difundiese rápidamente. A continuación la propia fuerza del emergente mercado ha ido exigiendo todo tipo de productos, desde acceso a bases de datos (JDBC), servidores web (JavaServer), integración en electrodomésticos y dispositivos diversos de uso cotidiano (EmbeddedJava, JavaCard, etc.),

ayudas para desarrollo de aplicaciones corporativas (JavaBeans, Java Transactions, JavaMail, JavaHelp, JavaPC), etc.

Aparte de convertirse en el núcleo del negocio del software para la compañía Sun Microsystems, Java constituye una nueva forma de entender las aplicaciones informáticas, su distribución y la interrelación entre ellas. Otros proveedores de software intentan aprovechar esta ola con sus propios productos. Así Microsoft intenta convertir Visual Basic en su producto para generar aplicaciones que interaccionen con la red; General Magic impulsa su *Telescript*, otro lenguaje interpretado, etc.

Este artículo está formado por nueve apartados en los que se trata de introducir la mayor parte de la tecnología Java. Estos apartados se agrupan en bloques que pueden leerse de forma independiente, si el lector está interesado sólo en alguna parte en concreto.

En un primer bloque, formado por los tres primeros apartados, se presentan algunos de los componentes de la familia Java. En primer lugar se realiza un breve análisis del lenguaje en si mismo, haciendo especial hincapié en sus particularidades, como los mecanismos de herencia, la gestión de excepciones, el tratamiento de memoria dinámica, etc. A continuación se presentarán los programas descargables a través de la red y ejecutables en un navegador, los programas conocidos como *applets*. Para finalizar este primer bloque se analizarán brevemente las características básicas de la biblioteca que proporciona Java para la construcción de interfaces de usuario (denominada AWT *Abstract Window Toolkit*).

En un segundo bloque, formado por los siguientes cuatro apartados, se analizan otro tipo de bibliotecas y partes del lenguaje que se suelen tratarse de forma independiente en la literatura sobre Java. En primer lugar se presenta el acceso a servidores SQL mediante la biblioteca JDBC. Seguidamente, se analiza el modelo de concurrencia que proporciona Java, basado en *Threads*. A continuación el de programación distribuida, basado en su propia versión de la llamada a procedimientos remotos RMI *Remote Method Invocation*. Por último, se presenta el mecanismo de programación visual con componentes reutilizables denominado JavaBeans.

El tercer bloque, el apartado 9, se dedica a comentar algunos aspectos de la tecnología que presentan algunos problemas desde el punto de vista del diseño de los lenguajes de programación, de la legibilidad y mantenibilidad del código; así como algunas consideraciones sobre su uso en docencia.

2 El lenguaje Java

Como ya se ha descrito, el origen del lenguaje Java fue el proyecto *Green*, basado en C++, lo que explica que su sintaxis sea muy parecida a la de C++ y por tanto a la de C. Probablemente no sea la sintaxis más adecuada para un lenguaje de programación moderno ya que en algunos casos es oscura, puede provocar errores si no se utiliza correctamente. Sin embargo, es una de las más conocidas.

En este capítulo se realiza un breve recorrido por la misma, de forma que cualquier lector con conocimientos de programación pueda leer y entender los ejemplos de sucesivos capítulos, además de proporcionarle una introducción básica al lenguaje.

2.1 Sintaxis de Java

C es un lenguaje imperativo, pensado para que los programadores pudieran acceder a la información a nivel de posiciones de memoria. Su estructura imperativa se mantuvo en C++, así como la capacidad para el manejo directo de la memoria. En C++ se incluyó la orientación a objetos, aunque se preservó toda la compatibilidad con C, es decir, cualquier programa en C puede en principio compilarse sin problemas en C++.

En Java sin embargo, aunque la sintaxis es muy similar, la semántica ha cambiado en muchos casos. De hecho, aunque comparten la mayor parte de la sintaxis básica, el lenguaje ha cambiado completamente, es un lenguaje nuevo con una sintaxis antigua. La orientación a objetos es ahora mucho más estricta. De hecho, no se puede escribir ningún programa que no incluya la definición de al menos una clase. Como homenaje a sus orígenes incluimos la versión Java del tradicional "Hola Mundo", el primer programa que suele escribirse en cualquier lenguaje.

```
// Fichero HolaMundo.java
import java.io.*;

public class HolaMundo {
       public static void main (String args[]) {
              System.out.println (``Hola Mundo'');
       }
}
```

Este primer programa se puede usar para analizar algunas de las características básicas del lenguaje. Así, en primer lugar aparece un comentario de línea (es decir, código que no se interpreta) al estilo C++. Se ha utilizado la línea para indicar cual es el nombre con el que deberemos almacenar el programa, en esta caso `HolaMundo.java`, siendo `.java` la extensión estándar de los ficheros que contienen código escrito en Java. Para finalizar los aspectos relativos a los comentarios, también se dispone en Java de comentarios de bloque con la misma sintaxis de C++: comienza el comentario con `/*` y termina con `*/`, no siendo posible anidarlos.

Java es un lenguaje interpretado, lo que siguiendo el sentido tradicional de "lenguaje interpretado" querría decir que nuestro fichero `.java` no se traduciría a instrucciones entendibles por el ordenador y después se ejecutaría dicha traducción, sino que sería otro programa, denominado "intérprete", el encargado de traducir y ejecutar instrucción a instrucción nuestro programa. Ese programa en el caso de Java es lo que se denomina la Máquina Virtual de Java

(*Java Virtual Machine* o JVM), pues su propósito lo que hace es convertir la plataforma en la que se ejecute en otra "virtual" capaz de interpretar lenguaje Java.

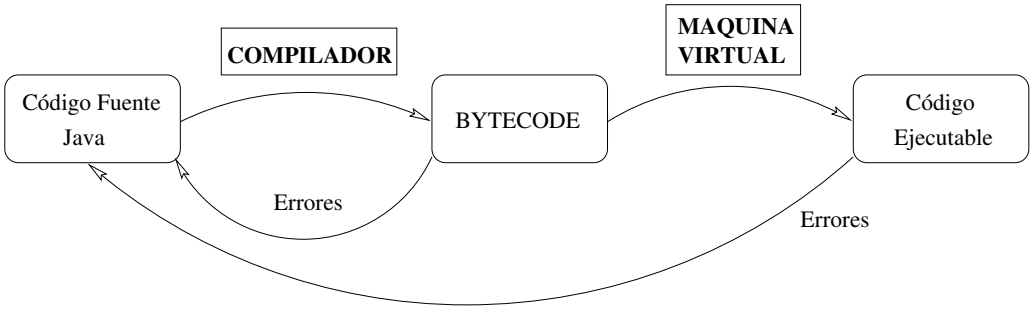

Figura 1: Ciclo de generación de código en Java

En el caso de Java se utiliza además un paso intermedio antes de la interpretación: el fuente Java se compila, generándose un código de bytes (*bytecode*). Con este paso se consigue acelerar la interpretación del código, facilitar la portabilidad, depurar errores sintácticos sin necesidad de ejecutar el programa, etc. El fichero intermedio obtenido de esta compilación sería para el ejemplo anterior `HolaMundo.class`. Es decir, la extensión del fichero en el que se almacena el *bytecode* es `.class`. Con ello se intenta reflejar que este fichero compilado corresponde a una clase, es más, si el fichero `HolaMundo.java` contuviese más de una clase se generaría un fichero `.class` para cada una de ellas. La Figura 1 representa el proceso descrito para generar código ejecutable en una determinada arquitectura.

Esta organización basada en el concepto de clase se extiende también a la gestión de las bibliotecas del lenguaje. Siguiendo con el análisis del programa anterior, la siguiente línea es `import java.io.*`, similar a un `#include` de C++. Sin embargo, su significado es diferente. En C++ al colocar una directiva de precompilación de ese tipo lo que se hace es "incluir" directamente el código contenido en el fichero, el cual se compila junto con el programa. Es decir, se incluye todo el código del fichero, se use o no. En Java la instrucción `import` lo único que indica es dónde debe buscar la máquina virtual aquellas clases que se soliciten (buscando a partir del directorio indicado en la variable de entorno `CLASSPATH`) y no se encuentren en el fichero. Dichas clases no se incluyen en el fichero `.class`, sólo contiene el *bytecode* del fichero fuente. Por ello, el tamaño del fichero compilado `HolaMundo.class` es muy pequeño.

¿Cómo hace la máquina virtual para ejecutar el programa si no todo el código necesario está incluido? Pues definiendo un conjunto de bibliotecas como básicas o estándar. Así, se supone que las bibliotecas estándar (y `java.io` lo es) están en todas las instalaciones Java. De esta forma, la máquina virtual sabrá encontrar el método `println`.

Para buscarlo se hace uso de otra de las características de las bibliotecas Java: su jerarquización. Al indicar que se quieren importar las clases de `java.io.*` lo que se debe leer es que se importan todas las clases (`*`) del subdirectorio `io` del directorio donde están las clases estándar de Java (`java`). De igual forma, cuando el programador desee crear sus propias bibliotecas de clases deberá seguir una estructura similar de bibliotecas. En Java se declaran mediante `package`, por lo que se suele hacer referencia a ellas también por su traducción: paquete.

Las instrucciones de un programa Java pueden ser, como en la mayoría de lenguajes, de cinco tipos:

1. Asignación: modificación del contenido de una variable usando un valor del mismo tipo (Java es un lenguaje fuertemente tipado). Su sintaxis, igual que en C++, utiliza el operador =.

2. Invocación: ejecución de un método. Su sintaxis es el nombre del método que se está invocando, precedido si es el caso, por el nombre del objeto al que pertenece y seguido por los parámetros entre paréntesis: `objeto.método(parámetros)`. En el caso de un método de tipo `static` (solo existe una copia del método aunque existan varios objetos de la misma clase) declarado en otra clase, la invocación del método va precedida del nombre de la clase y no del objeto.

   En cuanto a los parámetros, el número, tipo y orden definido en la declaración del método tiene que coincidir con el de la llamada. Además, en Java todos los parámetros se pasan **por copia**, situación que debe considerarse cuidadosamente, como se comenta en el apartado 9.1.

3. Repetición: indica que una instrucción o grupo de instrucciones se realizará un número de veces. En Java se dispone de las siguientes construcciones para realizar bucles:

   - `for (comienzo; comprobación; actualización) { instrucciones };`
   - `while (condición) { instrucciones };`
   - `do {instrucciones} while (condición);`

4. Selección: decide si un determinado grupo de instrucciones se va a ejecutar o no. En Java existen dos instrucciones básicas de selección:

   - `if (condición) { instrucciones } else { instrucciones };`
   - ```
     switch (expresión) {
             case valor: { instrucciones
               break;}
             case valor: { instrucciones
               break;}
       }
     ```

5. Excepción: sirve para detectar y reaccionar ante situaciones extrañas o erróneas. En Java se usa la instrucción `try` para indicar una zona donde ciertas operaciones pueden causar una excepción. `catch`, a continuación de una instrucción `try`, indica uno o más manejadores para las excepciones que puedan producirse en el `try`. Por último, `throw` sirve para propagar excepciones hacia otros métodos.

2.2 Declaración de clases y creación de objetos

Java es un lenguaje orientado a objetos, lo que se refleja en que la unidad básica de programación es la clase. Es decir, el menor de los programas obligatoriamente incluye la declaración de una clase. Siguiendo con las líneas del programa `HolaMundo.java`, el código propiamente dicho comienza con la declaración de la clase `HolaMundo`.
     ```

La declaración especifica que esta clase es de tipo `public`. Éste es un modificador de clase que indica que se puede instanciar un objeto de esta clase en cualquier lugar, incluso fuera del paquete en el que está declarada: es la visibilidad por defecto. En particular, la JVM creará un objeto de la clase `HolaMundo` cuando se le pida ejecutar este programa.

La forma de ejecutar programas Java consiste en invocar al interprete de Java (la JVM) [2] seguido del nombre de la clase precompilada (el `.class`): `java HolaMundo`. La JVM creará un objeto de dicha clase e invocará al método `main` que obligatoriamente debe existir (excepto para el caso de los *applets*, ver sección 3.1).

En este primer ejemplo la clase `HolaMundo` está compuesta por un único método, `main`. Al igual que las clases, los métodos también pueden estar precedidos por una serie de calificadores. En este caso, el método es `public`, accesible desde fuera de la clase, puesto que será invocado por la JVM; `void` para indicar que este método no devuelve ningún valor y `static` que implica que sólo existirá una copia de dicho método.

En general, un método declarado en Java puede no devolver nada, que se indica con `void`, puede devolver un valor de un tipo básico o una referencia a un objeto de una clase determinada.

Los tipos básicos en Java son:

Numéricos: `byte` 8 bits `short` 16 bits para enteros con signo, `int` enteros de 32 bits, `long` enteros de 64 bits, `float` coma flotante de 32 bits y `double` coma flotante de 64 bits.

Lógicos: con valores posibles `true` (verdadero) y `false` (falso). Éste es el tipo que devuelven las comparaciones lógicas (<, >=, etc.) y admite además los operadores lógicos típicos: `&` para la conjunción (AND), `|` para la disyunción (OR), `!` para la negación (NO) y `^` para la disyunción exclusiva (XOR).

Caracteres: mediante el tipo `char` de 16 *bits* se representan las letras de la "a" a la "z" y de la "A" a la "Z", los símbolos (!, #, ?, etc.) y los dígitos. Además, proporciona algunas secuencias de escape para tabulador, nueva línea, etc.; así como para el conjunto de caracteres *Unicode*.

El resto de los tipos se implementan en Java mediante clases, como por ejemplo las cadenas de caracteres que serán objetos `String`.

Ya se ha presentado cual es la sintaxis para declarar una clase. Se analiza a continuación con algo de más de detalle el modelo de orientación a objetos implementado en Java.

En primer lugar, la forma de crear una referencia a un objeto es: `nombreclase objeto;` . Una vez creada la referencia se le puede asignar un objeto con `new` de la siguiente forma: `nombreclase objeto = new nombreclase (parámetros);`

Donde la accesibilidad a los objetos creados vendrá dada por los modificadores que aparezcan en la declaración de la clase correspondiente. Existen modificadores a nivel de la declaración de la clase y también a nivel de atributos y métodos. En la Tabla 1 aparecen resumidas las posibilidades que Java ofrece en este aspecto.

Algunos ejemplos de creación de objetos son:

```
1) PrintStream ficheroSalida = new PrintStream (new FileOutputStream ("p.out"));

2) private static Random cualquiera;
```

[2] O arrastrando el icono correspondiente sobre el de la JVM en el caso de los sistemas operativos gráficos, como el de los MacOS

Modificador	Visibilidad y uso
sin modificador	Objeto, atributo o método visible en el paquete
`public`	Objeto visible donde lo sea la clase
`private`	Atributo o método visible solo en su propia clase
`protected`	Atributo o método visible en su paquete y para sus clases descendientes en otros paquetes
`final`	Atributo no modificable o método no redefinible
`static`	Atributo o método de clase (único para todas sus instancias)
`abstract`	Método que no incluye código o clase que contiene métodos abstractos

Tabla 1: Modificadores aplicables

```
   cualquiera = new Random (Text.readInt(in));

3) throw new TargetNotFoundException ( );
   try {...}
   catch (TargetNotFoundException e) {...};
```

El primer ejemplo crea un objeto anónimo `FileOutputStream` que se pasa como parámetro al constructor [3] de la clase `PrintStream` para crear a su vez el objeto "referenciado" por `ficheroSalida`.

Las objetos, para los que no conozcan los principios de la orientación a objetos, son muy distintos de las variables. Una variable almacena un único valor, que tiene a su vez definido un conjunto de operaciones predefinidas en el lenguaje que se pueden realizar sobre él dependiendo de su tipo. Un objeto, sin embargo, puede estar compuesto por distintos valores, sobre los que se pueden realizar un conjunto de operaciones definidas por el usuario en la especificación de la clase de la que es instancia el objeto.

En Java los objetos se manejan mediante **referencias** al grupo de valores que representan. Una referencia es nombre (como el de una variable) que contiene una indicación del lugar donde está realmente almacenado el objeto (el mismo concepto de los punteros en C++). Los tipos básicos o primitivos (enteros, caracteres, etc.) se manejan mediante variables, las manipulan el valor real.

Cuando un objeto deja de estar "referenciado" por algún nombre, la máquina virtual se encarga de destruirlo. Este mecanismo se conoce como *recolección automática de basura* (*garbage collection*), simplificando la gestión de memoria dinámica.

El segundo ejemplo de creación de objetos muestra cómo se puede crear un nombre (`cualquiera`) para un objeto (sin crear ningún objeto) para posteriormente asignarle un objeto que se crea con `new`. Además, se especifica en esa misma declaración del nombre la visibilidad del mismo.

En el tercer ejemplo, se genera un objeto excepción cada vez que se lanza una excepción. Dicho objeto es capturado mediante un `catch` que le asigna el nombre `e` para su posterior tratamiento.

[3] Un constructor es un método, que tiene el mismo nombre que la clase y que se invoca de forma automática al crearse un objeto de dicha clase. Su misión es realizar aquellas operaciones de inicialización que sean necesarias.

2.3 Orientación a objetos

La orientación a objetos es un mecanismo de programación muy potente gracias a una serie de operaciones que se pueden realizar con las clases. Algunas ya se han analizado, como la **encapsulación** (integrar métodos y datos dentro de las clases para ofrecer una interfaz simplificada a otras clases) o la **composición** que permite crear un objeto en base a otro (como se ha visto en la creación anterior de objetos); y otras, como la **abstracción** y la **herencia**, que son aún más potentes, se describen a continuación.

Abstracción

La abstracción se utiliza para que una clase se concentre en lo que es esencial, es decir, su comportamiento e interfaz con el mundo. Por ejemplo, imagínese clases distintas que encapsulan información relativa a automóviles, personas y animales. Si se quisiera realizar una operación, por ejemplo de ordenación, sobre todas ellas, nos interesaría poder "abstraer" los diferentes métodos de ordenación para esas clases.

En Java la abstracción se consigue mediante la declaración de clases abstractas. Una clase abstracta es aquella que incluye al menos un método abstracto, es decir, un método del que sólo ofrece su cabecera pero no su implementación. No se permite instanciar clases abstractas pero sí heredarlas y completar el código de sus métodos abstractos para su posterior utilización. Además de la posibilidad de utilizar el modificador `abstract`, se dispone también de las `interfaces`. Una interfaz es una clase cuyos métodos son todos abstractos y cuyos atributos (si los tiene) son todos `final`, aunque no sea necesario que aparezcan precedidos de `abstract` ni de `final` al encontrarse dentro de una `interface`. Una interfaz se declara y se utiliza de la siguiente forma:

```
interface nombreinterfaz {
     especificación de métodos
}

class nombreclase  implements nombreinterfaz {
     cuerpo de los métodos del interfaz y los propios
}
```

Herencia

En Java una clase puede heredar de otra utilizando la siguiente sintaxis:

```
class ClaseHija  extends ClaseMadre {
     ...
}
```

Mediante esta declaración `ClaseHija` tendrá acceso a los atributos y métodos de `ClaseMadre` que ésta le proporcione (ver Tabla 1). En Java no se permite la herencia múltiple, de modo que una clase sólo puede heredar de otra. Así se evitan los problemas que ese tipo de herencia podría acarrear, aunque se limitan las posibilidades a la hora de diseñar aplicaciones. Pero gracias a los interfaces, esta limitación no es tal, ya que, si bien una clase sólo puede heredar de otra, puede además implementar varios interfaces.

Con estas breves pinceladas sobre el lenguaje cualquier programador, aunque no esté familiarizado con Java, debería ser capaz de leer y entender los programas de ejemplo de los siguientes apartados.

A continuación se abordan las particularidades de Java tanto para transportar su código a través de la red Internet como para escribir programas que hagan uso de los servicios de la misma.

3 Java e Internet

Como ya se ha comentado anteriormente, entre las principales razones por las que se creó Java destacan la necesidad de desarrollar aplicaciones que se pudiesen ejecutar en cualquier plataforma y el potencial valor del empleo de recursos de red. Mientras que la ejecución en cualquier plataforma supone que cualquier aplicación programada con este lenguaje pueda ser ejecutada cuando se invoque desde un navegador Web; la utilización de los recursos de red implica la distribución física de los recursos necesarios para su ejecución. Estas dos razones justifican que la definición de Java incluya clases estándar orientadas a facilitar el trabajo de los desarrolladores en aplicaciones que hagan uso de la red Internet y, por lo tanto, que se puedan ejecutar independientemente de donde se encuentre el usuario y de los recursos que necesiten. Estas clases se encuentran en dos paquetes denominados `java.applet`, que se utiliza para el desarrollo de aplicaciones denominadas *applets* que se incluyen en páginas HTML, y `java.net`, que se usa para el desarrollo de aplicaciones que tengan que realizar transmisiones de datos.

3.1 Applets

Un *applet* es una aplicación que permite aumentar la potencia que proporcionan las páginas HTML pudiendo, por ejemplo, acceder a la información de una base de datos, representar las ventas de una compañía o controlar un reactor químico.

Para ejecutar estas aplicaciones se debe utilizar un navegador que sea capaz de interpretar el *Java Byte Code*, como pueden ser las versiones que en la actualidad ofrecen las compañías Netscape o Microsoft. Otra forma de ejecutar *applets* es utilizando la aplicación `appletviewer` incluida en el JDK.

Para desarrollar sus propios *applets* los programadores disponen de la clase predefinida `Applet`, contenida en el paquete `java.applet`, heredando los métodos definidos en ella. En la siguiente definición de clase se puede ver cómo se utiliza la palabra reservada `extends` para indicar que el *applet* definido en la clase `MiEjemplo` es una especialización de la clase `Applet`:

```
import java.applet.Applet; /* Utiliza la clase Applet */

public class MiEjemplo extends Applet
   {/* Entre las llaves se incluyen los métodos y campos del applet MiEjemplo */}
```

El *applet* debe incluir al menos uno de los métodos `init`, `start` o `paint`, heredados de la clase `Applet`, para que se inicien las acciones que se quieren llevar a cabo. `init` es el primer método que se invoca de forma automática cuando el *applet* empieza su ejecución, utilizándose para dar un valor inicial a las variables y referencias incluidas en él. Después de este método, se invoca al método `start` que incluye las acciones que se tengan que realizar cada vez que se accede a la página HTML. Mientras el método `init` sólo se ejecuta una vez cuando se carga en memoria el *applet*, `start` se invoca cada vez que se accede a la página. Por último, `paint` sirve para representar gráficamente el *applet* después de que el método `init` haya finalizado y haya comenzado `start`. Este método también se invoca cuando se quiere redibujar el *applet*, por ejemplo, cuando la información cambia su colocación en el *applet*. La clase `Applet` contiene además otros dos métodos `stop` y `destroy` que se invocan automáticamente cuando el *applet* termina su ejecución y cuando se debe eliminar de memoria para liberar los recursos utilizados, respectivamente.

En el siguiente código se define un *applet* que cuenta el número de veces que ha empezado (método `start`) y las que ha parado (método `stop`) su ejecución. Además de la clase `Applet`, hace uso del paquete `awt` que contiene las clases necesarias para poder representar los elementos del sistema gráfico de interfaz de Java y que se describirá en profundidad en la siguiente sección.

```
import java.applet.Applet;
import java.awt.*;

public class MiEjemplo extends Applet {
        String cadenaStart, cadenaStop;
        int contStart, contStop;

        public void init() {
              cadenaStart= new String ("Ha empezado: ");
              cadenaStop= new String ("Ha finalizado: ");
              contStart =0;
              contStop =0;
        }
        public void start() {
              contStart = contStart + 1;
              repaint();
        }

        public void paint(Graphics g) {
              g.drawString (cadenaStart + contStart, 10, 10);
              g.drawString (cadenaStop + contStop, 10, 20);
        }

        public void stop() {
              contStop = contStop+1;
        }

        public void destroy() {
              cadenaStart.finalize();
              cadenaStop.finalize();
        }
}
```

3.2 Applets en páginas Web

Para poder ejecutar el *applet* se debe construir una página HTML que contenga la etiqueta *applet*, y que a su vez contenga los atributos `code`, `width` y `height`. Estos atributos sirven respectivamente para indicar el nombre del fichero que contiene la clase a ejecutar y la longitud y la altura que va a ocupar el *applet* en el navegador, ya que su interfaz se representa en un área rectangular.

Además de estos atributos, también existen otros opcionales: `codebase` (ubicación relativa del *applet*), `alt` (texto alternativo que se mostrará en los navegadores que sólo pueden visualizar texto), `name` (nombre simbólico del *applet*), `align` (tipo de alineamiento dentro de la página

que se quiere que cumpla el *applet*), `vspace` (espacio vertical alrededor del *applet* si el atributo `align` tiene el valor `right` o `left`) y `hspace` (espacio horizontal alrededor del *applet* si el atributo `align` tiene el valor `right` o `left`).

Es posible utilizar parámetros para pasar diversos valores al *applet* desde el fichero HTML de manera que se pueda modificar su comportamiento. Para ello se incluirá en la definición del *applet* del fichero HTML la etiqueta `param` y sus dos atributos `name` y `value`, que indicarán el nombre y el valor del parámetro, respectivamente. Si en el *applet* se quiere conocer el valor asignado en el fichero HTML se invocará a la función `getParameter(nombreParametro)` que devolverá el valor del nombre del parámetro que coincida con `nombreParametro`.

Cuando sea cargado por un navegador, el siguiente fragmento de fichero HTML ejecutará el *applet* llamado `nombreApplet` que está en el fichero `ficheroClase` en la dirección `http://www.servidor.net` reservándose un área de 200x200 para su representación gráfica. El *applet* estará alineado a la izquierda con una separación vertical de 7 pixels y horizontal de 5. Si se utiliza un navegador no gráfico se visualiza el texto Texto alternativo. Por último, se le pasan dos parámetros al *applet* llamados `parametro1` y `parametro2` que tienen los valores 2 y `http://www.servidor.net`, respectivamente.

```
<applet code="ficheroClase" width=200
        heigh=200 codebase="http://www.servidor.net"
        alt="Texto alternativo" name="nombreApplet"
        align=left vspace=7 hspace=5>
        <param name="parametro1" value="2">
        <param name="parametro2" value="http://www.servidor.net">
</applet>
```

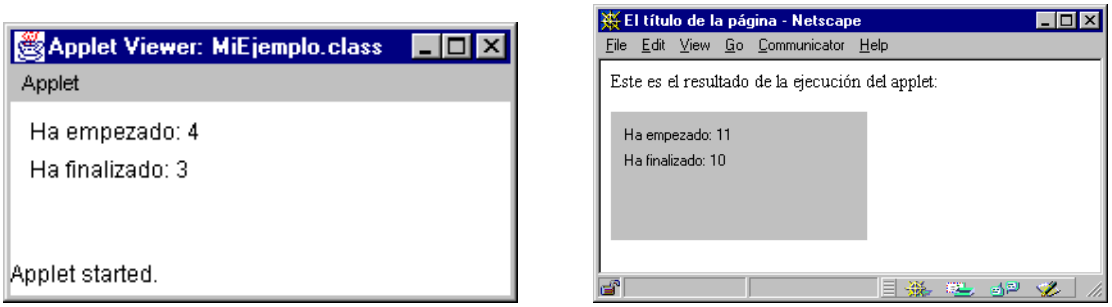

Figura 2: Ejecución de un *applet* en el *appletviewer* y un navegador

Si se quisiera ejecutar el *applet* `MiEjemplo` definido anteriormente, se deberá compilar en primer lugar para crear el fichero `MiEjemplo.class` que contendrá el *bytecode* de la clase `MiEjemplo`. A continuación, se deberá crear un fichero HTML que contenga la etiqueta *applet* y como valor del parámetro `code` la clase `MiEjemplo`. Por último, se leerá el fichero HTML con un navegador compatible con Java o con la aplicación *appletviewer* (que sólo muestra la ejecución del *applet* y no el resto del contenido del fichero HTML). En la Figura 2 se muestra el resultado final utilizando *appletviewer* y un navegador (Netscape) después de leer el siguiente fichero HTML.

```
<HTML>
```

```
   <HEAD><TITLE>El título de la página</TITLE></HEAD>
   <BODY>
     Este es el resultado de la ejecuci&oacute;n del applet:<P>
     <applet code="MiEjemplo.class" width=200 height=100>
     </applet>
</BODY>
</HTML>
```

A los *applets* se los considera seguros porque no pueden dañar al ordenador del cliente que lo recupera. Por ejemplo, un *applet* no puede escribir sobre el disco duro del ordenador donde se ejecuta. Antes de la ejecución de un *applet* se verifica su *bytecode*, comprobándose que las instrucciones que se van a ejecutar tienen permiso para hacerlo, eliminando así posibles riesgos.

3.3 Sockets

Como ya se ha dicho, el otro paquete del que disponen los programadores para realizar sus programas para Internet es `java.net`, que contiene un conjunto de clases (ej. la clase `URL` y la clase `InetAddress`) que le permiten el desarrollo de aplicaciones cliente-servidor de forma muy sencilla utilizando *sockets* para la transmisión de datos. Esta transmisión se puede hacer bien mediante la creación de una conexión entre el cliente y el servidor por la que fluyan los datos, o bien mediante la emisión/recepción de paquetes de información a través de la red.

Para ilustrar como se desarrollaría una aplicación orientada a conexión que haga uso de este paquete, se presenta a continuación el código fuente de la clase `MiServidor` desarrollado para escribir las direcciones IP que se le envían y el de la clase `MiCliente` que manda la dirección IP de la máquina donde se está ejecutando.

```
1  import java.net.*;
2  import java.io.*;
3
4  public class MiServidor
5  {
6       public static void main (String argumentos[]) {
7            MiServidor s= new MiServidor();
8            s.ejecutarServidor();
9       }
10
11      public void ejecutarServidor () {
12           ServerSocket nombreServidor;
13           Socket nombreSocket;
14           InputStream datosEntrada;
15           char car;
16           StringBuffer cadena = new StringBuffer(0);
17           try {
18                nombreServidor= new ServerSocket (8000, 100);
19                nombreSocket= nombreServidor.accept();
20                datosEntrada= nombreSocket.getInputStream();
21                cadena= new StringBuffer("");
22                while ((car= (char) datosEntrada.read())!= '#') {
```

```
23                      cadena.append(car);
24                  }
25                  System.out.println ("La cadena es: "+cadena);
26                  cadena=null;
27                  nombreSocket.close();
28                  nombreServidor.close();
29              }
30              catch (IOException e) {
31                  e.printStackTrace();
32              }
33      }
34 }
```

En primer lugar, en la clase `MiServidor` se invoca al método encargado de la ejecución del servidor. Esta ejecución consiste en crear un *socket* que espera alguna conexión (línea 18) en el puerto 8000 pudiendo admitir un máximo de 100. Después de aceptar a un cliente (línea 19) la zona de entrada de datos (línea 20), lee la dirección IP del cliente (bucle `while`, donde el carácter # se utiliza como final de la cadena), la escribe (línea 25) y, por último, cierra el *socket* (línea 27) y el servidor (línea 28). La sentencias `try - catch` manejan las excepciones derivadas de algún problema con la entrada de datos.

Además de `java.net`, el ejemplo siguiente hace uso del paquete `java.io` que contiene los métodos necesarios para la entrada/salida de datos y que se utiliza para la lectura/escritura del *socket*.

La clase `MiCliente` es un *applet* que se encarga de enviar la dirección IP del ordenador en el que se está ejecutando (línea 24) un *socket* entre él y el ordenador donde se encuentra el servidor (línea 27) para lo cual se pasa la dirección IP de éste y el puerto por el que se va a realizar la conexión. Después de preparar el área de memoria donde se pondrán los datos (línea 28), se escriben (líneas 29-31) y, por último, se cierra el cliente (línea 32).

```
1  import java.applet.Applet;
2  import java.awt.*;
3  import java.net.*;
4  import java.io.*;
5
6  public class MiCliente extends Applet; {
7        private String resultado;
8
9        public void init () {
10             resultado= new String("");
11             ejecutarCliente();
12       }
13
14       public void paint (Graphics g) {
15             g.drawString(resultado, 10,10);
16       }
17
18       private void ejecutarCliente () {
19             Socket cliente;
```

```
20              OutputStream datos;
21              int i;
22
23              try {
24                   final InetAddress miIP= InetAddress.getLocalHost();
25                   StringBuffer cadena= new StringBuffer(0);
26                   cadena.append(miIP + "#");
27                   cliente = new Socket(miIP, 8000);
28                   datos= cliente.getOutputStream();
29                   for (i=0; i<cadena.length(); i++) {
30                       datos.write ((int) cadena.charAt(i));
31                  }
32                   cliente.close();
33                   resultado=null;
34                   resultado= new String("Enviada dirección IP: " + cadena);
35                   repaint();
36              }
37              catch (UnknownHostException e1) {
38                   e1.printStackTrace();
39              }
40              catch (IOException e2) {
41                   e2.printStackTrace();
42              }
43       }
44 }
```

Casi todos los programas de ejemplo vistos hasta ahora hacen uso de las capacidades de gráficas de Java. Por ejemplo, en este último se utiliza el método `drawString` para escribir en la pantalla del navegador en el que se use el programa. El siguiente apartado analiza el modelo diseñado en Java para la generación de interfaces gráficos.

4 Interfaces de usuario

El desarrollo de cualquier tipo de programa implica en general el desarrollo de una o varias interfaces de usuario. La extensión de las interfaces gráficas a la práctica totalidad de aplicaciones y la enorme variedad de *hardware* gráfico (tarjetas de vídeo, monitores, etc.) han supuesto en general un freno a la portabilidad de las aplicaciones entre distintas plataformas.

Este problema se ha resuelto en Java proporcionando una biblioteca independiente de la arquitectura de ejecución denominada AWT (*Abstract Window Toolkit*) incluida en el núcleo básico del JDK.

Esta sección está dedicada a presentar brevemente la historia de esta biblioteca, describir sus componentes básicos y mostrar un ejemplo de su uso.

4.1 La estructura de AWT

El paquete AWT proporciona los componentes básicos para el desarrollo de aplicaciones gráficas, como botones, menús, listas, cajas de texto, etc. En su primera versión esto, que no es poco, era todo. Con ello se podía ejecutar el mismo programa, sin necesidad de recompilarlo, en un sistema operativo MacOS, en Windows o en una estación Unix con X-Windows.

Con la aparición de la versión 1.1 del JDK se añadieron nuevas funcionalidades. Por ejemplo, un mecanismo de tratamiento de eventos, la transferencia de información gráfica entre aplicaciones, soporte para el desarrollo de componentes "ligeros" que consumen menos recursos, etc. El inconveniente fue que las modificaciones necesarias para añadir estas funcionalidades supusieron un cambio radical en el modelo de programación de las aplicaciones gráficas, haciendo incompatibles las aplicaciones desarrolladas con JDK 1.0 con las del JDK 1.1.

La nueva versión del AWT está formada por seis paquetes, como muestra la Tabla 2

Paquete	Contenido
`java.awt`	Funcionalidad básica
`java.test`	Applet de prueba
`java.awt.datatransfer`	Soporte para la transferencia de información
`java.awt.event`	Gestión de eventos
`java.awt.peer`	Componentes nativos
`java.awt.image`	Manipulación de imágenes

Tabla 2: Paquetes AWT

El funcionamiento de AWT se basa en delegar gran parte de su funcionalidad en componentes nativos (*peers*). Las clases de AWT simplemente "empaquetan" las clases nativas. Esta decisión de diseño facilitó el rápido desarrollo de las interfaces gráficas de Java. A cambio, el modelo resulta difícil de trazar y en algunos casos de entender.

En los siguientes apartados se van a analizar dos programas para ilustrar el funcionamiento del modelo del AWT. El primero presenta la versión gráfica del "Hola Mundo" (un programa que simplemente muestra una cadena de texto como el de la sección 2). El segundo tiene una interfaz más complejo, incluyendo un menú, un botón y una caja de texto.

4.2 Un ejemplo de AWT

La versión gráfica del programa `HolaMundo` tiene una apariencia como la que muestra la figura 3, y aunque su código es reducido, sirve para introducir una gran parte de la interfaz para aplicaciones gráficas.

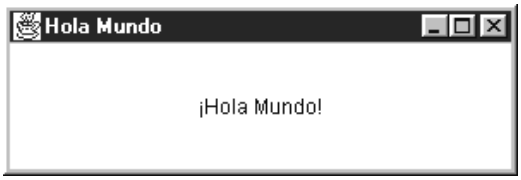

Figura 3: Mostrando texto con el AWT

A continuación aparece el código correspondiente a la aplicación anterior, que comienza importando los paquetes de AWT que se utilizan en la definición de la clase `InicioAWT`.

```
import java.awt.Event;
import java.awt.Frame;
import java.awt.event.*;
import java.awt.Label;

public class InicioAWT extends Frame {
    public static void main(String args[]) {
        InicioAWT programa = new InicioAWT("Hola Mundo");

        programa.setSize(300,100);
        programa.show();
        System.out.println("Ejecutando el main()");
    }

    public InicioAWT(String Titulo) {
        super(Titulo);
        add (new Label("Primer programa AWT", Label.CENTER), "Center");

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent event) {
                dispose();
                System.exit(0);
            }
        });
    }
}
```

Las aplicaciones gráficas heredan de la clase `Frame` (marco), es decir, su esqueleto básico es una ventana. Así, en el ejemplo `InicioAWT` hay que especificar las dimensiones de la ventana, para lo que se utiliza el método `setSize`. Para mostrarla se invoca el método `show`. En resumen, el método `main` de este programa dimensiona la ventana, la muestra y finalmente muestra un mensaje en la pantalla.

En el caso de los `applets` es el navegador el que proporciona la ventana. Por tanto, y como se vió en la sección anterior, el navegador es el que se encarga de indicar su tamaño inicial y de controlar eventos como los de cierre de la ventana. En caso de utilizar *frames* dichas tareas recaen en la propia aplicación.

Para insertar el texto dentro de la ventana se utiliza uno de los componentes del AWT, una etiqueta (`Label`). Éste objeto se crea a la vez que se inserta en la ventana mediante el método `add`, en el que se indica el objeto a insertar (la etiqueta) y su posición ("Center"). La etiqueta se crea mediante un constructor en el que se indica el texto que aparecerá en la misma y la posición dentro de ella.

El último componente que se añade a la aplicación es un `EventListener`. Como indica su nombre, "EscuchaEventos", su misión es atender los eventos que se produzcan. Por ejemplo, el hecho de pulsar el botón de cierre de la aplicación no la cierra directamente, simplemente genera un evento que debe tratarse.

Los `EventListener` aparecieron con el nuevo modelo de gestión de los eventos introducido en la versión 1.1. Anteriormente el modelo se basaba en la "herencia", mientras que ahora se basa en la "delegación". La idea es sencilla: los componentes como botones, cajas de texto, etc. generan eventos como respuesta a las acciones del usuario. Estos eventos se reciben y tratan mediante "escuchadores" adecuados.

Un `XYZListener` hay que "registrarlo" en el componente al que "escucha" mediante un método de tipo `addXYZListener`. Una vez hecho, los métodos adecuados de la interfaz del `Listener` se disparan cuando se reciben los eventos para los que se haya registrado.

En el programa `InicioAWT` no se utiliza directamente un objeto `WindowListener`, se utiliza en su lugar un "adaptador", `WindowAdapter` [4], que sirve para simplificar el trabajo de sobrescribir los métodos que no son interesantes para la aplicación. Concretamente un `WindowListener` tiene los métodos que aparecen en la Tabla 3, en la que también se muestra el evento que los dispara. Por tanto, si se usara un `WindowListener` habría que definir todos estos métodos en la declaración del "escuchador", aunque la parte correspondiente a su código estuviese vacía.

Método	**Evento**
`void windowActivated(WindowEvent)`	Ninguno
`void windowDeactivated(WindowEvent)`	Ninguno
`void windowClosing(WindowEvent)`	Ninguno
`void windowClosed(WindowEvent)`	`WINDOW_DESTROY`
`void windowOpened(WindowEvent)`	`WINDOW_EXPOSE`
`void windowIconified(WindowEvent)`	`WINDOW_ICONIFY`
`void windowDeiconified(WindowEvent)`	`WINDOW_DEICONIFY`

Tabla 3: Métodos de un *WindowListener*

Un "adaptador" como el usado en el ejemplo `InicioAWT` los proporciona ya rellenos. Así, solamente hace falta reemplazar el método `windowClosing`, que es el que se activa al recibir el evento de cerrar la ventana, que a su vez se produce cuando el usuario pulsa el botón superior de la ventana (el que cierra la aplicación).

[4] Es una clase interna, ver apartado 4.3

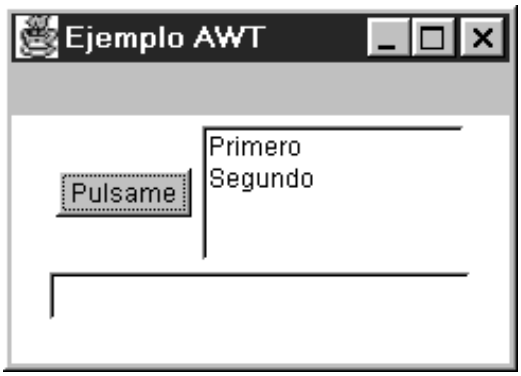

Figura 4: Una interfaz con botones, listas y cajas de texto

En el caso de la aplicación de la Figura 4 se han incluido más componentes: un botón, una caja de texto y una lista que a su vez tiene dos elementos. Además se ha utilizado directamente un `ActionListener`, que sólo incluye el método `actionPerformed`.

```
import java.awt.*;
import java.awt.event.*;

public class EjemploAWT extends Frame {
    // Elementos de la aplicación: un botón, una lista y una caja
    private Button   boton      = new Button("Pulsame");
    private List     lista      = new List();
    private TextField cajatexto = new TextField(25);
    private MenuItem elemento   = new MenuItem ("Elemento Menú");

    static public void main(String args[]) {
        EjemploAWT f = new EjemploAWT();
        f.setBounds(200,200,200,125);
        f.show();
        System.out.println("Arrancando...");
    }

    public EjemploAWT() {
        super("Ejemplo AWT");
        // Componentes básicos de un menú
        MenuBar barramenu = new MenuBar();
        Menu    menu      = new Menu();
        // Relleno del menu
        menu.add(elemento);
        barramenu.add(menu);
        setMenuBar(barramenu);
        // Relleno la lista
        lista.add("Primero");
        lista.add("Segundo");
        // Relleno de la ventana
        setLayout(new FlowLayout());
```

```
        add(boton);
        add(lista);
        add(cajatexto);
        // Manejadores de los eventos de los componentes
        boton.addActionListener(new GestorEventos());
        lista.addActionListener(new GestorEventos());
        cajatexto.addActionListener(new GestorEventos());
        elemento.addActionListener(new GestorEventos());
        // Manejador de eventos para la ventana
        addWindowListener(new GestorEventosVentana());
    }
}

class GestorEventosVentana extends WindowAdapter {
    public void windowClosing (WindowEvent evento) {
        Window ventana = (Window)evento.getSource();
        ventana.setVisible(false);
        ventana.dispose();
        System.exit(0);
    }
}

class GestorEventos implements ActionListener {
    public void actionPerformed(ActionEvent evento) {
        System.out.println(evento);
    }
}
```

El programa `EjemploAWT` muestra la forma de integrar los distintos componentes en una aplicación. Por ejemplo, se añaden los distintos elementos de un menú, instancias de `MenuItem`, dentro de una barra de menú, `MenuBar`, que a su vez se introduce en el menú de la aplicación (objeto `Menu`). Para ello se utiliza siempre el método `add` que poseen los componentes gráficos.

De la misma forma se incluyen el resto de los elementos dentro de la ventana. La pregunta que surge es cómo se colocan los elementos. Se dice por ejemplo que se añade el botón (`add(boton)`) pero no se dice en qué posición ni con qué tamaño. La forma en que se gestiona la colocación es usando `Layouts` (disposiciones).

La idea vuelve a ser delegar la colocación. Así, en `EjemploAWT` se ha elegido una colocación flotante. Para ello se indica que el manejador del diseño utilizado es `FlowLayout`. Esto quiere decir que si la ventana se redimensiona, el entorno intentará colocar los componentes de la mejor forma posible. Por supuesto, existen otros manejadores de diseño como `BorderLayout` que permite especificar posiciones relativas como arriba, abajo, etc.

La acción que se realiza cuando se recibe cualquier evento de acción sobre cualquiera de los componentes de los que escucha (el botón, la lista, la caja de texto o los elementos de la lista) es mostrar el nombre del evento producido. Con este ejemplo se muestra que es posible que un mismo gestor pueda atender eventos de más de un componente.

4.3 Tratamiento de eventos con clases internas

En el ejemplo `EjemploAWT` se tiene que asociar de alguna forma la clase `GestorEventosVentana` con el componente para el cual gestiona los eventos. La forma de hacerlo es mediante el método `java.util.EventObject.getsource()` que devuelve una referencia al objeto que recibió el evento. Otra solución podría ser asociar directamente el gestor de eventos con el componente del que va a recibir eventos, indicándolo en el constructor:

```
class GestorEventosVentana extends WindowAdapter {
     private Window ventana;
     public GestorEventosVentana (Window ventana) {
        this.ventana = ventana;
    }

    public void windowClosing (WindowEvent evento) {
        ventana.setVisible(false);
        ventana.dispose();
        System.exit(0);
    }
}
```

Por otra parte, la clase `GestorEventosVentana` sólo trata eventos para la clase `EjemploAWT`, por lo que su código podría incluirse directamente dentro de esta clase. Esta solución es sin embargo poco aconsejable, pues mezcla la funcionalidad de la clase con el tratamiento de los eventos y no permite "extender" adaptadores, teniendo que "implementar" escuchadores.

Sin embargo, y ya que la idea de incluir el código "dentro" de la clase aporta encapsulación, se incluyó en el JDK 1.1 la posibilidad de declarar una clase dentro de otra, creandose las denominadas *clases internas*.

Las clases internas tienen acceso a las variables y métodos definidos en las clases que las contienen (excepto los definidos como privados). Además, al estar declaradas como clases separadas pueden extender adaptadores y evitar por tanto tener que rellenar todos los métodos.

El programa `EjemploAWT` implementado con clases internas tendría entonces una estructura mucho más compacta:

```
public class EjemploAWT extends Frame {
...
     public EjemploAWT() {
     ...
     addWindowListener(new GestorEventosVentana());
    }

    class GestorEventosVentana extends WindowAdapter {
    public void windowClosing (WindowEvent evento) {
        setVisible(false);
        dispose();
        exit(0);
    }
}
```

Esta solución, aunque ingeniosa, pues respeta la sintaxis del lenguaje, puede dar lugar a código de difícil comprensión si se usa mal. La sección 9.5 trata con más detalle estos problemas.

Por supuesto, este capítulo ha mostrado una visión muy reducida del entorno que proporciona Java para el desarrollo de aplicaciones gráficas. Sólo se han presentado algunos de los componentes básicos, aunque existen muchos otros como la tipografía del texto, las imágenes, las cajas de diálogo, las barras de desplazamiento (*scroll*), etc. Además, existen otros aspectos que no se han mencionado, como la generación y tratamiento de eventos de usuario, los componentes ligeros, etc.

La siguiente sección introduce otra biblioteca de uso muy extendido en la comunidad Java que se llama `jdbc`. Su misión es facilitar el acceso a bases de datos relaciones mediante el estándar SQL.

5 Acceso a Bases de Datos: JDBC

Un aspecto interesante en cualquier lenguaje de programación que pretenda ser de uso general es su capacidad para comunicarse con bases de datos. En esta sección se presenta una breve introducción a la integración de bases de datos con aplicaciones de red desarrolladas en Java.

5.1 JDBC

JDBC es el paquete para el acceso a base de datos que se incluye en el núcleo de Java. JDBC proporciona una interfaz independiente del tipo de base de datos para establecer conexiones con servidores de bases de datos relacionales, permitiendo consultas SQL a la base de datos y recibiendo conjuntos de datos como respuesta.

Dicho de una forma más técnica, JDBC es la implementación de la interfaz de nivel de llamada CLI (*Call-Level Interface*) definido por el consorcio X/Open y que implementan la mayor parte de los vendedores de bases de datos relacionales.

Desde el punto de vista práctico, para realizar transacciones con una determinada base de datos se necesita un manejador (*driver*) JDBC que actúe como puente entre las llamadas a métodos JDBC y la interfaz nativa de la base de datos. En general, será el vendedor quien suministre dicho *driver*. De hecho la mayoría de las bases de datos comerciales ya lo suministran.

5.2 Un vistazo a la API de JDBC

La API JDBC ofrece un conjunto de interfaces que reflejan los aspectos básicos del acceso a bases de datos relacionales. Dichas interfaces forman parte del paquete `java.sql`, incluyendo una interfaz para un `DriverManager`, una `Connection`, un `Statement` y un `ResultSet`, que se analizarán brevemente a continuación.

DriverManager

La clase `DriverManager` constituye el mecanismo básico en JDBC para establecer una conexión con una base de datos. La forma de usarla es crear una instancia de dicha clase haciendo a continuación una conexión con la base de datos llamando al método `getConnection`. La dirección de la base de datos a la que acceder se pasa como parámetro en un formato similar al de una una URL. La conexión con la base de datos se devuelve en forma de objeto `Connection` si se tuvo éxito, o se recibe una excepción en caso contrario.

Cada base de datos necesita un `driver` particular que se implementa con un objeto de la clase `Driver`. El objeto `DriverManager` se encarga de gestionar los `drivers` disponibles en cada instalación de Java.

Connection

La especificación de la base de datos se realiza mediante una cadena con formato similar al de una URL. El objeto `DriverManager` no realiza ningún procesamiento de dicha cadena, simplemente la pasa a cada `driver` para preguntar si entiende y soporta ese tipo de base de datos. Por ejemplo, los `drivers` que usen el protocolo ODBC para establecer la conexión utilizarán una cadena como: `jdbc:odbc:<nombre_base_datos>` para acceder a una base de datos local. La especificación de la API JDBC indica el formato para referirse a una base

de datos: `jdbc:<protocolo>:<informacion_BD>` donde el `protocolo` especifica el tipo de conexión e `información_BD` proporciona la información que se necesita para conectar con la base de datos.

Statement

La *interface* del objeto `Connection` permite realizar consultas a la base de datos. Dichas consultas se representan mediante objetos de tipo `Statement`.

La clase `Connection` proporciona los métodos necesarios para crear las consultas SQL:

createStatement(): se usa para realizar consultas SQL simples que no necesiten ningún parámetro. Su resultado es un objeto de tipo `Statement` que a su vez tiene un método `executeQuery` para realizar la consulta.

prepareStatement(): devuelve un objeto de tipo `preparedStatement`, una subclase de `Statement`, que permite que se fijen parámetros de entrada.

prepareCall(): devuelve un objeto de tipo `CallableStatement` que permite fijar parámetros de entrada y obtener parámetros de salida.

Una vez obtenido el objeto de tipo `Statement` se puede realizar la consulta propiamente dicha utilizando alguno de los métodos que proporciona la interfaz de esta clase:

executeQuery(): recibe un `Statement` y devuelve el resultado de la consulta dentro de un objeto de tipo `ResultSet`.

execute(): para consultas que pueden tener múltiples conjuntos de resultados.

executeUpdate(): permite realizar operaciones de tipo `INSERT`, `UPDATE` o `DELETE`.

Por defecto, todas las consultas se confirman (`commit`) individualmente de forma automática. Si por alguna razón se desea "confirmar" más de una consulta en una misma transacción, o por ejemplo deshacerla, se puede eliminar el mecanismo automático llamando al método `Connection.setAutoCommit(false)`. A continuación se puede realizar una serie de consultas (mediante objetos `Statement`) sobre la base de datos (usando la correspondiente `Connection`) y confirmarla con una única llamada al método `commit` de la clase `Connection`.

ResultSet

En JDBC se utiliza esta clase para representar el flujo de datos sin procesar (tal y como se obtienen de la base de datos) generado a partir de una consulta mediante la llamada a `executeQuery` de un objeto `Statement`.

Esta clase proporciona un mecanismo para iterar sobre cada una de las tuplas recibidas de la base de datos mediante el método `next` (como muestra el ejemplo siguiente). A su vez, cada uno de los campos de dichas tuplas puede ser accedido mediante métodos del tipo `getXXX`, como `getString`, `getInt`, `getFloat` etc. Esto implica que el usuario deberá conocer el tipo de datos que espera en cada elemento, puesto que cada dato se obtiene con su `getXXX` correspondiente.

Por último indicar que el mecanismo de gestión de las transacciones depende del fabricante del driver de la base datos. Es decir, cuando se itera sobre las tuplas de un `ResultData` mediante el método `next`, puede ser que se generen accesos individuales en la base de datos o que simplemente se consulte una *cache* local.

5.3 Un Ejemplo de acceso con JDBC

Quizá la forma más sencilla de mostrar el funcionamiento básico del JDBC sea el siguiente fragmento de código:

```
// Dirección de la base de datos
String URLBaseDatos = "jdbc:protocolo://host/basededatos";
// Estableciendo de la conexión
Connection ConexionBD =
    DriverManager.getConnection(URLBaseDatos, "usuario", "clave");
// Se ejecuta una consulta
Statement consulta = ConexionBD.getStatement();
ResultSet resultado =
    consulta.executeQuery("SELECT nombre, apellido FROM tabla_usuarios");
// Impresión de los resultados
while (resultado.next()) {
    String nombre = resultado.getString("nombre");
    String apellido = resultado.getString("apellido");
    System.out.println("Usuario " + nombre + " " + apellido );
}
```

En primer lugar, se identifica la base datos mediante una cadena tipo URL. En dicha cadena se especifica el servicio `jdbc` y se permite que los proveedores añadan sus subprotocolos en `protocolo`.

Una vez construida la URL de la base de datos, ésta se pasa como parámetro al método `getConnection` del manejador JDBC, junto al usuario y la clave. Este método se encarga de crear la conexión con la base de datos.

Para realizar las consultas se necesita construir un objeto de tipo `Statement`. El contenido de este objeto será una consulta SQL que devolverá como resultado todo el conjunto de tuplas (`nombre`, `apellido`) de la tabla `tabla_usuarios`. Dicho resultado se devuelve en un objeto de tipo `ResultSet`.

Con esta sección finaliza el primer bloque de este artículo. En él se han descrito las características básicas del lenguaje, así como algunas de sus bibliotecas más usadas. En el segundo bloque se abordarán cuestiones más avanzadas como el manejo de la concurrecia, el desarrollo de aplicaciones distribuidas o el empleo de JavaBeans.

6 Concurrencia en Java

Muchas aplicaciones necesitan mantener más de un flujo de control para realizar de forma eficiente su trabajo. Java proporciona un mecanismo para permitir distintos "hilos" (*threads* o procesos ligeros) dentro de un mismo programa.

Esta sección analiza este modelo, explicando su utilidad y presentando ejemplos de uso. Además, se analizan temas relacionados como la sincronización, los grupos de procesos, etc.

6.1 ¿Qué son los *threads*?

Los *threads* son entidades que permiten efectuar más de una tarea simultáneamente. En realidad un *thread* es tan sólo una abstracción que corresponde a un flujo de control. Cada *thread* se encarga pues de proporcionar un flujo de control que permite ejecutar instrucciones dentro del programa considerado.

Una característica de Java es el *multithreading*, lo que quiere decir que un programa Java en ejecución puede poseer más de un *thread* y, por tanto, más de un flujo de control. Es importante reseñar que **todas** las variables y código del programa están accesibles para todos los *threads*.

La razón de utilizar *threads* y no procesos diferentes (distintas máquinas virtuales de Java ejecutando distintos programas) es principalmente la eficiencia. Repartir el tiempo de procesador entre varios *threads* que se ejecutan en el mismo proceso es más "barato" que repartir dicho tiempo entre distintos procesos.

En términos prácticos, esto quiere decir que resulta factible emplear *threads* para actividades **superfluas**, como animar un icono en la interfaz de usuario, sin desperdiciar recursos. Una consecuencia inmediata es la posibilidad de aumentar la interactividad de la interfaz.

Java incluye el soporte necesario para *threads*, por lo que no es necesario el uso de bibliotecas adicionales que suministren dichas abstracciones. En realidad, la JVM requiere de la existencia de *threads* para su correcto funcionamiento (los utiliza, por ejemplo, el recolector automático de basura), con lo que no es de extrañar que dicho servicio esté disponible de forma estándar.

6.2 Un *thread*

Todo programa Java dispone de un *thread* [5] cuando comienza a ejecutarse. Dicho *thread* es el que ejecuta el punto de entrada principal: el método `main`.

Para crear un *thread*, basta con crear una instancia de la clase `java.lang.Thread`:

```
class MiThread extends Thread {
   public void run()
   { // hacer esto y lo otro...
   }
}
MiThread m = new Thread();  // Se instancia un thread.
m.start();                  // Se ejecuta.
```

Naturalmente, cualquier instancia de una clase derivada creará también un *thread*.

[5] Hay algunos otros que **siempre** existen y tienen relación con el funcionamiento de la JVM. No se discuten aquí por estar fuera del ámbito del presente artículo.

Un *thread* comienza a ejecutarse cuando se invoca el método `start`, que dispone lo necesario para que el *thread* pueda ejecutar. Acto seguido, hace que el nuevo *thread* comience su ejecución con el método `run`, el punto de entrada en el que comienza a ejecutarse el *thread* una vez inicializado. Puede considerarse como el `main` del *thread*. Cuando dicho método termina el *thread* concluye su ejecución.

En realidad, no es obligatorio implementar el método `run`, aunque como su implementación por defecto sencillamente no hace nada, es lo mas habitual.

6.3 ¿Cómo heredar de *thread* sin herencia múltiple?

También se dispone de la interfaz `Runnable`, que contiene el método `run`. Si una clase es ya una derivada de otra, pero aun así se desea que represente un *thread* basta con implementar `Runnable`.

```
class MiThread extends OtraCosa implements Runnable {
   public void run()
   { // hacer esto y lo otro...
   }
}
```

Disponiendo de un objeto que pertenezca a una clase que implementa `Runnable` es posible crear un *thread*. Basta con suministrarle a dicho *thread* el objeto que implementa el método `run`

```
MiThread  o;
Thread m = new Thread(o,"un thread");
m.start();
```

6.4 Estado de ejecución y métodos

Un *thread* pasa por diversos estados desde su creación hasta su destrucción:

new El *thread* está recién creado.

runnable El *thread* puede ejecutarse.

blocked El *thread* no puede ejecutarse por alguna razón. Por ejemplo, está esperando a que haya algo que leer de un fichero.

dead El *thread* ha terminado su ejecución.

El tiempo de procesador se reparte entre los *threads* que están en estado `runnable`.

La clase `Thread` dispone de diversos métodos que afectan al estado de ejecución y permiten inspeccionar dicho estado.

stop() El *thread* dejará de ejecutarse.

isAlive() ¿Está el *thread* entre `new` y `dead`?

yield() Cede el procesador a algún otro *thread*.

suspend() Bloquea el *thread* hasta que se llame a `resume()`.

resume() Permite que el *thread* vuelva a obtener procesador.

join() Detiene la ejecución hasta que el *thread* termine.

6.5 Planificación

El entorno de ejecución de Java reparte el tiempo de procesador entre los *threads* mediante un sistema de prioridades fijas. Dicho sistema consiste en asignar una *prioridad* a cada *thread* y ejecutar en cada momento aquella de mayor prioridad que esté disponible (`runnable`).

Hay disponibles distintos niveles (valores) de prioridad: `{MIN_`, `{MAX_` y `{NORM_PRIORITY` (la menor, la mayor y un valor intermedio). El nivel de prioridad de un *thread* puede alterarse con el método `setPriority` de la clase `Thread`.

Existen dos formas de repartir el tiempo de procesador entre estos *threads* de igual prioridad: repartiendo el tiempo o dejando que cada *thread* se ejecute mientras quiera. En el primer caso se expulsa al *thread* que esta ejecutándose cuando éste haya agotado su tiempo de procesador; en el segundo es el mismo *thread* el que debe abandonar el procesador voluntariamente. La primera técnica se conoce como *time-slicing*, y **no** se puede utilizar en todas las implementaciones del entorno de ejecución de Java.

Si en un entorno sin *time-slicing* un *thread* no cede el procesador voluntariamente (no se bloquea y no llama a `yield`) puede que otros *threads* de igual prioridad no se ejecuten.

En general, siempre es mejor ceder el procesador de vez en cuando. Nunca se sabe que entorno tenemos debajo y puede que no ceder el procesador cause problemas a otros *threads*.

6.6 Grupos de *threads*

En muchas ocasiones se crea un conjunto de *threads* que cooperan para efectuar una tarea determinada. Para manipular conjuntos de *threads* se dispone en Java de los grupos de *threads*.

Un grupo de *threads* es una instancia de la clase `ThreadGroup`. Cada *thread* pertenece a un grupo (que podemos obtener con `getThreadGroup`) y los grupos pueden a su vez pertenecer a otros grupos. Se pueden crear nuevos grupos si se desea:

```
ThreadGroup supergrupo = new ThreadGroup("mi grupo");
ThreadGroup subgrupo = new ThreadGroup(supergrupo, "mi subgrupo");
ThreadGroup migrupo = getThreadGroup();
```

Naturalmente, se puede iterar sobre los *threads* de un grupo empleando métodos de `ThreadGroup`. Además es posible manipular grupos enteros empleando operaciones existentes también en *threads*, tales como `resume`, `stop` y `suspend`.

6.7 Sincronización

Para sincronizar la actividad de diversos *threads* y evitar condiciones de carrera en el acceso a recursos compartidos, se dispone de la etiqueta `synchronized`.

```
public synchronized void sumar()
{
        // Un thread dentro como mucho
```

```
        balance = balance + mas_dinero();
  }
```

Si se califica un método como `synchronized`, sólo podrá ejecutar dicho método (u otros métodos `synchronized` del mismo objeto) un *thread* a la vez. Una vez que un *thread* comience a ejecutarlo, otros *threads* que invoquen alguno de los métodos `synchronized` deberán aguardar a que el *thread* que comenzó termine. A esta abstracción se la denomina *monitor*.

En realidad, cada objeto con métodos sincronizados es un monitor. Cuando un *thread* intenta ejecutar su código empleando un método sincronizado, debe adquirir el cierre antes de comenzar la ejecución de dicho método.

Una vez que el *thread* posee el cierre (está en el monitor), puede invocar libremente a cualquiera de sus métodos sin bloquearse. Es decir, los monitores de Java son reentrantes. De no ser reentrantes, se podrían sufrir interbloqueos.

Esta sincronización es suficiente para evitar condiciones de carrera. No obstante, en algunas ocasiones, un método sincronizado no puede continuar su ejecución hasta que se cumpla una determinada condición. Por ejemplo, el método `cogerElemento` de un *buffer* compartido deberá aguardar a que existan elementos que coger.

A tal efecto, cada objeto posee una *variable condición* y métodos `wait`, `notify` y `notifyAll` como muestra el siguiente ejemplo:

```
public synchronized Elemento cogerElemento(void){
   while (esVacio()){
     try {
       wait(); // Esperar que alguien ponga un elemento.
     } catch (InterruptedException e) {}
   }
   Elemento = unElemento;
   notifyAll();  // Avisar a ponerElemento que
                 // puede estar esperando que queden huecos libres.
}
```

El primero se emplea para esperar a que una condición se cumpla. El segundo se invoca cuando dicha condición se cumple; si existiesen varios *threads* esperando por dicha condición se despertaría a uno de ellos. El tercero despierta a *todos* los *threads* que esperan.

Se recomienda utilizar estas variables condición como en el ejemplo anterior. Este tipo de construcción emplea un bucle para evaluar la condición, lo que es necesario puesto que puede haber *threads* esperando condiciones distintas. Aunque podría padecer inanición, ésta es rara en la práctica y la existencia de una única variable condición por objeto hace más simple la implementación [6].

A continuación se presenta una descripción del modelo que proporciona Java para la construcción de sistemas distribuidos. Con estas dos secciones se intenta dar una visión de las ventajas de Java para sistemas de tipo concurrente y distribuido.

[6] De todas formas, esto presenta varios problemas que se analizan en el apartado 9.7

7 Invocación remota de métodos en Java (Java RMI)

Para construir cualquier tipo de sistema distribuido es necesario disponer de mecanismos que permitan la comunicación entre procesos que estén ejecutándose en máquinas diferentes. Para ello, Java proporciona mecanismos de relativo bajo nivel, como los *sockets* analizados anteriormente. Sin embargo, en muchas ocasiones, es más conveniente utilizar abstracciones de más alto nivel, con las que el programador no tenga que preocuparse de establecer protocolos de comunicación. Para ello, Java proporciona el mecanismo de invocación de métodos remotos (*Remote Method Invocation*, RMI). Dicho mecanismo puede entenderse como una extensión de la RPC (llamada a procedimiento remoto, *remote procedure call*) en un entorno distribuido basado en objetos.

En realidad, RMI no es un mecanismo exclusivo de Java. Otros sistemas para programación distribuida, como por ejemplo CORBA o Ada95, también están basados en la invocación de métodos remotos. La diferencia fundamental del RMI de Java es que ha sido diseñado teniendo la integración con el lenguaje como meta fundamental, igual que ocurre con Ada95. Por lo tanto, se supone que va a funcionar en un entorno homogéneo, donde todos los procesos que se comunican van a funcionar sobre una máquina virtual Java. Esto le permitirá aprovecharse de las particularidades del modelo de objetos de Java siempre que esto sea posible.

El RMI de Java[7] proporciona fundamentalmente la posibilidad de invocar transparentemente objetos en máquinas virtuales remotas. Sin embargo, esta transparencia no es completa, pues los objetos que pueden ser invocados de esta forma han de pertenecer a una cierta clase. Esto permite que el programador sea consciente de cuándo está haciendo una llamada potencialmente remota, lo que le permite tener en cuenta sus especificidades. Por lo demás, se ha integrado el modelo de objetos distribuidos dentro del lenguaje de una forma bastante natural, manteniendo para estos objetos la mayor parte de la semántica habitual de los objetos, y preservando también la seguridad que proporciona el sistema de tiempo de ejecución de Java.

Aunque en las implementaciones actuales el RMI de Java sólo permite la invocación de objetos "sencillos", la arquitectura del sistema está preparada para permitir invocaciones múltiples (por ejemplo, a todos los representantes de un objeto replicado), de forma relativamente transparente.

7.1 Soporte para RMI

El soporte para RMI en Java está basado en las interfaces y clases definidas en los paquetes `java.rmi` y `java.rmi.server`.

Para que una interfaz pueda ser utilizada de forma remota, ésta ha de extender, directa o indirectamente, la interfaz `java.rmi.remote`, que no define ningún método:

```
public interface Remote {};
```

Todos los métodos de cualquier interfaz que extienda `java.rmi.remote` deben declarar que en ellos puede surgir la excepción `java.rmi.RemoteException`. Esta excepción es la superclase de todas las excepciones que pueden surgir en el sistema de ejecución que soporta el RMI. Por ejemplo, se levantan excepciones de esta familia cuando la invocación remota falla por problemas en las comunicaciones.

[7] De ahora en adelante nos referiremos en muchos casos al RMI de Java simplemente como "RMI".

Además, si los métodos de la interfaz remota necesitan pasar algún objeto como argumento o valor de vuelta, debe declararse la interfaz remota que implementa (y no simplemente la clase a la que pertenece). En otras palabras, el acceso remoto se declara únicamente a través de interfaces remotas.

En cuanto a clases para implementar servidores, el lenguaje proporciona:

`java.rmi.server.RemoteObject` Implementa la interfaz `java.rmi.remote`, proporcionando además la semántica remota de `Object` (métodos `hashCode`, `equals` y `toString`).

`java.rmi.server.RemoteServer` Extiende `java.rmi.server.RemoteObject`, y proporciona métodos abstractos para crear objetos que puedan actuar como servidores y "exportar" estos objetos.

`java.rmi.server.UnicastRemoteObject` Extiende `java.rmi.server.RemoteServer`, proporcionando implementaciones para sus métodos abstractos. Estas implementaciones definen un objeto "sencillo" (no replicado). Por ahora, es la única clase no abstracta que se ofrece para implementar servidores.

Una clase para objetos servidores normalmente se genera extendiendo la clase básica `java.rmi.server.Uni` e implementando una o varias interfaces remotas. De `UnicastRemoteObject` se hereda la funcionalidad que permite que sea invocado mediante RMIs, y de las interfaces remotas, la interfaz del servicio o servicios que ha de implementar.

Además de poder construir objetos cuyos métodos se puedan invocar remotamente, es preciso que los clientes (los objetos que quieran invocarlos) puedan localizarlos. Para ello se utiliza habitualmente un servicio de directorio o de nombrado. En el entorno de Java pueden utilizarse los servicios de `java.rmi.Naming`, que permite registrar y obtener referencias a objetos, relacionándolas con las URLs de los lugares donde "residen".

7.2 Ejemplo: una cuenta corriente

El siguiente ejemplo está basado en `BankAccount`, descrito en [Sun, 1997]. Con él se va a mostrar cómo se construye el esqueleto de un objeto remoto sencillo: una cuenta corriente que permite depositar dinero en ella, reembolsarlo, y solicitar el saldo actual.

El primer paso será modelar la interfaz remota que va a ofrecer la cuenta, extendiendo la interfaz `java.rmi.remote`:

```
public interface CuentaCorriente
  extends Remote
{
      public void deposito (float cantidad)
        throws java.rmi.RemoteException;
      public void reembolso (float cantidad)
        throws NumerosRojos, java.rmi.RemoteException;
      public float saldo ()
        throws java.rmi.RemoteException;
}
```

A continuación, se escribe la clase que implementa esta interfaz (extendiendo la clase `UnicastRemoteObject` para obtener la funcionalidad de invocación remota):

```
package Banco;

public class ImplCuentaCorriente
  extends java.rmi.server.UnicastRemoteObject
  implements CuentaCorriente
{
      public void deposito (float cantidad)
        throws java.rmi.RemoteException {
      ...
      }

      public void reembolso (float cantidad)
        throws NumerosRojos, java.rmi.RemoteException {
      ...
      }

      public float saldo () throws java.rmi.RemoteException {
      ...
      }
}
```

Para realizar el registro con el servicio de nombrado (habitualmente llamado *binding*), el programa Java que quiera instanciar un objeto de esta clase puede usar un código similar al siguiente:

```
CuentaCorriente cuenta = new ImplCuentaCorriente();
String url = "rmi://algun.sitio.org/cuenta_corriente";
java.rmi.Naming.bind(url, cuenta);
```

A partir de este momento, cualquier objeto en otra máquina virtual puede acceder a esta cuenta, naturalmente si conoce su URL. Por ejemplo, podría usar un código de este estilo para conocer su saldo:

```
String url = "rmi://algun.sitio.org/cuenta_corriente";
CuentaCorriente cuentaRemota = (CuentaCorriente)java.rmi.Naming.lookup(url);
float saldoActual = cuentaRemota.saldo();
```

7.3 Arquitectura del sistema RMI

La especificación de la arquitectura del soporte para RMIs en Java diferencia tres niveles, cada uno con tareas bien definidas:

- El nivel de transporte. Se encarga de las comunicaciones, asegurando que los datos necesarios para realizar las invocaciones remotas fluyen entre las máquinas virtuales involucradas. Para ello, se encarga de mantener las conexiones necesarias, de monitorizar su estado, de escuchar (y establecer) nuevas conexiones, etc.

- El nivel de gestión de las referencias remotas. Trata de los aspectos relacionados con el comportamiento esperado de las referencias remotas. Por ejemplo, si una referencia dada corresponde a un servidor replicado, todos los detalles relativos a la replicación se gestionan en este nivel. O si se especifica algún mecanismo para recuperación en caso de problemas de conectividad, también se gestiona aquí.

- El nivel de representante/esqueleto (*proxy/skeleton*). El funcionamiento de la RMI se basa en la inclusión en el cliente de un *proxy* (representante local) que se encarga de la gestión de la invocación remota, y en el servidor de un "esqueleto", que recibe las peticiones de los clientes, realiza la invocación del método en cuestión, y devuelve los resultados pertinentes. Una de las labores de este nivel es el "aplanamiento" (*marshalling*) de los parámetros de la llamada, y del valor de vuelta, si lo hay. Para ello se utiliza el mecanismo de Java llamado "serialización de objetos", proporcionado por los *marshall streams* (clases de `java.io`).

Así, cuando un cliente realiza una invocación remota, en realidad hace una invocación de un método del representante local. Este representante local no es más que una implementación de la interfaz del objeto remoto, y es generado por el compilador de interfaces remotas (habitualmente la herramienta `rmic`). Los métodos del representante hacen llamadas al nivel de referencias remotas, que a su vez usa el nivel de transporte para enviar los datos necesarios hasta la máquina virtual donde reside el objeto remoto (o al conjunto de máquinas donde residen las réplicas, por ejemplo, en el caso de un objeto replicado). En el lado del objeto remoto, los datos que llegan al nivel de transporte son recogidos por el nivel de referencias remotas, que los pasa a su vez al esqueleto (en algunos casos, tras "activar" el objeto invocado). El esqueleto, con la información que recibe, decide qué método del objeto hay que invocar, y lo invoca localmente. El valor de vuelta de la invocación es devuelto hacia el objeto invocante de forma similar. El esqueleto que usa el objeto remoto también es generado por el compilador de interfaces remotas.

7.4 Carga dinámica de clases

A diferencia de los sistemas RPC, donde los clientes necesitan que el código del representante se una a ellos en tiempo de enlazado (*link*), en RMI se pueden cargar en tiempo de ejecución las clases que el cliente necesita para realizar la invocación remota de un objeto cualquiera. Para ello se utiliza el mecanismo llamado "carga dinámica de clases", que puede cargar en el cliente el *bytecode* de las clases que implementan el objeto remoto, su representante (en el lado del cliente), su esqueleto (en el lado del servidor) y otras clases usadas en las interfaces remotas involucradas.

Para realizar esta carga remota se usa habitualmente un cargador de clases especializado. Este suele ser `RMIClassLoader` (en lugar del cargador por defecto o el cargador de *applets*). `RMIClassLoader` trata primero de cargar cada una de las clases necesarias del almacén local (indicado en el `CLASSPATH`). A continuación, si las clases se refieren a objetos especificados como parámetros, se usa la URL que debe incluir su representación "aplanada". Si las clases corresponden a representantes o a esqueletos se usa la URL especificada por la propiedad `java.rmi.server.codebase`.

Si se quiere, por motivos de seguridad por ejemplo, se puede configurar el cargador de forma que sólo intente cargar clases del servidor especificado. En ese caso, si en ese servidor

no se encuentra alguna de las clases necesarias, la invocación del método correspondiente falla, levantándose una excepción.

También por motivos de seguridad, se requiere la presencia de un gestor de seguridad (*Security Manager*) para cargar clases desde un almacén remoto. Si se intenta esta carga sin tener gestor de seguridad, se produce una excepción. Una aplicación puede, si lo desea, utilizar su propio gestor de seguridad, o utilizar el gestor por defecto, `RMISecurityManager`. Los gestores de seguridad típicamente se aseguran que las clases en cuestión cumplen las garantías habituales en Java (por ejemplo, que no intentan invocar ciertos métodos considerados "sensibles").

Por último, indicar que el mayor problema de seguridad que queda abierto (y que los gestores de seguridad no pueden controlar) es el uso indiscriminado de recursos por parte de una clase.

En el siguiente apartado se analiza un tema muy diferente del abordado en este apartado y en el anterior: los Java Beans la programación visual basada en componentes Java. Se ha incluido en este segundo bloque por tratarse de una ampliación de las posibilidades del lenguaje.

8 JavaBeans

Un JavaBean, o un Bean, no es más que un componente software reutilizable que ofrece la ventaja de que se puede manipular fácilmente gracias a sencillas herramientas gráficas que simplifican su uso e integración dentro de aplicaciones más grandes, Hasta el límite de hacer accesible la generación de aplicaciones a personas que nunca antes habían programado.

El objetivo que Sun persigue con el lanzamiento de los Beans es la creación de un nuevo mercado del software alrededor de estos componentes, para acercar el mundo de la programación a un colectivo de usuarios más numeroso, al no necesitar más formación previa que la del manejo de cualquier entorno de ventanas. Con este fin ofrece un conjunto de directivas para el desarrollo tanto de Beans como de herramientas que permitan su creación y uso, ya que el éxito o fracaso de ese nuevo mercado radica en la calidad de los productos que ofrezca.

8.1 Introducción a JavaBeans

JavaBeans es un modelo de componentes software independiente de la plataforma y por lo tanto transportable. Los Beans son clases que se pueden manipular en una herramienta visual de construcción de aplicaciones que utiliza esos Beans como unidad básica de desarrollo. Cualquier clase de Java que incluya ciertas propiedades y respete ciertas reglas en cuanto a eventos se refiere, puede ser un Bean.

La herramienta básica de desarrollo de JavaBeans es el *Beans Development Kit* (BDK) donde se incluye la *BeanBox* que además de ofrecer la posibilidad de probar los Beans desarrollados, permite manejar Beans de forma visual. Además de esta herramienta básica que se puede conseguir a través de la red sin ningún coste [8], Sun ofrece también un producto comercial que incluye una herramienta gráfica más potente: *Java Workshop*. Además de los Beans, ambas permiten la construcción de applets de una forma más o menos sencilla a partir de la unidad básica: el Bean.

Terminología Beans

Tanto la construcción como la utilización de Beans introduce a continuación una terminología que aparece resumida:

- Propiedades: se llama propiedades a los atributos públicos de una clase. Las propiedades pueden ser de lectura/escritura, de sólo lectura o de sólo escritura. Existen cuatro tipos de propiedades:

  - Propiedades simples: representan un valor simple al que se puede acceder a través de los métodos *set* y *get* concatenados con el nombre del atributo correspondiente.
  - Propiedades indexadas: representan matrices de valores y sus correspondientes métodos *set* y *get* reciben un índice como parámetro aunque estas propiedades también pueden incluir métodos set y get que actúen sobre toda la matriz.
  - Propiedades *bound*: son propiedades que tienen asociados dos métodos que le permiten a la *BeanBox* añadir o eliminar un *listener* que se encarga de capturar el

[8] http://java.sun.com

evento que se produce cuando se modifica el valor de esa propiedad, es decir, en su correspondiente método *set*.

- Propiedades *constrained*: un objeto que posea este tipo de propiedades le permite a otros objetos vetar la posible modificación del valor de dicha propiedad. Para ello se sirve del método *set* asociado, el cual, antes de llevar a cabo la modificación propuesta, lanza una excepción alertando del cambio.

- *Personalización*: Es posible definir la apariencia y el comportamiento de un Bean en los entornos en los que vaya a utilizarse, sirviéndose de las siguientes interfaces:

  - Interfaz `Customizer`: mediante la implementación de esta interfaz es posible modificar la apariencia de los elementos gráficos del Bean, de modo que muestre una Interfaz Gráfica de Usuario distinta en la *BeanBox* o en cualquier otra herramienta gráfica.
  - Interfaz `PropertyEditor`: la implementación de esta interfaz permite la creación de un editor a medida para una propiedad.
  - Interfaz `BeanInfo`: cada Bean debería incluir alguna implementación de esta interfaz; de ese modo definiría los nombres con que aparecen las propiedades, métodos y eventos en las herramientas gráficas, así como una pequeña ayuda para cada uno de ellos.

- Persistencia: los atributos de una instancia de una clase que implemente `Serializable` se guardarán automáticamente, aunque existe la posibilidad de evitar que parte de ellos no se almacenen haciéndolas `transient` o `static`.

- Paquetes: los JavaBeans se distribuyen en forma de archivos JAR. Un archivo JAR es un archivo en formato ZIP que puede incluir un archivo MANIFEST describiendo su contenido. Este tipo de archivos pueden contener archivos `.class`, archivos de ayuda en formato HTML, etc. Si el archivo JAR no contiene un MANIFEST, entonces todas las clases dentro del paquete serán consideradas como Beans. La inclusión de este fichero ofrece la posibilidad de especificar qué clases son Beans y cuáles no.

8.2 Construcción de Beans

Las características de un componente software de este tipo se podrían resumir en las siguientes:

Creación: A la hora de crear un nuevo Bean se deben respetar dos reglas básicas impuestas por la arquitectura JavaBean:

- La clase del Bean debe incluir un constructor sin argumentos de modo que el método `Beans.instantiate()` pueda instanciar ese componente.
- El Bean debe soportar la persistencia, bien implementando *Serializable*, o bien *Externalizable*.

Métodos: Los métodos públicos son los encargados de describir el comportamiento de un Bean. De ellos se sirven las herramientas gráficas, siendo la base para la construcción de conexiones entre Beans.

Propiedades: Para las propiedades más importantes se debería definir un evento que se produjera cada vez que se modifique dicha propiedad. Por ejemplo, cuando se modifica el tamaño de un botón se debería hacer la modificación proporcional en el tipo de letra o el dibujo que contiene. La arquitectura JavaBeans ofrece soporte para las propiedades `bound` y `constrained`. Aunque es posible tener una propiedad que es `constrained` pero no `bound`, la recomendación es que ambas características vayan unidas.

BeanInfo: La misión de *BeanInfo* es la de proporcionar a las herramientas de construcción suficiente información como para guiar a los usuarios que utilizan Beans como herramienta básica de construcción de sus aplicaciones java.

Eventos: Se puede utilizar *BeanInfo* para facilitar el uso de los eventos de los Beans mediante su catalogación como `hidden` y `expert`.

Editores de propiedades: Se debería incluir en cada Bean un editor de propiedades a medida unido a éste via *BeanInfo* o `PropertyEditorManager`.

Customizers: Es posible que se necesite una forma específica de representar un Bean o que se deba configurar siguiendo un determinado orden. La forma de proporcionarle esta información a la herramienta que trabaja con los Beans es asignándole un `Customizer`. De esta forma se produce una separación entre el comportamiento y la apariencia de un Bean en diseño y en ejecución.

Paquetes: Si no se dispone de información adicional, un Bean empaquetado dentro de un archivo JAR necesita para su funcionamiento el total de los archivos que contiene, que van desde los `.class` hasta documentación *Javadoc*, para su funcionamiento, lo que implica que todos esos archivos deberán incluirse en la distribución final de un producto que utilice ese Bean. Con el fin de solucionar este problema y separar los archivos necesarios para el diseño de los que estrictamente necesita el producto final se han definido las cabeceras `Design-Time-Only` y `Depends-on`, al mismo tiempo que se aconseja su utilización a los desarrolladores de Beans.

Beans con Beans: Aunque la finalidad última de los Beans es su utilización en la construcción de aplicaciones finales mediante su instanciación en una herramienta capaz de trabajar con ellos, existe también la posibilidad utilizarlos para crear nuevos Beans.

8.3 Herramientas para trabajar con Beans

La herramienta básica para el desarrollo de Beans es el *Beans Development Kit* (BDK) que se puede obtener a través de la red. La única facilidad gráfica que ofrece este paquete es la BeanBox, una aplicación que permite probar los Beans que se han construido con el BDK. Para una interfaz de desarrollo de Beans gráfica, hay que recurrir al *Java Workshop* de Sun, cuya distribución no es gratuita.

Hasta el momento se han expuesto las distintas opciones que existen para el desarrollo de Beans, sin embargo, lo que realmente caracteriza este tipo de componentes es la forma de utilizarlos para la construcción de aplicaciones. Con este fin Sun ha creado el *Java Studio* (ver figura 5).

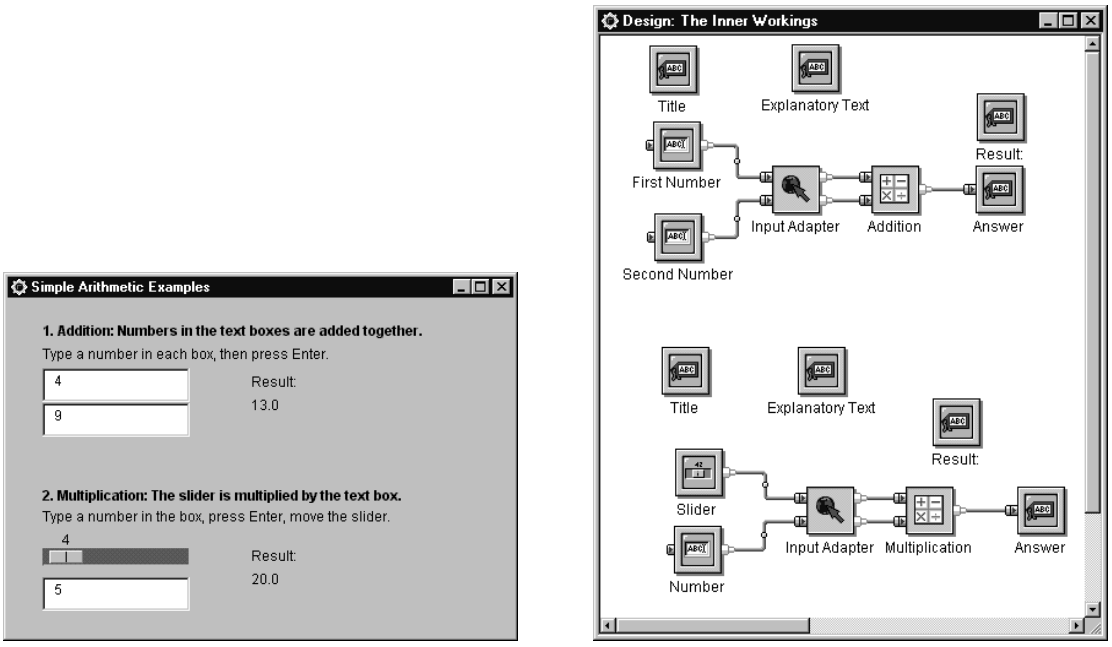

Figura 5: *Java Studio*: Diseño gráfico y estructura de Beans de una aplicación.

En la figura 5 aparecen las dos ventanas principales que se utilizan en el desarrollo de aplicaciones basado en Beans. Por un lado está el espacio dedicado al aspecto visual de la aplicación (ventana izquierda de la figura) y que es el lugar en el que el usuario va colocando cada uno de los componentes de que constará su aplicación, como pueden ser botones, campos de texto o etiquetas, entre otros. Esta ventana muestra el aspecto que tendrá la aplicación resultante y en ella se puede probar en cualquier momento el funcionamiento del trabajo en curso.

Por otro lado, en la parte derecha de la figura, aparece la estructura interna de los componentes utilizados en la aplicación, y que no necesariamente tienen que corresponderse con elementos gráficos incluidos en el diseño gráfico. Si se comparan las dos figuras se puede comprobar que hay componentes como `Title` que se corresponden con un elemento gráfico (como es el primero de los campos de texto) mientras que hay otros como `Addition` que no tienen correspondencia gráfica.

El proceso de creación de una aplicación utilizando *Java Studio* consta de dos partes. La primera consiste en la colocación de los elementos gráficos para formar la interfaz gráfica de usuario de la aplicación. La segunda consiste en proporcionarle a la aplicación la funcionalidad deseada, lo que se consigue conectando los componentes que la forman como muestra la ventana *Design: The Inner Workings*.

Una vez comprobado el correcto funcionamiento de la aplicación, tan solo resta generar el código Java correspondiente, tarea que lleva a cabo el entorno de desarrollo de forma automática.

Con esta sección finaliza el segundo bloque del artículo, en el que se han analizado partes más avanzadas de la tecnología Java.

Resta un tercer bloque, constituido por la sección 9. En ella se realiza un análisis crítico de algunas de las decisiones de diseño del lenguaje. Su objetivo es alertar a los programadores en

Java sobre algunos problemas que pueden surgir si no se usan con cuidado ciertas partes del lenguaje.

9 Consideraciones sobre el diseño de Java

La tecnología Java (término con el que se puede englobar a la máquina virtual JVM, a las bibliotecas y al lenguaje de programación) ha experimentado una fuerte expansión, desde su aparición en el año 1995, en el número de programadores, de bibliotecas más o menos estándar, de anuncios en medios de comunicación, de libros, de páginas de WWW, de modificaciones y extensiones de las bibliotecas y hasta del propio lenguaje, etc.

Esto nos da una idea del éxito que está alcanzando la tecnología Java, como experimento de mercadotecnia a escala mundial, pero quizá también como tecnología en auge en el mundo industrial y académico.

Sin embargo, la existencia de mucha información puede, paradójicamente, conllevar una considerable desinformación. Y es que a veces es difícil separar los anuncios y promociones de la tecnología, de los libros o páginas de WWW que tratan de proporcionar información precisa.

La tecnología Java es interesante fundamentalmente por haber difundido a gran escala elementos tecnológicos ya existentes, pero poco utilizados en la industria. Por ejemplo, la recolección automática de basura, la transportabilidad del código entre diferentes plataformas o la programación concurrente con soporte del lenguaje, por citar sólo algunos.

El lenguaje de programación es quizá el elemento más visible de la tecnología Java y, por lo tanto, el que más atención ha recibido. Tanto es así que empieza a ser habitual considerar dicho lenguaje como *canon* de la programación, pareciendo difícil en la actualidad abordar un proyecto de programación con otro lenguaje. Incluso se extiende la costumbre de recomendar y utilizar Java para enseñar a programar a estudiantes en el mundo entero. Sin embargo pueden encontrarse aspectos del lenguaje que pueden hacerlo poco adecuado tanto en el ámbito docente como para desarrollar grandes proyectos de programación. En algunos casos se trata de ausencias de mecanismos en el lenguaje, en otros de mecanismos propensos a ser mal utilizados, y en otros de verdaderas "bombas de relojería", como la introducción de nuevos elementos del lenguaje camuflados como pequeñas modificaciones (clases internas), o el mal uso de terminología bien establecida (monitores).

9.1 Pequeños detalles que pueden llevar a confusión

Sobre los paquetes

La extensa biblioteca de Java es indudablemente uno de los puntos fuertes de la tecnología Java. Sin embargo su estructuración en paquetes presenta algunas peculiaridades.

Desde cualquier clase puede accederse a los componentes públicos de cualquier paquete, sin ser necesario anotar en el lenguaje ninguna referencia. La cláusula `import` proporciona una forma más corta de referirse a los objetos y métodos del paquete: sin cualificar con el nombre del mismo. Por eso se produce la curiosa paradoja de que:

- Por un lado parece conveniente recomendar el uso de una cláusula `import` para todos los paquetes que se utilicen en un fichero: con esta disciplina simplemente mirando los `import` podría saberse qué paquetes se usan. En caso contrario, la única forma de saber qué paquetes se usan en un fichero sería leer todas las líneas de código buscando las referencias cualificadas con el nombre del paquete.

- Pero, por otro lado, es desaconsejable referirse a clases sin cualificar con el nombre del paquete que las contiene, pues si en un fichero aparece una declaración como `Elemento e;`

no se conoce qué paquete es el que proporciona la clase `Elemento`, y habría que revisar todos los que aparecen en las cláusulas `import`. Apareciendo de esta forma, además, una polución del espacio de nombres.

Arrays

Un *array* en Java no es más que una referencia a una zona de memoria, lo que conlleva a peculiaridades que pueden sorprender al programador inexperto o al alumno que se enfrenta por primera vez a un lenguaje de programación:

- Los *arrays* multidimensionales en Java son en realidad "*arrays* de *arrays*", es decir, no tienen que ser "rectangulares". Así, un array bidimensional puede ser "triangular":

  ```
  static int[][] bidim = {{1, 2}, {3, 4, 5}, {6, 7, 8, 9}}
  ```

- La doble notación permitida para la declaración de *arrays* puede resultar confusa. Así, en el siguiente ejemplo `a` y `b` son *arrays* unidimensionales, mientras que `c` es un array bidimensional:

  ```
  int[] a, b, c[];
  ```

Paso de parámetros

En Java todos los parámetros se pasan **por copia**, es decir, se le pasa al método una copia del parámetro. Los cambios efectuados sobre la copia no tienen efecto sobre el parámetro real en la llamada. Esta semántica, aunque unívoca y precisa, de nuevo puede resultar difícil de asimilar por quienes aprenden Java como primer lenguaje.

Cuando el parámetro es de un *tipo básico* de Java (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`), este paso por copia no plantea problemas de comprensión. Pero resulta imposible modificar el valor de un parámetro actual de tipo básico o devolver un valor a través de él. Cuando se requiere esta funcionalidad es necesario crear un objeto que contenga al tipo básico, para así poder modificarlo o devolver un valor a través de la referencia al objeto que se pasa ahora como parámetro, lo que resulta farragoso y poco intuitivo.

Pero cuando el parámetro es de un *tipo referenciado* (objetos o *arrays*), y dado que ejemplares de estos tipos siempre se manejan a través de referencias, lo que se pasa como parámetro es una copia de la referencia a él.

Los siguientes ejemplos pueden plantear problemas de compresión a los alumnos que comienzan a programar:

1. ```
   public void Intercambiar(int a, int b) {
      int temp = a;
      a = b;
      b = temp;
   }
   ```

   Los enteros con los que se haga la llamada **no** intercambian su valor, ya que sólo se pasan copias de ellos.
   ```

2.
```
public void Intercambiar(Object a, Object b) {
   Object temp = a;
   a = b;
   b = temp;
}
```
Las referencias a los objetos con los que se haga la llamada **no** se intercambian, ya que sólo se pasan copias de dichas referencias.

3.
```
public void Intercambiar(MiObjeto a, MiObjeto b) {
   int temp = a.entero;
   a.entero = b.entero;
   b.entero = temp;
}
```
Los campos `entero` de los objetos referenciados en la llamada **sí** intercambian sus valores, ya que las referencias copiadas siguen referenciando a los mismos objetos de la llamada.

4.
```
public void Intercambiar(MiObjeto a, MiObjeto b) {
   MiObjetoInterior temp = a.interior;
   a.interior = b.interior;
   b.interior = temp;
}
```
Las referencias de los campos `MiObjetoInterior` de los objetos referenciados en la llamada **sí** se intercambian, ya que las referencias copiadas siguen referenciando a los mismos objetos de la llamada, que contienen las referencias a los objetos interiores.

9.2 Sobre constructores, bloques de inicialización y el método finalize

La creación y destrucción de los objetos no primitivos se controla en Java mediante los constructores, los bloques de inicialización y el método `finalize`.

Bloques de inicialización estáticos y de instancia

Los bloques de inicialización estáticos o de clase sirven para inicializar el estado de una clase. Puede haber varios de ellos en una misma clase, ejecutándose éstos según el orden en el que aparecen en el código fuente, en el momento en que se carga la clase.

Los bloques de inicialización no estáticos o de instancia (iguales pero sin la palabra `static`) se introdujeron junto con las clases internas anónimas en la versión 1.1 de la especificación de Java (ver el apartado 9.5).

Pero una vez introducidos en el lenguaje se pueden utilizar también en cualquier clase. Estos bloques se ejecutan cada vez que se crea una instancia de la clase a la que pertenecen, antes de ejecutarse su constructor, y después de ejecutarse el constructor de la clase padre. De haber varios, se ejecutan según el orden en el que aparecen en el código fuente.

Como el lector puede apreciar, la función de los bloques de inicialización no estáticos se solapa con la función de los constructores.

El uso combinado de bloques de inicialización estáticos y no estáticos puede producir efectos no deseados, como podemos ver en el siguiente programa:

```
class T1 {

  {
    System.out.println ("  Primer bloque de inic. de instancia");
  }

  static {
    System.out.println ("Primer bloque de inic. estático");

    T1 b = new T1 ("b");
  }

  {
    System.out.println ("  Segundo bloque de inic. de instancia");
  }

  static {
    System.out.println ("Segundo bloque de inic. estático");
  }

  public T1 (String name){
    System.out.println ( "    Constructor de: " + name);
  }
}

public class Main {
  static public void main (String [] argv) {
        T1 a = new T1("a");
  }
}
```

En la clase `T1` se han incluido 4 bloques de inicialización, dos estáticos y dos de instancia. En el método `main` de la clase `Main` creamos una instancia de la clase `T1`. A partir de ese momento se carga la clase `T1`, y comienza a ejecutarse el primer bloque de inicialización estático. Hay otro bloque de inicialización estático, pero como se puede ver en la salida del programa por pantalla que se muestra a continuación, ¡antes de que se ejecute el segundo bloque de inicialización estático se llega a crear una instancia de T1!:

```
Primer bloque de inic. estático
  Primer bloque de inic. de instancia
  Segundo bloque de inic. de instancia
    Constructor de: b
Segundo bloque de inic. estático
  Primer bloque de inic. de instancia
  Segundo bloque de inic. de instancia
    Constructor de: a
```

Se trata de la instancia `b`, que se declara en el primer bloque de inicialización estático. En la salida del programa por pantalla podemos apreciar cómo en ese momento se ejecutan los dos

bloques de inicialización de la instancia `b`, y a continuación su constructor. Nótese que la clase aún no está inicializada (falta el segundo bloque estático) y, sin embargo, ya se ha creado una instancia de ella.

Más tarde se ejecuta por fin el segundo bloque de inicialización estático, para terminar con la ejecución de los bloques de inicialización de la instancia `a` y de su constructor.

Encadenamiento y herencia de constructores

Las reglas que gobiernan la interacción entre constructores de una misma clase y constructores de clases relacionadas mediante herencia están bien definidas, pero son suficientemente complicadas como para que merezca la pena dedicarle algún tiempo extra a su estudio.

Los constructores no se heredan, pero cuando se crea un objeto de una subclase deben acabar encadenándose llamadas a los constructores de las clases ancestrales para que todas éstas tengan la oportunidad de inicializarse oportunamente. Las siguientes reglas tratan de garantizar este objetivo:

- Si una subclase no declara un constructor explícitamente, se define implícitamente uno sin argumentos, en cuyo cuerpo aparece la sentencia "`super ()`", esto es, una llamada al constructor sin argumentos de la clase padre. Nótese que su naturaleza implícita implica que nada de esto es visible en el código fuente.

  La clase padre no tiene por qué tener definido un constructor sin argumentos, pues según el párrafo anterior, si se hubiera definido un constructor explícitamente, ya no habría definido uno sin argumentos de manera implícita. No obstante, puede existir un constructor sin argumentos si se ha escrito de manera explícita.

  De no existir en la clase padre el constructor sin argumentos (definido explícita o implícitamente), no compilará la subclase.

- En el constructor de una clase puede aparecer una sentencia que llame a un constructor de la clase padre de manera explícita ("`super (parametros);`"), pero sólo como primera sentencia.

- En el constructor de una clase puede aparecer una sentencia que llame a un constructor de la misma clase de manera explícita "`this (parametros);`" o "`this ()`"), pero sólo como primera sentencia.

- Si la primera sentencia de un constructor no es ni una invocación a un constructor de la misma clase ni a un constructor de la clase padre, se inserta implícitamente (sin que aparezca en el código) una sentencia "`super ();`".

- En caso de encadenar llamadas a varios constructores de la misma clase (mediante llamadas "`this (parametros)`"), sólo en el último constructor se acabará llamando implícita o explícitamente al constructor de la clase padre.

Una consecuencia de este conjunto de reglas es que a la hora de declarar constructores para una clase que se está escribiendo, se tiene que tener en cuenta si la clase padre tiene declarados constructores, y de qué forma:

- Si la clase padre tiene un constructor no privado sin argumentos (implícito o explícito), la hija no tiene por qué declarar ningún constructor explícitamente.
- Si la clase padre no tiene declarado un constructor no privado sin argumentos (implícito o explícito), pero tiene declarados explícitamente uno o más constructores no privados, la clase hija tiene que declarar al menos un constructor, cuya primera sentencia ha de ser una llamada a uno de los constructores con argumentos de la clase padre.
- Por lo expresado en los dos puntos anteriores, si todos los constructores que declara explícitamente una clase son privados, la clase no se puede extender, por no poder escribir constructores para las clases hijas.

Resurrección de objetos en el método finalize

A diferencia de lo que ocurre en C++, en Java no hay destructores. Sin embargo existe el método `finalize`, que cumple una función similar. No obstante, su funcionamiento es bastante distinto al de los constructores. En este apartado se analizan los siguientes aspectos:

- A diferencia de lo que ocurre con los constructores, las llamadas al método `finalize` no se encadenan.
- La interacción de `finalize` con el recolector automático de basura puede tener efectos sorprendentes.

Una clase puede definir un método `finalize`, que será ejecutado justo antes de que el espacio de memoria ocupado por un objeto sea reclamado por el recolector automático de basura. En el método `finalize` se pueden liberar recursos asociados al objeto que está a punto de desaparecer, como ficheros abiertos, etc. En caso de que haya que realizar alguna finalización en la clase padre antes de que el objeto desaparezca definitivamente, hay que llamar explícitamente al método `super.finalize()`.

El método `finalize` se ejecuta automáticamente en el momento en el que el objeto ha quedado huérfano, esto es, no hay ninguna referencia que apunte a él. Sin embargo, el objeto puede resucitar precisamente en ese mismo momento, si en el método `finalize` se obtiene y almacena una referencia al objeto. Esto puede ocurrir, por ejemplo, si se asigna `this` a una variable estática en el cuerpo de `finalize`.

A partir de ese momento el objeto ya no es huérfano, por lo que no se reclamará su memoria hasta que no lo vuelva a ser. Pero llegado ese momento ya no se volverá a llamar a su método `finalize`. Dicho de otro modo, un objeto sólo puede resucitar una vez, salvo que en lugar de almacenar una referencia a `this` lo que hagamos sea clonar `this`. En este caso sí se llamaría al método `finalize`, pero al del clónico.

9.3 ¿Hay punteros en Java?

Es habitual la frase "en Java no hay punteros", que resulta algo curiosa cuando todos los objetos se manejan a través de referencias. De hecho podría ser más cierto afirmar que "en Java todo son punteros". Y eso tiene sus problemas:

Todo está en el montículo

No es posible declarar en Java objetos que residan en memoria estática o en la pila, lo que conlleva problemas de eficiencia. En sistemas críticos es usual que una vez inicializado el sistema no se utilice el montículo para crear nuevos objetos dinámicos, con el objetivo de obtener predicciones deterministas de tiempos de ejecución. Con Java esto no es posible: todos los objetos de tipos no básicos se crean dinámicamente en el montículo.

Por otra parte, aun siendo Java un lenguaje en el que todos los objetos de tipos no básicos residen en el montículo, resulta sorprendente que la operación de asignación de memoria dinámica no sea fácilmente redefinible por el usuario, como ocurre en lenguajes como C++ o Ada. Esto complica el uso de algoritmos de asignación de memoria dinámica, definidos por el programador en función de las necesidades de cada aplicación.

Problemas de la semántica basada en referencias

Al utilizar Java una semántica basada en referencias y no en valores para los objetos de tipos no básicos, los programas están expuestos a la aparición de alias dinámicos (*dynamic aliasing*): la posibilidad de que un objeto esté accesible mediante dos nombres diferentes en tiempo de ejecución, como consecuencia de asignaciones entre referencias.

Los alias dinámicos son peligrosos porque un objeto puede estar siendo modificado a través de un alias, no siendo esto evidente para el programador. Si quiere detectar todos los lugares en los que un objeto es modificado, el programador ha de tener siempre presente cuál es el conjunto de alias de un objeto dado.

Por el contrario, en Java la semántica para los tipos básicos está basada en valores y no en referencias: para los tipos básicos, la declaración, la asignación y la comparación se aplican a valores, mientras que para los tipos no básicos esas operaciones se aplican a las referencias: se asignan, se comparan y se declaran referencias.

Problemas de aprendizaje

Los estudiantes que utilizan Java como primer lenguaje tienen que tener clara la noción de dirección de un objeto desde el principio. Tradicionalmente este concepto se introduce una vez que se dominan otros elementos del lenguaje, pudiéndose aprender una gran parte del mismo sin entender la diferencia entre valor y dirección.

En Java, dado que la semántica de los tipos básicos está basada en valores y la de los no básicos está basada en referencias, el alumno ha de aprender las diferencias a la hora de declarar, copiar o comparar objetos (ver el apartado 9.3). Así mismo existen diferencias en cuanto al paso de parámetros (ver el apartado 9.1) que ha de entender el alumno para poder aprender adecuadamente los conceptos de función y procedimiento (métodos en Java).

Imposibilidad de restringir el modo de paso de parámetros a sólo lectura

En Java no se puede impedir que un método altere los objetos de tipos no básicos que se le pasen como parámetros.

Con los objetos de tipos básicos ocurre justo al contrario: no hay forma de que el método llamado altere los contenidos de los parámetros. De querer modificarlos, hay que encapsularlos en objetos de tipos no básicos, como ya se ha explicado en el apartado 9.1.

Los parámetros de los métodos se pueden especificar como `final`, lo que significa que en la implementación del método no se puede modificar el valor del parámetro real. Sin embargo, esto no resuelve gran cosa, pues al pasar un objeto como parámetro lo que se pasa es su referencia por copia. De estar declarado el parámetro como `final`, lo que no se puede hacer es modificar el parámetro real, esto es, la referencia, ¡pero sí el objeto apuntado por la referencia!

Si se quiere tener la garantía de que un objeto de un tipo no básico no resulta modificado, sólo hay una posibilidad: clonar el objeto (no basta hacer una copia de la referencia), y pasar como parámetro la referencia del nuevo objeto obtenido tras la clonación. La solución no es muy atractiva, entre otras razones por el decremento de eficiencia que conlleva.

En el apartado 9.1 se comentan otros aspectos del paso de parámetros.

9.4 Problemas del despacho dinámico

En Java la invocación de métodos siempre utiliza despacho dinámico[9]. Esto plantea problemas cuando se extiende una clase: puede haber código que ya estaba probado y que ahora, al heredarlo, deja de funcionar, aun cuando éste no ha sido modificado. Este grave problema se presenta cuando el código heredado hace llamadas a métodos que sí se han reemplazado.

A continuación se presenta un ejemplo de este problema. Se tiene una clase `T1`, con los métodos `A` y `B`, y el atributo `i`. La clase `T2` extiende `T1`, heredando el método `A`, y reemplazando el método `B` y el atributo `i`. Además, en la implementación de `A` se llama a `B`:

```
class T1 {
  int i = 1;
  public void A (){
    System.out.println (``i='' + i);
    B ();
  }
  public void B (){
    System.out.println
      (``En B de T1'');
  }
}
```

```
class T2 extends T1 {
  int i = 2;
  public void B () {
    System.out.println
      (``En B de T2'');
  }
}
```

En el método `main` de la clase `Main` se declaran dos objetos, `t1` y `t2`, de tipos `T1` y `T2` respectivamente. Luego se llama al método `A` de ambos objetos:

```
public class Main {
  static public void main (String [] argv ) {
    T1 t1 = new T1 ();
    T2 t2 = new T2 ();

    System.out.println (``>>>> t1.A()'');
    t1.A();
    System.out.println (``\n'' + ``>>>> t2.A()'');
    t2.A();
  }
}
```

[9]Salvo cuando se utiliza *super()*

Al ejecutar este programa se obtiene la siguiente salida en la pantalla:

```
>>>> t1.A()
i = 1
En B de T1

>>>> t2.A()
i = 2
En B de T2
```

Como en Java todas las llamadas despachan dinámicamente (salvo `super`), la llamada `t2.A()` acaba invocando al método `B` del tipo `T2`, y no al método `B` del tipo `T1`. En el caso del atributo `i`, en la llamada `t2.A()` se obtiene en la pantalla el valor 2, al reemplazarse el atributo de `T1` por el proporcionado por `T2`.

¿Cómo se puede forzar a que la llamada que se hace dentro de `A` quede ligada al método `B` de la clase `T1` en lugar de despachar dinámicamente? ¿Y si se quiere que el atributo `i` que se imprime en el método `A` sea siempre el de la clase `T1`?

Para responder a ambas preguntas se modifica el método `A`, para forzar la conversión a tipo `T1` del parámetro implícito `this` tanto en la llamada a `B` como en el acceso al atributo `i`:

```
class T1 {
  int i = 1;

  public void A (){
    System.out.println (``i='' + ((T1)this).i);
    ((T1)this).B ();
  }
  public void B (){
    System.out.println
      (``En B de T1'');
  }
}
```

La salida obtenida ahora en pantalla es la siguiente:

```
>>>> t1.A()
i = 1
En B de T1

>>>> t2.A()
i = 1
En B de T2
```

Como puede apreciarse, en el caso del atributo `i` la conversión del tipo de `this` sirve para conseguir el efecto buscado: en nuestro ejemplo, la llamada `t2.A()` acaba imprimiendo el valor 1 del campo `i` del tipo `T1`, como era de esperar.

Pero, sorprendentemente, con los métodos el comportamiento es distinto: a pesar de la conversión explícita de `this`, la llamada a `((T1)this).B()` dentro de `A` despacha dinámicamente, y es el método `B` de `T2` el que se ejecuta.

Este problema es grave, pues complica enormemente la reutilización del código:

- El programador que creó `T1` con los métodos `A` y `B` nunca puede diseñar `A` de forma que la llamada a `B` funcione adecuadamente para cualquier posible reemplazamiento que se haga en una futura extensión de `T1`. Tampoco podría escribir un código que forzara a que esa llamada a `B` no despachara nunca[10].

- El programador que cree `T2` extendiendo `T1` puede que no disponga de la implementación de `T1`, por lo que puede resultarle difícil saber que el código de `A` llama a `B`, y que puede dejar de funcionar si se reemplaza dicha `B` en `T2`.

Para realizar un *software* que no padezca estos problemas en Java hay dos alternativas:

- Declarar la clase `T1` o el método `B` de `T1` como `final`, de forma que el método `B` de `T2` no pueda declararse. Esto coarta la extensibilidad del software.

- Volver a probar la biblioteca reutilizada (en este caso `T1`). Las desventajas de esta alternativa son evidentes: coste de desarrollo incrementado, posibilidad de errores por pruebas no exhaustivas, etc. Para paliar esta situación pueden emplearse soluciones no basadas en el lenguaje: el programador de `T1` debería documentar que en `A` se realiza una llamada al método `B`, o bien entregar el código fuente de su biblioteca.

9.5 Las clases internas

Las clases internas fueron introducidas en Java a partir de la versión 1.1. Se trata de proporcionar una construcción que facilite la codificación en todas aquellas situaciones en las que se precise pasar como parámetro un método a otro para que este último haga algo invocándolo. Este problema se resuelve típicamente con punteros a funciones en lenguajes como C, C++ o Ada. En cambio, y dado que en Java un método no es un un objeto de un tipo del lenguaje (ni básico ni referenciado), la solución adoptada en este lenguaje pasa por la creación de una clase auxiliar con el método en cuestión y pasar esta clase como parámetro. Podría haberse adoptado alguna solución alternativa como la propuesta por Pizza[11].

Al utilizar clases cuyo único objetivo es ser contenedores de los métodos que se quieren pasar como parámetro, el código se vuelve oscuro: la farragosa sintaxis necesaria en Java 1.0 y la proliferación por el código de estas clases auxiliares mezcladas con las clases normales llevó a los diseñadores del lenguaje a introducir las clases internas.

Las clases internas facilitan, en comparación con la situación previa a Java 1.1, la escritura de programas que requieran la funcionalidad proporcionada por los inexistentes punteros a métodos. Pero la introducción de esta construcción ha tenido su precio:

- Se ha necesitado realizar nuevos cambios en otros aspectos del lenguaje para eliminar problemas o ambigüedades con el uso de las clases internas. Ninguno de dichos cambios era necesario antes de la introducción de las clases internas, con lo que se complica innecesariamente el lenguaje.

[10] Nótese que en C++ las llamadas a los métodos se despachan dinámicamente por defecto, pero puede forzarse a que no lo hagan.

[11] Pizza es una extensión a Java que proporciona funciones similares a los bloques de Smalltalk, que pueden ser pasadas como parámetros de métodos, almacenadas en variables, o devueltas como resultado de otra función. Pizza aporta también tipos genéricos, otra carencia notable de Java. Pizza está disponible en http://www.cis.unisa.edu.au/~pizza/.

- Las posibles utilizaciones sintácticamente correctas de las clases internas permiten escribir código poco intuitivo y escasamente legible.

Estos dos aspectos se analizan por separado a continuación.

Nuevos cambios que se han hecho necesarios

1. Nueva sintaxis para `this`

   Para poder referirse de forma explícita dentro de una clase interna a la clase que la contiene (y/o a sus miembros) se utiliza la sintaxis `clase_contenedor.this`[12].

   Un ejemplo:

```
public class A {
  int j = 1;

  public class B {
    int k = 2;

    public void printValues () {
      System.out.println(k);
      System.out.println(this.k);    // igual que siempre
      System.out.println(A.this.j);  // nueva sintaxis
    }
  }
}
```

   Nótese que esta sintaxis es genérica y puede utilizarse incluso con clases no internas.

2. Nueva sintaxis para `new`

   Como cada instancia de una clase interna debe tener asociada una instancia de la clase contenedor, al construir con `new` una instancia de una clase interna es necesario extender la sintaxis para indicar cuál es la instancia de la clase contenedor asociada. Esta nueva sintaxis es `instancia_contenedor.new`:

```
A a = new A();
B b = a.new B();
```

3. Nueva sintaxis para `super`

   Como nada impide que una clase de nivel máximo (no interna) pueda extender a una clase interna, es necesario que en el constructor de la clase derivada se llame al constructor de la interna de que deriva, pasándole información de qué instancia contenedor tiene asociada. Para ello se habilita una nueva sintaxis para `super`, consistente en `instancia_contenedor.super`:

[12] Como se verá en ejemplos posteriores, esta sintaxis puede usarse en Java 1.1 incluso con una clase no contenedor como prefijo de *this*.

```
public class C extends B {

  public C (A a) {
    a.super();
  }
}
```

En este ejemplo se reutilizan las clases `A` y `B` del ejemplo anterior.

4. Más usos del modificador `final`

   Las clases internas también pueden declararse localmente a un bloque de código Java. Esto permite a dichas clases acceder también a las variables y parámetros locales a ese bloque. Ahora bien, detalles de implementación de las clases internas hacen que ese acceso no sea seguro ante la presencia de otros hilos (*threads*) que estén accediendo concurrentemente a esos datos. Por ello en Java 1.1 se puede aplicar el modificador `final` también a variables locales y parámetros (en Java 1.0 sólo se podía aplicar a atributos, métodos y clases). A las clases internas declaradas localmente dentro de un bloque de código sólo se les permite acceder a estos datos que ya no pueden cambiar su valor.

5. Nuevos bloques de inicialización de instancia

   Las clase internas también pueden declararse de forma anónima dentro de una expresión, lo que constituye el máximo atajo sintáctico a la hora de construir una clase auxiliar de un sólo uso. El principal inconveniente de esta construcción es que no se permite que estas clases internas anónimas tengan constructor. Por ello se generaliza el concepto de bloque de inicialización de clase ya existente en Java 1.0 y se permite colocar dentro de la definición de cualquier clase bloques de código que, si no van precedidos de `static`, se ejecutarán cada vez que se cree una nueva instancia. Es decir, son igual que un constructor sin parámetros, pero pueden usarse para las clases internas anónimas (ver el apartado 9.2).

6. Las clases internas no pueden tener miembros `static`

   Nada impediría que las clases internas (como cualquier otra clase) pudieran contener miembros `static`. Pero como tal uso en ningún caso es imprescindible (siempre podrían colocarse dichos miembros en la clase contenedor) y permitiría codificaciones sumamente oscuras, los diseñadores decidieron prohibir dichos miembros `static` en las clases internas.

Construcciones peculiares

Las clases internas introducen el concepto de "inclusión" de manera independiente y ortogonal al de "herencia". Así, además de una jerarquía de clases ordenadas por relaciones de herencia aparece ahora otra jerarquía de clases incluidas unas dentro de otras. Ambas jerarquías pueden además entremezclarse a voluntad.

Esto conduce a la posibilidad de escribir código perfectamente legal pero de oscuro significado y comportamiento. A continuación se muestran algunos ejemplos:

1. Clase contenedor que contiene como miembro a una clase interna suya

```
public class A {
  int j = 1;
  B b = new B ();

  public class B {
    int k = 2;

    public void printValues () {
      System.out.println(k);
      System.out.println(this.k);      // lo mismo
      System.out.println(A.this.b.k);  // lo mismo
      System.out.println(j);           // acceso a j por inclusión
//    System.out.println(this.j);      // ilegal, B no tiene una j
      System.out.println(A.this.j);    // acceso a j por inclusión
    }
  }
}
```

2. Clase interna que extiende a una clase de nivel máximo distinta de su clase contenedor

```
public class A {
  int j = 1;
  B b = new B ();

  public class B extends C {
    int k = 2;

    public void printValues () {
      System.out.println(k);
//    System.out.println(j);        // ilegal, hay que explicitar
      System.out.println(this.j);   // acceso a la j heredada de C
      System.out.println(C.this.j); // lo mismo, es legal
      System.out.println(A.this.j); // acceso a la j de A
    }
  }
}

public class C {
  int j = 3;
}
```

3. Clase interna que extiende a su propia clase contenedor

   Este caso es especialmente peculiar. Se propone al lector que antes de continuar intente predecir cuál debe ser la salida del siguiente programa:

```
public class A {
```

```
  public int j = 1;
  public int k = 0;

  public class B extends A {
    public void printValues () {
      j++;
      System.out.println(j);
      System.out.println(A.this.j);
      System.out.println(A.this.k);

    }
  }
}


public class Main {
  public static void main (String [] argv )  {
    A a = new A ();
    A.B b = a.new B ();
    b.printValues ();
    System.out.println (a.j);

  }
}
```

La salida que da el programa es `2 2 0 1`. Es decir, las dos referencias a `j` en `printValues()` se refieren a la `j` heredada (que se incrementa en 1), mientras que el `1` final corresponde a la `j` de la instancia contenedor, que aún sigue valiendo 1.

Nótese lo sorprendentes que resultan los siguientes aspectos:

- no es posible referirse en `printValues()` a la `j` de la instancia contenedor
- la sintaxis `A.this.k` sí se refiere a la `k` de la instancia contenedor, mientras que `A.this.j` hace referencia a la `j` heredada en B

9.6 Parametrización en tiempo de compilación: ausencia de genéricos

Una de las ausencias notables de Java es el soporte de genéricos, presentes en lenguajes como C++ (*templates*) o Ada (genéricos). Los genéricos son útiles para realizar bibliotecas de componentes reusables, como contenedores de elementos de diversos tipos.

En Java la ausencia de genéricos se puede suplir utilizando contenedores (como vectores o listas) cuyos elementos son de un tipo básico (generalmente Object), de forma que cualquier elemento que sea de un subtipo de aquél pueda almacenarse en el contenedor. Sin embargo esta solución implica múltiples promociones de tipos en tiempo de ejecución, con la consiguiente merma de rendimiento al comprobar compatibilidades de tipos. Con los genéricos estas comprobaciones se pueden realizar en tiempo de compilación, obteniendo programas más rápidos y robustos.

Existen múltiples propuestas para añadir genéricos a Java, y es posible que el lenguaje acabe incorporando este útil mecanismo. El propio creador del lenguaje, James Gosling, prevé la próxima incorporación de genéricos al lenguaje.

Una de las propuestas actuales es GJ, que extiende el lenguaje Java para incluir tipos y métodos genéricos[Bracha *et al.*, 1998] y se puede consultar en WWW[13].

9.7 Aspectos relacionados con la concurrencia

La interfaz de la clase Thread

El método `run` de la clase `Thread` está definido como `public`. Cuando se extiende esta clase y se reemplaza el método `run` no se puede restringir su visibilidad haciéndolo, por ejemplo, `private`.

Por lo tanto el método `run` de un objeto *thread* puede ser llamado desde cualquier parte del código que tenga visibilidad de la clase a la que pertenece el objeto *thread*. Esto no es lógico, pues este método está pensado para que sea llamado por el sistema, una vez que se activa el *thread* asociado a la clase.

Dispersión del código sincronizado

Las sentencias `synchronized` se introdujeron en Java para poder utilizar el monitor asociado a un objeto de una clase que no tiene métodos `synchronized`:

```
synchronized (o) {
 // código protegido
}
```

Esta construcción puede utilizarse también con objetos de clases que sí tienen métodos `synchronized`. Por lo tanto, el código de sincronización ligado a un monitor puede aparecer disperso en el código. No basta con ver el código de los métodos `synchronized`, sino que hay que buscar todas las sentencias `synchronized (o)`. En caso de existir alias del objeto la búsqueda puede resultar aún más complicada.

Los monitores de Java no tienen variables condición

En otros lenguajes de programación los monitores se utilizan junto con el mecanismo de las variables condición. Cada variable condición puede utilizarse para que se bloqueen en ella los *threads* que no cumplen una condición determinada. A la hora de llamar a `wait()` (o la sentencia equivalente en cada lenguaje) se especifica en qué variable condición se quiere quedar bloqueado el *thread* llamante. Cuando la condición por la que esperaban los *threads* que están bloqueados en una variable condición se cumple, otro *thread* puede despertarlos: al llamar a `notify()` (o la sentencia equivalente en cada lenguaje) se especifica la variable condición.

En Java sólo existe una variable condición asociada a cada monitor, por lo que al llamar a `wait()` o a `notify()` no hace falta explicitar su nombre (de hecho no se puede referir en el código).

[13] http://www.cis.unisa.edu.au/~pizza/gj/

En esa misma variable condición puede haber varios *threads* bloqueados, cada uno de ellos esperando por condiciones distintas. Cuando otro *thread* compruebe que alguna de las condiciones se cumple puede despertar a alguno de los *threads* bloqueados ejecutando `notify()`. Pero puede ocurrir que el *thread* despertado no esté esperando por esa condición. Para paliar este problema en Java se utiliza el método `notifyAll()` en lugar de `notify()`. Llamando a `notifyAll()` el *thread* llamante despierta a todos los *threads*. Se está por tanto gastando un tiempo innecesario, pues cada uno de los *threads* despertados tiene que ejecutar código para comprobar si es su condición la que se cumple, y si no volver a bloquearse llamando a `wait()`. Además no está acotado el tiempo que puede tardar en entrar el *thread* para el que se cumple la condición.

En lugar de utilizar `notifyAll()` se pueden simular las variables condición existentes en otros lenguajes de programación utilizando un monitor extra para cada una de ellas, además del "verdadero monitor":

```
class VariableCondición {
  Boolean deboDormir = false;
};

class VerdaderoMonitor {
  VariableCondición c1 = new VariableCondición ();
  VariableCondición c2 = new VariableCondición ();

  public void unMetodo () {

    // Adquirimos primero el monitor de la variable condición
    synchronized (c1) {
      // Ahora adquirimos el monitor
      synchronized (this) {
        // Sólo ahora estamos realmente dentro del monitor,
        // y podemos consultar o actualizar el estado protegido

        if (condicion2)
          c2.notify ();

        if (not condicion1)
          c1.deboDormir = true;
      }

      if (c1.deboDormir)
        c1.wait ();

    }
  }
}
```

Como se puede apreciar en el ejemplo, los *threads* nunca se quedan bloqueados dentro del "verdadero monitor", sino dentro del monitor de la variable condición (cuando se ejecuta `c1.wait()` en el ejemplo). Sin embargo, la condición hay que comprobarla cuando está dentro

del "verdadero monitor", pues se necesita que otros *threads* no estén modificando el estado mientras que se comprueba la condición. Por ello se anota en la variable `deboDormir` si la condición se cumplió o no. Si la llamada a `c1.wait()` se hiciese cuando está dentro del "verdadero monitor", ningún otro *thread* podría pasar al "verdadero monitor" a despertarnos.

Al tener ahora que adquirir siempre dos monitores, esta solución puede plantear problemas de rendimiento. A cambio se obtiene la ventaja de poder despertar a un *thread* que espera por una condición determinada, sin despertar a los demás como ocurre cuando se utiliza `notifyAll()` (en el ejemplo, `c2.notify()`).

Al no estar soportadas en el lenguaje las variables condición hay que recurrir a técnicas como ésta, cuyo uso no es sencillo, pudiendo tener consecuencias catastróficas en caso de usarse mal (interbloqueos, inanición,...).

Falta de equidad en el acceso a monitores

Los *threads* que son despertados de un `wait()` tienen que competir por la adquisición del monitor con otros *threads* que en ese momento estuviesen intentando acceder a través de métodos o sentencias `synchronized`.

Esto hace que la implementación de políticas equitativas de acceso al monitor no sea inmediata. En otros lenguajes de programación que tienen monitores, cuando se despierta a un *thread* que ya estaba esperando en un `wait()`, éste tiene prioridad sobre los que esperan para entrar en el monitor, no teniendo que competir con ellos para adquirir el monitor.

Mezcla de métodos sincronizados y no sincronizados

En Java se pueden mezclar en una misma clase métodos `synchronized` con otros que no lo son. En las instancias de este tipo de clases puede haber varios *threads* ejecutando código a la vez: si bien sólo uno de ellos puede haber entrado[14] a través de alguno de los métodos o sentencias `synchronized`, a través del resto de métodos podría haber entrado un número cualquiera de *threads*.

Usualmente el mecanismo del monitor en otros lenguajes no permite esta flexibilidad. Utilizando la terminología de Java, en estos lenguajes todos los métodos de un monitor son `synchronized`.

La flexibilidad que aporta Java al permitir la mezcla de métodos `synchronized` con otros que no lo son puede ser fuente de frecuentes errores de programación. Una de las ventajas de los monitores respecto a otros mecanismos de sincronización como los semáforos es el mayor nivel de abstracción que proporcionan aquéllos. La exclusión mutua se obtiene de manera implícita al utilizar un monitor, mientras que si se utilizan semáforos hay que implementarla explícitamente, por lo que aumenta la probabilidad de equivocarse al programar. La posibilidad de mezclar métodos `synchronized` con otros que no lo son convierten a los monitores de Java en un mecanismo a medio camino entre los semáforos y los monitores de otros lenguajes, en lo que a posibilidad de cometer errores por parte del programador se refiere.

Podría pensarse que esta flexibilidad permite implementar esquemas de sincronización útiles como lectores/escritor etc. Sin embargo no es así: si hay dentro del monitor *threads* (lectores) que han entrado a través de métodos no `synchronized`, no se prohibe la entrada a un nuevo *thread* (escritor) a través de uno de los métodos `synchronized`.

[14] Entrar en el monitor equivale a adquirir la exclusión mutua del monitor asociado a la instancia de la clase

Si los métodos que no son `synchronized` no se hacen públicos, o bien no se escriben métodos no `synchronized` en una clase que se vaya a utilizar como monitor, este problema desaparece. Pero cualquiera de estas soluciones requiere una disciplina por parte del programador, no contando con la ayuda del lenguaje.

Un problema similar afecta también al monitor que cada clase tiene asociado en Java. Pueden utilizarse métodos *static synchronized* o sentencias `synchronized (getClass())` para acceder a datos protegidos por el monitor de la clase. Pero nada impide acceder a esos mismos datos sin sincronizarse en el monitor de la clase.

9.8 Ausencia de estándar del lenguaje Java

La ausencia de un estándar puede ser una de las amenazas más importantes para la tecnología Java, y especialmente para el lenguaje. La experiencia del proceso de estandarización del lenguaje C++ (no ha habido un estándar ISO hasta noviembre de 1997) debería bastar para comprender los problemas asociados a la ausencia de un estándar: múltiples versiones incompletas de elementos del lenguaje, herramientas de desarrollo continuamente modificadas, etc. El lenguaje Java ha venido padeciendo este tipo de problemas hasta la fecha: el lenguaje ha estado en constante evolución, por lo que no existe una definición completa, detallada y estable de Java.

Los primeros pasos del proceso de estandarización ISO de la plataforma Java los inició Sun Microsystems Inc, en su calidad de organización que desarrolla y posee las especificaciones públicas de la tecnología, en marzo de 1997. Sun Microsystems mantiene unas páginas en WWW con información relativa al proceso de estandarización[15].

[15] http://www.javasoft.com/aboutJava/standardization/index.html

10 Consideraciones finales

Bajo la denominación Java se agrupa toda una serie de tecnologías del mundo del software como la programación independiente de plataforma, el soporte en el lenguaje para la programación de sistemas concurrentes, distribuidos, gráficos, etc. muchas de las cuales se han descrito a lo largo de este artículo.

Es posible que esta integración sea uno de los mayores méritos del equipo que desarrolla Java y la justificación del éxito alcanzado (aparte del gran soporte publicitario de Sun). Prácticamente ninguna de estas tecnologías es novedosa: la portabilidad existía en muchos otros lenguajes interpretados, lenguajes como Ada tienen desde hace tiempo integrada la programación de sistemas concurrentes y distribuidos, etc. Sin embargo, ninguno de ellos las integraba todas.

Por otra parte, el hecho de que Java haya sido diseñado por una sola compañía (más concretamente por un pequeño equipo de desarrolladores) supone una serie de problemas de diseño, algunos de los cuales se han analizado en la sección 9, que en otros lenguajes creados después de un estudio riguroso de la comunidad informática no existen.

En resumen, bienvenido sea Java por haber popularizado definitivamente conceptos como los programas distribuidos, la orientación a objetos, etc. pero mucho cuidado al afirmar que es la panacea de los lenguajes de programación.

11 Glosario

***7:** Primer dispositivo de control basado en Java.

API: Acrónimo en inglés correspondiente a *Application Program Interface.*

AWT: *Abstract Windowi Toolkit.* Conjunto de clases Java para la realización de interfaces de usuario basados en ventanas, botones, etc.

Applet: Pequeño programa escrito en Java que se ejecuta en un navegador web.

Green: Proyecto de investigación de Sun dedicado a la investigación en electrónica de consumo del que surgió Java.

GreenOS: Sistema operativo resultado del proyecto *Green* del que son herederos el sistema operativo JavaOS y la propia máquina virtual de Java.

HotJava: Navegador web escrito en Java, desarrollado por Sun.

Java: Lenguaje de programación basado en las ideas de la orientación a objetos, interpretado, portable y que soporta la interacción entre objetos remotos.

JavaBeans: Plataforma de desarrollo de componentes de software reutilizables y que pueden manejarse con una herramienta gráfica.

JavaDoc: Herramienta desarrollada por Sun para la documentación del código Java. Es capaz de generar comentarios a partir del código fuente de las clases.

JavaOS: Sistema operativo diseñado originalmente para ejecutar programas en Java en pequeños dispositivos como teléfonos, televisores, etc.

Jini: Es un proyecto de investigación de Sun, basado en tecnología Java, que permitirá conectar a la red multitud de dispositivos de forma transparente.

JDBC: *Java DataBase Connectivity.* Es el API desarrollado por Javasoft para conectar programas en java y applets con servidores de bases de datos que implementan el estándar SQL.

JDK: *Java Development Kit.* Es el conjunto de herramientas que Sun proporciona a los desarrolladores de aplicaciones Java. Está compuesta entre otras herramientas por un compilador, una máquina virtual, un conjunto de clases, un depurador, etc.

JFC: *Java Foundation Classses.* Conjunto de clases realizado por IBM, Sun y Netscape para facilitar la interacción de Java con algunos sistemas operativos existentes.

JIT: Compilador *Just-In-Time.* Compilador específico para una determinada plataforma que traduce directamente el código Java a instrucciones máquina, sin necesidad de interpretarlo.

JNI: *Java Native Interface.* Mecanismo previsto en Java para realizar llamadas a métodos escritos en otros lenguajes de programación.

Oak: Lenguaje de programación desarrollado para el proyecto *Green* que posteriormente se modificó para ser Java.

ODBC: Protocolo estándar de acceso a base de datos desde Java.

PDA: *Personal Digital Assistant.* Ordenador de bolsillo usado generalmente como agenda.

RMI: *Remote Method Invocation* Versión Java de los protocolos de llamada a procedimientos remotos. Es decir, el mecanismo por el que un programa puede llamar a métodos de un objeto que se está ejecutando en una máquina remota.

Servlets: Programas que se ejecutan en un servidor permitiendo a las aplicaciones clientes ejecutar código en el propio servidor.

Virtual Machine: Programa que permite ejecutar otro programa. De esta forma se puede hacer que un ordenador se comporte como otro de arquitectura distinta.

WebRunner: El primer navegador Web escrito en Java, precursor de HotJava.

Referencias

[Arnold and Gosling, 1998] Ken Arnold y James Gosling. *The Java Programming Language Second Edition.* Addison Wesley, 1998.

[Bishop, 1997] Judy M. Bishop. *Java Gently.* International Computer Science Series. Addison Wesley, 1997.

[Bracha *et al.*, 1998] Gilad Bracha, Martin Odersky, David Stoutamire, y Philip Wadler. Making the Future safe for the Past: Adding Genericity to the Java Programming Language. En *Proceedings of the Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98) Conference*, Vancouver, Canada, Octubre 1998.

[Brosgol, 1997] Benjamin M. Brosgol. Java for Ada Programmers. Tutorial Notes. Tri-Ada'97 Conference., Noviembre 1997.

[Deitel and Deitel, 1998] Harvey Deitel y Paul Deitel. *Cómo programar en Java.* Prentice Hall, 1998.

[Farley, 1997] Jim Farley. *Java Distributed Computing.* O´Reilly, 1997.

[Flanagan, 1997] David Flanagan. *Java in a Nutshell 2nd edition.* O´Reilly, 1997.

[Harold, 1997] Elliotte Rusty Harold. *Java: network programming.* O'Reilly, 1997.

[Lea, 1996] Doug Lea. *Concurrent Programming in Java. Design Principles and Patterns.* Addison Wesley, 1996.

[Sun, 1997] Sun Microsystems, Inc., Mountain View, CA, EE.UU. *Java Remote Method Invocation Specification*, Octubre 1997. Revision 1.42, JDK 1.2 Beta1.

[Taft, 1997] S. Tucker Taft. High Integrity Object-Oriented Programming with Ada 95. Keynote Speech. Tri-Ada'97 Conference., Noviembre 1997.

[van Hoff *et al.*, 1996] Arthur van Hoff, Sami Shaio, y Orca Starbuck. *Hooked on Java.* Addison Wesley, 1996.

[Vanderburg et al., 1997] Glenn Vanderburg et al. *Maximum Java 1.1.* Sams.net Publishing, 1997.