Manual de Visual J++

Visual J++, integrada en Visual Studio, es la plataforma de desarrollo que Microsoft propone a los programadores para la creación de sus aplicaciones en el lenguaje Java. En este curso nos proponemos presentar esta plataforma y las funcionalidades que añade el nuevo SDK 2.0

El lector está a punto de iniciar la primera entrega de este curso de Visual J++ 1.1 que PC Actual, con la colaboración de HISPAN TECNOLOGIC, se dispone a publicar a lo largo de seis números de la revista. Puede parecer una osadía atreverse a iniciar la publicación de un curso de Visual J++ con la que está cayendo. Está granizando café, nunca mejor dicho, pero no en el campo, sino sobre los despachos y los palacios de justicia. La demanda de Sun contra Microsoft puede tentar al lector a guarecerse a cubierto a la espera de que pase la tormenta, y dedicar sus esfuerzos a lenguajes y plataformas de desarrollo que naveguen por mares más plácidos. Quizá estoy colocándome en una posición demasiado dramática, y el lector no haya percibido los rumores, o el estruendo, según se mire, que la frenética actividad de los bufetes de abogados de Microsoft y Sun está provocando, pero esa posibilidad se me antoja poco probable.

¿Qué es *Java?*

Java, Java, Java. Originalmente el equipo de *Sun Microsystems* que en los primeros 90 desarrolló *Java* se planteó la posibilidad de desarrollar *software* que pudiese utilizarse en electrodomésticos con componentes electrónicos diversos, tales como equipos de vídeo o de audio. En los electrodomésticos se producen continuas renovaciones de *hardware,* bien por motivos comerciales que conlleven el deseo de la actualización del producto, bien por las prosaicas pero siempre prioritarias motivaciones de costos.

Este hecho inspiró el proyecto de desarrollo de un lenguaje independiente de la que en primera instancia recibió el ecológico nombre de *Oak* (Roble), simplemente porque desde el despacho del equipo de creación del lenguaje podía verse uno de estos árboles. El lector caerá rápidamente en la cuenta de que la historia de *Java* está trufada de casualidades bien aprovechadas. El objetivo del equipo era conseguir evitar tener que desarrollar en C o C++ por la sencilla razón de que estos lenguajes obligan a recompilar el código para cada nueva plataforma en la que quiera implantarse.

Todos aquellos que hayan desarrollado o reflexionado en torno a la problemática de crear contenidos en *WWW* habrán pensado alguna vez en la naturaleza heterogénea de la red y los problemas que ello provoca a la hora de desarrollar programas. Si *Java* había nacido como un lenguaje que podía ejecutarse en cualquier máquina, qué podía ser más apropiado para un entorno, el *web,* en el que jamás puedes saber qué equipo, ni tan siquiera qué sistema operativo estará utilizando el usuario de tu futuro módulo de *software.* Esta universalidad, que conlleva obvias repercusiones económicas en el ahorro de las empresas de desarrollo, se expresa en el adagio de *Sun "Write Once, run anywhere",* algo así como "escribe tu programa una vez, y ejecútalo en cualquier parte".

La independencia de la plataforma se expresa en que los programas *Java* no necesitan ser recompilados cuando se desea cambiar de plataforma porque los *ejecutables Java,* a diferencia de los de la mayoría de los lenguajes compilables, no están formados por un conjunto de instrucciones en el código máquina del procesador sobre el que serán ejecutados, sino que

contienen lo que la terminología *Java* se denominan *bytecodes***,** parte de un juego ficticio de instrucciones independiente del procesador.

Para conseguir esta independencia de la plataforma, *Java* propone la definición de una máquina *virtual* sobre la que se ejecutará en realidad cualquier programa que se desarrolle en este lenguaje, sin importar la *CPU* ni el sistema operativo que la máquina esté utilizando. Esta máquina virtual es la que ejecuta los anteriormente mencionados *bytecodes*, al mismo tiempo que proporciona al lenguaje gran parte de los servicios que usualmente son una funcionalidad exigible al sistema operativo.

Bien, hemos llegado al quid de la cuestión. El lector avezado resoplará, ¡otro lenguaje interpretado! Sí, *Java* no es un lenguaje compilado a código fuente nativo, como pueda serlo C++ o Pascal, pero tampoco se trata de un lenguaje de macros. El entorno de desarrollo *Java* funciona en buena media como si utilizase un lenguaje compilable, salvo porque el compilador traduce el programa fuente *Java* a *bytecodes* en vez de a código máquina, y que estos *bytecodes* no se ejecutan directamente en la *CPU,* sino mediante la máquina virtual, que hace las funciones de intérprete.

Java como lenguaje de programación

Java es un lenguaje de programación orientado a objetos que, en buena medida, proviene de C y C++, y por tanto hereda la mayoría de su sintaxis y mecanismos de orientación a objetos. Incluye una librería de clases, que proporciona una base para posteriores desarrollos, los tipos básicos, las esenciales funcionalidad de entrada/salida y de trabajo en red, y los protocolos *Internet* más comunes.

Sin embargo, a pesar de que *Java* resulta similar a C++, tiene la agradable vocación de ser sencillo y carece de los aspectos más complejos de este lenguaje. Como ejemplo basta reseñar que no existen punteros y sí cadenas de caracteres *(strings)* o matrices *(arrays)* como auténticos objetos o que el manejo de memoria resulta en buena medida automático. Es decir, *Java* presume de ser un lenguaje de aprendizaje sencillo. Bien, esto es cierto, sobretodo si ya sabes programar en C++. Vamos a enunciar, siquiera brevemente, algunos aspectos reseñables del lenguaje.

Clases e interfaces

Java agrupa las clases en librerías que se llaman paquetes (*package*s). Los paquetes también pueden contener definiciones de interfaces. Además, *Java* permite la creación de clases abstractas.

Por otra parte, *Java* crea de manera explícita la entidad denominada interfaz, y que no es más que es una colección de declaraciones de métodos, sin implementación que representan una suerte de contrato entre el desarrollador que crea una clase que implementa una interfaz y los que reutilicen su código Las interfaces se utilizan para definir un tipo de comportamiento que puede ser implementado por cualquier clase y resultan útiles para capturar similitudes entre clases que no están relacionadas, sin forzar una relación directa, revelar la interfaz de programación de un objeto sin revelar su clase o para reflejar el comportamiento y la naturaleza de los objetos componentes reutilizables que, por ejemplo, se definen en COM.

Liberación de memoria

Java es *Garbage Collected*. El sistema, a la hora de ejecución, mantiene un control de la memoria que actualmente se está utilizando y de la que está libre. Cuando una porción de memoria ya no se está utilizando por el programa, el sistema la devuelve. Esta característica resulta importante porque en los lenguajes orientados a objetos es común que existan métodos que se encarguen de destruir el objeto que se está utilizando (destructores), algo que en *Java* no tiene un sentido tan estricto.

Soporte de excepciones

Java incorpora las excepciones como partes esenciales del lenguaje. Una excepción es una condición de error que se supone no debe ocurrir al tiempo de ejecución. Por ejemplo, en una división por cero, el programa se detiene si la situación no se evita por parte del programador. Un buen programador trata de evitar que esto suceda y muchas veces hay que añadir muchas validaciones para lograrlo. Es muy incómodo que un usuario tenga que volver a empezar el programa por algún error de ejecución, y ello debe evitarse, por ejemplo, indicando cual fue el error y como remediarlo.

Multitarea

El concepto de *multithreading* puede resumirse de manera grosera en que podemos realizar más de una cosa al mismo tiempo. Esto quiere decir que mi programa puede realizar más de una operación en un mismo momento. Para hacer esto se debe dividir el programa en diferentes caminos de ejecución, los *threads*. *Java* proporciona soporte completo para esta técnica de programación.

¿Qué podemos desarrollar con *Java*?

Java es un lenguaje de programación de propósito general y, en contra de una creencia bastante extendida, su uso no se limita al *WWW*. De hecho, existen aplicaciones completas en *Java,* y proyectos (algunos de ellos en revisión) para desarrollar versiones de suites de aplicaciones en este lenguaje. Sin embargo, el uso más común está relacionado con *WWW* y no es otro que el desarrollo de *applets*. Un *applet* es un pequeño programa interactivo que se ejecuta en el seno de una página *Web*. Pueden utilizarse para presentar animaciones, figuras, juegos, para responder a acciones del usuario, o cualquier otra tarea. El proceso para incluir un *applet* en una página *Web* consiste en escribir el código *Java,* compilarlo, e insertar el *applet* utilizando el lenguaje *HTML*.

La disputa legal

Llegado este punto, en el que el lector posee una visión general de qué es *Java,* qué puede ofrecernos y cuales son sus puntos fuertes y sus debilidades, vamos a retomar el inicio de este artículo e intentaremos arrojar algo de luz sobre la situación. Esto es necesario porque el objeto de este curso es presentar al lector la plataforma de *Microsoft* para el desarrollo de aplicaciones en *Java: Visual J++,* y antes de hacerlo resulta poco menos que imprescindible asegurar al lector que *Visual J++* es un entorno con presente y futuro garantizados.

En los ámbitos de discusión de desarrolladores está muy extendida la idea de que *Microsoft* está de algún modo boicoteando la expansión y afianzamiento de *Java*. Los que sustentan esta teoría argumentan que *Microsoft* está adoptando una táctica poco menos que de caballo de Troya, es decir, está entrando con fuerza en el mercado de *Java*, para desnaturalizar el

lenguaje, acercándolo a la plataforma *Windows,* y haciéndole perder en el proceso su característica quizá más reseñable: la independencia de la plataforma.

Como hemos comentado anteriormente, *Java* requiere para su ejecución la existencia en el ordenador en cuestión de un intérprete, la máquina virtual *Java.* El lector que visione *applets Java* en sus exploraciones del *Web* quizá no sea consciente de que en su ordenador tiene instalada una máquina virtual. ¿Cómo llegó allí? ¿Puede ese ente rebelarse contra nosotros? No sufráis, probablemente el intérprete *Java* llegó allí durante la instalación del explorador, supongamos *Microsoft Internet Explorer.* Es lógico, si un explorador soporta *Java,* su instalación debe de una u otra manera incluir la máquina virtual. Esta máquina virtual no es inocente, tiene padre y madre. Si el intérprete ha sido instalado durante la ejecución de *Microsoft Internet Explorer,* esa máquina virtual habrá sido desarrollada por *Microsoft.*

¿Dónde está *Sun*? Hace rato que no hablamos de ellos. Bien, *Sun, (*o *Javasoft* tanto monta, monta tanto) se encarga, o eso se supone, de garantizar que *Java* sigue fiel a sus características y filosofía iniciales. A medida que la informática va evolucionando, el lenguaje también lo hace, añadiendo nuevas funcionalidades, que deben ser incorporadas por las nuevas máquinas virtuales. Y cuando sigo deben, me refiero al punto de vista de *Sun,* respetable en cuanto a padre de *Java.* Estas nuevas versiones de los elementos que conforman el entorno de *Java,* se agrupan en los denominados *Kits de Desarrollo de Java (Java Development Kit).* La versión actual de *JDK* es la 1.1.

No todas las máquinas virtuales soportan las mismas funcionalidades, y además las empresas luchan por hacer que la suya sea la más rápida, la más novedosa tecnológicamente, la que aporte más elementos de valor añadido. En este sentido, *Microsoft* ha desarrollado su máquina virtual, que se distribuye con *Internet Explorer 4* y en el ámbito de su propio *Kit de Desarrollo con Java,* que, en consonancia con el resto de productos de la empresa, se denomina *Java SDK 2.0 (Software Development Kit).* A lo largo de este curso hablaremos largo y tendido de *SDK 2.0,* pero me permito adelantar que se trata de un estupendo entorno de desarrollo en *Java* que llenará de satisfacción a los desarrolladores en *Java* para el entorno *Windows.*

La publicación del *SDK 2.0* y, con él, de la nueva máquina virtual, es el detonante de la disputa entre *Sun* y *Microsoft.* Existe un contrato que vincula a ambas compañías y con el que *Microsoft* se compromete a incluir ciertas funcionalidades en sus implementaciones de la máquina virtual *Java* y otros aspectos farragosos en los que nos vamos a entrar, tales como el uso de logotipos que *Sun* licencia para la utilización en plataformas compatibles con el estándar *Java. Sun* dice que *Microsoft* incumple el contrato, y que no está implementando ciertas partes del estándar. *Microsoft* replica, por su parte, que ni siquiera *Sun* incluye todas las funcionalidades del estándar y que sin duda la máquina virtual de *Microsoft* es la que más se ajusta al mismo. Dejemos que el tiempo y los tribunales hagan justicia, algo a lo que nos estamos acostumbrando en España en otros ámbitos, pero tengamos por seguro que la máquina virtual de *Microsoft* es un estupendo entorno y que colmará las aspiraciones y necesidades de la práctica totalidad de los desarrolladores en *Java.*

¿Qué pasa con Visual J++?

Hemos de dejar bien claro que *Visual J++* no es objeto de la controversia a la que estamos haciendo referencia. La plataforma de desarrollo que va a servir como sustento a este curso es la más utilizada por los desarrolladores en *Java* (más de la mitad de los programadores la utilizan) y permite el desarrollo abierto y multiplataforma en el lenguaje creado y bautizado

por *Sun*. Es decir, y a pesar de lo que algún desarrollador desinformado pueda decir, *Visual J++* permite la creación de *applets Java* estándar, que funcionarán en cualquier plataforma y en absoluto restringidos a *Windows 95, Windows NT,* ni ningún otro sistema operativo. La demanda no afecta a *Visual J++,* sino a la implementación de *Microsoft* de la máquina virtual. Dicho de otro modo, la batalla está librándose alrededor del intérprete, y no del compilador.

Visual J++ permite, pues, si se ha instalado *SDK 2.0,* desarrollar aplicaciones y *applets Java* que hagan uso de las más modernas tecnologías *Java:* componentes *JavaBeans,* acceso a bases de datos utilizando *JDBC* (en posteriores entregas de este curso presentaremos las técnicas de acceso a bases de datos en *Java* con *Visual J++)* y cualquier otra que pueda desearse. Siendo esto cierto, tal y como ilustraremos más adelante, *Visual J++ / SDK 2.0* es mucho más. La tecnología *J/Direct,* de la que ya hablaremos, permite utilizar todas las funcionalidades de la programación *API* en *Windows,* borrando de un plumazo las barreras que hacían *Java* inadecuado para desarrollos críticos que sí podían abordarse con C++. Es decir, *Visual J++* permite desarrollar *applets* y aplicaciones multiplataforma sin ningún problema, pero si deseamos crear aplicaciones que se ejecuten en *Windows,* ninguna otra combinación de plataforma/kit de desarrollo ofrece mejores posibilidades.

• Una introducción a *Visual J++*

Como bien conocerán los programadores en *Visual C++, Microsoft* ha integrado algunas de sus herramientas de desarrollo en un entorno común que facilita la automatización de las tareas de programación. Dicho entorno es *Microsoft Developer Studio (*véase la figura A) y sirve de sustento a *Visual C++, Visual InterDev* y *Visual J++*.

Microsoft Visual J++, en el marco de *Microsoft Developer Studio,* nos permitirá escribir, compilar, depurar y ejecutar aplicaciones y *applets Java,* todo ello desde un único entorno de desarrollo. El entorno consiste en una serie de herramientas incluyendo herramientas de compilación y depuración, asistentes para la creación automática de aplicaciones, un editor de texto que resalta la sintaxis de *Java* y editores de recursos.

Gestión de proyectos

En *Microsoft Developer Studio*, el trabajo se organiza en *workspaces*. La ventana del *workspace* es una herramienta versátil que ofrece la posibilidad de personalizar el proyecto y manipular sus componentes. Cada *workspace* incluye diferentes vistas de los componentes del proyecto *Visual J++:* clases, recursos y ficheros. En el caso del que el proyecto incluya bases de datos*,* también de dispondrá de una visión jerarquizada de los objetos de datos residentes en el proyecto. Desde estas vistas puede, por ejemplo, accederse a las funciones miembro de las clases.

Dentro de un área de trabajo o *workspace* pueden coexistir diversos proyectos. De hecho, el *workspace* en *Microsoft Developer Studio* puede incluir proyectos de diversos tipos, (por ejemplo los diagramas de una base de datos relacional en *SQL Server,* un proyecto de *Microsoft Visual C++,* un segundo de *Microsoft Visual InterDev* y otro de *Microsoft Visual J++,* todos ellos parte de un proyecto *Web* completo). Cuando un espacio de trabajo se crea por vez primera, el entorno gestiona la creación de un nuevo y propio directorio y un fichero para el *project workspace* con la extensión *.dsw*.

Cada proyecto o subproyecto existente en el entorno de trabajo, está definido por un fichero *.dsp,* y formado por una serie de ficheros que se relacionan entre sí de manera que la construcción de la aplicación necesita que las relaciones sean actualizadas y respetadas. El entorno se encarga de estas actualizaciones, de manera que si se modifica un fichero, que depende de un segundo dichas modificaciones sean tenidas en consideración al trabajar con el de superior rango. La inclusión de múltiples subproyectos en un proyecto permite mantener y gestionar dependencias entre aplicaciones que se encuentren relacionadas.

• Componentes del entorno de desarrollo

Los asistentes o wizards

Como hemos comentado anteriormente, *Visual J++* incluye un conjunto de asistentes o *Wizards*, que facilitan la labor de construcción de aplicaciones. Dichos asistentes son *Applet Wizard*, *Resource Wizard, ActiveX Wizard for Java, Database Wizard for Java* y *JavaType Library Wizard*

Applet Wizard

Esta herramienta permite crear un para la creación de un *applet*, es decir a una pequeña aplicación escrita en lenguaje *Java* y que se ejecuta en el marco de una página *Web*. Por decirlo de otro modo, *Applet Wizard* escribe de manera automática el conjunto mínimo de ficheros fuente necesarios para crear un *applet* básico, a partir de las clases predefinidas en *Java*. En los desarrollos que abordemos en este curso a menudo crearemos un esqueleto básico del *applet* mediante *Applet Wizard* para, posteriormente, personalizarlo y dotarlo de funcionalidad añadiendo nuevas clases, variables miembro y métodos. El aspecto del primero de los pasos que deben seguirse para crear un *applet* utilizando *Applet Wizard* puede verse en la figura B.

Resource Wizard

Este asistente permite traducir a código *Java* una plantilla de recursos existente en un proyecto. Estas plantillas de recursos podrán incluir, entre otros, menús, cuadros de diálogo, elementos de la interfaz de usuario, u otros. El código que resulte de la traducción de estas plantillas de recursos a lenguaje *Java* se almacena en ficheros que pueden incluirse en los proyectos. De este modo, los proyectos que incluyan este código generado crearán diálogos análogos a los que, por ejemplo podrían crearse con *Visual C++*.

Database Wizard for Java

Este asistente facilita la creación de un *applet* que acceda a una base de datos, bien sea a través de *DAO (Access)* o *RDO (Remote Data Objects, ODBC)*. El asistente nos conduce por una serie de pasos para especificar la fuente de datos a la que deseamos el *applet* acceda, las tablas que deseamos manejar y los campos que queremos que se presenten. El resultado del proceso es un *applet* con botones de navegación y campos de edición de texto que acceden a la base de datos y nos presentan los resultados en el documento *Web* en el que ubiquemos el *applet*. En la figura D presentamos el aspecto de este asistente.

Java Type Library Wizard

Para utilizar cualquier objeto *ActiveX* desde *Java* es necesario importarlos creando una clase que lo envuelva *(wrapper class)* en el contexto y con la sintaxis de *Java*. Esto puede realizarse mediante *Java Type Library Wizard,* que se ilustra en la figura E.

COM (el modelo subyacente a *ActiveX)* define un mecanismo denominado bibliotecas de tipos *(type library)*, para almacenar la información del tipo de los objetos *COM*. Cada una de estas bibliotecas contiene información sobre uno o más objetos *ActiveX*.

El asistente creará una serie de paquetes para los objetos *COM* de las librerías seleccionadas, y nos presentará un archivo en el que se hallan las descripciones de las interfaces y métodos definidos, que podrán llamarse desde un programa en *Java*.

ActiveX Wizard for Java

Este asistente permite realizar el proceso inverso al anteriormente descrito, es decir, permite crear un control *ActiveX* con la misma funcionalidad que una determinada clase *Java*.

Editores de Recursos

Microsoft Developer Studio posee una serie de editores para los diversos tipos de recursos *Windows* existentes. Estos editores permiten crear y modificar menús, diálogos, *string tables, accelerator tables, version resources*, y objetos gráficos, como iconos, cursores, y bitmaps. Cuando se crea o se abre un recurso, el editor apropiado se abre automáticamente. Estos recursos, como hemos comentado, podrán ser después traducidos a código *Java* mediante *Java Resource Wizard*

Paneles de vistas jerárquicos

Microsoft Developer Studio organiza la información relativa a una determinada aplicación en árboles jerárquicos que permiten apreciar las relaciones entre los diversos componentes incluidos en un proyecto. Estos paneles jerárquicos permite ver las clases y los ficheros que incluye en proyecto. El primero de ellos es *ClassView* y nos permite explorar la estructura jerarquizada de nuestro proyecto desde el punto de vista de las clases. El árbol en el que aparecen dispuestas las clases puede expandirse para acceder directamente a las funciones miembro y los datos asociados, y de ellos se puede llegar al punto exacto del código en el que se definen. Este árbol de jerarquías puede verse en la figura F.

FileView es el nombre del segundo panel (véase Figura G), que presenta las relaciones entre los ficheros incluidos en e proyecto, permitiendo acceder directamente al código fuente

Un tercer panel, *Data View,* nos permite examinar nuestro proyecto jerarquizado desde el punto de vista de las bases de datos asociadas al mismo, siempre que estén asociadas a fuentes de datos *ODBC*. El aspecto de este nuevo visor puede apreciarse en la figura H.

Manos a la obra

Todo lo que hemos explicado puede llegar a ser incluso interesante, pero este artículo es la primera entrega de un curso de *Desarrollo en Java con Visual J++,* y por lo tanto el lector estará esperando que la literatura anterior se plasme en algún programa concreto.

Vamos a crear un par de sencillos proyectos para presentar la plataforma de desarrollo, demostrando, al tiempo, que *Visual J++* no es la plataforma de desarrollo de *Java* para *Windows,* sino un entorno de desarrollo de propósito general con *Java* que permite crear *applets* y aplicaciones independientes de la plataforma. Para los más incrédulos vamos a crear un proyecto *Visual J++* que contendrá dos *applets* desarrollados por *Sun* y propuestos como ejemplos por los creadores del lenguaje. No es el objeto de este proyecto el someter a un test de compatibilidad a la máquina virtual de *Microsoft Internet Explorer 4,* sino comprobar que con *Visual J++* pueden crearse y compilarse los *applets* del *JDK,* como ejemplo de *applets* que se ejecutarán en cualquier plataforma compatible. Es decir, *Visual J++* es un entorno de desarrollo abierto.

El primero de estos dos ejemplos es uno de los que *Sun* proporciona como un *sample* en su *JDK 1.0,* y su funcionalidad se limita a presentar un texto parpadeante. El lector estará pensando que el *applet* no realiza nada especialmente sorprendente y que puede obtenerse el mismo efecto sin necesidad de utilizar *Java.* Además también es posible que el lector que no conozca nada de *Java* se irrite al ver que vamos a compilar un pequeño programa en este lenguaje sin presentar ni un pequeño resumen de su sintaxis. Efectivamente, el *applet* no ganaría ningún concurso, pero convinamos que en este punto lo que se pretende es comprobar que *Visual J++* es adecuado para compilarlo. En lo que a la segunda duda se refiere, remitimos a la siguiente entrega del curso para una somera descripción del lenguaje.

El segundo ejemplo está extraído de los ejemplos que *Sun* ha ubicado en su sitio *Web* para la difusión de las informaciones relacionadas con *Java:* http://www.javasoft.com/. Es una versión sencilla del juego del ahorcado. Los lectores interesados en el código fuente de ambos ejemplos pueden encontrarlos en la sede *Web* de *Hispan: www.hispan.com*

Creación del proyecto principal

Antes de compilar ambos *applets* es preciso crear sendos proyectos *Java* dentro de un mismo *workspace,* cada uno de ellos destinado a la compilación de uno de los *applets* vamos a presentar el proceso de creación siguiendo los pasos necesarios para llevarlo a cabo. Figura I. Creación de un proyecto.

- Iniciaremos *Microsoft Developer Studio*
- Pulsaremos el menú ***New.*** Aparecerá el cuadro de diálogo de la figura J, en el que se nos permitirá escoger el tipo de elemento que deseamos crear, sea un proyecto, un fichero fuente o un asistente que nos facilite la tarea.
- Elegiremos la pestaña *Projects* y dentro de ella el tipo de proyecto *Java Project.* Especificar como nombre *Blinking*
- Especificaremos el directorio en el que se deseamos se almacenen los ficheros del proyecto
- Elegiremos la opción *Create New Workspace*
- Una vez hemos pulsado *OK* obtendremos un proyecto vacío, al que deberemos añadir nuestro fichero fuente *blink.java* que habremos obtenido bien de la documentación de *Visual J++,* o de la sede web de *Sun* (en el *JDK 1.0)* o de la sede *Web* de *Hispan www.hispan.com.*
- Para añadir el fichero pulsaremos el menú *Project / Add To Project/ Files* y en el cuadro de diálogo resultante elegiremos el fichero *blink.java.* El aspecto del panel de ficheros *(File View)* del entorno de desarrollo una vez hemos añadido el fichero fuente puede verse en la figura J.

Una vez creado el proyecto y añadido el fichero fuente deberemos proceder a compilarlo para obtener el *bytecode blink.class,* es decir, el fichero con el código que se ejecutará sobre la máquina virtual *Java*. Para ello bastará con pulsar con el botón derecho sobre el fichero *blink.java* y en el menú contextual elegir *Compile blink.java.*

Para ver el resultado de la compilación, es decir, para ejecutar el *applet,* es necesario que se cree una página *HTML* en la que se incluya el citado objeto. Recuérdese que el ámbito de ejecución de un *applet* es el navegador. *Visual J++* utiliza *Microsoft Internet Explorer* como soporte para la ejecución y depuración, de manera que es posible depurar un *applet* mientras está ejecutándose. Es decir, nos va a hacer falta una página *Web. Visual J++* nos permite crearla, como un fichero fuente más, e incluirla en el proyecto. De hecho, *Visual J++* proporciona un editor *HTML* que colorea las etiquetas del lenguaje para hacer más legibles los documentos y más sencilla la tarea de crearlos. Sin embargo, no es necesario llevar a cabo este proceso manualmente, ya que la plataforma de desarrollo, si no se especifica un fichero *HTML* que incluya al *applet,* crea uno, temporal, que facilita su visualización. Figura K Opciones de proyecto

En nuestro caso vamos a escribir un pequeño documento *HTML* que incluya el *applet* que acabamos de crear aunque sólo sea para ilustrar el proceso. Mediante *Visual J++* crearemos un nuevo fichero *HTML* que llamamos *hispan.htm* y escribiremos el código siguiente:

HISPAN.HTM

```
<html>

<head>

<title>blink</title>

</head>

<body background = "logohispan.jpg">

<applet

code=blink

width=200

height=200>

</applet>

</body>

</html>
```

Una vez creado el fichero deberemos incluirlo en el proyecto y decirle al entorno que deseamos que lo utilice para depurar nuestro *applet*. Estas opciones pueden configurarse en el menú *Build/Settings,* tal como puede verse en la figura K. Si se especifica una página *Web*

en ese cuadro de diálogo, esa página será la utilizada, en caso contrario, el sistema creará el documento *Web* temporal. Una vez introducido el nombre del fichero, pulsaremos el menú *Build/Execute* para iniciar *Microsoft Internet Explorer*. Deberá abrirse una ventana del explorador en la que se mostrará el *applet,* que se muestra en la figura L. El código fuente del *applet* es el siguiente:

BLINK.JAVA

```
/*
* Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved.
*/
import java.awt.*;
import java.util.StringTokenizer;
/**
* I love blinking things.
* @author Arthur van Hoff
*/
public class blink extends java.applet.Applet implements Runnable {
Thread blinker;
String lbl;
Font font;
int speed;
public void init() {
font = new java.awt.Font("TimesRoman", Font.PLAIN, 24);
String att = getParameter("speed");
speed = (att == null) ? 400 : (1000 /
Integer.valueOf(att).intValue());
att = getParameter("lbl");
lbl = (att == null) ? "Blink" : att;
}
```

```
public void paint(Graphics g) {

int x = 0, y = font.getSize(), space;

int red = (int)(Math.random() * 50);

int green = (int)(Math.random() * 50);

int blue = (int)(Math.random() * 256);

Dimension d = size();

g.setColor(Color.black);

g.setFont(font);

FontMetrics fm = g.getFontMetrics();

space = fm.stringWidth(" ");

for (StringTokenizer t = new StringTokenizer(lbl) ; t.hasMoreTokens()
; ) {

String word = t.nextToken();

int w = fm.stringWidth(word) + space;

if (x + w > d.width) {

x = 0;

y += font.getSize();

}

if (Math.random() < 0.5) {

g.setColor(new java.awt.Color((red + y * 30) % 256, (green + x / 3) %
256, blue));

} else {

g.setColor(Color.lightGray);

}

g.drawString(word, x, y);

x += w;

}
```

```
}

public void start() {

blinker = new Thread(this);

blinker.start();

}

public void stop() {

blinker.stop();

}

public void run() {

while (true) {

try {Thread.currentThread().sleep(speed);} catch (InterruptedException
e){}

repaint();

}

}

}
```

Figura L El applet Blink, en el navegador.

En cuanto al segundo proyecto, actuaremos de manera completamente análoga, con el fichero *hangman.java,* creando un nuevo proyecto que puede incluirse en el mismo *workspace* o en un nuevo entorno de trabajo. En la figura M presentamos el aspecto del panel visor de clases para nuestro *workspace,* en el que hemos incluido los dos proyectos. Procederemos a la compilación y ejecución del *applet,* cuyo aspecto se muestra en la figura N. Como el lector comprobará, ¡el *applet* funciona! En este caso no adjuntamos impreso el código fuente, que como ya hemos comentado está a disposición del lector en *www.hispan.com* o en *javasoft.*

Bien, hasta aquí esta primera entrega, en la que tan sólo hemos pretendido presentar *Visual J++,* la plataforma de desarrollo en la que nos sumergiremos en las siguientes lecciones del curso, poniéndola en situación en el convulso mundo del desarrollo en *Java.* En la próxima entrega nos dedicaremos a utilizar *Visual J++* para crear nuestros primeros *applets* propios, tanto utilizando los asistentes como escribiendo por completo nuestro código fuente.

Alberto Delgado / HISPAN TECNOLOGIC

• Sintaxis básica del lenguaje

La sintaxis del lenguaje *Java* resultará familiar a todos aquellos que conozcan C++ o incluso C, ya que en su proceso de definición de partió de la de estos. Sin ir más lejos, y a modo de ejemplo, ambos lenguajes coinciden en que son sensibles a las mayúsculas. Podríamos decir que *Java* es una progresión de C++ hacia una mayor simplicidad y orientación a objetos.

Como en todos los lenguajes de programación la acción más simple que puede llevarse a cabo es la sentencia, que en *Java* se termina con un punto y coma. Es posible también agrupar sentencias en bloques mediante la utilización de llaves (**{}**). Un bloque puede estar situado en cualquier punto de programa en el que pueda escribirse una sentencia.

Variables y tipos de datos

Java, por supuesto, permite la definición de variables, es decir, localizaciones en memoria en la que se almacena información. Una variable posee un nombre, un tipo y un valor. En *Java* existen tres tipos de variables: de objeto, de clase y locales. Las primeras definen propiedades de objeto y las segundas comunes a todos los de una clase, como ya veremos más adelante. Lo más reseñable es que en *Java* no existen variables globales.

Las declaraciones de variables en *Java*, como las de C++, consisten simplemente en un tipo y el nombre de una variable:

```
int Numero;

String Nombre;
```

Cada declaración de variable debe poseer un tipo de datos que defina cuales son los valores que se encuentran en el dominio de la misma. *Java* no es un lenguaje fuertemente tipado. Tan sólo se permiten 8 tipos básicos predefinidos, clases, y *arrays*. Los ocho tipos predefinidos comprenden los usuales para enteros, números en coma flotante, caracteres y valores booleanos. Sus tamaños se muestran en una tabla adjunta. Al contrario de los que sucede en C++ en *Java* no existen definiciones de tipos de datos de usuario mediante primitivas tipo *typedef*. Si un usuario pretende crear un tipo de datos deberá crear una clase y derivar objetos de ella.

Categoría	Tipo	Rango de valores	Memoria requerida (bytes)
Valores enteros	*byte*	0-255	1
	short	-32768 - 32767	2
	int	-2147483648 - 2147483648	4
	long	-2^{63} a $2^{63}-1$	8
Valores reales	*float*	Reales de precisión simple	4
	double	Reales de precisión doble	8
Booleano	*boolean*	true o false	1

Caracteres	***char***	Caracteres ANSI o Unicode	2

Como habrá podido verse, no existe ningún tipo para representar cadenas de caracteres. En *Java* las variables que almacenen cadenas de caracteres serán ejemplares de la clase *String,* incluida en el lenguaje dentro de la librería de clases, como veremos próximamente. La clase proporciona un extenso conjunto de métodos y propiedades para manipular las cadenas de caracteres. Una constante que representa una cadena se define en *Java* incluida entre comillas dobles**.**

Los operadores de *Java* resultan prácticamente idénticos a los de C++, por lo que nos limitamos a presentarlos en una tabla adjunta.

Categoría	Operador	Operación
Asignación	**=**	Asignación
	+=	x = x + y
	-=	x = x - y
	***=**	x = x * y
	/=	x = x / y
Aritméticos	**+**	Suma
	-	Resta
	*****	Multiplicación
	/	División
	%	Módulo
Booleanos	**!a**	Negación (NOT)
	a & b	Producto lógico (AND)
	a && b	Producto lógico (AND)
	a | b	Suma lógica (OR)
	a | | b	Suma lógica (OR)
	a ^ b	Suma exclusiva (XOR)
Bit	*&*	Producto a nivel de bit
	|	Suma a nivel de bit
	^	*or* exclusiva a nivel de bit
	~	Complemento

	<<	Desplazamiento izquierda
	>>	Desplazamiento derecha
	>>>	Desplazamiento a la derecha con relleno con ceros.
	&=	x = x & y
	| =	x = x | y
	^=	x = x ^ y
Comparación	==	Igualdad
	!=	Desigualdad
	<	Menor que
	>	Mayor que
	<=	Menor o igual que
	>=	Mayor o igual que

Arrays

Los *arrays* contienen un conjunto homogéneo e indexado de datos. Los elementos del *array* pueden procesarse individualmente simplemente indicando la posición que dicho elemento ocupa dentro de la estructura de datos. El primer elemento de un *array* se halla en la posición 0. La declaración de una variable *array* consta simplemente de una declaración normal de variable en la que el nombre de la misma o el del tipo aparecen seguidos de corchetes. A continuación presentamos dos versiones de declaraciones de variables *array*

```
String arraydecadenas[ ];

String[ ] arraydecadenas;
```

Una vez el *array* se ha declarado, deberemos asignarle espacio en memoria e inicializarlo utilizando el operador *new*

```
String[ ] arraydecadenas = new String[3];

String[ ] arraydecadenas = { "hola","adios","fin" };
```

Control del flujo de programa

Los programas pueden llevar a cabo sus tareas en buena medida gracias a la posibilidad de tomar decisiones y ejecutar acciones repetidamente en función del estado del programa. Estas capacidades se fundamentan en la ejecución condicional y repetida de código que en *Java* se expresa en las siguientes estructuras

- If...Else: Es la más fundamental de las sentencias de ejecución condicional. Permite condicionar la ejecución de una porción de código al cumplimiento de una determinada condición. La porción de código denominada *código1* se ejecutará si la *condición* se cumple, en caso contrario y si la cláusula *else* está presente, se ejecutará *código2*

```
If condición

código1;

[else // opcional

código2; ]
```

- Switch: Permite al programador comparar una variable con un conjunto predefinido de valores y ejecutar una porción u otra de código en función del contenido de dicha variable. Esta misma función puede realizarse mediante una serie de *if/else* anidados.

```
switch (variable test) {

case valor1:

Código1;

break;

…

case valorn:

Códigon;

break;

default:

Código por defecto;
```

- for: Esta sentencia permite repetir la ejecución de un bloque de código un número definido de veces. Al comenzar su ejecución se inicializa *variable* para que adopte el *valor inicial*. Al final de la ejecución de *código* se incrementa el valor de la variable y se comprueba si se cumple una determinada *condición;* en caso de que no se cumpla la ejecución de la sentencia finaliza.

```
for (variable = valor inicial; condición; incremento){
código }
```

- do…while: Permite repetir la ejecución de una porción de código mientras se satisfaga una cierta condición. En esta sentencia *código* se ejecuta como mínimo una vez.

```
do {
```

```
código;

} while (condición)
```

- while: Es muy similar a la anterior con la diferencia de que la verificación del cumplimiento de la condición se lleva a cabo antes de ejecutar el.

```
while (condición) {

código;

}
```

Java es un lenguaje orientado a objetos

La Programación Orientada a Objetos (POO) es una filosofía de programación que se ha ido imponiendo en los últimos años frente a las técnicas tradicionales de programación estructurada, sobretodo en ámbitos que responden a modelos de programación controlada por sucesos. En POO el programa está formado por una serie de componentes independientes y cerrados o mejor, autocontenidos, que cooperan para realizar las acciones de la aplicación completa. La POO se fundamenta en los conceptos de objeto y clase. Una clase es una *plantilla* que abstrae las características de una cierta entidad. Un objeto es cada representación concreta de la abstracción de la clase.

Java, a diferencia de otros lenguajes como por ejemplo C++, basa gran parte de su funcionalidad en su *librería de clases,* que es parte integrante del lenguaje y que se distribuye con el *JDK* o *SDK,* conjuntamente con el compilador y la máquina virtual. Estas clases implementan gran parte de las tareas de programación básicas tales como soporte para *arrays,* funciones matemáticas, *strings*, o gestión de ventanas y diálogos, por lo que en la mayoría de los casos nuestra labor consistirá en crear clases que utilicen a las de la librería. A esta librería de clases estándar se añaden otras que ofrecen funcionalidades añadidas. Por ejemplo, *Sun* acaba de hacer pública una librería de clases denominada *JFC (Java Foundation Classes),* y *Microsoft* ha creado, en el ámbito de su *SDK 2.0,* la librería de clases *AFC (Application Foundation Classes),* que será objeto de nuestra próxima entrega del curso. A propósito, ¿es casualidad que las clases de *Sun* posean un nombre tan parecido a *AFC* y a la inspiradora de ambas, *MFC*?

Cada clase está formada por métodos y propiedades. Las propiedades explicitan aquellas características individuales que definen y diferencian a un objeto de otro, determinando su estado. Cada propiedad está representada por una variable miembro. Estas variables son inicializadas en el momento de la creación del objeto y pueden modificarse a medida que dicho objeto va consumiendo su vida útil. También existen ciertas propiedades, que denominaremos de clase**,** cuyo valor es común a todos los ejemplares de la misma.

Los métodos gobiernan el comportamiento de la clase, y por tanto de cada uno de los objetos, determinando cómo responderá cada ejemplar de la clase a las modificaciones de sus atributos. Estos métodos se expresan en funciones que incluyen el auténtico código de los programas orientados a objetos pues especifican la manera en como los objetos realizarán todas y cada una de sus acciones

Jerarquía de clases

La herencia es uno de los conceptos más importantes en POO. Se trata de un mecanismo que permite a la hora de crear una nueva clase especificar simplemente en que se diferencia la nueva de otra u otras ya existentes, con el consiguiente ahorro de esfuerzo y tiempo. Cada clase derivada hereda todos y cada uno de los métodos y propiedades de su clase base. De esta manera si el comportamiento de la clase basa colma las necesidades del programador, no necesitará escribir ningún código adicional, pues podrá utilizar el que la clase base posee. La herencia es la base de la *reutilización del código.* De este modo las clases forman una jerarquía en la que cada una desciende o es hija de aquellas de las que deriva, que han servido de base para crearla. La librería de clases de *Java* es un ejemplo de este comportamiento jerarquizado. Podemos utilizar las clases de la librería para, derivando nuestras clases de ellas, añadir la funcionalidad que necesitemos sin que sea preciso escribir todo el código. Figura A, ejemplo de jerarquía de clases.

Es posible que en una de nuestras clases, inmersa en una jerarquía, deseemos modificar alguna de las funciones heredadas para mejor adaptarla a las necesidades de la clase en cuestión. Esto es posible utilizando un mecanismo denominado redefinición**,** en inglés *overriding*. Redefinir un método heredado es crear uno nuevo en una subclase con el mismo nombre y lista de argumentos que uno existente en una de sus clases base. Decimos que este nuevo método oculta el método de la clase base.

Encapsulación

Una de las más importantes características de la POO es la de mantener los objetos como *cajas negras* que cooperan entre sí para realizar acciones pero de las que se desconoce su estructura interna. Este fenómeno, denominado encapsulación, permite utilizar los objetos prescindiendo de los detalles concretos concernientes a su diseño. Cada clase posee dos partes: su estructura interna, que no es necesario conocer para utilizar los objetos que a partir de ella se creen, y una interfaz a través del cual las funcionalidades del objeto se hacen accesibles. La idea es mantener la estructura interna, o estado, oculta y protegida al usuario y dar acceso a la misma de manera controlada a través de funciones miembro de la clase que sí sean libremente utilizables. Figura B, encapsulamiento.

Esta filosofía se plasma en la calificación de cada uno de los métodos y propiedades de las clases con una palabra que indica si dichos miembros son o no accesibles desde el exterior. De esta manera cada uno de ellos puede ser público si es libremente accesible, privado si no puede ser utilizado desde el exterior de la clase, y protegido si deseamos que sea público para sus clases derivadas y privado para el resto del mundo.

• Definición de clases en *Java*

Java utiliza una palabra reservada: ***class*** para la definición de clases

```
class nombre clase {

…definición de la clase

}
```

Cuando deseamos derivar una clase como derivada de una clase base utilizaremos la palabra clave *extends*

```
class nombre clase derivada extends nombre clase base

{

…}
```

Clases y ficheros

Como en cualquier lenguaje, el código se escribe en ficheros, que después con compilados (en el caso de *Java* a *bytecodes,* como vimos en la anterior entrega del curso) Las clases, en *Java* son definidas como públicas o privadas. Las clases públicas son aquellas que son visibles fuera del fichero fuente en el que se hallan, siempre que se importen en el fichero en el que van a usarse. Sólo puede definirse una clase pública por fichero, cuyos nombres deben además coincidir.

Métodos y propiedades

La definición de la clase habrá de contener la especificación de las variables miembro y los métodos que la doten de funcionalidad y que la diferencien de las clases de las que deriva. Como ya hemos comentado, las variables miembro pueden dividirse en variables de objeto y variables de clase. Las variables miembro de objeto se definen en la propia definición de la clase de igual manera a como se definiría cualquier variable local. Las variables de clase utilizan la palabra *static* precediendo a la definición de la variable.

Los métodos definen el comportamiento de los objetos, pues contienen el código necesario para llevar a cabo todas las acciones existentes durante la vida del objeto. En *Java* no existen funciones externas a las clases, es decir, todo el código que escribamos estará en el interior de las definiciones de clase, como métodos de estas. Esto puede resultar chocante para los programadores en C++, pero tiene la ventaja de encapsular y confinar el código de manera muy coherente, aunque pueda hacer poco legibles nuestras definiciones de clase

```
class NombreClase {

tipo1 variabledeobjeto1;

tipo2 variabledeobjeto2;

static tipo3 variabledeclase

tipoderetorno metodo1( declaraciones de argumentos )

{

cuerpo del método

}
```

El valor de retorno se especificará a continuación de la palabra clave *return* en el interior del cuerpo del método.

```
class ClaseEjemplo {

int[] MatrizdeNumeros (int inferior, int superior) {

int arr[] = new int[ (superior - inferior) + 1 ];

for (int i = 0; i < arr.length; i++) {

arr[i] = inferior++;

}

return arr;

}

public static void main (String arg[]) {

int ElArray[];

ClaseEjemplo Ejemplo = new ClaseEjemplo();

ElArray = Ejemplo.MatrizdeNumeros(1,10);

}

}
```

Para acceder a una propiedad o método se hace uso de la sintaxis de punto. Es decir, para modificar una propiedad o llamar a un método de una clase lo haremos precediendo al nombre del método o propiedad con el nombre del objeto separado por un punto. En el caso de las variables de clase, podremos hacerlo a través del propio nombre de la clase, utilizando la misma notación. De hecho es recomendable hacerlo así, para hacer patente que se trata de una variable de clase.

Paquetes (*packages*)

Los paquetes o *packages* son el modo en como *Java* estructura el código, agrupando y clasificando las clases. A medida que el número de clases que utilicemos crezca cada vez será más necesaria la reutilización del código y funcionalidad de cada una de las clases que hayamos creado. La mejor manera de hacernos disponibles en el futuro las clases que diseñemos sin que nos resulte imprescindible recordar los detalles de implementación de las mismas, es ocultarlas en el marco de una entidad mayor, que en el caso de *Java* será el *package*. Para definir un *package* bastará con incluir una sentencia como la siguiente como primera línea de un fichero fuente *Java*

```
package NombrePackage
```

Los paquetes se estructuran también de manera jerárquica. El compilador de *Java* fuerza a que se cree una estructura de directorios que posea exactamente la misma estructura jerárquica que los paquetes que hayamos definido. Por ejemplo la clase *ColorModel*, que se

haya en el paquete *java.awt.image* será referenciada de manera completa como *java.awt.image.ColorModel* y se halla en el fichero *\java\awt\image\ColorModel.java*.

Cuando se hace referencia a una clase en nuestro código *Java*, lo haremos a través de un *package*. En la mayor parte de las ocasiones esto nos resulta inadvertido porque la clase se haya en un paquete que el compilador *importa* por defecto, *java.lang*. Cuando deseemos utilizar una clase que se halla en el marco de un determinado paquete bastará con que especifiquemos el paquete en el que esta se incluye mediante la sentencia *import*.

Protección de variables y métodos: acceso

Para implementar el encapsulamiento de las clases, *Java* proporciona un conjunto de palabras clave para limitar el acceso a las propiedades y métodos de los objetos. Estos especificadores de acceso preceden a las definiciones de los métodos y variables de las clases.

- *public*: Si un método o variable se define como *public*, será accesible para cualquier otra clase, de manera que cualquiera puede usar el método o acceder a la variable.
- *package*: Para permitir que un método o variable sea visible para el resto de las clases del mismo paquete bastará con no especificar ningún identificador de acceso. Es decir, el ámbito *package* es el que se considera por defecto.
- *private*: Los métodos y variables declarados *private* tan sólo son visibles en la propia clase, de manera que sólo métodos de la propia clase pueden acceder a sus variables y sólo objetos de la misma pueden llamar a sus métodos.
- *protected*: Lo que se declare con este especificador de acceso será público para sus clases derivadas y privado en cualquier otro caso.

Creación de objetos. Constructores

Una vez las clases, que podemos entender como unas plantillas que dan forma a los objetos, han sido definidas, nuestra tarea será definir ejemplares de estas clases. Para crear un nuevo objeto, se utiliza el operador *new* seguido por el nombre de la clase, seguido por paréntesis, tal como puede verse en el siguiente ejemplo

```
NombreClase nombreobjeto = new NombreClase();
```

El nombre de la clase seguido por paréntesis que sigue al operador *new* es lo que denominamos el constructor, un método especial que permite inicializar un objeto en el momento de su creación, utilizando una serie de argumentos. La existencia de una serie de diferentes constructores en una clase permite inicializar los objetos de maneras diversas.

El operador *new* crea un objeto de la clase deseada, le asigna memoria dinamicamente, y llama al constructor. Veamos un ejemplo en un pequeño programa de construcción de objetos.

Los paquetes de la librería de clases de *Java*

Como ya hemos comentado, el propio lenguaje incluye una librería de clases agrupadas en paquetes. Los paquetes de la implementación mínima son los siguientes:

- *Lang*: Clases del propio lenguaje, incluyendo *Object*, *String*, *Integer*, etc.

- *Util*: Clases de utilidad como *Date* y clases de colección, como *Vector*.
- *Io*: Lectura y escritura de la salida y la entrada estándar.
- *Net*: Soporte de red, como por ejemplo *Socket*
- *Awt*: *(Abstract Window Toolkit)* Clases para implementar el interfaz de usuario, por ejemplo *Window, Button, CheckBox,* etc.
- *Applet*: Clases para los *applet* incluyendo la propia clase *Applet* .

Creación de *applets*

La popularidad de *Java* estriba en buena medida en que sirve para construir pequeños programas, los *applets*, en el marco de páginas *Web*. En este apartado vamos a presentar los aspectos fundamentales necesarios para crear *applets*, y algunas de las clases de la librería estándar que intervienen en la creación de los mismos.

La clase Applet

El método que siempre debe seguirse para crear un *applet* es construir una subclase de la clase de librería *Applet***.** Esta clase proporciona por sí sola la funcionalidad necesaria para permitir al *applet* trabajar en conjunción con el *browser*:

```
public class miaplet extends Applet{

…

}
```

Cuando *Java* encuentra el *applet* que hemos creado en una página *Web,* carga la clase creada y crea un ejemplar de dicha clase, de manera que cuando existen diferentes *applets* en una misma página *Web*, se crean diferentes ejemplares, cada uno de ellos de una clase determinada. Para la creación de esta clase podemos apoyarnos, en *Visual J++,* en *Applet Wizard* el asistente para la creación de *applets* que introdujimos en la primera entrega del curso.

Ciclo de vida de un *applet*

A diferencia de lo que sucede con una aplicación *Java,* que posee un método *main()* que se ejecuta cuando se inicia y determina el desarrollo de la misma, un *applet* se comporta como es habitual en la programación orientada a objetos: respondiendo a diversos sucesos con actividades a ellos ligadas. Cada una de estas actividades están ligadas a métodos definidos en la clase *Applet,* y que deberán redefinirse en nuestra definición de clase para dotarla de la funcionalidad necesaria. Figura C, métodos básicos de un applet.

- Inicialización. Método *init:* Este método de la clase se ejecuta cuando el *applet* se carga por vez primera y determina su estado inicial.
- Inicio de ejecución. Método *start*: Se ejecuta tras la inicialización o tras una parada de la ejecución del *applet*. Este proceso puede producirse varias veces durante la vida del *applet,* mientras que el de la inicialización tan sólo se lleva a cabo una vez.
- Pausa de ejecución. Método *stop:* Se ejecuta cuando el lector de la página *Web* la abandona mientras el *applet* se está ejecutando.
- Destrucción. Método *destroy():* No suele redefinirse a no ser que se desee llevar a cabo labores de limpieza o liberación de recursos.

- Repintado. Método *paint:* Se ejecuta cuando el *applet* dibuja en la pantalla, texto o gráficos, por lo que dicho proceso puede llevar a cabo en múltiples ocasiones en la vida del mismo. Para proporcionar un comportamiento específico deberemos redefinir el método *paint()*. Este método toma un argumento de la clase *Graphics* que el *browser* pasará al *applet* de manera transparente al usuario y programador.

La clase *Graphics*

La mayoría de las capacidades de dibujo de *Java* se encapsulan como métodos de la clase *Graphics***.** Esta clase posee funciones miembro para dibujar líneas, formas, caracteres e incluso imágenes. Se encuentra en el paquete *java.awt.Graphics.* Otras clases auxiliares, como *Point,* o *polygon,* sirven también a efectos de la construcción y dibujo de figuras geométricas.

Como ya se ha visto, el repintado del *applet* se lleva a cabo mediante el método *paint(),* en el que el sistema pasa a nuestro *applet* un objeto de la clase *Graphics*. En esta función realizaremos las llamadas necesarias a los métodos de *Graphics* para dibujar lo que estimemos oportuno, algunos de los cuales son los siguientes:

- *drawLine;* Dibujo de líneas rectas.
- *drawRect* y *fillRect:* Dibujo y relleno de rectángulos
- *drawPolygon, fillPolygon:* Dibujo de líneas poligonales.
- *drawOval, fillOval:* Dibujo de elipses y círculos.
- *drawArc, fillArc:* Dibujo de arcos.

La clase Color

Java proporciona una clase, *Color,* para representar colores que puedan utilizarse tanto para el fondo del *applet* como para las líneas y rellenos de las figuras y los caracteres. Para dibujar un objeto de un determinado color en primer lugar deberá crearse un ejemplar de la clase. Esta clase define una serie de colores predefinidos, almacenados como variables de la clase. Algunos de estos colores estándar se presentan en la tabla adjunta

Color	Valor RGB
Color.white	255, 255, 255
Color.black	0, 0, 0
Color.blue	0, 0, 255
Color.green	0, 255, 0
Color.red	*255, 0, 0*
Color.yellow	*255, 255, 0*
Color.magenta	*255, 0, 255*
Color.cyan	*0, 255, 255*

Un applet como una herramienta de dibujo

Para finalizar esta entrega del curso de programación de *Visual J++* vamos a crear un *applet* que permitirá dibujar puntos y líneas rectas en la pantalla simplemente pulsando el botón izquierdo del ratón y desplazándolo sobre la ubicación del *applet* en la página *Web*. Para ello crearemos una clase derivada de *applet* y utilizaremos las funcionalidades de la clase *Graphics* para llevar a cabo las tareas de dibujo.

Para comenzar la creación del *applet* iniciaremos *Visual J++* y crearemos un nuevo proyecto mediante la opción de menú *File/New*. En el cuadro de diálogo resultante elegiremos la pestaña *Projects* y seleccionar como tipo de proyecto *Java Project*. Una vez creado, el proyecto no contendrá fichero alguno. Figura D: tipo de proyecto, figura E: panel de clases.

El siguiente paso es crear un fichero fuente en el que escribir el código de nuestro applet. Para ello, crear un nuevo fichero fuente Java mediante la opción de menú File/ New tras lo que, en la pestaña Files, elegiremos la opción Java Source File.

Vamos ya a escribir el código para dotar de funcionalidad a nuestro *applet* de dibujo. En primer lugar, vamos a importar las clases *graphics*, *color, event* y *point* todos ellos pertenecientes al paquete *AWT*. La clase de nuestro *applet* deberá tener el mismo nombre que el fichero que hemos creado, por ejemplo *Practica1*, derivada de la clase *Applet*. El código para ello será similar al siguiente:

```
import java.awt.Graphics;

import java.awt.Color;

import java.awt.Event;

import java.awt.Point;

class Practica1 extends java.applet.Applet {

...
```

Siempre dentro del código de la definición de la clase, definiremos un conjunto de propiedades de la clase:

- Dos *arrays* de 100 puntos, una para los extremos iniciales de las líneas a dibujar y otro para los finales de línea. Las coordenadas de estos extremos se almacenarán en objetos de la clase *Point*.
- Dos variables de la clase *Point* para almacenar el inicio y el final de la línea que está siendo dibujada en cada momento.
- Una variable entera que contenga el número de líneas que se han dibujado hasta el momento

Finalmente, deberemos redefinir los métodos de la clase *applet,* y añadir los que creamos conveniente para realizar nuestras tareas de dibujo:

- *init():* En este método ajustaremos el color de fondo del *applet* a blanco, para que la zona de dibujo quede claramente definida en la página *Web*. El código necesario podría ser el siguiente:

```
public void init() {

setBackground(Color.white);

}
```

- *paint(Graphics g):* En este método se llevará a cabo el dibujo de las líneas existentes y almacenadas en los *arrays*.

```
public void paint(Graphics g) {

// Dibuja las líneas existentes

for (int i = 0; i < nlineas; i++) {

g.drawLine(inicios[i].x, inicios[i].y,

finales[i].x, finales[i].y);

}

// Dibuja la línea actual

g.setColor(Color.blue);

if (finalactual != null)

g.drawLine(inicioactual.x,inicioactual.y,finalactual.x,fin
alactual.y);

}

}
```

- *nuevalínea(int x, int y):* Este método nuevo, no existente en la clase *applet, a*lmacena el valor del inicio de la línea actual en el *array* de inicios de línea, almacena el valor del final del array utilizando las coordenadas que se le han pasado como argumento a la función, incrementa el número de líneas existentes, asigna el valor *null* al final de la línea actual y fuerza el repintado de la pantalla.

```
void nuevalinea(int x,int y) {

inicios[nlineas] = inicioactual;

finales[nlineas] = new Point(x,y);

nlineas++;

finalactual = null;

repaint();

}
```

- *MouseDown(Event evt, int x, int y)*: Esta función gestionará el evento provocado por la pulsación en un botón del ratón. Creará e inicializará el punto inicial de la nueva línea a dibujar.

```
public boolean mouseDown(Event evt, int x, int y) {

inicioactual = new Point(x,y);

return true;

}
```

- *MouseUp(Event evt, int x, int y)*: Esta función gestionará el evento provocado al abandonar la pulsación en un botón del ratón. Llamará al método *nuevalínea*.

```
public boolean mouseUp(Event evt, int x, int y) {

nuevalinea(x,y);

return true;

}
```

- *MouseDrag(Event evt, int x, int y)*: Esta función gestionará el evento provocado al arrastrar el ratón mientras se pulsa un botón. Actualizará el valor de las coordenadas actuales del punto final de la línea que está siendo dibujada y forzará el repintado de la pantalla

```
public boolean mouseDrag(Event evt, int x, int y) {

finalactual = new Point(x,y);

repaint();

return true;

}
```

Figura F: opciones de proyecto, figura G: eligiendo el documento HTML de depuración, figura H: resultado de la práctica.

Una vez el código ya está acabado incluiremos el fichero en el proyecto, mediante la orden *Project / Add Files,* y modificaremos sus opciones de compilación mediante la orden *Project/Settings*

Una vez compilado, podremos ver el resultado del *applet* en una página *Web.* El código completo del *applet* puede encontrarse en el *CD* de la revista, así como en la sede *Web* de *HISPAN TECNOLOGIC,* en www.hispan.com/eltaller , el que además encontraréis información respecto al sorteo de un curso de Visual J++ a distancia con el que *HISPAN TECNOLOGIC* va a premiar a los lectores de *PCACTUAL* que sigan este curso. Por otra parte también podéis dirigiros para formular tantas consultas como creáis oportuno respecto a este curso al grupo de noticias *hispan.visualj,* en el servidor de noticias de

HISPAN TECNOLOGIC: news.hispan.com. En la próxima entrega del curso trataremos la librería de clases *AFC*.

¿ *SDK* ?

SDK son las siglas de *Software Development Kit,* que podría traducirse como el *kit* de desarrollo de *Software,* y su uso no se limita a *Java* sino que hace referencia a un gran número de librerías de *software* que *Microsoft* facilita a los programadores para la creación de aplicaciones en muy diversos ámbitos. De hecho, la propia *API* de *Windows* no es más que un *SDK* más. En las últimas semanas *Microsoft* ha publicado uno de estos *kits* para permitir a los desarrolladores crear aplicaciones en *Java* en los ámbitos de los sistemas operativos de *Microsoft*. Este kit recibe el nombre de *SDK 2.0 (*de hecho la versión actual es *SDK 2.01*), es gratuito y puede descargarse, con una cierta paciencia, de la sede *Web* de *Microsoft* *http://www.microsoft.com/Java/*

SDK 2.0 incluye la nueva máquina virtual de *Microsoft,* la librería de clases *AFC,* un conjunto de herramientas adicionales, aplicaciones de ejemplo y una extensa documentación que nos van a facilitar la tarea de desarrollar tanto aplicaciones completas como *applets* que cumplan las especificaciones del estándar *JDK 1.1* de *Sun*.

El resto de entregas de este curso van a recorrer el *SDK* presentando con cierta profundidad algunos de sus aspectos más reseñables. Esto no es óbice para que podamos anticipar las innovaciones que aporta, y que pueden resultar extraordinariamente interesantes:

- La biblioteca de clases *AFC:* Estas clases van a permitirnos crear interfaces de usuario más cercanas a lo que puede esperarse de una aplicación creada en el entorno *Windows* con otras plataformas de desarrollo como *Visual Basic o Visual C++*. Hablaremos de *AFC* con más detalle en el resto del artículo.
- La máquina virtual *Microsoft* para la plataforma *Win32:* Con esta entrega *Microsoft* se pone a la cabeza de las máquinas virtuales de *Java* para el entorno *Windows*. El nuevo producto nos va a permitir utilizar las funcionalidades de *JDK 1.1 (*aspectos de compatibilidad aparte), integrar *ActiveX* y *JavaBeans,* al tiempo que nos va a facilitar el acceso mediante *Java* al *API* de *Windows* si nuestras aplicaciones lo requieren. Multiplataforma, sí, *Java* es un lenguaje multiplataforma, pero si quiero crear una aplicación que sé que se va ejecutar en el entorno *Windows,* por qué no aprovechar las funcionalidades del sistema operativo.
- Gráficos avanzados: Se incorporan un conjunto de paquetes de clases que nos van a permitir crear aplicaciones que hagan uso intensivo de multimedia mediante *DirectX* o *DirectAnimation*.
- Acceso directo al *API:* Agrupado bajo el nombre de clases *JDirect*.

Las herramientas adicionales incluidas en el *SDK*

El conjunto de herramientas que nos proporciona el *SDK* es bastante impresionante. No es el objeto de esta entrega describirlas exhaustivamente, pero las enunciaremos:

- Compilador *Java*. El compilador *Java* de *Microsoft, jvc,* puede ejecutarse desde la línea de comandos, en lugar de hacerlo desde el entorno de desarrollo *Visual J++*
- Intérprete: Igualmente, el intérprete para ejecutar aplicaciones y *applets Java* desde la línea de comandos. El intérprete se denomina *Jview*.

- Intérprete en ventana: Se proporciona una versión del intérprete que crea una nueva ventana, separada de la ventana de comandos en la que se ejecuta el comando *wjview*.
- Visor de *applets:* Esta herramienta permite ejecutar un *applet* sin necesidad de navegador. Sería equivalente a *wjview,* pero de uso específico para *applets*.
- Conversor *AWT/ AFC:* La biblioteca de clases *AFC* extiende y mejora la funcionalidad de las clases incluidas en el *Abstract Windowing Toolkit,* siempre que los programas trabajen en el entorno *Windows* . Esta herramienta, *awt2afc.exe,* nos va a ayudar en el proceso de migrar nuestras aplicaciones o *applets* que utilicen *AWT* para construirlas con *AFC*. No realiza ningún cambio al código, sino que detecta las modificaciones necesarias y las sugiere mediante comentarios a nuestro programa.
- Creación de ficheros *CAB*. Los ficheros *CAB* sirven para la descarga de múltiples *applets* en un único fichero, comprimido. La utilidad *cabarc* permite la creación de estos ficheros a partir de nuestros *applets*.
- Herramientas de seguridad y validación de certificados.
- Visor de contenidos de un fichero .class: Una utilidad curiosa e interesante es *classvue*. Nos permite conocer algunas características de una clase a partir del *bytecode,* sin conocer el código fuente.

Figura A Salida de classview para una clase applet de ejemplo

- Creación de programas ejecutables desde programas *Java:* Como el lector ya sabrá sobradamente, y si no es así este curso no está funcionando como esperaba, los programas creados con *Java* no son propiamente ejecutables binarios, sino que se trata de *bytecodes* que son interpretados y ejecutados por la máquina virtual *Java*. La utilidad *jexegen* crea un ejecutable en plataforma *Windows* a partir de un fichero *.class*. esto puede resultar útil para agilizar la ejecución de programas escritos en *Java,* siempre que podamos asegurar que la plataforma en la que se ejecutarán es *Win32*. Es decir, escribe en *Java,* si te gusta, pero ejecuta en código nativo. También se incorpora otra utilidad que realiza conversiones pero hacia servicios para *Windows NT*.

- Conversor *java*/ C++: Conocido es el estrecho parentesco entre *Java* y C++. La utilidad *msjavah* convierte las definiciones de clase *Java* en archivos de cabecera .h, o mejor,
- *ActiveX/Java:* En *Visual J++ 1.1* ya se permitía, mediante un asistente incluido en el propio entorno, crear clases *Java* que envolviesen y facilitasen la utilización de controles *ActiveX* con código *Java*. Con este *SDK* se extiende esta posibilidad mediante la creación de *Java Beans* que envuelvan a los componentes *ActiveX*. Otra utilidad en este ámbito es *javareg*, que registra una clase *Java* como si de un componente *COM* se tratase.
- Generador de *GUID:* Se incluye la utilidad *guidgen* que facilita la creación de identificadores únicos para los componentes *COM* y *ActiveX*

La biblioteca de clases AFC

Bien, entremos en el punto central de la entrega del curso de este mes. *Microsoft* ha desarrollado una biblioteca de clases *(AFC, Application Foundation Classes)* centrada en la creación de elementos de la interfaz de usuario con mejores funcionalidades que las incorporadas en el *Abstract Windowing Toolkit* y que se someten a las especificaciones del

JDK 1.1. La mejora de rendimiento de las clases que representan a elementos de la interfaz de usuario se basa en que las clases base *AFC* son más pequeñas, con el consiguiente ahorro de memoria en las clases de la jerarquía., no poseen una ventana propia por componente, al contrario de lo que sucede con *AWT*.

Dentro de esta biblioteca encontraremos clases que representan botones, ventanas, y demás elementos de nuestras aplicaciones (el paquete *UI,* el más importante de la biblioteca), clases que mejoran y extienden las funcionalidades gráficas de la librería de clases (el paquete *fx*), además de implementar el modelo de eventos de *JDK 1.1*. Vamos a presentar las clases fundamentales de la biblioteca describiendo los paquetes que incluye:

Figura B Jerarquía de clases resumida del paquete UI

El paquete UI. Los elementos de la interfaz de usuario

Sin duda, este es el paquete estrella de la biblioteca. Contiene básicamente componentes, esto es, controles, paneles, ventanas, *applets* y cuadros de diálogo. Su estructura y modo de uso es muy similar a *AWT,* es decir, a las clases estándar del lenguaje que sirven a los mismos propósitos, pero *Microsoft* las ha diseñado procurando que posean una mayor potencia y rapidez de ejecución. Las clases análogas reciben el mismo nombre que sus parejas *AWT* añadiéndoles el prefijo *UI*. Así, por ejemplo, la clase *applet AFC* es *UIApplet*. No pueden mezclarse componentes de ambas librerías directamente, aunque existen mecanismos para hacerlo, que no detallaremos en este punto.

Una de las características fundamentales de estas clases es que son totalmente independientes de la plataforma. En *AWT* cada máquina virtual incorpora en la librería de clases el paquete *java.awt.peer* que contiene código que representa las particularidades de la plataforma concreta para la que la máquina real ha sido creada. Existe una clase para cada una de las clases del *AWT,* con el mismo nombre que su homóloga, pero con el sufijo *peer*. *AFC* no tiene clases *peer,* por lo que la librería no depende en ninguna de sus clases de la plataforma concreta.

Las controles disponibles en el paquete UI

Vamos a ver cuales son esos magníficos controles de los que podemos hacer uso en nuestras aplicaciones si utilizamos *AFC*. Los controles siguen el aspecto que se ha impuesto con *Internet Explorer 4.0*. Entre otros disponemos de botones de pulsación, menús (que permiten la inclusión de imágenes), listas desplegables, controles tipo árbol, cajas de texto multilínea, barras de desplazamiento, controles de progreso (los que se presentan, por ejemplo, mientras copiamos un fichero de una ubicación a otra del disco indicándonos el tanto por ciento completado de la operación), *tool tips* (las etiquetas que se nos presentan cuando dejamos reposar el ratón sobre un botón o menú indicándonos su funcionalidad) y otros.

Otros elementos que no son propiamente controles pero que también son representados por clases de este paquete son los cuadros de diálogo. Se incorporan a la biblioteca cajas de diálogo mono y multipágina *(property sheets*) y otros diálogos sofisticados como los que permiten elegir un color o una fuente y que tanto se utilizan en otros lenguajes, como las *MFC*. En las tablas adjuntas enumeramos las clases del paquete, agrupadas según su funcionalidad.

Control	Clase	Descripción
Etiqueta estática	*UIText*	Una etiqueta de texto estático
Imagen estática	*UIGraphic*	Una imagen estática, que podría ilustrar, por ejemplo, un botón
Etiqueta estática	*UIItem*	Una etiqueta estática que acepta texto o gráficos
Botón de pulsación	*UIPushButton*	Un botón de pulsación, cuyo aspecto puede personalizarse
Botón de opción	*UIRadioButton*	Permite seleccionar una de entre varias opciones
Check Box	*UICheckButton*	Permite indicar si una opción se escoge o no
Caja combinada *(combo Box)*	*UIChoice*	Una *comboBox*
Caja combinada editable	*UIEditChoice*	Una *comboBox* cuyo texto puede editarse
Lista de elementos	*UIList*	Una lista, con selección simple o múltiple
Menús	*UIMenuButton, UIMenuList, UIMenuItem*	Diferentes modos de presentar menús.
Caja de texto	*UIEdit*	Caja de edición de texto multilínea
Barra de desplazamiento	*UIScrollBar*	Barra de desplazamiento
Barra de botones desplazable	*UIBand, UIButtonBand*	Estas clases representan las nuevas barras de botones que pueden ocultar que aparecen, por ejemplo, en *Internet Explorer 4.*
Barra de estado	*UIStatus*	Barra de estado para presentar información al pie de las ventanas
Controles progreso	*UIProgress*	Control de progreso de operación
Control árbol	*UITree*	Nodo de un árbol, como por ejemplo los utilizados en el explorador de *Windows*
Visores	*UIScrollViewer, UIColumnViewer, UISplitViewer, UITabViewer*	Visores de información, en diversos formatos
Marquesina	*UIMarquee*	Permite desplazar cualquier otro control por encima de un panel contenedor

Diálogo	Clase	Descripción
Mensaje	*UIMessageBox*	Cuadros de mensaje, informativos, cuyo aspecto es totalmente configurable
Búsqueda/Sustitución	*UIFindReplaceDialog*	Cuadro de diálogo para buscar y sustituir texto
Color	*UIColorDialog*	Cuadro de diálogo para permitir al usuario escoger un color
Fuente	*UIFontDialog*	Cuadro de diálogo para permitir al usuario escoger una fuente

Página de propiedades	*UIPropertyDialog, UIPropertyPage*	Una *UIPropertyDialog* contiene una o varias *UIPropertyPage* para crear un cuadro de diálogo multipágina, es decir, una *property sheet.*
Asistente	*UIWizard, UIWizardStep*	Estas clases permiten implementar un cuadro de diálogo en múltiples pasos como los utilizados para los asistentes o *Wizards.*

Estos controles se utilizan de una manera totalmente análoga a como se haría en *AWT*. Para presentar un control es preciso disponer previamente de un objeto capaz de contenerlo. En *AWT* esta clase es *Container*, y sus derivadas, especialmente *Panel*, y en *AFC* juegan este papel las clases *UIContainer* y *UIPanel.*

Figura C Botones AFC

Las clases para manejo de gráficos. El paquete fx

El segundo paquete en importancia de *AFC,* a nuestro juicio, es el paquete gráfico *fx,* que contiene clases para realizar tareas de dibujo, incluyendo efectos sofisticados. Facilita el dibujo en los componentes *AFC,* que, como hemos comentado, carecen de ventana subyacente. Para los programadores en *Windows* los métodos de dibujo de estas clases van a resultarles más parecidos a los que ya están utilizando en sus programas.

En principio, la metodología que debe seguirse en estas clases para dibujar es la misma que la que ya se conoce en la biblioteca estándar de *Java* y *AWT*. Para llevar a cabo tareas de dibujo es preciso obtener un contexto gráfico, que en *AWT* era un objeto de la clase *Graphics,* mientras que en *AFC* será un ejemplar de la clase *FxGraphics.* Como el lector ya habrá supuesto, las clases *fx* reciben el mismo nombre que sus homólogas *AWT,* precedido por *fx.*

Una de las principales mejoras de este paquete es el soporte para texto multilengua, es decir, que con una única clase podemos representar texto en cualquier idioma, y que además puede presentarse orientado en cualquier dirección.

La clase fundamental es *FxGraphics.* Representa un contexto de dibujo extendido mediante el que creando elementos de dibujo, como un pincel o una fuente, y llamando a métodos de dibujo de la clase, podemos representar líneas, texto o figuras en la pantalla. Por ejemplo, la clase dispone de una función *drawLine* que dibuja una línea dadas sus coordenadas de inicio y final.

En la tabla adjunta mostramos las principales clases de este paquete

Objeto gráfico	Clase	Descripción
Contexto gráfico	*FxGraphics*	Contexto sobre el que podremos llevar a cabo las operaciones de dibujo
Color	*FxColor*	Representa un color con el que dinujar
Pincel	*FxPen*	Pincel para dibujar líneas, cuyo grosor y color son especificables
Pincel y Brocha	*FxBrushPen*	Representa a un color de fondo para rellenar figuras

		y un pincel para dibujar líneas
Pincel con estilo	*FxStyledPen*	Pincel con un estilo predeterminado de dibujo de línea
Textura	*FxTexture*	Permite rellenar una figura con una imagen de fondo de textura.
Curva	*FxCurve*	Clase abstracta, pero que sirve de base a *FxEllipse*, la clase utilizada para dibujar circunferencias y elipses.
Elipse	*FxEllipse*	Utilizable para dibujar circunferencias y círculos.
Fuente	*FxFont*	Representa una fuente, que puede crearse, modificarse y seleccionarse en un contexto gráfico.
Texto	*FxText*	Una cadena de caracteres de tamaño variable, similar a *StringBuffer* del paquete *lang*
Texto con formato	*FxFormattedText*	Texto con formato enriquecido, incluyendo alineamiento, justificación, formatos de fuente.
Gráfico vectorial	*FxGraphicMetaFile*	Permite almacenar y presentar archivos gráficos en el formato metaarchivo de *Windows*
Iconos	*FxSystemIcon*	Contiene variables de clase con iconos de sistema que suelen utilizarse en los cuadros de diálogo *AFC*
Region	*FxRegion*	Representa una región de forma arbitraria.
Caja de herramientas	*FxToolkit*	Contiene un conjunto de gráficos e imágenes que pueden sernos útiles.

Otras clases

La biblioteca contiene tres paquetes más, que no vamos a detallar de manera exhaustiva. Sin embargo, es interesante comentar, siquiera brevemente, alguna de las clases:

- Eventos: La biblioteca contiene un paquete completo, *com.ms.ui.event,* que sirven para permitir la utilización del nuevo modelo de eventos definido en el *JDK 1.1,* que no hace uso de la clase *Event.* Sin embargo, el modelo antiguo sigue estando vigente y puede usarse sin problemas en *AFC.*
- Gestión de plantillas de recursos *Win32: AFC* incluye un paquete de clases, *com.ms.ui.resource,* que van a permitirnos utilizar archivos con plantillas de recursos de *Windows.* Si el lector ha desarrollado alguna vez con *Visual C++,* o con *API,* conocerá la estupenda herramienta que suponen los archivos de recursos, esto es, plantillas que contienen descripciones de, por ejemplo, cuadros de diálogo, controles, etc. Estos recursos son creados y definidos visualmente por el programador utilizando un editor como por ejemplo el que se incluye en *Visual C++.* Posteriormente los ficheros de recursos pueden ser compilados e incluirse en nuestros proyectos sirviendo de aspecto a los cuadros de diálogo de nuestras aplicaciones. En *Java,* utilizando las clases de este paquete y concretamente *Win32SourceDecoder,* podemos utilizar las plantillas de recursos para dar aspecto a nuestros *applets* o aplicaciones sin necesidad de generar todos los controles escribiendo el código de instanciación de las clases de los controles.

- Gestión de archivos comprimidos de clase: En el ámbito del *WWW,* los *applets* se descargan del servidor hacia nuestro cliente a través de la red. Para facilitar la descarga de páginas que contengan un importante volumen de información en forma de ficheros .class asociados, es práctica común agrupar un conjunto de clases en ficheros .cab. Estos ficheros pueden crearse o manejarse mediante las clases del paquete *com.ms.util.cab*.

Figura D, figura E, figura F Algunas clases control

Utilizando los controles de *AFC*

En *AWT,* la ubicación de elementos de interfaz de usuario en una aplicación se basa en dos clases: *Component* y *Container*. La primera de ellas es la clase abstracta que sirve de base a cualquier elemento que puede aparecer en la pantalla en una aplicación *Java*. La segunda es una clase derivada de la primera que representa a cualquier componente que puede a su vez contener a otros componentes, es decir, *Container* representa un contenedor de componentes. En *AFC,* esta arquitectura se mantiene completamente con dos clases análogas a las anteriores: *UIComponent* y *UIContainer*.

Ejemplos de contenedores son los paneles *(Panel, UIPanel),* las ventanas *(Window, Frame, UIWindow, UIFrame),* los cuadros de diálogo *(Dialog, UIDialog),* y los propios *applets (Applet, UIApplet)*. Por su parte, los componentes no contenedores de uso más extendido son los controles, que no pueden contener ningún otro control.

Los componentes *AFC* pueden utilizarse en aplicaciones *AWT,* y viceversa. Para el primer caso, la librería *AFC* proporciona para cada una de las clases componentes una clase que sirve a modo de puente entre ambas bibliotecas, y cuyo nombre es idéntico al de la clase *AFC,* anteponiéndole el prefijo *Awt*. Por otra parte, los componentes *AWT* también pueden usarse en aplicaciones *AFC,* simplemente utilizando la clase *UIAwtHost,* que tiene el mismo objeto que las clases puente, pero en sentido contrario.

En este apartado vamos a ver cómo podemos utilizar los contenedores y los paneles para albergar controles *AFC*. Más adelante veremos cómo esos contenedores pueden ubicarse en los *applets* y las aplicaciones permitiendo que nuestros programas respondan a las interacciones del usuario. De eso se trata, ¿verdad?

La técnica más utilizada para albergar controles es crear una clase derivada de la clase *UIPanel*. Esta clase posee un método, *add,* que permite añadir al panel cualquier componente. Por ejemplo, en el siguiente código se añade a un panel un conjunto de controles.

```
class UnPanel extends UIPanel
{
  private UIPushButton botonpulsacion;
  private UIRadioButton botondeopcion;
  // Constructor
  public UnPanel()
  {

add(new UIText("Un conjunto de controles"));
add(botonpulsacion = new UIRadioButton("Pulsa, si te atreves",
UIPushButton.RAISED));
```

```
add(botondeopcion = new UICheckButton("Escoge",));
  }
}
```

El panel, como cualquier componente, es susceptible de ser modificado, en su aspecto, posición y demás características, haciendo uso de las funcionalidades de las clases gráficas. Así, podríamos añadir el siguiente código al panel anterior.

```
class UnPanel extends UIPanel
{
  private UIPushButton botonpulsacion;
  private UIRadioButton botondeopcion;
  // Constructor
  public UnPanel()
  {
setBackGround(FxColor.white);
add(new UIText("Un conjunto de controles"));
add(botonpulsacion = new UIRadioButton("Pulsa, si te atreves",
UIPushButton.RAISED));
add(botondeopcion = new UICheckButton("Escoge",));
  botonpulsacion.setBackground (FxColor.lightGray);
  }
}
```

• Cómo crear *applets* utilizando *AFC*

Bien, hemos visto un buen conjunto de clases que se supone van a llevarnos por el camino de la potencia y la buena programación en *Java,* clases que representan sobretodo los elementos del *AWT,* incluyendo cuadros de diálogo, controles de todo tipo, y, como no, *applets*. Todo esta información, evidentemente, no nos va a servir de nada si no sabemos como utilizar estas clases para crear *applets* o aplicaciones. Vamos a centrarnos en las técnicas necesarias para crear *applets*.

Crear un *applet* con *AFC* es una tarea algo más compleja que hacerlo con la biblioteca de clases estándar (la que incluye *AWT*). Es necesario añadir algo de código que haga de puente entre *AFC* y *AWT,* algo que, sin embargo, ya está previsto en la nueva librería de clases, que nos proporciona la clase *AwtUIApplet* a tal efecto. De este modo, un *applet AFC* requiere de la utilización de dos clases: la que representa al *applet* propiamente dicho: *UIApplet,* y la que sirve de puente entre el navegador y el *applet* la ya mencionada *AwtUIApplet*.

La clase puente *AwtUIApplet*

Esta clase, como hemos comentado, sirve únicamente para que un *applet AFC* pueda ejecutarse en los entornos *AWT* convencionales. Su código se limita prácticamente a un constructor que debemos llamar pasándole como argumento el objeto de la clase derivada

de *UIApplet* que va a representar al *Applet AFC* propiamente dicho. El código de la definición de una clase típica tal y como deberíamos escribirla podría ser el siguiente:

```
import com.ms.ui.*;

// Definición de la clase puente
public class MiAppletCompleto extends AwtUIApplet
{
  // Constructor que recibe como argumento mi clase derivada de
UIApplet
  public MiAppletCompleto () { super (new MiAppletAFC ()); }
}

// Mi Applet AFC
class MiAppletAFC extends UIApplet
{
  ...
}
```

El método constructor de la clase puente simplemente crea un ejemplar del *applet AFC,* y se lo pasa al constructor de la clase *AwtUIApplet,* para quien haga las tareas de intermediación entre *AFC y AWT,* sin que sea necesaria por nuestra parte ninguna otra labor. Esta clase va a ser la interlocutora del *browser* a todos los efectos, es decir, la que va a dar nombre a la clase que incluyamos en la etiqueta <APPLET> HTML, la que va a poder ser manipulada por código *script* o la que se comunicará con el resto de los objetos de la página *Web.*

La clase *applet AFC: UIApplet*

Una vez hemos creado la clase que va a permitir la inclusión de *applet* en la página *Web,* podemos proceder a escribir el código de nuestra clase *applet AFC* propiamente dicha. Esta clase, que deberá derivar de *UIApplet* es completamente análoga a *Applet,* de la librería convencional, y, por tanto, posee, entre otros, los métodos *init, start, stop, destroy*, ya conocidos.

• Un applet para dibujo utilizando AFC

Para ilustrar lo que hemos presentado hasta ahora vamos a modificar el *applet* que creamos el mes pasado, y que nos permitía dibujar líneas en un tablero de dibujo, para que utilice *AFC,* extendiendo además su funcionalidad. El *applet* ha sido desarrollado partiendo del mencionado ejercicio y de uno de los ejemplos que se proporcionan con el *SDK* de *Java,* concretamente el *applet FxShapes.* Recomendamos al lector que examine estos ejemplos, que os hemos hecho accesibles en la sede *Web* de *HISPAN TECNOLOGIC* http://www.hispan.com/, y que también pueden obtenerse, junto con el *SDK 2.0,* en la sede *Web* de *Microsoft.* Del ejemplo *FxShapes* podéis aprovechar el código de gestión de los paneles y de los *Insets,* que se utilizan para posicionar los controles en el panel del *applet.* La ubicación de controles en un *applet* es una técnica un poco engorrosa, que hace uso de un conjunto de clases denominados *Layout managers,* y que no hemos descrito para no complicar excesivamente el texto de este artículo.

El objetivo del *applet* es permitirnos dibujar líneas rectas, rectángulos y elipses, con trazos punteados o continuos, y de color arbitrario. Para escoger el tipo de figura y estilo de línea con el que se dibujará en cada momento, dispondremos en el *applet* sendos grupos de

botones de radio *AFC*. Para escoger el color de las líneas, colocaremos un botón de estilo plano que, tras ser pulsado, provocará la presentación de un cuadro de diálogo *AFC FxColorDialog*. Todas las clases de dibujo y controles utilizadas son *AFC*

Creación del proyecto

Para comenzar la creación del *applet* iniciaremos *Visual J++* y crearemos un nuevo proyecto mediante la opción de menú *File/New*. En el cuadro de diálogo resultante elegiremos la pestaña *Projects* y seleccionar como tipo de proyecto *Java Project*. Llamaremos al proyecto *Dibujo*. Una vez creado, el proyecto no contendrá fichero alguno.

Figura G Creación del proyecto

La clase *applet* de nuestra aplicación de dibujo

El siguiente paso es crear los ficheros fuentes necesarios para escribir el código de nuestro *applet*. Para ello, crearemos un nuevo fichero fuente *Java* mediante la opción de menú *File/New* tras lo que, en la pestaña *Files,* elegiremos la opción *Java Source File*. El primer fichero que vamos a crear será *Dibujo.java,* que contendrá el código de la clase *applet* y de la clase puente entre *AWT* y *AFC*. Todo lo que tenemos que escribir en la clase derivada de *UIApplet* es la redefinición del método *init,* que colocará en el *applet* un panel sobre el que colocar todos los controles. Este panel tendrá una cierta complejidad, pues deberá estar definido de manera que puedan posicionarse en él los botones con una cierta gracia. Para ello definiremos la clase *TableroDibujo,* en el fichero *TableroDibujo.java* que abordaremos inmediatamente después.

El código de *Dibujo.java* podría ser el siguiente:

```
// Curso de Visual J++. Dibujo con AFC

import java.awt.Insets;

import java.awt.Event;

import com.ms.ui.*;

import com.ms.fx.*;

public class Dibujo extends AwtUIApplet

{

public Dibujo() { super(new DibujoApplet()); }

}

// Applet CODE

class DibujoApplet extends UIApplet

{
```

```
public void init()

{//Creamos un Layout para colocar elementos en el applet

setLayout(new UIBorderLayout(0, 0));

setFont(new FxFont("Dialog", FxFont.PLAIN, 14));

// añade un panel de controles en el centro

add(new TableroDibujo(this), "Center");

}

}
```

El tablero de dibujo

Como hemos comentado, vamos a crear una clase que representará un panel sobre el que ubicaremos todos los controles. La clase, que denominaremos *TableroDibujo,* derivará de *UIPanel,* y utiliza una clase denominada *UIBorderLayout.*

Las clases de *Layout,* a las que pertenece *UIBorderLayout,* representan la disposición de los elementos que se incluyan en un contenedor. Por ejemplo, si un contenedor tiene asociado un *layout* vertical, representado por la clase *UIVerticalLayout,* los controles que vayamos colocando en el contenedor se dispondrán uno debajo del otro, en una columna o varias, dependiendo de las características del *layout. UIBorderLayout* representa una disposición tal y como la que se aprecia en la figura, utilizando los puntos cardinales.

Figura H Disposición de los elementos en un BorderLayout

La clase *TableroDibujo* es, pues, un panel que adopta un *UIBorderLayout* tal y como se aprecia en el siguiente fragmento de código.

```
import java.awt.Event;

import java.awt.Image;

import java.awt.Insets;

import com.ms.ui.*;

public class TableroDibujo extends UIPanel implements
DefinicionesConstantes{

private PanelDibujo pntpnl;

private PanelBotones btns;
```

```
public TableroDibujo(UIApplet applet)

{

//Creamos el layout

setLayout(new UIBorderLayout(0,0));
```

El panel va a contener dos subpaneles, que vamos a crear en dos clase más, posteriormente. El primero de estos paneles, que demominaremos *PanelDibujo*, representa el tapiz en el que vamos a dibujar realmente, mientras que el segundo, *PanelBotones*, contendrá los botones de radio y el botón de pulsación para escoger las opciones de dibujo. Ubicaremos los subpaneles en el panel *TableroDibujo* utilizando el siguiente código:

```
// Creamos un grupo de botones y lo añadimos al panel...

SDKInsetGroup main = new SDKInsetGroup("Un applet para dibujo con
AFC", 20,10,10,10);

// lo dividimos verticalmente

main.setLayout(new UISplitLayout(0, 130));

// Creamos un panel de dibujo

pntpnl = new PanelDibujo();

// Creamos un grupo de controles

UIGroup grupo = new UIGroup("Área de dibujo");

//Decimos que tendrá distribución con bordes

grupo.setLayout(new UIBorderLayout());

// Colocamos el panel de dibujo en el centro

grupo.add(pntpnl, "center");

// Creamos el panel de botones

btns = new PanelBotones(pntpnl);

//colocamos el panel de botones a la izquierda, y el borde a la
derecha

main.add(btns, "nw"); main.add(new SDKInsetPanelBL(grupo, 0,5,0,0),
"se");
```

```
//añadimos en el panel principal el grupo de botones

add(main, "center");
```

Como se habrá podido ver, la clase implementa una interfaz, *DefinicionesConstantes*, cuyo único objetivo es definir un conjunto de constantes, que como el lector ya sabrá, no pueden definirse en *Java* como variables globales, ya que estas no existen. Las constantes se utilizarán, más adelante, para permitir determinar el estado de dibujo en el que nos encontramos, es decir, la figura y estilo de línea a dibujar.

```
public interface DefinicionesConstantes

{

public static final int ID_ELIPSE = 1;

public static final int ID_LINEA = 2;

public static final int ID_POLY = 3;

public static final int ID_RECT = 4;

public static final int ID_SOLIDO = 5;

public static final int ID_PUNTEADO = 6;

public static final int FT_INSET = 16;

public static final int FT_MIN_WIDTH = 80;

public static final int FT_MIN_HEIGHT = 98;

}
```

Los botones del *applet*

Como se ha comentado, vamos a disponer en la parte izquierda de la superficie de nuestro *applet* una serie de botones que nos permitirán personalizar nuestro dibujo. Estos botones se colocarán sobre un panel *(PanelBotones)* con una disposición de *layout* vertical, es decir, en una columna. Tendremos un primer grupo de botones de radio, de la clase *UIRadioButton,* agrupados mediante la clase *UIGroup* (que representa un conjunto de botones con una etiqueta, y de los que sólo uno puede estar seleccionado), que permitirán seleccionar si deseamos dibujar una línea, una elipse o un rectángulo. La definición de la clase y la del primer grupo de botones podría ser la que sigue

```
class PanelBotones extends UIPanel implements DefinicionesConstantes
```

```
{ // estructura los botones en la aplicación de dibujo

private UIRadioButton elipse, linea, rect;

private UIRadioButton solido, punteado;

private UIPushButton colorbtn;

private UIColorDialog dialogo;

private UIFrame marco;

private PanelDibujo pntpnl;

private UIRadioGroup figuras, estilopen;

private UIGroup escogercolorgrp;

public PanelBotones(PanelDibujo pntpnl)

{

this.pntpnl = pntpnl;

//Grupo de botones para elegir el tipo de figura a dibujar

figuras = new UIRadioGroup("Figura");

elipse = (UIRadioButton)figuras.add("Elipse");

elipse.setID(ID_ELIPSE);

linea = (UIRadioButton)figuras.add("Línea");

linea.setID(ID_LINEA);

rect = (UIRadioButton)figuras.add("Rectángulo");

rect.setID(ID_RECT);
```

Un segundo grupo de botones de radio, definibles de manera totalmente análoga, nos permitirán elegir si deseamos que las líneas se dibujen como trazos continuos o discontinuos. Finalmente el tercer grupo va a contener un botón de pulsación, que definiremos con estilo plano, que nos permitirá escoger el color de las líneas:

```
// Create UIGroup el que pondremos un botón para escoger el color de
línea

escogercolorgrp = new UIGroup("Escoger color");
```

```
escogercolorgrp.setLayout(new UIBorderLayout());

colorbtn = new UIPushButton("Elegir color", 0);

escogercolorgrp.add(colorbtn, "center");
```

En esta clase deberemos añadir además el código para gestionar las pulsaciones sobre los botones. Las que se produzcan sobre los botones de radio se gestionan con el método *handleEvent,* mientras que cuando pulsamos el botón de elección del color se disparará la ejecución del método *action.* Remitimos al código fuente para una descripción del método *handleEvent,* pero vamos a detenernos un poco en el método *action.* Cuando se pulse el botón crearemos un cuadro de diálogo de la clase *UIColorDialog,* mediante el que el usuario podrá escoger el color de las líneas. Este diálogo es modal, es decir, que debe ser cerrado antes de poder proseguir con la ejecución del *applet.* Para poder abrir un diálogo es necesario tener una ventana marco, de la clase *UIFrame,* que obtendremos mediante el método *FxToolkit.getHelperUIFrame().* Esta ventana servirá como padre del diálogo. Para presentar el diálogo modal llamaremos al método *show* del objeto *UIColorDialog.* Cuando el usuario pulsa el botón *OK* del diálogo, tras elegir el color, este se halla accesible utilizando el método *getColor* del objeto diálogo.

```
public boolean action (Event evt, Object arg)

{

if ( arg instanceof UIButton ) {

if (arg == colorbtn) {

marco = FxToolkit.getHelperUIFrame();

dialogo = new UIColorDialog(marco);

dialogo.show();

pntpnl.colordibujo = dialogo.getFxColor();

}} return super.action(evt, arg);

}
```

Figura I El aspecto del *UIColorDialog*

El panel de dibujo

El segundo panel del tablero de dibujo es el tapiz donde los dibujos se llevarán realmente a cabo. Este panel se implementará mediante la clase *PanelDibujo*, en el fichero del mismo nombre.

Este fichero, para permitir que puedan dibujarse múltiples figuras, cada una de ellas con un color y estilo diferente, incluye tres clases, *Linea, Rectangulo* y *Elipse,* que contienen propiedades representando el *pincel FX* con el que se dibujan (representado por la clase *FxStyledPen*) y los puntos para delimitar cada una de las figuras. Como puede verse en el código de definición de una de ellas, *Elipse,* la figura tiene un método, *dibujar*, que permite que la figura se dibuje a sí misma, recibiendo como argumento el contexto gráfico del panel en el que se halla, un objeto de la clase *FxGraphics*.

```
class Elipse implements Dibujable{

private Point pini;

private int ancho, alto;

private FxStyledPen color;

public Elipse( Elipse e) {

this.pini = new Point(e.pini.x, e.pini.y);

ancho = e.ancho ;

alto = e.alto ;

this.color = new FxStyledPen(FxStyledPen.PLAIN, 1);

this.color = e.color;

}

public Elipse(Point p1, Point p2, FxStyledPen c) {

ancho = Math.abs(p2.x - p1.x);

alto = Math.abs(p2.y - p1.y);

pini = new Point(p1.x, p1.y);

if ( pini.x > p2.x ) pini.x = p2.x;

if ( pini.y > p2.y ) pini.y = p2.y;

this.color = new FxStyledPen(FxStyledPen.PLAIN, 1);

this.color = c;

}

public void dibujar(FxGraphics g) {

g.setColor(color);
```

```
g.drawOval(pini.x, pini.y, ancho, alto);

}

public Point inicio(){

return pini;

}

}
```

El resto de las clase figuras son totalmente análogas, con la salvedad de las rutinas de dibujo concretas. Como puede apreciarse, la clase implementa la interfaz *Dibujable*, que hemos creado con motivaciones exclusivamente docentes, para ilustrar el mecanismo de interfaces:

```
interface Dibujable

{

public void dibujar(FxGraphics g);

}
```

El resto del código del fichero está dedicado a la definición de la clase *PanelDibujo*. Esta clase contiene una propiedad *figuras*, de la clase *Vector,* del paquete *java.util*. La clase *Vector* representa un *array* dinámico y heterogéneo de objetos, que nos va a permitir almacenar cualquiera de las figuras, y dibujarla cuando la recuperemos, en función de cual sea su clase. Ello es posible por la determinación de la clase en tiempo de ejecución. Es decir, almacenaremos las figuras a medida que las vayamos dibujando en el vector *figuras,* y, al dibujarlas, utilizaremos el método *dibujar* propio de cada una de ellas.

```
public class PanelDibujo extends UIPanel implements
DefinicionesConstantes

{

private FxGraphics Graf;

private int mododibujo;

private int estilopen;

private FxStyledPen pincel;
```

```
public FxColor colordibujo;

private Vector figuras;

private Point inicioactual, finalactual;

public PanelDibujo(){

colordibujo = new FxColor (0, 0, 255);

setLayout(new UIBorderLayout());

mododibujo = ID_LINEA; estilopen = ID_SOLIDO;

// Creación de los pinceles

pincel = new FxStyledPen(FxStyledPen.PLAIN, 1, colordibujo);

Graf = getGraphics();

figuras = new Vector();

}
```

Para dibujar, vamos a utilizar cuatro métodos, los de control de la pulsación del ratón, *mouseDown, mouseUp* y *mouseDrag,* y el método *paint().* El modo de dibujo resulta trivial (sí, ya sé que esta afirmación puede resultar enojosa, pero pediría al lector un poco de paciencia en el estudio) examinando el siguiente código:

```
public boolean mouseDrag(Event evt, int x, int y) {

finalactual = new Point(x,y);

repaint();

return true;

}

public boolean mouseUp(Event evt, int x, int y) {

finalactual = new Point(x,y);

switch ( estilopen ) {

case ID_SOLIDO:

pincel = new FxStyledPen(FxStyledPen.PLAIN,1, colordibujo);
```

```
break;
case ID_PUNTEADO:
pincel = new FxStyledPen(FxStyledPen.DOT,1, colordibujo);
break;
}
switch ( mododibujo ) {
case ID_LINEA:
figuras.addElement( new Linea(inicioactual, finalactual, pincel));
break;
case ID_ELIPSE:
figuras.addElement(new Elipse(inicioactual, finalactual, pincel) );
break;
case ID_RECT:
figuras.addElement( new Rect(inicioactual, finalactual, pincel));
break;
}
finalactual = null;
repaint();
return true;
}
public void paint(FxGraphics fxg) {
Linea l;
Elipse e;
Rect r;
for( int i=0; i<figuras.size(); i++){
if( figuras.elementAt(i) instanceof Linea) {
l = new Linea( (Linea) figuras.elementAt(i));
```

```
l.dibujar(fxg);
}else if(figuras.elementAt(i) instanceof Elipse) {
e = new Elipse( (Elipse) figuras.elementAt(i));
e.dibujar(fxg);
}else if(figuras.elementAt(i) instanceof Rect) {
r = new Rect( (Rect) figuras.elementAt(i));
r.dibujar(fxg);
}
}
if(finalactual != null){
switch ( estilopen ) {
case ID_SOLIDO:
pincel = new FxStyledPen(FxStyledPen.PLAIN,1, colordibujo);
break;
case ID_PUNTEADO:
pincel = new FxStyledPen(FxStyledPen.DOT,1, colordibujo);
break;
}
switch ( mododibujo ) {
case ID_LINEA:
l=new Linea(inicioactual, finalactual, pincel);
l.dibujar(fxg);
break;
case ID_ELIPSE:
e =new Elipse(inicioactual, finalactual, pincel);
e.dibujar(fxg);
break;
```

```
case ID_RECT:

r = new Rect(inicioactual, finalactual, pincel);

r.dibujar(fxg);

break;

}

}
```

Figura J Aspecto del applet

Una vez el código ya está acabado incluiremos todos los ficheros en el proyecto, mediante la orden *Project / Add Files,* y modificaremos sus opciones de compilación mediante la orden *Project/Settings*. De estas opciones la más importante es la que indica la clase que debe ejecutarse, que será *Dibujo.class*. Si lo deseamos, podemos escribir también un fichero *HTML* para albergar el *applet,* aunque si no lo hacemos el entorno nos creará uno automáticamente, que será suficiente en general. El aspecto del *applet,* en pleno funcionamiento puede apreciarse en una figura adjunta.

Un nuevo mes, y con él una nueva entrega de este curso con el que espero que estéis familiarizándoos con la plataforma de desarrollo en *Java* de *Microsoft.* El mes pasado, una vez descritos los aspectos básicos relacionados con la programación en *Java,* comenzamos la presentación del *SDK 2.0* con la librería de clases *AFC.* Recordemos que *SDK* son las siglas de *Software Development Kit,* que podría traducirse como el *kit* de desarrollo de *Software,* y hace referencia a un conjunto de elementos que *Microsoft* ha publicado para permitir a los desarrolladores crear aplicaciones en *Java* en los ámbitos de los sistemas operativos de *Microsoft.* El kit puede descargarse de la sede *Web* de *Microsoft* http://www.microsoft.com/Java/

Como ya comentamos, *SDK 2.0* incluye la nueva máquina virtual de *Microsoft,* la librería de clases *AFC,* un conjunto de herramientas adicionales, aplicaciones de ejemplo y una extensa documentación que nos va a facilitar la tarea de desarrollar tanto aplicaciones completas como *applets* que cumplan las especificaciones del estándar *JDK 1.1* de *Sun.*

En esta entrega vamos a prestar atención a uno de los aspectos novedosos de *SDK 2.0* que más recelos y suspicacias puede provocar entre los desarrolladores en *Java: J/Direct.* Esta librería va a permitirnos realizar llamadas a funciones almacenadas en Bibliotecas de Enlace Dinámico *(Dynamic Link Libraries)* desde nuestros programas escritos en *Java.*

Dónde queda la independencia de la plataforma

¿No habíamos quedado que *Java* era independiente de la plataforma? Esta pregunta es perfectamente adecuada y no puedo reprochar al lector que se decida a formularla. Una *DLL* es un fichero binario, compilado, y, por lo tanto, contiene código nativo. El código nativo, por supuesto, sólo es válido para la plataforma para la que fue compilado, ya que contiene el código binario asociado a un cierto juego de instrucciones de un cierto

procesador. La independencia de la plataforma queda, pues, violada con la utilización de código nativo.

Sin embargo, debemos hacer notar que la independencia de la plataforma en *Java* es una prestación del lenguaje, no una obligación. Del mismo modo en que un detenido tiene derecho a guardar silencio pero puede hablar tanto como desee si se atiene a las consecuencias, si renunciamos a la independencia de la plataforma será por alguna buena razón, la primordial, porque conocemos el entorno operativo en el que el programa va a ejecutarse. Considerarlo de otro modo llevaría a la indeseable conclusión de que *Java* sólo es útil en entornos heterogéneos, con lo que no sería de interés, por ejemplo, para crear una aplicación *Windows*.

En cualquier caso, la inclusión de llamadas a código nativo no es una práctica exclusiva de *Microsoft* a través de *JDirect,* también *Sun,* en *JDK 1.1,* especifica un estándar para la llamada a código nativo denominado *Java Native Interface (JNI).* Incluso es conveniente notar que la inclusión de métodos nativos ya era posible anteriormente utilizando *Raw Native Interface (RNI).*

Java Native Interface define cómo llamar desde nuestras clases *Java* a funciones (métodos de clases *Java*) cuya implementación no está escrita en *Java* y compilada a *bytecode,* sino en un lenguaje cualquiera, típicamente en C o C++, y compilada a código nativo. Por otra parte, *JNI* también sirve de interfaz para que pueda manipularse desde el código nativo a los objetos *Java.* Esta interfaz se expresa en un conjunto de funciones que permiten el acceso, manipulación y creación de objetos *Java.*

En clara analogía con este planteamiento, *Microsoft* ha definido *J/Direct,* un estándar para poder realizar llamadas a código nativo, incluido en *DLL's.* Esta situación resulta especialmente interesante ya que el que el propio sistema operativo, el *API* de *Win32,* está constituido como un conjunto de *DLL* de sistema. Además, *J/Direct* hace el proceso de llamada en el entorno *Windows* mucho más sencillo que si se utiliza *JNI.* Simplemente deberemos declarar la función que deseamos llamar utilizando *@dll.import,* una primitiva similar a *import,* que recordemos se utilizaba para la importación de paquetes en *Java.*

• ¿Y por qué *J/Direct*, si ya existía *RNI?*

La razón fundamental es la simplicidad. Utilizar *RNI* provoca ciertos problemas a la hora de acceder a *DLL,* sobre todo desde el punto de vista de los nombres y de la liberación de la memoria, que, como ya sabemos, es un proceso que en *Java* se realiza de manera automática utilizando el *garbage collector.* Todas estas tareas de armonización de la liberación de memoria del código nativo y el interpretado deben llevarse a cabo manualmente por parte del programador si utiliza *RNI,* mientras que *J/Direct* se encarga de todo de manera transparente y automática.

Otro aspecto esencial de la sencillez de *J/Direct* radica en que es la propia librería la que se encarga de llevar a cabo todo el engorroso proceso de traducción de tipos entre los programas en *Java* y el código nativo, normalmente escrito en C. Si tenemos que llamar en nuestro programa a una función que recibe unos argumentos de un cierto tipo, será necesario tener unos tipos análogos en *Java. J/Direct* se encarga de estas traducciones sin que apenas tengamos que escribir código adicional. Por ejemplo, si una función de *API*, escrita en C, recibe como argumento un puntero a carácter que representa una cadena,

podemos escribir, utilizando *J/Direct,* un método en *Java* que estuviese vinculado con el *API,* pero que recibiese un argumento *String; J/Direct* se encargará de las traducciones.

RNI, sin embargo, tiene algunos ámbitos en los que es preferible a *J/Direct,* sobre todo en el acceso a objetos y en la rapidez en la carga de clases. Por ello, y gracias a que pueden utilizarse conjuntamente, los programadores en *Java* que estuvieran familiarizados con *RNI,* pueden seguirlo utilizando, al tiempo que incorporan *J/Direct* en los aspectos en los que les simplifique la vida.

• Declaración de métodos nativos

Para poder llamar a un fragmento de código compilado a código nativo es necesario que tenga un nombre que sea utilizable en una clase en *Java.* Dicho de otro modo, todo el código en *Java* está dentro de definiciones de clase, y sólo pueden ejecutarse fragmentos de código en forma de métodos de clase.

Según esta explicación, para poder ejecutar una función en código nativo es necesario que esta función sea un método de la clase en la que se va a llamar. Para hacer esto posible, el lenguaje incluye la palabra clave *native* que utilizaremos según la siguiente sintaxis

```
public static tipo native nombremétodo(listadeargumentos);
```

Como puede verse en esta definición, no se proporciona código para este método, sino que simplemente se declara para que el nombre sea conocido dentro de la definición de la clase y, por tanto, pueda usarse. Sin embargo, es preciso decir al compilador dónde encontrar ese código nativo para llevar a cabo la tarea que se supone el método va a aportar.

• Las directivas de importación

J/Direct facilita la inclusión de llamadas a funciones de *DLL* mediante la definición de tres directivas para el compilador *Java,* que serán las que indicarán que el código debe importarse utilizando *J/Direct,* y no *Raw Native Interface,* además de informar al compilador donde encontrar el código asociado a un cierto método definido como nativo.

Definición de métodos nativos: @dll.import

Esta directiva permite indicar al compilador que un cierto método nativo se halla definido en una *DLL:* Deberemos escribir esta directiva justo antes de la definición del método, indicando el nombre de la *DLL.*

La sintaxis para escribir esta directiva, y las otras dos que comentaremos posteriormente, es un poco enrevesada. El código, para que cumpla la sintaxis de *Java,* que no incluye entre sus palabras clave esta directiva, debe estar escrito como si se tratase de un comentario, para que sea así ignorado por el compilador. Veamos cómo hacerlo.

```
/**@dll.import("Nombre de la DLL",<Modificadores>)*/
...Declaración de método...;
```

La declaración de método sigue la sintaxis que presentamos anteriormente. Debo comentar que la sintaxis es muy estricta, de modo que no podemos dejar espacio en blanco en su interior

Veamos la sintaxis con un poco más de detalle. El parámetro `Modificadores` tomará diversos valores en función de la situación. Si la DLL que estamos importando es una función del *API* de *Windows,* es decir una de las *DLL* de sistema, no será necesario dar ningún valor de `modificador` como puede verse en el siguiente ejemplo:

```
/** @dll.import("USER32") */
private static native int MessageBox(int hwndOwner, String text,
String title, int fuStyle);
}
```

Como vemos, en ese código se indica que el método a importar será *MessageBox,* que se halla en la *DLL* de sistema *User32.dll*

El nombre del método en *Java* no tiene por qué ser el mismo que el de la función dentro de la DLL. Este proceso en el que damos un nombre diferente a un método nativo que el que tiene el código binario que lo sustenta en la DLL se denomina *aliasing*. Para poder utilizar esta posibilidad deberemos indicar en la directiva `@dll.import` el nombre que el método recibe en la *DLL*, o el orden del mismo dentro de la biblioteca. Esto puede hacerse proporcionando estos valores en `modificador` según la siguiente sintaxis:

```
/**@dll.import("Nombre DLL",entrypoint = nombre método)*/
/**@dll.import("Nombre DLL", entrypoint = #número de método)*/
```

Utilizando la primera de las posibilidades podríamos, por ejemplo, crear un método en una clase en *Java* que presentase un mensaje en una caja de diálogo, equivalente a lo que supondría la función *MessageBox* del *API,* pero adaptada al castellano. Para ello deberíamos escribir el siguiente código:

```
  /** @dll.import("USER32", entrypoint="MessageBox") */
  static native int mensaje(int nIndex);
```

La segunda sintaxis de la directiva permite utilizar funciones exportadas por *DLL* por su orden dentro del fichero, en lugar de por su nombre, como vemos a continuación:

```
  /** @dll.import("MiDll", entrypoint="#3") */
  public static native void Metodo3();
```

Definición de clases nativas: @dll.import

Una manera alternativa de utilizar esta directiva es definiendo como nativa una clase completa, en lugar de hacerlo método a método. Cuando una clase es definida como nativa se sobreeentiende que todos sus métodos son considerados como nativos.

Para definir una clase como nativa basta con anteponer a la definición de la clase la directiva `@dll.import`:

```
/** @dll.import("dll") */ class nombreclase
{
  public static native int método1 ();
  public static native int método2(String s);
  public static native boolean método2 ();
}
```

que sería equivalente a

```
class nombreclase
{
/** @dll.import("dll") */
  public static native int método1 ();

/** @dll.import("dll") */
  public static native int método2(String s);

/** @dll.import("dll") */
  public static native boolean método2 ();
}
```

Representación en *Java* de estructuras C: @dll.struct

Esta directiva permite definir una clase *Java* que represente una estructura del *API*. Una estructura, en este contexto, hace referencia a una entidad *struct* en C/C++*:* Como hemos visto anteriormente en este curso, en *Java* no existen las estructuras, por lo que se imporne la creación de un tipo de datos equivalente, que no puede ser otra cosa que una clase.

Deberemos escribir esta directiva justo antes de la definición de la clase que va a representar a la estructura y su sintaxis es la siguiente

```
  /**@dll.struct(<ansi/unicode/auto>,<pack=tamaño paquete>)*/
  ...declaración de clase..;
```

El primer parámetro especifica cómo se va a realizar la traducción entre cadenas de caracteres C y *String Java*. Supongo al lector informado de que *UNICODE* es una especificación de codificación de caracteres de 16 bits, aceptada por el lenguaje *Java*, y que se está imponiendo por su facilidad de codificar todos los juegos de caracteres necesarios en uno sólo. Evidentemente, la codificación *UNICODE* difiere de la *ANSI,* por lo que podemos escoger cómo deseamos que se haga la traducción de tipos en la utilización de la estructura nativa C. El segundo parámetro carece de interés en este punto ya que hace referencia al almacenamiento interno de la estructura en la memoria. Veamos como ejemplo la utilización de la estructura *SYSTEMTIME,* del *API* de *Windows*

```
typedef struct {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME;
```

Esta estructura C, puede ser utilizada en *Java* si la proporcionamos una definición de clase, precedida por la directiva `@dll.struct`

```
/** @dll.struct() */
class SYSTEMTIME {
  public short wYear;
  public short wMonth;
  public short wDayOfWeek;
```

```
  public short wDay;
  public short wHour;
  public short wMinute;
  public short wSecond;
  public short wMilliseconds;
}
```

Un par de aspectos a destacar. En primer lugar vemos que se ha producido una conversión del tipo `WORD` a `short`. Comentaremos posteriormente algunos aspectos relativos a la conversión de tipos. Por otra parte, esta definición de clase que representa a la estructura `SYSTEMTIME` no la hemos inventado, sino que se encuentra definida en el paquete com.ms.`win32` que describiremos más adelante.

¿Para qué necesitamos definir esta estructura? Bien, hace apenas unas líneas hemos presentado cómo podemos llamar a una función del *API* simplemente definiendo un método nativo en una clase y vinculándolo a la función en cuestión. El problema es que muchas de las funciones del *API* reciben como argumento estructuras del *API,* por lo que será ineludible definir alguna estructura de datos en *Java* que nos lo permita. Por ejemplo, si definimos la estructura `SYSTEMTIME` que presentamos hace un momento y la importamos en un fichero *Java,* podemos definir en su interior el siguiente método nativo

```
...
/** @dll.import("KERNEL32") */
    static native void GetSystemTime(SYSTEMTIME pst);
```

que después podrá ser utilizado, en algún punto del código del fichero del siguiente modo:

```
      SYSTEMTIME hora = new SYSTEMTIME();
      GetSystemTime(hora);
      System.out.println("Son las " + hora.wHour+ " horas, " +
hora.wMinute + " minutos, "+ hora.wSeconds + " segundos);
```

Definición de arrays en estructuras: @dll.structmap

En las estructuras C que pretendamos utilizar en *Java* mediante clases y la directiva `@dll.struct`, comúnmente encontraremos arrays, entendidos como vectores con un determinado número de elementos. Este aspecto es importante, estamos hablando de cadenas de caracteres de tamaño fijo o arrays estáticos, no de punteros de asignación dinámica de memoria.

Para definir estos arrays en *Java* haremos uso de una nueva directiva: `@dllstructmap`, que, como las anteriores, se consignará justo antes de la definición del *array Java,* siguiendo una de las dos siguientes sintaxis:

```
/**@dll.structmap([type=TCHAR[Tamaño]])*/

/**@dll.structmap([type=FIXEDARRAY,size=n])*/
```

La primera de las dos sintaxis se utiliza para incluir campos de una estructura que representan a cadenas de caracteres de tamaño fijo en C. El argumento *Tamaño* representará al número de caracteres del *array,* incluyendo el carácter nulo de terminación de cadena en C.

Como ejemplo podríamos tener la estructura LOGFONT, que representa en *API* a las características de una fuente en *Windows*. Uno de los campos de esta estructura es lfFaceName, una cadena de 32 caracteres que incluye el nombre de la fuente, es decir, *Arial, Times,* etc. Ve

```
typedef struct tagLOGFONT {

LONG lfHeight;

LONG lfWidth;

LONG lfEscapement;

LONG lfOrientation;

LONG lfWeight;

BYTE lfItalic;

BYTE lfUnderline;

BYTE lfStrikeOut;

BYTE lfCharSet;

BYTE lfOutPrecision;

BYTE lfClipPrecision;

BYTE lfQuality;

BYTE lfPitchAndFamily;

TCHAR lfFaceName[32]; } LOGFONT
```

Esta estructura puede representarse en *Java* utilizando *J/Direct,* de hecho así la encontramos en LogFont.Java, de la siguiente manera:

```
/** @dll.struct(auto) */

public class LOGFONT {

public int lfHeight;

public int lfWidth;
```

```
public int lfEscapement;

public int lfOrientation;

public int lfWeight;

public byte lfItalic;

public byte lfUnderline;

public byte lfStrikeOut;

public byte lfCharSet;

public byte lfOutPrecision;

public byte lfClipPrecision;

public byte lfQuality;

public byte lfPitchAndFamily;

/** @dll.structmap([type=TCHAR[32]]) */

public String lfFaceName;
```

La segunda sintaxis es utilizada para arrays de longitud limitada, pero no de caracteres. Recordemos que la definición de un *array* en *Java* se hacía en dos pasos. El primero de ellos creaba una variable que iba a albergar un array, pero no suponía ningún tipo de inicialización de la misma:

```
int miarray[];
```

En el segundo paso se le daba tamaño, se le asignaba memoria y se inicializaban sus elementos mediante el operador `new`

```
miarray = new int[17];
```

Dentro de la definición de una clase esto no es posible, por lo que si deseásemos incorporar la siguiente estructura, que contiene arrays de tamaño fijo:

```
struct miestructura
{
        SHORT   uno[10];
        int     dos[4];
        float   tres[14];
};
```

deberíamos hacerlo de la siguiente manera

```
/** @dll.struct() */
class miestructura
{
    /** @dll.structmap([type=FIXEDARRAY, size=10]) */
```

```
    short uno[];

    /** @dll.structmap([type=FIXEDARRAY, size=4]) */
    int dos[];

    /** @dll.structmap([type=FIXEDARRAY, size=14]) */
    float tres[];
}
```

• Conversión de tipos nativos a tipos *Java*

Un aspecto que aunque ya hemos comentado, merece que nos detengamos en él siquiera brevemente, es la conversión de tipos desde el lenguaje en el que estará escrita la *DLL* (en nuestro caso C) y los tipos aceptables en *Java*. Con *SDK 2.0,* su máquina virtual y *J/Direct,* la conversión de tipos se lleva a cabo automáticamente en la mayoría de las situaciones. Es decir, si la máquina virtual, en un método declarado como nativo y compilado a *bytecode,* encuentra un argumento de la clase *Java* `String`, entenderá que el código nativo al cual el método se halla vinculado recibirá una cadena de caracteres como un puntero a caracteres terminado por un carácter nulo. Para definir correctamente nuestros métodos nativos en nuestras clases *Java* resulta, pues, poco menos que imprescindible conocer cómo se realizan las conversiones.

Conversión de tipos numéricos

Los tipos numéricos se convierten de la manera más natural, es decir, hacia el tipo C que representa el mismo tipo de cantidad, con un similar tamaño en *bytes*. En la siguiente tabla detallamos la conversión de tipos

Tipo Java	Tipo API C
byte	BYTE o CHAR
short	SHORT o WORD
int	INT, UINT, LONG, ULONG, o DWORD
long	__int64
float	float
double	double

Los *arrays* de tipos numéricos se convierten a punteros al mismo tipo. Por ejemplo, un *array* de enteros `int [ ]`, se convierte en un `INT *`

Un tipo que no podemos considerar propiamente numérico, al menos en *Java,* pero que sí lo es en C, y por ello lo incluimos en esta categoría, es `boolean`, que se convierten a `BOOL`.

Conversión de caracteres y arrays o cadenas de caracteres

Los caracteres, representados por el tipo `char` en *Java,* se convierten al tipo *API C* `TCHAR`. Por defecto la conversión se lleva a cabo utilizando la codificación *ANSI,* pero si se especifica lo contrario en las directivas de importación es posible realizar conversión a

caracteres `UNICODE`. Por otra parte, las cadenas de caracteres de tamaño fijo `String`, se convierten a `LPCTSTR` y las de tamaño variable `StringBuffer` al tipo `LPCTSTR`

Los arrays de caracteres, como hemos visto sucedía con el resto de arrays en *Java,* se convierten a punteros al tipo base. Así, un array a carácter `char[ ]`, se transforma en el tipo `TCHAR *`

Conversión de punteros

Como el lector sabrá de sobras en *Java* no existen los punteros, al menos explícitamente. También le supongo informado de que los punteros son utilizados en C con extraordinaria frecuencia. ¿Cómo resolver esta importante diferencia?

En primer lugar, cuando necesitemos pasar un puntero a un método en *Java*, podemos hacerlo pasando un *array* con un único elemento. Esta técnica es utilizada cuando deseamos tratar en *Java* el caso del paso de parámetros a métodos por referencia utilizando punteros en C. Es una práctica muy común en las funciones del *API* recibir como argumento un puntero a una variable, cuyo valor será modificado en el interior de la función precisamente a través de ese puntero.

Veamos como ejemplo la siguiente función

```
BOOL Funcion1( DWORD *argumento1, DWORD *argumento2);
```

Una posible llamada a esta función en C podría ser

```
Función1( &variable1, &variable2);
```

donde `variable1` y `variable` 2 son dos variables de tipo `DWORD`.

El método nativo *Java* se definiría del siguiente modo:

```
/** @dll.import("Midll") */
  private native static boolean Funcion1(
                    int variable1[], int variable2[]);
```

y la llamada a este método en cualquier otro punto del programa en el que sea accesible, en el interior del método `main` de la propia clase, por ejemplo, podría ser algo similar a lo siguiente:

```
public static void main(String args[])
  {
    int variable[] = {0};
...
    Funcion1(variable1);
...
}
```

Por otra parte, los punteros son iguales en tamaño a los enteros en *Java,* por lo que podemos almacenar en un entero la dirección de memoria que se alberga en un puntero en C.

• Paquetes de clases y *J/Direct*

Evidentemente, no podíamos añadir ninguna funcionalidad a nuestra plataforma *Java* sin incorporar un nuevo conjunto de clases, es decir, un conjunto de paquetes. Los paquetes se añaden al fichero classes.zip ubicado en el subdirectorio `\windows\java\classes`, en *Windows 95,* tal y como se muestra en la figura A (paquetes de java). De este subdirectorio se derivarían todos los subdirectorios que representan la estructura de paquetes de la librería de clases de *Java.*

Dos paquetes son añadidos a nuestro entorno en la instalación de *SDK 2.0:* `com.ms.dll` y `com.ms.win32`

com.ms.dll: Las clases de la interfaz *J/Direct*

Este paquete, las clases que contiene, en realidad, nos facilita la utilización de funciones de *DLL,* dándonos control sobre aspectos avanzados de la importación de código nativo, tanto si este se incluye en *DLL's* de sistema *(Win32 API)* como si se trata de código de *API OLE*, o simplemente funciones exportadas por cualquier *DLL* creada por el usuario. El paquete se instala automáticamente con el *SDK 2.0*. En la tabla adjunta se presentan las clases incluidas en `com.ms.dll`

Clase	Descripción
`Dlllib`	Contiene métodos para obtener información y realizar acciones con métodos importados mediante la directiva `@dll.import`
`Callback`	Permite la creación de *callbacks* a las librerías del *API Win32*
`Root`	Permite controlar la liberación de memoria para ciertos objetos, protegiéndolos del *Garbage collector*
`Win32Exception`	Gestión de errores devueltos por *Win32*
`ParameterCountMismatchError`	Permite el control del número de parámetros pasados a un método

Clases de acceso a API Win32: *com.ms.win32*

Las clases que nos van a permitir acceder al *API* de *Windows* utilizando *J/Direct* se hallan en el paquete `com.ms.win32`. La instalación de *SDK 2.0* no instala por defecto este paquete, sino que debemos hacerlo manualmente una vez el *SDK* se haya instalado. También están disponible los códigos fuente. En la figura adjunta

Hemos visto que para utilizar en *Java* una función del *API* bastaba con definir una clase que contuviese un método declarado como nativo que se asociase a la función deseada. Sin embargo, esto no es en general necesario si se instalan las clases de este paquete, que contienen definiciones de clase que siguen la sintaxis y tecnología *J/Direct* para muchas de las funciones, estructuras y constantes definidas en *Win32*. Nuestro único trabajo será importarlas en nuestras clases, y, evidentemente, utilizarlas. Ejemplos de estas clases ya se han visto en este texto; basta con recordar la clase definida para la estructura `LOGFONT`.

Las clases incluidas en el paquete pueden clasificarse a tres categorías:

Clases que representan las funciones del API

Un primer conjunto de clases proporcionan definiciones de clases que importan en métodos nativos las funciones incluidas en las *DLL* de sistema siguientes:

- `Kernel32.dll`: *API* básico
- `Gdi32.dll`:*API* para gráficos
- `User32.dll`:Funciones relacionadas con la interfaz de usuario
- `Advapi32.dll`:*API* de criptografía
- `Shell32.dll`*API* del explorador de *Windows*
- `Winmm.dll`Multimedia
- `Spoolss.dll`:*API* de impresión

Cada *DLL* está representada por un fichero *.java,* del mismo nombre, que contiene una clase también homónima, con un conjunto de métodos nativos, ya definidos. Destaquemos que estos métodos son estáticos por lo que no es preciso instanciar ningún ejemplar de la clase en cuestión para poder utilizar el método, que indirectamente va a conllevar la llamada a una función del *API:* Basta con importar la clase que representa a la *DLL* en cuestión y llamar a sus métodos a través del propio nombre de la clase.

A modo de ejemplo presentamos el inicio del fichero `User32.java`, código fuente de `User32.class`, cuya clase `User32`, contiene definiciones nativas *Java* para importar las funciones de `User32.dll`, utilizando *J*/`Direct`

```
// Copyright (C) 1997 Microsoft Corporation All Rights Reserved

// These classes provide direct, low-overhead access to commonly used

// Windows api. These classes use the new J/Direct feature.

// Information on how to use J/Direct to write your own declarations

// can be found in the Microsoft SDK for Java 2.0.



package com.ms.win32;



/** @security(checkClassLinking=on) */

public class User32 {

/** @dll.import("USER32",auto) */

public native static int ActivateKeyboardLayout (int hkl, int Flags);



/** @dll.import("USER32",auto) */
```

```
public native static boolean AdjustWindowRect (RECT lpRect, int dwStyle, boolean bMenu);



/** @dll.import("USER32",auto) */

public native static boolean AdjustWindowRectEx (RECT lpRect, int dwStyle, boolean bMenu, int
dwExStyle);

...
```

Clases que contienen constantes

El paquete contiene la interfaz win, que a su vez basa su funcionalidad en un conjunto de interfaces que contienen el código necesario para importar las definiciones de constantes del API. Concretamente, existe una interfaz para cada una de las letras del alfabeto, del tipo `wina.java`, `winb.java`, *etc. Cada una de ellas contiene las definiciones para todas las constantes que comienzan por la letra que sigue al prefijo* `win`. *Así, tal y como se ve a continuación, en el fragmento inicial del fichero* `wina.java`, *esta interfaz define las constantes que comienzan por la letra a*

```
  package com.ms.win32;



public interface wina {

int APPCMD_CLIENTONLY = 0x00000010,

APPCMD_FILTERINITS = 0x00000020,

APPCMD_MASK = 0x00000FF0,

APPCLASS_STANDARD = 0x00000000,

APPCLASS_MASK = 0x0000000F,

APPCLASS_MONITOR = 0x00000001,

ATTR_INPUT = 0x00,

...
```

Por su parte el código de la interfaz `win`, *que será la que efectivamente usaremos en nuestras propias clases, es el siguiente*

```
package com.ms.win32;

public interface win extends wina, winb, winc, wind, wine, winf, wing, winh,

wini, winj, wink, winl, winm, winn, wino, winp,

winq, winr, wins, wint, winu, winv, winw, winx,

winy, winz, winmisc, windynamic

{
```

```
}
```

Una vez más, para utilizar esta interfaz bastará con escribir un código como el que sigue:

```
import com.ms.win32.*;

System.out.println("IDOK = " + win.IDOK);
```

Estructuras

Por último, y como vimos en la estructura `LOGFONT`, cada estructura API está definida mediante una clase con el mismo nombre

• Un ejemplo

Para finalizar este capítulo del curso ilustrando algunos de los conceptos expuestos vamos a crear un pequeño proyecto utilizando Visual J++ que presentará un mensaje en pantalla, utilizando la función MessageBox del API. Este mensaje nos indicará la memoria que tenemos disponible, haciendo uso de una estructura definida en el API y que tendremos que importar.

Un aspecto importante es configurar Visual J++ 1.1 para que utilice el compilador y la máquina virtual incluidos con SDK 2.0. Este aspecto es importantísimo, ya que si no lo hiciésemos así, no podríamos utilizar J/Direct y el compilador nos presentaría errores. Para ello deberemos acudir al menú Tools/Options y en el cuadro de diálogo que se presenta en la figura escoger la pestaña Directories. Escogiendo como plataforma la Java Virtual Machine y como directorios de elección los de Executable files, deberemos añadir, en la cabecera de la lista, el subdirectorio donde tengamos instalados el compilador y el JView del SDK 2.0, en nuestro caso D:\SDK-JAVA2.0\BIN.

Figura 0. Añadiendo el directorio del compilador SDK 2.0

Una vez hecho esto crearemos un nuevo proyecto, del tipo Java Project, que llamaremos JavaDLL, y le añadiremos un fichero fuente: JavaDLL.java

Figura B. Creando el proyecto

Figura C. Añadiendo el fichero fuente

En este fichero fuente, importaremos la función MessageBox, que se halla en Util32.dll, y GlobalMemoryStatus de Kernel32.dll. El código que proponemos en primer lugar, que se muestra a continuación, importa MessageBox utilizando la directiva @dll.import("user32") y GlobalMemoryStatus utilizando la misma directiva para kernel32.dll

```
import com.ms.dll.DllLib;

public class JavaDLL

{



public static void main(String args[])
```

```
{

MEMORYSTATUS mstatus = new MEMORYSTATUS();

String cadena;



cadena = "Memoria disponible: " + Integer.toHexString(mstatus.dwTotalPhys);

GlobalMemoryStatus(mstatus);



try {



MessageBox(0, "Memoria disponible: " + mstatus.dwAvailPhys, "Curso VJ++ / HISPAN TECNOLOGIC",
0);



} catch (UnsatisfiedLinkError ule) {

System.err.println("Se ha detectado una excepción: " + ule);

System.err.println("¡No tienes instalado J/Direct!.");

}

}





/** @dll.import("USER32") */

static native int MessageBox(int hwndOwner,

String text,

String title,

int style);



/** @dll.import("KERNEL32") */

static native void GlobalMemoryStatus(MEMORYSTATUS lptMemStat);

}
```

```
/** @dll.struct() */

class MEMORYSTATUS {



public int dwLength = DllLib.sizeOf(MEMORYSTATUS.class);

public int dwMemoryLoad;

public int dwTotalPhys;

public int dwAvailPhys;

public int dwTotalPageFile;

public int dwAvailPageFile;

public int dwTotalVirtual;

public int dwAvailVirtual;

}
```

Sin embargo, lo mismo puede hacerse haciendo uso de las clases del paquete com.ms.win32, tal y como veremos a continuación:

```
import com.ms.dll.DllLib;

import com.ms.win32.*;



public class JavaDLL

{

public static void main(String args[])

{

MEMORYSTATUS mstatus = new MEMORYSTATUS();

String cadena;

cadena = "Memoria disponible: " +
Integer.toHexString(mstatus.dwTotalPhys);

Kernel32.GlobalMemoryStatus(mstatus);

try {
```

```
User32.MessageBox(0, "Memoria disponible: " +
mstatus.dwAvailPhys, "Curso VJ++ / HISPAN TECNOLOGIC", 0);

} catch (UnsatisfiedLinkError ule) {

System.err.println("Se ha detectado una excepción: " +
ule);

System.err.println("¡No tienes instalado J/Direct!.");

}

}



}



/** @dll.struct() */

class MEMORYSTATUS {



public int dwLength = DllLib.sizeOf(MEMORYSTATUS.class);

public int dwMemoryLoad;

public int dwTotalPhys;

public int dwAvailPhys;

public int dwTotalPageFile;

public int dwAvailPageFile;

public int dwTotalVirtual;

public int dwAvailVirtual;

}
```

El resultado de ambas aplicaciones se presenta en la figura D.

En esta entrega vamos a dedicarnos a describir las particularidades de *Visual J++* en una de las vertientes fundamentales de la tarea de desarrollar aplicaciones: el acceso a bases de datos. El lector será consciente de que la mayoría de las aplicaciones empresariales acaban en el almacenamiento u obtención de información de una fuente estructurada de datos, sea esta una base de datos relacional, un conjunto de archivos del sistema de fichero, o mensajes de correo almacenados por un sistema de mensajería. *Java* no es una excepción: como lenguaje de propósito general proporciona mecanismos para acceder a bases de datos.

El soporte de bases de datos en el lenguaje *Java,* en cuanto a estándar universal definido por *Sun,* tiene un nombre: *JDBC*. Posteriormente describiremos esta tecnología, y como podemos utilizarla con *Visual J++,* pero podemos adelantar que *JDBC* es una estupenda, coherente e inteligente estrategia para acceder a bases de datos con el lenguaje *Java,* que se integra perfectamente con el espíritu y filosofía del lenguaje en cuanto a su universalidad e independencia de la plataforma. *Visual J++,* tal y como el producto fue lanzado al mercado, no soporta la programación con *JDBC,* pero el *SDK 2.0* vino a cubrir esa carencia.

Sería imperdonable, por mi parte y sin embargo, obviar los cambios a los que *Visual J++* va a verse sometido en la nueva versión, y por ello me propongo presentar al lector hacia donde se dirigen los pasos de la tecnología *Microsoft* en este ámbito. En definitiva, podríamos dividir el contenido de este artículo en tres partes: el acceso a bases de datos proporcionado por *Visual J++ 1.1;* el soporte *JDBC* añadido en el *SDK 2.0,* y las nuevas tecnologías de acceso a datos añadidas en *Visual J++ 6.0*.

Aspectos previos: qué significa la gestión de datos

En un sistema de bases de datos existe invariablemente un elemento que resulta imprescindible: el gestor de bases de datos. Este gestor o motor de bases de datos se encargará de gestionar las informaciones contenidas en la mismas, facilitándonos el acceso a ellas y manejándolas de manera que no se produzcan errores ni incoherencias en la adición y eliminación de datos. Ejemplo de gestores de bases de datos podrían ser *Access, Microsoft SQL Server, Interbase, Oracle* o *Paradox*.

La mayoría de los gestores de bases de datos en la actualidad son relacionales. Las bases de datos relacionales se diseñan a partir de un modelo que nos facilita entender los datos como algo *independiente* del modo de almacenamiento y manipulación. Esto hace que en teoría pueda utilizarse cualquier aplicación para acceder a cualquier base de datos.

La independencia de los datos y su implementación en las bases de datos relacionales permite crear una misma base de datos en diversos gestores. Es decir, puede diseñarse una base de datos que funcione con cualquier gestor. Para que este ejercicio sea posible será necesario, evidentemente, utilizar tan sólo las estructuras mínimas que son comunes a todos los gestores.

Interfaz entre las aplicaciones y el gestor

La multiplicidad de gestores de bases de datos, con arquitecturas y sintaxis diversas, conlleva en que las aplicaciones que acceden a los datos deban, en principio, someterse a las particularidades del gestor a la hora de interaccionar con él. Esto motivaría que las aplicaciones sirviesen exclusivamente para un gestor en particular y que fuese necesario, para el desarrollador, conocer multitud de diferentes sintaxis de acceso.

Para evitar estos problemas se han creado diversas interfaces que permiten un acceso a las diversas plataformas de datos utilizando medios comunes. Esta interfaz suele estar conformada por un *driver* que estandariza las llamadas a los gestores de bases de datos. De este modo se definen funciones que pueden ser utilizadas para acceder a los datos de cualquier gestor que disponga del citado *driver*. Este traduce las llamadas al lenguaje y sintaxis propia del gestor.

La interfaz de bases de datos a la que podremos hacer referencia en *Visual J++* es *Open Database Connectivity (ODBC):* Esta interfaz es la más utilizado para los gestores de bases de datos relacionales en el entorno *Windows*. Para cada uno de esos gestores se dispone de un gestor que permite a los desarrolladores acceder a los datos desde cualquier plataforma que disponga de esta funcionalidad. *ODBC* define una serie de funciones *API* que pueden ser utilizadas en los programas para acceder a las bases de datos del gestor

JDBC es otra interfaz que va a permitirnos comunicarnos bidireccionalmente con los gestores. En este caso, *JDBC* tiene exclusiva aplicación en *Java,* ya que se trata de una interfaz orientada a objetos escrita en este lenguaje.

OLE DB es un conjunto de interfaces de reciente aplicación, que *Microsoft* propone como extensión de *ODBC* y que tiene como principal aplicación plasmar una nueva tecnología denominada *UDA (Universal Data Access)* y que se basa en permitir el acceso a cualquier tipo de información estructurada, sea o no una base de datos relacional, como podría ser el sistema de archivos, por ejemplo, siempre que se disponga del *driver* adecuado, que en esta tecnología se llama *provider*.

Por encima de *OLEDB* y *ODBC, Microsoft* propone un nuevo modelo de acceso a bases de datos denominado *ADO (Active Data Objects). ADO* consiste en una interfaz orientada a objetos para la programación de aplicaciones de muy sencillo uso que, apoyándose en *OLEDB* y por consiguiente en *UDA,* permite acceder tanto a bases de datos relacionales como a otras fuentes de información estructuradas

• Visual J++ 1.1 y el acceso a datos

Como comentamos en otras entregas de este curso , *Sun* y *Microsoft* están tratando de imponer su plataforma tecnológica, entre otros aspectos, para el acceso a bases de datos con *Java*.

Sun ha desarrollado un marco de conectividad a bases de datos, denominado *JDBC,* que supone un avance importante, pero al que *Microsoft* no se incorporó inicialmente. *Visual J++ 1.1,* tal y como el producto fue publicado, daba soporte para los ya tradicionales *RDO (ODBC)* y *DAO,* que se desarrollaron para las otras plataformas de desarrollo de la casa. Esta situación se halla en clara contradicción con la premisa fundacional de la transportabilidad del código de *Java,* al hacer uso de técnicas propietarias de *Microsoft. Visual J++,* por tanto no posee ningún asistente que genere código basado en *JDBC* y las clases de este entorno deben obtenerse separadamente en *SDK 2.0,* tal y como comentaremos posteriormente.

Visual J++ 1.1 facilita sin embargo una herramienta, el *Database Wizard for Java,* para el acceso a bases de datos desde un *applet* en un documento *Web*. En un proyecto *Web* el acceso a bases de datos puede servir, por ejemplo, para almacenar información recabada a los usuarios, para obtener información que presentar a los mismos o, simplemente, para almacenar los nombres de usuario y contraseña de los socios de una sede de acceso restringido. *Visual J++,* por otra parte y en el ámbito de *Visual Studio,* incorpora otras herramientas muy valiosas para la creación y gestión de bases de datos *ODBC*, de aplicación especialmente con *SQL Server: Visual Database Tools*. Estas herramientas están disponibles con la versión empresarial de *Visual Studio* e incluyen un diseñador de consultas y uno de bases de datos. Para los proyectos que se creen o gestionen utilizando estas herramientas el entorno de desarrollo añade un nuevo visor, *Data View*, que presenta de manera

jerarquizada los diferentes objetos de las bases de datos asociadas a las fuentes de datos *ODBC* de nuestro proyecto de bases de datos.

Visual Database Tools, por tanto, nos permiten crear y modificar bases de datos de *Microsoft SQL Server 6.5,* diseñar los objetos de esas base de datos, conectarse a bases de datos *ODBC,* ejecutar consultas y añadir y eliminar datos e integrar los datos con proyectos existentes.

• Acceso a datos mediante *DAO* y *RDO*: *Database Wizard for Java*

DAO y *RDO* son dos tecnologías ya conocidas por los programadores, sobre todo si desarrollan con *Visual C++* o *Visual Basic,* para el acceso a bases de datos. Las citadas estrategias de desarrollo con bases de datos facilitan la tarea con abstracciones de diferente naturaleza que la hacen más simple e independiente del gestor. Así, *DAO* proporciona una interfaz orientada a objetos para acceder a bases de datos del motor *Microsoft Jet,* o a fuentes *ODBC,* mientras que *RDO* permite la gestión de datos relacionales exclusivamente a través de *ODBC* resultando más adecuada que *DAO* a estos efectos.

Visual J++ 1.1 incorpora soporte para el acceso a bases de datos utilizando cualquiera de estas dos tecnologías. De hecho se proporciona con el entorno un asistente que automatiza algunos aspectos de la tarea. *Database Wizard for Java* crea de manera automática *applets* que incluyan el código necesario para acceder a datos, al tiempo que genera un documento *HTML* en el que estos datos se muestren. Este acceso no será tan sólo para la lectura o consulta de los mismos, sino, bien al contrario, para la actualización o adición de nuevos registros. Sin embargo, el *applet* creado tiene algunas limitaciones que deberán ser resueltas manualmente. Por ejemplo, no permite crear consultas que obtengan datos de más de una tabla

Para utilizar este asistente, deberemos acudir al menú *File / New / Projects/ Database Wizard for Java*

Figura C Database Wizard

Figura D Paso 1 de Database Wizard

A lo largo de los pasos del asistente podremos introducir el tipo de fuente de datos: *DAO* o *RDO*. Si se escoge *DAO* deberá especificarse la ubicación del fichero *.mdb* de la base de datos a acceder. Si se escoge *RDO* se deberá introducir el nombre de una conexión de datos *ODBC*. Además debe indicarse cual será el objeto que nos servirá como origen de los datos, usualmente una tabla, el nombre de la clase del *applet* y si deseamos que la conexión a datos sea de sólo lectura, o de lectura y escritura.

Una vez completado el proceso del asistente el entorno creará un proyecto con las clases necesarias para acometer el acceso a los datos. Según las opciones elegidas a lo largo del asistente, se crearán tres ficheros*:*

- Proyecto*.java:* Este fichero contiene la definición de la clase del *applet* y otras relacionadas con el acceso a datos. Su nombre será el mismo que el del proyecto, en nuestro ejemplo: *Biblioteca.java*

- Proyecto.*html:* Fichero *html* en el que se inserta el *applet* y que presenta los registros de la consulta. También se incluirán botones de navegación.
- *Alert.java:* Proporciona un cuadro de diálogo que se mostrará si hay errores en el trabajo del *applet*

En cuanto a las clases que el asistente crea para nuestro *applet,* son las siguientes:

- Proyecto: Clase del *applet,* con el mismo nombre del proyecto, en nuestro caso *Biblioteca.* Incluye datos miembros para la cadena de caracteres que define la fuente de datos, los campos de la tabla resultado, etc
- *DBField:* Clase que define el nombre del campo y su tipo. La clase pricipal contiene un *array* de objetos de esta clase para definir los campos que se presentarán.
- *DBFrame:* Ventana marco en la que ubica el *applet*
- *Alert:* Presenta un cuadro de diálogo en situaciones de error.

Estas clases pueden ser modificadas para hacer que la interfaz de usuario sea más cercana a nuestros deseos, así como para permitir que la base de datos acceda a más de una tabla, si eso es necesario..

Figura E Página Web creada por Database Wizard for Java

• *JDBC y Visual J++.* Amigos irreconciliables

JDBC es la clave para el acceso a bases de datos utilizando una solución *sólo Java* que satisfaga a los puristas de la filosofía del lenguaje. *JDBC* es un *API* en *Java* que consiste en una serie de clases e interfaces escritas en el lenguaje *Java,* que proporcionan un mecanismo para escribir aplicaciones en *Java* que accedan a bases de datos. Al contrario de lo que sucede con *ODBC*, *JDBC* no es un acrónimo, es decir, sus letras no corresponden a siglas. Podríamos decir que *Sun* ha jugado con la similitud con *ODBC* de manera que el significado de la nueva palabra sea casi inmediato para los potenciales usuarios.

Al igual que lo que sucede con *ODBC, JDBC* permite crear aplicaciones de bases de datos con una gran independencia del gestor concreto al que se accederá. Al crear la aplicación se utilizan las clases y el *API JDBC* y será esta interfaz la que se encargará de enviar los comandos adecuados al gestor

JDBC es una interfaz de acceso a datos de bajo nivel

Esto quiere decir que se utiliza para ejecutar comandos *SQL* de manera directa. En cualquier caso, pueden crearse interfaces de más alto nivel que se apoyen en *JDBC,* y que resulten más agradables para su utilización en entornos de desarrollo visuales. De hecho se están desarrollando dos *API's* de alto nivel que se apoyan en *JDBC:*

- *SQL* Inmerso en Java *(Embedded SQL)*
- Clases para la representación de bases de datos relacionales: En esta visión, cada tabla será una clase y cada fila de la tabla será un ejemplar de la clase

JDBC versus ODBC

Microsoft ODBC (Open DataBase Connectivity) es en la actualidad el *API* mas utilizado para acceder a bases de datos relacionales, pues es razonablemente sencillo y permite la conexión de virtualmente todas las plataformas. Si esto es así, ¿para qué se necesita *JDBC.?*

Existen numerosas razones de las que enunciaremos las siguientes:

- *ODBC* no es del todo apropiada para el uso directo con *Java* porque está diseñada en C y eso motiva que la aplicación *Java* que utiliza *ODBC* pierda algunas de sus características de seguridad y portabilidad. Por otra parte, *Java* no tiene punteros, por lo que la traducción del *API ODBC* a *Java* no es sencilla. De hecho, puede pensarse en *JDBC* como una traducción de *ODBC* de un modo orientado a objetos que es apropiado para la programación en *Java.*
- *ODBC* no es precisamente simple, coexistiendo aspectos sencillos con otros muy complejos. *JDBC*, ha sido diseñado para parecer sencillo, aunque pueden complicarse las cosas tanto como se desee.
- *JDBC* es necesario para una solución totalmente *Java,* ya que *ODBC* requiere la instalación del *driver manager* y los *drivers* en la máquina cliente. La portabilidad es, pues, más problemática.

Tipos de drivers JDBC

Los *drivers JDBC* se pueden clasificar en cuatro tipos:

- Puente *JDBC-ODBC (*Nivel I): Permite utilizar *drivers ODBC* como *drivers JDBC.* Este tipo de solución traduce las llamadas a los métodos *JDBC* a llamadas *ODBC,* y requiere que existan los *drivers ODBC* necesarios en la máquina, con lo que la portabilidad se resiente. Sin embargo, facilita la utilización de *JDBC* con gestores de bases de datos que aun no proporcionan *drivers JDBC* puros, pero que sí facilitan los homólogos *ODBC.*
- *Drivers JDBC* de acceso al *API* del gestor (Nivel II): Estos *drivers,* llamados habitualmente *Native API partly Java,* traducen las llamadas *Java* al *API* del gestor de datos, con lo que se requiere código binario que recoja esa llamada al *API* del gestor, con lo que elimina la capa *ODBC,* con la consiguiente mejora de rendimiento, pero no se elimina la necesidad de código binario dependiente de la plataforma en el cliente
- *Drivers JDBC* de red (Nivel III): En este caso las llamadas son dirigidas por el *driver* a través de un protocolo de red al gestor. Esto es conveniente cuando el protocolo de red y el del gestor no coinciden. Este tipo de *drivers* ya no requieren de ningún tipo de código binario en el cliente.
- *Driver JDBC nativo* (Nivel IV)*:* Este tipo de *Drivers* conforman una solución completamente *Java,* que traduce las llamadas directamente el protocolo de red utilizado por el gestor, normalmente *TCP/IP* a través de *Internet*

Utilización de JDBC con Visual J++

Como ya hemos comentado anteriormente, *Visual J++* no soporta directamente *JDBC,* y será necesario instalar las clases necesarias, agrupadas en el paquete *java.sql*, bien del *JDK 1.1* de *Sun,* o del *SDK 2.0* de *Microsoft.* Este último paquete, objeto de algunas entregas de este curso, nos proporciona también un puente *JDBC/ODBC* para uso con la máquina virtual *Java* de *Microsoft (Microsoft Java VM).* A continuación enumeramos las clases e interfaces incluidas en este paquete

Clases

- Date
- DriverManager
- DriverPropertyInfo
- Time
- Timestamp
- Types

Interfaces

- CallableStatement
- Connection
- DatabaseMetaData
- Driver
- PreparedStatement
- ResultSet
- ResultSetMetaData
- Statement

El proceso básico para utilizar *JDBC* en nuestro código comienza con importar el paquete *java.sql,* y se completa con una serie de pasos sencillos:

- Apertura de una conexión, mediante la utilización de la interfaz *Connection.* El primer paso para abrir la conexión es registrar uno o varios *drivers JDBC* haciendo uso de la clase *DriverManager.* Para ello recomiendo al lector utilizar el método *Class.forName*, que cargará de manera explícita la clase del *driver* cuestión:

```
Class.forName("driver.name")
```

Una vez el *driver* está disponible, procederemos realmente a abrir la conexión, mediante el método *getConnection* del *DriverManager,* que nos devolverá un objeto que implementa la interfaz *Connection* y que representa a una conexión con la fuente de datos solicitada.

Este método recibe como argumento una cadena de conexión, análoga a la que se utiliza al establecer una conexión *ODBC,* y que contiene tres partes: la cadena "*JDBC",* que identifica siempre al protocolo de conexión*;* el subprotocolo, que será el nombre del *driver* a usar o *ODBC,* si se utiliza el puente; y el nombre del servidor o el *DSN* de la fuente de datos *ODBC.* El siguiente podría ser el método para crear una conexión a la base de datos *Neptuno* de *Access,* siempre que utilicemos el puente, y que exista una fuente de datos en nuestro sistema con el mismo nombre que la base de datos

```
Connection c = DriverManager.getConnection("JDBC:ODBC:dsn=Neptuno;")
```

- Creación de la sentencia SQL para el acceso a los datos. El objeto conexión nos proporciona un conjunto de objetos que van a permitirnos crear y remitir la consulta, mediante las interfaces *Statement (*para la ejecución de consultas simples), *PreparedStatement* (que facilita el uso de procedimientos almacenados en el servidor y que acepta parámetros de entrada) y *CallableStatement (*que igualmente nos permite ejecutar procedimientos almacenados pero con la funcionalidad añadida de soportar el retorno de parámetros de salida)

Si nos centramos en *Statement,* esta clase provee de tres métodos que nos permitirán ejecutar sentencias *SQL: executeQuery, execute,* y *executeUpdate*. *ExecuteQuery* envía una sentencia SQL y retorna un *ResultSet* con los registros obtenidos, mientras que *Execute* permite obtener múltiples *ResultSets*. *ExecuteUpdate* es utilizado para enviar a gestor sentencias que no son estrictamente consultas, sino sentencias de actualización tales como *INSERT, UPDATE,* o *DELETE,* y devuelve un número indicando el número de registros afectados.

- Gestión y manejo de los resultados de la consulta, haciendo uso de un *ResultSet*. Esta clase proporciona un conjunto de métodos para navegar por los registros que contiene. En cada registro podemos acceder a los valores de cada uno de los campos refiriéndonos bien a su nombre o a su posición relativa. El siguiente código obtendría los registros contenidos en la tabla *titles* de la base de datos *pubs* de *SQL Server*, y abordaría la navegación por ellos.

```
Connection con = DriverManager.getConnection (

"jdbc:odbc:pubs", "sa", "password");

Statement stmt = con.createStatement();

ResultSet rs = conn.executeQuery("select * from titles")

while (rs.next()) {

String s = getString("b");

...

}
```

El acceso a bases de datos con Visual J++ 6.0

Visual J++ 6.0 está llamado a convertirse en una plataforma controvertida ya que ahonda aún más en la vinculación e integración de *Java* con el sistema operativo *Windows* y las tecnologías propias de *Microsoft*. Así, este producto va a resultar realmente maravilloso para aquellos desarrolladores que deseen crear aplicaciones en *Java* destinadas a *Windows,* con una rapidez y facilidad de desarrollo totalmente equiparables, incluso en la filosofía *RAD,* a *Visual Basic,* por ejemplo. Por otra parte, sus prestaciones resultarán heréticas para los partidarios de la plataforma *Sun*. Sin embargo, y a pesar de que vamos a dejar la descripción exhaustiva de *Visual J++ 6.0* para la entrega del mes que viene, creo que es de justicia comentar, de entrada, que si bien *Visual J++ 6.0* va bastante más allá de lo que se solicita de un compilador *Java,* permite su uso estrictamente restringido a la plataforma estándar, ya que las extensiones *Microsoft* no son en absoluto de uso obligatorio. Es decir, podemos crear programas y *applets* multiplataforma.

En lo que a la entrega de este mes se refiere, vamos a centrarnos en las funcionalidades de la nueva plataforma de desarrollo en cuanto a la creación de aplicaciones de acceso a bases de datos. *Visual J++* 6.0 está diseñado para la creación de aplicaciones cliente servidor multicapa que utilicen las últimas tecnologías de *Microsoft* para el acceso a bases de datos: *OLEDB* y *ADO*. Para ello proporciona diversos modos: la utilización de un asistente análogo a *Database Wizard for Java*, pero extraordinariamente más potente; la vinculación *RAD* de elementos de un formulario a campos de una base de datos; o la utilización de *J/ADO,* una tecnología que permite el acceso en *Java* a *ADO,* encapsulándolo y proporcionando toda su funcionalidad. En todas ellas se dispone de la posibilidad de crear aplicaciones tanto en *bytecode Java* portable tradicional, como el empaquetamiento de las clases en ficheros *.cab* de fácil distribución a través de *Internet* como la compilación a código nativo, útil si la portabilidad no es nuestra preocupación. Así mismo, se nos permite crear tanto *applets* como aplicaciones independientes. En cualquiera de los casos, el entorno incorpora un potentísimo depurador integrado.

Data Form Wizard

La primera modalidad de creación de aplicaciones de acceso a bases de datos con *Visual J++ 6* es *Data Form Wizard,* que crea *applets* o aplicaciones que acceden a bases de datos *DAO* u *ODBC*. Esta herramienta nos permite seleccionar la fuente de datos, la tabla y los campos que deseamos visualizar y los botones de navegación que deberán añadirse al formulario. El asistente nos construye un formulario que permite la visualización, edición y eliminación de registros, con sólo unos cuantos clics. Así mismo, *Data Form Wizard* nos permite escoger la plataforma de salida de nuestra aplicación, como hemos comentado anteriormente.

Figura F, Figura G, Figura H, Figura I Data Form Wizard

Vinculación a datos de controles *WFC*

WFC (Windows Foundation Classes) es una nueva biblioteca de clases, análogas a *MFC,* escritas en *Java* y desarrolladas por *Microsoft* para la creación de aplicaciones para *Win32*. Se trata pues de facilitar la creación de aplicaciones *Win 32,* pero sólo eso, no permite la creación de aplicaciones multiplataforma. Las *WFC* van a darnos el acceso a la creación de ricas interfaces de usuario con controles tan sofisticados como los que podemos crear en *Visual C++* o *Visual Basic,* que hacen uso de *J/Direct,* que como el lector ya sabe permite el acceso al *API* de *Windows*.

La técnica que estamos comentando, el *WFC Data Binding,* nos permite, en tiempo de diseño *RAD* o mediante código, la vinculación de controles *WFC* con fuentes de datos. No existen en *Visual J++ 6* controles de usuario específicamente vinculables a datos, sino que cualquier control, una caja de edición de texto, una lista, etc, pueden servir como destino de los mismos. También podemos establecer vínculos más complejos entre controles no visuales y datos a través de *recordsets*.

Esta vinculación se realiza a través del control *DataBinder* que permite vincular campos de tablas con cualquiera de las propiedades de un control. Este control está visible en tiempo de diseño y tiene una función meramente auxiliar, pero extraordinariamente útil.

Otro control, *DataSource* encapsula una conexíon y una consulta realizada utilizando *ADO*, y concretamente a través de los objetos *Connection, Command*, and *Recordset*.

Además, VJ++ también facilita un control que nos permite navegar por los datos:. *DataNavigator*

Para ilustrar la creación de una aplicación mínima que utilice *WFC Data Binding* vamos a crear un proyecto con *Visual J++ 6*. En primer lugar hemos de hacer constar que la versión con la que estamos trabajando es una versión preliminar, que puede obtenerse en la sede *Web* de *Microsoft*. Nuestra intención es simplemente presentar al lector una visión previa de la funcionalidad que va a poder utilizar dentro de unos meses. En cualquier caso, en el CD que acompaña a esta revista, así como en la sede *Web* de *HISPAN TECNOLOGIC* www.hispan.com, se encontrará el código de la pequeña aplicación que vamos a crear.

Vamos a crear un proyecto con un formulario que contendrá unos cuantos controles *WFC* y que accederá a tabla *pubs* de la base de datos *pubs* de *SQL Server*. El lector podrá modificar la aplicación para acceder a cualquier otra base de datos si es que así es su deseo. Para comenzar deberemos crear un nuevo proyecto del tipo *Application* escogiendo como tipo de aplicación *Windows Application,* opción que hará que el entorno automatice la importación de los paquetes relacionados con *wfc*. *Visual J++* creará un formulario, vacío, al que deberemos añadir dos cajas de edición de texto, en las que presentaremos dos campos de la base de datos, y un par de etiquetas, para identificarlos. Asimismo, y a título meramente de utilización de más controles, hemos colocado un par de controles *Picture Box,* y un conjunto de etiquetas agrupadas que describen el proceso a seguir.

El siguiente proceso es añadir el control *DataSource,* que facilitará la conexión con la base de datos. Bastará con modificar algunas de sus propiedades, en el *Explorador de propiedades* del entorno, para posibilitar esta conexión. La primera de ellas es *ConnectionString,* que contiene el nombre de la fuente de datos *ODBC* o *OLEDB* que va a permitirnos establecer la conexión. La modificación de esta propiedad puede hacerse mediante un cuadro de diálogo que nos permite identificar la conexión de entre las existentes o crear una nueva.

Figura J Establecimiento de la cadena de conexión

Una vez establecida la cadena de conexión, que será en nuestro ejemplo la base de datos *pubs,* deberemos escribir la consulta SQL que dará origen a los datos en la propiedad *CommandText,* en nuestro caso *Select * from authors*. También es necesario modificar la propiedad *locktype* a *optimistic (*el tipo de bloqueo de los registros) y el tipo de cursor *(cursortype)* a *KeySet,* por ejemplo, para permitir la actualización de registros.

El siguiente paso es depositar un control *DataBinder,* que sólo aparecerá en tiempo de diseño, al igual que *dataSource,* y que va a permitirnos vincular los controles a los campos del recorset obtenido con la cadena *SQL* de la fuente de datos. Las propiedades de este control de diseño van a permitirnos escoger un *DataSource* de entre los que tengamos en el formulario y vincular sus campos a cualquiera de las propiedades de sus controles. Esto es una novedad, porque aunque vamos a vincular los campos del nombre y apellido de los autores de la tabla a las propiedades *Text* de las dos cajas de edición de texto, podríamos haberlo hecho con cualquier propiedad no visual, tales como el color, el ancho, o la posibilidad de lectura y escritura a través del control.

Figura K, Vinculación de propiedades

Finalmente incluiremos un control *DataNavigator* para facilitar la navegación por los registros del resultado. Bastará compilar y ejecutar para comprobar que hemos creado una aplicación que accede a bases de datos sin escribir ni una sola línea de código.

Figura L Aspecto de la aplicación ejemplo

En la próxima y última entrega de este curso describiremos con más detalle *Visual J++ 6.0* y las grandes posibilidades que nos brinda.