

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Especialidad de Ingeniería Informática



Lenguajes de Programación 2

Temas de Estudio

TERCERA EDICIÓN

OTTO FERNANDO PFLÜCKER LÓPEZ

CLAUDIA MARÍA DEL PILAR ZAPATA DEL RÍO

JOHAN BALDEÓN MEDRANO

ESPECIALIDAD DE INGENIERÍA INFORMÁTICA DE LA PUCP

Lenguajes de Programación 2: Temas de Estudio, Segunda Edición

Otto Fernando Pflücker López

Profesor de la especialidad de Ingeniería Informática
Pontificia Universidad Católica del Perú

Claudia María Del Pilar Zapata Del Río

Profesora de la especialidad de Ingeniería Informática
Pontificia Universidad Católica del Perú

Johan Baldón Medrano

Profesora de la especialidad de Ingeniería Informática
Pontificia Universidad Católica del Perú

Con la colaboración de los alumnos de la especialidad:

Carreño Berastain, Juan Carlos
Laguna Gutierrez, Victor Antonio
Pinto Ortiz, Carol Denisse
Vallejos Contreras, Carlos Alberto

Con la colaboración del Profesor de la especialidad:

José Antonio Pow Sang

© Pontificia Universidad Católica del Perú

Av. Universitaria Cdra. 18, San Miguel, Lima 32 • Perú, 2006

Teléfono (511) 626 2000

Nota sobre la tercera edición

Tanto los contenidos teóricos como los ejercicios de la primera edición fueron revisados para esta segunda. Agradecemos el gran trabajo realizado por los alumnos Juan Carreño, Víctor Laguna, Carol Pinto y Carlos Vallejos en la revisión completa de la redacción de este compendio, y de sus muchas sugerencias que contribuyeron a mejorar la comprensión de los temas aquí tratados.

Tabla de Contenido

Introducción a Java y C#	2
<i>Introducción a Java</i>	2
Breve Historia	2
La Concepción de Java	2
Entorno de Trabajo de Java	3
Ciclo de Desarrollo	4
Tipos de Programas en Java	4
Compiladores, Intérpretes Puros e Intérpretes JIT	4
Paralelo entre Java y C++	5
Introducción a la Programación	7
Aplicaciones Java: Ejemplo 1	7
Compilación y Ejecución de una Aplicación Java	7
Aplicaciones Java: Ejemplo 2	8
Conclusiones Preliminares	9
Elementos de la Programación	10
Identificadores	10
Operadores	10
Estructuras de Control	11
Constantes Literales	11
El API de Java	12
<i>Introducción a C#</i>	13
Breve Historia	13
La Concepción de C#	13
Entorno de Trabajo de C#	13
Ciclo de Desarrollo	13
Primer Programa	16
Comparación con C++	17
El API de .NET	18
Tipos de Datos	24
<i>Conceptos Previos</i>	24
<i>Conceptos sobre Tipos de Datos</i>	24
Objetos de Datos	25
Variables y Constantes	26
Tipo de Datos	26
Tipos de Datos Primitivos	27
Tipos de Datos Primitivos en Java	28
Tipos de Datos en C#	29
Las Operaciones sobre los Tipos de Datos	35
<i>Ejercicio</i>	35
Arreglos y Cadenas	29
<i>Arreglos en Java</i>	29
Arreglos Unidimensionales	29
Arreglos Multidimensionales	30
Acceso a los Elementos de un Arreglo	31
Paso de un Arreglo a un Método	31
Preguntas de Repaso	32

<i>Arreglos en C#</i>	32
<i>Cadenas de Carácteres en Java</i>	34
<i>Cadenas de Carácteres en C#</i>	37
Programación Orientada a Objetos	2
<i>Conceptos Básicos</i>	2
Sobre las Clases	2
Sobre los Objetos	3
Sobre los Miembros	3
<i>Las Clases</i>	3
La Declaración de las Clases	4
La Creación de los Objetos	5
Los Constructores y Destructores	6
Los Constructores	6
Los Constructores Estáticos	8
Los Destructores	8
La Recolección de Basura	9
Manejo Manual y Automático de la Memoria	9
Finalización Determinista y No-Determinista	10
El Acceso a los Miembros	11
El Uso de las Variables	12
La Declaración de las Variables	12
La Duración y el Ámbito	13
La Inicialización Automática	14
Los Modificadores	15
El Uso de los Métodos	17
La Declaración de los Métodos	17
El Paso de Parámetros	19
Las Propiedades y los Indizadores	20
<i>Las Estructuras</i>	23
<i>La Herencia y el Polimorfismo</i>	24
La Herencia	24
La Declaración de la Herencia	25
El Proceso de Construcción y Destrucción	25
Acceso a los Miembros Heredados	26
El Polimorfismo	27
La Sobrecarga	27
El Ocultamiento	29
La Sobrescritura Virtual	30
La Implementación de Interfaces	31
Las Clases Abstractas	33
Declaración de una Clase Abstracta	33
Diferencias entre las Interfaces y las Clases Abstractas	33
<i>Los Tipos de Datos Anidados</i>	34
La Declaración de Clases Anidadas	34
La Creación de Objetos de Clases Anidadas	35
Los Conflictos de Nombres	36
Las Clases Anidadas Anónimas	37
<i>La Reflexión</i>	38
Definición y Uso	38
RTTI	39
Espacios de Nombres y Librerías	79

<i>Los Espacios de Nombres</i>	79
Definición de un Espacio de Nombres	79
Anidamiento de Espacios de Nombres	80
Publicación de un Espacio de Nombres	81
Uso de un Alias	82
Definición por Partes.....	82
<i>Las Librerías</i>	83
Librerías en C/C++	83
Librerías Estáticas	84
Librerías Dinámicas	84
Estructura Interna	85
Espacio de Direccionamiento	86
Creación de una DLL	87
Utilización de una DLL	90
Mecanismo de Búsqueda de una DLL	91
Librerías en Java.....	92
Uso de un Paquete	92
Ubicación de un Paquete	93
Creación de un Paquete	93
Utilización de un Nuevo Paquete	94
Librerías en C#.....	95
Los Ensamblajes.....	95
El Ensamblaje Tipo Librería	97
Relación con los Espacios de Nombres.....	98
Programación genérica	2
<i>Introducción</i>	2
<i>Terminología</i>	3
<i>Funciones genéricas y métodos genéricos</i>	3
<i>Tipos de datos genéricos</i>	8
<i>Genéricos en las librerías para el manejo de colecciones</i>	12
La STL de C++.....	12
La librería para el manejo de colecciones de Java.....	17
La librería para el manejo de colecciones de .NET	20
<i>Alcances, limitaciones y diferencias en el soporte a la programación genérica en C++, Java y C#</i>	23
<i>Ventajas y desventajas de la programación genérica</i>	25
Archivos, Flujos y Persistencia de Objetos	210
<i>Archivos y Flujos</i>	210
<i>Objetos Persistentes</i>	211
<i>Manejo desde C#</i>	212
Descripción General de las Capacidades	212
Acceso al Sistema de Archivos	213
Manejo de Consola	215
Manejo de Archivos de Texto	217
Manejo de Archivos Binarios Secuencialmente	218
Manejo de Archivos Binarios Aleatoriamente	219
<i>Manejo desde Java</i>	222
Flujos.....	222
Manejo de Consola	226
Manejo de Archivos	229
Archivos de Texto.....	229

Archivos Binarios Secuenciales.....	230
Archivos Binarios Aleatorios	232
Programación con GUI.....	99
<i>Interfaces GUI con Ventanas.....</i>	<i>99</i>
<i>Tipo de Interfaces GUI con Ventanas</i>	<i>99</i>
Programas Stand-Alone	100
Programas Basados en Web.....	100
<i>Creación y Manejo de GUI con Ventanas</i>	<i>100</i>
Creación de una Ventana	100
Elementos de una Ventana.....	106
<i>Interacción con el Entorno</i>	<i>107</i>
Manejo de Eventos con API de Windows.....	107
Manejo de Eventos con Interfaces en Java	110
Manejo de Eventos con Delegados en C#.....	115
Tipos de Eventos.....	118
<i>Gráficos en 2D.....</i>	<i>118</i>
Dibujo con API de Windows	120
Funciones de Dibujo	120
Dibujo Asíncrono	121
Dibujo Síncrono.....	123
Dibujo Sincronizado	124
Dibujo en Memoria.....	124
Dibujo en Java	125
Dibujo Asíncrono	126
Dibujo Síncrono.....	126
Dibujo Sincronizado	127
Dibujo en Memoria.....	127
Dibujo en C#.....	128
Dibujo Asíncrono	128
Dibujo Síncrono.....	128
Dibujo Sincronizado	129
Dibujo en Memoria.....	129
<i>Manejo de Elementos GUI.....</i>	<i>130</i>
Elementos GUI del API de Windows	130
Elementos GUI de Java.....	132
Elementos GUI de C#.....	134
Manejo Asistido del Diseño.....	135
<i>Tipos de Ventana.....</i>	<i>136</i>
Ventanas con API de Windows	137
Ventanas en Java.....	138
Ventanas en C#.....	140
<i>Notas sobre Localización de Archivos</i>	<i>143</i>
Excepciones.....	144
<i>¿Qué son las excepciones?</i>	<i>144</i>
<i>Implementación.....</i>	<i>146</i>
C++	146
Java	153
C#.....	158
Ventajas, desventajas y criterios de uso.....	159
Programación con GUI.....	99

<i>Interfaces GUI con Ventanas</i>	99
<i>Tipo de Interfaces GUI con Ventanas</i>	99
Programas Stand-Alone.....	100
Programas Basados en Web	100
<i>Creación y Manejo de GUI con Ventanas</i>	100
Creación de una Ventana.....	100
Elementos de una Ventana	106
<i>Interacción con el Entorno</i>	107
Manejo de Eventos con API de Windows.....	107
Manejo de Eventos con Interfaces en Java	110
Manejo de Eventos con Delegados en C#	115
Tipos de Eventos	118
<i>Gráficos en 2D</i>	118
Dibujo con API de Windows.....	119
Funciones de Dibujo	120
Dibujo Asíncrono	121
Dibujo Síncrono	123
Dibujo Sincronizado.....	123
Dibujo en Memoria	124
Dibujo en Java.....	125
Dibujo Asíncrono	125
Dibujo Síncrono	126
Dibujo Sincronizado.....	126
Dibujo en Memoria	127
Dibujo en C#	127
Dibujo Asíncrono	128
Dibujo Síncrono	128
Dibujo Sincronizado.....	129
Dibujo en Memoria	129
<i>Manejo de Elementos GUI</i>	130
Elementos GUI del API de Windows.....	130
Elementos GUI de Java	131
Elementos GUI de C#.....	134
Manejo Asistido del Diseño	135
<i>Tipos de Ventana</i>	136
Ventanas con API de Windows.....	137
Ventanas en Java	138
Ventanas en C#.....	140
<i>Notas sobre Localización de Archivos</i>	143
Programación Concurrente	161
<i>Procesos e Hilos</i>	161
Espacio de Memoria Virtual.....	162
Ciclo de vida de un Hilo	163
Planificación.....	163
Soporte para la Programación Concurrente.....	164
<i>Manejo de Hilos</i>	165
Creación	165
<i>Clases para el Manejo de Hilos</i>	167
C#	167
Java.....	168
La Clase Thread	168

<i>Aspectos Generales de la Ejecución de un Hilo</i>	<i>170</i>
<i>Trabajo con Hilos en C++</i>	<i>170</i>
Sincronización	172
<i>Trabajo con Hilos en C#.....</i>	<i>175</i>
Sincronización	178
<i>Trabajo con Hilos en Java.....</i>	<i>182</i>
Sincronización	186

Introducción a Java y C#

El presente material presume que el lector conoce los principios de programación estructurada y modular utilizando lenguaje C++. Este capítulo tiene como objetivo dar las bases para que el alumno sea capaz de aplicar los mismos principios de programación pero en los lenguajes Java y C#, dado que estos serán utilizados a lo largo de los siguientes capítulos.

Introducción a Java

Breve Historia

1991: Sun Microsystems, dentro de un proyecto interno (Green) crea un lenguaje Oak (autor: James Gosling), destinado a lo que ellos pensaban sería el próximo gran desarrollo: dispositivos electrónicos de consumo inteligentes. Lenguaje basado en C++.

1993: La WWW explota en popularidad. SUN encuentra nuevas posibilidades para su lenguaje, en el desarrollo de páginas WEB con contenido dinámico. Se revitaliza el proyecto.

1995: Se anuncia JAVA. Generó interés inmediato debido al ya existente interés sobre la WWW.

La Concepción de Java

Mientras que otros lenguajes fueron concebidos para facilitar la investigación científica, la enseñanza académica o para aplicaciones específicas, JAVA tuvo una motivación comercial. Esto significaba que debía cumplir con objetivos tales como:

- **Ser fácil de aprender y de utilizar.** Un lenguaje que si bien está orientado a objetos, debe evitar todas las sofisticaciones que un lenguaje como C++ implementa y lo hacen difícil de aprender y de utilizar.
- **Tener un ciclo de desarrollo rápido.** El lenguaje deberá proporcionar herramientas para todos los trabajos más comunes durante la programación, de forma que el programador sólo se concentre en agregar nueva funcionalidad. Los ciclos de prueba y depuración también deben ser rápidos.
- **Ser confiable.** El lenguaje deberá llevar al desarrollador a programar de forma tal que el programa final sea estable (sin caídas). Además, el lenguaje deberá prevenir que cualquier acción del programador afecte al sistema operativo inestabilizándolo.
- **Ser seguro.** Al estar orientado al desarrollo para Internet, el lenguaje deberá brindar características de seguridad tanto en la protección del programa distribuido (contra

procesos de ingeniería inversa) como en la facilidad de poder establecer permisos de acceso.

- **Ser portable.** Java debía de ser independiente de la plataforma de ejecución (concepto de Arquitectura Neutra). Un programa Java debería generarse una única vez y poder ejecutarse en cualquier plataforma (hardware + software) que cuente con un Intérprete Java.

El producto final de esta concepción, JAVA, es un lenguaje de programación orientado a objetos. Las piezas de programación se denominan clases.

Java esta formado por librerías de clases. El conjunto de librerías estándar es extenso y cubren las actividades más comunes durante el desarrollo de programas de modo que el programador pueda concentrarse sólo en desarrollar la nueva funcionalidad.

Las librerías de clases de Java pueden agruparse en 3 grupos:

1. **Estándar.** Desarrolladas y distribuidas libremente por SUN Microsystems conjuntamente con las herramientas básicas de desarrollo del lenguaje.
2. **Desarrolladas por terceros.** El programador necesitará comprarlas para poder utilizarlas.
3. **Desarrolladas por el mismo programador.** Java permite crear nuevas librerías de forma que pueda ser utilizada por muchos programas desarrollados por el programador.

El punto de referencia respecto a la sintaxis del lenguaje fue C++. En este sentido, ambos lenguajes comparten muchos conceptos como, por ejemplo, la programación multitarea.

Entorno de Trabajo de Java

Aunque el desarrollo de un programa en Java puede realizarse utilizando únicamente el Kit de desarrollo proporcionado por SUN llamado JDK (Java Developer Kit), existen muchos entornos de desarrollo integrados (IDE) que facilitan este proceso. Algunos de los IDE's utilizados en nuestro medio son:

- JBuilder IDE de Borland
- Borland C++ 5.0 IDE para C++ de Borland con extensión para Java
- Visual J++ IDE de Microsoft.
- Visual Age for Java IDE de IBM.

Existen JDK's apropiados para trabajar en los principales sistemas operativos. El JDK consiste principalmente en:

- Las librerías que implementan el lenguaje
- El resto de librerías estándar
- Un compilador
- Uno o más intérpretes
- Programas utilitarios adicionales

Ciclo de Desarrollo

El desarrollo de un programa en JAVA pasa por 5 fases. Las 2 primeras corresponden a la creación del programa

4. **Edición.** Las fuentes de Java son archivos de texto con extensión JAVA. Un programa en Java puede escribirse utilizando cualquier editor de texto. Si el archivo contiene una clase pública, su nombre debe corresponder exactamente al nombre de esta clase.
5. **Compilación.** Se utiliza un compilador para JAVA (javac.exe para Borland, jvc.exe para Microsoft). Este programa recibe como entrada el archivo fuente (*.java) y genera un archivo compilado con el mismo nombre que el anterior, pero con extensión CLASS (ejemplo: Hola.class). Este archivo compilado contiene juegos de instrucciones llamadas BYTECODE, no código máquina, por lo que no puede ser ejecutado directamente por el sistema operativo.

Las 3 últimas corresponden a la ejecución del programa generado

6. **Carga de la Clase.** Lo realiza un programa Cargador de Clases que se encarga de colocar en memoria el bytecode del archivo a procesar.
7. **Verificación del BYTECODE.** Lo realiza un programa Verificador de BYTECODE que se encarga de revisar que todo el BYTECODE leído y cargado a memoria sea válido y que no viole las restricciones de seguridad de JAVA.
8. **Interpretación.** Lo realiza un programa Intérprete, el cual lee el BYTECODE y lo convierte a instrucciones en lenguaje máquina, de forma que pueda ser ejecutado.

Tipos de Programas en Java

Existen 2 tipos de programas que se pueden generar con JAVA:

- **Aplicaciones:** Programas que normalmente se almacenan y ejecutan en la propia máquina del usuario. Un programa de JAVA se carga utilizando un intérprete de JAVA (como se explicó líneas arriba).
- **Applets:** Programa hechos para correr dentro de un browser de WEB (como el Netscape, Explorer, etc.) el cual, internamente, es el responsable de llamar al intérprete. Un applet comúnmente es cargado desde una computadora remota.

Un applet también puede ser visto desde un visor de applets (llamado **appletviewer.exe** en el JDK, **jview.exe** en Visual J++). Este programa es el browser mínimo (pues sólo reconoce el tag applet) necesario para probar un applet y recibe como entrada un HTML.

Nota: JavaScript es un lenguaje independiente y diferente al JAVA, además de tener propósitos diferentes.

Compiladores, Intérpretes Puros e Intérpretes JIT

Cuando un archivo fuente Java es compilado, lo que se genera es BYTECODE, que no es otra cosa que una serie de instrucciones “genéricas” (su formato y significado no está amarrado a ninguna arquitectura), que se corresponden con las instrucciones dadas en el archivo fuente. Sin embargo, a diferencia del archivo fuente que tiene un formato de texto, el BYTECODE es una secuencia binaria, haciendo que su lectura y velocidad de ejecución sea mucho más eficiente.

Java es un lenguaje interpretado. Esto quiere decir que las instrucciones de un programa en Java son traducidas por otro programa que hace de intermediario (Intérprete) entre el lenguaje Java y el entorno de ejecución (hardware + sistema operativo). Los intérpretes leen una instrucción Java y ejecutan su equivalente en instrucciones que pueden ser ejecutadas directamente por el computador (código nativo).

Al comienzo, del lado del cliente sólo se tenía disponibles programas intérpretes. Sin embargo, el costo adicional que el proceso de interpretación significa hace que un programa interpretado sea comparativamente mucho más lento que un programa compilado. Debido a esto se desarrollaron Compiladores Reales, de BYTECODE a código nativo, orientados a programas en donde la máquina del cliente es conocida. El programa generado, al igual que cualquier programa compilado a código nativo, es dependiente de la plataforma de ejecución del cliente.

Dado que la interpretación de un código cuyas instrucciones no están amarradas a ninguna arquitectura (arquitectura neutra) es esencial para cumplir con el objetivo de portabilidad que se buscaba con el lenguaje, y que Java fue pensado para funcionar como lenguaje para desarrollo de aplicaciones en Internet, un ejecutable de Java tendría que ser transferido al computador en donde se desea ver una página HTML que contenga un applet de Java. Dado que un ejecutable Java es mucho más extenso que un compilado a BYTECODE, se crea un conflicto entre la velocidad de ejecución y la velocidad de carga inicial de una página HTML que contenga un programa applet de Java.

Una solución parcial a este problema lo resuelven los llamados Intérpretes JIT (Just-In-Time). Un JIT funciona de la misma manera que un intérprete normal la primera vez que ejecuta cada parte de un BYTECODE, pero genera a la par un cuasi-código-nativo de dichas instrucciones de forma que la siguiente vez que se intente ejecutar las mismas instrucciones se utilizará el nuevo código generado. Esto hace un poco más lenta la ejecución inicial pero hace rápida la carga de la página HTML que contiene el applet (dado que lo que se transfiere es BYTECODE) y acelera las sucesivas ejecuciones del programa.

Paralelo entre Java y C++

Algunas diferencias notorias entre la programación bajo C++ para Windows y JAVA se muestran en la Tabla 1 - 1.

Tabla 1 - 1 Diferencias entre Java y C++

C++	JAVA
Los programas utilizan programación estructurada y/o programación orientada a objetos.	Toda la programación es orientada a objetos.
La implementación de clases sigue el esquema del bosque de árboles.	La implementación de clases sigue el esquema del árbol único.
La implementación de los métodos de una clase puede estar dentro de ella o fuera de su declaración.	Toda la implementación de una clase debe estar dentro de la clase.
Herencia de clases múltiple.	Herencia de clases simple + implementación de interfaces.

Los fuentes están formados por archivos cabecera y un archivo de implementación.	Toda la implementación se hace en un sólo archivo.
Se compila generando CODIGO NATIVO.	Se compila generando BYTECODE.
La compilación genera archivos EXE, LIB, DLL, OCX, etc.	La compilación genera archivos CLASS.
EXISTEN PUNTEROS.	NO EXISTEN PUNTEROS.
Existen clases y estructuras, que permiten almacenar datos de diversa índole.	Sólo existen clases.
En Windows (y en otros sistemas operativos donde se trabaje bajo el esquema de ventanas) la programación se maneja mediante mensajes. Se trabaja en base a funciones de procesamiento de mensajes.	Programación manejada por eventos. Se trabaja mediante métodos que se disparan en respuesta a una acción del usuario. El trabajo interno con los mensajes es ocultado por el lenguaje.
Se pueden sobrecargar los operadores para una clase.	No existe sobrecarga de operadores.
La liberación de la memoria solicitada durante el programa es responsabilidad del mismo programa.	La liberación de la memoria solicitada durante el programa es responsabilidad del propio lenguaje JAVA, mediante el Garbage Collector.

Algunas semejanzas notorias entre la programación bajo C++ para Windows y JAVA se muestran en la Tabla 1 - 2.

Tabla 1 - 2 Semejanzas entre Java y C++

C++ y JAVA
Ambas utilizan la misma sintaxis (salvo mínimas diferencias) al escribir expresiones, sentencias, funciones, declarar variables, declarar clases, etc.
Ambos utilizan los mismos operadores aritméticos y lógicos, con las mismas reglas de precedencia y agrupación.
La implementación de una clase (la herencia, el polimorfismo, etc.) son muy similares.
El juego de palabras reservadas y su uso es muy similar, por ejemplo, los objetos pueden hacer referencia a sí mismos mediante la palabra clave this.
Tienen la misma implementación de las estructuras de control (if, if/else, while, do/while, switch, for).
Los modificadores de ámbito de acceso son muy similares en su uso y reglas (public, private, protected, etc.).

Introducción a la Programación

Las siguientes secciones dan una introducción a la programación en Java y C# mediante ejemplos de programas básicos.

Aplicaciones Java: Ejemplo 1

Para mostrar los detalles más básicos sobre la programación en Java analizaremos una aplicación Java simple que muestre un mensaje.

Los fuentes de un programa en Java son archivos de texto con extensión JAVA, por lo que el siguiente código puede ingresarse y guardarse con cualquier editor de texto disponible.

```
// Archivo: Aplicacion1.java
/* Descripción:
   El programa muestra un mensaje en pantalla
*/
public class Aplicacion1 {
    public static void main( String args[ ] ) {
        System.out.println( "Primer programa en Java" );
    }
}
```

Los comentarios en un archivo fuente de Java siguen la misma sintaxis y reglas que C/C++.

Al igual que C/C++, Java especifica un juego de palabras reservadas. Las palabras class, public, void, etc. son reservadas por Java y no deben de utilizarse para definir identificadores. Todas las palabras reservadas de Java están en minúscula.

En el ejemplo, las palabras Aplicacion1, main, System, out, println, String, etc. son identificadores.

La sintaxis para especificar literales (numéricos, caracteres y cadenas de caracteres) es la misma que C/C++. En el ejemplo anterior se especifica un literal tipo cadena.

Las sentencias en Java siguen las mismas reglas que C/C++. Una sentencia siempre debe finalizar con un ';'; sin embargo, note que Java no requiere que se coloque un ';' al final de la implementación de la clase.

La implementación de una clase en Java es, en términos generales, muy similar a C++.

A continuación analizaremos los aspectos de este programa referentes a la programación en Java.

Compilación y Ejecución de una Aplicación Java

Para compilar un fuente JAVA se puede utilizar el compilador correspondiente al IDE o entorno de trabajo con que se cuente, por ejemplo:

- JVC.EXE Correspondiente al IDE de Microsoft
- JAVAC.EXE Correspondiente al IDE de Borland y al SDK de Sun.

En el caso de contar con un IDE, normalmente es más sencillo dejar que éste se encargue de la tarea de compilación en vez de realizarla nosotros mismos desde la línea de comandos; sin embargo, por motivos didácticos, realizaremos esto último.

En nuestro ejemplo, para compilar el archivo Aplicacion1.java se puede utilizar la siguiente instrucción:

```
C:\>javac.exe Aplicacion1.java
```

Esta instrucción utiliza el compilador de Borland (parte de su IDE para Java, JBuilder) para compilar el archivo editado. Si la compilación fue exitosa, se genera el correspondiente archivo: Aplicacion1.class.

Para ejecutar un archivo CLASS obtenido de la compilación de un archivo fuente JAVA necesitaremos correr el intérprete de Java y pasarle como parámetro nuestro archivo compilado. Nuevamente, se puede utilizar el intérprete correspondiente al IDE o entorno de trabajo con que se cuente, por ejemplo:

- JVIEW.EXE y WJVIEW.EXE Correspondiente al IDE de Microsoft
- JAVA.EXE y APPLVIEWER.EXE Correspondiente al IDE de Borland y al SDK de Sun.

En nuestro ejemplo, para ejecutar el archivo Aplicacion1.class se puede utilizar la siguiente instrucción:

```
C:\>java.exe Aplicacion1
```

Al ejecutar esta instrucción, se carga y ejecuta la aplicación Java contenida en el archivo Aplicacion1.class, mostrándose el mensaje "Aplicacion1 a Java" en pantalla.

Note que en los ejemplos anteriores se asume que tanto el archivo fuente, el compilador, el archivo compilado y el intérprete se encuentran en la misma carpeta, la raíz de la unidad C, o bien en carpetas donde el sistema operativo sea capaz de encontrarlos (para Windows, los directorios del sistema y las carpetas configuradas en la variable de entorno PATH). Si éste no fuese el caso, se tendría que especificar la ruta completa de estos archivos en la instrucción. Para más detalles acerca de las formas en que se puede compilar y ejecutar un programa Java y los posibles errores que pueden ocurrir, se puede consultar el tutorial de Java en Internet:

<http://java.sun.com/docs/books/tutorial/getStarted/cupojava/win32.html>

Note que el ejemplo anterior corresponde a una aplicación de consola. El archivo implementado contiene un método "main" que es el punto de entrada que utilizará el intérprete cuando lo que se ejecuta es una aplicación Java. El intérprete llamará a éste método y éste será el encargado de ejecutar todas las acciones de la aplicación. A las clases para una aplicación Java que contengan el método "main" se les podrá utilizar como punto de inicio de la aplicación. Debido a esto, a estas clases se les llama clases ejecutables. El intérprete no crea un objeto Aplicacion1, su única responsabilidad es llamar al método estático "main" de esta clase (lo que, al igual que C/C++, no requiere que se cree el objeto para ser llamado). El método "main" será el encargado de crear los objetos necesarios para realizar las tareas de la aplicación.

Aplicaciones Java: Ejemplo 2

El siguiente ejemplo agrega utiliza algunas características adicionales comunes a los programas en Java.

```
// Archivo: Aplicacion2.java
// Descripción:
//   Se utiliza una caja de dialogo para mostrar un mensaje.

import javax.swing.JOptionPane;

class Aplicacion2 {
    public static void main( String args[ ] ) {
        String sMensaje;

        sMensaje = "Aplicacion1 a java!!!\n";
        sMensaje += "Los argumentos pasados en la línea de comando son:\n";
        for( int i = 0; i < args.length; i++ )
```

```
        sMensaje += args[i] + "\n";
        sMensaje += '\n';

        double dResultado = RaizCuadrada( 10.5 );
        sMensaje += "La raíz cuadrada de 10.5 es " + dResultado;

        int iDato = 40;
        long lResultado = (long)Multiplicar( 24, iDato );
        sMensaje += "\nLa multiplicación de 24 * " + iDato + " es " + lResultado;
        JOptionPane.showMessageDialog( null, sMensaje );

        System.exit( 0 );
    }

    public static double RaizCuadrada( double dValor ) {
        double dRaiz;
        dRaiz = Math.sqrt( dValor );
        return dRaiz;
    }

    public static long Multiplicar( int iValor1, int iValor2 ) {
        return iValor1 * iValor2;
    }
}
```

La aplicación anterior utiliza una caja de diálogo para mostrar un mensaje, en lugar de hacerlo por la salida estándar, la consola.

Note que los operadores, las estructuras de control, la declaración de variables y su inicialización, la declaración de los métodos y sus parámetros, la llamada a métodos y el paso de parámetros, el retorno de valores de un método y las operaciones de tipo cast para conversión entre tipos primitivos son iguales a C++.

El ejemplo utiliza una variable de tipo arreglo. Los detalles acerca del uso de arreglos se verán más adelante. También utiliza la clase `JOptionPane` que pertenece a la librería “`javax.swing`”, por lo que una sentencia “`import`” al inicio del archivo debe de ser agregada indicándole al compilador dónde se define la clase que estoy utilizando. Las librerías en Java se denominan “paquetes”.

La clase `JOptionPane` permite mostrar ventanas con mensajes y botones de selección. Siempre que se utilicen elementos gráficos (como las ventanas) se deberá finalizar una aplicación llamando al método `System.exit`, el cual recibe como parámetro un número entero que representa el valor de retorno de la aplicación. Estos valores de retorno son comúnmente utilizados en archivos de procesamiento por lotes, como los archivos BAT de Windows, de la misma forma que el valor de retorno de la función “`main`” de un programa en C/C++.

Conclusiones Preliminares

Por lo revisado en los 2 simples códigos de ejemplo anteriores, todo programa en Java consiste básicamente en la definición de una o más clases. Nada está fuera de las clases.

Aunque no se vio en los ejemplos, cada archivo Java puede contener la implementación de más de una clase, sin embargo sólo una de ellas puede ser pública (y su nombre debe corresponder exactamente al del archivo en donde está definida), esto es, declarada como `public`. El uso de estos modificadores de acceso se verá más adelante.

Elementos de la Programación

Identificadores

Todo identificador es un conjunto de caracteres, sin espacios, que no pueden comenzar con un dígito y que pueden contener:

- Letras (a-z, A-Z).
- Dígitos (0-9).
- Los caracteres '\$' y '_'.

El carácter '\$' debe evitarse dado que es usado por el compilador para representar internamente los identificadores.

Al igual que C/C++, Java es sensitivo a las mayúsculas y minúsculas en el nombre de los identificadores.

Operadores

Tabla 1 - 3 Operadores de Java

Operadores Aritméticos: Son operadores binarios (actúan sobre 2 operandos). El tipo del valor resultado depende del tipo de dato de los operandos, siguiendo las reglas de promoción de tipos.							
+	Suma	-	Resta	*	Multiplicación	/	División
%	Módulo (sólo con operandos enteros)						
Operadores Relacionales: Son operadores binarios. El resultado es un valor booleano.							
==	Igualdad	!=	Diferencia	>	Mayor que	<	Menor que
>=	Mayor o igual que			<=	Menor o igual que		
Operadores Lógicos: Trabajan sobre operandos booleanos o expresiones que al ser evaluadas devuelvan un booleano. El resultado es un booleano. Siguen las reglas de evaluación de circuito corto.							
&&	Y		O inclusivo	!	NO		
Operadores Lógicos Booleanos: Trabajan a nivel de bits sobre operandos de tipo primitivo e integral. Si los operandos son de diferente tipo (por ejemplo int y byte) el resultado es de un tipo promovido (byte se promociona a int y el resultado es de tipo int). Cuando los operandos son booleanos, estos operadores se comportan como los Operadores Lógicos. Siguen las reglas de evaluación de circuito largo.							
&	Y		O inclusivo	^	O exclusivo		

Operadores de Incremento y Decremento: Son operadores unarios (actúan sobre un sólo operando). El valor resultado es del mismo tipo del operando. Puede anteceder o preceder al operando. Cuando anteceden, se evalúan antes que la expresión de donde forma parte. Cuando están precediendo, se evalúan después que la expresión de donde forman parte.

++	Incrementa en uno el operando	--	Decrementa en uno el operando				
----	-------------------------------	----	-------------------------------	--	--	--	--

Al igual que en C++, en una expresión con varios operadores, se aplica la misma regla de precedencia. Cuando se utilizan varios operandos con iguales o distintos operadores, se aplica la misma regla de agrupación de C++.

Estructuras de Control

Existen las mismas estructuras de control que en C++, con la misma sintaxis y las reglas de evaluación. Éstas son:

if / else

switch

for

while y do-while

Dentro de los bucles (for, while, do / while) se pueden utilizar los controladores de flujo:

continue

continue <nombre de etiqueta>

break

break <nombre de etiqueta>

Nótese que a diferencia de C++, existen versiones con etiqueta para cada controlador. El detalle de estas características de Java escapa del alcance del curso.

Constantes Literales

Se siguen las mismas reglas que C++. Los sufijos para los literales numéricos, al igual que C++, modifican el tipo por defecto de éstos. La Tabla 1 - 4 muestra ejemplo de declaración de constantes para diferentes tipos de datos.

Los literales de tipo cadena aceptan los caracteres de escape de C: “\n”, “\t”, “\r”, “\\”, “\””.

Tabla 1 - 4 Ejemplos de constantes literales en Java

Literal	Tipo de Dato
178	int
8864L	long
37.266	double
37.266D	double
87.363F	float
26.77e3	double
'c'	char

trac	boolean
------	---------

El API de Java

Java cuenta con una extensa gama de clases predefinidas. A este conjunto de clases, agrupados en paquetes, se le conoce como el API de Java.

Algunos paquetes más usados del API de JAVA se muestran en la Tabla 1 - 5.

Tabla 1 - 5 Paquetes estandar de Java

Nombre	Descripción
java.awt	Java Abstract Window Toolkit. Contiene los componentes básicos para crear y manejar aplicaciones con interfaz gráfica de usuario (GUI).
java.awt.event	Clases e interfaces que permiten manejar eventos para los componentes de este paquete.
java.io	Permiten la lectura y escritura desde los dispositivos de entrada y salida estándar (teclado y pantalla por defecto).
java.lang	Clases e interfaces que forman parte del lenguaje Java.
java.rmi	Remote Method Invocation. Clases e interfaces que permiten crear programas Java distribuidos. Usando RMI un programa puede llamar a un método de otro programa en la misma PC u otra que sea accesible a través de Internet.
java.sql	Clases e interfaces que permiten la conexión y manejo de bases de datos.
java.util	Clases e interfaces utilitarios de JAVA. Permiten, entre otras cosas: <ul style="list-style-type: none">▪ Manipulación de fecha y hora▪ Generación de números aleatorios▪ Almacenamiento y procesamiento de largas cantidades de datos▪ Tokenizar cadenas▪ Manejo de bits
javax.swing	Clases, interfaces y otros sub-paquetes que permiten utilizar el nuevo juego de componentes SWING. Reemplazan a muchos de los elementos proporcionados por los paquetes (y sub-paquetes bajo éstos) java.applet y java.awt.

Introducción a C#

Breve Historia

.Net es una estrategia que desarrolla Microsoft para el fácil desarrollo y manejo de aplicaciones uso de Web Services. Para el flujo de información a través de sistemas y dispositivos interconectados.

Los servicios Web son módulos de software autodescriptivo que encapsula funcionalidad que es encontrada y accedida a través de protocolos estándares de comunicación como SOAP y XML.

La Concepción de C#

C# es un lenguaje orientado a objetos que se basa en la familia de lenguajes basados en C como C++ y Java.

C# está estandarizado según ECMA e ISO/IEC y sus compiladores también han sido desarrollados de acuerdo a estos estándares.

A pesar de ser orientado a objetos también tiene soporte para componentes permitiendo desarrollar ensamblajes auto descriptivos.

Entre sus características incluye un recolector de basura que permite gestionar de manera automática la memoria. También permite la utilización de excepciones para poder controlar errores y desarrollar código de recuperación. Finalmente permite el manejo de código seguro haciendo imposible la utilización de variables no inicializadas, la señalización de índices fuera de los límites de un arreglo y los casteos no verificados.

Entorno de Trabajo de C#

Aunque el desarrollo de un programa en C#, al igual que en Java, puede realizarse utilizando únicamente el Kit de desarrollo, existen muchos entornos de desarrollo integrados (IDE) que facilitan este proceso. Algunos de los IDE's utilizados en nuestro medio son:

- Visual Studio IDE de Microsoft
- SharpDevelop IDE de uso libre.

El SDK consiste principalmente en:

- Las librerías que implementan el lenguaje
- El resto de librerías estándar
- Un compilador
- Uno o más intérpretes
- Programas utilitarios adicionales

Ciclo de Desarrollo

El proceso de ejecución administrada incluye los pasos siguientes:

1. Elegir un compilador.

Para obtener los beneficios que proporciona Common Language Runtime, se deben utilizar uno o más compiladores de lenguaje orientados al tiempo de ejecución.

Para aprovechar las ventajas que ofrece Common Language Runtime, se deben utilizar uno o varios compiladores de lenguaje orientados al tiempo de ejecución, como Visual Basic, C#, Visual C++, Jscript o uno de los muchos compiladores de otros fabricantes como un compilador Eiffel, Perl o COBOL.

Como se ejecuta en un entorno multilenguaje, el motor en tiempo de ejecución es compatible con una gran variedad de tipos de datos y características de lenguajes. El compilador de lenguaje utilizado determina las características en tiempo de ejecución que están disponibles y el código se diseña con esas características. El compilador, y no el motor en tiempo de ejecución, es el que establece la sintaxis que se debe utilizar en el código. Si todos los componentes escritos en otros lenguajes deben ser totalmente capaces de utilizar un componente, los tipos exportados de ese componente deben exponer sólo las características del lenguaje incluidas en Common Language Specification (CLS).

2. Compilar el código a Lenguaje intermedio de Microsoft (MSIL).

Cuando se compila a código administrado, el compilador convierte el código fuente en Lenguaje intermedio de Microsoft (MSIL), que es un conjunto de instrucciones independiente de la CPU que se pueden convertir de forma eficaz en código nativo. MSIL incluye instrucciones para cargar, almacenar, inicializar y llamar a métodos en los objetos, así como instrucciones para operaciones lógicas y aritméticas, flujo de control, acceso directo a la memoria, control de excepciones y otras operaciones. Antes de poder ejecutar código, se debe convertir MSIL al código específico de la CPU, normalmente mediante un compilador Just-In-Time (JIT). Common Language Runtime proporciona uno o varios compiladores JIT para cada arquitectura de equipo compatible, por lo que se puede compilar y ejecutar el mismo conjunto de MSIL en cualquier arquitectura compatible.

Cuando el compilador produce MSIL, también genera metadatos. Los metadatos describen los tipos que aparecen en el código, incluidas las definiciones de los tipos, las firmas de los miembros de tipos, los miembros a los que se hace referencia en el código y otros datos que el motor en tiempo de ejecución utiliza en tiempo de ejecución. El lenguaje intermedio de Microsoft (MSIL) y los metadatos se incluyen en un archivo ejecutable portable (PE), que se basa y extiende el PE de Microsoft publicado y el formato Common Object File Format (COFF) utilizado tradicionalmente para contenido ejecutable. Este formato de archivo, que contiene código MSIL o código nativo así como metadatos, permite al sistema operativo reconocer imágenes de Common Language Runtime. La presencia de metadatos junto con el Lenguaje intermedio de Microsoft (MSIL) permite crear códigos autodescriptivos, con lo cual las bibliotecas de tipos y el Lenguaje de definición de interfaces (IDL) son innecesarios. El motor en tiempo de ejecución localiza y extrae los metadatos del archivo cuando son necesarios durante la ejecución.

3. Compilar MSIL a código nativo.

Para poder ejecutar el lenguaje intermedio de Microsoft (MSIL), primero se debe convertir éste, mediante un compilador Just-In-Time (JIT) de .NET Framework, a código nativo, que es el código específico de la CPU que se ejecuta en la misma arquitectura de equipo que el compilador JIT. Common Language Runtime proporciona un compilador JIT para cada arquitectura de CPU compatible, por lo que los programadores pueden escribir un conjunto de MSIL que se puede compilar mediante un compilador JIT y ejecutar en equipos con diferentes arquitecturas. No obstante, el código administrado sólo se ejecutará en un determinado sistema operativo si llama a las API nativas específicas de la plataforma o a una biblioteca de clases específica de la plataforma.

La compilación JIT tiene en cuenta el hecho de que durante la ejecución nunca se llamará a parte del código. En vez de utilizar tiempo y memoria para convertir todo el MSIL de un archivo ejecutable portable (PE) a código nativo, convierte el MSIL necesario durante la ejecución y almacena el código nativo resultante para que sea accesible en las llamadas posteriores. El cargador crea y asocia un código auxiliar a cada uno de los métodos del tipo cuando éste se carga. En la llamada inicial al método, el código auxiliar pasa el control al compilador JIT, el cual convierte el MSIL del método en código nativo y modifica el código auxiliar para dirigir la ejecución a la ubicación del código nativo. Las llamadas posteriores al método compilado mediante un compilador JIT pasan directamente al código nativo generado anteriormente, reduciendo el tiempo de la compilación JIT y la ejecución del código.

El motor en tiempo de ejecución proporciona otro modo de compilación denominado generación de código en el momento de la instalación. El modo de generación de código en el momento de la instalación convierte el MSIL a código nativo, tal y como lo hace el compilador JIT normal, aunque convierte mayores unidades de código a la vez, almacenando el código nativo resultante para utilizarlo posteriormente al cargar y ejecutar el ensamblado. Cuando se utiliza el modo de generación de código durante la instalación, todo el ensamblado que se está instalando se convierte a código nativo, teniendo en cuenta las características de los otros ensamblados ya instalados. El archivo resultante se carga e inicia más rápidamente que si se hubiese convertido en código nativo con la opción JIT estándar.

Como parte de la compilación MSIL en código nativo, el código debe pasar un proceso de comprobación, a menos que el administrador haya establecido una directiva de seguridad que permita al código omitir esta comprobación. En esta comprobación se examina el MSIL y los metadatos para determinar si el código garantiza la seguridad de tipos, lo que significa que el código sólo tiene acceso a aquellas ubicaciones de la memoria para las que está autorizado. La seguridad de tipos ayuda a aislar los objetos entre sí y, por tanto, ayuda a protegerlos contra daños involuntarios o maliciosos. Además, garantiza que las restricciones de seguridad sobre el código se aplican con toda certeza.

El motor en tiempo de ejecución se basa en el hecho de que se cumplan las siguientes condiciones para el código con seguridad de tipos comprobable:

- La referencia a un tipo es estrictamente compatible con el tipo al que hace referencia.
- En un objeto sólo se invocan las operaciones definidas adecuadamente.
- Una identidad es precisamente lo que dice ser.

Durante el proceso de comprobación, se examina el código MSIL para intentar confirmar que el código tiene acceso a las ubicaciones de memoria y puede llamar a los métodos sólo a través de los tipos definidos correctamente. Por ejemplo, un código no permite el acceso a los campos de un objeto si esta acción sobrecarga las ubicaciones de memoria. Además, el proceso de comprobación examina el código para determinar si el MSIL se ha generado correctamente, ya que un MSIL incorrecto puede llevar a la infracción de las reglas en materia de seguridad de tipos. El proceso de comprobación pasa un conjunto de código con seguridad de tipos definido correctamente, y pasa de forma exclusiva código con seguridad de tipos. No obstante, algunos códigos con seguridad de tipos no pasan la comprobación debido a las limitaciones de este proceso, y algunos lenguajes no producen código con seguridad de tipos comprobable debido a su diseño. Si la directiva de seguridad requiere código con seguridad de tipos y el código no pasa la comprobación, se produce una excepción al ejecutar el código.

4. Ejecutar código.

Common Language Runtime proporciona la infraestructura que permite que la ejecución administrada tenga lugar, así como una gran cantidad de servicios que se pueden utilizar durante la ejecución. Para poder ejecutar un método, primero se debe compilar a código específico del procesador. Los métodos para los que se genera el Lenguaje intermedio de Microsoft (MSIL) se compilan mediante un compilador JIT cuando se les llama por primera vez y, a continuación, se ejecutan. La próxima vez que se ejecuta el método, se ejecuta el código nativo existente resultante de la compilación JIT. El proceso de compilación JIT y, a continuación, la ejecución de código se repite hasta completar la ejecución.

Durante la ejecución, el código administrado recibe servicios como la recolección de elementos no utilizados, seguridad, interoperabilidad con código no administrado, compatibilidad de depuración entre lenguajes diferentes y compatibilidad mejorada con el control de versiones y la implementación.

En Microsoft Windows XP, el cargador del sistema operativo comprueba los módulos administrados examinando un bit en el encabezado de formato COFF (Common Object File Format). El bit que se establece indica un módulo administrado. Si el cargador detecta módulos administrados, cargará mscorlib.dll, y _CorValidateImage así como _CorImageUnloading notifican al cargador cuándo se han cargado y descargado las imágenes de los módulos administrados. _CorValidateImage realiza las siguientes acciones:

5. Garantiza que el código es código administrado válido.
6. Cambia el punto de entrada en la imagen a un punto de entrada en el motor en tiempo de ejecución.

En Windows de 64 bits, _CorValidateImage modifica la imagen que está en la memoria transformando el formato PE32 en PE32+.

Primer Programa

El siguiente es un programa simple, en modo consola, que escribe un mensaje en pantalla.

```
using System;
class MainClass {
    public static void Main(string[] args) {
        Console.WriteLine("Hello World!");
    }
}
```

Este código se coloca dentro de un archivo de texto, comúnmente con extensión CS, por ejemplo “Introduccion.cs”. Una vez grabado, se puede compilar desde una ventana de comandos utilizando el compilador de C#, el programa CSC.EXE instalado como parte de .NET Framework SDK, de la siguiente forma:

```
csc Introduccion.cs
```

Al ejecutarse este comando se genera el archivo “Introduccion.exe”, el cual contiene código MSIL, por lo que no debe considerarse como un ejecutable estándar de código nativo.

Todo el código, inclusive el punto de entrada Main, debe definirse dentro de clases u otras definiciones de tipos de datos, como se verá más adelante. Todos los tipos de datos, predefinidos o definidos por el programador, están organizados en espacios de nombres. Cuando se definen tipos sin especificar un espacio de nombres, se asume que forman parte del espacio de nombres global. Se verán ejemplos más adelante.

La directiva using especifica un espacio de nombres, System en el ejemplo, donde el compilador puede buscar los tipos de datos utilizados en el programa y definidos fuera del espacio de nombres actual. La clase Console pertenece al espacio de nombres System, por lo que si no se declara éste último utilizando using, se debería colocar el “nombre completo de la clase”, esto es:

```
System.Console.WriteLine("Hello World!");
```

El formato de definición de una clase, si bien muy similar a C++, presenta algunas diferencias claras:

- Todos los miembros de una clase, datos y funciones, se declaran con todos sus modificadores de manera individual, por lo que estos modificadores sólo afectan a dicho elemento.
- No se finaliza una declaración de clase con un símbolo ‘;’.

El formato general de declaración de un método es:

```
[modificadores] <tipo de retorno> <nombre del método>([parámetros]) {  
    <cuerpo del método>  
}
```

Los modificadores especifican características de los métodos, por ejemplo los modificadores de acceso public, private y protected. El uso de los modificadores se verá más adelante.

El método Main es el punto de entrada de todo programa ejecutable en C#. Existen varios formatos, según se desee retornar un valor o manejar los argumentos ingresados en la línea de comando. Este método se define como estático, con el modificador static, debido a que el intérprete de .NET, el CLR, accederá a él sin instanciar un objeto de la clase dentro de la cual se define. Dado que este acceso se realiza desde fuera de la clase, el método Main debe definirse como público, con el modificador public.

El cuerpo del método Main imprime un mensaje en una ventana de consola, utilizando el método estático WriteLine de la clase Console. Más adelante se verá el tema de la entrada y salida de datos.

Comparación con C++

C# esta basado en C++, por lo que comparte su misma sintaxis para la declaración de variables, métodos, sentencias, operadores, controles de flujo y tipos de datos, con algunas modificaciones que se indicarán a lo largo del presente documento.

La declaración de variables tiene, salvo algunas excepciones, la misma sintaxis de C++. A diferencia de C++, en C# no es posible declarar variables en un ámbito global, sólo en ámbito de clase (datos miembros) y en ámbito de funciones (variables locales).

La declaración de funciones tiene, salvo algunas excepciones, la misma sintaxis de C++. A diferencia de C++, en C# no es posible declarar funciones en un ámbito global (funciones globales), sólo en ámbito de clase (métodos).

En general, la sintaxis de las expresiones así como sus reglas de evaluación son las mismas de C++.

Los operadores y su regla de precedencia son muy similares a C++, con las siguientes modificaciones:

- Los operadores de acceso a los miembros de clases y estructuras, ‘->’ y ‘::’, se reemplazan por el operador ‘.’ (punto).

- El operador ‘&’ desaparece, tanto para obtener una dirección de memoria como para definir una variable tipo referencia al estilo de C++.
- El operador ‘*’ para direccionamiento desaparece.
- Se agregan los siguientes operadores:
- checked y unchecked: Sirven para verificar errores de desbordamiento.
- is: Sirve para verificar si un objeto de datos puede ser interpretado como un tipo de dato en particular.
- Los controles de flujo de C++ se utilizan en C# con la misma sintaxis pero con los siguientes cambios:
- Las expresiones que evalúan una condición deben retornar un tipo de dato booleano.
- El control de flujo switch soporta literales de texto.
- Se define el control de flujo foreach y using.
- Se define el bloque finally, como un bloque de ejecución obligatoria, para el manejo de excepciones.

El API de .NET

.NET cuenta con una extensa gama de clases predefinidas. A este conjunto de clases, agrupados en espacios de nombres.

Algunos espacios de nombres más usados se muestran en la Tabla 1 - 6.

Tabla 1 - 6 Espacios de Nombres

Nombre	Descripción
System.Collections	Contiene interfaces y clases que definen diversas colecciones de objetos, tales como listas, colas, matrices, tablas hash y diccionarios.
System.IO	Contiene tipos que permiten lectura y escritura sincrónica y asincrónica en archivos y secuencias de datos.
System.Threading	Proporciona clases e interfaces que permiten la programación multiproceso. Este espacio de nombres incluye una clase ThreadPool que administra grupos de subprocesos, una clase Timer que permite llamar a un delegado después de un período de tiempo determinado y una clase Mutex para sincronizar subprocesos mutuamente excluyentes. System.Threading también proporciona clases para la programación de subprocesos y la notificación de espera

Tipos de Datos

La programación orientada a objetos (POO) lleva al programador a centrar su esfuerzo en el desarrollo de tipos de datos. Este capítulo es un breve resumen de ideas acerca de la teoría general de los tipos de datos, lo que dará al lector una base teórica útil antes de entrar al tema de la POO.

Conceptos Previos

Un lenguaje de programación se implementa construyendo un traductor, el cual traduce los programas escritos en dicho lenguaje de programación, a programas escritos en lenguaje máquina o algún otro lenguaje más cercano al lenguaje máquina.

Se distinguen los siguientes tipos de computadoras:

- Computadora: Conjunto integrado de algoritmos y estructuras de datos capaz de almacenar y ejecutar programas.
- Computadora real o de hardware: Formada por dispositivos físicos.
- Computadora simulada por software: Formada por software que se ejecuta en otra computadora.
- Computadora virtual: Formada por partes de hardware y de software. Es la que ejecuta los programas traducidos por el traductor.

Conceptos sobre Tipos de Datos

Todo programa se puede considerar como la especificación de un conjunto de operaciones que se van a aplicar sobre ciertos datos en un orden determinado. Todo lenguaje, con diferencias, incluye:

- **Datos:** Tipos de datos permisibles.
- **Operaciones:** Tipos de operaciones disponibles.
- **Mecanismos de control:** Mecanismo de control del orden en que las operaciones se aplican a los datos.

Se revisarán los tipos de datos más representativos de los lenguajes de programación:

- **Tipos elementales de datos:** Basados en características disponibles a partir del hardware del computador.
- **Tipos de datos estructurados:** Características simuladas por software.

Objetos de Datos

Mientras que los datos almacenados en la memoria del computador tienen una estructura simple, tratados como bytes, los datos almacenados en una computadora virtual tienen una estructura compleja, tratados como pilas, arreglos, etc.

Un **objeto de datos** es un dato o una agrupación de datos que existe en una computadora virtual durante su ejecución.

Un programa en ejecución utiliza muchos objetos de datos. Estos objetos de datos y sus interrelaciones cambian dinámicamente durante la ejecución.

Según quién los define, los objetos de datos se pueden clasificar en:

- **Definidos por el programador:** Los crea y manipula explícitamente, a través de declaraciones y enunciados.
- **Definidos por el sistema:** Los crea la computadora virtual (comúnmente, de manera automática para el programador, según se requieran) para “mantenimiento” durante la ejecución del programa, y a los cuales el programador no tiene acceso directo, como por ejemplo: pilas de almacenamiento en tiempo de ejecución, registros de activación de subprogramas, memorias intermedias de archivos y listas de espacio libre.

Los objetos de datos funcionan como contenedores de valores de datos, por ejemplo: un número, un carácter, un apuntador a otro objeto de datos, etc.

Es fácil confundir objeto de datos y **valores de datos**. La distinción se aprecia mejor por su implementación: El primero representa un almacenamiento en la memoria del computador; mientras que el segundo; un patrón de bits.

Los objetos de datos tienen un **tiempo de vida** durante el cual pueden usarse para guardar y recuperar valores de datos.

Los objetos de datos se caracterizan por un conjunto de **atributos**, los que determinan el **número y tipos de valores** que pueden contener, así como la organización lógica de estos valores. Los atributos no varían durante el tiempo de vida de un objeto de datos.

Los objetos de datos participan en **enlaces** durante su tiempo de vida, algunos de los cuales pueden cambiar durante este tiempo. Estos enlaces son:

- **Localidad:** Es la posición que ocupa en la memoria.
- **Valor:** Por lo general resulta de una operación de asignación.
- **Nombre:** Se establece mediante declaraciones y se modifica mediante llamadas y devoluciones de programas.
- **Componente:** Enlace de un objeto de datos a otros de los que forma parte. Se suele representar a través de un apuntador.

Un objeto de datos es **elemental** si su valor de datos se manipula como una unidad, y es una **estructura de datos** si es un agregado de otros objetos de datos.

Variables y Constantes

Una **variable** es un objeto de datos definido y nombrado explícitamente en un programa. Si el objeto de datos es **elemental**, la variable es **simple**. Por lo común, el(los) valor(es) de una variable es(son) modificable(s), mediante operaciones de asignación.

Una **constante** es un objeto de datos, definido y nombrado explícitamente en un programa, y enlazado permanentemente a un valor de datos durante su tiempo de vida.

Una **constante literal** es una constante cuyo nombre es la representación por escrito de su valor.

Una **constante definida por el programador** es una constante cuyo nombre es elegido por el programador.

Dada la característica de una constante, el traductor puede utilizar esta información para realizar optimizaciones.

Tipo de Datos

Un **tipo de dato** es una clase de objeto de datos ligados a un conjunto de operaciones para crearlos y manipularlos.

Todo lenguaje tiene un conjunto de **tipos primitivos (o tipos predefinidos)** de datos que están integrados al lenguaje. Un lenguaje puede proveer recursos para definir nuevos tipos de datos, llamados **tipos definidos por el programador (o bien tipos definidos por el usuario)**.

Un **sub-tipo** de datos se define a partir de un tipo de dato o **súper-tipo**, donde su conjunto de valores posibles es un subconjunto del súper-tipo.

Los **tipos referencias** son aquellos cuyos objetos de datos apuntan a otros objetos de datos. Cuando lo que guarda una referencia es una dirección de memoria, se le llama **puntero**.

Los elementos básicos de la especificación de un tipo de datos son:

- Los **atributos** que distinguen objetos de datos de ese tipo.
- Los **valores** que los objetos de datos de ese tipo pueden tener.
- Las **operaciones** que se pueden realizar con los objetos de datos de ese tipo.

Ejemplo: Para un arreglo de enteros tendríamos:

- Atributos: Número de dimensiones, rango de los índices de cada dimensión, tipo de dato de los componentes, es decir, entero.
- Valores: El conjunto de los números enteros soportados por el tipo de datos entero.
- Operaciones: Sub-indización para seleccionar componentes del arreglo, creación de un arreglo, modificación de las dimensiones, etc.

Los elementos básicos de la implementación de un tipo de datos son:

- La **representación de almacenamiento** usada para representar los objetos de datos de ese tipo.
- Los **algoritmos, procedimientos u operaciones** que manipulan la representación de almacenamiento elegida.

Por último, un tipo de datos tiene una **representación sintáctica**, en la que, tanto la especificación como la implementación son en gran medida independientes.

Ejemplo: Para un objeto de datos de un tipo en particular:

- Los atributos se representan con declaraciones o definiciones de tipo.
- Los valores se representan con constantes.
- Las operaciones se representan con símbolos especiales, procedimientos integrados o funciones.

Esta información suele ser utilizada por los traductores para determinar el tiempo de creación de enlaces, la representación de almacenamiento a utilizar, revisar errores de tipos, etc.

Tipos de Datos Primitivos

Los lenguajes de programación suelen tener un conjunto de tipos de datos primitivos elementales, como son: entero, real, carácter, booleano, de enumeración y apuntador, entre otros. La Tabla 2 - 1 muestra algunos de los tipos de datos primitivos elementales más comunes de los lenguajes de programación utilizados a lo largo del curso.

Tabla 2 - 1 Tipos de datos primitivos elementales

Tipos de datos	C++	Java	C#
Booleanos	bool	boolean	Bool
Enteros	char, unsigned char short, unsigned short int, unsigned int long, unsigned long	byte short int long	sbyte, byte short, ushort int, uint long, ulong
Reales	float double long double	float double	float double decimal
Caracteres	char, wchar_t	char (UNICODE)	char (UNICODE)
Enumerados	enum		Enum

Es interesante notar que si bien los tipos de datos utilizados para representar texto (las clases **string** de C++, **String** de Java y **string** de C#) forman parte de la librería estándar de cada lenguaje, se les deben considerar primitivos estructurados, pues son clases y por tanto, contienen una estructura interna.

Tipos de Datos Primitivos en Java

La Tabla 2- muestra los tipos de datos primitivos en Java, el espacio de memoria que ocupan y el rango de valores que pueden tomar.

Tabla 2- 2 Tipos de datos primitivos de Java

Nombre	Tamaño en bits	Valores
Boolean	1	true o false
char	16	'\u0000' a '\uFFFF'
byte	8	Igual que C++
short	16	Igual que C++
int	32	Igual que C++
long	64	Igual que C++
float	32	Igual que C++
double	64	Igual que C++

Note que Java utiliza 2 bytes para representar un carácter. Esto se debe a que Java maneja caracteres siguiendo el estándar UNICODE.

Cuando en una expresión o una sentencia de asignación se mezclan datos primitivos de distinto tipo, cada evaluación de un operador requiere una homogenización de sus operandos, lo que se realiza siguiendo las reglas de Promoción de Tipos que se muestran en la Tabla 2 - . Estas reglas de promoción se realizan automáticamente dado que no causan pérdida de información. Sin embargo, una conversión puede ser forzada mediante una operación cast, por ejemplo:

```
long x = 10;
int y = (int)x;
```

Tabla 2 - 3 Reglas de promoción de tipos de datos en Java

Tipo primitivo	Puede ser promovido a
double	Ningún tipo, no existe ningún dato primitivo más largo.
float	double
long	float o double
int	long, float o double
char	int, long, float o double
short	int, long, float o double

byte	short, int, long, float o double
boolean	Ningún tipo, los booleanos no son considerados números

Al igual que C/C++, este tipo de operaciones puede significar una pérdida de información.

Tipos de Datos en C#

C#, al igual que todos los lenguajes de programación compatibles con .NET, basan sus tipos de datos en el sistema de tipos comunes de .NET o Common Type System (CTS). Este sistema incluye tanto tipos simples (int, char, float, etc.) como complejos (como string y decimal). Todos estos tipos de datos son realmente clases, las que tienen métodos para acciones comunes, por ejemplo: conversión a texto, serialización, identificación del tipo en tiempo de ejecución, conversión a otros tipos, etc.

C# es un lenguaje fuertemente tipificado, a diferencia de C y C++. Por ejemplo, un tipo de dato booleano no se convertirá automáticamente a un entero, a menos que se indique explícitamente que se desea esa conversión, mediante una operación cast. También es posible especificar el comportamiento de un tipo de dato definido por el usuario, cuando se enfrenta a una conversión de tipos explícita e implícita.

Los objetos de datos en C#, variables y constantes, pueden estar almacenados en el stack o en el heap. Los objetos de datos del stack pueden ser primitivos y estructuras. Los objetos de datos del heap corresponden al resto de tipos de datos. A diferencia de C y C++, en C# es el lenguaje el que escoge la ubicación de un objeto de datos, en base a su tipo. A los tipos de datos en stack se les llama tipo valor, a los tipos de datos en heap se les llama tipo referencia.

El heap en C# funciona de manera diferente al de C y C++. En C#, el CLR crea y gestiona los objetos de datos durante la ejecución del programa, teniendo la responsabilidad de liberar la memoria no utilizada mediante un recolector de basura, el cual es un hilo que corre en paralelo al resto de hilos del programa pero con una prioridad baja, realizando la revisión de los objetos de datos del heap y liberando aquellos marcados como no-utilizados. A este heap se le llama heap gestionado.

La ubicación de almacenamiento de un tipo de dato implica cómo sus objetos se comportarán en una operación de asignación. Para los tipos valor se creará una copia del valor, teniéndose dos objetos de datos distintos almacenando el mismo valor. Para los tipos referencia se creará una copia de la referencia, teniéndose dos objetos de datos referenciando al mismo valor en la misma posición de memoria. En el siguiente ejemplo se crea una copia del valor de una variable entera, el cual es un tipo valor:

```
int a1 = 10;  
int a2;  
a2 = a1;
```

En este ejemplo, a1 y a2 son variables de tipo valor que almacenan en posiciones de memoria distintas, el mismo valor. En el siguiente ejemplo se crea una copia de la referencia de una variable cadena, la cual es un tipo referencia:

```
string s1 = "jose";  
string s2 = s1;
```

En este ejemplo, s1 y s2 son variables de tipo referencia, que refieren al mismo objeto de datos en memoria, el cual almacena el literal "jose".

Luego, para datos primitivos o complejos pero de poco tamaño, es más eficiente trabajarlos en el stack, como tipos valor, dado que se evita la sobrecarga que implica la creación y manejo de objetos de datos en el heap. Por ejemplo, no es deseable tener que crear dinámicamente cada entero que se utilice en un programa. Como contraparte, para datos predefinidos complejos y otros definidos por el programador, en donde se almacenan un número considerable de datos, es más eficiente crearlos en el heap y mantener variables que los refieran, eliminando la sobrecarga de mantener múltiples copias de datos extensos.

TIPOS PREDEFINIDOS

La librería estándar de .NET ofrece un conjunto amplio de tipos de datos. La Tabla 1 - 7 muestra los tipos de datos predefinidos de C#.

Tabla 1 - 7 Tipos de datos predefinidos de C#

Tipo	Rango	Tamaño
Tipos de datos integrales		
sbyte	-128 al 127	Entero de 8 bits con signo
byte	0 al 255	Entero de 8 bits sin signo
char	U+0000 al U+ffff	Carácter UNICODE de 16 bits
short	-32,768 al 32,767	Entero de 16 bits con signo
ushort	0 al 65,535	Entero de 16 bits sin signo
int	-2,147,483,648 a 2,147,483,647	Entero de 32 bits con signo
uint	0 al 4,294,967,295	Entero de 32 bits sin signo
long	-9,223,372,036,854,775,808 al 9,223,372,036,854,775,807	Entero de 64 bits con signo
ulong	0 al 18,446,744,073,709,551,615	Entero de 64 bits sin signo
Tipos de datos de punto flotante		
float	$\pm 1.5 \times 10^{-45}$ al $\pm 3.4 \times 10^{38}$	32 bits, 7 dígitos de precisión
double	$\pm 5.0 \times 10^{-324}$ al $\pm 1.7 \times 10^{308}$	64 bits, 15 a 16 dígitos de precisión
Otros tipos de datos		
decimal	1.0×10^{-28} al 7.9×10^{28}	128 bits, 28 a 29 dígitos significativos

bool	true o false	
char	U+0000 al U+ffff	Carácter Unicode de 16 bits
object	La clase raíz de la que todos los tipos de datos, predefinidos y nopredefinidos, tipo valor o tipo referencia, derivan.	
string	Cadena de caracteres Unicode.	

Dado que el tipo string es un tipo referencia, podríamos esperar que sea sencillo cometer errores de programación al asignar una variable string a otra, y luego al modificar la cadena de la primera estaríamos modificando, quizá inadvertidamente la segunda. Sin embargo, el tipo string presenta una característica particular que evita este tipo de error. Cuando se realiza cualquier operación que modifica el contenido de una variable string, se crea un nuevo objeto de datos, manteniendo el valor de la variable original sin cambios. Luego, la regla es, una vez creado un objeto string, el valor que almacena no puede ser modificado. El siguiente código ejemplifica esta característica.

```
using System;
class MainClass {
    public static void Main(string[] args) {
        string s1 = "Una cadena";
        string s2 = s1;
        Console.WriteLine("s1 es " + s1);
        Console.WriteLine("s2 es " + s2);
        s1 = "Otra cadena";
        Console.WriteLine("s1 es ahora " + s1);
        Console.WriteLine("s2 es ahora " + s2);
    }
}
```

DEFINIDOS POR EL PROGRAMADOR

Todos los tipos de datos se clasifican en dos grupos:

- Tipo valor: Simples (incluyendo los enumerados) y las estructuras.
- Tipo referencia: Clases, interfaces y delegados.

La declaración y uso de las estructuras, enumerados y clases es muy similar a C++. Los siguientes formatos corresponden a la declaración de éstos:

```
[modificadores] struct <nombre del tipo> { <cuerpo de la estructura> }
[modificadores] enum <nombre del tipo> [:<tipo entero>] { <declaración de las constantes> }
[modificadores] class <nombre del tipo> [: <clases e interfaces>] { <cuerpo de la clase> }
```

El siguiente código muestra un ejemplo simple del uso de estos tres tipos:

```
using System;
enum Sexo {
    Masculino,
    Femenino
}
struct Persona {
    public string nombre;
    public int edad;
    public Sexo sexo;
}
class MainClass {
    public static void Main(string[] args) {
        Persona p = new Persona();
        p.nombre = "Jose";
        p.edad = 25;
        p.sexo = Sexo.Masculino;
        string saludo = CrearSaludo(p);
    }
}
```



```
        Console.WriteLine(saludo);
    }

    public static string CrearSaludo(Persona p) {
        string saludo = "Bueno dias";
        switch(p.sexo) {
            case Sexo.Masculino:
                saludo += " Sr. ";
                break;
            case Sexo.Femenino:
                saludo += " Sra. ";
                break;
        }
        saludo += p.nombre;
        return saludo;
    }
}
```

Nótese que los datos miembros de una estructura se deben declarar public para ser accedidos directamente. Tanto para las clases como para las estructuras cuando uno de sus miembros no especifica el nivel de acceso, mediante un modificador, se asume por defecto private. C++ asume por defecto private para las clases y public para las estructuras. Aunque para la inicialización de una estructura se utiliza el operador new, es importante recordar que los datos de ésta se encuentran en el stack, no en el heap. Existen otras diferencias entre las estructuras de C++ y C# que van más allá de una introducción al lenguaje.

Nótese que el formato para acceder a los elementos de un enumerado difiere del de C++. C# no permite definir un tipo enumerado anónimo, como sí lo permite C++. A diferencia de C++, el nombre de un tipo enumerado define un espacio de nombres. Es por ello que para acceder a un elemento del enumerado se requiere el formato:

```
<nombre del enumerado>.<nombre de la constante>
```

Las interfaces y los delegados son temas avanzados que van más allá de una introducción al lenguaje.

CONVERSIÓN ENTRE TIPOS

El siguiente código produce un error poco claro:

```
byte valor1 = 50;
byte valor2 = 100;
byte total;
total = valor1 + valor2;
Console.WriteLine(total);
```

El error es: CS0029: Cannot implicitly convert type 'int' to 'byte'. Esto se debe que, debido a que la suma de dos números tipo byte, cuyo rango de valores va del cero al 255, puede producir fácilmente un número mayor a 255, lo que requeriría un tipo de dato entero de por lo menos dos bytes. Debido a esto, la suma de tipos byte en C# retorna un valor entero tipo int, lo que origina el error en el código, dado que el resultado de la suma, un int, se intenta asignar a una variable byte, lo que podría producir una pérdida de información. Luego, para solucionar este problema requerimos de una conversión de un tipo de dato a otro.

En C# existen dos formas de conversión del valor de una variable: implícita y explícita. La conversión implícita es realizada en forma automática por el compilador del lenguaje, sin que el programador lo solicite, solamente en los casos donde es seguro que no se perderá información o se modificará el valor original. En los demás casos, se requerirá una conversión explícita, donde el programador solicita la conversión mediante una sintaxis especial.

El siguiente es un ejemplo de conversión implícita:

```
byte valor1 = 50;
byte valor2 = 100;
long total;
total = valor1 + valor2;
Console.WriteLine(total);
```

Dado que toda suma de dos valores tipo byte siempre pueden ser almacenados en una variable tipo long, el compilador realiza una conversión implícita. La Tabla 1 - 8 muestra las conversiones implícitas para los tipos de datos primitivos.

Tabla 1 - 8 Conversiones implícitas para tipos de datos primitivos en C#

Tipo primitivo	Puede ser convertido a
sbyte	short, int, long, float, double, o decimal
byte	short, ushort, int, uint, long, ulong, float, double, o decimal
short	int, long, float, double, o decimal
ushort	int, uint, long, ulong, float, double, o decimal
int	long, float, double, o decimal
uint	long, ulong, float, double, o decimal
long	float, double, o decimal
char	ushort, int, uint, long, ulong, float, double, o decimal
float	double
ulong	float, double, o decimal

Para realizar una conversión explícita requerimos realizar una operación cast, al igual que en C++. El siguiente código realiza una conversión explícita:

```
long valor1 = 30000;
int valor2 = (int)valor1;
```

Es importante recordar que el riesgo de pérdida o modificación de valores en este tipo de operación corre por cuenta del programador.

Para los datos primitivos, sólo se permiten las conversiones entre tipos enteros y caracteres. No es posible realizar una conversión ni implícita ni explícita desde estos tipos a un booleano y viceversa.

Para convertir un tipo primitivo a una cadena se puede utilizar el método ToString. Este método es heredado por todos los tipos de datos de la clase base object. El siguiente código muestra la conversión de un entero a una cadena.

```
int i = 10;
string s = i.ToString();
```

El método ToString es llamado automáticamente cuando se concatena, con una operación de suma '+', cualquier objeto de datos, variables o constantes (la declaración de éstas se verán más adelante), con un objeto tipo string. El siguiente ejemplo muestra este caso.

```
int entero = 100;
double real = 534.65;
bool booleano = true;
string cadena = "entero=" + entero + ", real=" + real +
", booleano=" + booleano;
Console.WriteLine("cadena=" + cadena);
```

Al ejecutar este código se mostrará en la ventana de consola lo siguiente:

```
cadena=entero=100, real=534.65, booleano=True
```

Para convertir una cadena a un tipo primitivo es posible utilizar el método estático Parse de estos. El siguiente código convierte un objeto string a un objeto int.

```
string s = "123";
int e = int.Parse(s);
```

Este código refuerza la idea de que “todos” los tipos de datos en C# son objetos, aun los datos primitivos e inclusive las constantes literales, por lo que el siguiente código sería válido.

```
string s = 10.ToString();
```

Sin embargo, lo que ocurre en el fondo en este código es la creación de un objeto temporal en el heap que encajone un valor entero, de forma que pueda llamarse al método requerido. A este proceso se llama encajonamiento o boxing. También es posible realizar un boxing explícitamente, como en el siguiente código.

```
int i = 123;
object obj = i;
Console.WriteLine("obj = " + obj);
```

Es interesante notar que al ejecutarse este código se imprime en pantalla la siguiente línea.

```
obj = 123
```

La variable tipo referencia o realmente referencia a un objeto tipo int pero en heap. Dado que el método ToString originario de la clase object es sobrescrito por su clase derivada int, su llamada se realiza polimórficamente, lo que origina el resultado mostrado. Este proceso se muestra en la Figura 1 - 1.

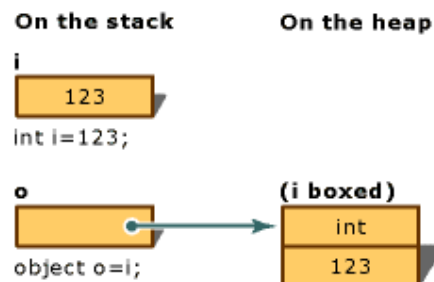


Figura 1 - 1 Proceso de boxing en C#

Un proceso boxing puede realizarse sobre cualquier objeto tipo valor, como los enteros y las estructuras. El proceso inverso se llama unboxing. El siguiente código muestra un ejemplo de su uso.

```
int i = 20;
object obj = i;
int j = (int)obj;
```

```
j++;  
Console.WriteLine("i=" + i + ", j=" + j);
```

La salida producida al ejecutar este código es la siguiente.

```
i=20, j=21
```

Como se puede ver, el modificar el valor de la variable `j` no afecta a la variable original `i`. Igualmente, el objeto entero encajonado por la variable `obj` es una copia del valor de la variable `i`, esto es, `obj` no es una referencia a la variable `i` original, `obj` únicamente contiene una copia del valor original de `i`.

Si bien es poco usual que se requiera realizar un proceso de boxing o unboxing explícitamente, el que existan los mecanismos que permitan hacerlo permitiría, en caso de requerirlo un programador, trabajar con cualquier tipo de dato, inclusive los primitivos, de manera uniforme.

Las Operaciones sobre los Tipos de Datos

Se definen dos tipos de operaciones sobre los tipos de datos:

- Las **operaciones primitivas** son las que se especifican como parte de la definición del lenguaje.
- Las **operaciones definidas por el programador**.

Los objetos de datos requeridos en una operación se denominan **argumentos**.

La **aridad** de una operación corresponde al número de argumentos que utiliza.

El **dominio** de una operación define el conjunto de posibles argumentos de entrada que pueden utilizarse.

El **intervalo** de una operación define el conjunto de posibles resultados que puede producir.

La **acción** de una operación define los resultados que se producen para un conjunto dado de valores.

Ejemplo: La operación binaria de suma entera se puede representar como:

```
+ : entero x entero → entero
```

Ejemplo: La operación raíz cuadrada se puede representar como:

```
SQRT : real → real
```

Una especificación precisa de la acción de una operación requiere más información que únicamente los tipos de datos de los argumentos. Esta especificación se ve dificultada por:

- Operaciones que no están definidas para ciertas entradas.
- Argumentos implícitos.
- Efectos colaterales, esto es, resultados implícitos.
- Auto modificación, esto es, sensibilidad historial.

Ejercicio

1. Encuentre los errores si es que hubiera en los siguientes códigos y corrijalos.

- `byte sizeof = 200;`

- short mom = 43;
- short hello mom;
- int big = sizeof * sizeof * sizeof;
- long bigger = big + big + big // ouch
- double old = 78.0;
- double new = 0.1;
- boolean consequence = true;
- boolean max = big > bigger;
- char maine = "New Peru Economy";
- char ming = 'd';

2. ¿Qué tipo de dato utilizaría para almacenar?

- Su calificación final en LP2.
- La velocidad de la luz.
- Su nota media de este trimestre.
- El número de alumnos de la facultad.
- Localización de un punto en la pantalla.
- 2^{65}
- 234,77 dólares.
- La mitad de 234,77 dólares.
- Bits por segundo transmitidos por un módem.

3. Cómo utilizar los tipos de datos Java

```
public class DataTypes {
    public static void main(String[] args) {
        boolean isReal=true; // Los nombres son sensibles a
                               // mayúsculas y minúsculas,
        // deben empezar por una letra y
        // pueden contener números,_, $
        byte d= 122; // Deben ser inferiores a 127
        short e= -29000; // Deben ser inferiores a 32767
        int f= 100000; // Deben ser inferiores a 2100 mil.
        long g= 999999999999L; // Deben poner L al final
        float h= 234.99F; // Deben ser < 3E38; F al final
        double i= 55E100; char cvalue= '4'; // char '4' no es el entero 4
        //Las cadenas (strings) son objetos, no primitivos.
        //Ejemplo: String nombre= "Claudius";
    }
}
```


Arreglos y Cadenas

Debido a que el tratamiento de cadenas de caracteres (o simplemente cadenas) está estrechamente relacionado al de arreglos en los lenguajes de programación que nos competen, se verá en este capítulo ambos temas.

Arreglos en Java

Los arreglos en Java son objetos especiales. Cuando se crea un arreglo lo que internamente realiza Java es crear un objeto de tipo arreglo. Se dice que un arreglo es un objeto especial dado que el programador nunca puede usar directamente una clase Arreglo (no hacemos un “new” de alguna clase tipo Arreglo), tampoco podemos crear una clase que herede de la clase Arreglo que internamente implementa Java.

Los arreglos en Java son estáticos en el sentido de qué una vez creados éstos, no pueden redimensionarse. Un objeto arreglo, una vez creado, conserva su tamaño durante todo su tiempo de vida.

Arreglos Unidimensionales

El formato de declaración de un arreglo es:

```
<tipo de los elemento> <nombre del arreglo> [ ];
```

Por ejemplo, para declarar una referencia a un arreglo de enteros podríamos utilizar:

```
int ArregloEnteros[ ];
```

ArregloEnteros es el nombre de una referencia que puede ser utilizada para referenciar a cualquier objeto arreglo de enteros. La referencia ArregloEnteros se declara pero aún no se inicializa. Como todo objeto, una referencia no inicializada tiene el valor null.

Para inicializar un arreglo se usa el operador new, indicando entre corchetes el tamaño del arreglo. Por ejemplo, para referenciar nuestra variable anterior ArregloEnteros a un objeto arreglo de 10 enteros se puede utilizar:

```
ArregloEnteros = new int [ 10 ];
```

La sentencia anterior crea un objeto arreglo de enteros y asigna a ArregloEnteros su referencia. Otra forma de hacer lo mismo sería:

```
int ArregloEnteros[ ] = new int [ 10 ];
```

Al igual que C++, un arreglo puede inicializarse utilizando un inicializador, por ejemplo:


```
int ArregloEnteros[ ] = { 10, 5, 2, 3 };
```

Donde el tamaño del arreglo creado y asignado corresponde al número de elementos en el inicializador.

Cuando se crea un objeto arreglo (con el operador new), los elementos del arreglo son inicializados según las mismas reglas de inicialización automática:

- Datos primitivos numéricos son inicializados en cero (0).
- Datos primitivos booleanos son inicializados en false.
- Datos tipo referencia son inicializados en null.

En Java, una referencia a un arreglo no puede declararse e inicializarse al estilo de C/C++. Por ejemplo, la siguiente sentencia arrojaría un error de compilación:

```
int ArregloEnteros[ 10 ];
```

Como en cualquier declaración de variables, puede inicializarse más de una referencia a variables de tipo arreglo durante su declaración:

```
int Arreglo1[ ] = new int [ 10 ],  
    Arreglo2[ ] = { 10, 5, 2, 3 };
```

Al igual que el resto de datos miembro de una clase, los datos miembros que son referencias a objetos de tipo arreglo también pueden declararse e inicializarse a la vez. Por ejemplo, el siguiente código es correcto:

```
class MiClaseConArreglos {  
    ...  
    double Arreglo[ ] = new double [ 205 ];  
    ...  
}
```

Arreglos Multidimensionales

Java no soporta originalmente la declaración directa de arreglos de múltiples dimensiones. En contraparte, se pueden crear arreglos de arreglos. Por ejemplo, para crear un arreglo bidimensional de enteros usaríamos:

```
int b [ ] [ ];  
b = new int [ 10 ] [ ];  
b[ 0 ] = new int [ 5 ];  
b[ 1 ] = new int [ 5 ];
```

En el ejemplo anterior se declara un arreglo bidimensional de 10x5. También podemos utilizar un inicializador:

```
int b [ ] [ ] = { { 1, 2, 3 }, { 4, 7, 10, 6 } };
```

Donde tendríamos un arreglo de 2 elementos en el cual:

- b[0] refiere a un arreglo de 3 elementos
- b[1] refiere a un arreglo de 4 elementos

Nótese que, dado que se crea realmente un arreglo de arreglos, cada elemento de la primera dimensión puede estar refiriendo a un arreglo de dimensiones distintas.

Si lo que deseamos crear es un arreglo bidimensional, por ejemplo “b”, donde cada elemento de la primera dimensión de “b” sea un arreglo con el mismo tamaño, podemos usar la sintaxis:

```
int b[ ] [ ] = new int [ 5 ] [ 3 ];
```

Donde crearíamos un arreglo bidimensional de 5x3.

Acceso a los Elementos de un Arreglo

La primera posición de un arreglo, al igual que C++, es cero. Los índices que se utilizan para referirse a los elementos de un arreglo deben ser valores enteros o expresiones que, al ser evaluadas, produzcan un valor entero.

Dado que los arreglos son objetos en Java, cada arreglo creado conoce cuál es su tamaño. El programador puede averiguar el tamaño de un arreglo mediante la variable miembro “length”.

Para referirnos a los elementos de un arreglo utilizamos la misma sintaxis que en C++. Por ejemplo:

```
int a[ ] = new int [ 4 ];
int b[ ] [ ] = new int [ 5 ] [ 3 ];
int iValor = 65;
a[ 0 ] = 400;
a[ 1 ] = iValor;
a[ 2 ] = 30;
a[ 3 ] = 555;
b[ 2 ] [ 1 ] = a[ 0 ] + a[ 1 ];
```

Si quisiéramos recorrer los elementos de un arreglo podemos utilizar la variable miembro “length” para averiguar su longitud. Por ejemplo:

```
void MostrarArreglo( int a[ ] [ ] ) {
    for( int i = 0; i < a.length; i++ )
        for( int j = 0; j < a[ i ].length; j++ )
            System.out.println( "Valor = " + a[ i ] [ j ] );
}
```

Como último ejemplo, la creación de un arreglo de objetos String podría ser:

```
String a[ ] = new String [ 2 ];
a[ 0 ] = "Hola";
a[ 1 ] = "Adiós";
```

Recuerde que, como los elementos en este arreglo son referencias, éstas son inicializadas automáticamente a null cuando se crea el arreglo. Luego, el siguiente código generaría un error en tiempo de ejecución:

```
String a[ ] = new String [ 10 ];
System.out.println( a[ 0 ] );
```

Dado que el elemento de índice cero del arreglo se está utilizando antes de ser inicializado.

Paso de un Arreglo a un Método

Los arreglos, al igual que todos los objetos, son pasados a un método por valor, pero como los objetos son referencias permiten la modificación de su contenido. Por ejemplo:

```
class MiClaseConArreglos {
    ...
    void Metodo1( ) {
        ...
        double Arreglo[ ] = new double [ 205 ];
        Metodo2( Arreglo );
        ...
    }
    ...
    void Metodo2( double Arr [ ] ) {
        ...
    }
    ...
}
```

El método Metodo2 recibe un parámetro tipo arreglo en la referencia Arr. Tanto Arreglo en Metodo1 como Arr en Metodo2 son referencias al mismo objeto arreglo, lo cual significa que al modificar los valores de los elementos del arreglo mediante la referencia Arr se estaría también modificando los valores a los que refiere Arreglo. Sin embargo, si a la variable Arr en Metodo2 se le referencia a otro arreglo, la variable Arreglo en Metodo1 seguirá referenciando al arreglo original.

Preguntas de Repaso

- Cuando se pasa un arreglo a un método, si se modifica el parámetro correspondiente dentro del método (por ejemplo, se le asigna otro arreglo), ¿se modificará también la referencia que se usó en la llamada al método?
- Cuando asigno a una referencia de un arreglo el valor de otra referencia de otro arreglo, ¿estoy sacando una copia? Si no, ¿cómo se sacaría una copia?
- ¿Cómo se declara un método que debe recibir como parámetro un elemento de un arreglo multidimensional?
- Si tengo un arreglo multidimensional y paso uno de sus elementos a un método, si modifico el parámetro referencia correspondiente, ¿modifico el elemento del arreglo multidimensional también?

Arreglos en C#

Los arreglos en C# son un tipo de clase predefinida especial, dado que su creación y manipulación difiere de las clases estándar. Aunque los programadores tienen acceso a la clase base de todos los arreglos, la clase Array, no puede derivar directamente de ésta, sino indirectamente a través de los formatos de declaración de arreglos.

El siguiente es el formato de declaración de una variable de tipo arreglo unidimensional:

```
[modificadores] <tipo de los elementos> [ ] <nombre de la variable>;
```

A diferencia de C++, los corchetes van entre el tipo de los elementos y el nombre de la variable, no especificándose las dimensiones del arreglo. El siguiente código muestra un ejemplo del uso de un arreglo unidimensional.

```
using System;
class MainClass {
    public static void Main(string[] args) {
        int[] a;
        a = new int[10];
        a = new int[3];
        a[0] = 10;
        a[1] = 20;
        a[2] = 30;
        for(int i = 0; i < a.Length; i++)
            a[i] += i;
        int suma = 0;
        foreach(int elemento in a)
            suma += elemento;
        Console.WriteLine("Suma total = " + suma);
    }
}
```

Nótese que la variable “a” es de tipo referencia, dado que los arreglos son objetos. En la primera asignación, “a” recibe una referencia a un objeto de tipo arreglo de enteros de diez elementos. En la segunda asignación, “a” recibe una referencia a un nuevo objeto arreglo, esta vez de 3

elementos, quedando el primer objeto arreglo sin modificación, aunque dado que ya no es referenciado por ninguna variable será eliminado de memoria por el recolector de basura. Como se ve, los objetos arreglo, una vez creados, no pueden modificar sus dimensiones.

Dado que un arreglo es un objeto, contiene datos miembro y métodos. El dato miembro “Length” (más adelante veremos que realmente se trata de una propiedad, un concepto definido en C#) retorna el número de elementos del arreglo.

Los objetos arreglo se crean en el heap gestionado, a diferencia de los arreglos de C++, que se crean en el stack. En C++, los arreglos en el heap se crean mediante punteros.

C# permite manejar arreglos de arreglos, así como arreglos multidimensionales. Para el primer caso la sintaxis es la siguiente:

```
[modificadores] <tipo de los elementos> [ ][ ] ... <nombre de la variable>;
```

El siguiente código muestra un ejemplo del uso de un arreglo tridimensional.

```
using System;
class MainClass {
    public static void Main(string[] args) {
        int[][][] b;
        b = new int[2][][ ];
        b[0] = new int[3][ ];
        b[1] = new int[3][ ];
        b[0][0] = new int[2];
        b[0][1] = new int[3];
        b[0][2] = new int[4];
        b[1][0] = new int[7];
        b[1][1] = new int[8];
        b[1][2] = new int[9];
        for(int i = 0; i < b.Length; i++)
            for(int j = 0; j < b[i].Length; j++)
                for(int k = 0; k < b[i][j].Length; k++)
                    b[i][j][k] += i+j+k;

        int acumulado = 0;
        foreach(int[] e1 in b)
            foreach(int[] e2 in e1)
                foreach(int e3 in e2)
                    acumulado += e3;
        Console.WriteLine("Acumulado total = " + acumulado);
    }
}
```

El arreglo “b” se dice que es “ortogonal”, debido a que no todos los arreglos, dentro de una misma dimensión, tienen el mismo número de elementos. La contraparte es un arreglo rectangular. Si lo que deseamos es crear un arreglo de arreglos rectangular debemos asegurarnos que todos los arreglos creados, para una misma dimensión, tengan el mismo número de elementos.

La sintaxis para el caso de un arreglo unidimensional es la siguiente:

```
[modificadores] <tipo de los elementos> [<tantas comas como dimensiones menos 1>] <nombre de la variable>;
```

El siguiente código muestra un ejemplo del uso de un arreglo multidimensional de tres dimensiones:

```
using System;
class MainClass
{
    public static void Main(string[] args)
    {
        // arreglo tridimensional de enteros
        int[, ,] c;
        c = new int[5,10,8];
        for(int i = 0; i < c.GetLength(0); i++)
```

```
        for(int j = 0; j < c.GetLength(1); j++)
            for(int k = 0; k < c.GetLength(2); k++)
                c[i,j,k] += i+j+k;

int resultado = 0;
foreach(int e in c)
    resultado += e;
Console.WriteLine("Resultado total = " + resultado);
Console.WriteLine("Para el arreglo 'c'");
Console.WriteLine("= " + c.Length);
Console.WriteLine("= " + c.Rank);
    }
}
```

En el ejemplo, el método “GetLength” reemplaza al uso de Length, dado que si bien Length también se define para arreglos multidimensionales, retorna el total de elementos del arreglo multidimensional sumados los elementos de todas sus dimensiones. Para el ejemplo anterior, Length retorna 24. El método GetLength recibe como parámetro el número de la dimensión de la que se desea saber su tamaño. Para el ejemplo anterior GetLength(0) retorna 5, GetLength(1) retorna 10, GetLength(2) retorna 8.

Un objeto tipo arreglo de arreglos se diferencia de un arreglo ortogonal en que el primero tiene su memoria dispersa, mientras que el segundo, junta. La Figura 3 - 1 muestra esta diferencia.

La segunda diferencia es que todos los arreglos multidimensionales son forzosamente rectangulares, debido a que el tamaño de todas las dimensiones debe especificarse al momento de crear el objeto arreglo.

También existe la posibilidad de mezclar ambos tipos de arreglos al declarar una variable. Por otro lado, la clase base Array ofrece métodos estáticos que permiten realizar operaciones comunes sobre arreglos, como son ordenamiento, inversión, etc. Estos aspectos escapan de los alcances de la presente introducción.

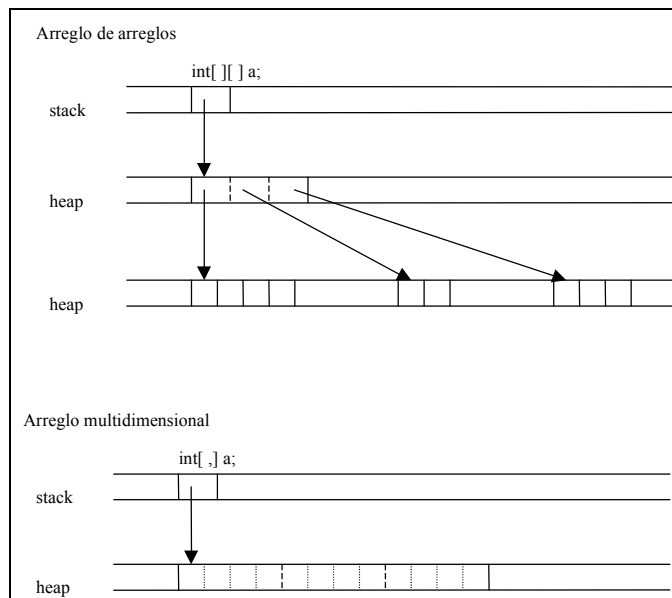


Figura 3 - 1 La distribución de la memoria en los arreglos en C#

Cadenas de Carácter en Java

Java tiene una clase especial (como parte del lenguaje, por lo que está definido en el paquete `java.lang`) para manejar cadenas: “String”. Las referencias de tipo String pueden concatenarse

utilizando el operador “+”. La declaración de un literal de texto provoca que Java cree un objeto String sin nombre de forma automática, la que comúnmente suele asignarse a una referencia para su manipulación. Por ejemplo:

```
String Cadena1 = "Hola";
String Cadena2 = "Mundo";
String Cadena3;
Cadena3 = Cadena1 + " " + Cadena2;
System.out.println( Cadena1 );
System.out.println( Cadena2 );
System.out.println( Cadena3 + "!!!" );
System.out.println( Cadena3 );
```

Provocará la salida:

```
Hola
Mundo
Hola Mundo!!!
Hola Mundo
```

La clase String provee adicionalmente una serie de constructores para la creación de cadenas.

La concatenación de 2 cadenas provoca la creación de un nuevo objeto String, el objeto original no es modificado. Como ejemplo, el siguiente código:

```
String Cadena3;
Cadena3 = "Hola";
Cadena3 += " Mundo";
```

Realizará primera la creación de un objeto String que contiene la cadena “Hola”, luego se creará un nuevo objeto String, en base al primero y a otro objeto String “ Mundo”, y es asignada a la referencia Cadena3. Dado que los objetos String “Hola” y “ Mundo” ya no son referenciados, en algún momento Java los eliminará de memoria (mediante el sistema de Recolección de Basura que se explicará más adelante).

Se puede acceder a los caracteres individuales de una cadena utilizando el método “charAt”. El siguiente ejemplo muestra esto:

```
String s = "Hola";
for(int i = 0; i < s.length(); i++)
    System.out.println("caracter " + i + " = " + s.charAt(i));
```

Es común que en los programas se realicen conversiones entre cualquier tipo de dato a texto. Para los datos primitivos, esta conversión es automática cuando se concatenan con por lo menos una referencia a alguna cadena. Por ejemplo el siguiente código:

```
int Valor = 10;
String Cadena = "Valor = ";
System.out.println( Cadena + Valor );
String Cadena2;
Cadena2 = "Otra concatenación: " + 50.2;
System.out.println( Cadena2 );
```

Provocará la salida:

```
Valor = 10
Otra concatenación: 50.2
```

Sin embargo, cuando no se realiza una concatenación y se desea convertir un número a una cadena se pueden utilizar las clases especiales de Java para esta labor (también en java.lang), las que encapsulan a los datos primitivos en clases y brindan también métodos estáticos (podemos llamarlos sin necesidad de crear un objeto) que nos facilitan acciones comunes como la conversión desde y hacia cadenas de texto (String). Por ejemplo:

```
int Valor = Integer.parseInt( "10" );
```

```
String Cadena = Integer.toString( Valor );
```

La clase Integer encapsula un dato primitivo int y contiene métodos utilitarios que permiten realizar acciones comunes. Para el resto de datos primitivos existen clases equivalentes, por ejemplo la clase Double, Boolean, etc.

La clase String también cuenta con el método estático “valueOf” que permite la conversión de cualquier dato de tipo primitivo a una cadena, por ejemplo:

```
String Cadena = String.valueOf( 10 );
```

La clase String está diseñada para ser eficiente en el manejo de textos no modificables. Si se desea manejar un texto grande que se modificará repetidamente, es más eficiente el uso de la clase “StringBuffer”. Ésta provee métodos para la modificación de su cadena. El siguiente ejemplo muestra el uso de esta clase.

```
StringBuffer sb = new StringBuffer();  
sb.append("hola ");  
sb.append("mundo");  
System.out.println("sb=" + sb);
```

Como puede verse en el ejemplo, sólo un objeto StringBuffer es creado y manipulado, lo que es más eficiente que la creación constante de nuevos objetos cuando se concatenan objetos String.

Al igual que C++, en Java la comparación de dos variables String utilizando el operador “==” produce la comparación de las referencias mismas, y no de los objetos String referenciados, en otras palabras, a menos que ambas variables referencien al mismo objeto una comparación así devolvería false. Si se desea comparar los textos almacenados en los objetos referenciados se debe utilizar métodos como equals y equalsIgnoreCase (equivalentes a compareTo y compareToIgnoreCase). El siguiente ejemplo muestra estas diferencias.

```
String var1 = "Hola";  
String var2 = " a todos";  
String var3 = var1 + var2;  
String var4 = "Hola a todos";  
if(var3 == var4)  
    System.out.println("var3 y var4 SI referencian al mismo objeto");  
else  
    System.out.println("var3 y var4 NO referencian al mismo objeto");  
if(var3.equals(var4))  
    System.out.println("var3 y var4 contienen textos iguales");  
else  
    System.out.println("var3 y var4 contienen textos diferentes");
```

Al ejecutarse este código provocará la salida:

```
var3 y var4 NO referencian al mismo objeto  
var3 y var4 contienen textos iguales
```

Sin embargo algunas políticas de optimización del compilador pueden dar la impresión de que lo anterior no se cumple siempre. Es así como, al ejecutarse:

```
String var1 = "Hola";  
String var2 = "Hola";  
if(var1 == var2)  
    System.out.println("var1 y var2 SI referencian al mismo objeto");  
else  
    System.out.println("var1 y var2 NO referencian al mismo objeto");
```

Se obtiene:

```
var3 y var4 SI referencian al mismo objeto
```

Lo que aquí sucede es que el compilador, tomando en cuenta que los String no son modificables y que ambas variables apuntarían a objetos equivalentes, decide hacer que ambos apunten al

mismo objeto en lugar de crear dos objetos iguales. Por tanto, en este caso ambas variables si están refenciando al mismo objeto.

Por último, existen varias formas de crear objetos String. A continuación algunos ejemplos:

```
char [] arr1 = {'H','o','l','a'};  
String var1 = new String(arr1);  
String var2 = new String(var1);  
String var3 = new String(sb); // sb es tipo StringBuffer
```

Cadenas de Carácteres en C#

Al igual que todos los tipos nativos de C#, el tipo “string” es un alias del tipo CTS System.String. Un objeto string se crea automáticamente cuando se ejecuta alguna sentencia que incluya literales de texto. Por ejemplo, el siguiente código crea un objeto string conteniendo la palabra “hola” y se le asigna su referencia a una variable.

```
string cad = "hola";
```

También puede crearse objetos string utilizando cualquiera de sus constructores. Algunos de estos son:

```
public string(char[]); // Crea un string en base a un arreglo de caracteres.  
public string(char, int); // Crea un string de n carácteres iguales.  
// Crea un string en base a un conjunto de valores de un arreglo de caracteres.  
public string(char[], int, int);
```

El siguiente ejemplo utiliza estos constructores:

```
char[] arr = {'H','o','l','a'};  
string cad = new string(arr);  
Console.WriteLine("cad = " + cad);  
  
string cad2 = "Adios";  
Console.WriteLine("cad2 = " + cad2);  
  
string cad3 = new string('l',5);  
Console.WriteLine("cad3 = " + cad3);  
  
cad3 = new string(arr, 1, 2);  
Console.WriteLine("cad3 = " + cad3);  
  
Una cadena puede ser trabajada como un arreglo, por lo que el siguiente código es válido:  
string cad = "Como un arreglo";  
for(int i = 0; i < cad.Length; i++)  
Console.WriteLine(" cad(" + i + ") = " + cad[i]);
```

Una vez creada una cadena, esta no es modificable, por lo que la siguiente instrucción no es válida:

```
cad[i] = 'P';
```

Si bien es de esperarse que la comparación de dos variables tipo referencia corresponda a la comparación de los valores de referencia y no de los contenidos de cada objeto referenciado, la clase string sobrecarga el operador de comparación, de manera que lo que se compare sea el contenido. Adicionalmente la clase sobrescribe el método “Equals” heredado de System.Object (la clase base de todos los tipos de datos) e implementa los métodos “Compare” y “CompareOrdinal”. El siguiente código muestra algunos casos de comparación entre cadenas utilizando estos métodos.

```
using System;  
class Principal {  
    public static void Main(string[] args) {  
        string cad1 = "Hola Mundo";
```



```
for(int idx = 0; idx < cad1.Length; idx++)
    Console.WriteLine(cad1[idx]);
string cad2;
string cad3;
Console.Write("Primera cadena: ");
cad2 = Console.ReadLine();
Console.Write("Segunda cadena: ");
cad3 = Console.ReadLine();
if(cad2 == cad3)
    Console.WriteLine("cad2 y cad3 iguales");
else
    Console.WriteLine("cad2 y cad3 diferentes");
if(cad2.Equals(cad3))
    Console.WriteLine("cad2 y cad3 iguales");
else
    Console.WriteLine("cad2 y cad3 diferentes");
if(String.Compare(cad2, cad3) == 0)
    Console.WriteLine("cad2 y cad3 equivalentes");
else
    Console.WriteLine("cad2 y cad3 no-equivalentes");
}
```

La clase `string` implementa adicionalmente otros métodos que permiten crear un texto con formato (`Format`), buscar un carácter o conjunto de caracteres (`IndexOf`, `IndexOfAny`, `LastIndexOf`, `LastIndexOfAny`), completar una cadena con caracteres de relleno (`PadRight`, `PadLeft`), reemplazar caracteres (`Replace`), partir una cadena en dos cadenas (`Split`), obtener una sección de una cadena (`Substring`), colocar los caracteres a mayúsculas o minúsculas (`ToLower`, `ToUpper`), eliminar los caracteres de relleno a la izquierda y derecha (`Trim`).

Debe tenerse siempre en consideración que la clase `string` esta diseñada para ser eficiente en el manejo de cadenas de longitud pequeña y media. Cada método de la clase que produciría una modificación de la cadena original, retorna como resultado una nueva cadena con las modificaciones, dejando la cadena original intacta. Por esto, se dice que los objetos `string` no son modificables luego de su creación. Si se desea manejar textos de gran tamaño, se puede utilizar la clase

`System.Text.StringBuilder`

Algunos constructores de esta clase son:

```
StringBuilder( )
StringBuilder( int cap )
StringBuilder( string cad )
StringBuilder( int cap, int capmax )
StringBuilder( string cad, int cap )
```

Algunas propiedades son:

<code>Length</code>	(para lectura y escritura)
<code>Capacity</code>	(para lectura y escritura)

Algunos métodos son:

`Append`, `AppendFormat`, `Insert`, `Remove`, `Replace`, `ToString`

Es común que en los programas se realicen conversiones entre cualquier tipo de dato a texto. Para los datos primitivos, esta conversión es automática cuando se concatenan con por lo menos una referencia a alguna cadena. Por ejemplo el siguiente código:

```
int Valor = 10;
string Cadena = "Valor = ";
Console.WriteLine( Cadena + Valor );
string Cadena2;
Cadena2 = "Otra concatenación: " + 50.2;
Console.WriteLine( Cadena2 );
```

Provocará la salida:

```
Valor = 10  
Otra concatenación: 50.2
```

Sin embargo, cuando no se realiza una concatenación y se desea convertir un número a una cadena se pueden utilizar el método ToString. Dado que en C# todos los tipos de datos son clases, inclusive los tipos primitivos, todos los tipos de datos poseen un método ToString heredado de la clase base object. Por ejemplo:

```
string Cadena = 10.ToString( );
```

Para convertir una cadena a un tipo primitivo se puede utilizar el método Parse incluido en todos los tipos primitivos. Por ejemplo:

```
bool valorBool = bool.Parse("true");
```

También se pueden utilizar los métodos de la clase Convert. Por ejemplo:

```
bool          valorBool          =          System.Convert.ToBoolean("true")
```


Programación Orientada a Objetos

Este capítulo asume que el lector posee conocimientos básicos de la Programación Orientada a Objetos (POO) en C++, así como conocimientos de la estructura general y funcionamiento de programas en Java y C#. Los temas tratados refuerzan dichos conocimientos básicos de POO y los profundizan. Estos conceptos son vistos bajo la implementación de los lenguajes de programación C++, Java y C# con el objetivo de que el lector pueda aprender dichos conceptos más allá de la implementación particular de un lenguaje, mediante la identificación de las diferencias y similitudes entre éstos.

Conceptos Básicos

En esta sección se busca revisar los conceptos básicos relacionados a las clases y a los objetos, así como a los miembros que los componen.

Sobre las Clases

Uno de los primeros progresos de abstracción en el manejo de tipos de datos fue la especificación de los llamados Abstract Data Types o ADT. Un ADT es un conjunto de valores de datos y operaciones asociados a éstos, los que son especificados de manera precisa, independientemente de alguna implementación en particular. Los ADT fueron llevados a la práctica en lenguajes como Clu, Mesa y C. Por ejemplo, los ADT en C se implementaron como estructuras y un conjunto de funciones capaces de manipularlas. Siempre que el acceso a la composición interna de estas estructuras sea realizado por dicho conjunto especializado de funciones, se oculta dicha composición.

A este concepto se le conoce como encapsulación. Este concepto buscaba los siguientes objetivos:

- Reducir el efecto producido en un sistema por la modificación de una estructura de datos, facilitando este tipo de labor.
- Focalizar los cambios en las estructuras de datos y en sus operaciones relacionadas, siempre que ambas se puedan circunscribir a la misma ubicación en el código fuente.
- Facilitar la localización y corrección de errores, reduciendo además el alcance de éstos.

La POO va más allá, haciendo que dichas funciones de manipulación formen parte de la descripción del tipo de dato. Juntos, la declaración de datos y las funciones de manipulación, forman el concepto de clase, donde un objeto es una instancia de ésta. La POO también agregó los conceptos de herencia de implementación y comportamiento polimórfico. Los conceptos de POO fueron llevados a la práctica por lenguajes como Ada, C++ y SmallTalk. Es importante señalar que existen diferentes opiniones acerca de los conceptos relacionados con los ADT y las clases, algunos autores incluso los intercambian indistintamente.

Dado que la encapsulación realiza el ocultamiento de información de un objeto, debido a la inseguridad de su manipulación libre y directa, se hace necesario definir un mecanismo que permita realizarla de manera segura. Es aquí donde se define el concepto de interfaz de una clase, la cual es el mecanismo que dicha clase define para comunicarse con su entorno. En la práctica, la interfaz de una clase está formada principalmente por un conjunto de funciones, que forman parte de la definición de la clase, a las que se puede acceder y llamar desde fuera de la clase.

Sobre los Objetos

Las clases son un mecanismo de definición de nuevos tipos de datos. Un objeto es una instancia de una clase. Dentro del contexto de la POO, el concepto de clase corresponde al de tipo de dato, mientras que el concepto de objeto corresponde al de objeto de datos (ver Capítulo 2 la definición de “objeto de datos”).

Al igual que los tipos predefinidos de un lenguaje de programación, los nuevos tipos de datos creados utilizando clases deben implementarse de manera que sus objetos siempre estén en un estado consistente. Por ejemplo, sería inconsistente que la suma de dos enteros provoque inadvertidamente la modificación de los sumandos. Debido a que el estado de un objeto corresponde al de sus datos miembros, la implementación de los métodos de una clase deberá realizarse de manera que cuide que dichos datos se mantengan en un estado consistente. Esta consistencia debe asegurarse durante todo el tiempo de vida de cada objeto de la clase, desde su construcción hasta su destrucción. Para lograrlo, los lenguajes orientados a objetos permiten definir “constructores” y “destructores” en las clases.

Sobre los Miembros

Los miembros son los elementos que pueden declararse dentro de una clase y forman parte de ella.. Las clases pueden tener 3 tipos de miembros:

- **Datos:** Llamados datos miembro o atributos. El término atributo es utilizado para un concepto diferente en .NET; por lo tanto, no se utilizará en este documento.
- **Funciones:** Llamadas métodos.
- **Tipos de datos:** Llamados tipos anidados.

Cada miembro de una clase tiene un modificador de acceso relacionado ya sea implícita o explícitamente. En el caso implícito se aplica el modificador de acceso por defecto correspondiente al lenguaje.

Las Clases

En esta sección veremos las diferencias y similitudes entre la implementación de los lenguajes de programación tratados respecto a la POO.

C++ es una extensión de un lenguaje de programación estructurado, por lo que muchos se refieren a éste como "un lenguaje estructurado con características orientadas a objetos", más que "un lenguaje orientado a objetos real". Todo programa en C++ contiene por lo menos una función fuera de las clases que define, la función de entrada "main", por lo que ningún programa en C++ es completamente orientado a objetos.

A diferencia de C++, los lenguajes Java y C# son más estrictos respecto a su implementación del paradigma orientado a objetos. En estos lenguajes, todo programa consiste en un conjunto de clases, donde una de dichas clases debe definir un método que será tratado por el intérprete como el punto de entrada del programa. A una clase que define un punto de entrada de un programa la llamaremos la "clase ejecutable".

En Java y C# toda clase debe contener también su implementación completa. Java y C#, a diferencia de C++, no permiten la declaración de prototipos de métodos dentro de una clase para su posterior implementación fuera de ésta. No existe por tanto, en estos lenguajes, un equivalente a la declaración de una clase en un archivo cabecera (*.H) y su implementación en un archivo de implementación (*.CPP). En Java y C# una clase se declara e implementa en un mismo archivo. Tampoco permiten la declaración de variables globales ni funciones fuera de las clases.

La Declaración de las Clases

El formato general de declaración de una clase en Java es:

```
[modificadores] class <nombre> [extends <clase base>] [implements <lista interf.>]
{
    <definición de campos y métodos>
}
```

El formato general de declaración de una clase en C# es:

```
[modificadores] class <nombre> [ : <clase base y/o lista de interfaces> ]
{
    <definición de campos y métodos>
}
```

A diferencia de C++, Java y C# permiten utilizar modificadores en la declaración de las clases. La Tabla 4 - 1 muestra los modificadores que pueden utilizarse.

Tabla 4 - 1 Modificadores de clases para Java y C#

Modificadores de Java	Modificadores de C#	Descripción
abstract	abstract	Permiten que una clase no sea instanciada.
final	sealed	Evitan que una clase sea utilizada como base para otra clase.
public, de-paquete (*)	public, protected, private, internal (*)	Determinan el acceso a la clase (por ejemplo, para crear objetos de ésta) desde fuera de su ámbito de declaración. Dependiendo del contexto en que la clase se declare, alguno de estos modificadores no pueden utilizarse.

(*) Modificador por defecto.

La especificación de la herencia se hace mediante la palabra reservada `extends` en Java, y con el símbolo `“:”` en el caso de C#. Cuando una clase no hereda explícitamente de otra, Java asume que ésta hereda de la clase `Object`, mientras que C# asume que lo hace de la clase `object`. En este sentido, toda clase en Java y C# hereda, directa o indirectamente, de una clase base común. A este esquema de implementación de clases se le llama de “árbol único”, dado que toda clase tiene una raíz común. La implementación de C++ se considera de “árbol múltiple”. La implementación de árbol único permite darle una funcionalidad básica común a todos los objetos creados por un programa, sacrificando algo de la eficiencia que se puede lograr con la implementación de árbol múltiple.

Es importante notar que los privilegios de acceso, a diferencia de C++, se definen al declararse la clase, no al usarla como base de una herencia. En el sentido de C++, se puede considerar que todas las herencias en Java y C# son públicas. Se verá el tema de la herencia con detalle más adelante.

La Creación de los Objetos

Los objetos son las instancias de las clases. En C++, Java y C# la creación de un objeto utiliza la palabra reservada `“new”` bajo una sintaxis similar.

El siguiente programa define y utiliza una clase `Usuario` en Java.

```
class Usuario {
    public static final int Administrador = 0;
    public static final int Registrador = 1;
    public static final int Verificador = 2;
    public static final int Consultor = 3;
    private String nombre;
    private String contraseña;
    private int tipo;

    public Usuario(String nom, int tipo) {
        nombre = nom;
        contraseña = "";
        this.tipo = tipo;
    }
    public String ObtenerNombre() {
        return nombre;
    }
}
class Principal {
    public static void main(String[] args) {
        Usuario p;
        p = new Usuario("Jose", Usuario.Administrador);
        System.out.println("Buenos días Sr(a). " + p.ObtenerNombre());
    }
}
```

El siguiente programa es el equivalente en C#.

```
using System;
enum TipoUsuario {
    Administrador,
    Registrador,
    Verificador,
    Consultor
}
class Usuario {
    private string nombre;
    private TipoUsuario tipo;
    public Usuario(string nom, TipoUsuario tipo) {
        nombre = nom;
        this.tipo = tipo;
    }
    public string ObtenerNombre() {
        return nombre;
    }
}
```



```
    }  
}  
class Principal {  
    public static void Main(string[] args) {  
        Usuario p;  
        p = new Usuario("Jose", TipoUsuario.Administrador);  
        Console.WriteLine("Buenos dias Sr(a). " + p.ObtenerNombre());  
    }  
}
```

Si bien la sintaxis es muy similar a C++, existen algunas diferencias importantes:

- Todos los modificadores de acceso, public y private, en el ejemplo, deben especificarse independientemente por cada miembro de la clase, sea un método o un campo. Si no se especifica, se asume un modificador de acceso por defecto (ver tabla 4.2).
- La definición de una clase no termina en un ‘;’.
- La creación de objetos se realiza con el operador new, pero no existe un operador delete, dado que los intérpretes de Java y C# se encargan de gestionar la liberación de la memoria.
- El acceso a los miembros de una clase u objeto siempre es con el operador punto. En Java y C# los objetos son reservados automáticamente en el montón (en pila en algunos casos en C#, como se verá más adelante). el programa no puede decidir esto como sí sucede en C++.
- Todo el código del programa debe estar dentro de los métodos de las clases.

Entre Java y C# también existen diferencias importantes:

- No existen enumerados en Java, por lo que el ejemplo anterior utilizó constantes. Existen otros tipos de datos propios de C#, sin equivalente en Java e incluso en C++.
- El uso de librerías difiere en Java y C#. Para el programa en C# se utilizó la palabra reservada “using”, dado que C# no importa por defecto ninguna librería, ni siquiera las básicas. Para el programa en Java no se requirió realizar un “import”, debido a que solo se utilizó las librerías básicas, las que son importadas por defecto.

Existen otras diferencias que se irán viendo a lo largo del presente capítulo.

Los Constructores y Destructores

El concepto de encapsulación de la POO tiene como objetivo que los programas mantengan sus objetos en un estado consistente durante todo su ciclo de vida. Los constructores son métodos llamados durante la creación de los objetos de forma que éstos tengan un estado consistente desde su nacimiento. Los destructores aseguran una correcta liberación de los recursos reservados por el objeto antes de que éste sea eliminado.

Los Constructores

Las declaraciones de los constructores en C++, Java y C# son:

C++:

```
class MiClase {  
    public:  
        MiClase( .... ) { .... }  
};
```

Java:

```
class MiClase {  
    public MiClase( .... ) { .... }  
}
```

C#:

```
class MiClase {  
    public MiClase( ..... ) { ..... }  
}
```

Los constructores pueden recibir un conjunto de parámetros, igual que cualquier método, para permitir establecer un estado inicial consistente. Cuando un constructor no recibe parámetros se le llama “constructor por defecto”, cuando su parámetro es un objeto del mismo tipo se le llama “constructor copia”. Dado que la implementación interna de muchos lenguajes de programación orientados a objetos requiere que siempre se llame a un constructor cuando se crea un objeto, cuando no se implementa uno explícitamente (sea éste uno “por defecto” o no), el compilador del lenguaje crea un constructor por defecto implícitamente.

C++ implementa una llamada automática al “constructor copia” de una clase cuando se crea un objeto en pila y se le inicializa, pasándole otro objeto del mismo tipo, en su misma declaración. Este manejo especial de los constructores copia no es implementado ni en Java ni en C#, ni tampoco permiten la sobrecarga del operador de asignación para estos fines, lo que sí permite C++.

Java y C# permiten llamar a un constructor desde otro constructor de la misma clase. Esto permite simplificar la definición de varios constructores opcionales para la creación de un objeto. El siguiente código muestra esta característica en C#.

```
class Usuario {  
    private string nombre;  
    private string contraseña;  
    public Usuario(string nombre) : this(nombre, nombre) {  
        // este constructor llama a su vez al segundo constructor pasándole como  
        // contraseña el mismo nombre  
    }  
    public Usuario(string nombre, string contraseña) {  
        this.nombre = nombre;  
        this.contraseña = contraseña;  
    }  
}
```

El siguiente código es el equivalente en Java:

```
class Usuario {  
    private String nombre;  
    private String contraseña;  
    public Usuario(String nombre) {  
        this(nombre, nombre)  
    }  
    public Usuario(String nombre, String contraseña) {  
        this.nombre = nombre;  
        this.contraseña = contraseña;  
    }  
}
```

Aunque la sintaxis utilizada en C# corresponde a la lista de inicialización de los constructores en C++, no es posible utilizar esta sintaxis para inicializar datos miembros de la clase, como sí ocurre en C++.

Como se verá más adelante, en Java y C# también es posible llamar desde un constructor a otro constructor de la clase base.

Los constructores, como cualquier miembro de una clase, pueden también recibir modificadores de acceso en su declaración. Los modificadores de acceso que pueden utilizarse dependen en algunos casos de si la clase es anidada o no. Las clases anidadas se verán más adelante.

Los Constructores Estáticos

A los constructores vistos en la sección anterior se les llama también “de instancia”, debido a que se llaman cuando se instancia una clase. Su finalidad es la de establecer el estado inicial consistente de un nuevo objeto. Sin embargo, una clase puede requerir inicializar datos cuyo tiempo de vida abarquen a todos los objetos de la clase, es decir, datos miembros estáticos. Para este tipo de inicialización se definen constructores estáticos. Java y C# permiten este tipo de constructores. Ejemplos de su declaración en Java y C# son:

Java:

```
class MiClase {  
    static { ..... }  
}
```

C#:

```
class MiClase {  
    static MiClase() { ..... }  
}
```

El siguiente programa en C# muestra el uso de un constructor estático.

```
class NumeroAleatorio {  
    private static int semilla;  
    static NumeroAleatorio() {  
        // imaginemos que obtenemos un valor semilla de,  
        // por ejemplo, el reloj del sistema  
        semilla = 100;  
    }  
    public static int sgte() {  
        // imaginemos que generamos un número aleatorio  
        return semilla++;  
    }  
}  
class Principal {  
    public static void Main(string[] args) {  
        Console.WriteLine("aleatorio 1 = " + NumeroAleatorio.sgte());  
        Console.WriteLine("aleatorio 2 = " + NumeroAleatorio.sgte());  
    }  
}
```

Nótese que el constructor estático no define un nivel de acceso ni tampoco puede tener parámetros, debido a que nunca es llamado por otro código del programa, sólo por el intérprete de Java o .NET cuando la clase es cargada. C++ no tiene un equivalente a un constructor estático.

Los Destruidores

Mientras que la construcción de un objeto sigue esquemas similares de funcionamiento en la mayoría de lenguajes orientados a objetos, la destrucción de los mismos depende de si el esquema de manejo de la memoria es determinista o no-determinista, lo que está relacionado con el momento en que un destructor es llamado. El concepto mismo de destructor de C++ es determinista, lo que significa que el programador puede determinar el momento exacto en que un destructor es llamado durante la ejecución de un programa. Java y C# implementan “finalizadores”, los cuales son no-deterministas.

La sintaxis de declaración de un finalizador en Java es:

```
<modificador protected o public> finalize ( ) {
```

```
<Cuerpo del finalizador>  
}
```

La sintaxis de declaración de un finalizador en C# es:

```
~ <nombre de la clase> ( ) {  
    <Cuerpo del finalizador>  
}
```

En ambos casos no se pueden definir parámetros. Tanto en C++ como en C# no es posible llamar a un destructor o finalizador directamente desde fuera de la clase. En el caso de Java el finalizador es un método más pero con un nombre reservado, por lo que es posible llamarlo directamente desde fuera de la clase. Más aún, este método sobrescribe el método de la clase base dado que la clase Object lo implementa. Las consecuencias de esto se verán en el tema de la herencia.

Debido a que el manejo de la memoria, el recurso más común de los programas, es realizado por el recolector de basura en Java y C#, los finalizadores son rara vez requeridos y se recomienda utilizarlos sólo cuando es estrictamente necesario (dado que su gestión disminuye la performance de un programa), para liberar recursos que no corresponden a la memoria (como por ejemplo, una sesión abierta en una base de datos, el acceso a un puerto del computador, entre otros), o para realizar un seguimiento o depuración de un programa.

La Recolección de Basura

Si bien un objeto se crea utilizando el operador “new”, no existe en Java y C# un operador “delete”. El motivo de esto es que Java y C# le quitan al desarrollador la responsabilidad de la liberación de los recursos creados dinámicamente. Los intérpretes de estos lenguajes guardan un registro de todos los objetos creados y cuántas variables de tipo referencia apuntan a cada uno. Cuando una nueva referencia apunta a un objeto, el contador de referencias de éste aumenta. Cuando una referencia a un objeto se destruye (por ejemplo, una referencia local, la cual se destruye cuando se sale del método que la define) el contador de referencias del objeto relacionado disminuye. De esta forma, cuando el contador de un objeto llega a cero se sabe que ya no es referenciado por ninguna variable del programa. Paralelamente, estos intérpretes cuentan con un sistema llamado recolector de basura, el cual es llamado cada cierto tiempo de forma automática y verifica todos los objetos que ya no cuentan con referencias, esto es, su contador de referencias es cero. Cuando el recolector encuentra un objeto sin referencias lo elimina de memoria. De esta forma Java y C# simplifican enormemente el trabajo con la memoria dinámica.

Manejo Manual y Automático de la Memoria

C++ otorga al programador toda la responsabilidad del manejo de la memoria, lo cual le permite desarrollar programas significativamente más eficientes que sus equivalentes en Java o C#. Sin embargo, la experiencia ha demostrado que esta responsabilidad ha generado más sobrecostos que beneficios para la mayoría de proyectos de software. Estos sobrecostos se traducen en mayores tiempos de desarrollo y depuración de errores durante las etapas de desarrollo y mantenimiento de los programas, así como en posteriores proyectos que involucran la modificación y ampliación de éstos.

La automatización de la gestión de la memoria, ofrecida por Java y C#, quita esta responsabilidad al programador, permitiéndole concentrar su trabajo en la implementación de la llamada “lógica del negocio”, lo que ha permitido generar programas más estables en menor tiempo, con un menor número de errores durante y después de su producción, y facilitando la posterior modificación de dichos programas. Todo esto, a costa de una pérdida “aceptable” de

la performance final de estos programas respecto a sus contrapartes. Más aún, se ha encontrado que en muchos casos un gestor automático de memoria puede tomar decisiones más eficientes en cuanto al manejo de memoria que un programador promedio. Como consecuencia, para proyectos de mediana y gran envergadura, las empresas requerirían contar con programadores expertos en C++ para lograr resultados “significativamente” más eficientes que los obtenidos con programadores “no-expertos” utilizando Java y C#.

Sin embargo, existen indudablemente proyectos cuya naturaleza requiere obtener la mayor eficiencia que el hardware y el software disponible puede brindar, por lo que lenguajes como Java y C# no pueden reemplazar completamente a lenguajes como C++. Como en muchos aspectos del desarrollo de software, el escoger un lenguaje apropiado para el desarrollo de un programa, pasa por conocer cuáles son los requerimientos del producto final.

Finalización Determinista y No-Determinista

El manejo automático de la memoria ofrecido por Java y C# implica que dichos lenguajes faciliten el desarrollo de programas no-deterministas. En un esquema no-determinista el programador no puede determinar en qué momento los objetos que crea, luego de que ya no son referenciados por el programa, son efectivamente eliminados de la memoria. Esto trae consecuencias al momento de decidir la forma en que los recursos reservados por un programa deberán ser eliminados.

Para esto es importante considerar que si bien la memoria es el recurso más comúnmente manejado por los programas, no es el único, por lo que el resto de recursos deberán seguir manejándose de una manera determinista.

Por ejemplo, el sistema operativo Windows permite a los programas crear un número limitado de “manejadores de dispositivos de las ventanas”. Debido a esto, una librería que permita crear ventanas en Java y C# deberá controlar en forma determinista la liberación de estos recursos. Un programa que utilice dicha librería puede crear y luego desechar una gran cantidad de ventanas en un corto período de tiempo. Si el diseñador de esta librería coloca en el finalizador de su clase “Ventana” la llamada al sistema operativo que libera el recurso mencionado, es posible que las llamadas a estos finalizadores tarden lo suficiente como para que el programa trate de crear una nueva ventana y se produzca un error, puesto que ya se crearon todas las que el sistema operativo permitía y su liberación aún está pendiente.

Problemas como el descrito requieren que el programador implemente manualmente un manejo determinista de estos recursos, es decir, se requiere tener la certeza de en qué momento se libera un determinado recurso.

Los recolectores de basura de Java y C# pueden ser controlados hasta cierto punto por el programador, haciendo uso de clases especiales. Estos recolectores ofrecen métodos para despertar manualmente el procedimiento de recolección de basura. Sin embargo, esta solución no es confiable, puesto que el recolector de basura siempre tiene la potestad de decidir si es conveniente o no iniciar la recolección. El no hacerlo así, podría ocasionar que el programa se cuelgue debido a, por ejemplo, un finalizador que es ejecutado por el recolector y que nunca finaliza por un error en la lógica de su código, lo que ocasionaría que el programa se quede esperando a que el proceso de recolección termine, lo que nunca ocurrirá.

Un esquema comúnmente utilizado pasa por colocar el código encargado de la liberación de la memoria en un método del objeto que la crea. Adicionalmente, el objeto deberá contener un dato miembro que funcione como bandera y que indique si dicho método ya fue llamado o no. Si el programador determina que ya no requiere utilizar más un objeto, y sus recursos, deberá llamar manualmente a dicho método. Adicionalmente, el finalizador del objeto también llamará

a este método. Finalmente, todos los métodos del objeto deberán hacer una verificación de la bandera de liberación de los recursos antes de realizar su trabajo, de forma que se impida que el objeto siga siendo utilizado si es que sus recursos ya fueron liberados.

Puede parecer que finalmente la programación no-determinista no es tan buena idea después de todo. Sin embargo, la mayor parte de los recursos que un programa suele utilizar corresponden a la memoria, y para aquellos recursos que no son memoria suelen contarse con librerías especializadas que hacen casi todo el trabajo no-determinista por nosotros.

El Acceso a los Miembros

El acceso a los miembros de una clase desde fuera de la misma se determina mediante los modificadores de acceso de cada miembro. La tabla 4.2 muestra los modificadores de acceso de cada lenguaje según los ámbitos desde dónde se desea que un miembro sea accesible.

Tabla 4 - 2 Modificadores de acceso para miembros de una clase

Ámbitos	C++	Java	C#
Sólo desde dentro de la clase donde es declarada.	private (*)	private	private (*)
Desde la clase donde es declarada y las que deriven de ella, estén o no en la misma librería.	protected	<no definido>	protected
Desde cualquier clase de la misma librería.	<no definido>	de-paquete (*)	internal
Desde cualquier clase de la misma librería y desde otras clases en otras librerías siempre que deriven desde la clase donde es declarada.	<no definido>	protected	protected internal
Desde cualquier clase en cualquier librería.	public	public	public

(*) Modificador por defecto.

De la tabla se puede ver que los modificadores de acceso están relacionados con dos conceptos: La herencia y las librerías. También se puede ver que Java modifica el concepto original de C++ de lo que es un miembro protegido. El siguiente ejemplo en Java muestra los efectos del modificador protected en Java.

```
// Archivo Alpha.java
package PaqueteX;
public class Alpha {
    protected int protectedData;
    protected void protectedMethod() {
        System.out.println("protectedMethod");
    }
}

// Archivo Gamma.java
package PaqueteX;
public class Gamma {
    public void accessMethod(Alpha a) {
        a.protectedData = 10;    // legal
        a.protectedMethod();    // legal
    }
}
```

```
// Archivo Delta.java
package PaqueteY;
import PaqueteX.Alpha;
public class Delta extends Alpha {
    public void accessMethod(Alpha a) {
        a.protectedData = 10;    // ilegal
        a.protectedMethod();    // ilegal
        protectedData = 10;    // legal
        protectedMethod();    // legal
    }
}

// Archivo PruebaDeProtected.java
import PaqueteX.Alpha;
import PaqueteX.Gamma;
import PaqueteY.Delta;
public class PruebaDeProtected {
    public static void main(String[] args) {
        Alpha a = new Alpha();
        Gamma g = new Gamma();
        Delta d = new Delta();
        a.protectedData = 10;    // ilegal
        a.protectedMethod();    // ilegal
        g.accessMethod(a);
        d.accessMethod(a);
    }
}
```

En el caso de la clase Gamma es posible acceder a los miembros protegidos de Alpha, puesto que ambas clases pertenecen al mismo paquete. La clase Delta no puede acceder a los miembros protegidos de Alpha dado que no pertenecen al mismo paquete, excepto sus propios miembros heredados. La clase PruebaDeProtected no puede acceder a los miembros protegidos puesto que no pertenece al mismo paquete.

El Uso de las Variables

Los datos de un programa son manejados mediante variables. La forma y ámbito en que éstas son declaradas determinan muchas de sus características.

La Declaración de las Variables

En Java y C# la declaración de variables en Java es similar a C++. La sintaxis general de declaración de una variable es:

```
[modificadores] <tipo> <nombre> [ = <inicializador>];
```

En Java, las variables se clasifican en:

- Variables de tipo primitivo, correspondientes a los tipos byte, short, int, long, float, double, char y boolean.
- Variables de tipo objeto, creadas sobre la base de tipos de dato “clase”.

En C#, las variables se clasifican en:

- Variables por valor, correspondientes a los tipos primitivos y a las estructuras. Son variables que se reservan en pila.
- Variables por referencia, correspondientes a las clases, interfaces, entre otros. Son variables que representan una referencia en pila a datos reservados en el montón.

Los tipos primitivos no son objetos en Java, lo que permite que su manejo sea más eficiente en cuanto a la memoria utilizada para su almacenamiento y el tiempo de acceso a sus datos. En C# los datos primitivos sí son objetos, lo que le permite a un programa manejar cualquier objeto de

datos como un objeto, sacrificando un poco de eficiencia. En este sentido, C# es más estrictamente orientado a objetos que Java. La librería estándar de Java ofrece además clases que encapsulan a cada uno de sus datos primitivos, para aquellos programas que requieren manipular todos sus datos como objetos.

Por el ámbito donde son declaradas, las variables se clasifican en:

- Variables de clase, que corresponden a los datos miembros de la clase.
- Variables locales, que corresponden a las variables declaradas dentro de los métodos.

C++ permite adicionalmente la declaración de variables globales, lo que no permiten ni Java ni C#.

La Duración y el Ámbito

Todo objeto de datos relacionado a una variable en C++, Java y C#, posee dos características que determinan cómo puede ser utilizado: Su duración y su ámbito.

La duración determina el período en el que un objeto de datos existe en memoria y por tanto, su variable puede ser utilizada. Existen 3 casos:

- Creados y destruidos automáticamente. Corresponde a los datos locales a los métodos (declarados dentro de éstos). Se suele decir que tiene una duración automática, dado que son creados automáticamente cuando el programa entra en el bloque en que son declarados y se destruyen automáticamente al salir de éste. Sus variables son llamadas “variables automáticas”.
- Creados y destruidos conjuntamente con los objetos de los que son miembros. Su duración está ligada a la duración del objeto creado. Sus variables son llamadas “variables de instancia”.
- Creados una sola vez y que existen durante toda la ejecución del programa. Corresponden a los datos miembro estáticos de una clase. Corresponden a las “variables estáticas” y “variables globales” para el caso de C++.

El ámbito determina desde dónde un dato puede ser referido dentro de un programa. Existen 3 tipos de ámbito:

- Ámbito de Clase: Son los datos miembros de una clase. Pueden ser referidos desde cualquier método dentro de dicha clase y desde métodos fuera de ésta siempre que los modificadores de ámbito aplicados lo permitan. Se verá más adelante cómo se aplican dichos modificadores.
- Ámbito de Bloque: Son datos definidos dentro de un bloque, como por ejemplo el bloque que representa el cuerpo de un método o el bloque de una estructura de control (if, for, while y do / while). Estos datos sólo pueden ser referidos dentro del bloque donde son declarados.
- Ámbito Global: Son los datos correspondientes a variables declaradas fuera de las clases y métodos. Sólo soportado en C++.

Un dato en ámbito de bloque oculta otro, con el mismo nombre, en el ámbito de la clase a la que pertenece. El siguiente programa en Java muestra un ejemplo de esto:

```
class PruebaDeAmbito
{
    int iValor = 10;
    ...
}
```



```
void Prueba()
{
    int iValor = 20;
    System.out.println( "iValor = " + iValor );
    System.out.println( "this.iValor = " + this.iValor );
}
...
}
```

Al ejecutar el método Prueba se producirá la siguiente salida:

```
iValor = 20
this.iValor = 10
```

La variable local iValor oculta a la variable de la clase. Para poder hacer referencia a la variable miembro de la clase usamos la palabra reservada this, de la misma forma que se realiza en C++. this representa la referencia de un objeto a sí mismo dentro de uno de sus métodos.

Dentro de un método no se pueden declarar dos variables con el mismo nombre, aún si están declaradas en bloques distintos. El siguiente programa en Java muestra un ejemplo de esto:

```
void Prueba()
{
    int iValor = 20;
    boolean Flag = true;
    if( Flag )
    {
        int iValor;
        ...
    }
}
```

Al ejecutarse se producirá un error de compilación. El siguiente ejemplo muestra esta misma restricción en C#:

```
class PruebaDeAmbito {
    public int entero;
    public void prueba(int entero) {
        this.entero += entero;
        if(entero < 0) {
            int entero = 0; // ERROR
            double real = 2.5;
        } else {
            double real = 3.8;
        }
    }
}
```

En el código, definir un argumento o variable local con el mismo nombre de una variable de clase oculta ésta última, por lo que se requiere utilizar la palabra reservada this, como en C++. Sin embargo, la definición de la variable “entero” dentro del cuerpo de la estructura de control if produce un error de compilación, dado que ya existe una variable local con el mismo nombre en un ámbito padre, el del método. Esto difiere al caso de las variables “real”, cuya declaración no produce un error dado que el ámbito de una no es padre del otro.

La Inicialización Automática

A diferencia de C++, en Java y C# toda variable puede ser inicializada al momento en su declaración, inclusive los datos miembros, con excepción de los parámetros de un método, que son también variables locales, dado que su inicialización corresponde a los valores pasados al momento de llamarse a dicho método. El siguiente código en C# muestra esta inicialización:

```
class Usuario {
    private string nombre = "Jose";
}
```

```
private int edad = 10;
}
```

Las variables locales deben inicializarse antes de ser utilizadas. Por ejemplo, el siguiente código en Java arrojará un error al momento de compilarse:

```
String sTexto;
sTexto += "Hola ";
sTexto += "mundo";
```

Este código puede corregirse de la siguiente forma:

```
String sTexto;
sTexto = "Hola ";
sTexto += "mundo";
```

De la forma:

```
String sTexto = new String( );
sTexto += "Hola ";
sTexto += "mundo";
```

O bien de la forma:

```
String sTexto = new String( "Hola " );
sTexto += "mundo";
```

A diferencia de las variables locales, cuando una variable de clase no es inicializada por el programa durante la creación del objeto al que pertenece, recibe una inicialización por defecto. Esta inicialización se realiza de la siguiente forma:

- Los datos primitivos numéricos se inicializan en cero.
- Los datos primitivos lógicos (boolean en Java, bool en C#) se inicializa en false.
- Las referencias se inicializan en null.

Los Modificadores

Dentro de un método, los únicos modificadores permitidos para las variables son “final” en Java, para variables cuyo valor sólo puede asignarse una vez, y “const” en C#, para constantes.

La Tabla 4 - 3 muestra los modificadores permitidos, fuera de los modificadores de acceso, en Java y C# para variables declaradas en el ámbito de una clase.

Tabla 4 - 3 Modificadores de Variables en Java y C#

C#	Java	Descripción
static	static	Variable estática.
const	final static	Constante.
<sin equivalente>	final	Variable cuyo valor sólo puede asignarse una vez, en la declaración o después.
readonly	<sin equivalente>	Variable cuyo valor sólo puede ser asignado durante la creación del objeto.

<sin equivalente>	transient	Variable no serializada con el objeto durante un proceso de serialización. (*)
<sin equivalente>	volátil	Variable para que el compilador no realice ciertas optimizaciones al momento de generar el BYTECODE que hará uso de él. (*)
new	<sin equivalente>	Variable que oculta otra variable, con el mismo nombre, en una clase base.

(*) El uso de estos modificadores va más allá del alcance del presente curso.

El acceso a los datos miembros estáticos en C# se diferencia a C++ y Java en que:

- No pueden accederse, desde dentro de la clase, mediante la palabra reservada `this`.
- No pueden accederse, desde fuera de la clase, mediante una referencia, debe hacerse mediante el nombre de la clase.

El siguiente código en C# presenta estos dos casos.

```
class Usuario {
    ...
    public static int cuenta = 0;
    public Usuario(string nom, TipoUsuario tipo) {
        this.cuenta++; // ERROR
        cuenta++; // CORRECCIÓN
    }
    ...
}

class Principal {
    public static void Main(string[] args) {
        Usuario p;
        p = new Usuario("Jose", TipoUsuario.Administrador);

        Console.WriteLine("Número de usuarios = " + p.cuenta); // ERROR
        Console.WriteLine("Número de usuarios = " + Usuario.cuenta); // Corrección
    }
}
```

La palabra reservada `readonly` define un dato miembro de sólo lectura, el que sólo puede recibir un valor durante el proceso de creación del objeto, esto es, en la misma declaración de la variable o dentro de un constructor. El siguiente código muestra un ejemplo de este tipo de variable.

```
using System;
class PruebaDeSoloLectura {
    private readonly int valor = 50;
    public PruebaDeSoloLectura() {
        valor = 100;
        valor += 10;
    }
    public void modificar(int nuevo) {
        valor = nuevo; // ERROR
    }
}

class Principal {
    public static void Main(string[] args) {
        PruebaDeSoloLectura obj = new PruebaDeSoloLectura();
        obj.modificar(200);
    }
}
```

En el código anterior, el dato miembro “valor” se inicializa al declararse y su valor es modificado dos veces dentro del constructor. Sin embargo, la modificación de dicho dato miembro en el método modificar produce un error en tiempo de compilación.

La palabra reservada `const` tiene el mismo significado en C# y C++. Sin embargo, una constante en C# es implícitamente estática, mientras que en C++ no. En C#, los datos miembro `const` se diferencian de los `readonly` en que:

- Se pueden definir constantes tanto de clase como locales.
- Deben de ser inicializadas en su declaración.
- Su valor debe poder ser calculado en tiempo de compilación.
- Son implícitamente `static`.

La palabra reservada `new` se verá como parte del tema del manejo de la herencia.

El Uso de los Métodos

Mientras que en C++ pueden definirse funciones tanto dentro como fuera de las clases, en Java y C# sólo pueden definirse funciones dentro de las clases. A las funciones definidas dentro de una clase se les denomina funciones miembro o métodos.

La Declaración de los Métodos

En Java, el formato general de definición de un método es:

```
[modificadores] <tipo-retorno> <nombre> ([lista de parámetros])
[throws <lista de excepciones>]
{
    <cuerpo del método>
}
```

En C#, el formato general de definición de un método es:

```
[modificadores] <tipo-retorno> <nombre> ( [lista de parámetros] )
{
    <cuerpo del método>
}
```

En Java y C#, tanto la lista de parámetros como los modificadores son opcionales. El valor de retorno puede ser cualquier tipo de dato o bien puede ser `void`, lo que indica que el método no devuelve ningún valor.

En Java y C#, un método sin parámetros no puede llevar la palabra `void` entre los paréntesis, como sí se puede hacer en C++. A diferencia de C++, no se pueden declarar prototipos de métodos para definirlos fuera de la clase. Un método sólo puede ser definido dentro de una clase.

Un método no puede definirse dentro de otro método, ni fuera de una clase, lo que sí puede hacerse en C++.

La Tabla 4 - 4 muestra los modificadores permitidos, fuera de los modificadores de acceso, en Java y C# para los métodos.

Tabla 4 - 4 Modificadores de métodos en Java y C#, fuera de los de acceso

C#	Java	Descripción
----	------	-------------

static	static	Método estático.
abstract	abstract	Método abstracto, esto es, no implementado.
<sin equivalente>	final	Método que no puede ser sobrescrito en una clase derivada.
extern	native	Método implementado externamente en otro lenguaje.
<sin equivalente>	synchronized	Método sincronizado.
virtual	<sin equivalente>	Método que puede ser sobrescrito en una clase derivada.
override	<sin equivalente>	Método que sobrescribe otro, declarado como virtual o abstract, en una clase base.
new	<sin equivalente>	Método que oculta otro en una clase base.
sealed override	<sin equivalente>	Método que sobrescribe otro, declarado como virtual o abstract, en una clase base, evitando también que vuelva a ser sobrescrito en una clase derivada.

A diferencia de C++ y Java, los métodos estáticos en C#, al igual que los datos miembros estáticos, no pueden ser llamados utilizando “this” desde dentro de la clase a la que pertenecen, y sólo pueden ser llamados utilizando el nombre de la clase desde fuera de ésta. En los tres lenguajes, desde un método estático no puede accederse a ningún elemento no-estático de la clase. Un buen ejemplo es el método Main de una clase ejecutable. El siguiente código en C# muestra este caso.

```
using System;
class MetodosEstaticos {
    public static void saludar() {
        Console.WriteLine("Hola");
    }
    public void despedir() {
        Console.WriteLine("Adios");
    }
}
class Principal {
    public static void Main(string[] args) {
        MetodosEstaticos.saludar();
        MetodosEstaticos.despedir();// ERROR
        MetodosEstaticos obj = new MetodosEstaticos();// CORRECCIÓN
        obj.despedir();
    }
}
```

En el código anterior, desde Main sólo puede llamarse directamente a “saludar”, más aún, si la clase “Principal” tuviese otros miembros, sólo podrían accederse desde Main a los estáticos, dado que Main es estático. Para llamar al método “despedir” se requiere contar con una referencia a un objeto “MetodosEstaticos”. Como contraparte, desde un método no-estático, o de instancia, sí se puede acceder a los miembros estáticos de la clase.

En Java y C#, un método abstracto no tiene implementación, dado que se espera que ésta sea dada en las clases que se hereden. Si un método es abstracto, la clase también deberá declararse como abstracta. Esto es similar a los métodos virtuales puros de C++.

En Java, un método con el modificador “final” imposibilita a las clases que heredan de sobrescribirlo. El equivalente más cercano en C# es “sealed override”, con la diferencia que éste no se puede utilizar en la primera implementación del método, sino en una de sus sobrescrituras.

El modificador synchronized es utilizado en la programación concurrente y se verá en capítulos posteriores. El uso del modificador native va más allá de los alcances del curso.

Los modificadores relacionados con el comportamiento polimórfico de un método (new, virtual, abstract, override y sealed) se verán más adelante.

A diferencia de C++, no se pueden asignar valores por defecto a los parámetros.

El Paso de Parámetros

Los lenguajes de programación suelen diferenciar entre dos tipos de pasos de parámetros:

- **Paso por valor:** El parámetro recibe una copia del dato original. Toda modificación al dato del parámetro no modificará al dato original.
- **Paso por referencia:** El parámetro recibe una referencia del dato original. Toda modificación al dato del parámetro modificará al dato original.

En Java, cuando se llama a un método y se pasan parámetros todas las variables de tipo primitivo y objetos son pasadas por valor. Los objetos por ser referencias permiten la modificación de su contenido tal y como ocurre cuando pasamos punteros como parámetros en C++. Como consecuencia de esto, no es posible modificar el valor de una variable primitiva llamando a un método al que se le pase dicha variable como parámetro. Tampoco es posible hacer que una referencia apunte a otro objeto llamando a un método y pasando dicha referencia como parámetro.

En C# se ofrece mayor control en el paso de parámetros que en Java. Por defecto el paso de parámetros es como el indicado en Java. Adicionalmente se puede pasar “por referencia” las variables tipo valor (lo que incluye las variables primitivas) y “por referencia a referencia” las variables tipo referencia. Pasar “por referencia a referencia” es similar a pasar “un puntero a un puntero” en C++, sin sus complicaciones sintácticas. Para esta característica adicional de C# se utilizan las palabras reservadas “ref” y “out”. El siguiente programa muestra el uso de estos tipos de pasos de argumentos en C#.

```
using System;
class Principal {
    public static void mostrar(string mensaje, int[] arreglo, int indice) {
        Console.WriteLine(mensaje);
        Console.WriteLine("  arreglo.Length=" + arreglo.Length);
        Console.WriteLine("  arreglo[0]=" + arreglo[0]);
        Console.WriteLine("  arreglo[1]=" + arreglo[1]);
        Console.WriteLine("  arreglo[2]=" + arreglo[2]);
        Console.WriteLine("  indice=" + indice);
    }
    public static void modificar1(int[] arr, int indice) {
        arr[indice++] += 100;
        arr = new int[10];
    }
    public static void modificar2(ref int[] arr, ref int indice) {
        arr[indice++] += 100;
        arr = new int[10];
    }
    public static void obtener(out string nombre, out string apellido) {
        nombre = "Jose";
```

```
        apellido = "Perez";
    }
    public static void Main(string[] args) {
        int[] arreglo = {1,2,3};
        int indice = 0;
        mostrar("al inicio", arreglo, indice);
        modificar1(arreglo, indice);
        mostrar("luego de llamar a 'modificar1'", arreglo, indice);
        modificar2(ref arreglo, ref indice);
        mostrar("luego de llamar a 'modificar2'", arreglo, indice);
        string nombre, apellido;
        obtener(out nombre, out apellido);
        Console.WriteLine("nombre=" + nombre + ", apellido=" + apellido);
    }
}
```

A diferencia de C++, no es posible ni en Java ni en C# definir valores por defecto para los parámetros.

C# permite declarar una lista indeterminada de parámetros utilizando la palabra reservada “params”. El parámetro declarado con “params” debe ser un arreglo y debe ser el último de la lista de parámetros del método. El siguiente código muestra el uso de params.

```
using System;
class Principal {
    public static void F(params int[] args) {
        Console.WriteLine("args contiene {0} elementos:", args.Length);
        foreach (int i in args)
            Console.WriteLine(" {0}", i);
    }
    public static void Main(string[] args) {
        int[] arr = {1, 2, 3};
        F(arr);
        F(10, 20, 30, 40);
        F();
    }
}
```

Como puede verse en el ejemplo anterior, la llamada a un método con un argumento “params” es más flexible que en C++ con los tres puntos “...”. Se puede pasar tanto un arreglo como parámetro, como una secuencia de parámetros independientes. En este último caso, el lenguaje agrupa dichos argumentos en un arreglo pasándoselos como tal al método.

Las Propiedades y los Indizadores

Es común definir métodos de acceso y obtención (set/get) a los datos encapsulados u ocultos por una clase. Estos métodos permiten contar con bloques de código que pueden controlar que dicho acceso se realice de manera conveniente, por ejemplo, que no se pueda establecer un valor incorrecto a un dato. Como contraparte, estos métodos pueden quitarle una significativa legibilidad al código que los utiliza.

Por ejemplo, supongamos que tenemos una clase que implementa un número complejo, y quisiéramos multiplicar su parte real por dos. Utilizando métodos getReal/setReal tendríamos un código como el siguiente:

```
Complejo c = new Complejo(1,2);
c.set( c.get() * 2);
```

Lo que es significativamente menos legible si tuviésemos acceso directo al dato miembro real y codificáramos:

```
Complejo c = new Complejo(1,2);
c.real = c.real * 2;
```

La ilegibilidad aumenta conforme la expresión se haga más compleja. Más aún, si las clases que utilizamos no guardan un estándar, el programador que hace uso de ellas requerirá encontrar el nombre de dos funciones para manejar el mismo dato.

Como una manera de simplificar el acceso a los datos de una clase, C# define el concepto de “propiedad”. Una propiedad representa a uno o dos métodos de acceso a un dato miembro (o uno calculado) de una clase. La sintaxis de una propiedad es:

```
[modificadores] <tipo de dato> <nombre de la propiedad> {  
    get { <cuerpo del bloque get> }  
    set { <cuerpo del bloque set> }  
}
```

Se puede definir sólo el bloque set (con lo que tendríamos una propiedad de sólo escritura), sólo el bloque get (propiedad de sólo lectura) o ambos bloques (propiedad de lectura y escritura). Como puede verse en la sintaxis, las propiedades pueden recibir modificadores al igual que los datos miembros y los métodos.

Por ejemplo, la siguiente clase en C# implementa el acceso a un archivo sólo para lectura. Debido a esto, el dato del tamaño del archivo nunca se cambiará, por lo que este dato puede implementarse como una propiedad de sólo lectura. Por otra parte, la posición actual de lectura del archivo sí puede desearse que sea leída o modificada por el programa que usa esta clase, por lo que puede implementarse como una propiedad de lectura y escritura.

```
class ArchivoSoloLectura {  
    ...  
    public int Tamano {  
        get {  
            // aquí va el código que permite obtener el tamaño del archivo  
            return TamanoCalculado;  
        }  
    }  
    public int Posicion {  
        set {  
            // asumiendo que el método interno (privado) establecerPosicion  
            // realiza el trabajo de llamar a las librerías del sistema  
            // operativo para recolocar el puntero a un archivo  
            establecerPosicion( value );  
        }  
        get {  
            // aquí iría el código que permitiría calcular la posición actual  
            return PosicionCalculada;  
        }  
    }  
}
```

Note que la palabra reservada “value” es utilizada dentro del bloque “set” para hacer referencia al valor que se le quiere establecer a la propiedad al utilizarla en un programa. El siguiente código hace uso de esta clase.

```
ArchivoSoloLectura arch = new ...;  
Console.WriteLine("El tamaño del archivo es " + Arch.Tamano);  
// Aquí se realizaría algún trabajo de lectura  
Console.WriteLine("La posición actual en el archivo es " + Arch.Posicion);  
Arch.Posicion = 0; // se retorna al inicio del archivo
```

La última línea del programa anterior provoca una llamada al bloque set de la propiedad Posicion, dentro del cual la palabra reservada “value” contendría en valor “0”.

Bajo el mismo concepto de la claridad del código brindado por las propiedades, cuando un objeto representa una colección de elementos (por ejemplo, como una lista enlazada) sería más claro su manejo si fuera posible acceder a estos elementos utilizando la misma sintaxis que con los arreglos. Para esto, C# define los **indizadores**.

Un indizador es como una propiedad pero para manejar un objeto con la misma sintaxis que la de un arreglo unidimensional. La sintaxis de declaración de un indizador es como sigue.

```
[modificadores] <tipo de dato> this[<tipo del indice> <nombre del indice>] {  
    get { <cuerpo del bloque get> }  
    set { <cuerpo del bloque set> }  
}
```

Los indizadores no tienen un nombre particular, como las propiedades, puesto que son llamados automáticamente cuando se utiliza la sintaxis del acceso a los elementos de un arreglo con ellos. El parámetro entre corchetes sirve para declarar el tipo y nombre de la variable que servirá para hacer referencia al índice (o lo que equivalga al mismo) dentro de los bloques set y get. Esto implica que no requerimos necesariamente utilizar un dato de tipo entero como índice, sino cualquier tipo de dato que deseemos.

Como ejemplo, imaginemos que deseamos crear una clase que maneje una lista enlazada de objetos `Nodo`. Adicionalmente, queremos que el programador tenga la posibilidad de recorrer los elementos de esta lista de la misma forma como lo haría con un arreglo, por simplicidad. Tendríamos el siguiente esqueleto del código de esta clase.

```
class ListaEnlazada {  
    // Acá van los datos miembros de la clase y sus métodos correspondientes al  
    // manejo de una lista enlazada. Se asume que la clase "Nodo" ya esta  
    // implementada  
    Nodo primerNodo;  
  
    // Declaración del indizador  
    public Nodo this [ int indice ] {  
        set {  
            Nodo unNodo = primerNodo;  
            for( int i = 0; i < indice; i++)  
                unNodo = unNodo.siguiente();  
            unNodo.valor = value;  
        }  
        get {  
            Nodo unNodo = primerNodo;  
            for( int i = 0; i < indice; i++)  
                unNodo = unNodo.siguiente();  
            return unNodo.valor;  
        }  
    }  
  
    // Declaración de una propiedad  
    public int Tamano {  
        get {  
            // Aquí se recorre la lista contándose los nodos hasta llegar al  
final  
            return TamanoCalculado;  
        }  
    }  
}
```

Note en el ejemplo anterior el uso del parámetro utilizado como índice dentro de los bloques get y set del indizador. Note además que es posible definir propiedades en una clase con un indizador. El siguiente programa hace uso de esta clase.

```
ListaEnlazada lista = ...;  
// Aquí se inicializa la lista con valores para sus nodos  
for( int i = 0; i < lista.Tamano; i++ )  
    // Acá la expresión "lista[i]" equivale a una llamada al bloque "get"  
    // del indizador de la clase ListaEnlazada  
    Console.WriteLine("Valor en la posición " + i + " = " + lista[i]);
```

Note como en el ejemplo anterior se accede a la variable `lista` como si se tratase de un arreglo. Otro ejemplo del uso de indizadores y propiedades es la clase `"string"` de C#, la que implementa

el acceso a los caracteres de la cadena como un indizador de sólo lectura, así como la propiedad de sólo lectura `Length` para obtener su longitud.

Las Estructuras

C++ permite definir tipos de datos “estructura” y “clase”. Las estructuras son herencia de C, donde se utilizaban como forma de crear nuevos tipos de datos compuestos. Sin embargo, la implementación interna de C++ para las estructuras es casi la misma que para las clases, con la única diferencia que por defecto los miembros de una estructura son públicos, por lo que la decisión de mantener la palabra reservada “struct” en C++ responde solo a razones de compatibilidad.

C# también permite definir estructuras como un tipo especial de clases. Las estructuras en C# están diseñadas para ser tipos de datos compuestos más limitados que el resto de clases pero más eficientes que éstas. A continuación se listan las capacidades de las estructuras en C#:

- Sus variables son reservadas automáticamente en memoria de pila y se manejan como tipos de dato valor.
- No permiten la herencia de otras estructuras ni clases, pero sí la implementación de interfaces (lo que se verá más adelante).
- Permiten la definición de métodos, pero no el polimorfismo.
- Permiten la definición de constructores, menos los constructores por defecto, dado que el compilador siempre incluye uno.
- No permiten la inicialización de sus datos miembros de instancia en la misma declaración.
- No se puede utilizar una variable estructura hasta que todos sus campos hayan sido inicializados.

Una variable tipo estructura en C# se reserva automáticamente en pila, mientras que en C++ el programador es quien decide si se almacena en pila o en montón.

El uso de estructuras en C# es recomendado cuando se cumple una o más de las siguientes condiciones:

- Cuando la información que contendrán las estructuras será manipulada como un tipo de dato primitivo.
- Cuando la información que se almacena es pequeña, esto es, menor o igual a 16 bytes.
- Cuando no se requiere utilizar herencia y/o polimorfismo.

La sintaxis de declaración de una estructura en C# es:

```
[modificadores] struct <nombre> [: <lista de interfaces que implementan>]
{
    <datos, tipos, funciones>
}
```

El siguiente código corresponde a un ejemplo de definición y uso de una estructura.

```
using System;
struct Punto {
    public int x, y;
    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

```
    public string Descripcion() {  
        return "(" + x + ", " + y + ")";  
    }  
}  
class PruebaDeEstructuras  
{  
    public static void Main(string[] args)  
    {  
        Punto p1 = new Punto();  
        Punto p2 = new Punto(10, 20);  
        Console.WriteLine("p1 = " + p1.Descripcion());  
        Console.WriteLine("p2 = " + p2.Descripcion());  
        Punto p3;  
        Console.WriteLine("p3 = " + p3.Descripcion()); // ERROR  
        p3.x = 30;  
        p3.y = 40;  
        Console.WriteLine("p3 = " + p3.Descripcion());  
    }  
}
```

Puede observarse en la inicialización de la variable “p1” que, efectivamente, el compilador agrega un constructor por defecto “siempre” a la estructura. Este constructor por defecto inicializa los datos miembros del objeto estructura a sus valores por defecto. La variable “p3” se intenta utilizar sin haber sido inicializada, lo que arroja un error. Enseguida se procede a inicializar individualmente cada dato de la estructura. Sólo cuando todos los datos de la estructura “p3” fueron inicializados es posible utilizar la variable “p3” para otros fines, como por ejemplo llamar a su método “Descripcion”. Note, del uso de la variable “p3”, que no se requiere utilizar el operador new para inicializar una variable estructura, más aún, luego de ejecutar la línea que declara la variable “p3” la memoria de este objeto estructura ya está reservada en pila. Por tanto, el operador new sólo permite llamar a un constructor para inicializar la variable, y no reservarle memoria como sí ocurre con las clases declaradas como “class”.

La Herencia y el Polimorfismo

Cuando un tipo de dato extiende la definición de otro, se dice que el primero hereda del segundo. En esta sección revisaremos el manejo de la herencia en los tres lenguajes estudiados, así como las capacidades que cada uno ofrece en cuanto al manejo del polimorfismo.

La Herencia

Un tipo de dato hereda de otro todos sus miembros: Datos, métodos y otros tipos de datos. El que un tipo de dato herede de otro es una característica que facilita el desarrollo de programas gradualmente más complejos.

A la clase de la que se hereda se le llama clase base o súperclase. A la clase que hereda de otra se le llama clase derivada o subclase. Cuando una clase hereda de más de una base, se le conoce como herencia múltiple. Como veremos más adelante, la herencia múltiple tiene una serie de beneficios y problemas, por lo que no es soportada en algunos lenguajes orientados a objetos.

Esta herencia puede producir diferentes tipos de conflicto:

- Conflicto de espacio de nombres: Dado que los miembros de la clase base pasan a formar parte del espacio de nombres de la clase derivada, existe el problema potencial de que un miembro de una clase base coincida en nombre con uno de la clase derivada.
- Conflicto en la resolución de las llamadas a los métodos: Dado que es posible diferenciar a un método de otro por otros elementos además de su nombre, es posible definir más de un método con el mismo nombre, tanto en la clase base como en la

derivada, por lo que se requiere de una estrategia para determinar a qué método se está llamando realmente dentro del contexto de la llamada. Esta situación se conoce como “polimorfismo”.

- Problemas de duplicidad de datos: Lo que es un problema cuando se tiene herencia múltiple de clases que a su vez heredan, directa o indirectamente, de una misma clase base. En dichos casos, los objetos de la clase más derivada contendrán datos miembros duplicados, uno por cada rama de la herencia.

La Tabla 4 - 5 muestra con qué tipos de datos es posible utilizar herencia y qué tipo de herencia es soportada:

Tabla 4 - 5 Tipos de herencia por lenguaje

	C++	Java	C#
Tipos de datos	Clases y estructuras	Clases e interfaces	Clases e interfaces
Tipo de herencia	Herencia simple y múltiple	Herencia simple de clases y múltiple de interfaces	Herencia simple de clases y múltiple de interfaces

La Declaración de la Herencia

La sintaxis de declaración de la herencia en C++ es:

```
class <nombre>: [mod. de acceso]<clase base1>, [mod. de acceso]<clase base2>, ...  
{ <cuerpo de la clase> };
```

La sintaxis de declaración de la herencia en Java es:

```
class <nombre> extends <clase base> implements <interfaz1>, <interfaz2>, ...  
{ <cuerpo de la clase> }
```

La sintaxis de declaración de la herencia en C# es:

```
class <nombre>: <clase base>, <interfaz1>, <interfaz2>, ...  
{ <cuerpo de la clase> }
```

Los modificadores de acceso permitidos en la declaración de la herencia en C++ son `private`, `protected` y `public`.

El Proceso de Construcción y Destrucción

Cuando una clase hereda de otra, el proceso de construcción y destrucción de un objeto sigue la siguiente lógica:

- Durante la construcción, se llama uno a uno a los constructores, desde las clases base y avanzando por el árbol de herencia hacia las clases más derivadas. Esto permite que el código de construcción de un objeto derivado se ejecute dentro de un contexto en el que se asegure que sus datos heredados están inicializados y en un estado consistente.
- Como contraparte, durante la destrucción, se llama uno a uno a los destructores, desde las clases más derivadas y avanzando por el árbol de herencia hacia las clases base. Esto permite que el código de destrucción de un objeto derivado se ejecute dentro de un contexto en el que se asegure que sus datos heredados aún existen y contienen valores consistentes.

Cuando las clases, en un árbol de herencia, contienen constructores por defecto, la secuencia de llamada durante el proceso de construcción de un objeto se realiza automáticamente. Sin embargo, cuando una clase base no posee un constructor por defecto o se desea ejecutar determinado constructor con parámetros, se puede especificar dicho paso de parámetros desde el constructor de la clase derivada, de la siguiente forma:

C++:

```
class Base {...};
class Derivada: public Base{
    public Derivada(...) : Base(...)
    {...}
};
```

Java:

```
class Base {...}
class Derivada extends Base{
    public Derivada(...)
    { super(...); ... }
}
```

C#:

```
class Base{...}
class Derivada: Base{
    public Derivada(...): base(...)
    {...}
}
```

Note que mientras en C++ se especifica este paso de parámetros, desde el constructor de una clase derivada al de una clase base, con el nombre mismo de la clase base, en Java y C# se utilizan las palabras reservadas “super” y “base” respectivamente. Esto se debe a que C++ soporta herencia múltiple, por lo que podría requerirse pasar parámetros a más de un constructor, mientras que Java y C# soportan sólo herencia simple de clases. La herencia múltiple se verá más adelante.

Note además que la llamada a “super” en Java debe hacerse obligatoriamente en la primera línea del constructor.

Acceso a los Miembros Heredados

En la herencia, la clase derivada puede contener miembros que coincidan en nombre con algunos de la base. Esto no es un error, y es aceptado en los tres lenguajes estudiados. Sin embargo, es necesario contar con un mecanismo para diferenciar, en estos casos, a qué miembro se refiere un pedazo de código, tanto dentro como fuera de la clase.

Dentro de los métodos de la clase derivada, cuando se utiliza directamente el nombre del miembro en conflicto, el compilador asume que nos referimos al de la clase derivada. Para referirnos al de la clase base debemos utilizar una sintaxis especial:

- En C++: <nombre de la clase base> :: <nombre del miembro>
- En Java: **super.**<nombre del miembro>
- En C#: **base.**<nombre del miembro>

Como puede verse, en Java y C# sólo puede accederse al miembro heredado de la clase base. El siguiente ejemplo en Java muestra esta limitación:

```
class Base {
    void m() { System.out.println("Llamada a Base.m"); }
}
```

```
class Derivada1 extends Base {
    void m() { System.out.println("Llamada a Derivada1.m"); }
}
class Derivada2 extends Derivada1 {
    void m() { System.out.println("Llamada a Derivada2.m"); }
    void prueba() {
        m();
        super.m();
        super.super.m(); // ERROR: Llamada inválida
    }
}
public class PruebaDeHerencial {
    public static void main(String[] args) {
        Derivada2 obj = new Derivada2();
        obj.prueba();
        obj.m();
    }
}
```

Como se ve en el ejemplo, no es posible acceder a la implementación del método “m” en “Base” desde un método de la clase “Derivada2”, sólo a la implementación en “Derivada1”, su clase base inmediata. Asimismo, la llamada a “m” desde “main” ejecutará la última implementación de este método, la de la clase “Derivada2”. Como veremos más adelante, esto es un tipo de polimorfismo.

El Polimorfismo

El polimorfismo ocurre cuando la llamada a un método se resuelve sobre la base del contexto de la llamada, dado que existe más de una implementación para dicho método, es decir, la llamada a un método puede tomar distintas formas, según como ésta se realice. Veremos tres casos de polimorfismo:

- La sobrecarga de funciones y métodos.
- La sobrescritura en la herencia.
- La sobrescritura en la implementación de las interfaces.

La Sobrecarga

La forma más simple de polimorfismo es la sobrecarga, en donde dos métodos con el mismo nombre son diferenciados por el lenguaje basándose en sus argumentos.

El siguiente ejemplo muestra la sobrecarga en C#.

```
using System;
struct Punto {
    public int x, y;
    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
class Figura {
    Punto posicion;
    public Figura() : this(new Punto()) {
    }
    public Figura(Punto p) {
        Mover(p);
    }
    public void Mover(int x, int y) {
        Mover(new Punto(x, y));
    }
    public void Mover(Punto p) {
        posicion = p;
    }
}
```

```
class Circulo : Figura {
    double radio;
    public Circulo(double r) : this(new Punto(), r) {
    }
    public Circulo(Punto p, double r) : base(p) {
        radio = r;
    }
    public void Mover(Punto p, double r) {
        Mover(p);
        radio = r;
    }
}
```

En el ejemplo anterior el método “Mover” es sobrecargado tres veces, dos en la clase “Figura” y una en la clase “Circulo”. Además, los constructores de ambas clases están sobrecargados. Cuando se llama al método “Mover” se resuelve dicha llamada sobre la base del tipo de la variable utilizada y los parámetros pasados. El siguiente código utiliza las clases anteriores.

```
Figura f = new Figura();
f.Mover(2,3);
Circulo c = new Circulo(10);
c.Mover(new Punto(7,4), 2);
f.Mover(new Punto(7,4), 2); // ERROR
```

La última línea anterior genera un error debido a que “P” es tipo “Figura” y dentro de esta clase no existe una sobrecarga apropiada para los argumentos pasados en la llamada.

Es importante notar que dos métodos no pueden diferenciarse por su valor de retorno, por lo que dos métodos con el mismo nombre y los mismos argumentos no se podrán diferenciar. La sobrecarga es soportada por C++, C# y Java y no depende de la herencia.

Adicionalmente C++ y C# permiten la sobrecarga de operadores. C++ ofrece una amplia gama de opciones en cuanto al juego de operadores que pueden ser sobrecargados y la forma en que puede declararse dichas sobrecargas. C# por el contrario, ofrece un conjunto restringido de operadores que pueden ser sobrecargados y un único formato de declaración de dicha sobrecarga. Por ejemplo, la sobrecarga del operador de suma en C# tiene el siguiente formato:

```
public static Tipo operator+(Tipo1 op1, Tipo2 op2) { ... }
```

El siguiente código utiliza una sobrecarga del operador “+” para una clase Vector:

```
class Vector {
    private double x, y;
    public Vector(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public static Vector operator+(Vector op1, Vector op2) {
        return new Vector(op1.x + op2.x, op1.y + op2.y);
    }
    public void Imprimir() {
        Console.WriteLine("Vector[x={0}, y={1}]", x, y);
    }
}
class PruebaDeSobrecargaDeOperadores {
    public static void Main(string[] args) {
        Vector a = new Vector(1, 2), b = new Vector(3, 4), c;
        c = a + b;
        c.Imprimir();
    }
}
```

La declaración de una sobrecarga debe ser pública y estática y el tipo del primer parámetro debe coincidir con el tipo de la clase dentro de la que se declara la sobrecarga.

El Ocultamiento

La sobrecarga funciona bien cuando podemos diferenciar dos métodos por sus argumentos, pero es posible tener un método en una clase base con el mismo nombre y los mismos argumentos en la clase derivada. Esto es deseable cuando lo que deseamos conseguir es un ocultamiento de la implementación original de dicho método, de forma que en el código que utilice nuestra clase se llame a esta nueva implementación. El ocultamiento es un tipo de sobrescritura, y es tan eficiente como la sobrecarga.

El siguiente programa en C++ muestra el uso del ocultamiento.

```
class Base {
    public: void Metodo( ) { cout << "Base::Metodo\n"; }
};
class Derivada : public Base {
    public: void Metodo( ) { cout << "Deri::Metodo\n"; }
};
void main() {
    Base ObjBase; Derivada ObjDerivada; Base* pBase;
    ObjBase.Metodo( );
    ObjDerivada.Metodo( );
    pBase = &ObjBase;
    pBase->Metodo( );
    pBase = &ObjDerivada;
    pBase->Metodo( );
}
```

Al ejecutar este programa se ve que las dos primeras llamadas a “Metodo” utilizando las variables “ObjBase” y “ObjDerivada” ejecutan la implementación respectiva en cada clase. En estos casos, el compilador ya no ha podido basarse en los argumentos de la llamada al método para decidir a cuál implementación llamar, sino en el tipo de las variables utilizadas. Las dos últimas llamadas utilizan un puntero del tipo de la clase base “Base” para, apuntando a cada objeto, llamar al “Metodo”. Ambas llamadas se resuelven hacia la implementación en la clase “Base”, dado que, al igual que en las dos primeras llamadas, el compilador se basó en el tipo de la variable utilizada para decidir a qué implementación llamar.

El ejemplo anterior muestra una clara ventaja y limitación del ocultamiento. Si bien el ocultamiento es eficiente dado que la resolución de la llamada se realiza en tiempo de compilación, la capacidad de ocultar es limitada sólo a los casos donde el tipo del objeto creado coincide con el tipo de la variable que lo referencia o apunta. Este es el caso del puntero de tipo “Base” utilizado para llamar al método “Metodo” de un objeto de tipo “Derivada”. Aquí la nueva implementación de “Metodo” no ocultó a la implementación original, lo que probablemente se desea que ocurra.

El siguiente programa corresponde a una versión en C# del programa anterior.

```
using System;
class Base {
    public void Metodo( ) {
        Console.WriteLine("Base.Metodo");
    }
}
class Derivada : Base {
    public new void Metodo( ) {
        Console.WriteLine("Derivada.Metodo");
    }
}
class PruebaDeEstructuras {
    public static void Main(string[] args) {
        Base ObjBase = new Base();
        Derivada ObjDerivada = new Derivada();
        ObjBase.Metodo( );
        ObjDerivada.Metodo( );
    }
}
```



```
Base refBase;  
refBase = ObjBase;  
refBase.Metodo( );  
refBase = ObjDerivada;  
refBase.Metodo( );  
}  
}
```

Otra consecuencia de la forma cómo funciona el ocultamiento es que no se permite que las implementaciones en la clase base utilicen las versiones actualizadas de los métodos ocultos. Los métodos en la base siempre trabajan con “lo conocido” para su clase. El siguiente ejemplo en C++ muestra este caso.

```
class Base {  
public: void Met1() { cout << "Base::Met1\n"; }  
       void Met2() { Met1(); }  
};  
class Derivada : public Base {  
public: void Met1() { cout << "Deri::Met1\n"; }  
};  
void main() {  
    Base ObjB; ObjB.Met2();  
    Derivada ObjD; ObjD.Met2();  
}
```

En el ejemplo mostrado, el ocultamiento de “Met1” no es aprovechado por “Met2”, por lo que el texto mostrado será “Base::Met1”, cuando lo que quizá se deseaba era “Deri::Met1”.

Como puede observarse, la implementación del método “Metodo” en la clase “Derivada” requiere el uso de la palabra reservada new. C# busca con esto que sea claro que dicho método está ocultando a otro en alguna de las bases del árbol de herencia.

En resumen, el ocultamiento es eficiente, dado que se resuelve en tiempo de compilación, pero puede originar llamadas incorrectas a métodos sobrescritos cuando los objetos son tratados con variables cuyo tipo corresponda a alguna de las clases base del objeto. Para solucionar esto, el programa debería analizar en tiempo de ejecución, y ya no durante la compilación, a qué objeto verdaderamente apunta o referencia una variable, y sobre la base de esto decidir a qué método llamar. Esto es lo que realiza la sobrescritura virtual.

La Sobrescritura Virtual

Cuando un método se sobrescribe virtualmente el compilador agrega información adicional a la clase, la que es utilizada en tiempo de ejecución para determinar a que tipo de objeto se está apuntando realmente y por tanto, a qué método se debe de llamar.

Las siguientes sintaxis corresponden a la manera cómo declarar una sobrescritura virtual, en cada lenguaje:

- C++: Se coloca “virtual” en la base.

```
class Base { ... virtual void Met( ) {...} ... };
```

- Java: No existe ocultamiento. El polimorfismo siempre es mediante sobrescritura virtual.
- C#: Se coloca “virtual” en la base y “override” en la derivada.

```
class Base { ... virtual void Met( ) {...} ... };  
class Deri : Base { ... override void Met( ) {...} ... };
```

En resumen, la sobrescritura virtual permite que desde cualquier método de la clase base, de la clase derivada o desde fuera de ellas, una llamada a un método se resuelva sobre la base del tipo

del objeto apuntado o referenciado, no el de la variable que lo apunta o referencia. Como es de suponerse, la sobrescritura virtual es menos eficiente que el ocultamiento.

La Implementación de Interfaces

En general se suele decir que la interfaz de una clase se refiere al conjunto de métodos que ésta expone públicamente, para el manejo de los objetos de la misma.

El concepto de interfaz no está relacionado a la implementación de los métodos de una clase, sólo a la declaración de los mismos. De esta forma, se puede encontrar que varias clases, no relacionadas en una herencia, pueden exponer interfaces similares. Por tanto, si se pudiera tipificar y homogenizar los elementos similares de las interfaces de varias clases, podrían manejarse sus objetos de manera homogénea, aún cuando no pertenezcan a una misma herencia.

Esta tipificación de la interfaz de una clase se realiza en Java y C# mediante la palabra reservada “interface”. Las sintaxis correspondientes a la declaración de una interfaz en Java y C# son:

En Java:

```
[modificadores] interface <nombre> [ extends <lista de interfaces> ]  
{ <declaración de los métodos> }
```

En C#:

```
[modificadores] interface <nombre> [ : <lista de interfaces> ]  
{ <declaración de los métodos> }
```

Las interfaces soportan la herencia múltiple, debido a que no puede existir conflicto de implementación (dado que nada se implementa) ni tampoco duplicidad de datos (dado que no se permiten declarar datos). En C++ no existe el concepto de interfaz, pero puede ser simulado mediante una clase que contenga sólo métodos virtuales puros públicos.

Una interfaz declara, no implementa, un conjunto de métodos que corresponden a una funcionalidad que una clase puede exponer. Por ejemplo, la interfaz “dibujable” puede ser implementada tanto por una clase “marquesina” como una clase “fotografía”. Por tanto, las interfaces agrupan a un conjunto de métodos públicos que pueden ser implementados por una clase.

El siguiente programa Java hace uso de una interfaz.

```
interface Dibujable {  
    void Dibujar( );  
}  
class Imagen implements Dibujable {  
    public void Dibujar( ) { System.out.println("Llamando a Dibujar en Imagen"); }  
}  
class Texto implements Dibujable {  
    public void Dibujar( ) { System.out.println("Llamando a Dibujar en Texto"); }  
}  
public class PruebaDeInterfaces {  
    public static void main(String[] args) {  
        Imagen img = new Imagen();  
        Texto txt = new Texto();  
        Dibujable dib = img;  
        dib.Dibujar();  
        dib = txt;  
        dib.Dibujar();  
    }  
}
```

El mismo programa en C# sería:

```
using System;
```

```
interface Dibujable {
    void Dibujar( );
}
class Imagen : Dibujable {
    public void Dibujar( ) { Console.WriteLine("Llamando a Dibujar en Imagen"); }
}
class Texto : Dibujable {
    public void Dibujar( ) { Console.WriteLine("Llamando a Dibujar en Texto"); }
}
public class PruebaDeInterfaces {
    public static void Main() {
        Imagen img = new Imagen();
        Texto txt = new Texto();
        Dibujable dib = img;
        dib.Dibujar();
        dib = txt;
        dib.Dibujar();
    }
}
```

El mismo programa en C++ sería:

```
class Dibujable {
public: virtual void Dibujar( ) = 0;
};
class Imagen: public Dibujable {
public: virtual void Dibujar( ) { cout << "Llamando a Dibujar en Imagen"; }
};
class Texto: public Dibujable {
public: virtual void Dibujar( ) { cout << "Llamando a Dibujar en Texto"; }
};
void main() {
    Imagen* img = new Imagen();
    Texto* txt = new Texto();
    Dibujable* dib = img;
    dib->Dibujar();
    dib = txt;
    dib->Dibujar();
}
```

Una clase puede implementar más de una interfaz y en el caso de C#, una estructura también. Como puede verse, el concepto de herencia múltiple de interfaces en Java y C# reemplaza al de herencia múltiple de clases en C++, evitando los problemas que ésta última tiene.

El siguiente código de programa en C# muestra una herencia múltiple de interfaces.

```
interface IArchivo {
    void posicion( );
}
interface IArchivoBinario : IArchivo {
    byte leerByte( );
    void escribirByte(byte b);
}
interface IArchivoTexto : IArchivo {
    char leerChar( );
    void escribirChar(char b);
}
class ArchivoBinario : IArchivoBinario { ... }
class ArchivoTexto : IArchivoTexto { ... }
class ArchivoBinarioTexto : IArchivoBinario, IArchivoTexto { ... }
```

Aún cuando no se declare como públicos los métodos de una interfaz, éstos siempre lo son. Por tanto, las implementaciones de estos métodos en las clases deberán ser declaradas como públicas.

Las Clases Abstractas

Cuando una clase no implementa un método se dice que ésta es abstracta. Dado que las clases abstractas son clases “a medio implementar”, éstas no pueden utilizarse para instanciar objetos, pero sí como clases base de otras clases que sí implementen dichos métodos faltantes.

Declaración de una Clase Abstracta

En C++ las clases que contienen métodos virtuales puros son implícitamente abstractas. En Java y C#, las clases que sólo declaren algunos métodos, sin implementarlos, deben ser declaradas explícitamente mediante la palabra reservada “abstract” como un modificador más en la declaración de la clase.

Es importante recalcar que una clase abstracta se diferencia de una interfaz en que puede contener métodos no abstractos y por tanto, funcionalidad implementada.

El siguiente programa en C# muestra el uso de una clase abstracta.

```
using System;
abstract class Base{
    public void imprimir() {Console.WriteLine("Imprimir: Clase Base");}
    abstract public void metodo();
}
class Derivada : Base{
    new public void imprimir(){Console.WriteLine("Imprimir: Clase Derivada");}
    override public void metodo(){Console.WriteLine("Metodo: Clase Derivada");}
}
class MainClass{
    public static void Main(string[] args){
        Base objBase ;//= new Base(); // si se descomenta, habria un error
                                // pues no se puede instanciar la clase
                                // por ser abstracta

        Derivada objDerivada = new Derivada();
        objbase = objDerivada;
        objBase.imprimir(); objBase.metodo();
        objDerivada.imprimir(); objDerivada.metodo();
    }
}
```

Note que en C# el método abstracto en la clase base debe de declararse explícitamente como “abstract”, lo que no requiere Java. Note también que la implementación del método en la clase derivada se declara como “override”, debido a que los métodos abstractos son implícitamente virtuales. Como en Java todo polimorfismo es virtual, un equivalente de este programa en Java no requeriría utilizar ninguna palabra reservada especial en la declaración de la implementación en la clase derivada.

Todo método declarado dentro de una interfaz en Java y C# es implícitamente abstracto, virtual y público.

Java permite la declaración de constantes dentro de sus interfaces. C# permite declarar propiedades e indizadores. La siguiente interfaz en C# declara una propiedad de sólo lectura y un indizador de lectura y escritura.

```
class UnaInterface {
    int UnaPropiedad { get; }
    int this [ int indice ] { get; set; }
}
```

Diferencias entre las Interfaces y las Clases Abstractas

Las clases abstractas son adecuadas cuando se desea tener una implementación básica común para clases derivadas que deberán implementar la funcionalidad faltante y posiblemente,

sobrescribir parte de la básica. Por otro lado, las interfaces son adecuadas cuando no se requiere contar con una implementación básica común, dado que toda se realizará en las clases que las implementen. La tabla 4.6 resume estas diferencias.

Tabla 4 - 6 Diferencias entre las clases abstractas y las interfaces.

	Interfaz	Clase abstracta
Declara métodos abstractos	Sí	Sí
Implementar métodos	No	Sí
Añadir datos miembros	No	Sí
Crear objetos	No	No
Crear arreglos y referencias	Sí	Sí

Los Tipos de Datos Anidados

Hasta el momento sólo se han estudiado los datos y las funciones como miembros de una clase, pero una clase también puede contener la definición de tipos como miembros. A un tipo de dato declarado dentro de otro tipo de dato se le conoce como tipo de dato anidado o “inner”, y al tipo que lo contiene “outer”.

En esta sección nos concentraremos en la declaración y uso de las clases anidadas o clases inner.

La Declaración de Clases Anidadas

Una clase anidada o inner se declara como cualquier otra clase, con la diferencia que, dado que son miembros de una clase, pueden aplicárseles modificadores propios de los miembros de éstas. Por ejemplo, es posible declarar una clase inner como privada, por lo que sólo se podrán declarar variables de ésta dentro de la clase outer y ser manipulados desde los métodos de ésta.

Se puede tener varios niveles de anidación en la declaración de clases, es decir, una clase inner puede ser anidar otras clases inner. Una clase que no es inner de ninguna otra se le llama clase de alto nivel o “top-level”.

Las clases anidadas son útiles cuando:

- Los objetos de dichas clases sólo van a ser utilizados dentro de su clase outer. Estas clases inner funcionarían como una librería interna de la clase outer. Sin embargo, si es posible que estas clases inner sean reutilizables fuera de la clase outer, es recomendable sacarlas de la clase outer y definir las en un espacio de nombres independiente. La definición y uso de los espacios de nombres se verá más adelante.
- Los objetos de dichas clases requieren trabajar intensamente con los datos de la clase outer. Dado que la clase inner está definido dentro del ámbito de la clase outer, tiene acceso directo a todos los miembros de ésta, aún a los privados, lo cual puede ser beneficioso si es que el trabajo entre ambas clases es muy intenso.

- Cuando carezca de sentido crear objetos de esta clase sin que existan previamente objetos de su clase outer. En este caso, se requiere de un objeto de la clase outer para poder crear, sobre la base de su información interna y otros datos externos, objetos de la clase inner.
- Cuando la clase outer le da sentido a la inner. En este caso, la clase outer funciona como un espacio de nombres. Al igual que en el primer caso, si no se cumpliera esta condición se recomienda utilizar clases en un espacio de nombres independiente en lugar de clases inner.

El siguiente programa en Java muestra el uso de una clase inner.

```
class A {  
    class B {  
        public void metodo1() {  
            System.out.println("Llamada a B.metodo1");  
            metodo2();  
        }  
    }  
    private void metodo2() {  
        System.out.println("Llamada a A.metodo2");  
    }  
    public void metodo3() {  
        System.out.println("Llamada a A.metodo3");  
        B objB = new B();  
        objB.metodo1();  
    }  
}  
public class PruebaDeInterfaces {  
    public static void main(String[] args) {  
        A objA = new A();  
        objA.metodo3();  
    }  
}
```

En el ejemplo anterior, la clase B es inner de la clase A, por lo que puede acceder a todos los miembros de ésta última, inclusive los privados, como la llamada al método “metodo2” desde su implementación de “metodo1”. Dado que la clase B es un miembro más de la clase A, al no habersele especificado un modificador de acceso posee el modificador de-paquete. Por tanto, es posible acceder desde “main” a dicha clase, declarar variables y crear objetos con ella.

La Creación de Objetos de Clases Anidadas

En el ejemplo anterior se creó un objeto de la clase B dentro del “metodo3” de la misma forma como se crean objetos de clases no inner. Sin embargo, para crear objetos inner fuera de su clase outer la sintaxis es diferente. El siguiente código puede haberse incluido en “main” para crear un objeto de la clase B:

```
A.B objB = objA.new B();  
objB.metodo1();
```

Es interesante notar como el nombre de la clase outer forma parte del nombre de la clase inner, es decir, fuera de la clase A, el nombre completo de la clase B es “A.B”. El uso de la palabra reservada “new” también requiere una sintaxis especial, dado que los objetos de la clase B son considerados “de instancia”, esto es, se requiere utilizar una instancia de la clase A para crear una de la B. Piense en esto: Si la instancia del objeto referenciado por “objB” no hubiera sido creada haciendo referencia a un objeto de la clase A, ¿al método de qué objeto se estaría accediendo en la llamada que hace “metodo1” a “metodo2”, dado que éste último es igualmente un método de instancia? Luego, cuando se crea un objeto de una clase inner de instancia, dicho objeto conserva una referencia al objeto outer en base al que fue creado, de forma que pueda acceder

tanto a sus miembros estáticos como no-estáticos. Sin embargo, si esta característica no se deseara, la clase B pudo declararse como estática, es decir:

```
class A {
    static class B {
        public void metodo1() { ... }
    }
    private static void metodo2() { ... }
    public void metodo3() { ... }
}
public class PruebaDeInterfaces {
    public static void main(String[] args) {
        A objA = new A();
        objA.metodo3();
        A.B objB = new A.B();
        objB.metodo1();
    }
}
```

En este caso, los métodos de la clase B sólo pueden acceder a los miembros estáticos de la clase A, dado que no son creados con referencia a un objeto de A, por lo que “metodo2” requiere ser estático para poder ser llamado desde “metodo2”. La sintaxis de creación de un objeto B desde fuera de la clase A, en “main”, también cambia.

En el caso de C++ y C#, los objetos de las clases inner no conservan automáticamente una referencia a un objeto de la clase outer, por lo que se les puede considerar clases inner estáticas por defecto, por lo que no requieren ni permiten su declaración utilizando el modificador “static”.

Finalmente las clases inner pueden declararse con los mismos modificadores de acceso de los demás miembros. En el ejemplo anterior, si la clase “B” hubiera sido declarada como “private”, no hubiera podido crearse ni declarar objetos de ésta fuera de los métodos de la clase “A”.

Los Conflictos de Nombres

Cuando un miembro de la clase inner coincide en nombre con uno de la clase outer, es necesario utilizar una sintaxis especial que permite resolver a qué miembro se está llamando. La siguiente sintaxis corresponde a la forma de referirse a un miembro de una clase outer desde dentro de una clase inner:

C++:

```
<nombre de la clase outer>::<nombre del miembro>
```

Java: Desde una inner de instancia:

```
<nombre de la clase outer>.this.<nombre del miembro>
```

Java: Desde una inner estática:

```
<nombre de la clase outer>.<nombre del miembro>
```

C#:

```
<nombre de la clase outer>.<nombre del miembro>
```

El siguiente programa en Java muestra este caso para una clase inner de instancia.

```
class Outer {
    class Inner {
        void m() {
            System.out.println("Llamada a Inner.m");
            Outer.this.m();
        }
    }
}
```

```
void m() {
    System.out.println("Llamada a Outer.m");
}
void prueba() {
    Inner objInner = new Inner();
    objInner.m();
}
}
public class PruebaDeInterfaces {
    public static void main(String[] args) {
        Outer objOuter = new Outer();
        objOuter.prueba();
    }
}
```

Note en el ejemplo anterior, que la expresión “Outer.this” es la forma en que se referencia al objeto de la clase outer en base al que se creó el objeto de la clase inner.

Las Clases Anidadas Anónimas

Java ofrece la capacidad de crear clases anidadas anónimas. Estas clases anónimas se instancian en la misma expresión que las define. El siguiente programa en Java muestra el uso de este tipo de clase.

```
class Base {
    public void Imprimir() { System.out.println("Base.Imprimir"); }
}
class Principal {
    public static void main(String args[]) {
        Base obj = new Base() { // Aquí comienza la definición de la clase anónima
            public void Imprimir() {
                System.out.println(this.getClass().getName()+"Imprimir");
            }
        }; // Aquí termina la definición de la clase anónima
        obj.Imprimir();
    }
}
```

En el programa anterior, la clase inner anónima está definida entre los dos corchetes que siguen a la expresión de creación “new Base()”. Esta clase anónima hereda de la clase “Base” y su nombre no es requerido, debido a que el programa sólo requiere crear un objeto de dicha clase. El objeto de esta clase anónima es manejado siempre utilizando una variable del tipo de la clase base. El siguiente ejemplo en Java define una clase anónima que implementa una interfaz.

```
interface UnaInterface { void Imprimir(); }
class Principal {
    public static void main(String args[]) {
        UnaInterface obj = new UnaInterface() {
            public void Imprimir() {
                System.out.println(this.getClass().getName()+"Imprimir");
            }
        };
        obj.Imprimir();
    }
}
```

El código del método “Imprimir” muestra en consola el nombre interno que el compilador de Java le asigna a la clase anónima definida, en este caso “Principal\$1”. Si se revisa el directorio donde se generan los archivos compilados de Java para este programa se verá el archivo “Principal\$1.class” correspondiente a esta clase inner.

Las clases anónimas son útiles cuando:

- Sólo se desea crear objetos de esta clase en una sola parte del programa, para extender una clase base o implementar una interfaz.

- La implementación de la clase anónima es relativamente pequeña.

La Reflexión

Cuando se trabaja con árboles de clases con métodos polimórficos o con interfaces es común tratar los objetos con referencias a las clases base o al tipo de las interfaces. A la asignación de un objeto de una clase derivada a una variable de una clase base se le conoce como “up-cast” o “widening”. Este tipo de asignaciones no requiere de una operación de cast explícita y es segura, dado que un objeto de un tipo derivado “siempre” pueden tratarse como un objeto de un tipo base. La operación contraria no es segura, es decir, asignar una referencia desde una variable de un tipo base a una de un tipo derivado “no siempre” es correcta, dado que siempre es posible que se esté referenciando a un objeto que no puede ser tratado como el tipo de la variable destino. Este tipo de operación requiere de un cast explícito, y se conoce como “down-cast” o “narrowing”. El siguiente código es un ejemplo de esto:

```
ClaseBase ref1 = new ClaseDerivada( ); // Up-cast o widening, siempre es seguro
... // otras líneas de código

ClaseDerivada ref2 = ref1; // Down-cast o narrowing, error de compilación,
                          // se requiere una operación "cast"
ClaseDerivada ref3 = (ClaseDerivada)ref1; // Esto si compila
```

En el código anterior, la última línea puede ocasionar un error al ser ejecutada, si la variable “ref1” referenciara a un objeto de la ClaseBase, producto de una operación de asignación ejecutada en alguna de las “otras líneas de código”.

Dado que este tipo de operaciones cast son requeridas en algunas ocasiones, suele ser necesario contar con algún mecanismo para poder verificar si el objeto referenciado o apuntado por una variable puede o no ser tratado como un tipo determinado. Éste es un ejemplo en donde la técnica conocida como “reflexión” es útil.

Definición y Uso

Los lenguajes de programación que implementan la reflexión guardan información adicional en los objetos que son creados durante la ejecución de un programa de forma que, en tiempo de ejecución, sea posible obtener información sobre el tipo con que fue creado dicho objeto. La identificación de tipos en tiempo de ejecución (o RTTI por sus siglas en inglés) es sólo una de las capacidades que ofrece la reflexión. La reflexión permite conocer el árbol de herencia de un tipo de dato, los modificadores con que fue definido, los miembros que incluye así como información sobre cada uno de estos miembros, entre otros datos.

Algunos ejemplos del uso de la reflexión son:

- Verificación de errores en tiempo de ejecución. Al realizar un cast de una referencia de un tipo base a un tipo derivado, la reflexión puede servir para averiguar si dicho objeto puede ser o no interpretado como tal o cual tipo, es decir, si su clase es o hereda del tipo destino. Si el objeto no puede ser tratado como del tipo destino, se produce un error que puede ser interceptado y tratado por el programa.
- Entornos de desarrollo con componentes. Estos entornos requieren exponer al programador los elementos de dichos componentes, así como su descripción, tipo, parámetros, etc. Se pueden instalar nuevos componentes y el sistema deberá poder reconocerlos automáticamente, siempre que cumplan con el estándar pedido por el entorno, esto es, que dichos componentes expongan la interfaz requerida.

- Programas orientados a dar servicios. Estos programas trabajan con otros programas que deben cumplir con ofrecer cierta interfaz. El programa servidor determina, en tiempo de ejecución, si otro programa cumple la interfaz necesaria para dar cierto servicio y si lo hace, puede trabajar con él. Un ejemplo de un programa de servicio es la misma arquitectura de un sistema operativo, donde los programas ejecutables y las librerías deben exponer cierta interfaz para que el sistema operativo pueda ejecutarlos. Otro ejemplo son los servidores Web y en general, cualquier servidor distribuido.

La reflexión no es la única técnica posible para éstos y otros casos donde un sistema requiera conocimiento de si mismo y de su entorno para adaptarse, pero ofrece la ventaja de ser simple y uniforme.

El tema de la reflexión es amplio, por lo que la siguiente sección sólo abarca lo que corresponde a RTTI.

RTTI

Uno de los aspectos de la reflexión es la identificación de tipos en tiempo de ejecución o RTTI. Esta consiste en determinar si un objeto puede ser manejado como un tipo de dato determinado.

C++ expone una implementación limitada de RTTI y requiere que los programas que la utilizan sean compilados con opciones especiales del compilador. El siguiente programa en C++ utiliza esta técnica.

```
#include <iostream.h>
#include <typeinfo.h>
class Base {
    public: virtual void Imprimir() { cout << "Base::Imprimir" << endl; }
};
class Derivada : public Base {
    public: virtual void Imprimir() { cout << "Derivada::Imprimir" << endl; }
};
void Identificar(Base* pObj) {
    Derivada* pDer = dynamic cast<Derivada*>(pObj);
    if(pDer != 0)
        cout << "Objeto 'Derivada'" << endl;
    else
        cout << "Objeto 'Base'" << endl;
    const type_info& ti = typeid(*pObj);
    cout << "typeid: name=" << ti.name() << ", raw_name=" << ti.raw_name() << endl;
}

void main() {
    Base* pBase = new Base();
    Identificar(pBase);
    Derivada* pDer = new Derivada();
    Identificar(pDer);
}
```

El operador “typeid” retorna una referencia de tipo “const type_info &”, que es un objeto que contiene información acerca del tipo de objeto pasado al operador.

El operador “dynamic_cast” permite obtener un puntero de un tipo derivado, pasándole como parámetro un tipo base y el puntero original. Sin embargo, requiere que el tipo base contenga por lo menos un método “virtual”. Si el puntero pasado como parámetro no apunta a un objeto que puede interpretarse como el tipo indicado entre los símbolos “< >”, el operador retorna cero “0”.

Java utiliza el operador “instanceof” para determinar si el objeto referenciado por una variable puede o no ser manejado como un tipo de dato determinado. El siguiente programa muestra su uso.

```
class Base { }
class Derivada extends Base { }

class PruebaDeInstanceof {
    static void Identificar( Base obj ) {
        if ( obj instanceof Derivada ) System.out.println("Objeto 'Derivada'");
        else System.out.println("Objeto 'Base'");
    }
    public static void main( String args[] ) {
        Identificar( new Base( ) );
        Identificar( new Derivada( ) );
    }
}
```

Si un objeto puede ser tratado como un tipo de dato determinado, la asignación de ésta, utilizando una operación de cast explícita, a una variable del tipo destino es segura y no arrojará error durante su ejecución.

C# utiliza el operador “is” en lugar del operador “instanceof” de Java. El siguiente programa muestra su uso.

```
using System;
class Base { }
class Derivada : Base { }
class PruebaDeIs {
    static void Identificar(Base obj) {
        if ( obj is Derivada ) Console.WriteLine("Objeto 'Derivada'");
        else Console.WriteLine("Objeto 'Base'");
    }
    public static void Main() {
        Identificar(new Base());
        Identificar(new Derivada());
    }
}
```


Espacios de Nombres y Librerías

En este capítulo revisaremos la organización del código a un nivel distinto y complementario al propuesto por la POO, las librerías, así como las ventajas y problemas que estas conllevan.

Los Espacios de Nombres

Cuando conversamos con un grupo reducido de amigos es común que utilicemos sólo sus nombres, o apodos, para referirnos a cada uno de ellos. En estos casos es poco común que dos amigos tengan el mismo nombre, pero cuando esto ocurre utilizamos modificaciones de los nombres para aclarar a quién nos estamos refiriendo. Sin embargo, cuando el grupo al que nos dirigimos crece, el número de personas con el mismo nombre tiende a aumentar y con ello la dificultad de utilizar nombres cortos. En un entorno más formal se tiende a utilizar los nombres y los apellidos para aclarar a quién nos referimos en un momento dado. Aún así, siempre existe la posibilidad de encontrar homónimos.

Un problema similar ocurre cuando programamos. Por ejemplo, si dentro de un método definimos una variable con el mismo nombre que otra definida en ámbito de clase, la primera oculta a la segunda. Por tanto, cuando deseamos referirnos a la segunda requerimos utilizar un discriminante, esto es, una sintaxis especial que permita indicarle al compilador a quién nos queremos referir. Desde el punto de vista de los nombres que se le dan a los tipos de datos ocurre lo mismo, sobre todo si utilizamos librerías implementadas por terceros.

Este problema es atacado mediante la definición de espacios de nombres. Un espacio de nombres es un ámbito donde es posible declarar elementos de programación, como variables, métodos y tipos de datos, y dentro del cual se utilizan nombres únicos para cada uno de estos elementos.

Los ámbitos de una función, un método y una clase definen implícitamente un espacio de nombres. Pero es posible definir espacios de nombres explícitos adicionales que permitan organizar nuestros elementos de programación en una estructura jerárquica donde cada elemento tenga un nombre único.

Definición de un Espacio de Nombres

C++ y C# permiten definir espacios de nombres explícitos mediante la palabra reservada “namespace”. La sintaxis básica en ambos lenguajes es:

```
namespace <nombre> {  
    <declaración de elementos de programación>  
}
```

Java por su lado, une el concepto de espacio de nombres con el de librería, por lo que los detalles acerca de su definición los veremos más adelante, en la sección de librerías.

El siguiente ejemplo corresponde al uso de un espacio de nombres en C++:

```
namespace general {  
    int a, b;  
    void fn() {  
        a = 20;  
        b = a * 2;  
    }  
}  
void main() {  
    general::a = 10;  
    general::b = general::a;  
}
```

Nótese que dentro del namespace “general” es posible acceder directamente a todos los elementos declarados en él, esto es, utilizar su nombre corto. Fuera del namespace “general” requerimos utilizar su nombre completo o largo. El nombre del namespace forma parte del nombre de los elementos declarados dentro de él.

El siguiente ejemplo corresponde al uso de un espacio de nombres en C#:

```
namespace General {  
    class A {}  
    class B {}  
}  
  
public class PruebaDeNamespace {  
    public static void Main() {  
        General.A objA = new General.A();  
        General.A objB = new General.A();  
    }  
}
```

Nótese que mientras en C++ se pueden definir variables, funciones o tipos de datos en un espacio de nombres en C++, en C# sólo pueden definirse tipos de datos y como se verá más adelante, otros espacios de nombres.

Anidamiento de Espacios de Nombres

Ahora bien, los elementos no declarados dentro de un namespace explícito, como la función “main” del ejemplo anterior, forman parte del namespace global, el cuál es anónimo. Más aún, el namespace “general” forma parte de éste, lo que implica que los namespaces pueden anidarse. El siguiente ejemplo en C++ muestra la anidación de espacios de nombres y cómo hacer referencia a los elementos del namespace global de manera explícita.

```
#include <iostream.h>  
  
namespace Espacio1 {  
    namespace Espacio2 {  
        const int var = 10;  
    }  
    int var = Espacio2::var * 2;  
}  
float var = Espacio1::Espacio2::var + Espacio1::var;  
namespace Espacio3 {  
    double var = ::var * 2;  
}  
  
void main() {  
    int var = (int)Espacio3::var + 100;
```

```
// imprimimos todas las variables "var"
cout << "Espacio1::Espacio2::var = " << Espacio1::Espacio2::var << endl;
cout << "Espacio1::var = " << Espacio1::var << endl;
cout << "::var = " << ::var << endl;
cout << "Espacio3::var = " << Espacio3::var << endl;
cout << "var = " << var << endl;
}
```

Nótese cómo, conforme se sale del namespace “Espacio1”, se requiere utilizar de un nombre cada vez más largo para acceder a sus elementos desde fuera de él. Nótese además como los nombres de los elementos son cada vez más largos conforme la anidación de los namespace crece en profundidad.

El siguiente ejemplo corresponde al uso del anidamiento de espacios de nombres en C#:

```
using System;
namespace Espacio1 {
    namespace Espacio2 {
        class A {}
    }
    class A {}
}
class A {}
public class PruebaDeNamespace {
    public static void Main() {
        Espacio1.Espacio2.A obj1 = new Espacio1.Espacio2.A();
        Espacio1.A obj2 = new Espacio1.A();
        A obj3 = new A();
    }
}
```

Nótese que en el espacio de nombres global están definidas dos clases, A y PruebaDeNamespace, y un espacio de nombres, Espacio1.

Publicación de un Espacio de Nombres

Para hacer referencia a un elemento de un namespace desde fuera del mismo, utilizando su nombre corto, se utiliza la palabra reservada “using”. Esta palabra permite hacer públicos los elementos declarados dentro de un namespace, en otro namespace. Se pueden publicar elementos individuales o todos los elementos a la vez. El siguiente ejemplo en C++ muestra el uso de “using”.

```
#include <iostream.h>

namespace Espacio1 {
    int var1 = 10;
    int var2 = 20;
}

namespace Espacio2 {
    int var3 = 30;
    int var4 = 40;
}

// Se publica todo el namespace Espacio1 en el espacio de nombres global
using namespace Espacio1;
// Se publica solo var3 de Espacio2 en el espacio de nombres global
using Espacio2::var3;

void main() {
    cout << "var1 = " << var1 << endl;
    cout << "var2 = " << var2 << endl;
    cout << "var3 = " << var3 << endl;

    cout << "var4 = " << var4 << endl; // ERROR: variable no reconocida
    // en el presente espacio de nombres
    cout << "Espacio2::var4 = " << Espacio2::var4 << endl; // Corrección
}
```


El siguiente ejemplo en C# muestra el uso de “using”.

```
using System;
using Espacio1.Espacio2;

namespace Espacio1 {
    namespace Espacio2 {
        class A {}
        class B {}
    }
}

public class PruebaDeNamespace {
    public static void Main() {
        A obj1 = new A();
        B obj2 = new B();
    }
}
```

En el caso de C#, la directiva “using” sólo puede utilizarse antes de la declaración de cualquier espacio de nombres. Note que “System” es realmente un espacio de nombres, no una librería. Dentro de System están definidos todos los tipos de datos primitivos, tipos para el manejo de la entrada y salida estándar como la clase Console, entre otros tipos y espacios de nombres que forman la librería estándar de .NET.

Uso de un Alias

También se puede declarar un alias para hacer referencia a un namespace dentro de otro namespace. El siguiente programa en C++ muestra un ejemplo de esto.

```
#include <iostream.h>

namespace Espacio {
    const double var = 3.1416;
}

namespace EspacioX = Espacio; // EspacioX es el nuevo alias de Espacio

void main() {
    cout << "EspacioX::var = " << EspacioX::var << endl;
}
```

El siguiente programa en C# muestra un ejemplo de esto.

```
using System;
using UnAlias = Espacio1.Espacio2;

namespace Espacio1 {
    namespace Espacio2 {
        class A {}
        class B {}
    }
}

public class PruebaDeNamespace {
    public static void Main() {
        UnAlias.A obj1 = new UnAlias.A();
        UnAlias.B obj2 = new UnAlias.B();
    }
}
```

Definición por Partes

Los espacios de nombres no requieren ser declarados en un solo bloque, éstos pueden declararse en varios bloques independientes, incluso en bloques en archivos y librerías distintas. El siguiente programa en C++ muestra esta definición por partes.

```
#include <iostream.h>

namespace Espacio {
    int var1 = 10;
}

namespace Espacio {
    int var2 = 20;
}

int main(void) {
    cout << "Espacio::var1 = " << Espacio::var1 << endl;
    cout << "Espacio::var2 = " << Espacio::var2 << endl;
    cin.ignore();
    return 0;
}
```

Se dice que el segundo bloque namespace “Espacio” extiende el espacio de nombres definido por el primer bloque namespace “Espacio”. Nótese que “var1” y “var2” pertenecen al mismo espacio de nombres, aún cuando son declarados en bloques distintos. Los bloques pudieron incluso haberse colocado en archivos distintos. Si esto fuese así, el compilador iría completando el espacio de nombres conforme fuera compilando los archivos fuente de un proyecto.

El siguiente programa en C# muestra esta definición por partes.

```
namespace Espacio1 {
    namespace Espacio2 {
        class A {}
    }
}

namespace Espacio1.Espacio2 {
    class B {}
}

public class PruebaDeNamespace {
    public static void Main() {
        Espacio1.Espacio2.A obj1 = new Espacio1.Espacio2.A();
        Espacio1.Espacio2.B obj2 = new Espacio1.Espacio2.B();
    }
}
```

En el ejemplo anterior, las clases “A” y “B” forman parte del espacio de nombres “Espacio1.Espacio2”.

Las Librerías

En el contexto de los lenguajes de programación, una librería es una unidad de agrupación de los elementos de programación en una forma tal que puede ser distribuido para su reutilización desde otros programas. Los elementos del lenguaje que pueden formar parte de una librería, así como la forma de crearla y utilizarla dependen de cada lenguaje.

Las siguientes secciones describen el enfoque utilizado y la creación de librerías en C/C++, Java y C#.

Librerías en C/C++

En C/C++ se definen dos tipos de librerías:

- Las librerías estáticas.
- Las librerías dinámicas.

Las librerías **estáticas** (comúnmente, con extensión LIB) son archivos con código máquina ya enlazado (similar a los archivos OBJ, solo que estos últimos no están enlazados) que se utilizan durante el proceso de enlace de archivos OBJ's para formar un programa final, ya sea un

ejecutable u otra librería. En este sentido, utilizar librerías estáticas significa tener los compilados de un conjunto de archivos fuente, C y CPP, con lo que se ahorra tiempo al momento de compilar un programa que las utilice, dado que para dichos archivos ya no se requiere pasar por un proceso de compilación. Si una librería es extensa, este tiempo de compilación ahorrado puede ser significativo.

Las librerías **dinámicas** (comúnmente, con extensión DLL, abreviatura de Dinamic Link Library) son archivos con código máquina ya enlazado y datos que permiten a otro programa, en tiempo de ejecución, obtener la ubicación de sus funciones, para llamarlas. Los programas que hacen uso de estas librerías deben seguir un proceso, asistido por el sistema operativo, para cargar dicho código a memoria, buscar la ubicación en memoria de las funciones y llamar a éstas. La DLL no forma parte del archivo del programa que las utiliza. Si dos o más programas en ejecución solicitan una DLL, ésta se carga una sola vez a memoria, en la primera solicitud, con lo que se ahorra espacio de memoria.

Dado que las librerías no se ejecutan directamente sino a través de otros programas que llaman a sus funciones, no requieren implementar una función de entrada, como una función main o WinMain.

Cuando una función de una librería es declarada de manera tal que ésta pueda ser llamada desde otro programa, se dice que dicha función es exportada por la librería. Sólo las funciones exportadas por una librería pueden ser accedidas desde un programa que utilice dicha librería.

Las librerías estáticas son más sencillas de utilizar que las dinámicas. En contraparte, las librerías dinámicas permiten evitar la duplicidad de código. En consecuencia, se suele utilizar librerías estáticas cuando:

- Estas son significativamente pequeñas respecto a la cantidad de memoria disponible en las computadoras donde correrán los programas que las usen.
- Son pocos los programas, instalados en una misma computadora, que las utilizan.

En el resto de casos, se prefiere utilizar librerías dinámicas.

Las librerías pueden ser utilizadas tanto por programas ejecutables, como por otras librerías. Una librería puede contener, además de las funciones exportadas y no-exportadas, recursos como imágenes, audio, video, textos, etc.

Librerías Estáticas

La creación y manejo de una librería estática es sencillo:

- Se compila como librería estática los archivos fuente que la conforman. Si se está trabajando con un IDE, comúnmente se deberá crear el proyecto del tipo “para creación de una librería estática”. Por ejemplo, en Microsoft Visual C++ 6.0, el tipo de proyecto es “Win32 Static Library”.
- El proceso de compilación generará el archivo de la librería, con extensión LIB.
- Se incluye el archivo LIB dentro del proyecto del programa que utilizará esta librería. Si se está utilizando un IDE, comúnmente basta con agregar el LIB como un archivo más del proyecto, al igual que los archivos fuente.

Librerías Dinámicas

Las DLL son librerías a las que los programas acceden en tiempo de ejecución. Dado que estas librerías no forman parte de dichos programas, tampoco son cargadas a memoria

automáticamente al ejecutarse éstos. Por ello, si un programa requiere ejecutar una función exportada por una DLL, deberá solicitar al sistema operativo que cargue el archivo DLL a memoria, averigüe la dirección de la función exportada de interés y llamarla. Al proceso de averiguar la ubicación de una función para luego llamarla, se le conoce como **enlace dinámico**, de allí el nombre de este tipo de librería.

Las DLL no tienen un punto de entrada para un hilo primario, como sucede con los archivos ejecutables (función main o WinMain), debido a que el sistema operativo no crea un hilo de ejecución para ellas. Son los hilos de los procesos que hacen uso de una librería, los que ejecutan el código de ésta.

Windows está formado en gran parte por librerías dinámicas, desde donde se comparte la funcionalidad que otras aplicaciones necesitan importar para interactuar con el sistema operativo. Como ejemplo tenemos algunas de las DLL típicamente utilizados por las aplicaciones de Windows:

- KRNL386.EXE { Nótese que es un ejecutable }
- GDI.EXE
- USER.EXE
- KEYBOARD.DRV
- SOUND.DRV
- Winmm.dll
- Msvcrt40.dll, Msvcrt20.dll, Msvcrt.dll

Aunque es un uso poco frecuente, un archivo ejecutable también puede ser utilizado como una DLL, siempre que éste contenga funciones exportadas y datos que permitan a otros programas ubicarlas. Más adelante veremos cuáles son estos datos y cómo se crean.

Las DLL y los procesos que las llaman se cargan a memoria en tiempos y lugares distintos, por lo que originalmente tienen espacios de direccionamiento distintos. Sin embargo, el sistema operativo “mapea” las llamadas a las funciones de las DLL’s dentro de los espacios de direccionamiento de los procesos de los hilos llamadores. Esto significa que la dirección de la función exportada obtenida por el hilo llamador corresponde a un valor dentro del espacio de direccionamiento de su proceso, pero cuando es utilizada para llamar a la función, el sistema operativo “mapea” dicha dirección de forma que se acceda a la ubicación real de dicha función y se pueda ejecutar su código. Por tanto, nunca se rompe la regla de que “los hilos de un proceso no pueden acceder a direcciones en memoria fuera de su espacio de direccionamiento”.

Estructura Interna

Cuando un conjunto de archivos fuente es compilado y enlazado como una DLL, al archivo resultante se le agrega al inicio, una tabla de exportación. Esta tabla contiene los datos que permiten a los programas que hacen uso de una DLL, obtener la ubicación de una función exportada. La estructura de esta tabla es, de manera simplificada, la siguiente:

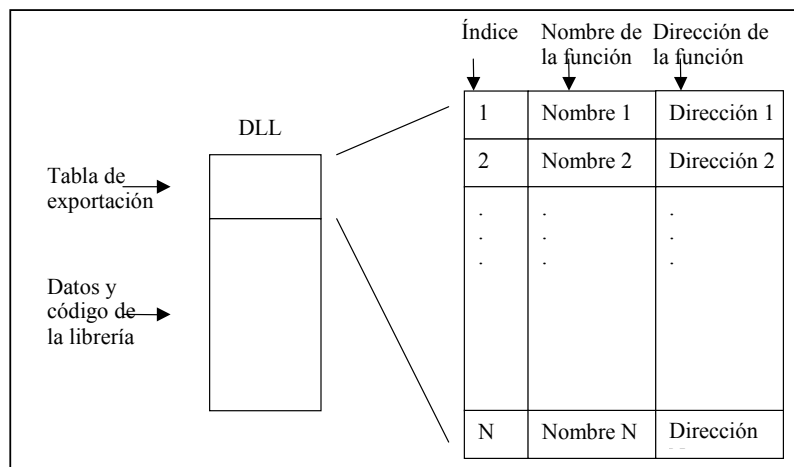


Figura 5 - 1 Estructura interna de una DLL.

Cuando un programa carga a memoria una DLL, mediante funciones del API de Windows, obtiene un handle a dicha librería. Mediante este handle el programa puede obtener la dirección mapeada de cualquiera de las funciones exportadas por la DLL, igualmente, mediante funciones del API de Windows.

Estas funciones del API de Windows realizan una búsqueda sobre la tabla de exportación, según los parámetros que se le pasen, bien por el nombre de la función o directamente utilizando un índice. Si se encuentra la función, se retorna su dirección mapeada. La búsqueda por nombre es más lenta que utilizando un índice, pero ofrece la ventaja de asegurar al programa usuario que la dirección devuelta corresponde a la función correcta. Como contraparte, la búsqueda por índice es más rápida pero, dado que la posición en la tabla de exportación para los datos de una función exportada puede variar entre versiones de una misma DLL, la dirección retornada puede no ser de la función buscada.

Todas las funciones dentro de una DLL que son declaradas para exportación, estarán incluidas en esta tabla. El resto de funciones son privadas de la librería, pero pueden ser llamadas desde las funciones exportadas.

Espacio de Direccionamiento

Cada proceso activo en el sistema tiene un espacio de direccionamiento virtual privado. El espacio de direccionamiento es un rango de direcciones virtuales, lo cual significa que dos procesos podrían tener punteros con el mismo valor pero estar realmente apuntando a lugares de memoria diferentes. La dirección a la que realmente se apunta es determinada por el sistema operativo mediante tablas de direccionamiento. Este esquema de trabajo permite al sistema operativo sacar de memoria (copiando sus datos a disco) procesos, o parte de ellos, que no se estén ejecutando para colocar, en la misma ubicación de memoria real, otros procesos que deban ejecutarse. De esta forma, el sistema puede simular que se está trabajando con una memoria mucho mayor de la que realmente existe. A la técnica de bajar y subir a memoria bloques de datos de procesos se le llama SWAPING. Al archivo en disco que se utiliza para esto se le llama archivo de SWAP.

En conclusión, el sistema puede trabajar con tantos procesos a la vez como espacio de memoria tenga sumando la memoria RAM más el espacio disponible en disco para el archivo de SWAP.

La técnica de mapeado de las direcciones de las funciones exportadas por las DLL's cargadas a memoria, permite no violar la regla de que los hilos de un proceso no pueden acceder a direcciones fuera del espacio de direccionamiento de su proceso.

Creación de una DLL

Se deben seguir los siguientes pasos:

7. Se crea un nuevo proyecto del tipo "quiero crear una DLL". Por ejemplo, en Microsoft Visual C++ 6.0, el tipo del proyecto es "Win32 Dynamic-Link Library".
8. Se agrega los archivos y el código necesario.
9. Se compila y se genera el archivo de la librería con extensión DLL y un archivo para enlace en modo implícito con extensión LIB. El uso de éste último se verá más adelante.

Una DLL comúnmente contiene los siguientes archivos:

1. **Un archivo cabecera** (*.H) donde se declaran las funciones exportadas.
2. **Los archivos fuentes** (*.C o *.CPP) y otros archivos de cabecera. Una de las fuentes deberá implementar la función DllMain. Las fuentes deberán implementar las funciones exportadas, junto con el resto de funciones utilizadas por éstas.
3. **Un archivo de definición** (*.DEF) donde se declaren qué funciones serán las que se exporten.

Como ejemplo, se explicarán los archivos correspondientes a una DLL que sólo exporte una función. Llamaremos a esta DLL "MiLibreria.DLL".

EL ARCHIVO CABECERA MILIBRERIA.H

Contiene los prototipos de las funciones a exportar. Los prototipos deben especificar además el tipo de convención de llamada que se usará. Cada convención de llamada provoca una decoración particular del nombre de cada función, de forma que quién llame a dicha función sepa distinguir a qué convención se refiere. Nosotros podemos dejar que el compilador coloque dichos adornos internamente (al momento de compilar) por nosotros o colocar dichas decoraciones manualmente.

La convención de llamada de una función determina la forma en que el código máquina que se genere, al momento de compilar las fuentes, realice la llamada a dichas funciones.

Si una función exportada por una DLL tiene una determinada convención de llamada, el programa que utilice dicha función deberá declararla con la misma convención de llamada. Luego, la manera más sencilla de salvar estos problemas es dejar que el compilador decida la convención de llamada por nosotros, tanto cuando se compila el DLL como el programa que lo utilizará.

Sin embargo, el compilador de C++ agrega adornos adicionales al nombre de las funciones exportadas, lo que no realiza el compilador de C. La manera más sencilla de solucionar este problema, si el programa y la DLL son compilados con compiladores distintos, es forzar a que las funciones de exportación sean declaradas de una misma forma, por ejemplo, de C. Para hacer esto, se debe declarar los prototipos de estas funciones, tanto en el programa como en el DLL, dentro de la sentencia:

```
extern "C" {  
    // Aquí se colocan los prototipos de las funciones exportadas  
}
```

Todo lo que esté dentro de los corchetes se compila con el compilador de C. Si el fichero es agregado a un archivo *.CPP, este código se compila igualmente como C, y el resto del archivo fuente se compila con el compilador de C++.

Como ejemplo, se desea declarar el archivo cabecera de la DLL para exportar una función que muestre un mensaje de saludo. El archivo contendrá:

```
////////////////////  
// Archivo MiLibreria.h  
////////////////////  
  
#include <windows.h>  
  
#ifdef __cplusplus  
extern "C" {  
#endif  
  
void WINAPI Saludame(char * szNombre);  
  
#ifdef __cplusplus  
}  
#endif
```

Las directivas `#ifdef` y `#endif` permiten que el código entre ellas sea tomado en cuenta por el compilador únicamente si éste es el compilador de C++. La macro `WINAPI` permite especificar la convención de llamada de la función a la del estándar utilizado por las librerías dinámicas de Windows.

EL ARCHIVO FUENTE MILIBRERIA.CPP

Contiene la implementación de las funciones cuyos prototipos hemos declarado en el archivo cabecera. Adicionalmente, este archivo debe contener la definición de la función `DllMain`. Esta función es llamada cada vez que el sistema operativo recibe una solicitud de carga o descarga de dicha DLL por parte de algún proceso. Esta función no es llamada nunca directamente por el proceso. Su prototipo y estructura típica es:

```
BOOL WINAPI DllMain ( HANDLE hModule, DWORD dwReason, LPVOID lpReserved ) {  
    switch ( dwReason ) {  
        case DLL_PROCESS_ATTACH:  
            // Código ejecutado la primera vez que un hilo de un proceso carga la DLL  
            break;  
        case DLL_THREAD_ATTACH:  
            // Código ejecutado las siguientes veces que un hilo de un proceso carga la DLL  
            break;  
        case DLL_THREAD_DETACH:  
            // Código ejecutado cuando un hilo de un proceso descarga la DLL  
            break;  
        case DLL_PROCESS_DETACH:  
            // Código ejecutado cuando el último hilo de un proceso descarga la DLL  
            // En este caso, al retornar de esta función el sistema descarga de memoria la DLL  
            break;  
    }  
    return TRUE;  
}
```

Donde:

- `hModule` : Es el handle para la instancia del DLL cargada en memoria.
- `dwReason` : Es la razón por la que se llama a la función.
- `LPVOID lpReserved` : Esta reservado para uso del sistema.

Si la función no se define, el entorno de Visual C++ agrega a nuestro código compilado la declaración de una función `DllMain` por defecto. Si la función devuelve `FALSE` significa que la

carga / descarga falló, por lo que la función, en el hilo del proceso desde dónde se llamó a la carga / descarga de la librería, recibirá un valor de error como resultado.

El resto del archivo fuente deberá contener la implementación de las funciones exportadas así como otras de uso interno. Para nuestro ejemplo, el código sería:

```
////////////////////////////////////
// Archivo MiLibreria.cpp
////////////////////////////////////

#include <windows.h>

BOOL WINAPI DllMain ( HANDLE hModule, DWORD dwReason, LPVOID lpReserved ) {
    // Aquí va lo indicado arriba
}

#include "MiLibreria.h"

void WINAPI Saludame(char * szNombre) {
    MessageBox(NULL, szNombre, "Hola", MB_OK);
}
```

EL ARCHIVO DE DEFINICIÓN MILIBRERIA.DEF

Existen 3 métodos para exportar una definición:

- Utilizar la palabra-clave `__declspec(dllexport)` en la declaración del prototipo de la función a exportar.
- Utilizar la sentencia `EXPORTS` en un archivo con extensión `DEF`.
- Utilizar la opción `/EXPORT` como parte de la información pasada al enlazador.

Usaremos el archivo con extensión `DEF`. Éste es un archivo de texto con un conjunto de sentencias:

- `NAME`
- `LIBRARY`
- `DESCRIPTION`
- `STACKSIZE`
- `SECTIONS`
- `EXPORTS`
- `VERSION`

La sentencia `EXPORTS` es la única obligatoria y marca el inicio de una lista de definiciones de exportación. Cada definición tiene el siguiente formato:

```
entryname[=internalname] [@ordinal[NONAME]] [DATA] [PRIVATE]
```

Para nuestro ejemplo, el archivo `DEF` contendrá:

```
////////////////////////////////////
// Archivo MyDll.def
// Este comentario no debe incluirlo en el archivo DEF
////////////////////////////////////

LIBRARY SALUDAMEDLL
DESCRIPTION "Implementación de un saludo."
EXPORTS
    Saludame @1
```


La inclusión de un archivo DEF en el proyecto también permite la creación, por parte de IDE, de un archivo LIB, el cual podrá agregarse a los demás proyectos C/C++ desde donde deseamos usar la librería enlazándola en modo implícito.

Cuando se utiliza la palabra-clave `__declspec(dllexport)`, ésta es la que le indica al IDE la creación del archivo LIB. Si para el ejemplo anterior deseáramos no utilizar un DEF, tendríamos que declarar el prototipo de la función a exportar de la forma:

```
__declspec(dllexport) void WINAPI Saludame(char * szNombre);
```

Utilización de una DLL

Para poder llamar a una DLL, el programa llamador debe enlazarse con ella. En este contexto, la palabra “enlazarse” significa “cargar a memoria la DLL y obtener las direcciones mapeadas de sus funciones exportadas. Dado que el “enlace” se realiza en tiempo de ejecución, se le llama dinámico, de allí el nombre de este tipo de librerías.

En enlace dinámico puede realizarse de dos formas:

- De modo implícito.
- De modo explícito.

EL MODO IMPLÍCITO

El enlace en modo implícito se consigue utilizando el archivo LIB generado cuando se compiló la DLL. Existen además programas utilitarios que permiten obtener un LIB directamente de un DLL ya generado, como el programa `implib.exe` de Borland. Este archivo LIB contiene código compilado que es agregado a nuestro programa. Dicho código se ejecutará al momento de iniciar el programa, realizando el enlace dinámico por nosotros. Adicionalmente, este código es sumamente eficiente y nos permite utilizar los prototipos de las funciones como si éstas fueran codificadas dentro de nuestro programa al momento de compilarlo.

El archivo LIB se utiliza durante el proceso de enlace del programa que hará uso de la librería. Si se usa un IDE, se tendrá que configurar el proyecto para que utilice los LIB's de las DLL's que deseamos utilizar. Como ejemplo, para agregar un archivo LIB a un proyecto en Microsoft Visual C++ 6.0, se colocan el nombre de éste en:

```
Project => Settings => Link => Category:General => Object/Library modules:
```

Luego, dentro del código de nuestro programa llamador podemos realizar la llamada a la función exportada de la siguiente manera:

```
#include <windows.h>
#include "..\MiLibreria\MiLibreria.h"

int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hInstancePrev, LPSTR lpCmdLine, int
nCmdShow ) {
    Saludame("JUAN");
    return 0;
}
```

Nótese que se utiliza el mismo archivo de cabecera de la DLL, `MiLibreria.h`.

EL MODO EXPLÍCITO

El enlace en modo explícito no requiere el uso del archivo LIB. Esto es útil cuando no disponemos de éste o cuando sabemos el nombre del DLL sólo después que la aplicación se está ejecutando, por ejemplo, siendo ingresado por el usuario.

Para este caso, necesitamos realizar los siguientes pasos:

- Solicitarle al sistema operativo que cargue la librería a memoria: **LoadLibrary**.
- Obtener la dirección de la función: **GetProcAddress**
- Solicitarle al sistema operativo que descargue la librería: **FreeLibrary**

Los prototipos de las funciones del API de Windows indicadas son:

```
HMODULE LoadLibrary( // HMODULE es un typedef de HINSTANCE
LPCTSTR lpzModuleName // Nombre del archivo DLL
);

BOOL FreeLibrary(
HINSTANCE hInstLib // Handle a la DLL, devuelta por AfxLoadLibrary
);

FARPROC GetProcAddress(
HMODULE hModule, // Handle a la DLL
LPCSTR lpProcName // Nombre de la función
);
```

Se debe de usar la macro MAKEINTRESOURCE para el parámetro lpProcName cuando se desea obtener la dirección de la función exportada en base a su índice en la tabla de exportación.

Además debemos declarar una variable de tipo “puntero a función” para cada función que deseemos utilizar de dicha librería. Para nuestro ejemplo, la declaración del puntero a función sería:

```
typedef void (WINAPI * PFUNC) (char *);
```

Luego de definir el tipo podemos declarar una variable de dicho tipo:

```
PFUNC pfnSaludo;
```

La que utilizaremos para ejecutar la función en la DLL. Nuestro código de ejemplo quedaría de la siguiente forma:

```
typedef void (WINAPI * PFUNC) (char *);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hInsPrev, LPSTR lpCLine, int nCShow) {
    PFUNC pfnSaludo;
    HINSTANCE hDll;

    hDll = LoadLibrary("MyDll.dll");
    if ( hDll != NULL ) {
        pfnSaludo = (PFUNC)GetProcAddress( hDll, "Saludame" );
        if( pfnSaludo != NULL ) {
            // cualquiera de los dos formatos siguientes es válido
            ( *pfnSaludo ) ( "OTTO" );
            pfnSaludo ( "OTTO" );
        }
        FreeLibrary( hDll );
    }
    return 0;
}
```

Las funciones de carga y descarga lo que hacen es manejar un contador, mantenido por el sistema operativo, del uso de una DLL. Cuando ese contador regresa a cero (una carga lo aumenta en uno, una descarga lo baja en uno) la librería es descargada de memoria dado que ya nadie la está utilizando.

Mecanismo de Búsqueda de una DLL

El enlace estático utiliza el siguiente mecanismo de búsqueda en directorios para encontrar el archivo de la DLL:

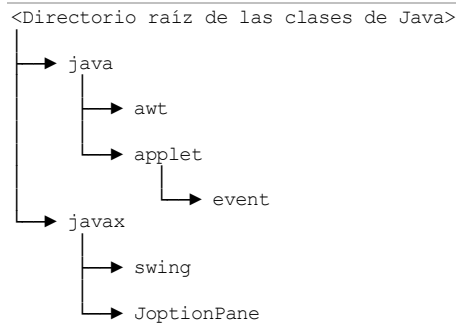
4. En el directorio donde se encuentra el ejecutable de la aplicación.

5. En el directorio de trabajo actual
6. En el directorio System. Si es NT o Windows 2000, en el directorio System32.
7. En el directorio de Windows
8. En la lista de directorios de la variable PATH

De igual forma, si en el enlace dinámico no se especifica una ruta en el nombre del archivo pasado a la función `AfxLoadLibrary`, ésta utilizará el mismo mecanismo de búsqueda anterior.

Librerías en Java

Las librerías en Java se conocen con el nombre de paquetes. Un paquete Java es realmente un directorio en algún parte de nuestro disco. Todos los archivos CLASS bajo un mismo directorio pertenecen a un mismo paquete. Los sub-directorios dentro del directorio de un paquete corresponden a otros paquetes que generalmente están relacionados a éste. Por ejemplo, la siguiente estructura de directorios corresponde a parte de la librería estándar de Java.



La estructura indica que el paquete (directorio) “java” contiene otros 2 paquetes: “awt” y “applet”. El paquete (directorio) “applet” contiene a su vez al paquete “event”. De igual forma, el paquete (directorio) “javax” contiene al paquete “swing” el cual contiene la clase “JOptionPane”.

Uso de un Paquete

El siguiente programa hace uso de la clase `JOptionPane`.

```
public class PruebaDeInterfaces {
    public static void main(String[] args) {
        javax.swing.JOptionPane.showMessageDialog(null, "Hola",
            "Un Mensaje", JOptionPane.ERROR_MESSAGE);
        System.exit(0);
    }
}
```

Nótese que el nombre completo de una clase indica el directorio donde se encuentra el archivo CLASS de ésta. Nótese además como Java une el concepto de espacio de nombres con el de librería. Un paquete define un espacio de nombres y todo espacio de nombres es un paquete.

También es posible publicar el contenido de un paquete, en otro paquete, utilizando la palabra reservada “import”, de forma que se puedan utilizar los nombres cortos de los elementos del paquete. El siguiente programa modifica el anterior utilizando “import”.

```
import javax.swing.*;
public class PruebaDeInterfaces {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, "Hola", "Un Mensaje",
            JOptionPane.ERROR_MESSAGE);
        System.exit(0);
    }
}
```

```
}  
}
```

El uso de “import” es equivalente al uso de “using” en C++ y C#. Adicionalmente Java permite publicar elementos individuales de un paquete dentro de otro paquete. La primera línea del programa anterior pudo haberse escrito de la siguiente manera:

```
import javax.swing.JOptionPane;
```

Sin embargo, si se utilizan muchos elementos definidos dentro de un paquete, resulta más conveniente publicar todo el paquete en lugar de publicar cada elemento individualmente.

Java no permite la definición de alias para los paquetes.

Ubicación de un Paquete

El directorio raíz bajo el que se encuentran todos los paquetes (directorios) de Java instalados en una máquina se indica mediante una variable de entorno guardada en algún archivo de configuración del sistema operativo. En el caso de Windows, esta variable de entorno se llama CLASSPATH. Para Windows 95/98/Millennium, ésta y otras variables de entorno se suelen colocar dentro del archivo autoexec.bat, donde CLASSPATH es inicializado de la siguiente forma:

```
set CLASSPATH = < rutas iniciales separadas por ";">
```

Por ejemplo, supongamos que el directorio raíz estuviese especificado de la siguiente forma.

```
set CLASSPATH = C:\CLASES
```

Supongamos además que incluimos la instrucción siguiente en un programa:

```
import java.awt.Graphics;
```

La instrucción anterior le dirá al compilador que la clase Graphics la podrá encontrar en la ruta

```
C:\CLASES\JAVA\AWT
```

Java es sensitivo a las diferencias entre letras capitales y no-capitales, por tanto las siguientes importaciones son consideradas distintas por el compilador:

```
import java.awt.Graphics;  
import java.awt.graphics;
```

Nótese que en los programas en Java se hace uso de algunas clases sin haber especificado su paquete, como la clase String y la clase System. Estas clases pertenecen al paquete “java.lang”. Este paquete corresponde a la librería básica de java y contiene definiciones que son parte del mismo lenguaje, siendo importado automáticamente por el compilador por lo que no es necesario declarar una sentencia “import” para dicho paquete.

Creación de un Paquete

Para definir clases que formen parte de un paquete se requiere utilizar la directiva “package” al inicio del archivo donde se declaran estas clases. El siguiente programa define la clase A y B como parte del paquete P.

```
package P;  
public class A {}  
class B {}
```

Al compilar el archivo “A.java” conteniendo el código anterior, se generarán los archivos “A.class” y “B.class” correspondientes a las clases A y B respectivamente. La clase A es pública, por lo que podrá ser utilizada por otros paquetes fuera de “P”, mientras que la clase B tiene el

modificador de acceso por defecto “de-paquete”, por lo que sólo podrá ser utilizada desde otras clases que pertenezcan al paquete “P”. Se pueden crear otros archivos que definan clases para el paquete “P”, por lo que un paquete puede ser definido por partes, y no necesariamente en un solo archivo.

Cuando un archivo de Java no declara la directiva “package”, las clases que define pertenecen al paquete global, el cual coincide con el directorio actual de ejecución del programa.

Utilización de un Nuevo Paquete

Para utilizar un paquete nuevo se requiere crear una estructura de directorios que coincida con la del nuevo paquete. Por ejemplo, si se tiene un paquete A con las clase A1, A2 y A3, y el subpaquete B con las clases B1 y B2, se requeriría crear un directorio A, colocar dentro los archivos A1.class, A2.class y A3.class, crear dentro el subdirectorio B y colocar dentro los archivos B1.class y B2.class.

La nueva estructura de directorios, para ser reconocida por el programa que requiere utilizarla, puede ir:

- Bajo el mismo directorio de los archivos “class” que lo utilizan.
- Bajo cualquier otro directorio en el computador, agregando dicho directorio a la lista de rutas especificadas en la variable de entorno utilizada por el compilador y el intérprete, CLASSPATH en el caso de Windows.

Si se revisa el contenido por defecto de la variable de entorno de ubicación de paquetes encontraremos que lo que indican es la ubicación de uno o más archivos con extensión “jar”. Ésta es una forma alternativa de distribuir un paquete.

Los archivos JAR empaquetan, de la misma forma que un archivo de compresión como el ZIP, toda una estructura de directorios que forman un paquete, junto con los archivos incluidos en éstos. De esta forma, cuando el compilador o intérprete de Java busca una clase y encuentra la especificación de archivos con extensión JAR, continúa la búsqueda dentro de éstos de manera similar a como lo haría en un directorio.

La ventaja de utilizar archivos JAR es el ahorro de espacio y facilita la distribución de los paquetes, dado que sólo se requiere copiar un archivo y no crear toda una estructura de directorios en el computador donde se desea utilizar un paquete.

Para generar un archivo JAR se puede utilizar el programa utilitario, distribuido junto con el JDK, “jar.exe”. La siguiente sintaxis corresponde a la llamada a este programa:

```
jar {ctxu}[vfmOMi] [archivo-jar] [archivo-manifest] [-C dir] archivos
```

El detalle de lo que cada opción significa se puede obtener ejecutando el programa sin ningún parámetro. El siguiente comando es un ejemplo de creación de un paquete MiPaquete.jar con las clases contenidas en el directorio DirectorioRaiz:

```
jar cvf MiPaquete.jar -C DirectorioRaiz / .
```

Por ejemplo, asumiendo:

```
C:\trab\repositorio.class  
C:\trab\miembros\GestorUsuario.class  
C:\trab\miembros\Usuario.class
```

Con una ventana de comandos con directorio actual “c:\trab”, ejecutar:

```
jar.exe      -cvf      Miembros.jar      repositorio.class      Miembros/GestorUsuario.class  
Miembros/Usuario.class
```

```
java.exe -cp Miembros.jar;. repositorio
```

Librerías en C#

Las DLL's presentan una carencia en su concepción: El sistema operativo no registra automáticamente que programas hacen uso de una DLL. Algunos casos típicos de problemas originados por esta carencia son:

- Si al desinstalar un programa, éste elimina una DLL que es utilizada por otro programa, éste último dejará de funcionar correctamente.
- Si un usuario cambia de posición el archivo de la DLL utilizada por un programa, éste quizá no lo encuentre, por lo que dejará de funcionar correctamente.

Adicionalmente, dado que las DLL's se ubican en el sistema de archivos, en directorios específicos, si se intenta copiar una nueva DLL, utilizada por ejemplo por un programa nuevo, y su nombre coincide con otra preexistente, se reemplazará el archivo de la DLL antigua. Por lo tanto, todos los programas que utilizaban la DLL reemplazada dejarán de funcionar correctamente. Aún en el caso que dicho reemplazo sea intencional, por ejemplo al actualizar la versión de una DLL, si la verificación de la versión de la DLL preexistente versus la nueva no se realiza correctamente, es posible que se reemplace una DLL más reciente con una más antigua. Incluso aún en el caso que el reemplazo sea realizado con una correcta verificación de las versiones, siempre es posible que un error de programación en la nueva versión haga que un programa que funcionaba correctamente con la antigua, deje de funcionar con la nueva.

Todos estos problemas son demasiado comunes, por lo que .NET desarrolla un nuevo concepto orientado a darles solución: Los ensamblajes.

Los Ensamblajes

Un ensamblaje es una unidad de instalación autodescriptiva. Todos los archivos generados en .NET son parte de algún ensamblaje. Por ejemplo, los ejecutables (*.EXE) son ensamblajes.

Un ensamblaje puede estar formado por uno o más archivos, los que en conjunto contienen los siguientes elementos:

- Metadata del ensamblaje
- Metadata de tipos
- Código MSIL
- Recursos

Los diagramas de la Figura 5 - 2 muestran dos ejemplos de distribución de estos elementos en los archivos de un ensamblaje. El primero, Ensamblaje1.dll, es un ensamblaje tipo librería (esto es, no existe un punto de entrada o método Main desde donde ejecutar un hilo primario) formado por un único archivo. El segundo es un ensamblaje tipo ejecutable (si existe un Main) formado por tres archivos. La Metadata del Ensamblaje del archivo Ensamblaje2.exe guarda la descripción exacta de los archivos que forman el ensamblaje. Fuera de la Metadata del Ensamblaje, el resto de elementos pueden existir en cada uno de los archivos que forman un ensamblaje.

Un ensamblaje puede estar formado por los siguientes tipos de archivos:

- Un archivo principal, EXE o DLL, donde se encuentra la Metadata del Ensamblaje, entre otros elementos.

- Cero, uno o más archivos de módulo de .NET, con extensión NETMODULE. Los módulos de .NET no contienen Metadata del Ensamblaje.
- Cero, uno o más archivos de recursos, como archivos de imagen, sonido, video, etc.

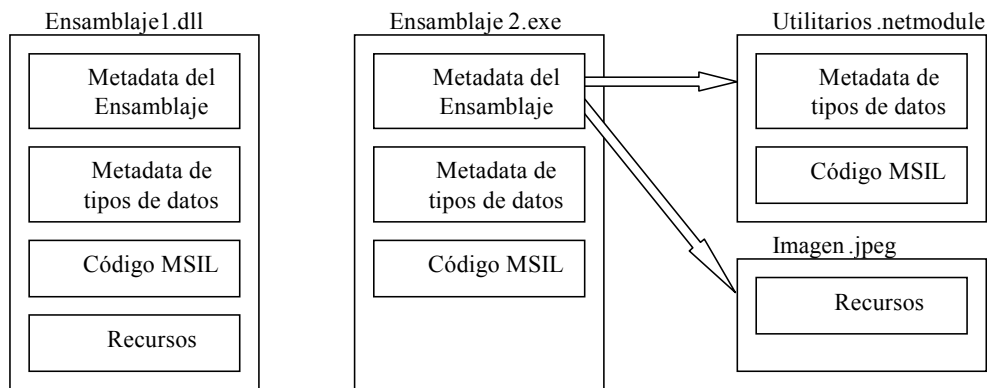


Figura 5 - 2 Ejemplo de distribución de elementos en un ensamblaje

Si bien el Ensamblaje2 de la figura es formado por tres archivos, es posible compilar el archivo principal, Ensamblaje2.exe, de forma que los demás archivos se incluyan dentro.

Las características más importantes de un ensamblaje son:

- Es autodescriptivo. Toda la información sobre la versión del ensamblaje, la descripción de los tipos de datos que contiene, los archivos que lo forman, etc., se encuentra dentro del propio ensamblaje, por lo que instalar un ensamblaje sólo requiere copiarlo. No se utiliza el registro de Windows.
- Registra sus dependencias a otros ensamblajes: Nombre, versión, etc.
- Pueden instalarse, en un mismo computador, diferentes versiones de un mismo ensamblaje, sin causar conflicto.
- Instalación sin impactos, es decir, el instalar un ensamblaje sólo puede afectar a los programas que lo utilizan. No hay posibilidad que un ensamblaje reemplace otro con el mismo nombre pero para otro uso, o con diferente versión. Los ensamblajes se identifican de manera única.
- Se ejecutan dentro de Dominios de Aplicación de un proceso.

Un ensamblaje en ejecución se le denomina aplicación. Dentro de un mismo proceso se pueden ejecutar varias aplicaciones, del mismo o distinto ensamblaje, cada uno en un Dominio de Aplicación distinto. Un Dominio de Aplicación es la frontera que aísla una aplicación del resto de aplicaciones. De esta manera, las fallas en una aplicación no pueden afectar a otra aplicación, aún perteneciendo ambas al mismo proceso. Para que un objeto de una aplicación acceda a uno en otra aplicación, requiere hacer uso de un objeto proxy, cuyo concepto es el mismo que el Stub de CORBA (ver capítulo 9).

Existen dos tipos de ensamblajes: Los **privados** y los **públicos** o compartidos.

Los ensamblajes privados son aquellos que se instalan junto con el programa que los utiliza, en el mismo directorio o en un subdirectorio de donde se encuentra este programa. No se manejan números de versión ni nombres únicos, dado que no se necesitan. Este tipo de ensamblaje puede causar conflictos (de versiones por ejemplo) pero sólo en la aplicación que lo utiliza, y como es natural se resuelven durante el proceso de desarrollo de dicho programa. Este tipo de ensamblaje no puede afectar otros programas.

Los ensamblajes públicos o compartidos pueden ser utilizados por más de un programa, por lo que se instalan en un lugar común. Estos ensamblajes deben seguir las siguientes reglas:

- Tener un nombre único (llamado nombre fuerte). Parte de este nombre es un número de versión mandatorio.
- Mayormente, estar instalado en la Global Assembly Cache, un directorio dentro del directorio de Windows (por ejemplo, C:\WINNT\assembly para Windows 2000).

El principal componente de la Metadata del Ensamblaje es el Manifiesto. Éste contiene:

- El nombre identificador para el ensamblaje, la versión, la cultura y una llave pública (una cadena de caracteres).
- Una lista de los archivos que forman el ensamblaje.
- Una lista de los ensamblajes referenciados por éste y por tanto, de los que depende para su ejecución.
- Un conjunto de Solicitudes de Permiso, que son los permisos necesarios para correr o utilizar el ensamblaje.
- Metadata de tipos de datos para aquellos tipos dentro de los archivos de módulo del ensamblaje.

Puede examinarse el contenido de un ensamblaje, incluyendo su manifiesto, utilizando el programa utilitario ILDASM.EXE.

El Ensamblaje Tipo Librería

Todos los programas hasta ahora generados son ensamblajes de un sólo archivo de tipo ejecutable. Para crear un ensamblaje tipo librería que sea utilizada por otro ensamblaje se debe de compilar como tal. Pongamos un ejemplo.

El siguiente código corresponde al archivo Ensamblaje1.cs:

```
using System;
public class Clase1 {
    public void Saludame(string nombre) {
        Console.WriteLine("Hola " + nombre);
    }
}
```

Para compilarlo como un ensamblaje tipo librería, utilizamos la opción /target:library del compilador csc.

```
csc /target:library Ensamblaje1.cs
```

Esto genera el archivo Ensamblaje1.dll. El siguiente código corresponde al archivo Ensamblaje2.cs, dentro del cual, se hace uso de la clase Clase1 definida dentro del ensamblaje Ensamblaje1.dll.

```
using System;
class Clase2 {
    public static void Main() {
```



```
        Clase1 obj = new Clase1();  
        obj.Saludame("Juan");  
    }  
}
```

Para compilarlo como un ensamblaje tipo ejecutable, que hace referencia a elementos dentro del ensamblaje Ensamblaje1.dll utilizamos la opción /reference:Ensamblaje1.dll del compilador csc.

```
csc /reference:Ensamblaje1.dll Ensamblaje2.cs
```

Esto genera el archivo Ensamblaje2.exe. Para ejecutar este programa, ambos archivos, Ensamblaje1.dll y Ensamblaje2.exe, deben de estar en el mismo directorio. Este ejemplo corresponde a un ensamblaje privado. Los ensamblajes públicos van más allá de los alcances del presente curso.

Relación con los Espacios de Nombres

C# separa los conceptos de espacios de nombres y el de ensamblaje. Un espacio de nombres puede estar definido por partes en varios ensamblajes.

Programación genérica

En este capítulo revisaremos la utilización de la Programación Genérica.

Introducción

Es una técnica que permite generalizar un código de forma que pueda ser utilizado con datos de distintos tipos. Para ejemplificar esta definición, revise la definición en Código 1 de una función de intercambio Swap en C++.

```
void Swap(int & a, int & b) { // cuide de usar S mayúscula, para no confundir
    int temp = b;           // con la versión de esta función en la librería
    b = a;                  // estándar
    a = temp;
}
...
int x = 1, y = 2;
Swap(x, y);
```

Código 1: Definición y uso de una función de intercambio Swap en C++.

Si deseáramos intercambiar datos de otros tipos (long, float, etc.) tendríamos que escribir varias veces prácticamente la misma función, cambiando solo el tipo de las variables. Para evitar esta repetición inútil de código los LP proveen diferentes mecanismos de programación genérica.

Por ejemplo, en C++ es posible utilizar punteros void para esta tarea, como lo muestra Código 2.

```
void Swap(void * a, void * b, int tamanyo) {
    char * pa = (char *)a;
    char * pb = (char *)b;
    char * ptemp = (char *)malloc(tamanyo);
    memcpy(ptemp, pb, tamanyo);
    memcpy(pb, pa, tamanyo);
    memcpy(pa, ptemp, tamanyo);
}
...
int x = 1, y = 2;
Swap(&x, &y, sizeof(x));
```

Código 2: Definición y uso de Swap en C++ generalizándola con punteros void.

Hay mucho que comentar respecto a este código, lo que se hará más adelante, pero en resumen esta y otras soluciones similares (que requieren que el programador baje a trabajar a nivel de bytes, perdiendo todas las ventajas de un LP fuertemente tipificado) adolecen de problemas de seguridad y eficiencia. Una mejor alternativa, y la que estudiaremos en este capítulo, la ofrecen los tipos y funciones genéricas (llamadas en conjunto genéricos), que son mecanismos que permiten generalizar la declaración de un tipo o una función, respecto a los tipos de datos que

utilizan, siendo especificados estos últimos recién al momento de utilizarse dicho tipo o llamarse a dicha función.

En otras palabras, los genéricos son declaraciones que no se amarran a tipos de datos particulares. Es cuando se utiliza un genérico cuando se especifican los tipos de datos con los que se desea trabajar, dándole así más información al compilador con la cual hacer una mejor verificación de tipos (y logrando un mejor static type-safe) y generar un código más eficiente. La programación genérica permite trabajar con conceptos genéricos en lugar de con casos particulares (ref. 13). En este capítulo veremos el uso de ésta técnica y las similitudes y diferencias entre los mecanismos de soporte que le dan C++, Java y C#.

Terminología

A los tipos de datos respecto a los que se generaliza la declaración de un genérico se les denominan parámetros-tipo formales o simplemente parámetros-tipo, mientras que a los tipos de datos que se indican al utilizar el genérico se les denominan argumentos-tipo actuales. En consecuencia, a los tipos genéricos se denominan también tipos parametrizados, y las funciones genéricas (y métodos genéricos) como funciones parametrizadas (y métodos parametrizados).

Es posible generalizar distintos tipos de datos, como las clases, las estructuras y las interfaces. Por ejemplo, una clase definida así se le denomina clase genérica o clase parametrizada.

En algunas implementaciones el uso de un genérico ocasiona que el compilador (o el intérprete) construya un nuevo tipo de dato equivalente a la particularización del genérico con determinados parámetros-tipo. A dichos tipos de datos se les denomina tipos construidos (constructed type).

Funciones genéricas y métodos genéricos

Las funciones genéricas y los métodos genéricos permiten generalizar el algoritmo implementado por una función o un método respecto a los tipos de datos que utiliza. Para ejemplificar cómo se realiza esta generalización y sus ventajas, sigamos con el ejemplo de la función Swap antes mostrado.

En Código 2 la función usa punteros void para recibir las direcciones de los datos que debe intercambiar. El principal problema de esta solución es que es muy insegura, pues no hay forma de verificar, ni en tiempo de compilación ni en tiempo de ejecución, que los punteros realmente apuntan a datos del mismo tipo y del tamaño indicado por el parámetro tamaño. No hay verificación de tipos y si el programador se equivoca puede estar intercambiando datos de distinto tipo sin que ni en compilación ni en ejecución esto sea detectado de alguna forma predecible. Por tanto, que el programa no se caiga al llamar a esta función depende en gran medida del programador que la usa. Además este tipo de soluciones pueden ser muy ineficientes, por ejemplo para esta función la cantidad de trabajo que se debe realizar para intercambiar datos primitivos (enteros por ejemplo) es innecesariamente grande.

La programación genérica ofrece una alternativa que da solución a estos problemas. En C++ los genéricos se implementan mediante plantillas (templates). Una plantilla para la función Swap podría definirse como en Código 3.

```
template<typename T> void Swap(T & a, T & b) {  
    T temp = b;  
    b = a;  
    a = temp;  
}
```

```
...  
int x = 1, y = 2, z = 3;  
Swap<int>(x, y); // los argumentos-tipo son pasados explícitamente  
Swap(y, z); // aquí el compilador infiere los argumentos-tipo
```

Código 3: Definición y uso de Swap usando una plantilla en C++.

Note como la implementación en Código 3 es casi igual al original en Código 1, salvo dos cambios: (1) La función comienza con la palabra reservada `template`, lo que indica que se trata de una plantilla de función; (ref. 2) el uso del identificador `T` como tipo de dato en toda la definición. El identificador `T` es llamado el parámetro-tipo de la plantilla `Swap` y mediante él se la especializará.

Al leer este programa el compilador de C++ literalmente genera el código correspondiente a cada especialización que encuentra. El código generado debido a una especialización equivale a tomar la plantilla involucrada y reemplazar los parámetros-tipo formales (`T` para el ejemplo) por los argumentos-tipo actuales (`int` para el ejemplo). En Código 3, la primera llamada a `Swap` genera el código de una función `Swap` donde cada ocurrencia del parámetro-tipo `T` es reemplazada por `int`. La segunda llamada requiere la misma especialización que la primera, por lo que el compilador no genera nuevo código en este caso.

Esta implementación de `Swap` como una plantilla permite que el compilador verifique los tipos de los parámetros utilizados, asegurándose de que ambos sean del mismo tipo. El código generado es tan eficiente como la implementación ideal mostrada al inicio, y además es completamente genérica. Cualquier tipo de dato que cuente con una sobrecarga del operador de asignación puede utilizarse con esta plantilla, que es básicamente lo único que exige el algoritmo de `Swap`.

En C# el equivalente sería un método genérico como en Código 4.

```
public class Ejml {  
    public static void Swap<T>(ref T a, ref T b) {  
        T temp = b;  
        b = a;  
        a = temp;  
    }  
    public static void Main()  
    {  
        int x = 1, y = 2;  
        float z = 3.0F;  
        Swap<int>(ref x, ref y);  
        //Swap(ref y, ref z); // ERROR de compilación  
        Console.WriteLine("x={0}, y={1}, z={2}", x, y, z);  
    }  
}
```

Código 5: Definición y uso de Swap con un método genérico en C#.

La sintaxis es similar a C++, solo que no se requiere utilizar una palabra reservada como `typename` por cada parámetro-tipo definido y la lista de parámetros-tipos va inmediatamente después del nombre del método. A pesar de esta similitud sintáctica, existen grandes diferencias en la implementación. Las dos principales son: (1) Una especialización de un genérico en C# genera código en tiempo de ejecución (con algunas optimizaciones), mientras que en C++ se genera en tiempo de compilación (lo que no podría ser de otra forma dado que C++ no es ejecutado por un intérprete); (2) la verificación de tipos en C# se hace sobre la propia definición del genérico, mientras que en C++ se realiza sobre el código generado por la especialización. Esto último es una ventaja para C#, dado que es posible verificar el correcto uso de los tipos por un genérico antes de ser utilizado, en lugar de esperar a que alguna especialización en particular arroje un error en su implementación, pero a su vez el no hacerlo así es una ventaja

para C++ porque permite definir genéricos y utilizarlos sin verse forzado a que los argumentos-tipo cumplan con todos los requerimientos del genérico.

En Java el equivalente también sería un método genérico, sin embargo una implementación de Swap en Java requeriría utilizar una clase auxiliar para permitir suplir la carencia de un mecanismo de paso por referencia. Dejaremos este caso para más adelante. Un ejemplo simple en Java sería un método que averigüe si un dato existe en un arreglo, para cualquier tipo de arreglo. Antes de los genéricos, dicho método tendría una implementación como en Código 5. Usando un método genérico la implementación sería como en Código 6.

```
public class Ejm2 {
    public static boolean Existe(Object dato, Object[] arr) {
        for(Object elemento : arr)
            if(elemento.equals(dato))
                return true;
        return false;
    }
    public static void main(String[] args) {
        Integer[] arr = {1, 3, 6, 9};
        System.out.println("2 existe?: " + Existe(2, arr));
        System.out.println("3 existe?: " + Existe(3, arr));
        System.out.println("\nHola\" existe?: " + Existe("Hola", arr));
    }
}
```

Código 6: Definición y uso de Existe utilizando la clase base Object en Java.

Ambos códigos, Código 5 y Código 6, no producen errores ni en compilación ni en ejecución. Esto es un resultado esperado para Código 5, pero no para Código 6, donde en la última línea se intenta buscar un objeto String en un arreglo de Integer's, lo que esperábamos que el compilador rechace dada nuestra definición de Existe donde se indica que ambos parámetros son o derivan del tipo T, y String ni es ni deriva de Integer. El código equivalente en C++ y C# capturaría en compilación esta inconsistencia arrojando un error. Entonces, ¿por qué pasa esto en Java?

```
public class Ejm2 {
    public static <T> boolean Existe(T dato, T[] arr) {
        for(T elemento : arr)
            if(elemento.equals(dato))
                return true;
        return false;
    }
    public static void main(String[] args) {
        Integer[] arr = {1, 3, 6, 9};
        System.out.println("2 existe?: " + Existe(2, arr));
        System.out.println("3 existe?: " + Existe(3, arr));
        System.out.println("\nHola\" existe?: " + Existe("Hola", arr));
    }
}
```

Código 7: Definición y uso de Existe como método genérico en Java.

La respuesta está en la estrategia de implementación de los genéricos en este lenguaje. A diferencia de C++ y C#, Java utiliza algo muy parecido a un proceso de pre-compilación (llamado front-end erasure) en donde un código con genéricos es transformado a uno sin genéricos equivalente. En otras palabras, un código fuente con genéricos es pasado en memoria RAM a uno sin genéricos y es este último el que es compilado. Esto se hizo así porque uno de los principales objetivos de implementación de genéricos en Java fue mantener una compatibilidad completa hacia atrás, esto es, todo el código generado al compilar las nuevas fuentes que utilicen genéricos se puedan ejecutar en cualquier JVM sin necesidad de modificarlas. Al leer un código, el compilador busca, por cada parámetro-tipo de un genérico, el tipo de dato existente más genérico (llamado upper bound) con el cual reemplazarlo, de forma que todo el código resultante luego del reemplazo sea válido. En los casos que no se pueda

encontrar un tipo que satisfaga esta condición, el compilador inserta operadores cast de forma que, al menos en tiempo de ejecución, si ocurre un error sea notificado mediante una excepción.

Para el Código 6, el compilador reemplaza T por Object, quedando el código compilado como en Código 5, dado que Object es el tipo de dato más genérico que permite que el código de Existe sea válido. Ese es el motivo por el cual la última llamada a este método es Código 6 no produce ningún error ni en compilación ni en ejecución. Sin embargo, hay mecanismos en Java que permiten refinar más el proceso de selección que hace el compilador. Estos mecanismos se denominan restricciones y actúan sobre los parámetros-tipo. En Java las restricciones son siempre respecto a la herencia y son de dos tipos: Límite superior (upper bound) y límite inferior (lower bound). En Código 7 se muestra una versión de Existe donde se especifica que el parámetro-tipo TD debe ser o igual o uno derivado del parámetro-tipo TA. Por tanto, a TA es el upper bound de TD.

```
public class Ejm2 {
    public static <TA, TD extends TA> boolean Existe(TD dato, TA[] arr) {
        for(TA elemento : arr)
            if(elemento.equals(dato))
                return true;
        return false;
    }

    public static void main(String[] args) {
        Integer[] arr = {1, 3, 6, 9};
        System.out.println("2 existe?: " + Existe(2, arr));
        System.out.println("3 existe?: " + Existe(3, arr));
        System.out.println("\"Hola\" existe?: " + Existe("Hola", arr));
    }
}
```

Código 8: Segunda versión de Existe como método genérico en Java con restricciones.

Al compilar Código 7 el compilador generará un error para la última llamada a Existe, pues no podrá encontrar ningún tipo de dato que cumpla la condición impuesta por la restricción del genérico. Las restricciones del tipo lower bound así como el uso del comodín “?” van más allá del alcance de este documento.

También es posible definir restricciones en C#. En Código 8 se muestra un ejemplo de aplicación de una restricción para una función de ordenamiento.

```
public class Ejm3 {
    public static void Ordenar<T>(T[] arreglo) where T : IComparable {
        for (int i = 0; i < (arreglo.Length - 1); i++)
            for (int j = i + 1; j < arreglo.Length; j++)
                if (arreglo[i].CompareTo(arreglo[j]) > 0) {
                    T temp = arreglo[i];
                    arreglo[i] = arreglo[j];
                    arreglo[j] = temp;
                }
    }

    public static void Main() {
        String [] arr = {"Jose", "Maria", "Ana"};
        Ordenar(arr);
        foreach (String s in arr)
            Console.WriteLine(s + ",");
    }
}
```

Código 9: Definición y uso de un método Ordenar en C# con restricciones.

En C# las restricciones van luego de los parámetros del método y antes de su cuerpo. Al igual que Java, las restricciones son respecto a la herencia. En la restricción se indican, luego de los dos-puntos, el nombre de la clase y/o los nombres de las interfaces que el parámetro-tipo deberá heredar y/o implementar. Adicionalmente se pueden colocar, inmediatamente después

de los dos-puntos, las restricciones especiales: (1) `class`, que indica que el parámetro-tipo restringido debe ser una clase; (2) `struct`, que indica que el parámetro-tipo restringido debe ser una estructura; (2) `new()`, que indica que el parámetro-tipo restringido debe tener un constructor sin parámetros. No se puede usar las restricciones `class` y `struct`, ni `struct` y `new()` a la vez. Adicionalmente es posible utilizar un tipo-construido para restringir un parámetro-tipo. Por ejemplo, en Código 8 podemos utilizar `where T : IComparable<T>` en la definición de `Ordenar`, utilizando así la versión genérica de `IComparable` con lo que permitimos que el CLR pueda generar código más eficiente en tiempo de ejecución. Lo mismo es posible en Java, como se muestra en Código 9, en donde se usa la interfaz genérica `Comparable`.

```
public class Ejm3 {
    public static <E extends Comparable<E>> void Ordenar(E[] arreglo) {
        for(int i = 0; i < (arreglo.length - 1); i++)
            for(int j = i + 1; j < arreglo.length; j++)
                if( arreglo[i].compareTo(arreglo[j]) > 0 ) {
                    E temp = arreglo[i];
                    arreglo[i] = arreglo[j];
                    arreglo[j] = temp;
                }
    }
    public static void main(String[] args) {
        String [] as = {"Jose", "Maria", "Ana"};
        Ordenar(as);
        System.out.println(java.util.Arrays.toString(as));
    }
}
```

Código 10: Definición y uso de un método `Ordenar` en Java con restricciones.

No existen restricciones tan claramente especificadas al inicio de la definición de un genérico en C++. Las restricciones en C++ se establecen en el propio cuerpo del genérico (esto es, están implícitas en el código que implementa al genérico) y son verificadas sobre el código generado por su especialización, y no sobre el genérico mismo. Por ejemplo, en Código 10 se muestra la versión de `Ordenar` con una plantilla en C++.

```
template<typename E> void Ordenar(E* arreglo, int tamanyo) {
    for(int i = 0; i < (tamanyo - 1); i++)
        for(int j = i + 1; j < tamanyo; j++)
            if( arreglo[i] > arreglo[j] ) {
                E temp = arreglo[i];
                arreglo[i] = arreglo[j];
                arreglo[j] = temp;
            }
}
int main() {
    int arr[] = {8, 5, 7, 2, 4}, tam_arr = sizeof(arr)/sizeof(arr[0]);
    Ordenar(arr, tam_arr);
    for(int i = 0; i < tam_arr; i++)
        std::cout << arr[i] << " ";
}
```

Código 11: Definición y uso de un método `Ordenar` en C++ como plantilla.

Como se ve en Código 10, para que una especialización de `Ordenar` no arroje un error de compilación debe existir una sobrecarga de `operator>` para el argumento-tipo actual utilizado, `int` en el ejemplo. Si bien esta forma de restringir no es tan clara como en Java y C#, ni tampoco permite que el compilador realice una verificación del correcto uso de los tipos sobre el propio genérico, es más flexible, dado que literalmente cualquier operación que se pueda realizar sobre un tipo o sobre un dato de un tipo puede representar una restricción.

Finalmente, existen características propias en la implementación dentro de cada lenguaje. Por ejemplo, C++ permite especificar valores por defecto a sus parámetros-tipo (por ejemplo, `typename T = int` indica que por defecto, si no se especifica, el parámetro-tipo `T` se especializa

con el argumento-tipo `int`) y utilizar constantes como parámetros de especialización de una plantilla (por ejemplo, `int C` utilizado como parámetro-tipo indicaría que debe de pasarse una constante entera al especializar la plantilla). Java permite restricciones de distinto tipo (`upper bound` y `lower bound`, así como el uso de un comodín para flexibilizar las reglas de subtipificación del lenguaje cuando así se requiere) relativos a la herencia de los parámetros-tipo. C# permite una restricción similar al `upper bound` de Java y restricciones respecto a la naturaleza (si es tipo valor o tipo referencia) y contenido (si contiene un constructor por defecto) de los parámetros-tipo.

Tipos de datos genéricos

Los tipos genéricos permiten generalizar un tipo de dato respecto a los tipos de datos que se utilizan en su definición. Otra forma de ver a un tipo genérico es como la definición de una familia de tipos de datos, donde sus miembros difieren solo en los tipos de datos que manipulan (ref. 1).

Una aplicación común de los tipos genéricos es en el manejo de colecciones. Por ejemplo, si deseáramos implementar una clase genérica para manejar un arreglo en C++ (de forma que se verifique que nunca se usen índices inválidos para accederlo) podríamos utilizar un código como en Código 11.

```
class ArregloEnteros {
    int tamanyo;
    int * pValores;

public:
    ArregloEnteros(int tamanyo) {
        this->tamanyo = tamanyo;
        pValores = new int[tamanyo];
    }
    ~ArregloEnteros() {
        delete [] pValores;
    }
    void estValor(int indice, int valor) {
        if(indice < 0 || tamanyo <= indice) throw -1;
        pValores[indice] = valor;
    }
    int obtValor(int indice) const {
        if(indice < 0 || tamanyo <= indice) throw -1;
        return pValores[indice];
    }
    int obtTamanyo() const {
        return tamanyo;
    }
};

int main() {
    ArregloEnteros arr(4);
    arr.estValor(0, 3);
    arr.estValor(1, 6);
    arr.estValor(2, 2);
    arr.estValor(3, 9);

    std::cout << "Arreglo=[";
    for(int i = 0; i < (arr.obtTamanyo() - 1); i++)
        std::cout << arr.obtValor(i) << ", ";
    std::cout << arr.obtValor(arr.obtTamanyo() - 1) << "]\n";

    return 0;
}
```

Código 12: Definición y uso de clase `ArregloEnteros` en C++.

El problema con el código en Código 11 es que la clase implementada no puede ser reutilizada para manejar arreglos de otros tipos de datos. Para evitar tener copias casi idénticas de la clase anterior podemos modificar su definición para generalizarla utilizando punteros void, como se puede apreciar en Código 12. Note en este código la necesidad de realizar operaciones cast tanto al ingresar los datos como al extraerlos. Esto evidencia un nuevo problema: Una clase así definida no me garantiza nada acerca de los valores que se le ingresan. No puedo garantizar que todos los elementos de la lista sean de un mismo tipo, o por lo menos que hereden de un mismo tipo. Como ejemplo, a un objeto ArregloPVoid se le podrían agregar punteros a enteros y a textos y no sucedería ningún error en compilación, pero muy probablemente sí en ejecución al extraer los valores, pues no sabríamos si todos éstos son del tipo esperado.

```
class ArregloPVoid {
    int tamanyo;
    void ** pValores;

public:
    ArregloPVoid(int tamanyo) {
        this->tamanyo = tamanyo;
        pValores = new void *[tamanyo];
    }
    ~ArregloPVoid() {
        delete [] pValores;
    }
    void estValor(int indice, void * valor) {
        if(indice < 0 || tamanyo <= indice) throw -1;
        pValores[indice] = valor;
    }
    void * obtValor(int indice) const {
        if(indice < 0 || tamanyo <= indice) throw -1;
        return pValores[indice];
    }
    int obtTamanyo() const {
        return tamanyo;
    }
};

int main() {
    ArregloPVoid arr(4);
    arr.estValor(0, (void*)3);
    arr.estValor(1, (void*)6);
    arr.estValor(2, (void*)2);
    arr.estValor(3, (void*)9);

    std::cout << "Arreglo=[";
    for(int i = 0; i < (arr.obtTamanyo() - 1); i++)
        std::cout << (int)arr.obtValor(i) << ", ";
    std::cout << (int)arr.obtValor(arr.obtTamanyo() - 1) << "]\n";

    return 0;
}
```

Código 13: Definición y uso de clase ListaPVoid en C++.

En C++ el resultado de realizar un cast incorrecta puede tener un resultado imprevisible. No ocurre así en Java y C#, en donde un cast incorrecto es detectado en tiempo de ejecución generando una excepción, la que puede ser capturada por la lógica del programa (si el programador recuerda hacerlo, y sino, por el intérprete). Si bien la solución de Java y C# es más segura que la de C++ (con el consiguiente costo en el desempeño), no elimina el problema, solo da un paliativo en caso de que éste suceda.

Los tipos genéricos permiten dar una real solución a un problema como el descrito, un problema que puede solucionarse si el lenguaje permite algún mecanismo por el cual el programador le pueda dar la información exacta de los tipos de datos con los que desea trabajar.

El código en Código 13 es un ejemplo de definición de un tipo genérico para el manejo de un arreglo.

```
template<typename T> class ArregloGen {
    int tamanyo;
    T * pValores;

public:
    ArregloGen(int tamanyo) {
        this->tamanyo = tamanyo;
        pValores = new T[tamanyo];
    }
    ~ArregloGen() {
        delete [] pValores;
    }
    void estValor(int indice, T valor) {
        if(indice < 0 || tamanyo <= indice) throw -1;
        pValores[indice] = valor;
    }
    T obtValor(int indice) const {
        if(indice < 0 || tamanyo <= indice) throw -1;
        return pValores[indice];
    }
    int obtTamanyo() const {
        return tamanyo;
    }
};

int main() {
    ArregloGen<int> arr(4);
    arr.estValor(0, 3);
    arr.estValor(1, 6);
    arr.estValor(2, 2);
    arr.estValor(3, 9);

    std::cout << "Arreglo=[";
    for(int i = 0; i < (arr.obtTamanyo() - 1); i++)
        std::cout << arr.obtValor(i) << ", ";
    std::cout << arr.obtValor(arr.obtTamanyo() - 1) << "]\n";

    return 0;
}
```

Código 14: Definición y uso de una plantilla de clase ArregloGen en C++.

Note como en Código 13 se especifica el argumento-tipo con el que se especializa el genérico al momento de definir la variable arr. Note además como ya no es necesario realizar un cast al momento de establecer y obtener valores del arreglo (con los métodos estValor y obtValor respectivamente). Este código no solo es tan eficiente como uno hecho a medida (como en Código 11) sino que además puede ser reutilizado para cualquier tipo de dato y el compilador es capaz de verificar el correcto uso de los tipos en tiempo de compilación. Como ejemplo, si Código 13 se modificase para agregar la sentencia arr.estValor(3, "un texto"), el compilador arrojaría un error indicando que se espera un parámetro de tipo int y no char*. De esta forma se evita que una especialización de un tipo genérico sea utilizada con tipos incorrectos de datos.

```
using System;

class ArregloGen<T> {
    int tamanyo;
    T [] pValores;

    public ArregloGen(int tamanyo) {
        this.tamanyo = tamanyo;
        pValores = new T[tamanyo];
    }
    public void estValor(int indice, T valor) {
        if(indice < 0 || tamanyo <= indice)
            throw new ArgumentException("Indice invalido");
        pValores[indice] = valor;
    }
}
```

```
    }
    public T obtValor(int indice) {
        if (indice < 0 || tamanyo <= indice)
            throw new ArgumentException("Indice invalido");
        return pValores[indice];
    }
    public int obtTamanyo() {
        return tamanyo;
    }
}

class Ejm5 {
    public static void Main() {
        ArregloGen<int> arr = new ArregloGen<int>(4);
        arr.estValor(0, 3);
        arr.estValor(1, 6);
        arr.estValor(2, 2);
        arr.estValor(3, 9);

        Console.WriteLine("Arreglo=[");
        for(int i = 0; i < (arr.obtTamanyo() - 1); i++)
            Console.WriteLine(arr.obtValor(i) + ", ");
        Console.WriteLine(arr.obtValor(arr.obtTamanyo() - 1) + "]\n");
    }
}
```

Código 15: Definición y uso de una clase genérica ArregloGen en C#.

El código en Código 14 es la versión en C# del ejemplo anterior. En Java existe un inconveniente respecto a este ejemplo: No se permite instanciar arreglos de un parámetro-tipo en la definición de un genérico (debido a la técnica *erased* utilizada para soportar genéricos, como se verá más adelante). Por tanto, veamos un ejemplo distinto en Java, una clase para manejar una lista simplemente enlazada. En Código 15 se muestra esto.

```
class ListaGen<T> {
    T valor;
    ListaGen<T> sgte;

    public ListaGen(T valor) {
        this.valor = valor;
        sgte = null;
    }
    public T obtValor() {
        return valor;
    }
    public ListaGen<T> obtSgte() {
        return sgte;
    }
    public void agregar(T valor) {
        agregar(new ListaGen<T>(valor));
    }
    public void agregar(ListaGen<T> nuevo) {
        if(sgte == null)
            sgte = nuevo;
        else
            sgte.agregar(nuevo);
    }
}

class Ejm4 {
    public static void main(String[] args) {
        ListaGen<Integer> lista = new ListaGen<Integer>(3);
        lista.agregar(6);
        lista.agregar(2);
        lista.agregar(9);

        System.out.print("Lista = (" + lista.obtValor());
        ListaGen<Integer> recorrer = lista.obtSgte();
        while (recorrer != null) {
            System.out.print(", " + recorrer.obtValor());
            recorrer = recorrer.obtSgte();
        }
    }
}
```

```
}  
    System.out.print(" ");  
}  
}
```

Código 16: Definición y uso de una clase genérica ListaGen en Java.

Note que en Código 15 se utiliza el tipo Integer en lugar del tipo int como argumento-tipo de la especialización del genérico. Esto se debe a que Java solo permite utilizar clases e interfaces como argumentos-tipo, y no tipos primitivos como int o double. Cuando se llama al método agregar se realiza una operación de auto-boxing. Como existe una clase por cada tipo primitivo, el uso de auto-boxing permite utilizar tipos primitivos con genéricos a pesar de la restricción anterior, con el costo adicional sobre el desempeño que esto implica.

Todo lo mencionado para los métodos genéricos respecto a las restricciones es aplicable a los tipos genéricos.

La programación genérica es un tema bastante extenso, y sus implicaciones en otros conceptos soportados por el mismo lenguaje son complejas. Por ejemplo, ¿se puede heredar de un genérico?, ¿se puede sobrecargar los métodos dentro de un genérico?, ¿los miembros estáticos de un tipo genérico son compartidos por todas las especializaciones? Este resumen solo busca acercar lo suficiente al alumno al tema como para entender claramente el concepto, no para que sea un experto en el mismo. En la siguiente sección se verá el uso más común que haremos de esta técnica a lo largo del curso: Los genéricos en las librerías estándar para el manejo de colecciones.

Genéricos en las librerías para el manejo de colecciones

En los 3 lenguajes de programación utilizados se cuentan genéricos para el manejo de colecciones como parte de su librería estándar. Veamos algunos ejemplos en cada lenguaje.

La STL de C++

La librería estándar de C++ para el manejo de colecciones es una librería de genéricos, de donde deviene su nombre Standard Template Library (o STL). La STL incluye muchos tipos genéricos que implementan estructuras de datos comúnmente utilizadas, y funciones genéricas que implementan algoritmos comúnmente utilizados para manejar dichas estructuras de datos. Sus objetivos de diseño principales incluyen el alto rendimiento y la flexibilidad de uso (ref. 3). Sus componentes principales son: Contenedores (estructuras de datos populares representadas como plantillas), iteradores (clases cuyos objetos son utilizados para recorrer los elementos dentro de los contenedores) y algoritmos (funciones genéricas que implementan algoritmos comunes de manipulación de datos sobre contenedores). Veremos ejemplos de cada uno.

Los contenedores en STL se agrupan en 5 categorías: Contenedores de secuencia, contenedores ordenados, contenedores asociativos, adaptadores de contenedores (las dos primeras categorías también se denominan contenedores de primera clase) y contenedores especializados. Los contenedores de secuencia representan estructuras de datos lineales (vectores y listas enlazadas). Los contenedores ordenados suelen utilizar árboles binarios balanceados para mantener sus elementos ordenados. Los contenedores asociativos representan colecciones de pares llave-valor, donde una llave es utilizada para recuperar su valor asociado y suelen utilizar estructuras de datos no lineales (como tablas hash y árboles) para permitir búsquedas rápidas. Los

adaptadores son clases que derivan de las anteriores, modificando y restringiendo la interfaz que exponen para darles un uso particular (por ejemplo, las pilas y las colas son adaptadores de la clase vector). Finalmente, existen los contenedores especializados (o casi-contenedores) que son sumamente eficientes pero ofrecen una interfaz más restringida que el resto de contenedores (por ejemplo la clase string y la clase bitset).

Los contenedores son clases genéricas definidas en varios archivos de encabezados. La Tabla 1 describe los contenedores más comúnmente utilizados de la STL.

Por mucho los contenedores más utilizados son vector (una clase genérica) y string (una clase). En Código 16 se muestra un ejemplo del uso de estos dos contenedores. Note el uso de los métodos `begin()` y `end()` a lo largo del código. Estos métodos devuelven objetos iteradores del tipo adecuado al contenedor de donde se obtienen. Un iterador es un objeto que encapsula un puntero a uno de los elementos de un contenedor y que además sobrecarga los típicos operadores utilizados con punteros (incremento y decremento, sumas y restas, asignación, direccionamiento y comparación) de forma que se les pueda utilizar como si de punteros se tratase.

Tabla 6 - 1: Algunos contenedores de la STL.

Tipo de contenedor	Archivo de encabezado	Clase	Características principales
De secuencia	vector	vector	Inserciones y eliminaciones rápidas al final. Acceso directo a cualquier elemento.
	deque	deque	Igual que vector, pero expandible por ambos extremos, al inicio y al final.
	list	list	Lista doblemente enlazada. Inserción y eliminación rápida en cualquier parte.
Ordenados	set	set	Búsqueda rápida. No se permiten duplicados.
		multiset	Búsqueda rápida. Si se permiten duplicados.
Asociativos	map	map	Mapeo de uno a uno, no se permiten duplicados. Búsqueda rápida basada en claves.
		multimap	Mapeo de uno a varios, si se permiten duplicados. Búsqueda rápida basada en claves.
Adaptadores	stack	stack	Último en entrar, primero en salir. (LIFO)
	queue	queue	Primero en entrar, primero en salir. (FIFO)

Tipo de contenedor	Archivo de encabezado	Clase	Características principales
Casi-contenedores		priority_queue	El elemento de mayor prioridad siempre es el primero en salir.
	bitset	bitset	Manipulación eficiente de un arreglo de bits.
	string	string	Manipulación de texto.
	valarray	valarray	Una implementación especialmente eficiente de un arreglo, pero con una interfaz restringida.

Lo importante sobre los iteradores es que: (1) A diferencia de un puntero, un objeto iterador verifica que no nos salgamos a apuntar a posiciones inválidas de memoria, esto es, que no nos vayamos más allá de los elementos que forman la colección; (2) un iterador es más general que un puntero, permitiendo de forma transparente y uniforme desplazarnos por cualquier contenedor sin importar como internamente éste organice sus elementos (sea un arreglo con elementos contiguos, una lista enlazada, un árbol, un grafo, etc.) pues cada contenedor define su propia clase iterador; (3) permite definir los algoritmos de manipulación de elementos de un contenedor de forma independiente a la definición de los contenedores mismos, como veremos más adelante.

```
#include <iostream>
#include <vector> // para la clase genérica "vector"
#include <string> // para la clase "string"
#include <sstream> // para las clases "ostringstream" y "istringstream"

using namespace std;

template<typename T> void reportar(string nombre, vector<T> vtor) {
    cout << nombre << ": size=" << vtor.size()
        << ", capacidad=" << vtor.capacity()
        << ", empty=" << vtor.empty()
        << ", valores=[";
    typename vector<T>::iterator it = vtor.begin(); // no todos los compiladores obligan
    a // usar "typename" aquí
    while(it < vtor.end()) {
        cout << *it << ", ";
        it++;
    }
    cout << "]\n";
}

void ejemploVectores() {
    cout << "***** Ejemplo de vectores *****" << endl;

    vector<int> vtorInt1; // Un vector de enteros vacío
    vector<int> vtorInt2(10); // Un vector de enteros con 10 valores "0"
    vector<int> vtorInt3(10, 2); // Un vector con 10 valores "2"

    reportar("Vector1", vtorInt1);
    reportar("Vector2", vtorInt2);
    reportar("Vector3", vtorInt3);

    vtorInt1.push_back(3); // inserto al final
    vtorInt1.push_back(6);
    vtorInt1.push_back(4);
    vtorInt1.push_back(9);
    vtorInt1.insert(vtorInt1.begin() + 2, 5); // inserto luego del 2do elemento
```

```

        for(int i = 0; i < vtorInt2.size(); i++)          // size() = número de elementos en el
vector
            vtorInt2[i] = i * i;    // sobrecarga de operador[]
        vtorInt2.erase(vtorInt2.begin(), vtorInt2.begin() + 2); // elimino los 2 primeros
        vtorInt2.erase(vtorInt2.end() - 2, vtorInt2.end()); // elimino los 2 últimos

        vtorInt3.front() = 100; // retorna una referencia al primer elemento
        vtorInt3.back() = 200; // retorna una referencia al último elemento
        vtorInt3.at(5) = 400; // a diferencia de "operator[]", "front" y "back", "at"
verifica
                                // el índice, arrojando una excepción si es inválido

        reportar("Vector1", vtorInt1);
        reportar("Vector2", vtorInt2);
        reportar("Vector3", vtorInt3);

        vtorInt2.assign(15, 77); // el vector pasa a tener 15 números "77".
        vtorInt3.assign(vtorInt2.begin() + 5, vtorInt2.end() - 5);
        vtorInt1 = vtorInt3;

        reportar("Vector1", vtorInt1);
        reportar("Vector2", vtorInt2);
        reportar("Vector3", vtorInt3);
    }

void ejemploStrings() {
    cout << endl << "***** Ejemplo de string's *****" << endl;

    string cad1("Texto de cadena 1"), cad2 = "Texto de cadena 2", cad3(8, 'x');
    //string cad4 = 'c', cad5 = 34; // Error: "string" no ofrece estas conversiones
    cout << "cad1=" << cad1 << ", cad2=" << cad2 << ", cad3=" << cad3 << endl;

    cout << "cad1 impreso caracter a caracter: ";
    for(int i = 0; i < cad1.length(); i++)
        cout << cad1[i];
    cout << endl;

    cout << "\nLuego de modificar las cadenas tenemos:\n";
    cad1 = cad2;
    cad3 = "[" + cad1 + "]" + cad2 + ";"; // no puede concatenar con otros tipos de
datos
    cout << "cad1=" << cad1 << ", cad2=" << cad2 << ", cad3=" << cad3 << endl;

    cout << "\nOperaciones con cadenas\n";
    if(cad1 > cad2) cout << "cad1 es mayor que cad2" << endl;
    else cout << "cad1 es menor o igual que cad2" << endl;
    cout << "La subcadena de cad1 de la posicion 2 a la 6 es [" << cad1.substr(2, 6) <<
"]\n";
    cout << "La posicion de la 1era ocurrencia de la palabra \"cadena\" en cad1 es "
        << cad1.find("cadena") << endl;

    cout << "\nTrabajar con un flujo de texto en memoria RAM\n";
    int entero = 100;
    char caracter = 'a';
    ostream strOut;
    strOut << "Así puedo concatenar numeros [" << entero << "], caracteres ["
        << caracter << "], "
        << "y cualquier tipo de dato predefinido o para el cual haya una sobrecarga "
        << "de \"operator<<\"";
    cout << "strOut=" << strOut.str() << endl; // "str()" retorna un objeto "string" con
una
                                                // copia de la cadena
}

int main() {
    ejemploVectores();
    ejemploStrings();
    return 0;
}

```

Código 17: Ejemplo de uso de contenedores de la STL de C++.

Para declarar una variable de tipo iterador se utiliza el tipo `inner iterator` definido dentro de cada clase contenedor. Eso se puede apreciar en la implementación de la función genérica `reportar`.

En Código 16 también se muestra un ejemplo del uso de la clase `string`. Esta clase ofrece varios constructores, sobrecargas del operador `+` (que sirve para concatenar datos de tipo `string` y/o datos de tipo `char*`), de los operadores relacionales (`'=='`, `'>='`, `'<'`, etc.), del operador de acceso a arreglo (`[]`), para acceder a cada carácter individual cómodamente), y de los operadores `'<<'` y `'>>'` (para escribir/leer un `string` hacia/desde un flujo). Si se desea concatenar texto con otros tipos de datos se debe utilizar la clase `ostringstream`, y si se desea procesar un texto en memoria (igual como se hace al leer de la entrada estándar con `cin`) se debe usar `istreamstring`.

Finalmente, la STL incluye un conjunto de funciones genéricas que implementan algoritmos para procesar datos de un contenedor. Algunas de estas funciones modifican los elementos del contenedor (por ejemplo `copy`, `remove`, `rotate`) y otras no (por ejemplo, `count`, `find`, `search`). En Código 17 se muestra un ejemplo de uso de estas funciones. Lo interesante aquí es el enfoque de diseño seguido. Normalmente las librerías de clases para manipular contenedores definen los algoritmos de manipulación de los mismos mediante métodos dentro de las clases. La STL utiliza un enfoque distinto, la definición de los algoritmos se separa de la definición de los contenedores y operan sobre los elementos de éstos sólo indirectamente, a través de los iteradores. Esta separación facilita la escritura de algoritmos genéricos que se apliquen a muchas clases de contenedores (ref. 3). Sin embargo, los algoritmos dependen de los iteradores y de las características de estos. Por ejemplo, algunos algoritmos solo requieren leer o escribir unidireccionalmente los elementos de un contenedor, otros en ambas direcciones (hacia adelante y hacia atrás) y otros acceder a cualquier elemento aleatoriamente. Debido a esto hay una clasificación de los iteradores según estas funcionalidades esperadas y cada algoritmo define con que tipo de iterador requiere trabajar. Por tanto, solo se puede aplicar un algoritmo a un contenedor si su iterador soporta las funcionalidades que dicho algoritmo requiere.

```
#include <iostream>           // cin, cout
#include <vector>              // vector
#include <algorithm>           // sort, copy, random_shuffle
#include <iterator>            // ostream_iterator

using namespace std;

int main() {
    ostream_iterator<int> salida(cout, " ");
    vector<int> v;
    int val;

    cout << "Ingrese una secuencia de enteros y finalice con CTRL+Z: ";
    while(cin >> val) // mientras no es fin de archivo
        v.push_back(val); // agrego al vector

    sort( v.begin(), v.end() ); // ordeno

    cout << "La secuencia de enteros ordenada es: ";
    copy ( v.begin(), v.end(), salida );

    cout << "\nLa secuencia de enteros desordenada es: ";
    random_shuffle ( v.begin(), v.end() );
    copy ( v.begin(), v.end(), salida );

    cout << "\nLa secuencia de enteros invertida es: ";
    reverse ( v.begin(), v.end() );
    copy ( v.begin(), v.end(), salida );

    cout << "\nIngrese un numero a buscar: ";
    cin.clear(); // elimino el valor CTRL+Z que quedó en el buffer de entrada
    cin >> val;
    vector<int>::iterator it = find( v.begin(), v.end(), val );
    cout << "El valor [" << val << "] " << (it == v.end() ? "no" : "si")
```

```
<< " se encontro en el vector\n";

cout << "\nUna secuencia aleatoria de numeros enteros es: ";
generate( v.begin(), v.end(), rand );
copy ( v.begin(), v.end(), salida );

return 0;
}
```

Código 18: Ejemplo de uso de algoritmos de la STL de C++.

La librería para el manejo de colecciones de Java

Bajo el nombre de Java Collections Framework la versión 1.5 de Java ofrece una librería para el manejo de colecciones que incluye versiones genéricas para las clases que la confirman. A semejanza de la STL, esta librería contiene clases para manejar colecciones secuenciales (las que implementan la interfaz List, como ArrayList y LinkedList), colecciones sin valores repetidos y/o ordenadas (implementan la interfaz Set, como HashSet, TreeSet y LinkedHashSet) y colecciones asociativas (implementan la interfaz Map, como HashMap, TreeMap y LinkedHashMap). Aparte de estas implementaciones (llamadas de propósito general), existen otras: de propósito especial, concurrentes, envolturas, convenientes, y abstractas.

Estas colecciones implementan una o más interfaces que son las que definen los comportamientos de diferentes tipos de colecciones, lo que facilita la extensión de la librería (agregando nuevas implementaciones a las interfaces) y la sustitución de una implementación por otra en un programa (donde este programa refiere a la implementación sólo al instanciarla, y refiere a dicho objeto mediante el tipo de la interfaz en el resto del código) para buscar el mejor desempeño o agregar nueva funcionalidad.

En Ilustración 6 - 1 se muestra el árbol de herencia de las interfaces más importantes de esta

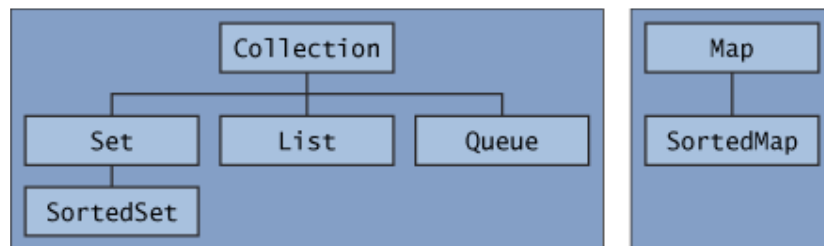


Ilustración 6 - 1: Las principales interfaces para colecciones de Java (extraído de The Java Tutorials)

librería.

La interfaz Map no hereda de Collection debido a que sus implementaciones no representan realmente una colección, sino un mapeo de los elementos de una colección ya existente. Todas estas interfaces son genéricas. Por ejemplo, la interfaz Collection comienza su definición: public interface Collection<E> ... Por tanto, al declarar una variable de algún tipo de colección o al instanciar alguna clase que las implemente el programador debería indicar el argumento-tipo correspondiente al tipo de dato que almacenará la colección. Si no se hace, el compilador utiliza el tipo Object. De esta forma el compilador podrá realizar verificaciones de tipo en tiempo de compilación.

En Tabla 6 - 2 se muestran las implementaciones de propósito general más utilizadas para las interfaces en Ilustración 1. Como regla general, el programador debe trabajar casi

completamente con los tipos de las interfaces, y solo referir a las implementaciones al momento de crear un objeto para manejar una nueva colección. Todas las implementaciones de una misma interfaz ofrecen lo mismo, y por tanto la elección de la implementación a utilizar solo afectará el desempeño final del programa, según el uso que se les den a los objetos de dicha implementación.

Interfaces	Implementaciones				
	Tabla hash	Arreglo redimensionable	Árbol	Lista enlazada	Tabla hash + L.Enlazada
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList, Vector		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap
Queue				LinkedList	
SortedSet			TreeSet		
SortedMap			TreeMap		

Tabla 6 - 2: Implementaciones más utilizadas de las diferentes interfaces de la librería para colecciones de Java.

No todas las implementaciones de las interfaces para colecciones soportan todos los métodos (lo que no es el caso de ninguna de las implementaciones de la Tabla 2). Cuando un método no es soportado y es llamado, la implementación dispara una excepción (UnsupportedOperationException). Esto permite mantener un número reducido de interfaces para colecciones en esta librería, facilitando su aprendizaje y uso.

Una de las implementaciones más comúnmente utilizada es ArrayList. En Código 18 se muestra un ejemplo del uso de este genérico.

```
import java.util.*;

class Ejm6 {
    public static <E> void reportar(String nombre, List<E> l) {
        System.out.print(nombre + ": size=" + l.size()
            + ", isEmpty=" + l.isEmpty()
            + ", valores=["
        );
        Iterator<E> it = l.iterator();
        while(it.hasNext())
            System.out.print(it.next() + ", ");
        System.out.print("]\n");
    }
    public static void main(String[] args) {
        List<Integer> l1 = new ArrayList<Integer>();
        List<Integer> l2 = new LinkedList<Integer>();
        List<Integer> l3 = new Vector<Integer>();
    }
}
```

```
System.out.println("Listas iniciales:");
reportar("Lista1", l1);
reportar("Lista2", l2);
reportar("Lista3", l3);

for(int i = 1; i <= 10; i++)
    l1.add(i);

l2.addAll(Collections.nCopies(15, 0));
for(int i = 3; i < l2.size() - 3; i++)
    l2.set(i, i - 2);
l2.subList(0, 5).clear();

Collections.addAll(l3, -1, -2, -3, -4, -5, -6);
l3.addAll(3, l1);
l3.add(6, 100);
l3.set(0, 200);
l3.set(l3.size() - 1, 300);
l3.set(l3.size() / 2, 400);

System.out.println("\nLuego agregar elementos:");
reportar("Lista1", l1);
reportar("Lista2", l2);
reportar("Lista3", l3);

System.out.println("\nBuscando elementos:");
System.out.println("Lista1 contiene '5': " + l1.contains(5));
System.out.println("Lista2 posicion de '0': " + l2.indexOf(0));
System.out.println("Lista3 elemento en posicion '4': " + l3.get(4));

System.out.println("\nRemoviendo elementos:");
l1.remove(2); // remuevo el elemento en posicion 2
l2.remove(new Integer(0)); // remuevo el 1er cero
l3.removeAll(l1); // remuevo de l3 todos los que hay en l1
reportar("Lista1", l1);
reportar("Lista2", l2);
reportar("Lista3", l3);
    }
}
```

Código 19: Ejemplo de uso de la interaz genérica List de la librería de Java.

Además de las interfaces y sus implementaciones, esta librería también ofrece algoritmos utilizables en muchas de las diferentes implementaciones. En Código 19 se muestra un ejemplo del uso de algunos de estos algoritmos.

```
import java.util.*;

class Ejm7 {
    public static <E> void imprimir(List<E> v) {
        Iterator<E> it = v.iterator();
        System.out.print("[");
        while(it.hasNext())
            System.out.print(it.next() + ", ");
        System.out.print("]");
    }

    public static void main(String[] args) {
        List<Integer> v = new ArrayList<Integer>();
        Scanner input = new Scanner(System.in);

        System.out.print("Ingrese una secuencia de enteros (finalice con *): ");
        while(input.hasNextInt())
            v.add(input.nextInt()); // leo y guardo
        input.next(); // descarto el '*' ingresado

        System.out.print("\nLa secuencia de enteros leida es: ");
        imprimir(v);

        Collections.reverse(v); // invierto
        System.out.print("\nLa secuencia de enteros invertida es: ");
        imprimir(v);
    }
}
```

```
        Collections.shuffle(v); // desordenada
        System.out.print("\nLa secuencia de enteros desordenada es: ");
        imprimir(v);

        Collections.sort(v); // ordeno
        System.out.print("\nLa secuencia de enteros ordenada es: ");
        imprimir(v);

        System.out.print("\nIngrese un numero a buscar: ");
        int val = input.nextInt();
        int pos = Collections.binarySearch(v, val); // ordeno
        System.out.println("El valor [" + val + "] se encontro en la posicion [" +
            + pos + "]");
    }
}
```

Código 20: Ejemplo de uso de los algoritmos para colecciones de la librería de Java.

La librería para el manejo de colecciones de .NET

La versión 2.0 de .NET Framework agregó los nuevos espacios de nombres `System.Collections.Generic` y `System.Collections.ObjectModel` los cuales contienen las versiones genéricas de los tipos (clases, interfaces y estructuras) para el manejo de colecciones. A semejanza de STL y la Java Collections Framework, este espacio de nombres contiene clases para manejar colecciones secuenciales (las que implementan la interfaz `ICollection`, como `List`, y las que implementan la interfaz `ICollection`, como `Collection`, `Stack`, `Queue`, `KeyedCollection`, `LinkedList`), colecciones asociativas (las que implementan la interfaz `IDictionary`, como `Dictionary`, `SortedList`, `SortedDictionary`) y colecciones ordenadas (sean secuenciales o asociativas, como `KeyedCollection`, `SortedList`, `SortedDictionary`). Sin embargo, como salta a la vista, la clasificación es aquí distinta.

La interfaz genérica `ICollection` representa una colección general, mientras que `ICollection` representa colecciones secuenciales a cuyos elementos se puede acceder utilizando índices. Las clases genéricas que limitan el acceso a sus elementos (como `Stack` y `Queue`) implementan solo `ICollection` y sus implementaciones internas son generalmente secuenciales, pero no es requisito que así lo sea. La interfaz genérica `IDictionary` representa colecciones de pares clave-valor, donde una clave es utilizada para recuperar su correspondiente valor. Los modelos de objetos internos de las clases genéricas que implementan `IDictionary` son variados, como tablas hash (`Dictionary`) y árboles binarios de búsqueda (`SortedList`, `SortedDictionary`).

De manera similar a Java, la librería de .NET contiene interfaces genéricas y clases que las implementan, pero a diferencia del enfoque de Java, las interfaces de .NET son minimalistas, esto es, solo incluyen los métodos elementales para manipular el tipo de colección que representan, lo que no propicia el uso extensivo de dichas interfaces en el código, en lugar de las implementaciones, facilitando el cambio de estas últimas para lograr un mejor desempeño del programa final.

El diagrama de clases en Ilustración 6 - 2 muestra las relaciones entre las interfaces genéricas y clases genéricas mencionadas.


```
Console.WriteLine("Lista2 posicion de '0': " + lista2.IndexOf(0));
Console.WriteLine("Lista3 elemento en posicion '4': " + lista3[4]);

Console.WriteLine("\nRemoviendo elementos:");
lista1.RemoveAt(2); // remuevo el elemento en posicion 2
lista2.Remove(0); // remuevo el 1er cero
foreach(int elemento in lista1) // remuevo lista1 de lista3
    lista3.Remove(elemento);
// la siguiente es una forma equivalente de remover lista1 de lista3,
// utilizando un predicado
//lista3.RemoveAll(delegate(int elemento) { return lista1.Contains(elemento);
});

Reportar("Lista1", lista1);
Reportar("Lista2", lista2);
Reportar("Lista3", lista3);
}
}
```

Código 21: Ejemplo de uso de la clase genérica List de la librería de .NET.

A diferencia de STL y la Java Collections Framework, .NET Framework no separa los algoritmos de manipulación de colecciones de las colecciones mismas. Como contraparte se hace uso extensivo de delegados para la manipulación de los elementos de una colección. Como ejemplo, métodos de List tales como Exist, Find, ForEach, entre otros, reciben un objeto delegado como parámetro y lo usan para procesar cada elemento de la colección. A estos delegados se les llama predicados.

En Código 21 se muestra un ejemplo de uso de algunos métodos de List para manipular sus elementos y el uso de un predicado. Esta práctica es similar al uso de las llamadas funciones-objeto de STL (object-function) cuyo estudio cae fuera de los objetivos del presente documento.

```
using System;
using System.Collections.Generic;

class Ejm7 {
    public static void Imprimir<E>(List<E> lista) {
        Console.Write("[");
        foreach (E elemento in lista)
            Console.Write(elemento + ", ");
        Console.Write("]\n");
    }
    static bool LeeEntero(out int valor) {
        try
        {
            string linea = Console.ReadLine();
            valor = Int32.Parse(linea);
            return true;
        }
        catch (FormatException)
        {
            valor = 0;
            return false;
        }
    }
    public static void Main() {
        List<int> lista = new List<int>();

        Console.Write("Ingrese una lista de enteros (finalice con *): ");
        int valor;
        while(LeeEntero(out valor))
            lista.Add(valor); // leo y guardo

        Console.Write("La lista de enteros leida es: ");
        Imprimir(lista);

        lista.Reverse(); // invierto
        Console.Write("La lista de enteros invertida es: ");
        Imprimir(lista);
    }
}
```

```
// List no ofrece un método para desordenar

lista.Sort(); // ordeno
Console.Write("La lista de enteros ordenada es: ");
Imprimir(lista);

Console.Write("Ingrese un numero a buscar: ");
if(LeerEntero(out valor)) {
    int pos = lista.BinarySearch(valor); // busco
    Console.WriteLine("El valor [{0}] se encontro en la posicion [{1}]",
        valor, pos);
}

// ejemplo del uso de un predicado
Console.Write("¿La lista contiene números pares? : " +
    lista.Exists(delegate(int elemento) { return (elemento % 2) == 0; }
));
}
```

Código 22: Ejemplo de uso de métodos de manipulación de los elementos de una colección con la clase List de la librería de .NET y el uso de un predicado.

Alcances, limitaciones y diferencias en el soporte a la programación genérica en C++, Java y C#

Los 3 lenguajes vistos utilizan estrategias muy distintas para soportar la programación genérica. Veamos un resumen de estas diferencias y las fortalezas y debilidades de cada una.

C++ utiliza el concepto de plantilla (template) para representar los genéricos, especializando dichas plantillas en tiempo de compilación. Esto implica generación de código en tiempo de compilación. Como consecuencia, en tiempo de ejecución no queda rastro alguno de que alguna vez se utilizaron plantillas. El formato de declaración de dichas plantillas no ofrece una sintaxis que permita centralizar los requerimientos mínimos esperados de los argumentos-tipo utilizados en una especialización, sino que dichos requerimientos están dispersos en toda la definición de la plantilla. Además, la especialización de una plantilla solo abarca la parte de ella que es utilizada, lo que se denomina especialización parcial. Por último, durante la especialización no se hace ninguna verificación de tipos, y esta es relegada al final, sobre el código ya especializado.

Las características citadas traen como consecuencia: (1) El código generado es fuertemente eficiente a costa de una casi-duplicación del código de la plantilla por cada especialización, tendiendo a crecer el código objeto; (2) se permite utilizar plantillas de clase con argumentos-tipo que no soportan todos los requerimientos de la plantilla, llamando sólo a los métodos cuyos requerimientos si cubren los argumentos-tipo (gracias a la especialización parcial); (3) la falta de una sintaxis que permita claramente indicar los requerimientos para los argumentos-tipo de una plantilla dificulta el uso correcto de esta por los programadores; (4) al relegarse la verificación de los tipos luego de la especialización ocasiona que los errores de tipo detectados sean extremadamente difíciles de relacionar con el código que el programador ve, dificultando la depuración de un programa; (5) dado que no quedan rastros de los genéricos en tiempo de compilación, no existe dificultad alguna de utilizar librerías de C++ desde cualquier programa que las soporte, aún si este último no tiene soporte para genéricos.

Java utiliza una técnica denominada erasure (borrado), la cual consiste en una casi translación fuente a fuente, desde un código con genéricos a uno sin genéricos. Los parámetros-tipo son eliminados (luego de verificar la consistencia de su uso) y todas las referencias a ellos son

reemplazadas por los tipos que cumplen con las restricciones (según sean estas upper bound o lower bound). Donde no se indica restricción alguna, el compilador utiliza el tipo Object para realizar los reemplazos. Por último, a todas las conversiones riesgosas (down-cast) se les inserta una operación cast explícita. Así, la declaración de un tipo genérico es compilada una sola vez, como un tipo ordinario. La técnica erasure obliga a tener algunas restricciones importantes: Solo es posible utilizar tipos referencia como argumentos-tipo al usar un genérico; no se permite utilizar los parámetros-tipo en contextos estáticos; no es posible utilizar el operador instanceof para verificar el tipo actual de un parámetro-tipo; no se permite hacer una operación cast con un parámetro-tipo; no se permite la declaración de arreglos de un parámetro-tipo; no se permite la creación de objetos de un parámetro-tipo.

Las características citadas traen como consecuencia: (1) La máquina virtual de Java no tuvo que ser modificada para soportar genéricos, contando así con una compatibilidad completa hacia atrás; (2) dado que toda la información sobre los parámetros-tipo es removida en compilación y dichos tipos son reemplazados mayormente con Object, insertando operaciones cast en cada lugar donde sea necesario, el desempeño final del programa es prácticamente tan buena como si se hubiera programado sin genéricos; (3) dado que los tipos primitivos no son soportados como argumentos-tipo, se hace un uso extensivo del boxing, con el correspondiente costo en desempeño; (4) el uso de la técnica erasure provoca que la representación que se tiene en tiempo de compilación sea distinta a la que se tiene en tiempo de ejecución, dificultando significativamente la aplicación de la reflexión sobre los genéricos.

C# compila sus genéricos como si de cualquier otra clase se tratase, agregando al IL resultante metadata que indique que el tipo compilado es un genérico, así como sus parámetros tipo. Por tanto, existe una versión binaria de cada genérico con una descripción completa del mismo. En ejecución, cuando el programa hace su primera referencia al genérico, el sistema busca si la especialización requerida ya fue instanciada, y si no la hay, pasa al compilador JIT el IL y la metadata del genérico, así como los argumentos-tipo. Con dicha información se genera la versión en código nativo del genérico en el lugar y momento que se necesite. Por tanto, se esta instanciando el genérico, pero a diferencia de C++, esta instanciación se produce en ejecución y no produce la misma duplicidad de código. El compilador JIT realmente no genera una versión diferente por cada tipo construido, sino que sigue la siguiente estrategia: Cuando se utilizan argumentos-tipo por valor, se genera una copia única de código nativo ejecutable para cada combinación distinta de dichos argumentos; cuando se utilizan argumentos-tipo por referencia, se genera solo una copia de código nativo ejecutable, la cual es compartida por todos los tipos por referencia, dado que son estructuralmente idénticas, cambiándose únicamente la tabla virtual utilizada en cada caso. Esta instanciación de los genéricos evita la necesidad de introducir operaciones cast. Por otro lado, los requerimientos esperados por un genérico para sus parámetros-tipo son especificados mediante restricciones. Finalmente, es posible utilizar cualquier tipo de dato como argumento-tipo de un genérico.

Las características citadas traen como consecuencia: (1) C# plantea un balance entre facilidad de reutilización de código (evitando duplicidad cuando no es necesaria) y buen desempeño (creando copias de código nativo ejecutable para los tipos valor, evitando en todo momento la inserción de operaciones cast, y evitando las operaciones de boxing); (2) dado que la instanciación del genérico se hace en tiempo de compilación, los genéricos han requerido una actualización mayor del CLR, lo que ocasiona incompatibilidad con versiones anteriores a esta máquina virtual, esto es, no es posible ejecutar un ensamblaje con genéricos en un computador con CLR 1.0 o 1.1; (3) el contar con información completa de los genéricos en binario hace posible que cualquier genérico pueda ser instanciado dinámicamente utilizando reflexión.

En resumen, C++ busca el mejor desempeño y flexibilidad posible, Java busca una perfecta compatibilidad hacia atrás, y C# busca el mejor balance entre flexibilidad-desempeño y eficiencia (no duplicación de código).

Ventajas y desventajas de la programación genérica

En resumen, la programación genérica tiene como ventajas:

Seguridad, puesto que se consiguen trasladar verificaciones de tipos comúnmente realizadas en tiempo de ejecución al tiempo de compilación, con lo que no se requiere esperar a realizar pruebas de ejecución para verificar que en todos los casos posibles el programa no genera ningún error de tipos. Sin embargo es importante señalar que no todas las verificaciones de tipos pueden trasladarse al tiempo de compilación.

Eficiencia, puesto que al pasar las verificaciones al tiempo de compilación, es posible (dependiendo de la estrategia utilizada) generar programas más eficientes y sin perjuicio de la seguridad de tipos.

Generalidad, con lo cual el código resultante es más expresivo (representa de forma más completa el objetivo que busca tanto el programador que implementa el genérico como el que lo utiliza) y más fácilmente reutilizable.

La programación genérica tiene desventajas. Entre estas están: El aumento de la complejidad del código y la dificultad de comprender todas las implicaciones que la programación genérica ocasiona en el resto de conceptos que soporta un lenguaje. Por ejemplo: ¿Cómo afecta la programación genérica la sub-tipificación?, ¿son los tipos construidos subtipos del genérico o de algún tipo construido común? Estas implicaciones van más allá del alcance del presente documento, pero sirven para ilustrar la amplitud del tema.

El ejemplo pendiente

Solo para no dejar cabos sueltos, quedó pendiente una solución para el método Swap en Java. Como se indicó, el problema con esta implementación es que Java no soporta el paso por referencia. Esto se puede solucionar (aunque de una forma no muy elegante) utilizando una clase wrapper (una clase envolvente) que encapsule un dato del tipo que se desea pasar por referencia para intercambiar. Las clases Integer y Double son ejemplos de clases wrapper. En Código 22 se muestra una clase wrapper genérica, Envoltura, utilizada para compensar la falta de un mecanismo de paso por referencia y poder implementar Swap. Note que gracias al uso de un genérico, el compilador es capaz de generar un error si se tratan de intercambiar datos de tipos distintos, como ocurre en la última llamada a Swap.

```
class Envoltura<T> {
    public T dato;
    public Envoltura(T dato) {
        this.dato = dato;
    }
}

public class Ejml {
    static <T> void Swap(Envoltura<T> a, Envoltura<T> b) {
        T temp = b.dato;
        b.dato = a.dato;
        a.dato = temp;
    }
    public static void main(String[] args) {
        Envoltura<String> x = new Envoltura<String>("Primero");
```

```
    Envoltura<String> y = new Envoltura<String>("Segundo");  
    Swap(x, y);  
    System.out.println("x=" + x.dato + ", y=" + y.dato);  
  
    Envoltura<Integer> z = new Envoltura<Integer>(new Integer(1));  
    Swap(y, z); // Error de compilación  
}  
}
```

Código 23: Definición y uso de Swap para Java.

Archivos, Flujos y Persistencia de Objetos

El objetivo de este capítulo es presentar el enfoque orientado a objetos del manejo de archivos, y flujos en general, que ofrecen los lenguajes Java y C#, dentro de un conjunto de casos frecuentes de uso.

Archivos y Flujos

La unidad mínima de representación de un valor en una computadora es el bit. La unidad mínima de procesamiento de datos es el byte. Uno o más bytes pueden representar un carácter (un dígito, una letra o un símbolo) o un número (integral o de punto flotante). Ésta es la plataforma base sobre la cual los lenguajes de programación dan un valor agregado a las capacidades brindadas al programador para manejar estructuras de datos. Una descripción clásica de la forma de organización de la información por los programas es:

“Un conjunto de bytes con significado agregado y que se operan como una unidad se le denomina campo. Un conjunto de campos con significado agregado y que se operan como una unidad se denomina registro. Un conjunto de registros pueden ser almacenados de forma persistente en un archivo. Un conjunto de archivos de registros forman una base de datos.”

El concepto de registro está estrechamente ligado al de estructura de un archivo: El registro es una estructura de datos que representa la unidad de lectura y escritura sobre un archivo. Algunos lenguajes de programación requieren la especificación del tipo de registro al manipular un archivo (por ejemplo Pascal), mientras que otros no (por ejemplo C, Java y C#), visualizando a los archivos como un conjunto de bytes a los que el programa da significado durante la misma lectura.

Los datos almacenados en memoria volátil (el caso de la RAM) tienen un ciclo de vida limitado al del programa que los crea. Estos se denominan datos temporales. Los datos almacenados en memoria no-volátil (el caso del disco duro) tienen un ciclo de vida independiente al de la instancia del programa que los crea o manipula. Estos datos se denominan datos persistentes. En el presente contexto, el término memoria refiere a cualquier dispositivo físico de almacenamiento o parte de él.

Existen dos formas de acceder a los datos persistentes: de forma **secuencial** y **aleatoria**. Dicho acceso puede tener un tipo de permiso asignado: Sólo lectura, sólo escritura o lectura-escritura. Si es factible que más de un programa acceda a dichos datos, pueden establecerse permisos para compartir dichos datos. Las formas y permisos dependen de las capacidades de la memoria que almacena dichos datos.

En general, el acceso a un dato involucra el traspaso de éste de una memoria a otra. Esta transferencia puede hacerse físicamente por bloques de bytes o de byte en byte. Esta transferencia puede involucrar una sola operación de copia (todos los datos transferidos están disponibles a la vez) o varias operaciones secuenciales. En este último caso, los datos están disponibles conforme van llegando. Es aquí donde el concepto de flujo es útil.

Un **flujo** es una secuencia de bytes que son accedidos en el mismo orden en que fueron creados. Un flujo opera de manera similar a una cola: Los datos son leídos desde un flujo en el mismo orden en que fueron escritos en él. Dado que cualquier memoria permite por lo menos un acceso secuencial a sus datos, ya sea para lectura o escritura, ya sea que los datos estén disponibles a la vez o secuencialmente, éstas pueden trabajarse siempre como flujos. Si bien por definición un flujo permite un acceso secuencial a los datos de la memoria que maneja, también puede permitir un acceso aleatorio, dependiendo del tipo de memoria.

Objetos Persistentes

En la POO, el concepto de objeto reemplaza sobremanera al de registro. Un objeto que es almacenado en memoria persistente se le conoce como **objeto persistente**. Una forma de lograr esta persistencia es mediante una técnica llamada **serialización**.

Se dice que un objeto es serializado cuando es escrito hacia un flujo de una forma tal que pueda ser leído luego y reconstruido. El proceso de lectura y reconstrucción de un objeto se conoce como **deserialización**. La serialización implica mucho más que sólo escribir los datos de un objeto en un orden y leerlos en el mismo orden. Existen dos problemas principales en la serialización de un objeto:

- Determinar quién es responsable de serializar y deserealizar un objeto: El mismo objeto o el mecanismo de serialización, ya sea éste implementado a nivel del lenguaje de programación o con una librería.
- Cómo manejar los diagramas de clases.

Encargar la responsabilidad de serializar/deserealizar un objeto al mismo objeto tiene la ventaja de darle al programador de la clase de dicho objeto el control completo del proceso. La clara contraparte es el mayor trabajo de programación requerido y por consiguiente, el aumento de la tasa de error. Encargar la responsabilidad al mecanismo de serialización tiene la ventaja de simplificar la programación, pero la desventaja de requerirse un mecanismo de reflexión de apoyo, dado que el mecanismo de serialización deberá poder reconocer en tiempo de ejecución a qué tipo de objeto corresponde los datos leídos y crearlo, todo de manera automática.

Los diagramas de clases se forman cuando los datos miembros de un objeto refieren a otros objetos, formando un diagrama de conexiones entre objetos. Cuando uno de los objetos de este diagrama debe serializarse, también deberá serializarse todos los objetos a los que refiere, directa o indirectamente. Esto no sólo implica un rastreo recursivo de todas las referencias del objeto serializado, sino además la resolución de conflictos tales como: ¿Qué sucede cuando al rastrear dicho diagrama durante la serialización se llega a un mismo objeto más de una vez?, ¿Cómo reconocer que ya se serializó un objeto previamente en el mismo flujo? Para el manejo de los

diagramas de objetos, como es claro, el delegar el trabajo de serialización al programador puede aumentar enormemente la complejidad del código final.

Manejo desde C#

Descripción General de las Capacidades

C# permite el manejo de archivos de texto y binarios, secuenciales y aleatorios. Dichos archivos son manejados como flujos. Cuando un archivo es abierto, C# crea un objeto y lo relaciona con el flujo de dicho archivo.

Existen tres flujos estándar con objetos relacionados que son creados automáticamente para todo programa en C#:

- El flujo de entrada estándar, mediante la referencia `System.Console.In`.
- El flujo de salida estándar, mediante la referencia `System.Console.Out`.
- El flujo de error estándar, mediante la referencia `System.Console.Error`.

Los métodos de la clase `Console` de lectura tales como `Read` y `ReadLine`, y de escritura tales como `Write` y `WriteLine` hacen uso de `Console.In` y `Console.Out` respectivamente.

Todas las clases para el manejo de las operaciones de entrada y salida se encuentran en el espacio de nombres `System.IO`. Algunas de las más usuales se muestran en la Figura 10 - 1.

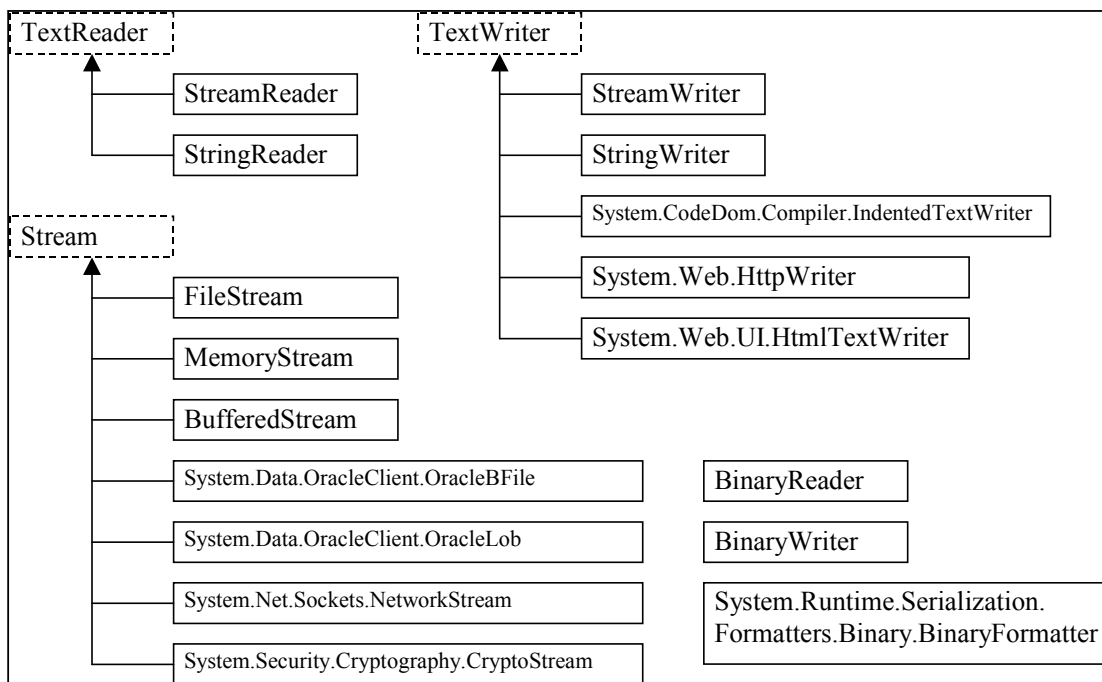


Figura 10 - 1 Las clases para manejo de flujos en C#.

Las clases `TextReader`, `TextWriter` y `Stream` son clases abstractas para la lectura y escritura de archivos de texto y binarios. Los objetos referenciados por `Console.In` y `Console.Out` son de tipo `TextReader` y `TextWriter` respectivamente.

Las clases `StreamReader` y `StringReader` permiten manejar como flujos de salida archivos de texto y cadenas de caracteres respectivamente. Su contraparte para flujos de entrada son `StreamWriter` y `StringWriter`.

Las clases `FileStream`, `MemoryStream` y `BufferedStream` son para manejo de flujos binarios, para lectura y escritura. A dichos flujos se pueden serializar objetos, para lo que se utiliza la clase `BinaryFormatter`. Esta clase puede serializar y deserializar cualquier objeto, junto con su diagrama de objetos, marcado como serializable. Dicha marca se realiza mediante el atributo estándar serializable, como se verá más adelante.

Finalmente, `C#` permite el acceso al sistema de archivos del sistema operativo, mediante las clases `File` y `Directory`, como se verá en la siguiente sección.

Acceso al Sistema de Archivos

Para el manejo del sistema de archivos se provee de dos clases:

- **File**, para manipular archivos
- **Directory**, para manipular directorios

Ambas clases proveen métodos estáticos para manejar archivos y directorios. La Tabla 10 - 1 muestra los métodos de `File` más comunes.

Tabla 10 - 1 Métodos de la clase `File` de .NET.

Manipulación de archivos	
Nombre	Descripción
<code>Copy</code>	Copia un archivo a un nuevo archivo.
<code>Move</code>	Mueve un archivo.
<code>Delete</code>	Elimina un archivo.
Información de un archivo	
Nombre	Descripción
<code>Exists</code>	Averigua si un archivo existe.
<code>GetCreationTime</code>	Retorna la fecha en que se creó el archivo.
<code>GetLastAccessTime</code>	Retorna la fecha en que se accedió por última vez al archivo.
<code>GetLastWriteTime</code>	Retorna la fecha en que se escribió por última vez en el archivo.
Archivos de texto	
Nombre	Descripción

CreateText	Crea un archivo de texto y retorna un objeto StreamWriter asociado.
OpenText	Apertura un archivo de texto y retorna un objeto StreamReader asociado.
AppendedText	Retorna un objeto StreamWriter para agregar datos a un archivo existente o crea uno nuevo.
Archivos binarios	
Nombre	Descripción
Create	Crea un archivo y retorna un objeto FileStream asociado.
Open	Abre un archivo, bajo distintos modos, y retorna un objeto FileStream asociado.
OpenRead	Abre un archivo existente para lectura.
OpenWrite	Abre un archivo existente para escritura.

Las Tabla 10 - 2 muestra los métodos de Directory más comunes.

Tabla 10 - 2 Métodos de la clase Directory de .NET.

Manipulación de directorios	
Nombre	Descripción
CreateDirectory	Crea un nuevo directorio.
Move	Mueve un directorio.
Delete	Elimina un directorio.
Información de un directorio	
Nombre	Descripción
Exists	Averigua si un directorio existe.
GetCreationTime	Retorna la fecha en que se creó el directorio.

GetLastAccessTime	Retorna la fecha en que se accedió por última vez al directorio.
GetLastWriteTime	Retorna la fecha en que se escribió por última vez en el directorio.

Manejo de Consola

La entrada, salida y error estándar se manejan mediante las propiedades estáticas públicas de sólo lectura In, Out y Error de la clase Console, las que retornan referencias de tipo TextReader, TextWriter y TextWriter respectivamente. Luego, las siguientes sentencias son equivalentes:

Console.Write(dato);	equivale a	Console.Out.Write(dato);
Console.WriteLine(dato);	equivale a	Console.Out.WriteLine(dato);
dato = Console.Read();	equivale a	dato = Console.In.Read();
dato = Console.ReadLine();	equivale a	dato = Console.In.ReadLine();

La clase abstracta TextReader permite la lectura de caracteres, uno a uno o por líneas, desde un flujo de texto. La clase abstracta TextWriter permite la escritura de caracteres, uno a uno o por líneas, hacia un flujo de texto. Los objetos referidos por las propiedades de Console realmente son de un tipo concreto que hereda de estas clases. Esto puede comprobarse fácilmente con el siguiente código:

```
Type tipo = Console.In.GetType();
Type tipoBase = tipo.BaseType;
Console.WriteLine("Console.In es de tipo = " + tipo.Name);
Console.WriteLine("El que hereda del tipo = " + tipoBase.Name);
```

El nombre del tipo que se muestra en la ventana de consola es SyncTextReader, el cual es un tipo interno (no público) del espacio de nombres System.IO.

El siguiente programa es un ejemplo del uso de la consola leyendo carácter por carácter.

```
using System;

class Consola1 {
    public static void Main(string[] args) {
        Console.Write( "Ingrese el texto a Leer." );
        Console.WriteLine( " Finalizar con la combinacion CTRL+Z." );
        while(true) {
            int Entero = Console.In.Read();
            if( Entero == -1 )
                break;
            char Caracter = (char)Entero;
            Console.Write( "Leido Entero=" + Entero );
            Console.WriteLine( ", correspondiente al carácter=" + Caracter );
        }
        Console.WriteLine( "Fin del ingreso" );
    }
}
```

La Figura 10 - 2 muestra el resultado de una ejecución de este programa.

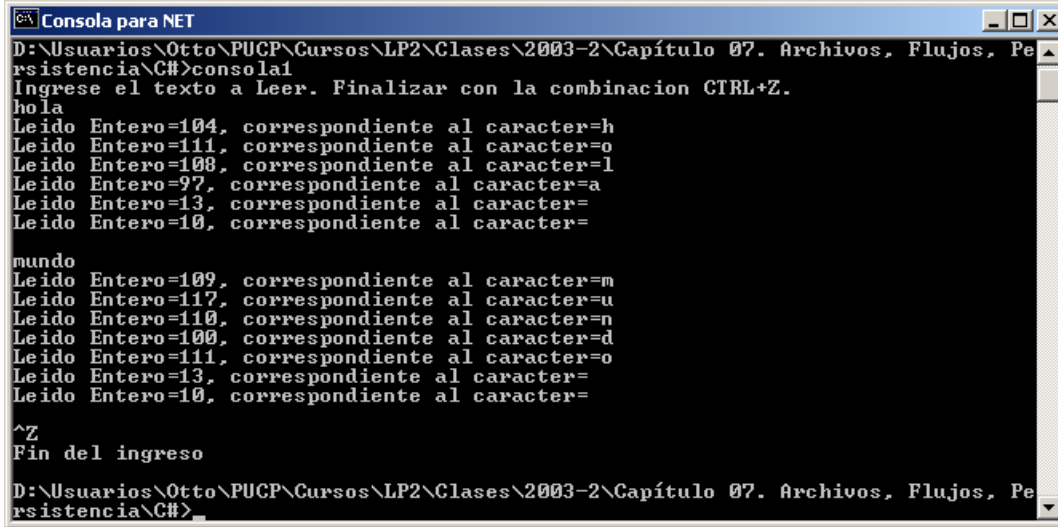
El siguiente programa es un ejemplo del uso de la consola leyendo línea por línea. Note que cuando, al inicio de la línea, el usuario ingresa CTRL+Z, el método ReadLine reconoce esto como un indicador de fin de ingreso, retornando null.

```
using System;

class Consola2 {
    public static void Main(string[] args) {
        Console.WriteLine( "Ingrese una secuencia de enteros." );
        Console.WriteLine( "Finalice el ingreso con '*'." );
        int Contador = 0;
        int Suma = 0;
        while(true) {
```

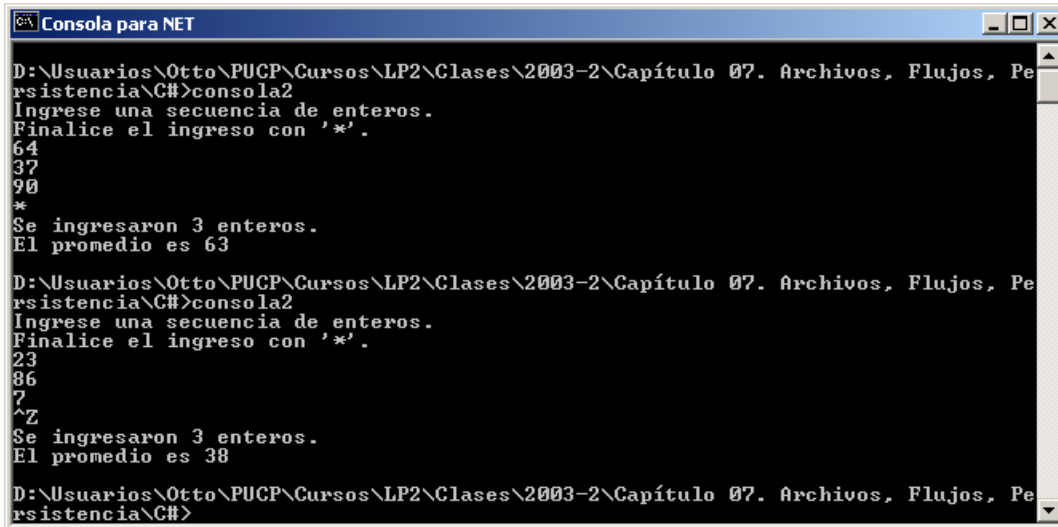
```
        string Linea = Console.In.ReadLine();
        if( Linea == null )
            break;
        if( Linea[0] == '*' )
            break;
        Suma += Int32.Parse(Linea);
        Contador++;
    }
    Console.WriteLine( "Se ingresaron " + Contador + " enteros." );
    Console.WriteLine( "El promedio es " + (Suma / Contador) );
}
```

La Figura 10 - 3 muestra el resultado de una ejecución de este programa.



```
Consola para NET
D:\Usuarios\Otto\PUCP\Cursos\LP2\Clases\2003-2\Capítulo 07. Archivos, Flujos, Pe
rsistencia\C#>consola1
Ingrese el texto a Leer. Finalizar con la combinacion CTRL+Z.
hola
Leido Entero=104, correspondiente al caracter=h
Leido Entero=111, correspondiente al caracter=o
Leido Entero=108, correspondiente al caracter=l
Leido Entero=97, correspondiente al caracter=a
Leido Entero=13, correspondiente al caracter=
Leido Entero=10, correspondiente al caracter=
mundo
Leido Entero=109, correspondiente al caracter=m
Leido Entero=117, correspondiente al caracter=u
Leido Entero=110, correspondiente al caracter=n
Leido Entero=100, correspondiente al caracter=d
Leido Entero=111, correspondiente al caracter=o
Leido Entero=13, correspondiente al caracter=
Leido Entero=10, correspondiente al caracter=
^Z
Fin del ingreso
D:\Usuarios\Otto\PUCP\Cursos\LP2\Clases\2003-2\Capítulo 07. Archivos, Flujos, Pe
rsistencia\C#>
```

Figura 10 - 2 Manejo de consola en C#: Ejecución del programa 1.



```
Consola para NET
D:\Usuarios\Otto\PUCP\Cursos\LP2\Clases\2003-2\Capítulo 07. Archivos, Flujos, Pe
rsistencia\C#>consola2
Ingrese una secuencia de enteros.
Finalice el ingreso con '*'.
64
37
90
*
Se ingresaron 3 enteros.
El promedio es 63
D:\Usuarios\Otto\PUCP\Cursos\LP2\Clases\2003-2\Capítulo 07. Archivos, Flujos, Pe
rsistencia\C#>consola2
Ingrese una secuencia de enteros.
Finalice el ingreso con '*'.
23
86
7
^Z
Se ingresaron 3 enteros.
El promedio es 38
D:\Usuarios\Otto\PUCP\Cursos\LP2\Clases\2003-2\Capítulo 07. Archivos, Flujos, Pe
rsistencia\C#>
```

Figura 10 - 3 Manejo de consola en C#: Ejecución del programa 2.

El siguiente programa es un ejemplo de la interpretación de la lectura, cuando se ingresa más de un dato a la vez, en cuyo caso se puede partir dicha lectura mediante el método Split de la clase string, pasándole como parámetro un arreglo de caracteres que deberán ser utilizados como delimitadores.

```
using System;

class Consola3 {
    public static void Main(string[] args) {
        Console.WriteLine( "Ingrese una secuencia de palabras." );
        string Linea = Console.In.ReadLine();
        char[] separadores = { ' ', ',', '.', ':' };
        string[] Tokens = Linea.Split( separadores );
        for(int i = 0; i < Tokens.Length; i++)
            Console.WriteLine("Token " + (i + 1) + " = " + Tokens[i]);
    }
}
```

La Figura 10 - 4 muestra una corrida de este programa.

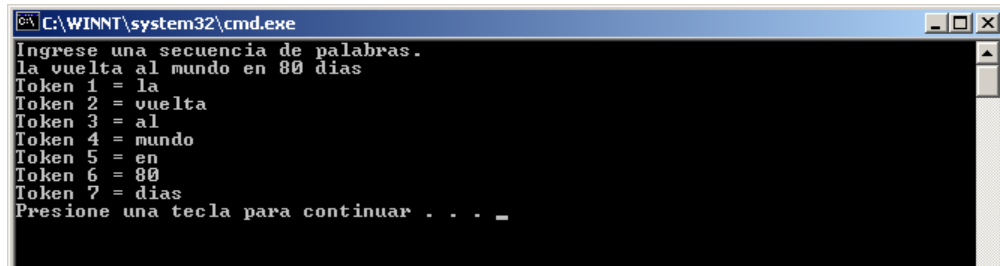


Figura 10 - 4 Manejo de consola en C#: Ejecución del programa 3.

Manejo de Archivos de Texto

Los archivos de texto se manejan de manera similar a la consola. El siguiente ejemplo muestra la creación y lectura de un archivo de texto.

```
using System;
using System.IO;

class ArchivoTexto {
    public static void Main( string[] args ) {
        FileStream Archivo;
        Archivo = new FileStream( args[0], FileMode.Create, FileAccess.Write );
        StreamWriter Escriitor = new StreamWriter( Archivo );

        Console.WriteLine( "Ingrese el texto a almacenar: " );
        while( true ) {
            string Linea = Console.ReadLine();
            if( Linea == null )
                break;
            Escriitor.WriteLine( Linea );
        }
        Escriitor.Close();
        Archivo.Close();

        Console.WriteLine( "Leyendo el archivo creado." );
        Archivo = new FileStream( args[0], FileMode.Open, FileAccess.Read );
        StreamReader Lector = new StreamReader( Archivo );
        while( true ) {
            string Linea = Lector.ReadLine();
            if( Linea == null )
                break;
            Console.WriteLine( Linea );
        }
        Lector.Close();
        Archivo.Close();
    }
}
```

Para crear un archivo se crea un flujo desde un archivo utilizando la clase `FileStream`. Esta clase provee métodos para leer y escribir únicamente bytes, sin ninguna interpretación. Para escribir o leer los datos de este flujo como texto, se crea otro flujo utilizando la clase `StringWriter` y

StreamReader respectivamente. Ambas clases ofrecen además un constructor que recibe directamente el nombre del archivo de texto, de forma que no sea necesario crear el objeto FileStream manualmente.

Manejo de Archivos Binarios Secuencialmente

Los archivos binarios secuenciales son comúnmente manejados mediante la técnica de serialización. En .NET, la serialización se realiza utilizando una clase utilitaria llamada formateador (que tiene la misma función que el formateador en .NET Remoting) que implemente los métodos Serialize y Deserialize. El único formateador de este tipo en la versión actual de la librería de .NET es BinaryFormatter.

La clase BinaryFormatter se basa en un atributo para determinar si los objetos de una clase deben ser serializados: Serializable. Un atributo en .NET es una metadata relacionada a un elemento de programación, como la definición de un tipo de dato, la declaración de un método o de un dato miembro, etc. El siguiente ejemplo muestra el uso del atributo Serializable y de la clase BinaryFormatter.

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;

[Serializable]
class Direccion {
    int numero;
    string calle;
    string distrito;
    public Direccion(int numero, string calle, string distrito) {
        this.numero = numero;
        this.calle = calle;
        this.distrito = distrito;
    }
    public override string ToString() {
        return calle + " " + numero + ", " + distrito;
    }
}

[Serializable]
class Persona {
    private string nombre;
    private int edad;
    private Direccion dir;
    public Persona(string nombre, int edad, Direccion dir) {
        this.nombre = nombre;
        this.edad = edad;
        this.dir = dir;
    }
    public override string ToString() {
        return "Sr(a). " + nombre + ", " + edad + " años, direccion = " + dir;
    }
}

class ArchivoBinarioSecuencial {
    public static void Main(string[] args) {
        if(args.Length < 2) {
            Console.WriteLine("Error en argumentos.");
            return;
        }
        if(args[0] == "/e") {
            FileStream Output = new FileStream(args[1], FileMode.Create,
                FileAccess.Write);
            BinaryFormatter Formateador = new BinaryFormatter();
            Persona[] ListaPersonas = {
                new Persona("Jose", 25, new Direccion(123, "Jose Leal", "San
Juan")),
```

```
        new Persona("Mara", 26, new Direccion(456, "Carrión", "Barranco")),
        new Persona("Ana", 35, new Direccion(789, "Fresnos", "Lince"))
    };
    foreach(Persona p in ListaPersonas)
        Formateador.Serialize(Output, p);
    Output.Close();
} else if(args[0] == "/l") {
    FileStream Input = new FileStream(args[1], FileMode.Open,
        FileAccess.Read);
    BinaryFormatter Formateador = new BinaryFormatter();
    while(true) {
        Persona p;
        try { p = (Persona)Formateador.Deserialize(Input); }
        catch(SerializationException){ break; }
        Console.WriteLine("Persona : " + p);
    }
    Input.Close();
}
}
```

El programa utiliza los argumentos de la línea de comandos para determinar si debe crear un archivo o si debe de leerlo.

Al crear el archivo se inicializa un arreglo de objetos Persona, los que serializan utilizando el método Serialize de la clase BinaryFormatter. Tome en cuenta que cuando se serializa un objeto, se serializa también todos los objetos a los que hace referencia directa o indirectamente. Si se intenta serializar un objeto no marcado con el atributo Serializable, el método Serialize dispara una excepción.

En forma similar al leer el archivo, se utiliza el método Deserialize, el cual retorna una referencia de tipo object a la que debe aplicársele una operación cast para obtener una referencia al tipo real. Hay dos cosas que pueden fallar durante una deserialización: Que se haya llegado al final del archivo sin haberse podido leer todos los datos de un objeto o bien que la operación de cast falle, es decir, se leyó un objeto diferente al que se esperaba. En ambos casos se produce una excepción.

Es importante recordar que la serialización no requiere que el flujo reciba objetos del mismo tipo. Se puede serializar a un mismo flujo objetos de distintos tipos, en cuyo caso deberá asegurarse que el proceso de deserialización sea realizado en el mismo orden que el proceso de serialización sobre el flujo tratado.

Manejo de Archivos Binarios Aleatoriamente

.NET no fuerza un formato sobre los archivos binarios, por lo que tampoco provee de mecanismos para acceder aleatoriamente a los objetos serializados hacia un flujo de la misma forma como lo hacen otros lenguajes como Pascal. Es importante tomar en cuenta que al serializar dos objetos del mismo tipo no necesariamente se escriben el mismo número de bytes en el flujo. Un ejemplo claro es el de los objetos string. Si serializamos dos objetos de este tipo podrán ocupar espacios de diferente tamaño dentro del flujo. Debido a esto, tampoco es sencillo colocar el apuntador del archivo al inicio del n-ésimo objeto y leerlo. Si deseamos hacer esto, es necesario garantizar que durante la serialización se escriba por cada objeto siempre la misma cantidad de bytes, o implementar manualmente algún otro mecanismo para realizar un acceso aleatorio.

El siguiente programa serializa manualmente objetos de la clase Combo a un archivo binario, garantizando que los datos miembros de cada objeto siempre ocupen el mismo espacio, de manera que nos podamos mover por el archivo y leer directamente cualquiera de los objetos del flujo.

```
using System;
using System.IO;
using System.Runtime.Serialization;

class Combo {
    public const int LONG_NOMBRE = 10;
    public const int LONG_DESCRIPCION = 30;
    public const int LONG_CARACTER = 1; // longitud de un char escrito en un archivo TXT
    public const int LONG_REGISTRO =
        (LONG_NOMBRE + LONG_DESCRIPCION) * LONG_CARACTER + sizeof(double);

    private char[] Nombre = new char[LONG_NOMBRE];
    private char[] Descripcion = new char[LONG_DESCRIPCION];
    private double Precio;

    public Combo(string Nombre, string Descripcion, double Precio) {
        Copiar(this.Nombre, Nombre);
        Copiar(this.Descripcion, Descripcion);
        this.Precio = Precio;
    }

    private static void Copiar(char[] Destino, string Origen) {
        if(Origen.Length < Destino.Length) {
            Origen = Origen.PadRight(Destino.Length);
            Origen.CopyTo(0, Destino, 0, Destino.Length);
        } else
            Origen.CopyTo(0, Destino, 0, Destino.Length);
    }

    public override string ToString() {
        string Nombre = new string(this.Nombre);
        string Descripcion = new string(this.Descripcion);
        return "Combo " + Nombre + ": " + Descripcion + ", a S/. " + Precio;
    }

    public void SetPrecio(double Precio) {
        this.Precio = Precio;
    }

    public void Serialize(BinaryWriter Output) {
        Output.Write(Nombre);
        Output.Write(Descripcion);
        Output.Write(Precio);
    }

    public static Combo Deserialize(BinaryReader Input) {
        char[] Nombre = Input.ReadChars(LONG_NOMBRE);
        char[] Descripcion = Input.ReadChars(LONG_DESCRIPCION);
        double Precio = Input.ReadDouble();
        return new Combo(new string(Nombre), new string(Descripcion), Precio);
    }
}

class ArchivoBinarioAleatorio {
    public static void Main(string[] args) {
        if(args.Length != 1) {
            Console.WriteLine("Error en parametros.");
            return;
        }

        // Creo el archivo de salida
        FileStream Flujo = new FileStream(args[0], FileMode.Create,
            FileAccess.Write);
        BinaryWriter Salida = new BinaryWriter(Flujo);
        Combo[] Combos = {
            new Combo("Tradicional", "Bembos trad. medium + papas chicas +
                gaseosa chica", 10.9),
            new Combo("Peruano", "Bembos peruana tradicional + papas chicas +
                helado", 12.9),
            new Combo("Frances", "Bembos francesa medium + ensalada + copa de " +
                "vino", 15.9)
        };
        foreach(Combo C in Combos)
            C.Serialize(Salida);
        Console.WriteLine("Tamaño del archivo creado = " + Flujo.Length);
        Salida.Close();
        Flujo.Close();
    }
}
```

```
// Leo el archivo creado
Flujo = new FileStream(args[0], FileMode.Open, FileAccess.Read);
BinaryReader Entrada = new BinaryReader(Flujo);
while(true) {
    try {
        Combo C = Combo.Deserialize(Entrada);
        Console.WriteLine("Leido = " + C);
    }
    catch(EndOfStreamException) {
        break;
    }
}
Entrada.Close();
Flujo.Close();

// Abro nuevamente para lectura/escritura
Flujo = new FileStream(args[0], FileMode.Open, FileAccess.ReadWrite);
Entrada = new BinaryReader(Flujo);
Salida = new BinaryWriter(Flujo);
bool Fin = false;
while(!Fin) {
    Console.WriteLine("Seleccione una opcion:");
    Console.WriteLine("1. Leer un registro");
    Console.WriteLine("2. Modificar un Registro");
    Console.WriteLine("3. Terminar");
    try {
        int Opcion;
        Opcion = Int32.Parse(Console.In.ReadLine());
        switch(Opcion) {
            case 1:
                Opcion1(Flujo, Entrada);
                break;
            case 2:
                Opcion2(Flujo, Entrada, Salida);
                break;
            case 3:
                Fin = true;
                break;
            default:
                Console.WriteLine("Opción inválida.");
                break;
        }
    }
    catch(FormatException) {
        Console.WriteLine("Debe ingresar un número.");
        continue;
    }
}
Entrada.Close();
Salida.Close();
Flujo.Close();

}
public static void Opcion1(Stream Flujo, BinaryReader Entrada) {
    Console.WriteLine("Número de registro a leer:");
    int Registro = Int32.Parse(Console.In.ReadLine());
    int TotalRegistros = (int)Flujo.Length / Combo.LONG_REGISTRO;
    if(Registro >= TotalRegistros)
        Console.WriteLine("Número de registro inválido.");
    else {
        Flujo.Position = Registro * Combo.LONG_REGISTRO;
        Console.WriteLine("Puntero del archivo en la pos:" + Flujo.Position);
        Combo C = Combo.Deserialize(Entrada);
        Console.WriteLine("Leido = " + C);
    }
}

public static void Opcion2(Stream Flujo, BinaryReader Entrada, BinaryWriter Salida){
    Console.WriteLine("Número de registro a leer:");
    int Registro = Int32.Parse(Console.In.ReadLine());
    int TotalRegistros = (int)Flujo.Length / Combo.LONG_REGISTRO;
    if(Registro >= TotalRegistros)
        Console.WriteLine("Número de registro inválido.");
```



```
        else {
            Flujo.Position = Registro * Combo.LONG_REGISTRO;
            Combo C = Combo.Deserialize(Entrada);
            Console.WriteLine("Leído = " + C);
            Console.WriteLine("Nuevo precio:");
            C.SetPrecio(Double.Parse(Console.ReadLine()));
            Flujo.Position = Registro * Combo.LONG_REGISTRO;
            C.Serialize(Salida);
        }
    }
}
```

La clase Combo utiliza arreglos de caracteres en lugar de objetos string de manera que se controle el tamaño de los datos escritos y leídos desde el flujo. Es interesante notar el hecho de que se utiliza, para el cálculo del tamaño de los datos escritos al flujo, el tamaño de un carácter como de 1 byte, cuando realmente un dato tipo char en .NET ocupa 2 bytes. Esto se debe a la forma al sistema de codificación de caracteres utilizados por defecto por los flujos creados. Bajo el sistema de codificación de caracteres por defecto, se escribe en el flujo la representación ASCII, de 1 byte por char, de cada carácter UNICODE utilizada por los tipos char, de 2 bytes por carácter.

Para moverse por el archivo de manera aleatoria, se modifica el valor de la propiedad pública de lectura y escritura **Position** de la clase FileStream. Esta clase también ofrece un método alternativo que provee de mayor funcionalidad para desplazarse por el archivo.

Manejo desde Java

Flujos

Java maneja las entradas y salidas de datos, que no sean mediante componentes GUI, mediante **flujos o streams**. Un flujo es una secuencia de datos que son comúnmente procesados o leídos en el mismo orden en que fueron colocados o escritos en dicho flujo. Un flujo puede ser de entrada o de salida y si bien comúnmente son procesados secuencialmente, según el tipo del flujo, también es posible procesarlos aleatoriamente.

El concepto de flujo es implementado por Java para manejar, de manera uniforme:

- La entrada, salida y error estándar.
- La lectura y escritura de archivos.
- La lectura y escritura de objetos en buffers de memoria.

Cada flujo es manejado por un objeto de alguna de las clases definidas en el paquete java.io para este fin. La Figura 10 - 5 muestra la jerarquía de las principales clases de manejo de flujos.

A continuación una breve descripción de la funcionalidad de cada una de estas clases:

- **InputStream / OutputStream:** Clases base abstractas para lectura / escritura de flujos de bytes. Permite leer/escribir bytes y arreglos de bytes.
- **ByteArrayInputStream / ByteArrayOutputStream:** Permiten el manejo de un arreglo de bytes como un flujo. ByteArrayInputStream se crea en base a un arreglo de bytes externo. ByteArrayOutputStream crea su propio arreglo de bytes.
- **FileInputStream / FileOutputStream:** Permiten el manejo de archivos como flujos. Ambos se crean en base a un archivo.

- **FilterInputStream / FilterOutputStream:** Clases base para todos los filtros de flujos de bytes. FilterInputStream se crea en base a un objeto InputStream. FilterOutputStream se crea en base a un objeto OutputStream.
- **BufferedInputStream / BufferedOutputStream:** Proveen buferización en memoria de entrada / salida, mejorando el desempeño de ésta. Esto es útil cuando se trabaja con un medio de acceso lento con respecto a la memoria directa (RAM), por ejemplo un archivo. BufferedInputStream se crea en base a un objeto InputStream. BufferedOutputStream se crea en base a un objeto OutputStream.
- **DataInputStream / DataOutputStream:** Permiten leer / escribir tipos de datos primitivos de Java de un flujo de bytes en forma independiente a la plataforma. DataInputStream se crea en base a un objeto InputStream. DataOutputStream se crea en base a un objeto OutputStream.
- **PushbackInputStream:** Permite retornar bytes leídos de un flujo de entrada, para volver a ser leídos posteriormente. Esto es útil cuando se realiza reconocimiento de elementos en donde existen casos donde sólo es posible determinar que un elemento fue completamente leído cuando se comenzó a leer otro, por lo que, lo más conveniente es retornar la última lectura al flujo para luego volver a iniciar el reconocimiento del nuevo elemento. Se crea en base a un objeto InputStream.
- **PrintStream:** Agrega funcionalidad a OutputStream permitiendo serializar datos primitivos y objetos en base a sus representaciones como texto. Cada carácter del texto enviado se convierte a su representación estándar en bytes, en base a la codificación por defecto de caracteres de la plataforma actual. Se crea en base a un objeto OutputStream.
- **ObjectInputStream / ObjectOutputStream:** Agregan funcionalidad a InputStream / OutputStream permitiendo serializar / deserializar datos primitivos y objetos. Los objetos deben implementar la interface java.io.Serializable para poder ser serializados. Dicha interfaz no define ningún método, sólo sirve como forma de marcar a un objeto como serializable. Si un objeto referencia a otros, éstos también son serializados. ObjectInputStream se crea en base a un objeto InputStream. ObjectOutputStream se crea en base a un objeto OutputStream.
- **PipedInputStream / PipedOutputStream:** Permiten leer / escribir bytes y arreglos de bytes a pipes (tuberías). Un pipe es un objeto de comunicación unidireccional entre hilos. Un objeto PipedInputStream es conectado a un objeto PipedOutputStream. El primero es utilizado por un hilo, el cual escribe bytes que son leídos por otro hilo utilizando el segundo objeto.
- **SequenceInputStream:** Permite la concatenación lógica de varios objetos InputStream de forma que cuando se llega al final de la lectura de los datos de uno de ellos, se continúa leyendo en el siguiente, de forma transparente. Se crea en base a 2 o más objetos InputStream.

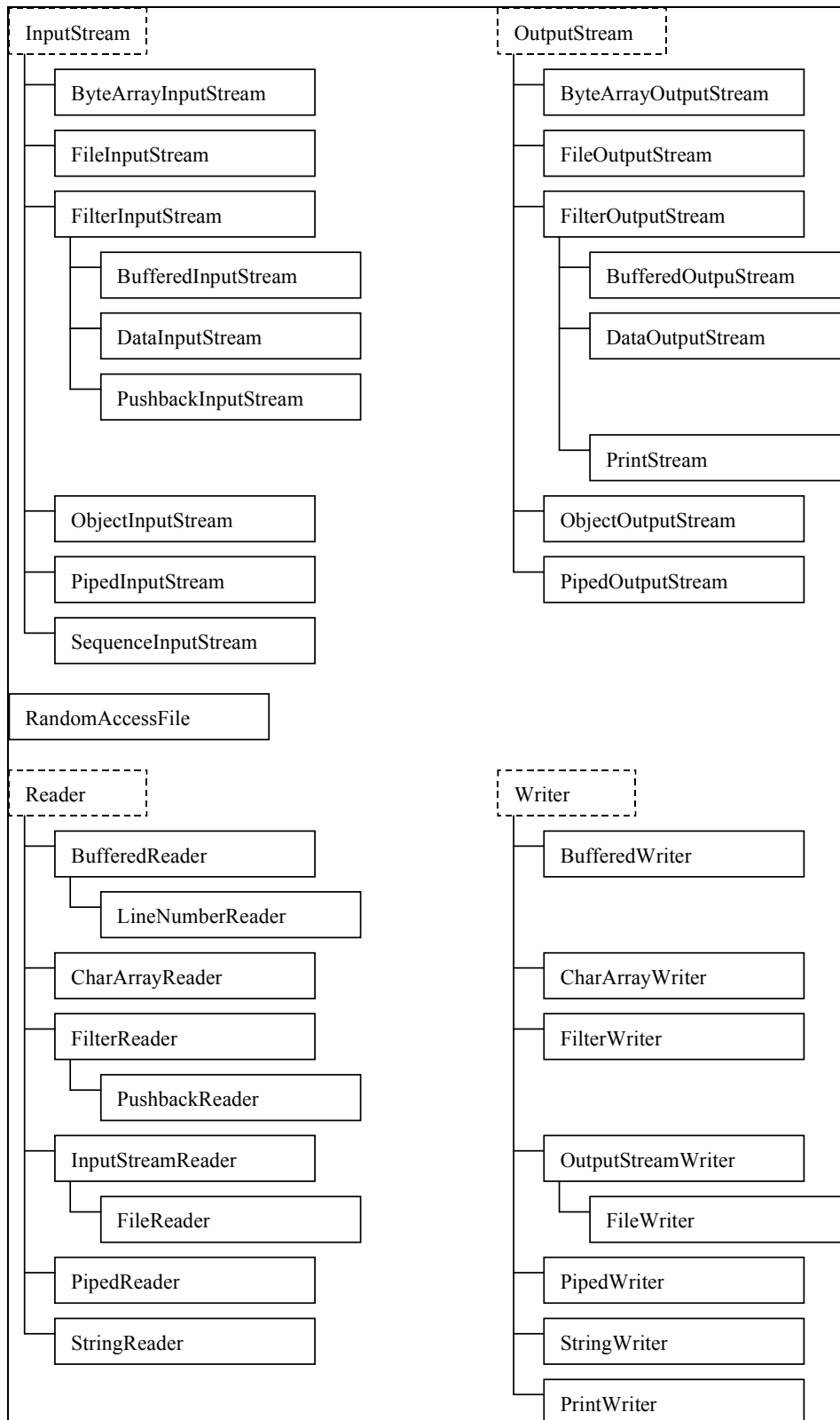


Figura 10 - 5 Las clases para manejo de flujos en Java.

- **RandomAccessFile:** Permite manejar un archivo como un flujo secuencial o aleatorio. Se crea en base a un archivo especificándose el tipo de acceso que se requiere: de lectura, de escritura o ambos. Permite la lectura y escritura de bytes, arreglos de bytes, datos primitivos y cadenas de texto en un formato especial.
- **Reader / Writer:** Clases base abstractas para lectura / escritura de flujos de caracteres. Permite leer / escribir caracteres y arreglos de caracteres. Writer permite adicionalmente escribir cadenas de texto (String).
- **BufferedReader / BufferedWriter:** Provee buferización en memoria de la entrada, mejorando el desempeño de ésta. Esto es útil cuando se trabaja con un medio de acceso lento con respecto a la memoria directa (RAM), por ejemplo un archivo. Agregan la capacidad de leer líneas. BufferedReader se crea en base a un objeto Reader. BufferedWriter se crea en base a un objeto Writer.
- **LineNumberReader:** Ofrece la misma funcionalidad que BufferedReader agregando métodos para obtener y establecer el número de línea actual de lectura.
- **CharArrayReader / CharArrayWriter:** Permite el manejo de un arreglo de caracteres como un flujo. CharArrayReader se crea en base a un arreglo de caracteres externo. CharArrayWriter crea su propio arreglo de caracteres.
- **FilterReader / FilterWriter:** Clase abstracta base para todos los filtros de flujos de caracteres. FilterReader se crea en base a un objeto Reader. FilterWriter se crea en base a un objeto Writer.
- **PushbackReader:** Permite retornar caracteres leídos de un flujo de entrada, para volver a ser leídos posteriormente. Esto es útil cuando se realiza reconocimiento de elementos en un texto (como por ejemplo, un intérprete de ecuaciones aritméticas ingresadas como texto) en donde existen casos donde sólo es posible determinar que un elemento fue completamente leído cuando se comenzó a leer otro, por lo que lo mas conveniente es retornar la última lectura al flujo para luego volver a iniciar el reconocimiento del nuevo elemento. Se crea en base a un objeto Reader.
- **InputStreamReader / OutputStreamWriter:** Proveen un puente entre un flujo de bytes y un flujo de caracteres. Los bytes leídos son convertidos a caracteres, los caracteres escritos son convertidos a bytes. Las conversiones se realizan según la codificación por defecto de caracteres de la plataforma actual (llamado también charset). InputStreamReader se crea en base a un objeto InputStream. OutputStreamReader se crea en base a un objeto OutputStream.
- **FileReader / FileWriter:** Permiten la misma funcionalidad que InputStreamReader y OutputStreamWriter, pero el flujo se obtiene desde un archivo. Ambos se crean en base a un archivo.
- **PipedReader / PipedWriter:** Permiten leer / escribir caracteres y arreglos de caracteres a pipes. Un pipe es un objeto de comunicación unidireccional entre hilos. Un objeto PipedReader es conectado a un objeto PipedWriter. El primero es utilizado por un hilo, el cual escribe caracteres que son leídos por otro hilo utilizando el segundo objeto.
- **StringReader / StringWriter:** Permite el manejo de un objeto String como un flujo de caracteres. StringReader se crea en base a un objeto String externo. StringWriter crea su propio objeto String.

- **PrintWriter:** Permite enviar representaciones de texto de datos primitivos y objetos a un flujo de caracteres de salida. Se puede crear en base a un objeto `OutputStream` o a un objeto `Writer`.

Una clase de flujo comúnmente es concatenada con otra de forma que se combine la funcionalidad de ambas. Es importante tener en cuenta que, los flujos de datos son interpretados por las clases asumiendo la forma en que fueron creados. Como ejemplo, si tuviésemos un flujo de bytes donde se escribieron datos primitivos enteros, se leerá basura si se intenta recuperarlo como los caracteres que representen dichos valores, y viceversa, si tuviésemos un flujo de caracteres con números enteros, no se puede recuperar directamente los valores enteros que estos pudiesen representar. Como conclusión, no toda combinación de flujos arrojará resultados correctos. Por lo común, un flujo de escritura es leído mediante su flujo de lectura correspondiente, por ejemplo, un flujo creado con la clase `DataOutputStream` es leído utilizando la clase `DataInputStream`.

En las siguientes secciones examinaremos los dos casos más comunes de manejo de flujos:

- El manejo de flujos desde consola.
- El manejo de flujos desde archivos.

Manejo de Consola

La salida desde consola se suele realizar utilizando directamente el objeto referenciado por el dato miembro estático **out** de la clase `System`. Este objeto es del tipo `PrintWriter`, el cual ofrece la funcionalidad suficiente para escribir datos primitivos y objetos convirtiendo estos a cadenas de texto. Los métodos más utilizados de esta clase son **print** y **println**.

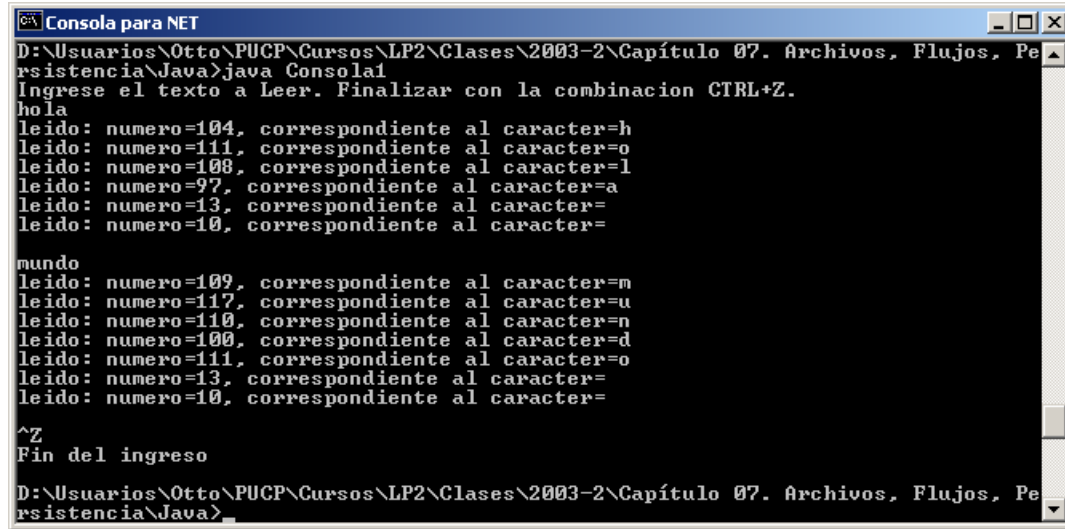
Como contraparte, la clase `System` provee un dato miembro estático **in** del tipo `BufferedReader`. Si se desea trabajar el flujo leyendo byte a byte, este flujo es suficiente. El siguiente ejemplo muestra el uso de esta clase directamente.

```
import java.io.*;

class Consola1 {
    public static void main( String args[] ) throws IOException {
        System.out.print( "Ingrese el texto a Leer." );
        System.out.println( " Finalizar con la combinacion CTRL+Z." );
        while(true) {
            int Entero = System.in.read();
            if( Entero == -1 )
                break;
            char Caracter = (char)Entero;
            System.out.println( "leido: numero=" + Entero +
                               ", correspondiente al caracter=" + Caracter );
        }
        System.out.println( "Fin del ingreso" );
    }
}
```

La Figura 10 - 6 muestra el resultado de una ejecución de ejemplo de este programa.

La sobrecarga del método `read` utilizada devuelve un valor entero entre 0 y 255 cuando la lectura es correcta. Dicho valor puede ser convertido directamente a un `char`. Cuando el método detecta un fin de archivo, que en el caso del ingreso por consola equivale a ingresar `CTRL+Z`, el valor devuelto es `-1`. Nótese que el retorno de carro (con código ASCII 13) y el cambio de línea (con código ASCII 10) también son leídos.



```
Consola para NET
D:\Usuarios\Otto\PUCP\Cursos\LP2\Clases\2003-2\Capítulo 07. Archivos, Flujos, Pe
rsistencia\Java>java Consola1
Ingrese el texto a Leer. Finalizar con la combinacion CTRL+Z.
hola
leido: numero=104, correspondiente al caracter=h
leido: numero=111, correspondiente al caracter=o
leido: numero=108, correspondiente al caracter=l
leido: numero=97, correspondiente al caracter=a
leido: numero=13, correspondiente al caracter=
leido: numero=10, correspondiente al caracter=

mundo
leido: numero=109, correspondiente al caracter=m
leido: numero=117, correspondiente al caracter=u
leido: numero=110, correspondiente al caracter=n
leido: numero=100, correspondiente al caracter=d
leido: numero=111, correspondiente al caracter=o
leido: numero=13, correspondiente al caracter=
leido: numero=10, correspondiente al caracter=

^Z
Fin del ingreso
D:\Usuarios\Otto\PUCP\Cursos\LP2\Clases\2003-2\Capítulo 07. Archivos, Flujos, Pe
rsistencia\Java>
```

Figura 10 - 6 Manejo de consola en Java: Ejecución del programa.

Sin embargo, en la mayoría de los casos, lo que se desea leer son líneas de texto que luego serán procesadas. Lo más común en este caso es combinar `System.in` con las clases `InputStreamReader` y `BufferedReader`. El siguiente ejemplo muestra el uso de esta combinación.

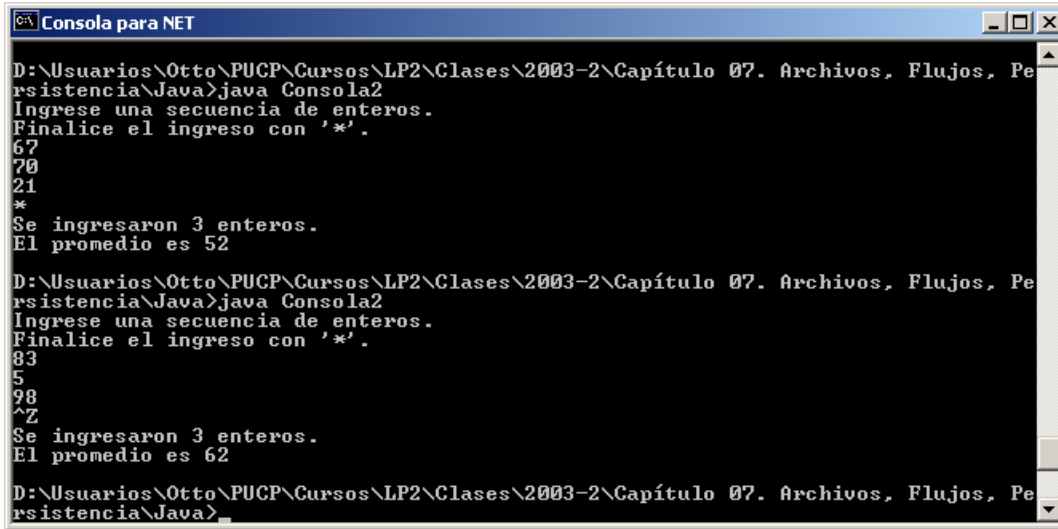
```
import java.io.*;

class Consola2 {
    public static void main(String args[]) throws IOException {
        System.out.println( "Ingrese una secuencia de enteros." );
        System.out.println( "Finalice el ingreso con '*'." );

        BufferedReader Entrada =new BufferedReader(new InputStreamReader(System.in));
        int Contador = 0;
        int Suma = 0;
        while(true) {
            String Linea = Entrada.readLine();
            if( Linea == null )
                break;
            if( Linea.charAt(0) == '*' )
                break;
            Suma += Integer.parseInt( Linea );
            Contador++;
        }
        System.out.println( "Se ingresaron " + Contador + " Enteros." );
        System.out.println( "El promedio es " + Suma / Contador );
    }
}
```

La Figura 10 - 7 muestra el resultado de una ejecución de ejemplo de este programa.

Es importante notar que el método `Integer.parseInt` puede arrojar una excepción si la conversión de la cadena pasada como parámetro no representa un número entero válido. Dado que no nos interesa, para este ejemplo, manejar nosotros mismo ese tipo de excepción, lo manifestamos mediante la palabra `throws` al final del encabezado del método `main`. En este caso no podemos dejar la excepción sin declararla, dado que `IOException` no es una excepción runtime, es decir, no deriva de la clase `RuntimeException`.



```
D:\Usuarios\Otto\PUCP\Cursos\LP2\Clases\2003-2\Capítulo 07. Archivos, Flujos, Pe
rsistencia\Java>java Consola2
Ingrese una secuencia de enteros.
Finalice el ingreso con '*'.
67
70
21
*
Se ingresaron 3 enteros.
El promedio es 52

D:\Usuarios\Otto\PUCP\Cursos\LP2\Clases\2003-2\Capítulo 07. Archivos, Flujos, Pe
rsistencia\Java>java Consola2
Ingrese una secuencia de enteros.
Finalice el ingreso con '*'.
83
5
98
^Z
Se ingresaron 3 enteros.
El promedio es 62

D:\Usuarios\Otto\PUCP\Cursos\LP2\Clases\2003-2\Capítulo 07. Archivos, Flujos, Pe
rsistencia\Java>
```

Figura 10 - 7 Manejo de consola en Java: Ejecución del programa.

En otros casos es necesario procesar una línea de texto de forma que se particione ésta en elementos, cada elemento puede ser una palabra, un número, etc. Los elementos son reconocidos dado que comúnmente los separamos unos de otros mediante delimitadores, como espacios en blanco o comas. A este tipo de trabajo se llama **tokenización**. Un token es una subcadena de caracteres dentro de una cadena, delimitados por caracteres especiales denominados “delimitadores”. Para realizar este trabajo se puede utilizar la clase `StringTokenizer` del paquete `java.util`. El siguiente ejemplo muestra el uso de esta clase.

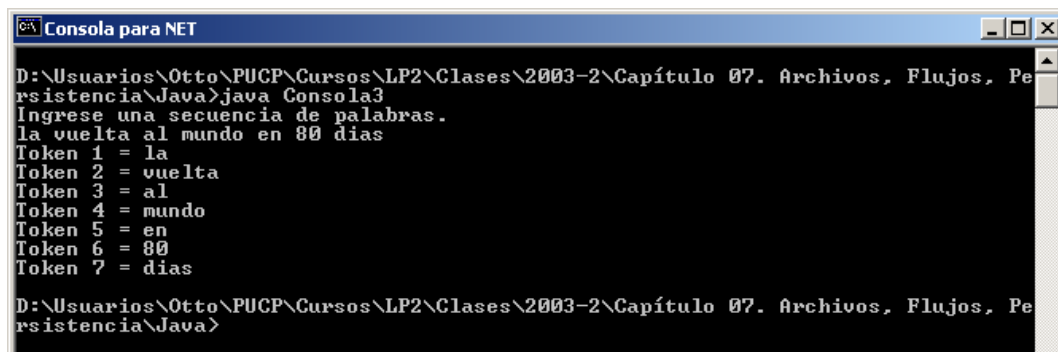
```
import java.io.*;
import java.util.StringTokenizer;

class Consola3 {
    public static void main(String args[]) throws IOException {
        System.out.println( "Ingrese una secuencia de palabras." );

        BufferedReader Entrada =new BufferedReader(new InputStreamReader(System.in));
        String Linea = Entrada.readLine();

        StringTokenizer Tokens = new StringTokenizer( Linea );
        int Contador = 0;
        while ( Tokens.hasMoreTokens() )
            System.out.println( "Token " + (++Contador) + " = " + Tokens.nextToken() );
    }
}
```

La Figura 10 - 8 muestra el resultado de una ejecución de ejemplo de este programa.



```
D:\Usuarios\Otto\PUCP\Cursos\LP2\Clases\2003-2\Capítulo 07. Archivos, Flujos, Pe
rsistencia\Java>java Consola3
Ingrese una secuencia de palabras.
la vuelta al mundo en 80 dias
Token 1 = la
Token 2 = vuelta
Token 3 = al
Token 4 = mundo
Token 5 = en
Token 6 = 80
Token 7 = dias

D:\Usuarios\Otto\PUCP\Cursos\LP2\Clases\2003-2\Capítulo 07. Archivos, Flujos, Pe
rsistencia\Java>
```

Figura 10 - 8 Manejo de consola en Java: Ejecución del programa.

El constructor utilizado para crear el objeto `StringTokenizer` recibe como parámetro la cadena a tokenizar. Este constructor utiliza como delimitadores por defecto la siguiente cadena: “`\t\n\r\p?`”. Es posible utilizar un segundo constructor que permite pasar como segundo parámetro una cadena con los caracteres que deseamos funcionen como delimitadores.

Si se desea un manejo más complejo de un flujo de texto, reconociéndose cadenas de texto, números, comentarios, etc. (a la manera como un compilador procesa un archivo fuente), se puede utilizar la clase `StreamTokenizer` del paquete `java.io`. El uso de esta clase escapa de los alcances del curso.

Por último, al igual que en la programación en C y C++, son 3 los flujos estándar que ofrece Java mediante datos miembro (referencias a objetos de flujo) de la clase `System`:

- `System.in` Entrada estándar, por defecto direccionado al teclado desde la consola.
- `System.out` Salida estándar, por defecto direccionado a la pantalla de la consola.
- `System.err` Salida de error estándar, por defecto direccionado a la pantalla de la consola.

Cualquiera de éstos puede ser redireccionado por el programa hacia, por ejemplo, un archivo. El proceso de redireccionamiento consiste simplemente es asignar un nuevo objeto a estas referencias.

Es importante recordar que todo programa en Java, sea de consola o gráfico, crea automáticamente una consola, por lo que a ésta siempre es posible utilizarla.

Manejo de Archivos

Cuando un programa accede a un archivo podría, dependiendo del modo de acceso solicitado, leer cualquier parte de el directamente, sin necesidad de seguir un orden, cosa muy diferente al concepto de un flujo el cual funciona como una cola o FIFO (First In First Out). Si bien los archivos pueden trabajarse lógicamente como un flujo, lo que equivale al conocido modo de acceso secuencial, también es posible, dado las características propias de éste, trabajarlo accediendo aleatoriamente.

Se examinarán los siguientes casos de manejo de archivos:

- Manejo de archivos de texto.
- Manejo de archivos binarios secuencialmente.
- Manejo de archivos binarios aleatoriamente.

Archivos de Texto

Los archivos de texto se manejan comúnmente utilizando las clases `FileWriter` y `FileReader`. El siguiente programa muestra en uso de estas clases.

```
import java.io.*;

class ArchivoTexto {
    public static void main(String args[]) throws IOException {
        FileWriter Escritor = new FileWriter( args[0] );
        System.out.println( "Ingrese el texto a almacenar: " );

        BufferedReader Entrada =new BufferedReader(new InputStreamReader(System.in));
        while(true) {
            String Linea = Entrada.readLine();
            if( Linea == null )
                break;
            Escritor.write( Linea + "\n" );
        }
    }
}
```



```
        Escritor.close();
        Entrada.close();

        System.out.println( "Leyendo el archivo creado." );
        FileReader Lector = new FileReader( args[0] );
        while( Lector.ready() )
            System.out.print( (char)Lector.read() );
        Lector.close();
    }
}
```

La Figura 10 - 9 muestra el resultado de una ejecución de ejemplo de este programa.

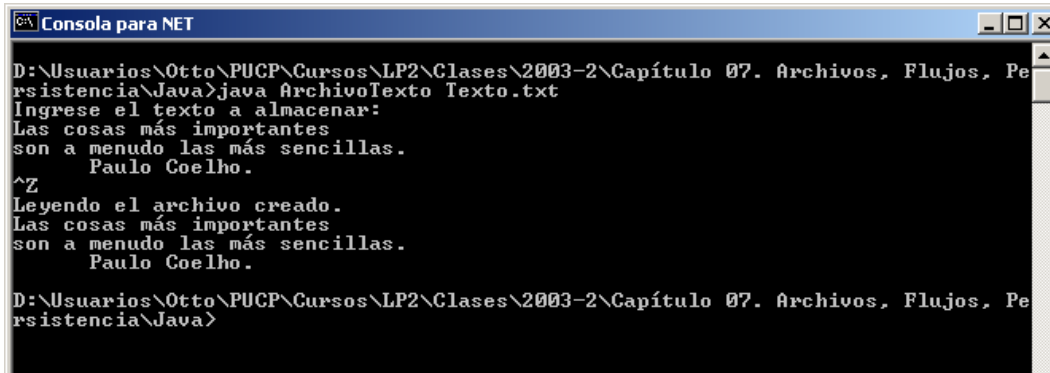


Figura 10 - 9 Manejo de consola en Java: Ejecución del programa.

Note que la funcionalidad ofrecida por `FileWriter` es diferente de la de `FileReader`. `FileWriter` extiende `Writer`, el cual permite escribir objetos de tipo cadena de texto (`String`), lo cual facilita mucho la escritura. `FileReader` extiende `Reader`, el cual sólo permite leer caracteres y arreglos de caracteres. Lo común es leer línea por línea un texto. Para realizar esto podemos combinar la clase `FileReader` con la clase `BufferedReader`, dado que la última provee lectura de líneas completas. El código de lectura del programa pudo reemplazarse de la siguiente forma:

```
BufferedReader Lector = new BufferedReader( new FileReader( args[0] ) );
while( Lector.ready() )
    System.out.println( Lector.readLine() );
Lector.close();
```

De igual forma, si deseamos utilizar las capacidades de la clase `PrintWriter`, como con el objeto referenciado por `System.out`, podríamos utilizar un código como el siguiente:

```
PrintWriter Escritor = new PrintWriter( new FileWriter( args[0] ) );
BufferedReader Entrada = new BufferedReader( new InputStreamReader( System.in ) );
while(true) {
    String Linea = Entrada.readLine();
    if( Linea == null )
        break;
    Escritor.println( Linea );
}
Escritor.close();
```

Archivos Binarios Secuenciales

Para crear un stream binario desde un archivo se utilizan las clases `FileInputStream` y `FileOutputStream` para lectura y escritura respectivamente. Una vez creado los flujos relacionados a los archivos, es común utilizar las clases `BufferedInputStream` y `BufferedOutputStream` respectivamente si se desea mejorar la performance cuando se realizan muchas lecturas o escrituras desde y hacia disco.

Para leer y escribir datos de tipo primitivo se suele utilizar las clases `DataInputStream` y `DataOutputStream`. Luego, la creación de los streams de lectura y escritura para estos casos sería:

```
DataInputStream input = new DataInputStream(  
    new BufferedInputStream( new FileInputStream( "MiArchivo.dat" ) ) );  
  
DataOutputStream output = new DataOutputStream(  
    new BufferedOutputStream( new FileOutputStream( "MiArchivo.dat" ) ) );
```

Para los programas que sólo requieren almacenar y recuperar algunos datos simples el uso de estas clases puede ser suficiente, para programas complejos la información que se maneja provienen de objetos con datos de configuración o bien arreglos de objetos, es decir, más que datos primitivos lo que se desea es almacenar y recuperar objetos completos. Aquí entran a tallar dos conceptos importantes: **Persistencia** y **serialización**.

Se dice que un objeto es persistente si su existencia trasciende el de su creador y sus usuarios. Para nuestro punto de vista como programadores, quien crea y usa un objeto es un programa.

La serialización es una forma de implementación del concepto de persistencia, utilizando para esto un medio de almacenamiento también persistente, como el disco duro (su información se conserva mas allá de que cada instancia de ejecución de los programas que crean o utilizan la información, inclusive del sistema operativo mismo y de que la maquina se apague o prenda). La serialización consiste en la lectura y escritura de objetos a y desde un flujo amarrado a dicho medio persistente.

Para serializar un objeto es necesario marcar su clase como serializable. Esto consiste en hacer que la clase a la que pertenece el objeto que deseamos serializar implemente la interfaz `Serializable`. Dicha interfaz no define ningún método ni constante, sólo sirve para marcar la clase. Si la clase contiene datos miembro que sean referencias, los tipos de dichas referencias también deben marcarse como serializables.

Las clases que permiten la serialización de objetos son `ObjectInputStream` y `ObjectOutputStream`.

`ObjectOutputStream` define un método `writeObject` el cual recibe como parámetro un objeto mediante una referencia de tipo `Object`. Este método se encarga de averiguar si la clase fue marcada como `Serializable`. Si no fue marcada se produce una excepción en tiempo de ejecución. Si fue marcada, se serializa tanto la información concerniente al tipo de objeto marcado (de forma que después pueda recrearse el mismo tipo de objeto), como los datos del objeto.

`ObjectInputStream` define un método `readObject`, el cual no recibe parámetros y retorna una referencia de tipo `Object` del objeto leído y recreado. El método utiliza la información almacenada sobre el tipo del objeto original para recrear uno del mismo tipo, y la información sobre sus datos miembro para inicializar los del objeto.

El siguiente programa muestra el uso de estas clases.

```
import java.io.*;  
  
class Direccion implements Serializable {  
    int numero;  
    String calle;  
    String distrito;  
    public Direccion(int numero, String calle, String distrito)    {  
        this.numero = numero;  
        this.calle = calle;  
        this.distrito = distrito;  
    }  
    public String toString() {  
        return calle + " " + numero + ", " + distrito;  
    }  
}
```

```
}

class Persona implements Serializable {
    private String nombre;
    private int edad;
    private Direccion dir;
    public Persona(String nombre, int edad, Direccion dir) {
        this.nombre = nombre;
        this.edad = edad;
        this.dir = dir;
    }
    public String toString() {
        return "Sr(a). " + nombre + ", " + edad + " años, direccion = " + dir;
    }
}

class ArchivoBinarioSecuencial {
    public static void main(String args[]) throws IOException, ClassNotFoundException {
        if(args.length < 2) {
            System.out.println("Error en argumentos.");
            return;
        }
        if(args[0].equals("/e")) {
            ObjectOutputStream Escritor = new ObjectOutputStream(
                new BufferedOutputStream(new FileOutputStream(args[1])));
            Persona[] ListaPersonas = {
                new Persona("Jose", 25, new Direccion(123, "Jose Leal", "San
Juan")),
                new Persona("Mara", 26, new Direccion(456, "Carrión", "Barranco")),
                new Persona("Ana", 35, new Direccion(789, "Fresnos", "Lince"))
            };
            for(int i = 0; i < ListaPersonas.length; i++)
                Escritor.writeObject( ListaPersonas[i] );
            Escritor.close();
        } else if(args[0].equals("/l")) {
            ObjectInputStream Lector = new ObjectInputStream(
                new BufferedInputStream(new FileInputStream(args[1])));
            while(true) {
                Persona p;
                try { p = (Persona)Lector.readObject(); }
                catch EOFException ex { break; }
                System.out.println("Persona : " + p);
            }
            Lector.close();
        }
    }
}
```

La Figura 10 - 10 muestra el resultado de una ejecución de ejemplo de este programa.

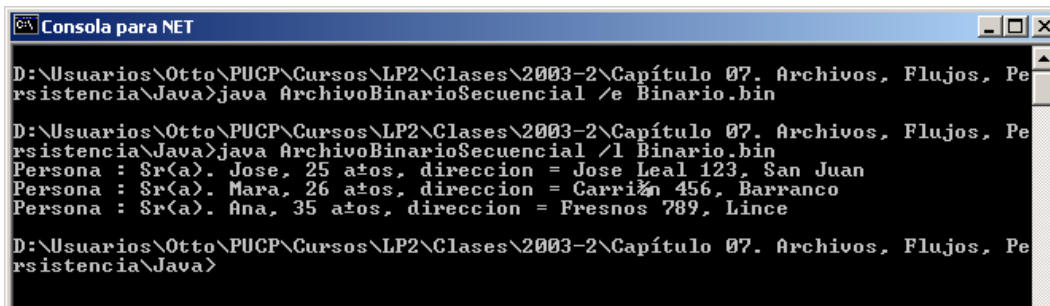


Figura 10 - 10 Archivos binarios secuenciales en Java: Ejecución de un ejemplo.

Archivos Binarios Aleatorios

Para manejar aleatoriamente un archivo se utiliza la clase `RandomAccessFile`. Esta clase provee la misma funcionalidad de las clases `DataInputStream` y `DataOutputStream`. Ésta permite

manejar un archivo como un arreglo de bytes, pudiendo posicionarnos en cualquier lugar del arreglo para realizar una lectura o escritura. El siguiente programa muestra el uso de esta clase:

```
import java.io.*;

class ArchivoBinarioAleatorio {
    public static void main(String args[]) throws IOException {
        RandomAccessFile Lector = new RandomAccessFile( "datos.dat", "rw" );
        Lector.writeInt( 100 );
        Lector.writeBoolean( false );
        Lector.writeDouble( 6.54 );
        Lector.writeUTF( "Hola mundo" );
        System.out.println( "Archivo llenado" );
        System.out.println( "Tamano del archivo = " + Lector.length() );

        System.out.println( "\nLeyendo el archivo" );
        Lector.seek( 0 );
        System.out.println( "Entero = " + Lector.readInt() );
        System.out.println( "Booleano = " + Lector.readBoolean() );
        System.out.println( "Punto Flotante = " + Lector.readDouble() );
        System.out.println( "Cadena = " + Lector.readUTF() );

        Lector.close();
    }
}
```

La Figura 10 - 11 muestra el resultado de una ejecución de ejemplo de este programa.

Para cada tipo de dato primitivo se cuenta con un método read y un método write adecuado. Cada vez que realizamos una lectura o una escritura, el apuntador del archivo se desplaza el número de bytes acorde con el tamaño del dato leído. En cualquier momento podemos recolocar el apuntador mediante el método seek, el cual recibe como parámetro un entero que expresa el número de bytes a desplazarse con respecto al inicio del archivo.

Note que sólo tenemos la capacidad de manejar datos primitivos y cadenas de texto en formato UTF y si bien es posible que nos desplazemos a cualquier posición del archivo para realizar una lectura a menos que conozcamos qué datos están guardados y en qué orden, o bien todos los datos guardados sean del mismo tipo y tamaño, resultará imposible leer cualquier dato aleatoriamente. Adicionalmente, dado que no es posible escribir y leer objetos, tampoco es posible manejar el archivo con una estructura de registros, como es la costumbre para archivos binarios en C y C++. Tampoco es posible combinar la clase RandomAccessFile con otra clase de flujo, dado que no cuenta con constructores que acepten estos. Tampoco hereda de otra clase de flujo, por lo que no es posible pasar una referencia del tipo RandomAccessFile a otro flujo. Esta clase está pensada para ser utilizada sola.

Figura 10 - 11 Archivos binarios aleatorios en Java: Ejecución del programa.

Debido a que la extensa librería de clases de Java está pensada para solucionar los problemas más comunes de programación, es de esperar que exista alguna clase o paquete que permita

poder manejar un juego de registros en archivos y acceder a estos registros directamente, esto es, de manera aleatoria, sin necesidad de tener que recorrer toda la información desde el inicio cada vez, hasta llegar a dicha información. Este tipo de trabajo es en mucho lo que permite un sistema de manejo de base de datos. Luego, la solución común cuando se desea realizar este tipo de trabajo es utilizar los paquetes de la librería JDBC para crear y administrar base de datos. Otra solución posible es extender la clase `RandomAccessFile` para permitir escribir y leer objetos y proveer alguna funcionalidad adicional para poder utilizar llaves únicas a cada objeto/registro guardado, de forma que se pueda acceder directamente a dicha información. Estos aspectos van más allá del alcance del curso.

Programación con GUI

El presente capítulo se centra en el trabajo con Interfaces Gráficas de Usuario (IGU por sus siglas en español, GUI por sus siglas en inglés) utilizando ventanas y otros elementos para el diseño de las mismas, dentro del sistema operativo Windows. Para el caso particular de Java, dada su característica multiplataforma, los conceptos dados aquí son aplicables a cualquier plataforma que soporte Java.

Interfaces GUI con Ventanas

Una interfaz gráfica de usuario (GUI por sus siglas en inglés) es el conjunto de elementos gráficos que un programa, ejecutándose en un computador, utiliza para permitirle al usuario interactuar “visualmente” con él. Un programa GUI utiliza hardware que lo asiste (monitor y tarjeta de video) trabajando en “modo gráfico”. Si bien el texto es, en esencia, un gráfico, las interfaces basadas exclusivamente en texto (TUI por sus siglas en inglés), con monitores trabajando en “modo texto”, son diferenciadas de las anteriores.

Un programa puede trabajar con una TUI, con una GUI o con ambas, aunque en éste último caso lo que se tiene es realmente un programa trabajando en “modo gráfico” emulando el comportamiento de un “modo texto”.

Casi desde sus inicios (que se remontan, por lo menos, a 1973, con la primera computadora Alto de la empresa Xerox PARC puesta en funcionamiento, o más conocida, la Star de Xerox en 1981), el concepto de ventanas formó parte del concepto de GUI, como una estrategia de organización del área gráfica.

Tipo de Interfaces GUI con Ventanas

En los programas GUI con ventanas, el usuario utiliza dispositivos como el teclado y el ratón, cuyo uso envía mensajes al computador y éstos son capturados por el sistema operativo, el cual decide a qué ventanas pertenecen dichos mensajes y los “envía” a las aplicaciones correspondientes para que éstas realicen alguna acción. Es importante tener en cuenta que este mecanismo de mensajes es utilizado por otros programas, incluyendo el propio sistema operativo, para “generar” nuevos mensajes, no relacionados con la interacción gráfica mediante las ventanas, y “enviarlos” a los programas en ejecución. Ejemplos de éstos son los mensajes de comunicación entre programas (corriendo en la misma computadora o en computadoras diferentes conectadas a una red), los mensajes de aviso de los cambios en la configuración del

sistema operativo (resolución de la pantalla, idioma del teclado, fecha, cambio de usuario, etc.), los relacionados a la escasez de recursos, etc.

Este sistema de mensajería, con distintas variantes, es utilizado por los sistemas operativos con soporte GUI. En particular veremos el caso del sistema operativo Windows y cómo su sistema de mensajería es manejado desde diferentes lenguajes de programación.

En la actualidad, existen dos tipos de programas GUI con ventanas comunmente utilizados: Los programas Stand-Alone y los programas dentro de navegadores de Internet (browsers).

Programas Stand-Alone

Los programas Stand-Alone crean una o más ventanas con las que el usuario interactúa. Comunmente estos programas poseen una ventana principal y una o más ventanas especializadas en alguna labor específica. Estos programas pueden ejecutar directamente (en caso de ser programas ejecutables) o mediante un programa intérprete.

Ejemplos de estos programas son los creados con C o C++ utilizando directamente el API de Windows, así como las Aplicaciones de Java y las Aplicaciones de Formularios de Ventanas de .NET.

Programas Basados en Web

Los programas basados en Web crean contenido para los navegadores Web clientes, utilizados por los usuarios de la Web, como por ejemplo el navegador Web “Internet Explorer” y el “Netscape”. Este contenido Web puede incluir código HTML, scripts ejecutados del lado del cliente, imágenes y datos binarios. Estos programas requieren, para su ejecución, de un navegador Web que los soporte. Si bien, de primera instancia, estos programas utilizan el área de dibujo de la ventana del programa navegador, pueden crear otras ventanas.

Ejemplos de estos programas son los Applets de Java y los Web Forms de .NET.

Creación y Manejo de GUI con Ventanas

Si bien un programa basado en Web puede crear ventanas, adicionalmente a la ventana del programa navegador que lo ejecuta, y muchos de los conceptos de creación y manipulación de los elementos de una ventana se aplican casi idénticamente que en los programas basados en Web, los programas Stand-Alone con ventanas son más simples y sólo dependen de las ventanas que ellos mismos crean. Debido a ésto, toda la explicación respecto a la creación y manejo de GUIs con ventanas se realizará para programas Stand-Alone.

Creación de una Ventana

Veremos cómo se crea un programa mínimo, con una única ventana, en Java, C# y C/C++ con API de Windows.

El usar directamente el API de Windows nos ayudará a entender cómo funciona el sistema de mensajería de Windows. El siguiente código corresponde a un programa en C o C++ que crea una ventana completamente funcional.

```
#include <windows.h>
#include <stdio.h>

LRESULT CALLBACK FuncionVentana(
    HWND hWnd,          // Manejador (handle) de la ventana a la que corresponde el mensaje.
    UINT uMsg,          // Identificador del mensaje.
```

```
WPARAM wParam, // Primer parámetro del mensaje.
LPARAM lParam ) // Segundo parámetro del mensaje.
{
    switch( uMsg )
    {
        case WM_DESTROY:
            // La función 'PostQuitMessage' crea un mensaje WM_QUIT y lo introduce
            // en la cola de mensajes. El parámetro que se le pasa, establece el
            // valor de 'wParam' del mensaje WM_QUIT generado.
            PostQuitMessage( 0 );
            break;
        default:
            // Todos los mensajes para los que no deseo realizar ninguna
            // acción, los paso a la función 'DefWindowProc', la que realiza
            // las acciones por defecto correspondientes a cada mensaje.
            return DefWindowProc( hWnd, uMsg, wParam, lParam );
    }
    return 0;
}

BOOL RegistrarClaseVentana( HINSTANCE hIns )
{
    WNDCLASS wc;
    wc.style          = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc    = FuncionVentana;
    wc.cbClsExtra     = 0;
    wc.cbWndExtra     = 0;
    wc.hInstance      = hIns;
    wc.hIcon          = LoadIcon( NULL, IDI_APPLICATION );
    wc.hCursor        = LoadCursor( NULL, IDC_ARROW );
    wc.hbrBackground  = ( HBRUSH )GetStockObject( WHITE_BRUSH );
    wc.lpszMenuName    = NULL;
    wc.lpszClassName  = "Nombre_Clase_Ventana";
    return ( RegisterClass( &wc ) != 0 );
}

HWND CrearInstanciaVentana( HINSTANCE hIns )
{
    HWND hWnd;
    hWnd = CreateWindow(
        "Nombre_Clase_Ventana",
        "Titulo de la Ventana",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        NULL,
        NULL,
        hIns,
        NULL
    );
    return hWnd;
}

int WINAPI WinMain(
    HINSTANCE hIns,          // Manejador (handle) de la instancia del programa.
    HINSTANCE hInsPrev,     // Manejador (handle) de la instancia de un programa,
                           // del mismo tipo, puesto en ejecución previamente.
    LPSTR lpCmdLine,        // Puntero a una cadena (char*) conteniendo la línea de
                           // comando utilizada al correr el programa.
    int iShowCmd )          // Un entero cuyo valor indica cómo debería mostrarse
                           // inicialmente la ventana principal del programa.
{
    // Registro una clase de ventana, en base a la cual
    // se creará una ventana.
    if( ! RegistrarClaseVentana( hIns ) )
        return 0;

    // Creo una ventana.
    HWND hWnd = CrearInstanciaVentana( hIns );
    if( hWnd == NULL )
```

```
        return 0;

    // Muestro la ventana.
    ShowWindow( hWnd, iShowCmd ); // Establece el estado de 'mostrado' de la ventana.
    UpdateWindow( hWnd );         // Genera un mensaje de pintado, el que es
                                // ejecutado por la función de ventana respectiva.

    // Se realiza un bucle donde se procesen los mensajes de la cola de mensajes.
    MSG Mensaje;
    while( GetMessage( &Mensaje, NULL, 0, 0 ) > 0 )
        DispatchMessage( &Mensaje );

    // GetMessage devuelve 0 cuando el mensaje extraído es WM_QUIT y
    // -1 cuando ha ocurrido un error. Note que esto produce que el mensaje
    // WM_QUIT nunca sea procesado por la función de ventana.

    // El parámetro 'wParam' del mensaje 'WM_QUIT' corresponde al parámetro
    // pasado a la función 'PostQuitMessage'. Este valor tiene la misma utilidad
    // que el entero retornado por la función 'main' en programas en C y C++ para
    // consola.
    return Mensaje.wParam;
}
```

En este programa existen dos funciones importantes: El punto de entrada del programa, la función “WinMain”, y la función de ventana “FuncionVentana”.

La función WinMain es el equivalente, para un programa en Windows, a la función “main” en programas en C y C++ en modo consola. El principal parámetro de esta función es el primero, el manejador de la instancia del programa. Dicho manejador es, en esencia, un número entero utilizado por Windows para ubicar los recursos relacionados al programa. Algunos de los recursos que un programa en Windows puede tener son: Registros de ventanas, textos, imágenes, audio, video, meta-archivos, otros manejadores (a archivos, puertos, impresoras, etc.), etc. El manejador de la instancia es utilizado como parámetro de las funciones del API de Windows donde se involucren, directa o indirectamente, los recursos de una aplicación. El segundo parámetro no es utilizado en programas de 32-bits (Windows 95 y Windows NT en adelante).

La función de ventana (que puede tener cualquier nombre pero con el mismo formato de declaración que el mostrado) es dónde se realiza toda la lógica relacionada a las acciones que una ventana toma en respuesta a los mensajes recibidos, como por ejemplo, los producidos por la interacción del usuario con la ventana. La dirección de esta función de ventana es el principal dato que forma parte del registro, con la estructura WNDCLASS, de una clase de ventana. Todos los mensajes enviados a ventanas, creados con una misma clase, son procesados por la misma función de ventana, la indicada en la estructura WNDCLASS al registrarse dicha clase de ventana.

Todo programa con ventanas en Windows tiene 3 etapas principales:

10. Registro de las clases de ventana que se utilizarán en el programa para crear ventanas.
11. Creación de la ventana principal del programa.
12. Ejecución del bucle de mensajes del programa.

Para crear una ventana se utiliza la función CreateWindow. Los parámetros de esta función indican, en este orden:

9. El nombre de la clase de ventana en base a la que se crea la nueva ventana.
10. El título a mostrarse en la barra superior de la ventana.

11. El tipo y atributos (incluyendo estilo) de la ventana.
12. La coordenada X, en píxeles, de la esquina superior izquierda en la que aparecerá la ventana dentro del área de la pantalla. La constante CW_USEDEFAULT indica que se utilice un valor por defecto.
13. La coordenada Y, en píxeles, de la esquina superior izquierda en la que aparecerá la ventana dentro del área de la pantalla. La constante CW_USEDEFAULT indica que se utilice un valor por defecto.
14. El ancho de la ventana, en píxeles. La constante CW_USEDEFAULT indica que se utilice un valor por defecto.
15. El alto de la ventana, en píxeles. La constante CW_USEDEFAULT indica que se utilice un valor por defecto.
16. El manejador de una ventana padre. Si se pasa NULL (una constante igual a cero) se indica que la ventana no tiene dueño.
17. Un manejador de menú o un identificador de ventana (dependiendo de los valores pasados en el tercer parámetro). Si se pasa NULL se indica que no se utilizará esta característica.
18. El manejador de la instancia del programa.
19. Un puntero genérico (void*) a cualquier información extra que el usuario quiera utilizar. Si se pasa NULL, no se utiliza esta característica.

Creada la ventana principal del programa, se ingresa al bucle de procesamiento de mensajes. Dicho bucle realiza, repetitivamente, las siguientes acciones:

20. Retirar un mensaje de la cola de mensajes, mediante la función GetMessage.
21. Mandar a llamar a la función de ventana correspondiente, mediante la función DispatchMessage.

La estructura MSG almacena los mismos datos que recibe una función de ventana como parámetros. La función GetMessage llena esta estructura con la información del mensaje retirado de la cola de mensajes del programa. La función DispatchMessage utiliza esta información para averiguar a qué ventana corresponde el mensaje, cuál es la clase de ventana de dicha ventana, cuál es la función de ventana registrada con esa clase y, finalmente, llama a dicha función de ventana pasándole los datos almacenados en la estructura MSG.

Es importante notar el hecho de que, si bien se dice en la literatura que la función de ventana es llamada por Windows, no es estrictamente así. Tanto DispatchMessage como otras funciones del API de Windows son las que realmente se encargan de llamar a las funciones de ventana, y dichas funciones son llamadas explícitamente desde el programa. Como puede deducirse, es debido a que estas funciones forman parte del API de Windows, que se dice que es Windows quién llama a las funciones de ventana. Un programa en ejecución, en un sistema operativo multitarea como Windows, no puede llamar directamente a una función de otro programa en memoria. Este aspecto está íntimamente relacionado al tema de programación concurrente, por lo que no ahondaremos por ahora más en esto.

Otro hecho importante es el que todo programa en Windows, sea hecho en C, en Java, en C# o en cualquier otro lenguaje de programación (salvo alguna excepción) presenta, escondida o no, esta estructura de trabajo. Todo programa en Windows requiere registrar las clases de ventana

que utiliza, crear dichas ventanas y tener un bucle de procesamiento de mensajes. Para ilustrar mejor ésto revisemos ejemplos equivalentes en Java y C#.

El siguiente programa es el equivalente en Java al programa anterior.

```
import javax.swing.*;
import java.awt.event.*;
class MiVentana extends JFrame {
    public MiVentana() {
        setSize(400, 400);
        setTitle("Título de la Ventana");
        setVisible(true);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                dispose();
                System.exit(0);
            }
        });
    }
}
class Aplicacion Minima {
    public static void main(String[] args) {
        MiVentana ventana = new MiVentana();
    }
}
```

En este código, el punto de entrada es el método estático “main”, dentro del cual lo único que se realiza es la creación de un objeto de la clase MiVentana que hereda de la clase JFrame (cuyo nombre completo es javax.swing.JFrame), que es la clase base para la creación de ventanas utilizando el paquete (el término utilizado en Java para referirse a una librería) SWING, cuyas clases son accesibles mediante la sentencia “import javax.swing.*”. Dentro del constructor de la clase “MiVentana” se configuran las características de la ventana y se establece “qué objeto” será al que se le llame su método “windowClosing” cuando el usuario del programa indique que desea cerrar la ventana.

Ahora bien, relacionemos todo esto con lo visto anteriormente en el programa en C o C++ con API de Windows.

Como es de esperarse, la clase JFrame debe crear en su constructor (o en algún constructor de sus clases base) una ventana llamando a “CreateWindow” (u otra función equivalente). Como ya hemos visto, no se debe poder crear una ventana antes de registrar la clase de ventana en base a la que se creará, por lo que es de esperarse (y realmente sucede así) que el programa intérprete de Java realice este registro antes de llamar a nuestro método “main”. Siguiendo el orden de ejecución de los constructores, luego de completada la ejecución del constructor de JFrame se llamará al de nuestra clase “MiVentana”, donde modificamos los valores por defecto con los que se creó inicialmente la ventana (por defecto, JFrame crea una ventana en la coordenada [X,Y] = [0,0], con cero píxeles de ancho y cero de alto y con su atributo de visibilidad puesto en “no-visible”). El constructor termina indicando, mediante el método de JFrame “addWindowListener”, a qué método de qué clase se llamará cuando la función de ventana, de la ventana creada por JFrame, reciba el mensaje WM_CLOSE. El procesamiento por defecto, realizado por “DefWindowProc”, para este mensaje es llamar a la función de API de Windows “DestroyWindow”, que es la que realmente destruye la ventana generando, de paso, el mensaje WM_DESTROY. Como puede deducirse, el método “dispose” de JFrame debería estar llamando a DestroyWindow y el método “System.exit” a PostQuitMessage.

Finalmente queda algo muy importante que no es directamente visible en nuestro código: ¿Dónde se ejecuta el bucle de procesamiento de mensajes? Para explicar ésto, debe aclararse que el programa intérprete de Java realiza algunas acciones luego de llamar a nuestro método “main”, entre ellas está la de verificar si después de ejecutarse este método se creó o no alguna

ventana. Si se creó entonces se entra a un bucle de procesamiento de mensajes del que, como puede esperarse, el programa no sale hasta haberse llamado al método “System.exit”. Si no se creó ninguna ventana, el programa intérprete finaliza. Es por esto que la literatura sobre programación con ventanas en Java indica, sin dar mayor detalle, que si se creara alguna ventana en un programa Java Stand-Alone (a éstos se les llama “Aplicaciones Java”), debe de llamarse en algún momento al método “System.exit”.

Ahora bien, el siguiente programa es el equivalente en C# del programa anterior.

```
using System;
using System.Windows.Forms;

class MiVentana : Form {
    public MiVentana() {
        Size = new System.Drawing.Size(400, 400);
        Text = "Título de la Ventana";
        Visible = true;
    }
}

class Aplicacion_Minima {
    public static void Main(string[] args) {
        MiVentana ventana = new MiVentana();
        Application.Run(ventana);
    }
}
```

En el código, el punto de entrada es el método estático “Main” (a diferencia de Java, C# ofrece varias sobrecargas posibles: Con y sin parámetros, con y sin valor de retorno), dentro del cuál se crea una objeto de la clase MiVentana que hereda de la clase Form (cuyo nombre completo es System.Windows.Forms.Form), que es la clase base para la creación de ventanas utilizando un ensamblaje (el término utilizado en C# para referirse a una librería) System.Windows.Forms, también llamado Windows Forms, cuyas clases son accesibles mediante la sentencia “using System.Windows.Forms”. Dentro del constructor de la clase “MiVentana” se configuran las características de la ventana.

Nuevamente, ¿cómo se relaciona todo ésto con lo visto anteriormente en el programa en C o C++ con API de Windows?

Como es de esperarse, la clase Form actúa en forma muy similar a la clase JFrame de Java, creando una ventana utilizando funciones como CreateWindow, para luego modificar los valores por defecto de creación (a diferencia de Java, dicha ventana es por defecto visible, con una posición [X,Y] y un ancho y alto con valores por defecto) en el constructor de “MiVentana”. A diferencia de Java, Form sí contiene una implementación por defecto cuando se desea cerrar la ventana, por lo que no se requiere escribir algún código al respecto. El porqué Java sí lo requiere y C# no, se debe a que Java no tiene forma de saber cuál es nuestra ventana principal, en caso hayamos creado más de una. Por el contrario, C# requiere que se lo indiquemos al llamar al método estático “Application.Run”. Como puede deducirse, este método realiza el bucle de procesamiento de mensajes.

Como puede verse, no importa el lenguaje de programación utilizado, o qué tanto dicho lenguaje nos oculte la implementación interna, tras capas de abstracción (en forma de clases por ejemplo), siempre debemos tener claro que dicha implementación debe necesariamente contener los elementos mostrados en el programa en C o C++ anterior, y por tanto, debe utilizar las funciones del API de Windows que ese programa utiliza. En algunos casos, dichos lenguajes de programación ofrecen acceso a elementos de bajo nivel del API de Windows, como los manejadores de las ventanas que crean y utilizan (por ejemplo Visual Basic, a manera de propiedades de algunos de sus controles visuales), de manera que se acceda a cierta funcionalidad que sólo provee dicha API.

Elementos de una Ventana

Un conjunto de programas GUI con ventanas, que comparten el mismo conjunto de librerías para la manipulación de éstas (en Windows, estas librerías forman parte del mismo sistema operativo, y se les llama API de Windows), comparten no sólo un aspecto común, sino un conjunto de elementos gráficos con un comportamiento y uso común. A esto se le conoce como el “Look And Feel” de dicho entorno GUI.

El hecho de tener elementos de aspecto y comportamiento común reduce la curva de aprendizaje para que un usuario, que ya aprendió a utilizar uno de estos programas, aprenda a utilizar otro que utilice la misma librería GUI.

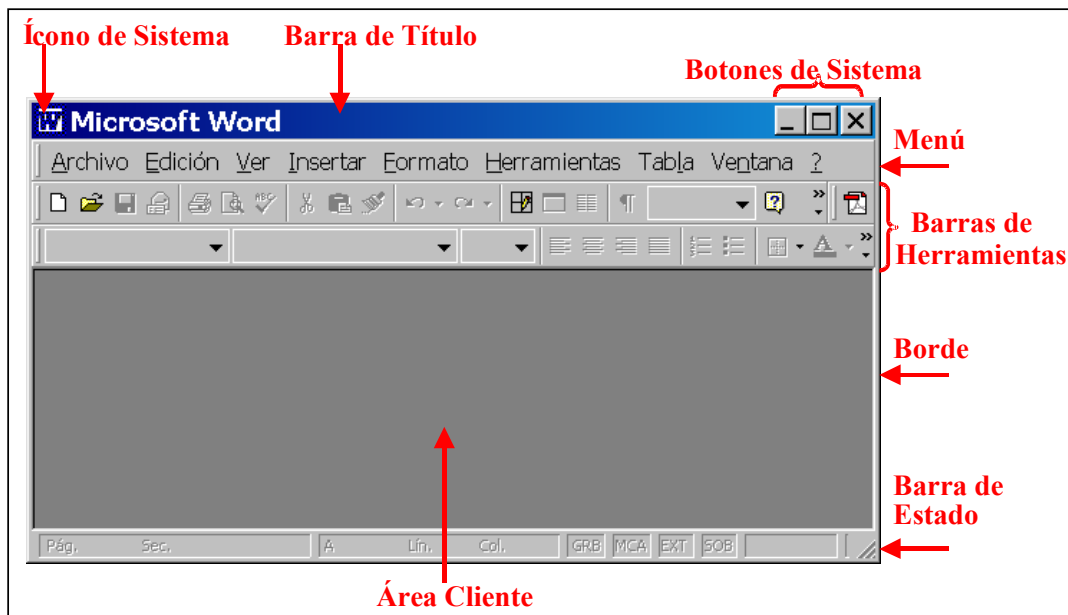


Figura 6 - 1 Elementos de una ventana

En Windows, como se aprecia en la Figura 6 - 1, las ventanas tienen los siguientes elementos:

- Un borde.
- Una barra de título.
- Un ícono de sistema.
- Un conjunto de botones de sistema.
- Un menú.
- Una o más barras de herramientas.
- Una barra de estado.
- Un área de dibujo o “área cliente”.

De estos elementos, sólo el último es obligatorio, y si bien los demás son comunes, algunas ventanas pueden no tenerlos.

Interacción con el Entorno

Cada uno de los elementos de una ventana tiene una forma de interacción con el usuario del programa al que pertenece dicha ventana. Internamente, como ya hemos visto, cada una de estas interacciones (el uso del teclado o el ratón) produce un mensaje, que es capturado por el sistema operativo, es colocado por éste en la cola de mensajes del programa correspondiente y finalmente es retirado de dicha cola por el bucle de procesamiento de mensajes de dicho programa. Ya hemos visto como el mecanismo original de procesamiento de estos mensajes, mediante una función de ventana, puede ser abstraído y ocultado al usuario mediante clases que hagan más sencillo este trabajo, como en el caso de Java y C#. En esta sección veremos en mayor detalle, cómo son estas estrategias de abstracción y qué tipos de mensajes se pueden procesar.

Manejo de Eventos con API de Windows

La estrategia de manejo de eventos del sistema operativo Windows, y por tanto del API de Windows, es mediante un sistema de mensajería, similar al sistema de mensajería de correo físico o electrónico.

Un mensaje es el conjunto de datos que el sistema operativo recolecta para un evento dado. Como ejemplo, para el evento click de un botón del ratón, el sistema operativo crea un mensaje que contiene, entre otras cosas, en qué posición de la pantalla y con cuál botón del ratón se hizo click. Dichos mensajes, como cartas de correo, son colocados en las colas de mensajes de las ventanas a las que les corresponden, como si fueran buzones para dichas cartas. Cuando el sistema operativo le da tiempo de CPU a un programa en ejecución, dicho programa verifica si hay mensajes en su cola de mensajes, como cuando nosotros tenemos un poco de tiempo libre (o cualquier otra excusa para tomarnos un descanso) y vemos nuestro correo electrónico. Si el programa encuentra un mensaje en su cola, lo saca y lo procesa.

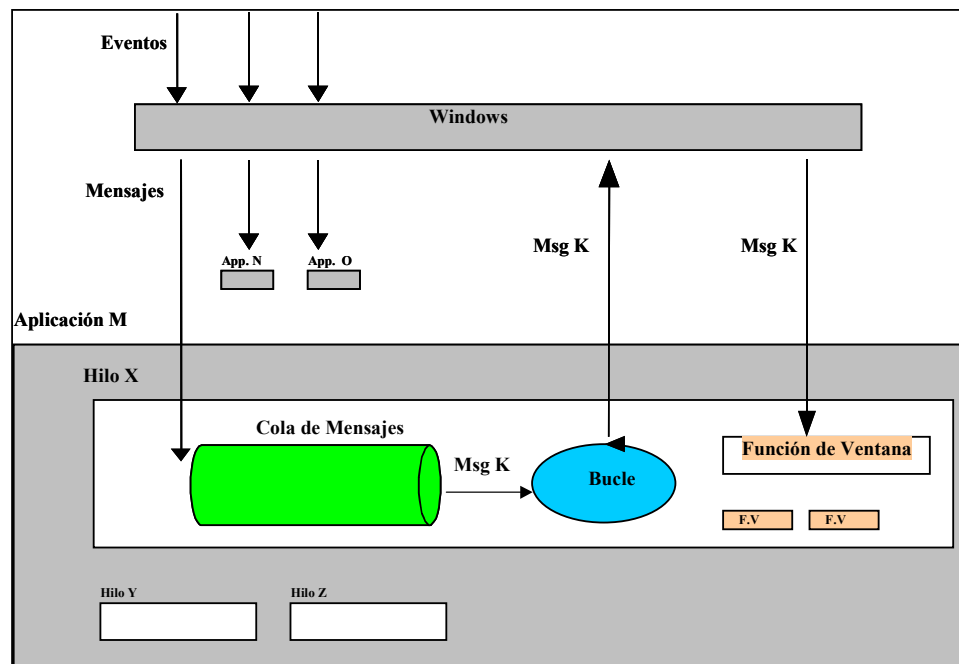


Figura 6 - 2 Procesamiento de un evento

La Figura 6 - 2 muestra el ciclo que siguen los eventos procesados en Windows para las aplicaciones que utilizan ventanas.

La secuencia de pasos seguida es:

- Windows detecta un evento.
- Crea el mensaje respectivo y lo envía a la aplicación involucrada.
- El bucle de procesamiento de mensajes detecta dicho mensaje y solicita a Windows que lo envíe a la ventana adecuada.
- Windows determina la ventana destinataria y averigua a qué clase pertenece.
- En base a la clase determina la función de ventana que le corresponde y envía el mensaje a dicho procedimiento.
- La función de ventana actúa según el mensaje recibido.

Para el procesamiento de los mensajes, toda ventana tiene una “función de procesamiento de mensajes” relacionada, a la que se le llama “función de ventana”. Dicha “relación” entre una ventana y su función de ventana se origina al crearse la ventana utilizando una plantilla de creación llamada “clase de ventana”, la cual debe haberse registrado previamente. Uno de los datos de dicha plantilla es la dirección de la función de ventana que deberá llamarse para procesar los mensajes de toda ventana que se cree utilizando dicha plantilla.

La función de ventana es pues, el área central de trabajo de todo programa desarrollado utilizando el API de Windows. El resto del código suele ser casi siempre el mismo. La función de ventana es la que determina cómo se comportará nuestro programa, nuestra ventana para el usuario, ante cada evento.

Existe un conjunto de mensajes estándar reconocidos por el sistema operativo y que nuestras funciones de ventana pueden manejar. Para cada uno de estos mensajes existe una constante relacionada. En el programa básico mostrado en la sección 3.1, se utilizó una de estas constantes: WM_DESTROY. Al igual que esta constante, definida dentro del archivo de cabecera Windows.h, existe una constante WM_XXX para cada mensaje reconocido por el sistema operativo. Es posible que un programador defina sus propios mensajes simplemente escogiendo un valor fuera del rango que Windows reserva para los desarrolladores de su sistema operativo. El manejo de mensajes definidos por el usuario cae fuera del alcance de esta introducción.

Cuando la función de ventana es llamada para procesar un mensaje, recibe los siguientes datos:

- El handle de la ventana a la que corresponde el mensaje, dado que, como hemos visto, una función de ventana puede utilizarse para procesar los mensajes de más de una ventana, cuando todas éstas fueron creadas utilizando la misma clase de ventana.
- La constante que identifica al mensaje.
- Dos parámetros que contienen información sobre dicho mensaje, o bien contienen direcciones de estructuras reservadas en memoria con dicha información.

El hacer que una ventana reconozca y reaccione a un nuevo mensaje suele consistir en agregar el “case” (al “switch” principal de la función de ventana) para la constante de dicho mensaje con el código que realice el comportamiento deseado. El siguiente código de una función de ventana muestra el manejo de un mensaje correspondiente al ratón y al teclado. El resto del programa no varía respecto al ejemplo anterior.

```
LRESULT CALLBACK FuncionVentana(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    int PosX, PosY;
    char Mensaje[100];
    int CtrlPres, ShiftPres;
    intCodigoTecla, EsTeclaExtendida;

    switch( uMsg )
    {
        case WM_LBUTTONDOWN:
            PosX = LOWORD(lParam); // el word menos significativo
            PosY = HIWORD(lParam); // el word más significativo
            CtrlPres = wParam & MK_CONTROL;
            ShiftPres = wParam & MK_SHIFT;
            sprintf(Mensaje, "X=%d, Y=%d, CtrlPres=%d, ShiftPres=%d",
                PosX, PosY, CtrlPres, ShiftPres);
            MessageBox(hWnd, Mensaje, "Posición del mouse", MB_OK);
            break;
        case WM_KEYUP:
            CodigoTecla = (int)wParam;
            EsTeclaExtendida = ( lParam & ( 1 << 24 ) ) != 0;
            if( EsTeclaExtendida == 1 )
                sprintf(Mensaje, "CodigoTecla=%d (extendida)", CodigoTecla);
            else
            {
                if( ( '0' <= CodigoTecla && CodigoTecla <= '9' ) ||
                    ( 'A' <= CodigoTecla && CodigoTecla <= 'Z' ) )
                    sprintf(Mensaje, "CodigoTecla=%d (%c)", CodigoTecla,
                        (char)CodigoTecla);
                else
                    sprintf(Mensaje, "CodigoTecla=%d", CodigoTecla);
            }
            MessageBox(hWnd, Mensaje, "Tecla presionada", MB_OK);
            break;
        case WM_DESTROY:
            PostQuitMessage( 0 );
            break;
        default:
            return DefWindowProc( hWnd, uMsg, wParam, lParam );
    }
    return 0;
}
```

La constante WM_LBUTTONDOWN corresponde al mensaje producido por el evento de soltar (UP) el botón izquierdo (LBUTTON) del ratón. Para este mensaje, el parámetro wParam se comporta como un bit-flag con información como “estaba la tecla CTRL presionada cuando se produjo el evento”. La información de wParam se extrae utilizando constantes definidas en Windows.h, utilizando operaciones booleanas a nivel de bits. El parámetro lParam contiene la posición del ratón al ocurrir el evento. Dicha posición es relativa a la esquina superior izquierda del área cliente de ventana. Los dos bytes menos significativos de lParam contienen la posición X del ratón; los dos bytes más significativos la posición Y, ambas coordenadas medidas en píxeles. Hay 35 mensajes relacionados con el ratón, de los cuales los más comúnmente procesados son:

WM_LBUTTONDOWN	// Se presionó el botón izquierdo del ratón
WM_LBUTTONUP	// Se soltó el botón izquierdo del ratón
WM_LBUTTONDOWNBLCLK	// Se presionó dos veces seguidas el botón izquierdo del ratón
WM_MBUTTONDOWN	// Similar a los anteriores pero para el botón del medio (M)
WM_MBUTTONUP	// ...
WM_MBUTTONDOWNBLCLK	// ...
WM_RBUTTONDOWN	// Similar a los anteriores pero para el botón derecho (R)
WM_RBUTTONUP	// ...
WM_RBUTTONDOWNBLCLK	// ...
WM_MOUSEACTIVATE	// Se pres. un botón del ratón estando sobre una ventana inactiva
WM_MOUSEHOVER	// Como consecuencia a una llamada a la función TrackMouseEvent.
WM_MOUSELEAVE	// Similar al anterior (ver documentación de dicha función)
WM_MOUSEMOVE	// El ratón se mueve sobre una ventana
WM_MOUSEWHEEL	// La rueda del ratón (si la hay) se ha rotado

La constante WM_KEYUP corresponde al mensaje producido por el evento de soltar (UP) una tecla del teclado (KEY). Para este mensaje, el parámetro wParam contiene el código de la tecla presionada. Sólo los códigos correspondientes a los números (0x30 al 0x39, '0' al '9') y las letras mayúsculas (0x41 a 0x5A, 'A' al 'Z') coinciden con la tabla ASCII. Toda tecla del teclado tiene un código distinto, existiendo constantes en Windows.h para cada una de ellas. Como ejemplo, podríamos reconocer si la tecla presionada fue F1 utilizando:

```
if ( wParam == VK_F1 ) { ... }
```

El parámetro lParam se comporta como un bit-flag con información como “es una tecla correspondiente al conjunto de teclas extendidas”. Hay 15 mensajes relacionados con el teclado, de los cuales los más comúnmente procesados son:

```
WM_KEYDOWN  
WM_KEYUP  
WM_CHAR  
WM_DEADCHAR
```

Manejo de Eventos con Interfaces en Java

En Java, la estrategia de manejo de los mensajes del sistema operativo, corresponde a un patrón de diseño de software conocido como “Patrón Observador”.

Bajo este patrón, existen dos objetos: El observador y el sujeto. El sujeto es una fuente de eventos (que pueden corresponder a mensajes del sistema operativo u otros producidos internamente por el programa) susceptibles de ser observados por objetos que cumplen con las características requeridas para ser observadores de dichos eventos.

Para que el objeto observador pueda realizar su trabajo, debe “registrarse” en el sujeto, esto es, notificarle de alguna forma su interés de observar ciertos eventos suyos. De esta forma, cuando el sujeto detecta que ha ocurrido un evento, notifica este hecho a todos los objetos observadores que se registraron para dicho evento. El siguiente programa muestra la aplicación de este patrón de diseño:

```
import java.util.Vector;  
  
class Observador {  
    public void notificar(String infoDelEvento) {  
        System.out.println("Sucedio el siguiente evento: " + infoDelEvento);  
    }  
}  
  
class Sujeto {  
    private Vector listaObservadores = new Vector();  
    public void registrarObservador(Observador ob) {  
        listaObservadores.add(ob);  
    }  
    public void simularEvento(String infoDelEvento) {  
        for(int i = 0; i < listaObservadores.size(); i++) {  
            Observador ob = (Observador)listaObservadores.get(i);  
            ob.notificar(infoDelEvento);  
        }  
    }  
}  
  
class PatronObservador {  
    public static void main(String args[]) {  
        Observador ob = new Observador();  
        Sujeto suj = new Sujeto();  
        suj.registrarObservador(ob);  
        suj.simularEvento("Evento1");  
        suj.simularEvento("Evento2");  
    }  
}
```

El programa crea un objeto observador y un sujeto observable, para luego simular que dos eventos ocurren. Note que el proceso de registro consiste simplemente en agregar la referencia al objeto observador a una lista del sujeto. Note además que al ocurrir el evento simulado, lo que hace el sujeto es recorrer la lista de referencias a objetos que se registraron como observadores, llamando a un método para cada uno de estos. Ésta es la forma en que el sujeto notifica al observador, llamando a un método de este último. Finalmente note que el método “notificar” de la clase observador es el acuerdo entre ambas partes, sujeto y observador, para que la notificación sea posible, es decir, todo observador de un objeto de mi clase Sujeto debe ser un objeto de la clase Observador o bien heredar de él, de forma que se garantice que dicho método existe.

El ejemplo anterior tiene un problema: Sólo podemos hacer que los objetos de una ClaseX observen los eventos de mi clase Sujeto, si mi ClaseX hereda de Observador. Esto es indeseable si pensamos que lenguajes como Java y C# no soportan herencia múltiple y, muy probablemente, deseáramos que un objeto, cuya clase padre no puedo modificar, escuche los eventos de otro. La solución a éste es aislar los métodos que forman parte del acuerdo en una interfaz. El siguiente programa modifica el anterior de forma que se utilice una interfaz en lugar de una clase:

```
import java.util.Vector;

interface IObservador {
    void notificar(String infoDelEvento);
}

class Sujeto {
    private Vector listaObservadores = new Vector();
    public void registrarObservador(IObservador ob) {
        listaObservadores.add(ob);
    }
    public void simularEvento(String infoDelEvento) {
        for(int i = 0; i < listaObservadores.size(); i++) {
            IObservador ob = (IObservador) listaObservadores.get(i);
            ob.notificar(infoDelEvento);
        }
    }
}

class ObservadorTipo1 implements IObservador {
    public void notificar(String infoDelEvento) {
        System.out.println("ObsevadorTipo1: Sucedió el siguiente evento: " + infoDelEvento);
    }
}

class ObservadorTipo2 implements IObservador {
    public void notificar(String infoDelEvento) {
        System.out.println("ObsevadorTipo2: Sucedió el siguiente evento: " + infoDelEvento);
    }
}

class PatronObservador2 {
    public static void main(String args[]) {
        ObservadorTipo1 ob1 = new ObservadorTipo1();
        ObservadorTipo2 ob2 = new ObservadorTipo2();
        Sujeto suj = new Sujeto();
        suj.registrarObservador(ob1);
        suj.registrarObservador(ob2);
        suj.simularEvento("Evento1");
        suj.simularEvento("Evento2");
    }
}
```

En el ejemplo anterior se tienen dos objetos, cada uno de una clase distinta pero que implementan la interfaz IObservador, que se registran para escuchar los eventos de un tercer objeto, uno de la clase Sujeto. Note que la implementación de la clase Sujeto se basa en la

interfaz IObservador y no en una clase. De esta forma se permite que los objetos de cualquier clase que implementen la interfaz IObservador sean utilizados como observadores de un objeto de la clase Sujeto. Ésta última es la estrategia que utiliza Java para sus eventos en ventanas.

En Java, el sujeto es un objeto ventana, una instancia de cualquier clase que herede de JFrame. Esta clase se comunica con una función de ventana internamente, la que procesa un subconjunto de todos los mensajes que se pueden generar con una ventana, llamando a los métodos adecuados de los objetos observadores registrados para dicho objeto ventana. Para esto, JFrame contiene listas de observadores, como datos miembros, para los distintos grupos de mensajes: Una lista para los mensajes de manipulación de la ventana, otra para los mensajes del ratón, otra para los mensajes del teclado, etc. De esta manera, cuando ocurre un evento, el mensaje es tratado por la función de ventana de JFrame, la que a su vez llama al método adecuado de cada objeto observador registrado para dicho mensaje.

En Java, las interfaces como IObservador en nuestro ejemplo, se llaman Listeners, y existe una definida para cada grupo de mensajes. Toda clase cuyos objetos se desea que puedan escuchar un evento de una ventana, debe de implementar la interfaz Listener adecuada. Veamos cómo esto se refleja, en el caso de los mensajes del ratón y del teclado, en el siguiente código.

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

class MiVentana extends JFrame {
    public MiVentana() {
        setSize(400, 400);
        setTitle("Titulo de la Ventana");
        setVisible(true);
        addWindowListener(new MiObservadorVentana(this));
        addKeyListener(new MiObservadorTeclado());
    }
}

class MiObservadorVentana implements WindowListener {
    MiVentana refVentana;
    public MiObservadorVentana(MiVentana refVentana) {
        this.refVentana = refVentana;
    }
    public void windowActivated(WindowEvent e) {
    }
    public void windowClosed(WindowEvent e) {
    }
    public void windowClosing(WindowEvent e) {
        refVentana.dispose();
        System.exit(0);
    }
    public void windowDeactivated(WindowEvent e) {
    }
    public void windowDeiconified(WindowEvent e) {
    }
    public void windowIconified(WindowEvent e) {
    }
    public void windowOpened(WindowEvent e) {
    }
}

class MiObservadorTeclado implements KeyListener {
    public void keyTyped(KeyEvent e) {
        displayInfo(e, "KEY TYPED: ");
    }
    public void keyPressed(KeyEvent e) {
        displayInfo(e, "KEY PRESSED: ");
    }
    public void keyReleased(KeyEvent e) {
        displayInfo(e, "KEY RELEASED: ");
    }
    private void displayInfo(KeyEvent e, String s){
```

```
String charString, keyCodeString, modString, tmpString;

char c = e.getKeyChar();
int keyCode = e.getKeyCode();
int modifiers = e.getModifiers();

if (Character.isISOControl(c)) {
    charString = "key character = "
        + "(an unprintable control character)";
} else {
    charString = "key character = "
        + c + "'";
}

keyCodeString = "key code = " + keyCode
    + " ("
    + KeyEvent.getKeyText(keyCode)
    + ")";

modString = "modifiers = " + modifiers;
tmpString = KeyEvent.getKeyModifiersText(modifiers);
if (tmpString.length() > 0) {
    modString += " (" + tmpString + ")";
} else {
    modString += " (no modifiers)";
}

System.out.println(s + "\n"
    + "    " + charString + "\n"
    + "    " + keyCodeString + "\n"
    + "    " + modString);
}

}

class EventosDeVentanas {
    public static void main(String args[]) {
        MiVentana ventana = new MiVentana();
    }
}
```

En el código anterior el sujeto observable es el objeto de clase `MiVentana` creado dentro del método `main`, y los observadores son dos: Un objeto de la clase `MiObservadorVentana`, implementando la interfaz `WindowListener`, y un objeto de la clase `MiObservadorTeclado`, implementando la interfaz `KeyListener`. Ambos observadores serán notificados de los eventos de la ventana y del teclado, respectivamente, que es capaz de detectar y/o producir el sujeto. Note además que el sujeto, conceptualmente y en la práctica, no requiere saber sobre la implementación interna de los objetos observadores, lo único que le concierne es que dichos objetos poseen los métodos adecuados para, mediante éstos, poderles notificar que ocurrió un evento del tipo para el que se registraron. Es por ello que Java utiliza interfaces para definir dicho contrato, los métodos que el objeto observador debe implementar y el objeto observable debe llamar. Note también que los métodos de registro siguen un formato común y forman parte de la interfaz que ofrece el sujeto, en este caso, los métodos `addWindowListener` y `addKeyListener`. Como es de esperarse, estos métodos reciben como parámetros referencias a objetos que implementen las interfaces respectivas.

La implementación de la interfaz de un evento particular requiere ser completa, si no lo fuera, la clase sería abstracta y no podríamos pasarle un objeto instanciado de dicha clase al método de registro respectivo. Sin embargo, es posible que no se requiera utilizar todos los métodos de la interfaz para un programa en particular, como es el caso, en el código anterior, de la clase “`MiObservadorVentana`”. Debido a esto, muchas interfaces relacionadas a eventos en Java tienen una clase que las implementa, a la que se le llama “Adaptador”. La siguiente clase es el

adaptador de la interfaz `WindowListener` (definido en el mismo paquete `java.awt.event` con dicha interfaz):

```
public abstract class WindowAdapter implements EventListener, WindowFocusListener,
WindowListener, WindowStateListener {
```

```
    // métodos de la interfaz WindowListener
    public void windowActivated(WindowEvent e) {
    }
    public void windowClosed(WindowEvent e) {
    }
    public void windowClosing(WindowEvent e) {
    }
    public void windowDeactivated(WindowEvent e) {
    }
    public void windowDeiconified(WindowEvent e) {
    }
    public void windowIconified(WindowEvent e) {
    }
    public void windowOpened(WindowEvent e) {
    }

    // métodos de las demás interfaces
    ...
}
```

Los adaptadores le dan una implementación vacía a las interfaces que implementan y son declarados como abstractos únicamente porque se espera que sirvan como clases base para otras clases que sobrescriban los métodos de las interfaces que requieran. Un ejemplo del uso de un adaptador puede verse en el primer código de ejemplo de Java, en la sección 3.1. Creación de una Ventana. En dicho código se utiliza el adaptador `WindowAdapter` como base de una clase anidada anónima.

Es importante señalar que no existe ningún impedimento para que el sujeto sea a su vez observador de otros sujetos o de sí mismo (como en el ejemplo de la sección 3.1), que un mismo observador pueda observar varios sujetos a la vez (del mismo tipo o de diferente tipo, de uno o más sujetos) y que un mismo evento sea observado por muchos observadores. Para este último caso podríamos, en el ejemplo anterior, haber registrado otros objetos para los mismos eventos (llamando más de una vez a `addWindowListener` y `addKeyListener` respectivamente) de manera que cuando dichos eventos ocurran, el sujeto, uno de la clase “`MiVentana`”, llamará en secuencia a los métodos correspondientes de todas los objetos registrados para dicho evento, en el orden en que se registraron.

Así como un objeto se registra para escuchar un evento, también se puede desregistrar. Para ésto existen los correspondientes métodos `removeXXXListener`, como son `removeWindowListener` y `removeKeyListener`.

Note que todos los métodos de una interfaz `Listener` reciben como parámetro una referencia de una clase `XXXEvent`, la que encapsula los datos del mensaje y provee de una interfaz para su fácil uso. En el caso de la interfaz `WindowListener` es `WindowEvent`, en el caso de la interfaz `KeyListener` es `KeyEvent`.

La Tabla 6 - 1 resume las clases involucradas en tres tipos de eventos comúnmente manejados en Java:

Tabla 6 - 1 Clases relacionadas con eventos en Java

Evento	Listener	Método de registro en JFrame	Adaptador	Parámetro de los métodos de la interfaz
Ventana	WindowListener	addWindowListener	WindowAdapter	WindowEvent
Teclado	KeyListener	addKeyListener	KeyAdapter	KeyEvent
Ratón	MouseListener	addMouseListener	MouseAdapter	MouseEvent

Manejo de Eventos con Delegados en C#

Los delegados son clases especiales en .NET que manejan internamente una referencia a un método de una clase, de manera que puede llamarla directamente. Es el equivalente a un puntero a función de C o C++, pero sin permitir un acceso directo a dicho puntero o referencia. Los delegados tienen un formato especial de declaración:

```
[Modificadores] delegate <tipo de retorno> <nombre>( [<Lista de Parámetros.>] );
```

La declaración de un delegado es muy similar a la de un método, pero con la palabra “delegate” entre los modificadores y el tipo de valor de retorno. Es importante tener en cuenta que esta declaración corresponde a un “tipo de dato”, no a una variable. Dado que esta declaración corresponde a un tipo de dato más, la declaración de variables de este tipo y su inicialización siguen las mismas reglas conocidas para las clases. El siguiente ejemplo muestra la creación de un objeto delegado y su uso.

```
using System;

class OtraClase {
    public static int Metodo3(string sMensaje) {
        Console.WriteLine("    OtraClase.Metodo3 : " + sMensaje);
        return 3;
    }
    public int Metodo4(string sMensaje) {
        Console.WriteLine("    OtraClase.Metodo4 : " + sMensaje);
        return 4;
    }
}

class Principal {

    private delegate int MiDelegado(string sMensaje);
    static event MiDelegado evento;

    private static int Metodo1(string sMensaje) {
        Console.WriteLine("    Principal.Metodo1 : " + sMensaje);
        return 1;
    }
    private int Metodo2(string sMensaje) {
        Console.WriteLine("    Principal.Metodo2 : " + sMensaje);
        return 2;
    }

    public static void Main(string[] args) {
        ///////////////////////////////////
        // Prueba con delegados

        Console.WriteLine("Prueba con delegados");
    }
}
```



```
MiDelegado delegado;

delegado = new MiDelegado(Metodo1);
Console.WriteLine("Llamando al delegado ...");
Console.WriteLine("    retorno = " + delegado("mensaje1"));

Principal refPrincipal = new Principal();
delegado = new MiDelegado(refPrincipal.Metodo2);
Console.WriteLine("Llamando al delegado ...");
Console.WriteLine("    retorno = " + delegado("mensaje2"));

delegado = new MiDelegado(OtraClase.Metodo3);
Console.WriteLine("Llamando al delegado ...");
Console.WriteLine("    retorno = " + delegado("mensaje3"));

OtraClase refOtraClase = new OtraClase();
delegado = new MiDelegado(refOtraClase.Metodo4);
Console.WriteLine("Llamando al delegado ...");
Console.WriteLine("    retorno = " + delegado("mensaje4"));

//////////////////////////
// Prueba con eventos

Console.WriteLine("Prueba con eventos");

evento += new MiDelegado(Metodo1);
evento += new MiDelegado(refPrincipal.Metodo2);
evento += new MiDelegado(OtraClase.Metodo3);
evento += new MiDelegado(refOtraClase.Metodo4);
Console.WriteLine("Llamando al evento ...");
Console.WriteLine("    retorno = " + evento("mensaje del evento"));
}
}
```

La salida de este programa es:

```
Prueba con delegados
Llamando al delegado ...
    Principal.Metodo1 : mensaje1
    retorno = 1
Llamando al delegado ...
    Principal.Metodo2 : mensaje2
    retorno = 2
Llamando al delegado ...
    OtraClase.Metodo3 : mensaje3
    retorno = 3
Llamando al delegado ...
    OtraClase.Metodo4 : mensaje4
    retorno = 4

Prueba con eventos
Llamando al evento ...
    Principal.Metodo1 : mensaje del evento
    Principal.Metodo2 : mensaje del evento
    OtraClase.Metodo3 : mensaje del evento
    OtraClase.Metodo4 : mensaje del evento
    retorno = 4
```

La declaración de una variable tipo delegado y su inicialización es similar a la de cualquier otra clase, excepto por el parámetro de su constructor. Note que si el método pasado como parámetro no es estático, se requiere contar con una referencia a un objeto de la clase que contiene dicho método, al que se desea llamar.

El método pasado como parámetro, al crear un objeto delegado, debe tener el mismo formato de declaración del delegado. Para el ejemplo anterior, tanto los métodos llamados, como la declaración del tipo delegado, reciben como parámetro una referencia a un objeto `System.String` y retornan un valor `System.Int32`.

Luego de inicializar una variable de tipo delegado, su uso es igual al de un puntero a función de C o C++, utilizando el nombre de la variable como si se tratara del nombre de un método.

Los objetos delegados utilizados en el código anterior, sólo nos permiten llamar a un único método a la vez. Sin embargo, es posible utilizar un objeto delegado enlazado con otros objetos delegados, de forma que se pueda llamar a más de una función. A este tipo de delegado se le llama **MulticastDelegate**. El uso de este tipo de delegados cae fuera del alcance del presente curso.

Sin embargo, como puede verse en el código anterior, una forma simple de llamar a un grupo de métodos es utilizando la palabra clave event. El formato de declaración de una variable event es:

```
[Modificadores] event <Nombre del Delegado> <Nombre>;
```

Esta declaración sólo puede ir en el ámbito de una clase, no dentro de un método. Esto se debe a que, a pesar de parecerse a la declaración de una variable, al agregarle la palabra event a la declaración, la sentencia se expande al ser compilado el código, generándose la declaración de una variable delegado, del tipo puesto en la declaración, y dos métodos, add_XXX y remove_XXX (donde XXX es el nombre del tipo del delegado). Para el código anterior, esta expansión sería de la forma:

```
private static MiDelegado evento = null;
private static MiDelegado add_MiDelegado(...) {}
private static MiDelegado remove_MiDelegado(...) {}
```

Estos métodos add y remove son llamados automáticamente cuando se utilizan los operadores '+' y '-', respectivamente, con la variable evento. Debido a esto, la variable declarada con la palabra event no requiere ser inicializada.

En .NET se utilizan los conceptos de delegado y evento para manejar la respuesta a la interacción del usuario con las ventanas del programa. Para cada tipo de evento que el usuario pueda generar, existen en las clases de .NET propiedades que encapsulan variables event, así como los tipos de delegados correspondientes. El siguiente código muestra un ejemplo del uso de estos eventos y delegados:

```
using System;
// Al siguiente espacio de nombres pertenecen:
// Form, KeyPressEventHandler, KeyPressEventArgs,
// MouseEventHandler, MouseEventArgs, PaintEventHandler, PaintEventArgs.
using System.Windows.Forms;

class Ventana : Form {
    public Ventana() {
        this.Size = new System.Drawing.Size(400, 400);
        this.Text = "Título de la Ventana";
        this.Visible = true;
        this.MouseUp += new MouseEventHandler(this.Ventana_MouseUp);
        this.MouseLeave += new EventHandler(this.Ventana_MouseLeave);
        this.KeyPress += new KeyPressEventHandler(this.Ventana_KeyPress);
    }
    private void Ventana_MouseUp(object sender, MouseEventArgs e) {
        MessageBox.Show("Evento MouseUp");
    }
    private void Ventana_MouseLeave(object sender, System.EventArgs e) {
        MessageBox.Show("Evento MouseLeave");
    }
    private void Ventana_KeyPress(object sender, KeyPressEventArgs e) {
        MessageBox.Show("Evento KeyPress");
    }
}

class Eventos_Ventana {
    public static void Main(string[] args) {
        Ventana refVentana = new Ventana();
    }
}
```

```
        Application.Run(refVentana);  
    }  
}
```

En el código anterior, se hace uso de las propiedades MouseUp, MouseLeave, KeyPress y Paint, las que nos permiten acceder a datos miembros internos declarados como event para los tipos de delegado MouseEventHandler, EventHandler, KeyPressEventHandler y PaintEventHandler respectivamente.

Es importante tener en cuenta que las variables declaradas como event son únicamente una forma de simplificar el trabajo con los delegados, cuando se desea tener la posibilidad de llamar a más de un método cuando un evento sucede.

Aunque enfocado en forma distinta a Java, el uso de delegados es realmente la implementación del mismo patrón de diseño, el patrón Observador. La única diferencia está en que, mientras en Java el objeto observador implementa una interfaz para que el sujeto observable le notifique de un evento llamando a los métodos de ésta, en .NET se utiliza un puntero a función encapsulado en una clase especial.

Tipos de Eventos

Los eventos con los que interactúa un programa son comunmente los producidos como consecuencia de un mensaje del sistema operativo, en particular los relacionados con el ratón, el teclado y con el manejo de la ventana.

Un programa también puede definir sus propios eventos o simular los ya existentes como una manera de independizar el programa de las capacidades del sistema operativo subyacente (como es el caso de Java para muchas de sus clases, del paquete Swing, con representación visual).

Un programa también puede disparar eventos al detectar que sucesos no visuales se producen en su entorno, como por ejemplo: El arribo de un paquete de datos por red, la recepción de un mensaje enviado desde otro programa, la baja del nivel de algún recurso por debajo del límite crítico (como la memoria), etc.

Gráficos en 2D

El manejo de los dispositivos gráficos en Windows se realiza mediante la librería GDI32 (Graphics Device Interface). Esta librería aísla las aplicaciones de las diferencias que existen entre los dispositivos gráficos con los que puede interactuar un computador.

El siguiente diagrama muestra el flujo de comunicación entre las aplicaciones en ejecución, la librería gráfica de Windows, las librerías por cada dispositivo y los dispositivos mismos.

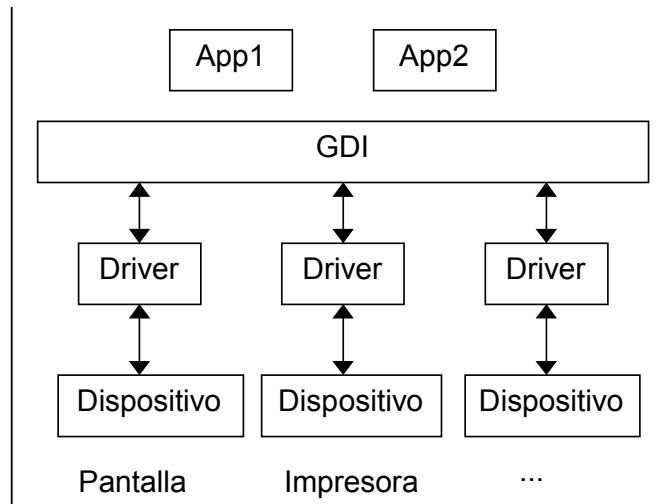


Figura 6 - 3 Flujo de comunicación entre aplicaciones y la librería GDI

Como se puede apreciar, las aplicaciones interactúan únicamente con la librería GDI. Una vez que una aplicación le indica a GDI con qué tipo de dispositivo desea interactuar, dicha interacción se realiza en forma independiente al dispositivo elegido. Las librerías por cada dispositivo se conocen como Drivers, y deben cumplir la especificación requerida por GDI para que puedan interactuar con él. De esta forma, cada fabricante de un nuevo dispositivo, que desee que éste sea utilizado desde Windows deberá proveer un Driver que sepa cómo manejar su dispositivo y que cumpla las especificaciones de GDI.

Un punto importante es ¿qué sucede cuando una aplicación le indica a GDI con qué tipo de dispositivo desea interactuar? La GDI busca si dicho dispositivo (es decir, su driver) existe, lo carga a memoria si no estuviese ya cargado, crea una entrada en la tabla de recursos para el nuevo recurso a utilizar (el dispositivo), llena dicha entrada y retorna al programa un handle al nuevo dispositivo creado. Dicha entrada en la tabla de recursos contiene:

- Información sobre el dispositivo mismo, su tipo, sus capacidades, etc.
- Información sobre su estado actual, lo que puede incluir referencias a otros recursos utilizados cuando la aplicación desea interactuar con el dispositivo.

A dicha información en conjunto se le llama Contexto del Dispositivo (Device Context), por lo que el tipo de su handle relacionado es HDC (Handle Device Context).

De lo anterior se resume que, para que una aplicación pueda interactuar con un dispositivo debe de obtener un HDC adecuado para dicho dispositivo. Al igual que con otros handles, la aplicación deberá liberar dicho HDC cuando ya no requiera trabajar más con él.

Existen 5 formas de dibujo bastante comunes (no son las únicas):

- Síncrono, donde las acciones de dibujo se realizan en cualquier lugar de la aplicación.
- Asíncrono, donde las acciones de dibujo se realizan en un lugar bien definido de la aplicación.
- Sincronizado, cuando el dibujo asíncrono se “sincroniza” con otras acciones en cualquier lugar de la aplicación.
- En Memoria, donde las acciones de dibujo se realizan en un lugar de la memoria distinta a la memoria de video.

- En Impresora, donde las acciones de dibujo se traducen en comandos enviados a una impresora.

En los tres primeros casos, los HDC que se obtendrían corresponden al área cliente de una ventana, siendo el dispositivo gráfico un monitor de computadora. En el penúltimo caso, el dispositivo gráfico es un espacio de memoria fuera de la memoria de video.

A continuación veremos cómo se realizan los distintos tipos de dibujo para los diferentes lenguajes utilizados. Es importante mantener siempre presente la idea de que, sin importar en qué lenguaje se trabaje, siempre se debe utilizar un HDC para dibujar sobre un dispositivo gráfico.

Dibujo con API de Windows

En esta sección se verá las diferentes formas de dibujo que permite la librería de Windows. Es importante tener en consideración que la implementación en Windows de las librerías de dibujo tanto en Java como C# utilizan internamente esta API para realizar su trabajo.

Funciones de Dibujo

A continuación se explica algunas de las funciones de dibujo de la librería GDI:

La función “DrawText” utiliza la fuente de letra, color de texto y color de fondo actual del DC, para dibujar un texto. Su prototipo es:

```
int DrawText(  
    HDC hDC,          // handle al DC  
    char* texto,      // texto a dibujar  
    int longitud,     // longitud del texto  
    RECT* area,       // rectángulo dentro del que se dibujará el texto  
    UINT opciones     // opciones de dibujo del texto  
);
```

Un ejemplo de uso sería:

```
RECT rc = {20, 50, 0, 0};  
DrawText( hDC, "hola", -1, &rc, DT_NOCLIP);
```

El código anterior dibujaría la cadena “hola” sobre el DC referido con el handle “hDC”. Utilizamos la opción DT_NOCLIP dado que no nos interesa restringir la salida del texto a un rectángulo específico. Por el mismo motivo sólo especificamos la posición X e Y inicial del texto en la variable “rc”. La estructura RECT se define como:

```
struct RECT { long left, top, right, bottom; };
```

La función MoveToEx establece el punto inicial de dibujo de líneas sobre un DC. A partir de dicha posición se realizará dibujos de líneas hacia otras posiciones, con funciones como LineTo. Cada nueva línea dibujada actualiza la posición actual de dibujo de líneas. Los prototipos de MoveToEx y LineTo son:

```
BOOL MoveToEx(  
    HDC hdc,          // handle al DC  
    int Xinicial,     // coordenada-x de la nueva posición  
                    // (la que se convertirá en la actual)  
    int Yinicial,     // coordenada-y de la nueva posición  
                    // (la que se convertirá en la actual)  
    POINT* PosAntigua // recibe los datos de la antigua posición actual  
);  
  
BOOL LineTo(  
    HDC hdc,          // handle al DC  
    int Xfinal,       // coordenada-x del punto final  
    int Yfinal        // coordenada-y del punto final  
);
```

Un ejemplo de uso sería:

```
MoveToEx( hDC, 10, 10, NULL );  
LineTo( hDC, 40, 10 );  
LineTo( hDC, 40, 40 );  
LineTo( hDC, 10, 10 );
```

El código anterior dibujaría un triángulo con vértices (10,10), (40,10) y (40,40). La posición actual de dibujo, para subsiguientes dibujos de líneas, quedaría en la coordenada (10,10).

Es importante señalar que todas las acciones de dibujo sobre un DC se realizan en base al sistema de coordenadas del mismo. En el caso de un DC relativo al área cliente de una ventana, el origen de su sistema de coordenadas en la esquina superior izquierda del área cliente, con el eje positivo X avanzando hacia la derecha, y el eje positivo Y avanzando hacia abajo.

Dibujo Asíncrono

Las acciones de dibujo se centralizan en el procesamiento del mensaje WM_PAINT. El siguiente código muestra un esquema típico de procesamiento de este mensaje:

```
case WM_PAINT:  
    hDC = BeginPaint(hWnd, &ps);  
    // Aquí van las acciones de dibujo  
    EndPaint(hWnd, &ps);  
    break;
```

La función BeginPaint retorna un HDC adecuado para dibujar sobre el área cliente invalidada de una ventana. La función EndPaint libera el HDC obtenido. Para entender el concepto de invalidación, imagine el siguiente caso:

- Se tiene en un momento dado dos ventanas mostradas en pantalla. La primera oculta parte de la segunda, como se muestra en la siguiente figura:

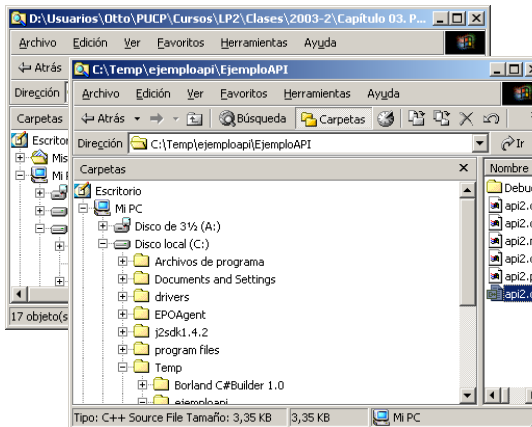


Figura 6 - 4 Dibujo asíncrono: Ventana ocultando otra ventana

- Luego se mueve la primera ventana de forma que descubre parte o toda el área oculta de la segunda ventana, como se muestra en la siguiente figura:

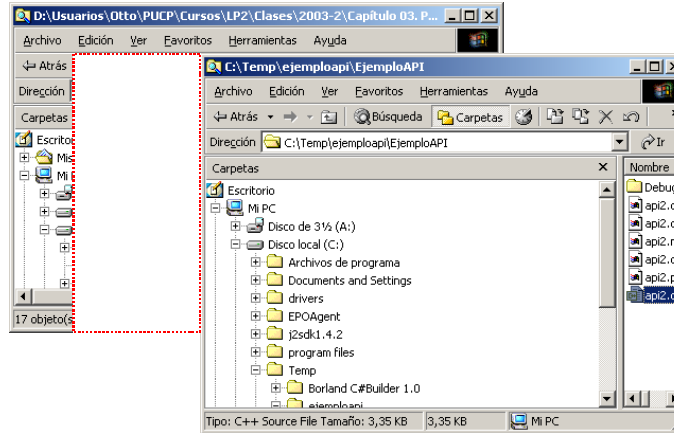


Figura 6 - 5 Dibujo asíncrono: Ventana descubriendo otra ventana

- El dibujo actual del área descubierta ya no es válido y debe de ser redibujado por el código de la aplicación correspondiente a la segunda ventana, dado que Windows no tiene forma de saber cómo se debe dibujar cada ventana de cada aplicación, sólo las aplicaciones mismas lo saben. Sin embargo, Windows sí reconoce que esta invalidación ha ocurrido, dado que sabe dónde se encuentra cada ventana y cuál está frente a cual, por lo que genera un mensaje WM_PAINT, con la información acerca del rectángulo invalidado, y lo deposita en la cola de mensajes de la aplicación a la que le corresponde dicha ventana invalidada.
- Finalmente, cuando el bucle de procesamiento de mensajes correspondiente extraiga y mande a procesar dicho mensaje WM_PAINT, la función de ventana de la ventana invalidada repintará el área de dibujo inválido.

La estructura PAINTSTRUCT es llenada por BeginPaint con información acerca del área invalidada. Esta información podría ser utilizada por el programa para aumentar la eficiencia del código de dibujo, dado que podría repintar solamente el área invalidada y no repintar toda el área cliente. Uno de los datos miembros de dicha estructura es el mismo valor retornado por BeginPaint. La función EndPaint utiliza dicho dato para eliminar el DC.

Los mensajes WM_PAINT son generados en forma automática por el sistema operativo cuando éste sabe que el área cliente de una ventana requiere repintarse. Si al generarse un mensaje WM_PAINT para una ventana, ya existe en la cola de mensajes otro WM_PAINT para la misma ventana, se juntan ambos mensajes en uno solo para un área invalidada igual a la combinación de las áreas invalidadas de ambos mensajes.

También es posible generar un mensaje WM_PAINT manualmente y colocarlo en la cola de mensajes respectiva de forma que se repinte la ventana. La función que hace ésto es:

```
BOOL InvalidateRect(  
    HWND hWnd,           // handle de la ventana  
    CONST RECT* lpRect,  // rectángulo a invalidar  
    BOOL bErase           // flag de limpiado  
);
```

El segundo parámetro es el rectángulo, dentro del área cliente, que deseamos invalidar. El tercer parámetro le sirve a la función BeginPaint. Si dicho parámetro es 1, BeginPaint pinta toda el área invalidada utilizando la brocha con la que se creó la ventana (dato miembro hbrBackground de la estructura WNDCLASS) antes de finalizar y retornar el HDC (de forma que se comience con un área de dibujo limpia). Si dicho parámetro es 0, BeginPaint no realiza este limpiado.

El siguiente código muestra el uso de esta función:

```
...
case WM_LBUTTONDOWN:
    iContador++;
    InvalidateRect(hWnd, NULL, TRUE);
    break;
case WM_RBUTTONDOWN:
    for(iBucle = 0; iBucle < 10; iBucle++) {
        iContador++;
        InvalidateRect(hWnd, NULL, TRUE);
    }
    break;
case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    sprintf(szMensaje, "Contador=%d", iContador);
    DrawText(hDC, szMensaje, -1, &rc, DT_NOCLIP);
    EndPaint(hWnd, &ps);
    break;
...
```

El fragmento de código anterior pertenece a una función de ventana que utiliza `InvalidateRect`. Cuando se presiona con el botón izquierdo del ratón se modifica un contador de forma que el dibujo actual ya no es correcto y debe repintarse. Para el botón derecho se desea que se muestre, como una secuencia animada, cómo se va modificando el contador. Sin embargo, esta secuencia no se muestra, y sólo se ve el valor final del contador en la ventana. Esto se debe al hecho que `InvalidateRect` “no es” una llamada directa al código en `WM_PAINT`, sino la colocación de un mensaje `WM_PAINT` en la cola de mensajes, por lo que sólo al retornar del procesamiento del mensaje actual, `WM_RBUTTONDOWN` en este caso, el bucle de procesamiento de mensajes podrá retirar el `WM_PAINT` de la cola de mensajes y procesarlo.

Como puede apreciarse, el dibujo realizado en `WM_PAINT` es asíncrono respecto a la solicitud de dibujo realizada en `WM_RBUTTONDOWN`. Este tipo de dibujo es adecuado para dibujos estáticos o de fondo, pero no para secuencias de animación.

Dibujo Síncrono

Para realizar dibujos desde cualquier otro lugar fuera del procesamiento del mensaje `WM_PAINT` se utiliza la combinación de funciones:

```
// Obtener un DC para el área cliente de la ventana referida con el handle hWnd
HDC GetDC( HWND hWnd );
// Liberar el DC obtenido con GetDC
int ReleaseDC( HWND hWnd, HDC hDC );
```

Para obtener un `HDC` relativo al área cliente de una ventana se utiliza la función `GetDC`. Una vez que se han finalizado las acciones de dibujo sobre la ventana, se debe liberar el `HDC` obtenido llamando a `ReleaseDC`.

El siguiente segmento de un programa muestra el uso de esta técnica:

```
...
case WM_LBUTTONDOWN:
    hDC = GetDC(hWnd);
    iContador++;
    sprintf(szMensaje, "Contador=%d", iContador);
    DrawText(hDC, szMensaje, -1, &rc, DT_SINGLELINE);
    ReleaseDC(hWnd, hDC);
    break;
case WM_RBUTTONDOWN:
    hDC = GetDC(hWnd);
    for(iBucle = 0; iBucle < 10; iBucle++) {
        iContador++;
        sprintf(szMensaje, "Contador=%d", iContador);
        DrawText(hDC, szMensaje, -1, &rc, DT_SINGLELINE);
        Sleep(100);
    }
    ReleaseDC(hWnd, hDC);
    break;
...
```



```
}  
ReleaseDC(hWnd, hDC);  
break;  
...
```

A diferencia del caso anterior, el HDC creado puede utilizarse en cualquier lugar del programa. Este HDC debe ser liberado cuando ya no sea requerido.

Dibujo Sincronizado

El dibujo sincronizado permite realizar modificaciones al estado del dibujo en cualquier parte del programa, concentrando el trabajo de dibujo dentro del mensaje de pintado WM_PAINT.

El siguiente segmento de un programa muestra el uso de esta técnica:

```
...  
case WM_LBUTTONDOWN:  
    iContador++;  
    InvalidateRect(hWnd, NULL, TRUE);  
    UpdateWindow(hWnd); // acá se fuerza el procesamiento del mensaje WM_PAINT  
                           // colocado en la cola de mensajes por  
InvalidateRect  
    break;  
case WM_RBUTTONDOWN:  
    for(iBucle = 0; iBucle < 10; iBucle++) {  
        iContador++;  
        InvalidateRect(hWnd, NULL, TRUE);  
        UpdateWindow(hWnd);  
        Sleep(100);    }  
    break;  
case WM_PAINT:  
    hDC = BeginPaint(hWnd, &ps);  
    sprintf(szMensaje, "Contador=%d", iContador);  
    DrawText(hDC, szMensaje, -1, &rc, DT_SINGLELINE);  
    EndPaint(hWnd, &ps);  
    break;  
...
```

Dibujo en Memoria

Cuando el dibujo se debe realizar en varios pasos, el realizarlo directamente sobre la ventana provoca que el usuario del programa vea todo el proceso de dibujo, produciéndose en algunos casos el parpadeo de la imagen. En estos casos es posible realizar la composición del dibujo en memoria, para luego pasar la imagen final a la ventana en una sola acción.

El siguiente segmento de un programa muestra el uso de esta técnica:

```
// Luego de creada la ventana  
  
hBM_Fondo = (HBITMAP)LoadImage(hInstApp, "Fondo.bmp",  
    IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE | LR_DEFAULTSIZE);  
hDC_Fondo = CreateCompatibleDC(hDC);  
hBM_Original = (HBITMAP)SelectObject(hDC_Fondo, hBM_Fondo);  
szMensaje = "Mensaje de Texto";  
TextOut(hDC_Fondo, 20, 20, szMensaje, strlen(szMensaje));  
  
ShowWindow(hWnd, iShowCmd);  
UpdateWindow(hWnd);  
  
// Luego del bucle de procesamiento de mensajes  
  
SelectObject(hDC_Fondo, hBM_Original);  
DeleteDC(hDC_Fondo);  
DeleteObject(hBM_Fondo);  
  
// En la función de ventana  
  
HBITMAP hBM_Fondo = 0;  
HDC hDC_Fondo = 0;
```

```
LRESULT CALLBACK FuncionVentana( ... ) {
    HDC hDC; PAINTSTRUCT ps; char * szMensaje;

    switch( uMsg ) {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &ps);
        if(hBM_Fondo != 0 && hDC_Fondo != 0)
            BitBlt(hDC, 0, 0, 800, 600, hDC_Fondo, 0, 0, SRCCOPY);
        else {
            szMensaje = "Error al cargar la imagen";
            TextOut(hDC_Fondo, 20, 20, szMensaje, strlen(szMensaje));
        }
        EndPaint(hWnd, &ps);
        break;
        ...
    }
    return 0;
}
```

Luego de creada la ventana se realiza lo siguiente:

- Crear un DC en memoria en base a otro DC, típicamente, en base al DC de una ventana.
- Crear un área de dibujo en memoria, ésto es, un BITMAP.
- Seleccionar el bitmap en el DC en memoria.
- Realizar los dibujos respectivos en el DC en memoria.

Luego del bucle de mensajes, donde la ventana ya fue destruida, se realiza lo siguiente:

- Se restaura el DC en memoria seleccionándole en bitmap con el que se creó.
- Destruir el DC de memoria.
- Destruir el bitmap.

Dado que el DC en memoria y el bitmap se utilizarán en la función de ventana, es conveniente definir sus variables como globales.

Dentro del procesamiento del mensaje WM_PAINT la secuencia de pasos suele ser la siguiente:

- Se obtiene un handle a un DC para el área cliente de la ventana: BeginPaint
- Utilizando dicho handle se llaman a las funciones de dibujo del API de Windows para dibujar sobre la ventana.
- Se libera el DC: EndPaint

Dibujo en Java

Java provee, a partir de su versión 1.2, el paquete SWING que simplifica significativamente el trabajo con ventanas. Esta librería está formada en su mayoría por componentes ligeros, de forma que se obtenga la máxima portabilidad posible de las aplicaciones con ventanas hacia las diferentes plataformas que soportan Java.

El paquete SWING se basa en el paquete AWT que fue desarrollado con las primeras versiones de Java. Para los ejemplos en las siguientes secciones, se realizará dibujo en 2D sobre la clase base para ventanas JFrame de SWING.

Dibujo Asíncrono

El dibujo en una ventana requiere sobrescribir el método “paint” de la clase “JFrame”. Dentro de este método se llama a la implementación de la clase base de paint y luego se realizan acciones de dibujo utilizando la referencia al objeto Graphics recibida. La clase Graphics contiene métodos adecuados para realizar:

- Dibujo de texto.
- Dibujo de figuras geométricas con y sin relleno.
- Dibujo de imágenes.

El escribir instrucciones de dibujo dentro del método paint equivale a hacerlo dentro del “case WM_PAINT” de la función de ventana en API de Windows. En su implementación para Windows, es de esperarse que la clase Graphics maneje internamente un HDC, obtenido mediante una llamada a “BeginPaint”. El siguiente programa muestra el dibujo asíncrono.

```
import ...
class Ventana extends JFrame {
    Point clicPos;
    public Ventana() {
        ...
        addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                clicPos = e.getPoint();
                repaint();
            }
        });
    }
    public void paint(Graphics g) {
        super.paint(g);
        if(clicPos != null)
            g.drawString("Clic en " + clicPos, 100, 100);
    }
}
...
```

Dibujo Síncrono

Para el dibujo síncrono se obtiene un objeto Graphics utilizando el método getGraphics de JFrame. Es importante que, si dicho método es llamado muy seguido, se libere los recursos del objeto Graphics obtenido (por ejemplo, el HDC obtenido mediante un GetDC para la implementación en Windows de esta clase) llamando al método “dispose” (que es de suponer debería llamar a ReleaseDC). Éste es un claro ejemplo donde el usuario debe preocuparse por liberar explícitamente los recursos dado que el recolector de basura puede no hacerlo a tiempo.

El siguiente programa muestra el dibujo síncrono.

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

class Ventana extends JFrame {
    public Ventana() {
        ...
        addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                Graphics g = getGraphics();
                g.drawString("HOLA", e.getX(), e.getY());
                g.dispose();
            }
        });
    }
}
...
```

En el programa anterior, cada vez que se realice un click con el botón del mouse, estando el mouse sobre el área cliente de la ventana, se dibujará el texto “HOLA” en dicha posición de la ventana.

Dibujo Sincronizado

Para sincronizar un dibujo se llama al método “update” de la clase JFrame y se concentra todo el dibujo en la sobrescritura del método “paint” de la ventana. El siguiente programa muestra el uso de “update”.

```
import ...
class Ventana extends JFrame {
    Point clicPos;
    public Ventana() {
        ...
        addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                Graphics g = getGraphics();
                clicPos = e.getPoint();
                update(g);
            }
        });
    }
    public void paint(Graphics g) {
        super.paint(g);
        if(clicPos != null)
            g.drawString("Clic en " + clicPos, 100, 100);
    }
}
...
```

En el programa anterior, cada vez que se realiza un click sobre la ventana se actualiza la variable de clase “clicPos” y se invalida la ventana llamando a “update”.

Dibujo en Memoria

Existen varias estrategias para realizar dibujo en memoria en Java, para cada cual un conjunto de clases adecuadas. Una de estas estrategias consiste en utilizar un objeto BufferedImage, el cual crea un espacio en la memoria sobre la cual se puede realizar un dibujo. Esta clase provee un método “getGraphics” que permite obtener un objeto Graphics adecuado para dibujar en esta memoria. Luego de compuesta la imagen en memoria, se puede utilizar el método “drawImage” del objeto Graphics en el método “paint” para dibujar dicha imagen en la ventana.

El siguiente programa muestra esta estrategia de dibujo.

```
import ...
import java.awt.image.*;
class Ventana extends JFrame implements ImageObserver {
    BufferedImage img;
    public Ventana() {
        ...
        img = javax.imageio.ImageIO.read(new File("Fondo.gif"));
        if(img != null) {
            Graphics g = img.createGraphics();
            g.fillOval(0, 0, 100, 100);
            g.dispose();
        }
    }
    public void paint(Graphics g) {
        ...
        g.drawImage(img, 0, 0, this);
    }
}
...
```

Dibujo en C#

En .NET las clases relacionadas con el dibujo sobre ventanas se encuentran dentro del espacio de nombres System.Drawing.

Dibujo Asíncrono

Realizar un dibujo síncrono sobre una ventana consiste en agregar un nuevo delegado al evento “Paint” de la clase Form. Dicho delegado deberá hacer referencia a un método que reciba como parámetro una referencia “object” y una referencia “PaintEventArgs”. Ésta última contiene las propiedades y métodos necesarios para realizar un dibujo sobre la ventana.

El siguiente programa muestra un ejemplo de este tipo de dibujo.

```
using System;
using System.Windows.Forms;
using System.Drawing;

class Ventana : Form {
    private Font drawFont = new Font("Arial", 16);
    private SolidBrush drawBrush = new SolidBrush(Color.Black);
    private Point clicPos = new Point(0, 0);
    public Ventana() {
        this.Size = new System.Drawing.Size(400, 400);
        this.Text = "Título de la Ventana";
        this.Visible = true;
        this.MouseUp += new MouseEventHandler(this.Ventana_MouseUp);
        this.Paint += new PaintEventHandler(this.Ventana_Paint);
    }
    private void Ventana_MouseUp(object sender, MouseEventArgs e) {
        clicPos = new Point(e.X, e.Y);
        Invalidate();
    }
    private void Ventana_Paint(object sender, PaintEventArgs e) {
        Graphics g = e.Graphics;
        g.DrawString("Clic en " + clicPos, drawFont, drawBrush, 10, 40);
    }
}

class DibujoSincronizado {
    public static void Main(string[] args) {
        Ventana refVentana = new Ventana();
        Application.Run(refVentana);
    }
}
```

Dibujo Síncrono

Para el dibujo síncrono se utiliza el método CreateGraphics de la clase Form desde cualquier punto del programa. Este método retorna un objeto Graphics (podemos suponer que internamente llama a GetDC) con el cual se puede dibujar sobre la ventana. Cuando ya no se requiera utilizar este objeto, el programa debe llamar al método “Dispose” del mismo, de forma que se liberen los recursos reservados por éste en su creación (podemos suponer que libera el HDC interno que maneja mediante un ReleaseDC).

El siguiente programa muestra el uso del dibujo síncrono.

```
using System;
using System.Windows.Forms;
using System.Drawing;

class Ventana : Form {
    private Font drawFont = new Font("Arial", 16);
    private SolidBrush drawBrush = new SolidBrush(Color.Black);
    public Ventana() {
        this.Size = new System.Drawing.Size(400, 400);
    }
}
```

```
        this.Text = "Título de la Ventana";
        this.Visible = true;
        this.MouseUp += new MouseEventHandler(this.Ventana_MouseUp);
    }
    private void Ventana_MouseUp(object sender, MouseEventArgs e) {
        Graphics g = this.CreateGraphics();
        g.DrawString("Hola", drawFont, drawBrush, e.X, e.Y);
        g.Dispose();
    }
}

class DibujoSincrono {
    public static void Main(string[] args) {
        Ventana refVentana = new Ventana();
        Application.Run(refVentana);
    }
}
```

Dibujo Sincronizado

Para sincronizar un dibujo se llama al método “Invalidate” de la clase Form y se concentra todo el dibujo en la sobrescritura del método donde se concentra el trabajo de dibujo asíncrono. El siguiente programa muestra el uso de “Invalidate”. El siguiente programa muestra el uso del dibujo sincronizado.

```
using System;
using System.Windows.Forms;
using System.Drawing;

class Ventana : Form {
    private Font drawFont = new Font("Arial", 16);
    private SolidBrush drawBrush = new SolidBrush(Color.Black);
    private Point clicPos = new Point(0, 0);
    public Ventana() {
        this.Size = new System.Drawing.Size(400, 400);
        this.Text = "Título de la Ventana";
        this.Visible = true;
        this.MouseUp += new MouseEventHandler(this.Ventana_MouseUp);
        this.Paint += new PaintEventHandler(this.Ventana_Paint);
    }
    private void Ventana_MouseUp(object sender, MouseEventArgs e) {
        clicPos = new Point(e.X, e.Y);
        Refresh();
    }
    private void Ventana_Paint(object sender, PaintEventArgs e) {
        Graphics g = e.Graphics;
        g.DrawString("Clic en " + clicPos, drawFont, drawBrush, 10, 40);
    }
}

class DibujoSincronizado {
    public static void Main(string[] args) {
        Ventana refVentana = new Ventana();
        Application.Run(refVentana);
    }
}
```

Dibujo en Memoria

Al igual que en Java, existen muchas estrategias de dibujo en memoria. El siguiente ejemplo crea un objeto de la clase “Image” que inicialmente contiene un dibujo guardado en un archivo. Dicho objeto crea un área en memoria, inicializada con la imagen leída del archivo, a la que puede accederse mediante un objeto Graphics creado mediante el método estático “FromImage” de la misma clase Graphics. Cuando dicho objeto Graphics ya no se requiera, el programa debe liberar sus recursos llamando al método “Dispose”. El siguiente programa muestra un ejemplo de este tipo de dibujo.

```
using System;
using System.Windows.Forms;
using System.Drawing;

class Ventana : Form {
    Image img;

    public Ventana() {
        this.Size = new System.Drawing.Size(400, 400);
        this.Text = "Título de la Ventana";
        this.Visible = true;
        this.Paint += new PaintEventHandler(Ventana_Paint);

        img = Image.FromFile("Fondo.bmp");
        Graphics g = Graphics.FromImage(img);
        Font f = this.Font;
        Brush b = Brushes.Black;
        g.DrawString("Es injusto que el coyote nunca alcance al correcaminos",
            f, b, 50, 30);
        g.Dispose();
    }
    public void Ventana_Paint(object sender, PaintEventArgs args) {
        Graphics g = args.Graphics;
        g.DrawImage(img, 0, 0);
    }
}

class DibujoEnMemoria {
    public static void Main() {
        Application.Run(new Ventana());
    }
}
```

Manejo de Elementos GUI

Las ventanas tienen un conjunto de elementos visuales que ocupan parte (o toda) su área cliente y cuyo comportamiento está bien estandarizado y es común a todos los programas que se ejecutan utilizando una misma librería gráfica. Algunos de estos elementos son: botones, cajas de texto, etiquetas, listas de selección, agrupadores, etc.

En esta sección veremos cómo se crean los elementos GUI más comunes, cómo se distribuyen en el área cliente y cómo se manejan los eventos que producen al interactuar el usuario con ellos.

Elementos GUI del API de Windows

En API de Windows, todos los elementos GUI son ventanas. Cada elemento se crea utilizando una clase de ventana preregistrada, y por consiguiente posee una función de ventana ya implementada en alguna de las librerías del API. El siguiente código muestra un ejemplo simple de creación de una ventana con una etiqueta, una caja de texto y un botón.

```
#include <windows.h>
#define ID_TEXTO 101
#define ID_BOTON 102

LRESULT CALLBACK FuncionVentana( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam ) {
    switch( uMsg ) {
        case WM_COMMAND:
            if( LOWORD( wParam ) == ID_BOTON ) {
                char szNombre[ 100 ];
                HWND hWndBoton = ( HWND )lParam;
                GetDlgItemText( hWnd, ID_TEXTO, szNombre, 100);
                MessageBox( hWnd, szNombre, "Hola", MB_OK );
            }
            break;
        // Aquí va el resto del switch
        ...
    }
}
```

```
    }
    return 0;
}

BOOL RegistrarClaseVentana( HINSTANCE hIns ) {
    ...
}

HWND CrearInstanciaVentana( HINSTANCE hIns ) {
    ...
}

int WINAPI WinMain( HINSTANCE hIns, HINSTANCE hInsPrev, LPSTR lpCmdLine, int iShowCmd ) {
    // Creo y registro la ventana principal.
    ...

    // Creo una etiqueta, una caja de texto y un botón.
    HWND hWndLabel = CreateWindow(
        "STATIC", "Ingrese un nombre:", WS_CHILD | WS_VISIBLE, 10, 10, 150, 20,
        hWnd, NULL, hIns, NULL );
    HWND hWndTextBox = CreateWindow(
        "EDIT", "", WS_CHILD | WS_VISIBLE | WS_BORDER, 170, 10, 100, 20,
        hWnd, (HMENU)ID_TEXTO, hIns, NULL );
    HWND hWndButton = CreateWindow(
        "BUTTON", "OK", WS_CHILD | WS_VISIBLE, 10, 40, 100, 20,
        hWnd, (HMENU)ID_BOTON, hIns, NULL );

    // Muestro la ventana.
    ...

    // Se realiza un bucle donde se procesen los mensajes de la cola de mensajes.
    MSG Mensaje;
    while( GetMessage( &Mensaje, NULL, 0, 0 ) > 0 )
        if( TranslateMessage( &Mensaje ) == FALSE )
            DispatchMessage( &Mensaje );

    return 0;
}
```

A los elementos GUI del API de Windows se les llama controles. Como puede observarse, los controles no son más que ventanas cuyos nombres de clase (STATIC, EDIT y BUTTON) corresponden a clases de ventana preregistradas. Dado que estas ventanas deben dibujarse dentro del área cliente de nuestra ventana principal, tenemos que asignarles la constante de estilo WS_CHILD y el octavo parámetro debe ser el identificador de esta ventana padre. El estilo WS_VISIBLE evita que tengamos que ejecutar ShowWindow para cada una de estas ventanas hijas.

El noveno parámetro de CreateWindow puede, opcionalmente, ser un número identificador que distinga dicha ventana hija de sus ventanas hermanas (hijas de la misma ventana padre). Este parámetro es aprovechado en la función de ventana de nuestra ventana principal. En dicha función se agrega una sentencia CASE para el mensaje WM_COMMAND, el cual es generado por diferentes objetos visibles cuando el usuario interactúa con ellos. En particular, cuando presionamos el botón creado, se agrega a la cola de mensajes del programa, un mensaje WM_COMMAND con los dos bytes menos significativos del parámetro wParam iguales al identificador del botón, ID_BOTON. También utilizamos el identificador de la caja de texto, ID_TEXTO, para poder obtener el texto ingresado llamando a la función GetDlgItemText.

En general, la interacción con los controles estándar del API de Windows, así como otros objetos visuales de una ventana, generan mensajes que son enviados a la ventana padre para su procesamiento. De igual forma, dicho procesamiento suele incluir el envío de nuevos mensajes a los objetos visibles, para obtener más información o para reflejar el cambio de estado del programa (internamente, GetDlgItemText envía un mensaje directamente a la función de ventana del control hijo, de manera que ésta devuelva el texto ingresado), en otras palabras, todo

se realiza enviando y recibiendo mensajes. Esto tiene la ventaja de unificar la forma de trabajo con ventanas a un modelo simple de envío-recepción de mensajes, pero con la desventaja de limitar el procesamiento a una sola ventana (comunmente, solo la ventana padre). Esto tiene sentido bajo el enfoque de que, todos los elementos manejados son ventanas, por tanto es de esperarse que sólo la ventana padre esté interesada en los mensajes de sus hijas. El problema sucede cuando queremos encapsular la funcionalidad de una ventana a sí misma para ciertos trabajos, es decir, ¿cómo hacer para que una caja de texto maneje por sí mismo los mensajes que sólo le competen a él, y envíe a la ventana padre el resto?. Veremos que Java y C# solucionan, de diferente forma, estas carencias del enfoque del API de Windows.

Elementos GUI de Java

En Java, los elementos GUI se denominan componentes. El siguiente código muestra una ventana equivalente en Java al código anterior en API de Windows.

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

class VentanaDePrueba extends JFrame {
    private JTextField txt;

    public VentanaDePrueba(String titulo) {
        super(titulo);

        JLabel lbl = new JLabel("Ingrese un nombre:");
        txt = new JTextField(20);
        JButton btn = new JButton("OK");
        btn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(VentanaDePrueba.this,
                    txt.getText(), "Hola", JOptionPane.INFORMATION_MESSAGE);
            }
        });

        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(lbl);
        cp.add(txt);
        cp.add(btn);
    }
}

public class Componentes {
    public static void main(String args[]) {
        VentanaDePrueba ventana = new VentanaDePrueba(
            "Ventana de prueba de componentes");
        ventana.setSize(400, 300);
        ventana.setVisible(true);
        ventana.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}
```

En el programa anterior, `JTextField`, `JLabel` y `JButton` son componentes que representan una caja de texto, una etiqueta y un botón respectivamente, al igual que las clases de ventana `EDIT`, `STATIC` y `BUTTON` del API de Windows.

Los elementos GUI de Java se denominan **componentes**. Como puede observarse, los componentes en Java son clases que se agregan a una ventana para ser visualizados. Es importante en este punto hacer una distinción entre lo que son internamente estos componentes, contra lo que son los controles del API de Windows. Mientras que los controles

del API de Windows son ventanas, los componentes de Java (a partir de la versión 1.2 del JDK, denominada Java 2) se dividen en dos categorías:

- Los **componentes pesados**. Su comportamiento está supeditado a las capacidades de la plataforma subyacente, en este caso particular, Windows. Estos componentes son JFrame, JDialog y JApplet. Estos componentes crean ventanas de Windows (o utilizan directamente una ya creada) y las administran internamente, ofreciendo al programador una interfaz más amigable. En otras palabras, por ejemplo, cuando realizamos click sobre una ventana creada con un objeto que deriva de JFrame, el evento que se genera es un mensaje WM_LBUTTONDOWN, el cual es sacado de la cola de mensajes y enviado a una función de ventana que llama a un método de nuestro objeto JFrame, el cual se encarga de llamar al método respectivo de MouseListener para todos los objetos registrados con un llamado a addMouseListener. Además de lo anterior, el dibujo de la ventana, el efecto de maximizado y minimizado, la capacidad de redimensionar la ventana y todos los efectos visuales posibles, son gestionados por las funciones del API de Windows, así como las capacidades ofrecidas por la propia clase JFrame, por ejemplo, al llamar al método setVisible se estaría llamando internamente a ShowWindow del API de Windows. Debido a esta dependencia, estos componentes son denominados pesados.
- Los **componentes ligeros**. Su comportamiento está supeditado a las capacidades ofrecidas por un componente pesado, del cual heredan o dentro del cual se dibujan, y no de la plataforma subyacente, en este caso particular, Windows. Si bien los mensajes producidos por la interacción del usuario siguen siendo producidos por el sistema operativo, el manejo de éstos (en forma de eventos), el dibujo de los componentes y de sus efectos visuales relacionados, están codificados completamente en Java. Estos componentes definen además sus propios eventos. Debido a esta independencia, estos componentes son denominados ligeros. Ejemplos de estos componentes son JButton, JLabel y JTextField. Los componentes ligeros se dibujan sobre el área cliente de componentes pesados y simulan el “Look And Feel” correspondiente, es decir, no crean una ventana child. Este tipo de ventanas, child, se verá mas adelante (sección “Tipos de Ventana”)

Por otro lado, la clase JFrame no administra directamente el área cliente de su ventana, sino que delega dicho trabajo a un objeto Contenedor, derivado de la clase Container. Un Contenedor es básicamente un Componente Java con la capacidad adicional de poder mostrar otros Componentes dentro de su área de dibujo. Es por esto que es necesario obtener una referencia a dicho contenedor de la ventana, llamando al método getContentPane, dado que es a dicho contenedor al que deberemos agregarle los componentes que deseamos visualizar.

Los componentes, al igual que una ventana, pueden generar mensajes como respuesta a la interacción del usuario con ellos. En el código anterior, un botón creado con la clase JButton genera el evento Action cuando el usuario, con el ratón o el teclado, presiona dicho botón. Para procesar dicho evento, definimos una clase inner anónima que implementa la interfaz ActionListener, instanciando dicha clase y pasándole la referencia a esta instancia al método addActionListener del botón.

Es interesante notar el hecho de que una clase inner anónima puede ser creada realmente a partir de una clase o de una interfaz, siempre que en éste último caso se implementen todos sus métodos, que para este caso, es uno sólo, actionPerformed. De igual manera, es interesante notar que se ha escogido configurar el objeto observador del evento de cerrado de la ventana desde el método main, no desde el constructor de la ventana. Ambos enfoques son equivalentes.

Dentro del método `actionPerformed` se hace uso del dato miembro `txt` de tipo `JTextField` para poder mostrar el mensaje respectivo mediante el método estático `showMessageDialog` de la clase `JOptionPane`. Note que el primer parámetro de este método debe ser una referencia a la ventana padre de la ventana que se mostrará, y que dicho parámetro se pasa utilizando la expresión `Ventana.this`. Esto se debe a que si pasáramos únicamente `this`, nos estaríamos refiriendo al objeto anónimo que implementa la interfaz `ActionListener`.

Finalmente para este código, note que antes de agregar los componentes al `content pane`, se llama al método `setLayout`. Esto permite determinar la forma en que los componentes son distribuidos dentro del área cliente del contenedor. El manejo del diseño (`layout`) de una ventana se tratará más adelante.

Sumarizando las diferencias entre API de Windows y Java, mientras los objetos visuales comunes llamados controles, preimplementados en dicha librería, son básicamente ventanas y las acciones son manejadas mediante mensajes enviados por estos controles a sus ventanas padres, en Java se trabajan con clases llamadas componentes, las cuales se agregan al contenedor `ContentPane` de la ventana, y sus acciones son manejadas mediante interfaces. Mientras que los controles del API de Windows tienen una posición fija respecto a la esquina superior izquierda del área cliente de su ventana padre y sus dimensiones son fijas, los componentes de Java no tienen una posición ni dimensión fija, y esto es manejado mediante objetos `Layout` que asisten en el diseño de la ventana. Mientras que en API de Windows los mensajes sólo pueden ser recibidos por ventanas y sólo una ventana, generalmente la ventana padre, es la que recibe los mensajes de los controles, en Java cualquier objeto de cualquier clase que implemente la interfaz correspondiente a un evento puede recibir la notificación del mismo.

Elementos GUI de C#

Los elementos GUI en C# se denominan controles. El siguiente código muestra una ventana equivalente en C# al código anterior en API de Windows.

```
using System;
using System.Windows.Forms;
using System.Drawing;

class Ventana : Form {
    private TextBox txt;

    public Ventana() {
        this.Text = "Prueba de Controles";
        this.Size = new Size(300, 300);

        Label lbl = new Label();
        lbl.AutoSize = true;
        lbl.Text = "Ingrese un nombre:";
        lbl.Location = new Point(10, 10);

        txt = new TextBox();
        txt.Size = new Size(100, lbl.Height);
        txt.Location = new Point(10 + lbl.Width + 10, 10);

        Button btn = new Button();
        btn.Text = "OK";
        btn.Size = new Size(100, lbl.Height + 10);
        btn.Location = new Point(10, 10 + lbl.Height + 10);
        btn.Click += new EventHandler(Btn_Click);

        this.Controls.Add(lbl);
        this.Controls.Add(txt);
        this.Controls.Add(btn);
    }

    private void Btn_Click(object sender, EventArgs args) {
```

```
        MessageBox.Show(txt.Text, "Hola");
    }
}

class Principal {
    public static void Main(string[] args) {
        Application.Run(new Ventana());
    }
}
```

A diferencia de Java, dichos controles sí se crean en base a ventanas de Windows y su posición en el área cliente de la ventana padre sí se determina explícitamente. Al igual que Java, existe un objeto que administra la colección de controles dibujados en una ventana: Controls. Al igual que Java, cualquier objeto puede ser notificado de un evento, utilizando un objeto delegado del tipo del evento y agregándolo al evento correspondiente del componente.

Manejo Asistido del Diseño

El diseño o layout de una ventana es la distribución del espacio de su área cliente entre los elementos GUI que contiene y que componen la interfaz que ofrece al usuario. El API de Windows no ofrece herramientas para asistir al programa en el manejo del diseño. Java y C# sí lo ofrecen.

En Java todo contenedor mantiene una referencia a un objeto que implemente la interfaz LayoutManager y que se encarga de determinar la posición y dimensiones de todos los componentes agregados al ContentPane del contenedor. Esto le quita la tarea al programador de especificar por código estos datos por cada componente. Algunos de los tipos de Layout predefinidos son:

- **BorderLayout:** El área del content pane se divide en 5 regiones (norte, sur, este, oeste y centro) colocando un componente agregado en cada región, por lo que sólo se permite mostrar hasta 5 componentes a la vez, independientemente de que se agreguen más.
- **FlowLayout:** Los componentes son colocados en líneas, uno después del otro, como si cada componente fuese una palabra, continuando con la siguiente línea cuando no queda espacio en la línea actual para visualizar completamente dicho componente. Las dimensiones de cada componente son obtenidas de los valores por defecto que cada uno tiene o de las especificadas por el programa, el layout no modifica estas dimensiones.
- **GridLayout:** El área del ContentPane se divide en cuadrícula o grilla, donde cada celda tiene las mismas dimensiones. Los componentes son colocados dentro de estas celdas, modificándoles sus dimensiones para que la ocupen completamente.

El siguiente código muestra el uso de estos layouts en una ventana que utiliza un tipo u otro según un parámetro pasado en su constructor.

```
public Ventana(String Nombre) {
    Container cp = getContentPane();
    if(Nombre.equals("BorderLayout")) {
        cp.setLayout(new BorderLayout());
        cp.add(new JButton("CENTER"), BorderLayout.CENTER);
        cp.add(new JButton("EAST"), BorderLayout.EAST);
        cp.add(new JButton("WEST"), BorderLayout.WEST);
        cp.add(new JButton("NORTH"), BorderLayout.NORTH);
        cp.add(new JButton("SOUTH"), BorderLayout.SOUTH);
    } else if(Nombre.equals("FlowLayout")) {
        cp.setLayout(new FlowLayout(FlowLayout.CENTER));
        for(int i = 0; i < 10; i++)
            cp.add(new JButton("Boton-" + i));
    } else if(Nombre.equals("GridLayout")) {
```

```
        cp.setLayout(new GridLayout(3, 2));
        for(int i = 0; i < 10; i++)
            cp.add(new JButton("Boton-" + i));
    }
    . . .
```

En C# el manejo del diseño se realiza mediante anclas (anchors) y muelles (docks). Todo control posee las siguientes propiedades:

- **Anchor:** Determina a qué borde del contenedor se anclará el control. Por ejemplo, si el control se coloca inicialmente a una distancia de 100 píxeles del borde inferior de su ventana, al redimensionarse el anchor modificará automáticamente la posición del control de forma que conserve dicha distancia.
- **Dock:** Determina a qué borde del contenedor se adosará el control. Por ejemplo, si el control se adosa al borde izquierdo de su ventana, su ancho inicial se conserva pero su alto se modifica de forma que coincida con el alto del área cliente de su ventana. Su posición también se modifica de forma que su esquina superior izquierda coincida con la del área cliente de su ventana.
- **DockPadding:** Se establece en el contenedor, por ejemplo, una clase que hereda de Form. Determina la distancia a la que los componentes, adosados a sus bordes, estarán de los mismos.

El siguiente código muestra el uso de estas propiedades sobre un botón que es agregado a una ventana.

```
Boton = new Button();
Boton.Text = "Boton1";
Boton.Dock = DockStyle.Top;
Controls.Add( Boton );

Boton = new Button();
Boton.Text = "Boton2";
Boton.Location = new System.Drawing.Point(100, 100);
Boton.Size = new System.Drawing.Size(200, 50);
Boton.Anchor = AnchorStyles.Bottom | AnchorStyles.Right;
Controls.Add( Boton );
```

Tipos de Ventana

En un sistema gráfico con ventanas, dichas ventanas pueden estar relacionadas. Estas relaciones determinan los tipos de ventanas que se pueden crear.

En Windows, existen dos tipos de relaciones entre ventanas:

- La **relación de pertenencia**. Cuando dos ventanas tienen esta relación, una ventana (llamada owned) le pertenece a la otra ventana (llamada owner), lo que significa que:
 - ⇒ La ventana owned siempre se dibujará sobre su ventana owner. A la ubicación de una ventana con respecto a otra en un eje imaginario Z que sale de la pantalla del computador, se le conoce como orden-z.
 - ⇒ La ventana owned es minimizada y restaurada cuando su ventana owner es minimizada y restaurada.
 - ⇒ La ventana owned es destruida cuando su ventana owner es destruida.
- La **relación padre-hijo**. Cuando dos ventanas tienen esta relación, una ventana (llamada hija) se dibujará dentro del área cliente de otra (llamada padre).

La relación de pertenencia se establece con el octavo parámetro de la función `CreateWindow`, `hWndParent`. La relación padre-hijo se establece con el tercer parámetro de la función `CreateWindow`, escogiendo como bit-flag `WS_CHILD` o `WS_POPUP`.

Una ventana popup tiene como área de dibujo el escritorio de Windows (Windows Desktop) y puede tener un botón relacionado en la barra de tareas (Windows TaskBar). Como contraparte, una ventana Child tiene como área de dibujo el área cliente de otra ventana, la que puede ser de tipo `Popup` o `Child`, y no puede tener un botón relacionado en la barra de tareas.

Las ventanas popup pueden o no tener una ventana owner. Cuando no la tienen se les llama `OVERLAPPED`. Un ejemplo de una ventana `OVERLAPPED` es la ventana principal de todo programa con ventanas de Windows. Las ventanas child siempre tienen una ventana owner.

En resumen, las ventanas popup pueden ser owner o owned, mientras que las ventanas child son siempre owned.

Una ventana que contiene una o más ventanas child del mismo tipo, cada una con su propia barra de título y botones de sistema, se le conoce como ventana **MDI** (Multiple Document Interface), donde cada ventana hija suele ser utilizada para manipular un documento. Ejemplos de estas ventanas son los programas editores de texto como Word. Las ventanas no-MDI son conocidas como ventanas SDI (Single Document Interface).

A las ventanas cuyos elementos permiten mostrar e ingresar información, éste es, establecer un diálogo con el usuario se les conoce como Cajas De Dialogo. Las cajas de diálogo no son estrictamente un tipo de ventana, su tipo real es `Popup`, más bien su concepto corresponde a “una forma de manejo” de una ventana. Existen dos formas de mostrar una caja de diálogo: Modal y amodalmente. Una caja de diálogo modal “detiene”, por así decirlo, la ejecución del código desde donde se le muestra, una caja de diálogo amodal no. Por ello las cajas de diálogo modales son adecuadas cuando, dentro de un bloque de instrucciones, se requiere pedir al usuario que ingrese alguna información necesaria para seguir con la ejecución del algoritmo implementado por el bloque. Un ejemplo son las ventanas mostradas por los programas al momento de imprimir. En estos casos “no es conveniente” que el usuario del programa pueda interactuar con la ventana principal de forma que modifique los datos que se imprimirán mientras se están imprimiendo, por lo que resulta imprescindible que el procesamiento de los eventos de la ventana principal sea bloqueado. Las cajas de diálogo amodales son adecuadas para mostrar e ingresar información mientras se sigue interactuando con otra ventana, comunmente la ventana principal del programa. Un ejemplo son las barras de herramientas de algunos programas gráficos como CorelDraw, el editor ortográfico de Word, etc.

El API de Windows implementa un conjunto de cajas de diálogo para acciones comunes como seleccionar un color, abrir un archivo, imprimir, etc. A estas cajas de diálogo se le conoce como Cajas de Diálogo Comunes.

En las siguientes secciones se detallará las capacidades del API de Windows manejado desde C/C++, de Java y de la plataforma .NET programada desde C#, para crear los diferentes tipos de ventanas.

Ventanas con API de Windows

Para crear una ventana owner se utiliza el estilo `WS_POPUP` (o algún estilo que lo incluya, como `WS_OVERLAPPED` o `WS_OVERLAPPEDWINDOW`) y se pasa `NULL` como el manejador de su ventana owner, lo que equivale a decir que no tiene ventana owner. El siguiente código crea una ventana owner:

```
hWndPopupOwner = CreateWindow(  
    "ClaseVentanaPopup",  
    "Título de la Ventana Owner",  
    WS_POPUP | WS_CAPTION,  
    100, 100, 200, 200,  
    NULL, // no tiene ventana owner  
    NULL, hIns, NULL  
);
```

Para crear una ventana owned se utiliza el estilo WS_POPUP y se pasa un manejador válido de su ventana owner. El siguiente código crea una ventana owned:

```
hWndPopupOwned = CreateWindow(  
    "ClaseVentanaPopup",  
    "Título de la Ventana Popup Owned",  
    WS_POPUP | WS_CAPTION,  
    100, 100, 200, 200,  
    hWndPopupOwner, // ventana owner  
    NULL, hIns, NULL  
);
```

Para crear una ventana child se utiliza el estilo WS_CHILD y se pasa un manejador válido de su ventana owner. El siguiente código crea una ventana owned:

```
hWndChild = CreateWindow(  
    "ClaseVentanaHija",  
    "Título de la Ventana Hija",  
    WS_CHILD | WS_BORDER,  
    100, 100, 200, 200,  
    hWnd, // debe tener una ventana owner  
    NULL, hIns, NULL  
);
```

La creación y manejo de cajas de diálogo y ventanas MDI con API de Windows va más allá de los alcances del presente documento.

Ventanas en Java

En Java cada nueva herencia de las clases base para la creación de ventanas (JFrame y JDialog) determina una nueva clase de ventana. Un programa puede definir y crear una o más ventanas, de igual o distinto tipo. Sin embargo, al crear una nueva ventana no se establece una relación de parentesco entre ellas, todas se comportan como popups owner.

El siguiente código muestra la creación de una ventana popup en Java desde la ventana principal del programa.

```
class VentanaPopup extends JFrame { ... }  
class VentanaPrincipal extends JFrame {  
    public VentanaPrincipal() {  
        JButton boton = new JButton("Crear Ventana");  
        boton.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent e) {  
                VentanaPopup vp = new VentanaPopup();  
            }  
        }); ...  
    }  
}
```

Java soporta la creación de aplicaciones MDI. La ventana MDI es llamada “backing window” y consiste en una ventana popup con un “Content Pane” del tipo “JDesktopPane”. Las “pseudo-ventanas child” son implementadas con la clase “JInternalFrame”, la que hereda de “JComponent” por lo que, como puede deducirse, no son realmente ventanas. El siguiente código muestra un ejemplo de este uso:

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

```
import javax.swing.*;

class Ventanas1 extends JFrame {
    public Ventanas1() {
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                dispose();
                System.exit(0);
            }
        });

        JDesktopPane desktop = new JDesktopPane();
        setContentPane(desktop);

        JInternalFrame child = new JInternalFrame();
        child.setSize(100, 100);
        child.setTitle("Ventana hija");
        child.setVisible(true);
        child.setResizable(true);
        child.setClosable(true);
        child.setMaximizable(true);
        child.setIconifiable(true);
        desktop.add(child);
    }

    public static void main(String args[]) {
        System.out.println("Starting Ventanas1...");
        Ventanas1 mainFrame = new Ventanas1();
        mainFrame.setSize(400, 400);
        mainFrame.setTitle("Ventanas1");
        mainFrame.setVisible(true);
    }
}
```

Las cajas de diálogo en Java se crean mediante clases que heredan de `JDialog`. El siguiente código muestra el uso de esta clase:

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

class CajaDeDialogo extends JDialog {
    JTextField texto;
    public CajaDeDialogo(JFrame padre, boolean EsModal) {
        super(padre, EsModal);
        setSize(300,100);

        texto = new JTextField(20);
        JButton boton = new JButton("OK");
        boton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                setVisible(false);
            }
        });

        Container cp = getContentPane();
        cp.setLayout(new GridLayout(3,1));
        cp.add(new JLabel("Ingrese un texto"));
        cp.add(texto);
        cp.add(boton);
    }
    public String ObtenerResultado() {
        return texto.getText();
    }
}

class Ventanas2 extends JFrame {
    JLabel etiqueta;

    public Ventanas2() {
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                dispose();
            }
        });
    }
}
```



```
        System.exit(0);
    }
});

etiqueta = new JLabel("Resultado = ...");
JButton boton1 = new JButton("Mostrar como modal");
boton1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        CajaDeDialogo dialogo = new CajaDeDialogo(Ventanas2.this,
true);
        dialogo.setVisible(true);
        etiqueta.setText("Resultado = " +
dialogo.ObtenerResultado());
    }
});
JButton boton2 = new JButton("Mostrar como amodal");
boton2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        CajaDeDialogo dialogo = new CajaDeDialogo(Ventanas2.this,
false);
        dialogo.setVisible(true);
        etiqueta.setText("Resultado = " +
dialogo.ObtenerResultado());
    }
});

Container cp = getContentPane();
cp.setLayout(new GridLayout(3,1));
cp.add(boton1);
cp.add(boton2);
cp.add(etiqueta);
}

public static void main(String args[]) {
    System.out.println("Starting Ventanas2...");
    Ventanas2 mainFrame = new Ventanas2();
    mainFrame.setSize(300, 200);
    mainFrame.setTitle("Ventanas2");
    mainFrame.setVisible(true);
}
}
```

El programa anterior muestra la diferencia entre utilizar una caja de diálogo modal y una amodal. También muestra una forma de pasar datos desde la caja de diálogo y la ventana que la crea.

Las cajas de diálogo de Java también son llamadas “ventanas secundarias”, mientras que las pseudo-ventanas hijas creadas con `JInternalFrame` son llamadas “ventanas primarias”.

Adicionalmente Java provee la clase `JOptionPane`, la que permite crear cajas de diálogo con funcionalidad común, como por ejemplo, cajas de diálogo con un texto como mensaje y botones YES, NO y CANCEL.

Ventanas en C#

Al igual que en Java, cada nueva herencia de las clases base para la creación de ventanas, `Form`, determina una nueva clase de ventana. Un programa puede definir y crear una o más ventanas, de igual o distinto tipo. A diferencia de Java, se pueden crear ventanas popups owner y owned.

El siguiente código muestra la creación de dos ventanas popup, una owner y la otra owned.

```
class VentanaPopup : Form {
    public VentanaPopup( Form OwnerForm ) {
        ...
        this.Owner = OwnerForm;
        this.Visible = true;
    }
}
class VentanaPrincipal : Form {
    public VentanaPrincipal() {
```

```
...
VentanaPopup vp1 = new VentanaPopup( null );
VentanaPopup vp2 = new VentanaPopup( this );
}
...
}
```

C# maneja ventanas child sólo como ventanas hijas de una ventanas MDI. La ventana MDI consiste en una ventana con la propiedad `IsMdiContainer` puesta en “true”. Para que una ventana sea child de otra, se establece su propiedad `MdiParent` con la referencia de una ventana MDI. El siguiente código muestra un ejemplo de este uso:

```
using System;
using System.Windows.Forms;

class VentanaHija : Form {
    public VentanaHija(Form padre) {
        this.SuspendLayout();
        this.Text = "Ventana Hija";
        this.Location = new System.Drawing.Point(10,10);
        this.Size = new System.Drawing.Size(100, 100);
        this.MdiParent = padre;
        this.Visible = true;
        this.ResumeLayout(false);
    }
}

class VentanaPrincipal : Form
{
    public VentanaPrincipal()
    {
        InitializeComponent();
        this.IsMdiContainer = true;
        VentanaHija hija = new VentanaHija(this);
    }

    void InitializeComponent()
    {
        this.SuspendLayout();
        this.Name = "MainForm";
        this.Text = "Esta es la Ventana Principal";
        this.Size = new System.Drawing.Size(300, 300);
        this.ResumeLayout(false);
    }

    public static void Main(string[] args)
    {
        Application.Run(new VentanaPrincipal());
    }
}
```

Las cajas de diálogo en C# son ventanas con la propiedad `FormBorderStyle` puesta a `FixedDialog`. El siguiente código muestra el uso de una caja de diálogo.

```
using System;
using System.Windows.Forms;

class CajaDeDialogo : Form
{
    private TextBox texto;
    public CajaDeDialogo() {
        this.Text = "Caja de Dialogo";
        this.FormBorderStyle = FormBorderStyle.FixedDialog;

        Label etiqueta = new Label();
        etiqueta.Text = "Ingrese un texto";
        etiqueta.Location = new System.Drawing.Point(24, 16);

        texto = new TextBox();
        texto.Size = new System.Drawing.Size(128, 32);
        texto.Location = new System.Drawing.Point(24, 64);
    }
}
```

```
        Button boton = new Button();
        boton.Text = "OK";
        boton.Size = new System.Drawing.Size(128, 32);
        boton.Location = new System.Drawing.Point(24, 112);
        boton.Click += new System.EventHandler(this.buttonClick);

        Controls.Add(etiqueta);
        Controls.Add(texto);
        Controls.Add(boton);
    }
    void buttonClick(object sender, System.EventArgs e)
    {
        //Visible = false;
        Close();
    }
    public string ObtenerResultado() {
        return texto.Text;
    }
}

class VentanaPrincipal : Form
{
    private System.Windows.Forms.Label label;
    private System.Windows.Forms.Button button2;
    private System.Windows.Forms.Button button;
    public VentanaPrincipal()
    {
        InitializeComponent();
    }

    void buttonClick(object sender, System.EventArgs e)
    {
        // caso de una caja de dialogo modal
        CajaDeDialogo dialogo = new CajaDeDialogo();
        dialogo.ShowDialog(this);
        // dado que la ejecución de la instrucción anterior se detiene hasta que la
        // caja de diálogo se cierre, es posible recuperar el valor ingresado
        label.Text = "Resultado = " + dialogo.ObtenerResultado();
    }

    void button2Click(object sender, System.EventArgs e)
    {
        // caso de una caja de dialogo amodal
        CajaDeDialogo dialogo = new CajaDeDialogo();
        dialogo.Show();
        // dado que la ejecución de la instrucción anterior "NO" se detiene hasta que
        // la caja de diálogo se cierre, el valor recuperado sera el valor que
        // inicialmente tiene la caja de texto de dicha caja de dialogo al crearse,
        // esto es, una cadena vacia
        label.Text = "Resultado = " + dialogo.ObtenerResultado();
    }

    void InitializeComponent() {
        this.button = new System.Windows.Forms.Button();
        this.button2 = new System.Windows.Forms.Button();
        this.label = new System.Windows.Forms.Label();
        this.SuspendLayout();
        //
        // Primer botón
        //
        this.button.Location = new System.Drawing.Point(24, 16);
        this.button.Name = "button";
        this.button.Size = new System.Drawing.Size(128, 32);
        this.button.TabIndex = 0;
        this.button.Text = "Mostrar Como Modal";
        this.button.Click += new System.EventHandler(this.buttonClick);
        //
        // Segundo botón
        //
        this.button2.Location = new System.Drawing.Point(24, 64);
        this.button2.Name = "button2";
        this.button2.Size = new System.Drawing.Size(128, 32);
    }
}
```

```
this.button2.TabIndex = 1;
this.button2.Text = "Mostrar Como Amodal";
this.button2.Click += new System.EventHandler(this.button2Click);
//
// Etiqueta
//
this.label.Location = new System.Drawing.Point(24, 112);
this.label.Name = "label";
this.label.Size = new System.Drawing.Size(128, 24);
this.label.TabIndex = 2;
this.label.Text = "Resultado = ...";
//
// Agrego los controles
//
this.ClientSize = new System.Drawing.Size(248, 165);
this.Controls.AddRange(new System.Windows.Forms.Control[] {
    this.label,
    this.button2,
    this.button});
this.Text = "Prueba con Cajas de Diálogo";
this.ResumeLayout(false);
}

public static void Main(string[] args)
{
    Application.Run(new VentanaPrincipal());
}
}
```

El programa anterior muestra la diferencia entre utilizar una caja de diálogo modal, con `ShowDialog`, y una amodal, con `Show`. También muestra una forma de pasar datos desde la caja de diálogo y la ventana que la crea.

Dado que es común validar la forma en que fue respondida una caja de diálogo modal, se provee la propiedad `DialogResult`, cuyo tipo es el enumerado `DialogResult` con los siguientes valores: `Abort`, `Cancel`, `Ignore`, `No`, `None`, `OK`, `Retry`, `Yes`. Esta propiedad se establece automáticamente en algunos casos (cuando se cierra la ventana, se establece a `Cancel`) o manualmente desde eventos programados.

Notas sobre Localización de Archivos

Los programas en ejecución (llamados procesos) tienen siempre un directorio de trabajo. Este directorio sirve para poder ubicar, de forma relativa, otros archivos (por ejemplo, para abrir dichos archivos). De esta forma un proceso no requiere utilizar siempre el directorio absoluto para acceder a archivos que se encuentran en su mismo directorio de trabajo o en algún otro directorio cercano a éste, como sus subdirectorios.

Por defecto, un proceso obtiene su directorio de trabajo heredándolo de su proceso padre (el que lo arrancó), a menos que este último indique explícitamente otro directorio. Por ejemplo, la siguiente línea de comando corresponde a una ventana de comandos (o shell) desde donde se arranca un programa `Abc`:

```
c:\prueba> c:\temp\Abc.exe
```

Note que el programa se encuentra en el directorio “`c:\temp`” mientras que el directorio de trabajo del shell es “`c:\prueba`”. Luego, el nuevo proceso `Abc` creado tendrá como directorio de trabajo “`c:\prueba`”, pues lo hereda del shell. Si se desea arrancar un programa con un directorio de trabajo distinto al del shell, se puede utilizar el comando `start`:

```
c:\prueba> start /Dd:\otroDirectorio c:\temp\Abc.exe
```

Es fácil ver cual es el directorio de trabajo actual del shell (en Windows lo indica el mismo prompt, en Linux se puede consultar con un comando, por ejemplo `pwd`) y cambiarlo (utilizando un comando como `cd`). De igual forma, utilizando las llamadas a las funciones adecuadas del API del sistema operativo, cualquier proceso puede modificar su directorio de trabajo durante su ejecución.

Para programas diferentes a los shells, desde donde también es posible arrancar otros programas, el directorio de trabajo actual puede no ser claro, por lo que heredarlo puede no ser lo que el usuario espera. Por ejemplo, al arrancar un programa desde el explorador de Windows haciendo un doble-click sobre el nombre del archivo ejecutable, el usuario espera que se inicie dicho programa teniendo como directorio de trabajo inicial el mismo directorio donde se encuentra el archivo ejecutable, independientemente de cual sea actualmente el directorio de trabajo del explorador de Windows. Este comportamiento puede modificarse creando accesos directos a dichos programas ejecutables y configurándolos para especificar un directorio distinto como directorio de trabajo inicial.

No todos los archivos a los que un proceso requiere acceder se ubican utilizando el directorio de trabajo como referencia. Por ejemplo, en el caso de las librerías, se suelen utilizar variables de entorno para definir conjuntos de directorios de búsqueda. Por ejemplo, para las DLL nativas de Windows, se utiliza la variable de entorno `PATH`, mientras el compilador e intérprete de Java utilizan la variable de entorno `CLASSPATH`.

Excepciones

El objetivo de este capítulo es dar una base teórica y práctica al lector sobre la creación y manejo de excepciones, base que será utilizada en los temas siguientes.

¿Qué son las excepciones?

Una excepción es un error excepcional, infrecuente, raro, que ocurre en un porcentaje pequeño de veces en la ejecución de un programa bajo condiciones normales esperadas. Es importante distinguir aquí lo ambiguo de esta definición: ¿A partir de qué porcentaje de frecuencia se le considera a un error infrecuente?, ¿qué condiciones se consideran normales? A lo largo de este capítulo se mostrarán ejemplos comunes de aplicación de excepciones así como algunos criterios de decisión que pueden ayudar a decidir cuándo tratar a un error como una excepción y cuándo no.

Algunos ejemplos típicos de excepciones son:

- Acceso a un elemento de un arreglo fuera de los límites de éste, o en general, a una dirección inválida de memoria.
- Una división por cero.
- Una llamada a un método con valores errados en sus parámetros. Ejemplos: Utilizar un descriptor de archivo aun no inicializado, o ya cerrado, para leer o escribir en un archivo; realizar una conversión de una texto a un número, cuando el texto no contiene una representación válida de un número; pasar una referencia nula de un objeto a una función que espera recibir una referencia válida, etc.
- Una falla en la reserva de memoria (la sentencia `new` falla).

Los errores tratados como excepciones suelen ser también aquellos para los que, dentro del ámbito del programa donde ocurre dicho error, no es posible darle una solución satisfactoria. Note que aquí también hay ambigüedad: ¿Qué es una solución satisfactoria?

Si dicho ámbito abarca a todo el programa, éste por consiguiente no puede continuar ejecutándose de la manera esperada, por lo que debería finalizar. Si en un ámbito que engloba al ámbito de la excepción, se cuenta con los elementos necesarios para darle una solución satisfactoria, el manejo de dicha excepción en el ámbito englobante podría permitir estabilizar el programa, de forma que vuelva a un estado de ejecución normal. Si ningún ámbito englobante, en todo el programa, puede solucionar la excepción, el programa podría notificar al usuario de lo ocurrido y finalizar ordenadamente, liberando los recursos que fueron reservados. En resumen, el tratamiento de excepciones en un programa permite:

- Que el programa se recupere de la excepción y siga ejecutándose normalmente.
- Que el programa notifique la excepción y finalice de manera controlada.

Ahora bien, dado que las excepciones son básicamente errores infrecuentes, podrían tratarse con las mismas técnicas tradicionales que se utilizan para los errores frecuentes. El siguiente pseudocódigo muestra una técnica típica de tratamiento de errores con estructuras de control de flujo.

```
Ejecutar una acción
Si ocurrió un error
    Reportar el error y finalizar
Ejecutar una acción
Si ocurrió un error
    Reportar el error y finalizar
...
```

Bajo esta técnica se tiene las siguientes ventajas:

- El tratamiento de un error se realiza en la vecindad donde ocurre.
- Es fácil reconocer el código que maneja cada error en el programa, y por tanto, entenderlo. Como contraparte, es fácil reconocer la fuente de un error.
- Es fácil realizar un seguimiento a la ejecución del programa.

Las desventajas son:

- Es fácil que el código del programa termine minado con código de manejo de error, lo que hace difícil distinguir la tarea limpia del programa, esto es, la tarea que se intenta realizar independientemente de los errores que puedan ocurrir o asumiendo que no ocurren. Un código así es difícil de mantener.
- Si un error es infrecuente pero potencialmente puede ocurrir en distintas partes del programa, el programa final contendrá mucho código repetitivo de manejo de dichos errores.
- Si un error es infrecuente, el programador podría pasarlo por alto inadvertidamente o bien tendría la tendencia a dejar su tratamiento para después, siendo dicho error olvidado o bien tratado sin un esfuerzo mucho mayor del que hubiese sido necesario en un inicio. Este tipo de errores son un motivo común de la falta de tolerancia a fallos de los programas.
- Si un error es infrecuente pero potencialmente puede ocurrir en distintas partes del programa, todas las verificaciones correspondientes a dicho error se realizarán, aún cuando éste no ocurra, lo que le resta eficiencia (mayor tiempo de CPU y recursos) al programa.

Como puede apreciarse, el tratamiento tradicional de errores es adecuado cuando éstos son frecuentes, pero no cuando son infrecuentes, como las excepciones.

Las excepciones pueden ser generadas por errores detectados por el hardware (como una división entera entre cero) y capturados por el sistema operativo, producidos por el propio sistema operativo debido a un error de lógica interno (detección de un nivel de memoria disponible debajo de un límite de seguridad), producidos por alguna librería utilizada por nuestro programa o por nuestro mismo programa.

Los lenguajes de programación que distinguen el concepto de excepción, como C++, Java y C#, utilizan una técnica muy diferente al tratamiento tradicional de errores. Esta técnica se basa

en la idea de separar el código principal del programa (el código propio de la tarea que el programa desea realizar, un código limpio de verificación de excepciones), del código de manejo de las excepciones. Como puede entenderse, este código limpio seguirá conteniendo el código de manejo de los errores frecuentes, bajo las técnicas tradicionales.

En las siguientes secciones veremos cómo se implementa el tratamiento de excepciones en C++, Java y C#, así como las ventajas y desventajas de su uso.

Implementación

Tanto Java como C# se basan en el modelo de C++ para el tratamiento de excepciones, tema que revisaremos a continuación para después tratar las diferencias que existen entre los tres lenguajes.

C++

Para manejar excepciones que pueden ocurrir dentro de un bloque de código, es necesario “delimitar” dicho bloque. Para esto, se utiliza la palabra reservada try. La sintaxis a utilizar es:

```
try {  
    // Aquí va el código del que se desea controlar las excepciones  
    // que produzca.  
}
```

Dentro del bloque try se colocará el código del que se espera monitorear las excepciones que produzca. Cuando se produce una excepción, se ejecutará un determinado bloque de código llamado manejador de excepción. Estos bloques de código deben de ir inmediatamente después del bloque try e igualmente deben ser “delimitados” para cada tipo en particular de excepción. Para esto, se utiliza la palabra reservada catch. La sintaxis a utilizar será:

```
catch( TipoDeExcepcion e ) {  
    // Aquí va el código que se ejecutara en caso que se produzca  
    // una excepción del tipo "TipoDeExcepcion".  
}
```

Un bloque try puede ir seguido por tantos bloques catch como tipos de excepciones se desee manejar. El tipo de variable TipoDeExcepcion determina el tipo de excepción que manejara un bloque catch.

Dentro del bloque try, una excepción puede ser producida por:

- Una sentencia throw que explícitamente dispare la excepción.
- Una llamada a una función que dispare la excepción. Dicha función podría disparar la excepción explícitamente o llamar a otra función (y ésta a otra y así sucesivamente) la cual sea quien realmente dispare la excepción.
- La creación de un objeto, debido a un error dentro del constructor utilizado.
- Un error del sistema operativo durante la ejecución de cualquier sentencia dentro del bloque try.
- Una interrupción capturada por el sistema operativo y notificada al programa en ejecución a manera de una excepción.

El siguiente código muestra un ejemplo simple de un código en C++ que produce y maneja una excepción.

```
#include <stdio.h>
```

```
class Fecha {
    int dia, mes, anho;
public:
    Fecha(char* pszFecha) {
        if(sscanf(pszFecha, "%d-%d-%d", &dia, &mes, &anho) != 3)
            throw -1;
        if(anho < 0 || dia < 1 || 31 < dia || mes < 1 || 12 < mes)
            throw -2;
    }
    void Imprimir() {
        printf("[%d/%d/%d]", dia, mes, anho);
    }
};

void main() {
    printf("Inicio\n");
    try {
        char szFecha[20];
        puts("Ingrese una fecha :");
        scanf("%s", szFecha);
        Fecha fecha(szFecha);
        puts("La fecha ingresada es :");
        fecha.Imprimir();
    }
    catch(int ex) {
        printf("Excepción: código = %d\n", ex);
    }
    printf("Fin\n");
}
```

En el código anterior, si el constructor de la clase Fecha encuentra que existe un error en la información pasada como parámetro, no tiene forma de remediarlo sin tener que agregar código duro al programa (por ejemplo, una dato miembro de Fecha que funcione como bandera y que indique si el constructor falló o no, de forma que cualquier llamada a métodos de la clase deban verificar dicho valor antes de realizar su tarea).

Note que en el constructor de la clase Fecha se utiliza la llamada a una sentencia throw para producir un error tipo excepción. A esto se le conoce como disparar una excepción, y al lugar donde ocurre, punto de excepción. La sentencia throw es seguida de una expresión cuyo tipo de dato al que se evalúa determina el tipo de la excepción. En el ejemplo, la excepción es de tipo int. Igualmente, el tipo de dato del argumento del bloque catch determina el tipo de este bloque.

El flujo de ejecución del programa es:

- Si no ocurre ninguna excepción, se ejecuta todo el bloque try y luego se salta la ejecución a la siguiente instrucción debajo del último bloque catch.
- Si ocurre una excepción, la ejecución se detiene en el punto de excepción. Si alguno de los bloques catch que siguen al bloque try coincide con el tipo de la excepción generada, entonces se ejecuta dicho bloque (se dice que dicho bloque catch ha capturado la excepción), luego de lo cual, se salta la ejecución a la siguiente instrucción debajo del último bloque catch. Si ninguno de los bloques catch tiene un tipo adecuado al tipo de la excepción, la ejecución salta fuera de la función o método donde ocurrió, en este caso, main.

De lo anterior se puede deducir que:

- Cuando ocurre una excepción, el código que sigue desde el punto de excepción hasta el fin del bloque try, nunca se ejecuta. A este modelo de manejo de excepciones se le llama termination model.

- No todas las excepciones son necesariamente capturadas donde se generan o siquiera por el mismo programa. Más adelante veremos estos casos.

El código anterior genera excepciones de tipo `int`, sin embargo, dado que es posible manejar, mediante el parámetro del bloque `catch`, la información sobre la excepción, lo más usual es utilizar clases, en lugar de datos primitivos, como tipos de excepción. El siguiente código modifica el ejemplo anterior para utilizar clases en lugar de datos primitivos.

```
#include <stdio.h>

class ErrorEnFormato {};
class ErrorFechaInvalida {};
class Fecha {
    int dia, mes, anho;
public:
    Fecha(char* pszFecha) {
        if(sscanf(pszFecha, "%d-%d-%d", &dia, &mes, &anho) != 3) {
            ErrorEnFormato ex;
            throw ex;
        }
        if(anho < 0 || dia < 1 || 31 < dia || mes < 1 || 12 < mes)
            throw ErrorFechaInvalida();
    }
    void Imprimir() {
        printf("[%d/%d/%d]", dia, mes, anho);
    }
};

void main() {
    try {
        char szFecha[20];
        puts("Ingrese una fecha :");
        scanf("%s", szFecha);
        Fecha fecha(szFecha);
        puts("La fecha ingresada es :");
        fecha.Imprimir();
    }
    catch(ErrorEnFormato ex) {
        printf("Error: el formato es incorrecto\n");
    }
    catch(ErrorFechaInvalida ex) {
        printf("Error: la fecha es inválida\n");
    }
}
```

El utilizar clases en lugar de datos primitivos permite la posibilidad de guardar más información acerca de la excepción ocurrida (como en qué archivo fuente ocurrió y dentro de éste, en qué línea) y pasar esta información al manejador de excepción correspondiente. El siguiente código modifica el ejemplo anterior para utilizar información adicional dentro de las clases de excepciones.

```
#include <stdio.h>

class ErrorEnFormato {
    char* pszLugar;
public:
    ErrorEnFormato(char* psz) : pszLugar(psz) {}
    char* Lugar() { return pszLugar; }
};

class ErrorFechaInvalida {
    char* pszLugar;
    char* pszFecha;
public:
    ErrorFechaInvalida(char* pszL, char* pszF) : pszLugar(pszL), pszFecha(pszF) {}
    char* Lugar() { return pszLugar; }
    char* LaFecha() { return pszFecha; }
};
```

```
class Fecha {
    int dia, mes, anho;
public:
    Fecha(char* pszFecha) {
        if(sscanf(pszFecha, "%d-%d-%d", &dia, &mes, &anho) != 3)
            throw ErrorEnFormato("Fecha::Fecha");
        if(anho < 0 || dia < 1 || 31 < dia || mes < 1 || 12 < mes)
            throw ErrorFechaInvalida("Fecha::Fecha", pszFecha);
    }
    void Imprimir() {
        printf("[%d/%d/%d]", dia, mes, anho);
    }
};

void main() {
    try {
        char szFecha[20];
        puts("Ingrese una fecha :");
        scanf("%s", szFecha);
        Fecha fecha(szFecha);
        puts("La fecha ingresada es :");
        fecha.Imprimir();
    }
    catch(ErrorEnFormato ex) {
        printf("Error: el formato es incorrecto en %s\n", ex.Lugar());
    }
    catch(ErrorFechaInvalida ex) {
        printf("Error: el formato de la fecha [%s] es incorrecto en %s\n",
            ex.LaFecha(), ex.Lugar());
    }
}
```

Note que en el código anterior, existe información común entre ambos tipos de excepciones. Este aspecto es explotado utilizando el polimorfismo en la implementación en C++. El siguiente código modifica el ejemplo anterior para utilizar una clase base para ambas excepciones.

```
#include <stdio.h>

class Excepcion {
    char* pszLugar;
public:
    Excepcion(char* psz) : pszLugar(psz) {}
    char* Lugar() { return pszLugar; }
};

class ErrorEnFormato : Excepcion {
public:
    ErrorEnFormato(char* psz) : Excepcion(psz) {}
};

class ErrorFechaInvalida : Excepcion {
    char* pszFecha;
public:
    ErrorFechaInvalida(char* pszL, char* pszF)
        : Excepcion(pszL), pszFecha(pszF) {}
    char* LaFecha() { return pszFecha; }
};

class Fecha {
    int dia, mes, anho;
public:
    Fecha(char* pszFecha) {
        if(pszFecha == 0)
            throw Excepcion("Fecha::Fecha");
        if(sscanf(pszFecha, "%d-%d-%d", &dia, &mes, &anho) != 3)
            throw ErrorEnFormato("Fecha::Fecha");
        if(anho < 0 || dia < 1 || 31 < dia || mes < 1 || 12 < mes)
            throw ErrorFechaInvalida("Fecha::Fecha", pszFecha);
    }
    void Imprimir() {
        printf("[%d/%d/%d]", dia, mes, anho);
    }
};
```

```
    }  
};  
  
void main() {  
    try {  
        char szFecha[20];  
        puts("Ingrese una fecha :");  
        scanf("%s", szFecha);  
        Fecha fecha(szFecha);  
        puts("La fecha ingresada es :");  
        fecha.Imprimir();  
    }  
    catch(ErrorEnFormato ex) {  
        printf("Error: el formato es incorrecto en %s\n", ex.Lugar());  
    }  
    catch(ErrorFechaInvalida ex) {  
        printf("Error: el formato de la fecha [%s] es incorrecto en %s\n",  
            ex.LaFecha(), ex.Lugar());  
    }  
    catch(Excepcion ex) {  
        printf("Error: en %s\n", ex.Lugar());  
    }  
}
```

Note, en el código anterior, el uso de una clase base para las excepciones anteriormente definidas. En el constructor se realiza la verificación de un nuevo tipo de error, que la cadena pasada sea un puntero nulo, para lo cual se usa la clase base `Exception`, dado que las otras no son adecuadas. Para esta clase base se le agrega un bloque `catch`.

Para mostrar porqué se dice que el comportamiento es polimórfico, modifique este código de forma que el bloque `catch Excepcion` esté antes del bloque `catch ErrorFechaInvalida`. Corra el programa y notará que las excepciones del tipo `ErrorFechaInvalida` son ahora capturadas por el bloque `catch Excepcion`. Si el bloque `catch Excepcion` lo colocara antes del bloque `catch ErrorEnFormato`, todas las excepciones serían capturadas por el bloque `catch Excepcion`. La regla de selección del bloque `catch` que capturará una excepción en C++ es:

“Dada una excepción ocurrida dentro de un bloque `try`, se ejecutará el primer bloque `catch`, según el orden de declaración de arriba hacia abajo, cuyo tipo coincida o sea una clase base del tipo de la excepción.”

En el código anterior, pudo utilizarse únicamente el bloque `catch Excepcion` para capturar todas las excepciones. Sin embargo, como puede entenderse, hay información extra que cada tipo de excepción podría manejar y ser útil para el adecuado manejo de ésta.

C++ permite definir además un bloque `catch` que capture todas las excepciones, sin importar su tipo, con el siguiente formato:

```
catch(...) {  
    // Aquí va el manejador de la excepción  
}
```

Este bloque `catch` debe ser el último de todos, sino se produce un error de compilación.

Aunque en todos los ejemplos anteriores, las excepciones generadas eran capturadas y todas se manejaban en un mismo bloque `try`, no siempre es así. El siguiente código muestra los diferentes casos que pueden ocurrir al generarse una excepción.

```
#include <iostream.h>
```

```
class Excepcion {};  
  
void NoGeneraExcepcion() {  
    cout << "Inicio de 'NoGeneraExcepcion'" << endl;  
    try {  
        cout << "Dentro del try de 'NoGeneraExcepcion'" << endl;  
    }  
    catch(Excepcion ex) {  
        cout << "Dentro del catch de 'NoGeneraExcepcion'" << endl;  
    }  
    cout << "Fin de 'NoGeneraExcepcion'" << endl;  
}  
  
void GeneraCapturaExcepcion() {  
    cout << "Inicio de 'GeneraCapturaExcepcion'" << endl;  
    try {  
        cout << "Dentro del try de 'GeneraCapturaExcepcion'" << endl;  
        throw Excepcion();  
        cout << "Este saludo nunca se muestra" << endl;  
    }  
    catch(Excepcion ex) {  
        cout << "Dentro del catch de 'GeneraCapturaExcepcion'" << endl;  
    }  
    cout << "Fin de 'GeneraCapturaExcepcion'" << endl;  
}  
  
void GeneraNoCapturaExcepcion() {  
    cout << "Inicio de 'GeneraNoCapturaExcepcion'" << endl;  
    try {  
        cout << "Dentro del try de 'GeneraNoCapturaExcepcion'" << endl;  
        throw Excepcion();  
        cout << "Este saludo nunca se muestra" << endl;  
    }  
    catch(int ex) {  
        cout << "Dentro del catch de 'GeneraNoCapturaExcepcion'" << endl;  
    }  
    cout << "Fin de 'GeneraNoCapturaExcepcion'" << endl;  
}  
  
void GeneraCapturaRedisparaExcepcion() {  
    cout << "Inicio de 'GeneraCapturaRedisparaExcepcion'" << endl;  
    try {  
        cout << "Dentro del try de 'GeneraCapturaRedisparaExcepcion'" << endl;  
        throw Excepcion();  
        cout << "Este saludo nunca se muestra" << endl;  
    }  
    catch(Excepcion ex) {  
        cout << "Dentro del catch de 'GeneraCapturaRedisparaExcepcion'" << endl;  
        throw ex;  
    }  
    cout << "Fin de 'GeneraCapturaRedisparaExcepcion'" << endl;  
}  
  
void main() {  
    cout << "Llamando a 'NoGeneraExcepcion'" << endl;  
    NoGeneraExcepcion();  
  
    cout << "Llamando a 'GeneraCapturaExcepcion'" << endl;  
    GeneraCapturaExcepcion();  
  
    try {  
        cout << "Llamando a 'GeneraNoCapturaExcepcion'" << endl;  
        GeneraNoCapturaExcepcion();  
    }  
    catch(Excepcion ex) {  
        cout << "Excepcion capturada desde 'GeneraNoCapturaExcepcion' en"  
            << " 'main'" << endl;  
    }  
  
    try {  
        cout << "Llamando a 'GeneraCapturaRedisparaExcepcion'" << endl;  
        GeneraCapturaRedisparaExcepcion();  
    }  
}
```

```
catch(Excepcion ex) {  
    cout << "Excepcion capturada desde 'GeneraCapturaRedisparaExcepcion'"  
        <<" en 'main'" << endl;  
}  
throw ex;  
cout << "Fin del programa";  
}
```

Al ejecutarse se mostrará una salida en una ventana de comandos, como la mostrada en la Figura 7 - 1. Inmediatamente después se muestra una ventana con el mensaje de la Figura 7 - 2.

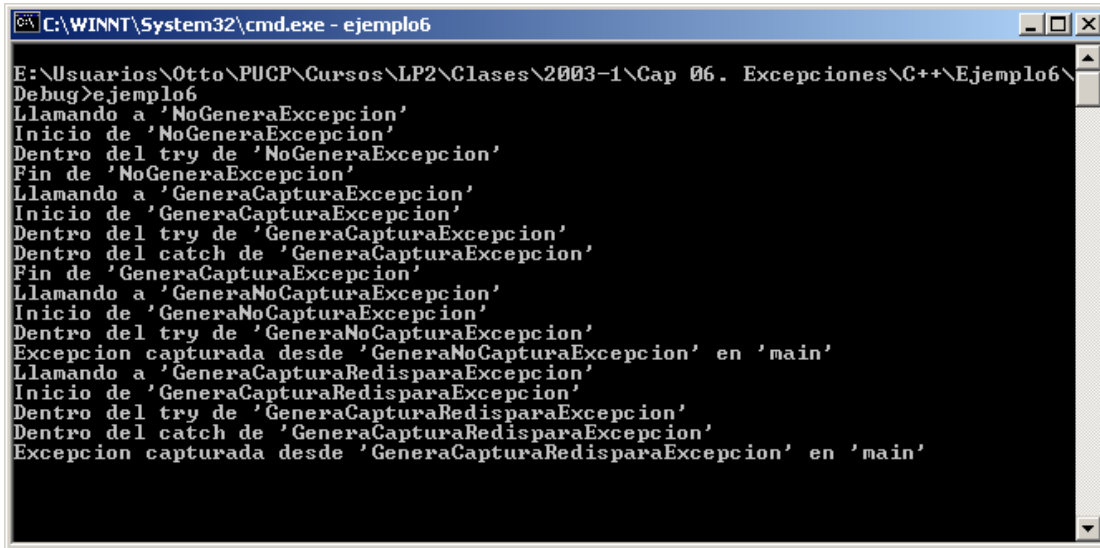


Figura 7 - 1 Ejecución de ejemplo de excepciones en C++

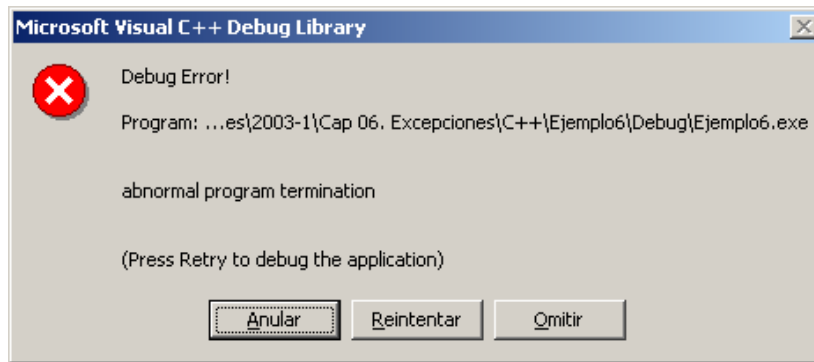


Figura 7 - 2 Mensaje en la ejecución de ejemplo de excepciones en C++

Como puede entenderse por la ejecución del programa, las excepciones que suceden en la llamada a una función son disparadas fuera de éste cuando la función no la captura. Luego, desde donde se llamó a la función que generó la excepción (la llamada a `GeneraNoCapturaExcepcion` desde `main`) se considera que dicha función generó una excepción, por lo que es factible capturarla. En el caso de la última excepción, al final de `main`, ésta no es capturada por nadie por lo que sale fuera del método `main`, siendo capturada por el propio sistema operativo que se encarga de mostrar la ventana mostrada en la figura 7.2 o una similar dependiendo la forma en que se compiló el programa y las capacidades de manejo de excepciones del sistema operativo.

Otro punto importante relacionado al código anterior, es que dentro de un manejador de excepción, un bloque catch, también es posible que ocurra una excepción, que es el caso de la función `GeneraCapturaRedisparaExcepcion`. En este caso, la excepción se dispara directamente fuera de la función donde está dicho bloque catch, aún cuando pudiera existir algún bloque catch debajo del anterior, cuyo tipo sea adecuado al tipo de la excepción generada.

Existe un punto importante no tratado en los códigos de ejemplo anteriores, la pérdida de recursos debido a una excepción. Es importante entender que en los códigos anteriores ésto no ocurre, debido a que los objetos creados en el bloque try eran reservados en la memoria de pila, por lo que al salir la ejecución del bloque try (sea o no por haberse producido una excepción) esta memoria es liberada automáticamente. Para mostrar un caso donde sí se pierden recursos, revisemos el siguiente código.

```
class Excepcion {};  
  
void main() {  
    try {  
        char* pCadena = new char[10];  
        throw Excepcion();  
        delete [] pCadena;  
    }  
    catch(Excepcion ex) {  
    }  
}
```

En el código anterior, debido a que ocurre una excepción antes de poder liberar el recurso reservado, esta liberación nunca podrá realizarse. Tampoco es posible liberar el recurso en el bloque catch debido a que la variable `pObj` es local al bloque try, por lo que para hacer esto posible debería definirse dicha variable como local a main. De esta forma, la liberación de memoria se debería hacer al final de todos los bloques catch del bloque try. Sin embargo, si la excepción producida no es capturada por ninguno de los bloques catch, entonces no habrá manera de liberar el recurso, a menos que la variable `pObj` se declare como global de forma que pueda ser liberado por algún otro manejador de excepción.

Como puede verse, C++ no ofrece una solución estándar a este tipo de pérdida de recursos. Más adelante veremos cómo es que Java y C# tratan este aspecto.

Por último, es importante señalar que el modelo de manejo de excepciones en C++ sólo permite manejar como excepciones, errores síncronos. Un ejemplo de esto y de cómo afecta en la decisión de si un error es buen candidato para ser tratado como excepción, se verá en el siguiente capítulo de Programación Concurrente.

Java

Al igual que C++, Java comparte el mismo modelo de manejo de excepciones que C++, pero a diferencia de éste, Java cuenta con un soporte más completo. A continuación se detallan las principales diferencias:

TODOS LOS TIPOS DE EXCEPCIONES HEREDAN DE UNA CLASE BASE

El árbol de herencia de las clases para excepciones es:

```
java.lang.Object  
└─> java.lang.Throwable  
    ├──> java.lang.Error  
    ├──> java.lang.Exception  
    └─> java.lang.RuntimeException
```


La clase **Throwable** es la clase base de todos los errores y excepciones de Java manejados por el mecanismo de manejo de excepciones. El intérprete de Java y el programador sólo pueden disparar excepciones de un tipo que derive de Throwable.

La clase **Error** es la clase base de excepciones generadas por el intérprete de Java, por lo que no deben ser manejadas por el programador, dado que son errores para los que sólo el intérprete puede dar una solución satisfactoria.

La clase **Exception** es la clase base de todas excepciones con las que el programador sí debe lidiar, a excepción de las que derivan de **RuntimeException**, las que tienen un tratamiento especial.

Si un programa deseara crear sus propias excepciones, heredaría de la clase Exception.

LAS EXCEPCIONES DEBEN SER TRATADAS POR EL PROGRAMA

El programador **debe capturar explícitamente** las excepciones de dichos tipos, o bien **indicar explícitamente** que no desea manejarlas. Las excepciones que derivan de RuntimeException corresponden a errores de lógica en la programación, por lo que el lenguaje acepta que no sean explícitamente tratadas, dado que sólo son usadas para depurar los programas y eliminar dichos errores. Una vez eliminados estos errores, esas excepciones ya no se presentarán, y el código escrito para manejarlas dejaría de ser útil. A continuación un ejemplo ilustra este aspecto:

```
public class Ejemplo1 {
    public static void main(String args[]) {
        // sentencias fuera del bloque de control de excepciones
        int Arreglo[] = { 1, 2, 3, 4 };

        // definición del bloque try
        try {
            for( int i = 0; i < 5; i++ ) {
                int iValor = Arreglo[ i ];
                System.out.println("Valor " + i + " = " + iValor );
            }
        }
        // definición del bloque catch
        catch( ArrayIndexOutOfBoundsException e ) {
            System.out.println("Ocurrió la siguiente excepción = " + e );
        }
        // definición del bloque finally
        finally {
            System.out.println("Ejecución del bloque finally" );
        }

        // sentencias fuera del bloque de control de excepciones
        System.out.println("Fin del programa." );
    }
}
```

La excepción ArrayIndexOutOfBoundsException hereda de RuntimeException y es generada cuando se accede a un elemento de un arreglo utilizando un índice inválido. Esto es justamente lo que sucede en el código anterior, donde se intenta acceder al quinto elemento del arreglo (índice $i = 4$) produciéndose una excepción, dado que el arreglo sólo tiene cuatro elementos. Este error se puede subsanar reemplazando la declaración del bucle, por ejemplo, de la siguiente manera:

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

Por tanto, una vez hecho ésto, dicha excepción nunca volverá a ocurrir, por lo que todo el código escrito para el manejo de ésta pasa a ser inútil. Todo el resto de excepciones que heredan de Exception y no de RuntimeException deben ser manejadas explícitamente por el programador. El siguiente programa muestra este caso:

```
import java.io.InputStreamReader;
```

```
import java.io.BufferedReader;
import java.io.IOException;

class Ejemplo3 {

    public static void main(String args[]) {
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
        try {
            System.out.print("Ingrese un primer entero : ");
            String cadena1 = input.readLine();
            System.out.print("Ingrese un segundo entero : ");
            String cadena2 = input.readLine();
            int entero1 = Integer.parseInt(cadena1);
            int entero2 = Integer.parseInt(cadena2);
            System.out.println("La suma da : " + (entero1 + entero2));
        }
        catch(IOException ex) {
            System.out.println("Ocurrio la sgte. excepcion al leer: " + ex);
        }
    }
}
```

El código anterior utiliza el método **readLine** de la clase **BufferedReader** para leer el texto ingresado por teclado desde la ventana de comandos del programa. Dicho método puede producir una excepción del tipo **IOException**, la cual no deriva de **RuntimeException**, por lo que tenemos la obligación de capturarla. Otra opción es indicar explícitamente que no deseamos manejar dicha excepción mediante la cláusula **throws**. Esta cláusula se coloca luego de la declaración de parámetros de un método. El siguiente código modifica el anterior para utilizar una cláusula **throws**.

```
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;

class Ejemplo3 {

    public static void main(String args[]) throws IOException {
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

        System.out.print("Ingrese un primer entero : ");
        String cadena1 = input.readLine();
        System.out.print("Ingrese un segundo entero : ");
        String cadena2 = input.readLine();

        int entero1 = Integer.parseInt(cadena1);
        int entero2 = Integer.parseInt(cadena2);
        System.out.println("La suma resultante es : " + (entero1 + entero2));
    }
}
```

La cláusula **throws** puede ir precedida de una lista de tipos de excepciones, separadas por comas. Esta cláusula le dice al compilador de Java “sé que el código en éste método produce estas excepciones, pero no deseo manejarlas aquí, sino en el método que llame a éste”.

Si el código en nuestro programa produce excepciones que deben ser manejadas explícitamente, ya sea mediante una secuencia **throw-catch** o con una cláusula **throws**, y no lo son, se generarán errores durante la compilación indicando que esto debe hacerse.

SE DEFINE UN BLOQUE ESPECIAL QUE SIEMPRE SE EJECUTA

El problema de la pérdida de recursos del modelo de manejo de excepciones en C++ se soluciona en Java mediante un nuevo bloque, llamado **finally**, que puede ir a continuación de un bloque **try** o del último bloque **catch**, este nuevo bloque siempre se ejecutará, ocurra o no una excepción dentro de **try** y en caso ocurra sea o no capturada por uno de los bloques **catch**. El siguiente código muestra un ejemplo del funcionamiento de **finally**.

```
class Ejemplo3 {
    static public void NoDisparaExcepciones() {
        System.out.println("Inicio de 'NoDisparaExcepciones'");
        try {
            System.out.println("Dentro del bloque try de 'NoDisparaExcepciones'");
        }
        finally {
            System.out.println("Dentro del bloque finally de 'NoDisparaExcepciones'");
        }
        System.out.println("Fin de 'NoDisparaExcepciones'");
    }

    static public void DisparaCapturaExcepcion() {
        System.out.println("Inicio de 'DisparaCapturaExcepcion'");
        try {
            System.out.println("Dentro del bloque try de 'DisparaCapturaExcepcion'");
            throw new Exception("Excepcion en try de 'DisparaCapturaExcepcion'");
            // el codigo restante de este try nunca se ejecutará
        }
        catch(Exception ex) {
            System.out.println("Dentro del bloque catch de " +
                "'DisparaCapturaExcepcion', excepcion=" + ex.getMessage());
        }
        finally {
            System.out.println("Dentro del bloque finally de " +
                "'DisparaCapturaExcepcion'");
        }
        System.out.println("Fin de 'DisparaCapturaExcepcion'");
    }

    static public void DisparaNoCapturaExcepcion() throws Exception {
        System.out.println("Inicio de 'DisparaNoCapturaExcepcion'");
        try {
            System.out.println("Inicio del bloque try" +
                " de 'DisparaNoCapturaExcepcion'");
            throw new Exception("Excepcion dentro del try" +
                " de 'DisparaNoCapturaExcepcion'");
            // el codigo restante de este try nunca se ejecutará
        }
        finally {
            System.out.println("Dentro del bloque finally " +
                " de 'DisparaNoCapturaExcepcion'");
        }
        // el codigo restante de este metodo nunca se ejecutará
    }

    static public void DisparaCapturaRedisparaExcepcion() throws Exception {
        System.out.println("Inicio de 'DisparaCapturaRedisparaExcepcion'");
        try {
            System.out.println("Inicio del try de " +
                "'DisparaCapturaRedisparaExcepcion'");
            throw new Exception("Excepcion en try de " +
                "'DisparaCapturaRedisparaExcepcion'");
            // el codigo restante de este try nunca se ejecutará
        }
        catch(Exception ex) {
            System.out.println("Dentro del bloque catch de " +
                "'DisparaCapturaRedisparaExcepcion', excepcion=" +
                ex.getMessage());
            throw ex;
        }
        finally {
            System.out.println("Dentro del bloque finally de " +
                "'DisparaCapturaRedisparaExcepcion'");
        }
        // el codigo restante de este try nunca se ejecutará
    }

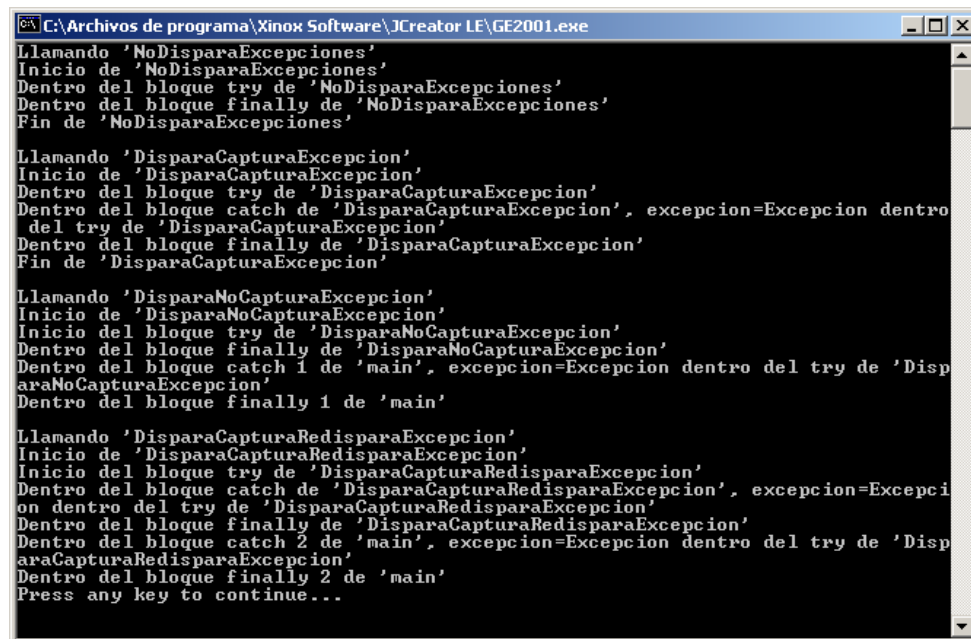
    public static void main(String args[]) {
        System.out.println("Llamando 'NoDisparaExcepciones'");
        NoDisparaExcepciones();
        System.out.println("\nLlamando 'DisparaCapturaExcepcion'");
        DisparaCapturaExcepcion();
    }
}
```

```
System.out.println("\nLlamando 'DisparaNoCapturaExcepcion'");
try {
    DisparaNoCapturaExcepcion();
}
catch(Exception ex) {
    System.out.println("Dentro del bloque catch 1 de 'main', excepcion="
+
    ex.getMessage());
}
finally {
    System.out.println("Dentro del bloque finally 1 de 'main'");
}
System.out.println("\nLlamando 'DisparaCapturaRedisparaExcepcion'");
try {
    DisparaCapturaRedisparaExcepcion();
}
catch(Exception ex) {
    System.out.println("Dentro del bloque catch 2 de 'main', excepcion="
+ex.getMessage());
}
finally {
    System.out.println("Dentro del bloque finally 2 de 'main'");
}
}
```

El código anterior genera una salida semejante a la Figura 7 - 3.

Ésto da una clara idea de que el bloque finally se ejecuta siempre que se haya entrado a ejecutar el bloque try correspondiente. La lista de combinaciones try-catch-finally que se pueden tener es:

- Un bloque try seguido de uno o más bloques catch.
- Un bloque try seguido de uno o más bloques catch, seguidos de un bloque finally.
- Un bloque try seguido de un bloque finally.



```
C:\Archivos de programa\Xinox Software\JCreator LE\GE2001.exe
Llamando 'NoDisparaExcepciones'
Inicio de 'NoDisparaExcepciones'
Dentro del bloque try de 'NoDisparaExcepciones'
Dentro del bloque finally de 'NoDisparaExcepciones'
Fin de 'NoDisparaExcepciones'

Llamando 'DisparaCapturaExcepcion'
Inicio de 'DisparaCapturaExcepcion'
Dentro del bloque try de 'DisparaCapturaExcepcion'
Dentro del bloque catch de 'DisparaCapturaExcepcion', excepcion=Excepcion dentro
del try de 'DisparaCapturaExcepcion'
Dentro del bloque finally de 'DisparaCapturaExcepcion'
Fin de 'DisparaCapturaExcepcion'

Llamando 'DisparaNoCapturaExcepcion'
Inicio de 'DisparaNoCapturaExcepcion'
Inicio del bloque try de 'DisparaNoCapturaExcepcion'
Dentro del bloque finally de 'DisparaNoCapturaExcepcion'
Dentro del bloque catch 1 de 'main', excepcion=Excepcion dentro del try de 'Disp
araNoCapturaExcepcion'
Dentro del bloque finally 1 de 'main'

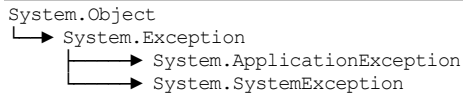
Llamando 'DisparaCapturaRedisparaExcepcion'
Inicio de 'DisparaCapturaRedisparaExcepcion'
Inicio del bloque try de 'DisparaCapturaRedisparaExcepcion'
Dentro del bloque catch de 'DisparaCapturaRedisparaExcepcion', excepcion=Excepci
on dentro del try de 'DisparaCapturaRedisparaExcepcion'
Dentro del bloque finally de 'DisparaCapturaRedisparaExcepcion'
Dentro del bloque catch 2 de 'main', excepcion=Excepcion dentro del try de 'Disp
araCapturaRedisparaExcepcion'
Dentro del bloque finally 2 de 'main'
Press any key to continue...
```

Figura 7 - 3 Ejecución de ejemplo de excepciones en Java

C#

C# comparte el mismo modelo de manejo de excepciones que C++, pero a diferencia de éste, C# cuenta con un soporte más completo, muy similar al de Java.

El árbol de herencia de las clases para excepciones es:



Al igual que Java, la clase Exception es la clase base de todos los tipos de excepciones. La clase SystemException es la clase base que utiliza el intérprete de .NET para todas las excepciones que puede generar. La clase ApplicationException es la clase base que debe utilizar el programador para definir sus propias excepciones.

A diferencia de Java, C# no diferencia entre excepciones que deben tratarse y las que sólo sirven para depuración, es decir, si un determinado código puede generar una excepción, no es obligatorio colocarlo dentro de un bloque try seguido de un catch que lo capture. Por el mismo motivo, tampoco existe una cláusula throws con la que se indique que una excepción no se desea tratar en el método donde ocurre.

Fuera de las diferencias antes indicadas, la implementación en Java y C# es igual. Un código de ejemplo de excepciones en C# es:

```
using System;

class ExcepcionFormatoInvalido : ApplicationException {
    public ExcepcionFormatoInvalido(string mensaje) : base(mensaje) {}
}
class ExcepcionFechaInvalida : ApplicationException {
    public ExcepcionFechaInvalida(string mensaje) : base(mensaje) {}
}
class Fecha {
    int dia, mes, anho;
    public Fecha(string sFecha) {
        char[] delimitadores = {'-', '/'};
        string[] tokens = sFecha.Split(delimitadores);
        if(tokens.Length != 3)
            throw new ExcepcionFormatoInvalido("Los delimitadores son
    inválidos");
        dia = Convert.ToInt32(tokens[0]);
        mes = Convert.ToInt32(tokens[1]);
        anho = Convert.ToInt32(tokens[2]);
        if(anho < 0 || dia < 1 || 31 < dia || mes < 1 || 12 < mes)
            throw new ExcepcionFechaInvalida("La fecha no existe");
    }
    public void Imprimir() {
        Console.WriteLine "[" + dia + "/" + mes + "/" + anho + "]";
    }
};
class MainClass {
    public static void Main(string[] args) {
        Console.WriteLine("Inicio");
        try {
            Console.Write("Ingrese una fecha : ");
            string sFecha = Console.ReadLine();
            Fecha fecha = new Fecha(sFecha);
            Console.Write("La fecha ingresada es : ");
            fecha.Imprimir();
        }
        catch(Exception ex) {
            Console.WriteLine("Excepción = " + ex);
        }
        finally {
            Console.WriteLine("Ejecutandose el bloque finally");
        }
    }
}
```

```
}  
    Console.WriteLine("Fin");  
}  
}
```

Ventajas, desventajas y criterios de uso

La separación del código limpio del código de manejo de excepciones permite eliminar las desventajas mencionadas con las técnicas tradicionales. Por lo tanto, las excepciones:

- Permiten tener un código limpio, por tanto, es más fácil de mantener.
- Eliminación del código repetitivo en el manejo de una misma excepción.
- Es más fácil agregar código para el manejo de una excepción que se dejó “para después” al inicio de un programa, por tanto, es más fácil aumentar la tolerancia a fallos de éste conforme se va avanzando en su desarrollo.
- Dado que no se realizan todas las verificaciones para las excepciones en el código limpio del programa, el compilador puede optimizar su verificación, lo que reduce y en algunos casos elimina el costo de eficiencia en la ejecución del programa final.

Es importante resaltar que la técnica mostrada del manejo de excepciones es más eficiente que la tradicional cuando no ocurre una excepción, pero es mucho menos eficiente cuando sí ocurre. Por lo tanto, es fácil deducir que si un error muy frecuente es tratado como una excepción, el programa final sería menos eficiente que si sólo utilizara técnicas tradicionales para dichos errores.

Es importante tomar en cuenta las siguientes desventajas propias al manejo de excepciones:

- Al estar el código de manejo de excepciones separado del código limpio del programa, se dificulta entender el flujo de ejecución del programa cuando ocurre una excepción.
- Por el mismo motivo anterior, es más difícil relacionar un código de manejo de una excepción con la fuente del mismo.

En general, la mayoría de los errores pueden ser tratados como excepciones (existen algunos que sólo pueden ser manejados como excepciones), pero no todos los errores son buenos candidatos. Algunos criterios que pueden servir para decidir si un error debe ser tratado como una excepción son:

- El error es infrecuente.
- En el contexto donde ocurre el error no es posible darle una solución satisfactoria. Un ejemplo típico es un error dentro del constructor de una clase.
- El error está relacionado con un mal funcionamiento del sistema operativo o de alguna librería con la que interactúa el programa. Un error en el ingreso de datos por parte del usuario es un mal candidato para una excepción, pero un error en la reserva de memoria sí lo es.
- El error es síncrono.

El **manejador de excepción** es un código que comunmente se coloca en un nivel más bajo en el árbol de llamada a métodos, que el código que dispara la excepción. Eso permite abarcar las excepciones posibles desde muchos métodos. Por ejemplo, si el método C es llamado por el método B, y éste por el método A, y el método C dispara una excepción, el código “exception handler” de éste puede colocarse tanto en el método A, como en el B o en el C. Luego, las

excepciones son utilizadas para tipos de errores que pueden producirse en diferentes lugares a lo largo de un programa y que se desea sean procesados de manera centralizada. Sin embargo, es importante tener en cuenta que cuanto más lejos esté el manejador de excepción del **punto de excepción**, más difícil podría ser manejar adecuadamente dicha excepción, dado que la misma podría ser generada por más de un punto de excepción y podría requerirse código especial para cada caso.

Por último, dado que las técnicas tradicionales de tratamiento de errores se realizan con las mismas estructuras utilizadas para el control del flujo de un programa, es frecuente la tendencia a querer utilizar las técnicas de manejo de excepciones para tareas diferentes al tratamiento de excepciones. Ésto debe evitarse, dado que reduciría la eficiencia del programa y la claridad de su código.

El tema de la sincronización se verá en el siguiente capítulo, Programación Concurrente.

Programación con GUI

El presente capítulo se centra en el trabajo con Interfaces Gráficas de Usuario (IGU por sus siglas en español, GUI por sus siglas en inglés) utilizando ventanas y otros elementos para el diseño de las mismas, dentro del sistema operativo Windows. Para el caso particular de Java, dada su característica multiplataforma, los conceptos dados aquí son aplicables a cualquier plataforma que soporte Java.

Interfaces GUI con Ventanas

Una interfaz gráfica de usuario (GUI por sus siglas en inglés) es el conjunto de elementos gráficos que un programa, ejecutándose en un computador, utiliza para permitirle al usuario interactuar “visualmente” con él. Un programa GUI utiliza hardware que lo asiste (monitor y tarjeta de video) trabajando en “modo gráfico”. Si bien el texto es, en esencia, un gráfico, las interfaces basadas exclusivamente en texto (TUI por sus siglas en inglés), con monitores trabajando en “modo texto”, son diferenciadas de las anteriores.

Un programa puede trabajar con una TUI, con una GUI o con ambas, aunque en éste último caso lo que se tiene es realmente un programa trabajando en “modo gráfico” emulando el comportamiento de un “modo texto”.

Casi desde sus inicios (que se remontan, por lo menos, a 1973, con la primera computadora Alto de la empresa Xerox PARC puesta en funcionamiento, o más conocida, la Star de Xerox en 1981), el concepto de ventanas formó parte del concepto de GUI, como una estrategia de organización del área gráfica.

Tipo de Interfaces GUI con Ventanas

En los programas GUI con ventanas, el usuario utiliza dispositivos como el teclado y el ratón, cuyo uso envía mensajes al computador y éstos son capturados por el sistema operativo, el cual decide a qué ventanas pertenecen dichos mensajes y los “envía” a las aplicaciones correspondientes para que éstas realicen alguna acción. Es importante tener en cuenta que este mecanismo de mensajes es utilizado por otros programas, incluyendo el propio sistema operativo, para “generar” nuevos mensajes, no relacionados con la interacción gráfica mediante las ventanas, y “enviarlos” a los programas en ejecución. Ejemplos de éstos son los mensajes de comunicación entre programas (corriendo en la misma computadora o en computadoras diferentes conectadas a una red), los mensajes de aviso de los cambios en la configuración del

sistema operativo (resolución de la pantalla, idioma del teclado, fecha, cambio de usuario, etc.), los relacionados a la escasez de recursos, etc.

Este sistema de mensajería, con distintas variantes, es utilizado por los sistemas operativos con soporte GUI. En particular veremos el caso del sistema operativo Windows y cómo su sistema de mensajería es manejado desde diferentes lenguajes de programación.

En la actualidad, existen dos tipos de programas GUI con ventanas comunmente utilizados: Los programas Stand-Alone y los programas dentro de navegadores de Internet (browsers).

Programas Stand-Alone

Los programas Stand-Alone crean una o más ventanas con las que el usuario interactúa. Comunmente estos programas poseen una ventana principal y una o más ventanas especializadas en alguna labor específica. Estos programas pueden ejecutar directamente (en caso de ser programas ejecutables) o mediante un programa intérprete.

Ejemplos de estos programas son los creados con C o C++ utilizando directamente el API de Windows, así como las Aplicaciones de Java y las Aplicaciones de Formularios de Ventanas de .NET.

Programas Basados en Web

Los programas basados en Web crean contenido para los navegadores Web clientes, utilizados por los usuarios de la Web, como por ejemplo el navegador Web “Internet Explorer” y el “Netscape”. Este contenido Web puede incluir código HTML, scripts ejecutados del lado del cliente, imágenes y datos binarios. Estos programas requieren, para su ejecución, de un navegador Web que los soporte. Si bien, de primera instancia, estos programas utilizan el área de dibujo de la ventana del programa navegador, pueden crear otras ventanas.

Ejemplos de estos programas son los Applets de Java y los Web Forms de .NET.

Creación y Manejo de GUI con Ventanas

Si bien un programa basado en Web puede crear ventanas, adicionalmente a la ventana del programa navegador que lo ejecuta, y muchos de los conceptos de creación y manipulación de los elementos de una ventana se aplican casi idénticamente que en los programas basados en Web, los programas Stand-Alone con ventanas son más simples y sólo dependen de las ventanas que ellos mismos crean. Debido a ésto, toda la explicación respecto a la creación y manejo de GUIs con ventanas se realizará para programas Stand-Alone.

Creación de una Ventana

Veremos cómo se crea un programa mínimo, con una única ventana, en Java, C# y C/C++ con API de Windows.

El usar directamente el API de Windows nos ayudará a entender cómo funciona el sistema de mensajería de Windows. El siguiente código corresponde a un programa en C o C++ que crea una ventana completamente funcional.

```
#include <windows.h>
#include <stdio.h>

LRESULT CALLBACK FuncionVentana(
    HWND hWnd,          // Manejador (handle) de la ventana a la que corresponde el mensaje.
    UINT uMsg,           // Identificador del mensaje.
```

```
WPARAM wParam, // Primer parámetro del mensaje.
LPARAM lParam ) // Segundo parámetro del mensaje.
{
    switch( uMsg )
    {
        case WM_DESTROY:
            // La función 'PostQuitMessage' crea un mensaje WM_QUIT y lo introduce
            // en la cola de mensajes. El parámetro que se le pasa, establece el
            // valor de 'wParam' del mensaje WM_QUIT generado.
            PostQuitMessage( 0 );
            break;
        default:
            // Todos los mensajes para los que no deseo realizar ninguna
            // acción, los paso a la función 'DefWindowProc', la que realiza
            // las acciones por defecto correspondientes a cada mensaje.
            return DefWindowProc( hWnd, uMsg, wParam, lParam );
    }
    return 0;
}

BOOL RegistrarClaseVentana( HINSTANCE hIns )
{
    WNDCLASS wc;
    wc.style          = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc    = FuncionVentana;
    wc.cbClsExtra      = 0;
    wc.cbWndExtra      = 0;
    wc.hInstance      = hIns;
    wc.hIcon           = LoadIcon( NULL, IDI_APPLICATION );
    wc.hCursor         = LoadCursor( NULL, IDC_ARROW );
    wc.hbrBackground   = ( HBRUSH )GetStockObject( WHITE_BRUSH );
    wc.lpszMenuName     = NULL;
    wc.lpszClassName   = "Nombre_Clase_Ventana";
    return ( RegisterClass( &wc ) != 0 );
}

HWND CrearInstanciaVentana( HINSTANCE hIns )
{
    HWND hWnd;
    hWnd = CreateWindow(
        "Nombre_Clase_Ventana",
        "Titulo de la Ventana",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        NULL,
        NULL,
        hIns,
        NULL
    );
    return hWnd;
}

int WINAPI WinMain(
    HINSTANCE hIns,          // Manejador (handle) de la instancia del programa.
    HINSTANCE hInsPrev,     // Manejador (handle) de la instancia de un programa,
                             // del mismo tipo, puesto en ejecución previamente.
    LPSTR lpCmdLine,        // Puntero a una cadena (char*) conteniendo la línea de
                             // comando utilizada al correr el programa.
    int iShowCmd )          // Un entero cuyo valor indica cómo debería mostrarse
                             // inicialmente la ventana principal del programa.
{
    // Registro una clase de ventana, en base a la cual
    // se creará una ventana.
    if( ! RegistrarClaseVentana( hIns ) )
        return 0;

    // Creo una ventana.
    HWND hWnd = CrearInstanciaVentana( hIns );
    if( hWnd == NULL )
```

```
        return 0;

    // Muestro la ventana.
    ShowWindow( hWnd, iShowCmd );    // Establece el estado de 'mostrado' de la ventana.
    UpdateWindow( hWnd );            // Genera un mensaje de pintado, el que es
                                    // ejecutado por la función de ventana respectiva.

    // Se realiza un bucle donde se procesen los mensajes de la cola de mensajes.
    MSG Mensaje;
    while( GetMessage( &Mensaje, NULL, 0, 0 ) > 0 )
        DispatchMessage( &Mensaje );

    // GetMessage devuelve 0 cuando el mensaje extraído es WM_QUIT y
    // -1 cuando ha ocurrido un error. Note que esto produce que el mensaje
    // WM_QUIT nunca sea procesado por la función de ventana.

    // El parámetro 'wParam' del mensaje 'WM_QUIT' corresponde al parámetro
    // pasado a la función 'PostQuitMessage'. Este valor tiene la misma utilidad
    // que el entero retornado por la función 'main' en programas en C y C++ para
    // consola.
    return Mensaje.wParam;
}
```

En este programa existen dos funciones importantes: El punto de entrada del programa, la función “WinMain”, y la función de ventana “FuncionVentana”.

La función WinMain es el equivalente, para un programa en Windows, a la función “main” en programas en C y C++ en modo consola. El principal parámetro de esta función es el primero, el manejador de la instancia del programa. Dicho manejador es, en esencia, un número entero utilizado por Windows para ubicar los recursos relacionados al programa. Algunos de los recursos que un programa en Windows puede tener son: Registros de ventanas, textos, imágenes, audio, video, meta-archivos, otros manejadores (a archivos, puertos, impresoras, etc.), etc. El manejador de la instancia es utilizado como parámetro de las funciones del API de Windows donde se involucren, directa o indirectamente, los recursos de una aplicación. El segundo parámetro no es utilizado en programas de 32-bits (Windows 95 y Windows NT en adelante).

La función de ventana (que puede tener cualquier nombre pero con el mismo formato de declaración que el mostrado) es dónde se realiza toda la lógica relacionada a las acciones que una ventana toma en respuesta a los mensajes recibidos, como por ejemplo, los producidos por la interacción del usuario con la ventana. La dirección de esta función de ventana es el principal dato que forma parte del registro, con la estructura WNDCLASS, de una clase de ventana. Todos los mensajes enviados a ventanas, creados con una misma clase, son procesados por la misma función de ventana, la indicada en la estructura WNDCLASS al registrarse dicha clase de ventana.

Todo programa con ventanas en Windows tiene 3 etapas principales:

13. Registro de las clases de ventana que se utilizarán en el programa para crear ventanas.
14. Creación de la ventana principal del programa.
15. Ejecución del bucle de mensajes del programa.

Para crear una ventana se utiliza la función CreateWindow. Los parámetros de esta función indican, en este orden:

22. El nombre de la clase de ventana en base a la que se crea la nueva ventana.
23. El título a mostrarse en la barra superior de la ventana.

24. El tipo y atributos (incluyendo estilo) de la ventana.
25. La coordenada X, en píxeles, de la esquina superior izquierda en la que aparecerá la ventana dentro del área de la pantalla. La constante CW_USEDEFAULT indica que se utilice un valor por defecto.
26. La coordenada Y, en píxeles, de la esquina superior izquierda en la que aparecerá la ventana dentro del área de la pantalla. La constante CW_USEDEFAULT indica que se utilice un valor por defecto.
27. El ancho de la ventana, en píxeles. La constante CW_USEDEFAULT indica que se utilice un valor por defecto.
28. El alto de la ventana, en píxeles. La constante CW_USEDEFAULT indica que se utilice un valor por defecto.
29. El manejador de una ventana padre. Si se pasa NULL (una constante igual a cero) se indica que la ventana no tiene padre.
30. Un manejador de menú o un identificador de ventana (dependiendo de los valores pasados en el tercer parámetro). Si se pasa NULL se indica que no se utilizará esta característica.
31. El manejador de la instancia del programa.
32. Un puntero genérico (void*) a cualquier información extra que el usuario quiera utilizar. Si se pasa NULL, no se utiliza esta característica.

Creada la ventana principal del programa, se ingresa al bucle de procesamiento de mensajes. Dicho bucle realiza, repetitivamente, las siguientes acciones:

33. Retirar un mensaje de la cola de mensajes, mediante la función GetMessage.
34. Mandar a llamar a la función de ventana correspondiente, mediante la función DispatchMessage.

La estructura MSG almacena los mismos datos que recibe una función de ventana como parámetros. La función GetMessage llena esta estructura con la información del mensaje retirado de la cola de mensajes del programa. La función DispatchMessage utiliza esta información para averiguar a qué ventana corresponde el mensaje, cuál es la clase de ventana de dicha ventana, cuál es la función de ventana registrada con esa clase y, finalmente, llama a dicha función de ventana pasándole los datos almacenados en la estructura MSG.

Es importante notar el hecho de que, si bien se dice en la literatura que la función de ventana es llamada por Windows, no es estrictamente así. Tanto DispatchMessage como otras funciones del API de Windows son las que realmente se encargan de llamar a las funciones de ventana, y dichas funciones son llamadas explícitamente desde el programa. Como puede deducirse, es debido a que estas funciones forman parte del API de Windows, que se dice que es Windows quién llama a las funciones de ventana. Un programa en ejecución, en un sistema operativo multitarea como Windows, no puede llamar directamente a una función de otro programa en memoria. Este aspecto está íntimamente relacionado al tema de programación concurrente, por lo que no ahondaremos por ahora más en esto.

Otro hecho importante es el que todo programa en Windows, sea hecho en C, en Java, en C# o en cualquier otro lenguaje de programación (salvo alguna excepción) presenta, escondida o no, esta estructura de trabajo. Todo programa en Windows requiere registrar las clases de ventana

que utiliza, crear dichas ventanas y tener un bucle de procesamiento de mensajes. Para ilustrar mejor ésto revisemos ejemplos equivalentes en Java y C#.

El siguiente programa es el equivalente en Java al programa anterior.

```
import javax.swing.*;
import java.awt.event.*;
class MiVentana extends JFrame {
    public MiVentana() {
        setSize(400, 400);
        setTitle("Título de la Ventana");
        setVisible(true);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                dispose();
                System.exit(0);
            }
        });
    }
}
class Aplicacion Minima {
    public static void main(String[] args) {
        MiVentana ventana = new MiVentana();
    }
}
```

En este código, el punto de entrada es el método estático “main”, dentro del cual lo único que se realiza es la creación de un objeto de la clase MiVentana que hereda de la clase JFrame (cuyo nombre completo es javax.swing.JFrame), que es la clase base para la creación de ventanas utilizando el paquete (el término utilizado en Java para referirse a una librería) SWING, cuyas clases son accesibles mediante la sentencia “import javax.swing.*”. Dentro del constructor de la clase “MiVentana” se configuran las características de la ventana y se establece “qué objeto” será al que se le llame su método “windowClosing” cuando el usuario del programa indique que desea cerrar la ventana.

Ahora bien, relacionemos todo esto con lo visto anteriormente en el programa en C o C++ con API de Windows.

Como es de esperarse, la clase JFrame debe crear en su constructor (o en algún constructor de sus clases base) una ventana llamando a “CreateWindow” (u otra función equivalente). Como ya hemos visto, no se debe poder crear una ventana antes de registrar la clase de ventana en base a la que se creará, por lo que es de esperarse (y realmente sucede así) que el programa intérprete de Java realice este registro antes de llamar a nuestro método “main”. Siguiendo el orden de ejecución de los constructores, luego de completada la ejecución del constructor de JFrame se llamará al de nuestra clase “MiVentana”, donde modificamos los valores por defecto con los que se creó inicialmente la ventana (por defecto, JFrame crea una ventana en la coordenada [X,Y] = [0,0], con cero píxeles de ancho y cero de alto y con su atributo de visibilidad puesto en “no-visible”). El constructor termina indicando, mediante el método de JFrame “addWindowListener”, a qué método de qué clase se llamará cuando la función de ventana, de la ventana creada por JFrame, reciba el mensaje WM_CLOSE. El procesamiento por defecto, realizado por “DefWindowProc”, para este mensaje es llamar a la función de API de Windows “DestroyWindow”, que es la que realmente destruye la ventana generando, de paso, el mensaje WM_DESTROY. Como puede deducirse, el método “dispose” de JFrame debería estar llamando a DestroyWindow y el método “System.exit” a PostQuitMessage.

Finalmente queda algo muy importante que no es directamente visible en nuestro código: ¿Dónde se ejecuta el bucle de procesamiento de mensajes? Para explicar ésto, debe aclararse que el programa intérprete de Java realiza algunas acciones luego de llamar a nuestro método “main”, entre ellas está la de verificar si después de ejecutarse este método se creó o no alguna

ventana. Si se creó entonces se entra a un bucle de procesamiento de mensajes del que, como puede esperarse, el programa no sale hasta haberse llamado al método “System.exit”. Si no se creó ninguna ventana, el programa intérprete finaliza. Es por esto que la literatura sobre programación con ventanas en Java indica, sin dar mayor detalle, que si se creara alguna ventana en un programa Java Stand-Alone (a éstos se les llama “Aplicaciones Java”), debe de llamarse en algún momento al método “System.exit”.

Ahora bien, el siguiente programa es el equivalente en C# del programa anterior.

```
using System;
using System.Windows.Forms;

class MiVentana : Form {
    public MiVentana() {
        Size = new System.Drawing.Size(400, 400);
        Text = "Título de la Ventana";
        Visible = true;
    }
}

class Aplicacion_Minima {
    public static void Main(string[] args) {
        MiVentana ventana = new MiVentana();
        Application.Run(ventana);
    }
}
```

En el código, el punto de entrada es el método estático “Main” (a diferencia de Java, C# ofrece varias sobrecargas posibles: Con y sin parámetros, con y sin valor de retorno), dentro del cuál se crea una objeto de la clase MiVentana que hereda de la clase Form (cuyo nombre completo es System.Windows.Forms.Form), que es la clase base para la creación de ventanas utilizando un ensamblaje (el término utilizado en C# para referirse a una librería) System.Windows.Forms, también llamado Windows Forms, cuyas clases son accesibles mediante la sentencia “using System.Windows.Forms”. Dentro del constructor de la clase “MiVentana” se configuran las características de la ventana.

Nuevamente, ¿cómo se relaciona todo ésto con lo visto anteriormente en el programa en C o C++ con API de Windows?

Como es de esperarse, la clase Form actúa en forma muy similar a la clase JFrame de Java, creando una ventana utilizando funciones como CreateWindow, para luego modificar los valores por defecto de creación (a diferencia de Java, dicha ventana es por defecto visible, con una posición [X,Y] y un ancho y alto con valores por defecto) en el constructor de “MiVentana”. A diferencia de Java, Form sí contiene una implementación por defecto cuando se desea cerrar la ventana, por lo que no se requiere escribir algún código al respecto. El porqué Java sí lo requiere y C# no, se debe a que Java no tiene forma de saber cuál es nuestra ventana principal, en caso hayamos creado más de una. Por el contrario, C# requiere que se lo indiquemos al llamar al método estático “Application.Run”. Como puede deducirse, este método realiza el bucle de procesamiento de mensajes.

Como puede verse, no importa el lenguaje de programación utilizado, o qué tanto dicho lenguaje nos oculte la implementación interna, tras capas de abstracción (en forma de clases por ejemplo), siempre debemos tener claro que dicha implementación debe necesariamente contener los elementos mostrados en el programa en C o C++ anterior, y por tanto, debe utilizar las funciones del API de Windows que ese programa utiliza. En algunos casos, dichos lenguajes de programación ofrecen acceso a elementos de bajo nivel del API de Windows, como los manejadores de las ventanas que crean y utilizan (por ejemplo Visual Basic, a manera de propiedades de algunos de sus controles visuales), de manera que se acceda a cierta funcionalidad que sólo provee dicha API.

Elementos de una Ventana

Un conjunto de programas GUI con ventanas, que comparten el mismo conjunto de librerías para la manipulación de éstas (en Windows, estas librerías forman parte del mismo sistema operativo, y se les llama API de Windows), comparten no sólo un aspecto común, sino un conjunto de elementos gráficos con un comportamiento y uso común. A esto se le conoce como el “Look And Feel” de dicho entorno GUI.

El hecho de tener elementos de aspecto y comportamiento común reduce la curva de aprendizaje para que un usuario, que ya aprendió a utilizar uno de estos programas, aprenda a utilizar otro que utilice la misma librería GUI.

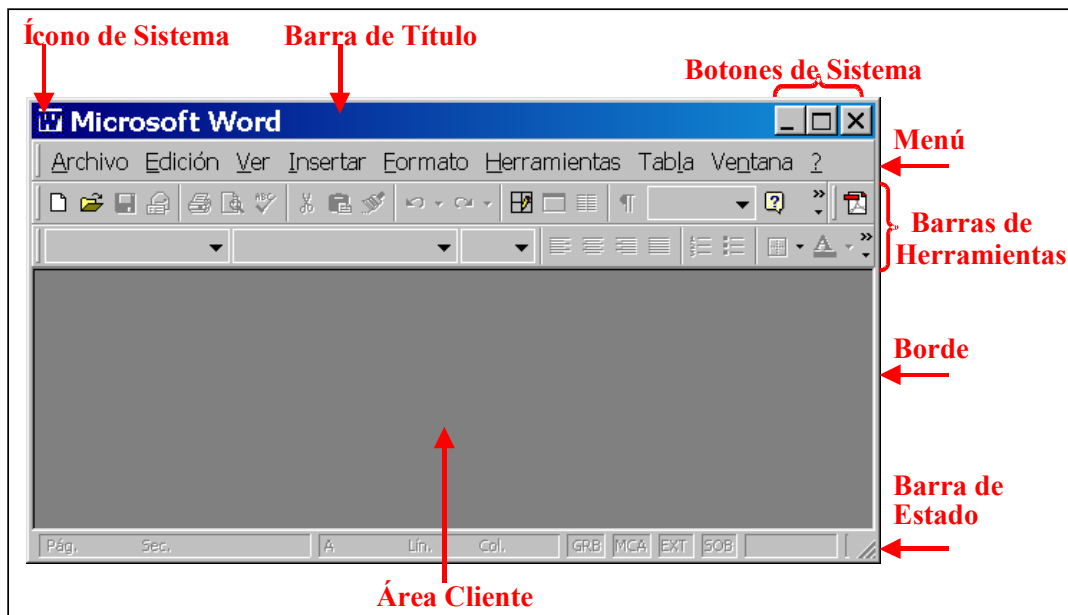


Figura 6 - 6 Elementos de una ventana

En Windows, como se aprecia en la Figura 6 - 1, las ventanas tienen los siguientes elementos:

- Un borde.
- Una barra de título.
- Un ícono de sistema.
- Un conjunto de botones de sistema.
- Un menú.
- Una o más barras de herramientas.
- Una barra de estado.
- Un área de dibujo o “área cliente”.

De estos elementos, sólo el último es obligatorio, y si bien los demás son comunes, algunas ventanas pueden no tenerlos.

Interacción con el Entorno

Cada uno de los elementos de una ventana tiene una forma de interacción con el usuario del programa al que pertenece dicha ventana. Internamente, como ya hemos visto, cada una de estas interacciones (el uso del teclado o el ratón) produce un mensaje, que es capturado por el sistema operativo, es colocado por éste en la cola de mensajes del programa correspondiente y finalmente es retirado de dicha cola por el bucle de procesamiento de mensajes de dicho programa. Ya hemos visto como el mecanismo original de procesamiento de estos mensajes, mediante una función de ventana, puede ser abstraído y ocultado al usuario mediante clases que hagan más sencillo este trabajo, como en el caso de Java y C#. En esta sección veremos en mayor detalle, cómo son estas estrategias de abstracción y qué tipos de mensajes se pueden procesar.

Manejo de Eventos con API de Windows

La estrategia de manejo de eventos del sistema operativo Windows, y por tanto del API de Windows, es mediante un sistema de mensajería, similar al sistema de mensajería de correo físico o electrónico.

Un mensaje es el conjunto de datos que el sistema operativo recolecta para un evento dado. Como ejemplo, para el evento click de un botón del ratón, el sistema operativo crea un mensaje que contiene, entre otras cosas, en qué posición de la pantalla y con cuál botón del ratón se hizo click. Dichos mensajes, como cartas de correo, son colocados en las colas de mensajes de las ventanas a las que les corresponden, como si fueran buzones para dichas cartas. Cuando el sistema operativo le da tiempo de CPU a un programa en ejecución, dicho programa verifica si hay mensajes en su cola de mensajes, como cuando nosotros tenemos un poco de tiempo libre (o cualquier otra excusa para tomarnos un descanso) y vemos nuestro correo electrónico. Si el programa encuentra un mensaje en su cola, lo saca y lo procesa.

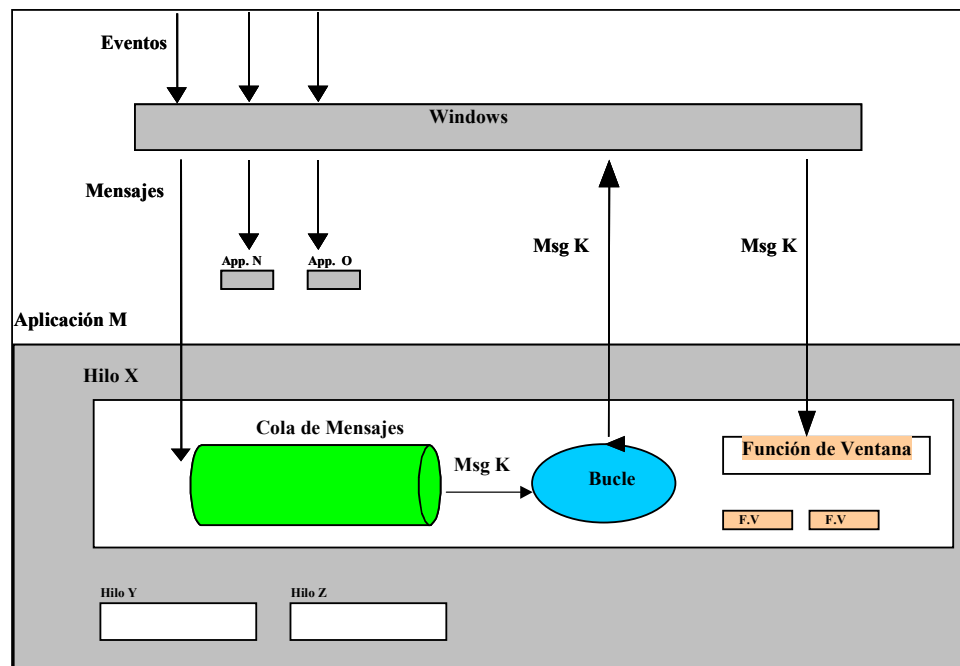


Figura 6 - 7 Procesamiento de un evento

La Figura 6 - 2 muestra el ciclo que siguen los eventos procesados en Windows para las aplicaciones que utilizan ventanas.

La secuencia de pasos seguida es:

- Windows detecta un evento.
- Crea el mensaje respectivo y lo envía a la aplicación involucrada.
- El bucle de procesamiento de mensajes detecta dicho mensaje y solicita a Windows que lo envíe a la ventana adecuada.
- Windows determina la ventana destinataria y averigua a qué clase pertenece.
- En base a la clase determina la función de ventana que le corresponde y envía el mensaje a dicho procedimiento.
- La función de ventana actúa según el mensaje recibido.

Para el procesamiento de los mensajes, toda ventana tiene una “función de procesamiento de mensajes” relacionada, a la que se le llama “función de ventana”. Dicha “relación” entre una ventana y su función de ventana se origina al crearse la ventana utilizando una plantilla de creación llamada “clase de ventana”, la cual debe haberse registrado previamente. Uno de los datos de dicha plantilla es la dirección de la función de ventana que deberá llamarse para procesar los mensajes de toda ventana que se cree utilizando dicha plantilla.

La función de ventana es pues, el área central de trabajo de todo programa desarrollado utilizando el API de Windows. El resto del código suele ser casi siempre el mismo. La función de ventana es la que determina cómo se comportará nuestro programa, nuestra ventana para el usuario, ante cada evento.

Existe un conjunto de mensajes estándar reconocidos por el sistema operativo y que nuestras funciones de ventana pueden manejar. Para cada uno de estos mensajes existe una constante relacionada. En el programa básico mostrado en la sección 3.1, se utilizó una de estas constantes: WM_DESTROY. Al igual que esta constante, definida dentro del archivo de cabecera Windows.h, existe una constante WM_XXX para cada mensaje reconocido por el sistema operativo. Es posible que un programador defina sus propios mensajes simplemente escogiendo un valor fuera del rango que Windows reserva para los desarrolladores de su sistema operativo. El manejo de mensajes definidos por el usuario cae fuera del alcance de esta introducción.

Cuando la función de ventana es llamada para procesar un mensaje, recibe los siguientes datos:

- El handle de la ventana a la que corresponde el mensaje, dado que, como hemos visto, una función de ventana puede utilizarse para procesar los mensajes de más de una ventana, cuando todas éstas fueron creadas utilizando la misma clase de ventana.
- La constante que identifica al mensaje.
- Dos parámetros que contienen información sobre dicho mensaje, o bien contienen direcciones de estructuras reservadas en memoria con dicha información.

El hacer que una ventana reconozca y reaccione a un nuevo mensaje suele consistir en agregar el “case” (al “switch” principal de la función de ventana) para la constante de dicho mensaje con el código que realice el comportamiento deseado. El siguiente código de una función de ventana muestra el manejo de un mensaje correspondiente al ratón y al teclado. El resto del programa no varía respecto al ejemplo anterior.

```
LRESULT CALLBACK FuncionVentana(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
```

```
{
    int PosX, PosY;
    char Mensaje[100];
    int CtrlPres, ShiftPres;
    int CodigoTecla, EsTeclaExtendida;

    switch( uMsg )
    {
    case WM_LBUTTONDOWN:
        PosX = LOWORD(lParam); // el word menos significativo
        PosY = HIWORD(lParam); // el word más significativo
        CtrlPres = wParam & MK_CONTROL;
        ShiftPres = wParam & MK_SHIFT;
        sprintf(Mensaje, "X=%d, Y=%d, CtrlPres=%d, ShiftPres=%d",
            PosX, PosY, CtrlPres, ShiftPres);
        MessageBox(hWnd, Mensaje, "Posición del mouse", MB_OK);
        break;
    case WM_KEYUP:
        CodigoTecla = (int)wParam;
        EsTeclaExtendida = ( lParam & ( 1 << 24 ) ) != 0;
        if( EsTeclaExtendida == 1 )
            sprintf(Mensaje, "CodigoTecla=%d (extendida)", CodigoTecla);
        else
            if( ( '0' <= CodigoTecla && CodigoTecla <= '9' ) ||
                ( 'A' <= CodigoTecla && CodigoTecla <= 'Z' ) )
                sprintf(Mensaje, "CodigoTecla=%d (%c)", CodigoTecla,
                    (char)CodigoTecla);
            else
                sprintf(Mensaje, "CodigoTecla=%d", CodigoTecla);
        MessageBox(hWnd, Mensaje, "Tecla presionada", MB_OK);
        break;
    case WM_DESTROY:
        PostQuitMessage( 0 );
        break;
    default:
        return DefWindowProc( hWnd, uMsg, wParam, lParam );
    }
    return 0;
}
```

La constante WM_LBUTTONDOWN corresponde al mensaje producido por el evento de soltar (UP) el botón izquierdo (LBUTTON) del ratón. Para este mensaje, el parámetro wParam se comporta como un bit-flag con información como “estaba la tecla CTRL presionada cuando se produjo el evento”. La información de wParam se extrae utilizando constantes definidas en Windows.h, utilizando operaciones booleanas a nivel de bits. El parámetro lParam contiene la posición del ratón al ocurrir el evento. Dicha posición es relativa a la esquina superior izquierda del área cliente de ventana. Los dos bytes menos significativos de lParam contienen la posición X del ratón; los dos bytes más significativos la posición Y, ambas coordenadas medidas en píxeles. Hay 35 mensajes relacionados con el ratón, de los cuales los más comunmente procesados son:

WM_LBUTTONDOWN	// Se presionó el botón izquierdo del ratón
WM_LBUTTONUP	// Se soltó el botón izquierdo del ratón
WM_LBUTTONDOWNBLCLK	// Se presionó dos veces seguidas el botón izquierdo del ratón
WM_MBUTTONDOWN	// Similar a los anteriores pero para el botón del medio (M)
WM_MBUTTONUP	// ...
WM_MBUTTONDOWNBLCLK	// ...
WM_RBUTTONDOWN	// Similar a los anteriores pero para el botón derecho (R)
WM_RBUTTONUP	// ...
WM_RBUTTONDOWNBLCLK	// ...
WM_MOUSEACTIVATE	// Se pres. un botón del ratón estando sobre una ventana inactiva
WM_MOUSEHOVER	// Como consecuencia a una llamada a la función TrackMouseEvent.
WM_MOUSELEAVE	// Similar al anterior (ver documentación de dicha función)
WM_MOUSEMOVE	// El ratón se mueve sobre una ventana
WM_MOUSEWHEEL	// La rueda del ratón (si la hay) se ha rotado

La constante WM_KEYUP corresponde al mensaje producido por el evento de soltar (UP) una tecla del teclado (KEY). Para este mensaje, el parámetro wParam contiene el código de la tecla

presionada. Sólo los códigos correspondientes a los números (0x30 al 0x39, '0' al '9') y las letras mayúsculas (0x41 a 0x5A, 'A' al 'Z') coinciden con la tabla ASCII. Toda tecla del teclado tiene un código distinto, existiendo constantes en Windows.h para cada una de ellas. Como ejemplo, podríamos reconocer si la tecla presionada fue F1 utilizando:

```
if ( wParam == VK_F1 ) { ... }
```

El parámetro lParam se comporta como un bit-flag con información como “es una tecla correspondiente al conjunto de teclas extendidas”. Hay 15 mensajes relacionados con el teclado, de los cuales los más comúnmente procesados son:

```
WM_KEYDOWN  
WM_KEYUP  
WM_CHAR  
WM_DEADCHAR
```

Manejo de Eventos con Interfaces en Java

En Java, la estrategia de manejo de los mensajes del sistema operativo, corresponde a un patrón de diseño de software conocido como “Patrón Observador”.

Bajo este patrón, existen dos objetos: El observador y el sujeto. El sujeto es una fuente de eventos (que pueden corresponder a mensajes del sistema operativo u otros producidos internamente por el programa) susceptibles de ser observados por objetos que cumplen con las características requeridas para ser observadores de dichos eventos.

Para que el objeto observador pueda realizar su trabajo, debe “registrarse” en el sujeto, esto es, notificarle de alguna forma su interés de observar ciertos eventos suyos. De esta forma, cuando el sujeto detecta que ha ocurrido un evento, notifica este hecho a todos los objetos observadores que se registraron para dicho evento. El siguiente programa muestra la aplicación de este patrón de diseño:

```
import java.util.Vector;  
  
class Observador {  
    public void notificar(String infoDelEvento) {  
        System.out.println("Sucedió el siguiente evento: " + infoDelEvento);  
    }  
}  
  
class Sujeto {  
    private Vector listaObservadores = new Vector();  
    public void registrarObservador(Observador ob) {  
        listaObservadores.add(ob);  
    }  
    public void simularEvento(String infoDelEvento) {  
        for(int i = 0; i < listaObservadores.size(); i++) {  
            Observador ob = (Observador)listaObservadores.get(i);  
            ob.notificar(infoDelEvento);  
        }  
    }  
}  
  
class PatronObservador {  
    public static void main(String args[]) {  
        Observador ob = new Observador();  
        Sujeto suj = new Sujeto();  
        suj.registrarObservador(ob);  
        suj.simularEvento("Evento1");  
        suj.simularEvento("Evento2");  
    }  
}
```

El programa crea un objeto observador y un sujeto observable, para luego simular que dos eventos ocurren. Note que el proceso de registro consiste simplemente en agregar la referencia al objeto observador a una lista del sujeto. Note además que al ocurrir el evento simulado, lo que hace el sujeto es recorrer la lista de referencias a objetos que se registraron como observadores, llamando a un método para cada uno de estos. Ésta es la forma en que el sujeto notifica al observador, llamando a un método de este último. Finalmente note que el método “notificar” de la clase observador es el acuerdo entre ambas partes, sujeto y observador, para que la notificación sea posible, es decir, todo observador de un objeto de mi clase Sujeto debe ser un objeto de la clase Observador o bien heredar de él, de forma que se garantice que dicho método existe.

El ejemplo anterior tiene un problema: Sólo podemos hacer que los objetos de una ClaseX observen los eventos de mi clase Sujeto, si mi ClaseX hereda de Observador. Esto es indeseable si pensamos que lenguajes como Java y C# no soportan herencia múltiple y, muy probablemente, deseáramos que un objeto, cuya clase padre no puedo modificar, escuche los eventos de otro. La solución a éste es aislar los métodos que forman parte del acuerdo en una interfaz. El siguiente programa modifica el anterior de forma que se utilice una interfaz en lugar de una clase:

```
import java.util.Vector;

interface IObservador {
    void notificar(String infoDelEvento);
}

class Sujeto {
    private Vector listaObservadores = new Vector();
    public void registrarObservador(IObservador ob) {
        listaObservadores.add(ob);
    }
    public void simularEvento(String infoDelEvento) {
        for(int i = 0; i < listaObservadores.size(); i++) {
            IObservador ob = (IObservador)listaObservadores.get(i);
            ob.notificar(infoDelEvento);
        }
    }
}

class ObservadorTipo1 implements IObservador {
    public void notificar(String infoDelEvento) {
        System.out.println("ObsevadorTipo1: Sucedió el siguiente evento: "+ infoDelEvento);
    }
}

class ObservadorTipo2 implements IObservador {
    public void notificar(String infoDelEvento) {
        System.out.println("ObsevadorTipo2: Sucedió el siguiente evento: "+ infoDelEvento);
    }
}

class PatronObservador2 {
    public static void main(String args[]) {
        ObservadorTipo1 ob1 = new ObservadorTipo1();
        ObservadorTipo2 ob2 = new ObservadorTipo2();
        Sujeto suj = new Sujeto();
        suj.registrarObservador(ob1);
        suj.registrarObservador(ob2);
        suj.simularEvento("Evento1");
        suj.simularEvento("Evento2");
    }
}
```

En el ejemplo anterior se tienen dos objetos, cada uno de una clase distinta pero que implementan la interfaz IObservador, que se registran para escuchar los eventos de un tercer objeto, uno de la clase Sujeto. Note que la implementación de la clase Sujeto se basa en la

interfaz IObservador y no en una clase. De esta forma se permite que los objetos de cualquier clase que implementen la interfaz IObservador sean utilizados como observadores de un objeto de la clase Sujeto. Ésta última es la estrategia que utiliza Java para sus eventos en ventanas.

En Java, el sujeto es un objeto ventana, una instancia de cualquier clase que herede de JFrame. Esta clase se comunica con una función de ventana internamente, la que procesa un subconjunto de todos los mensajes que se pueden generar con una ventana, llamando a los métodos adecuados de los objetos observadores registrados para dicho objeto ventana. Para esto, JFrame contiene listas de observadores, como datos miembros, para los distintos grupos de mensajes: Una lista para los mensajes de manipulación de la ventana, otra para los mensajes del ratón, otra para los mensajes del teclado, etc. De esta manera, cuando ocurre un evento, el mensaje es tratado por la función de ventana de JFrame, la que a su vez llama al método adecuado de cada objeto observador registrado para dicho mensaje.

En Java, las interfaces como IObservador en nuestro ejemplo, se llaman Listeners, y existe una definida para cada grupo de mensajes. Toda clase cuyos objetos se desea que puedan escuchar un evento de una ventana, debe de implementar la interfaz Listener adecuada. Veamos cómo esto se refleja, en el caso de los mensajes del ratón y del teclado, en el siguiente código.

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

class MiVentana extends JFrame {
    public MiVentana() {
        setSize(400, 400);
        setTitle("Titulo de la Ventana");
        setVisible(true);
        addWindowListener(new MiObservadorVentana(this));
        addKeyListener(new MiObservadorTeclado());
    }
}

class MiObservadorVentana implements WindowListener {
    MiVentana refVentana;
    public MiObservadorVentana(MiVentana refVentana) {
        this.refVentana = refVentana;
    }
    public void windowActivated(WindowEvent e) {
    }
    public void windowClosed(WindowEvent e) {
    }
    public void windowClosing(WindowEvent e) {
        refVentana.dispose();
        System.exit(0);
    }
    public void windowDeactivated(WindowEvent e) {
    }
    public void windowDeiconified(WindowEvent e) {
    }
    public void windowIconified(WindowEvent e) {
    }
    public void windowOpened(WindowEvent e) {
    }
}

class MiObservadorTeclado implements KeyListener {
    public void keyTyped(KeyEvent e) {
        displayInfo(e, "KEY TYPED: ");
    }
    public void keyPressed(KeyEvent e) {
        displayInfo(e, "KEY PRESSED: ");
    }
    public void keyReleased(KeyEvent e) {
        displayInfo(e, "KEY RELEASED: ");
    }
    private void displayInfo(KeyEvent e, String s){
```

```
String charString, keyCodeString, modString, tmpString;

char c = e.getKeyChar();
int keyCode = e.getKeyCode();
int modifiers = e.getModifiers();

if (Character.isISOControl(c)) {
    charString = "key character = "
        + "(an unprintable control character)";
} else {
    charString = "key character = '"
        + c + "'";
}

keyCodeString = "key code = " + keyCode
    + " ("
    + KeyEvent.getKeyText(keyCode)
    + ")";

modString = "modifiers = " + modifiers;
tmpString = KeyEvent.getKeyModifiersText(modifiers);
if (tmpString.length() > 0) {
    modString += " (" + tmpString + ")";
} else {
    modString += " (no modifiers)";
}

System.out.println(s + "\n"
    + "    " + charString + "\n"
    + "    " + keyCodeString + "\n"
    + "    " + modString);
}

}

class EventosDeVentanas {
    public static void main(String args[]) {
        MiVentana ventana = new MiVentana();
    }
}
```

En el código anterior el sujeto observable es el objeto de clase `MiVentana` creado dentro del método `main`, y los observadores son dos: Un objeto de la clase `MiObservadorVentana`, implementando la interfaz `WindowListener`, y un objeto de la clase `MiObservadorTeclado`, implementando la interfaz `KeyListener`. Ambos observadores serán notificados de los eventos de la ventana y del teclado, respectivamente, que es capaz de detectar y/o producir el sujeto. Note además que el sujeto, conceptualmente y en la práctica, no requiere saber sobre la implementación interna de los objetos observadores, lo único que le concierne es que dichos objetos poseen los métodos adecuados para, mediante éstos, poderles notificar que ocurrió un evento del tipo para el que se registraron. Es por ello que Java utiliza interfaces para definir dicho contrato, los métodos que el objeto observador debe implementar y el objeto observable debe llamar. Note también que los métodos de registro siguen un formato común y forman parte de la interfaz que ofrece el sujeto, en este caso, los métodos `addWindowListener` y `addKeyListener`. Como es de esperarse, estos métodos reciben como parámetros referencias a objetos que implementen las interfaces respectivas.

La implementación de la interfaz de un evento particular requiere ser completa, si no lo fuera, la clase sería abstracta y no podríamos pasarle un objeto instanciado de dicha clase al método de registro respectivo. Sin embargo, es posible que no se requiera utilizar todos los métodos de la interfaz para un programa en particular, como es el caso, en el código anterior, de la clase “`MiObservadorVentana`”. Debido a esto, muchas interfaces relacionadas a eventos en Java tienen una clase que las implementa, a la que se le llama “Adaptador”. La siguiente clase es el

adaptador de la interfaz `WindowListener` (definido en el mismo paquete `java.awt.event` con dicha interfaz):

```
public abstract class WindowAdapter implements EventListener, WindowFocusListener,
WindowListener, WindowStateListener {
```

```
    // métodos de la interfaz WindowListener
    public void windowActivated(WindowEvent e) {
    }
    public void windowClosed(WindowEvent e) {
    }
    public void windowClosing(WindowEvent e) {
    }
    public void windowDeactivated(WindowEvent e) {
    }
    public void windowDeiconified(WindowEvent e) {
    }
    public void windowIconified(WindowEvent e) {
    }
    public void windowOpened(WindowEvent e) {
    }

    // métodos de las demás interfaces
    ...
}
```

Los adaptadores le dan una implementación vacía a las interfaces que implementan y son declarados como abstractos únicamente porque se espera que sirvan como clases base para otras clases que sobrescriban los métodos de las interfaces que requieran. Un ejemplo del uso de un adaptador puede verse en el primer código de ejemplo de Java, en la sección 3.1. Creación de una Ventana. En dicho código se utiliza el adaptador `WindowAdapter` como base de una clase anidada anónima.

Es importante señalar que no existe ningún impedimento para que el sujeto sea a su vez observador de otros sujetos o de sí mismo (como en el ejemplo de la sección 3.1), que un mismo observador pueda observar varios sujetos a la vez (del mismo tipo o de diferente tipo, de uno o más sujetos) y que un mismo evento sea observado por muchos observadores. Para este último caso podríamos, en el ejemplo anterior, haber registrado otros objetos para los mismos eventos (llamando más de una vez a `addWindowListener` y `addKeyListener` respectivamente) de manera que cuando dichos eventos ocurran, el sujeto, uno de la clase “`MiVentana`”, llamará en secuencia a los métodos correspondientes de todas los objetos registrados para dicho evento, en el orden en que se registraron.

Así como un objeto se registra para escuchar un evento, también se puede desregistrar. Para ésto existen los correspondientes métodos `removeXXXListener`, como son `removeWindowListener` y `removeKeyListener`.

Note que todos los métodos de una interfaz `Listener` reciben como parámetro una referencia de una clase `XXXEvent`, la que encapsula los datos del mensaje y provee de una interfaz para su fácil uso. En el caso de la interfaz `WindowListener` es `WindowEvent`, en el caso de la interfaz `KeyListener` es `KeyEvent`.

La Tabla 6 - 1 resume las clases involucradas en tres tipos de eventos comúnmente manejados en Java:

Tabla 6 - 2 Clases relacionadas con eventos en Java

Evento	Listener	Método de registro en JFrame	Adaptador	Parámetro de los métodos de la
--------	----------	------------------------------	-----------	--------------------------------

				interfaz
Ventana	WindowListener	addWindowListener	WindowAdapter	WindowEvent
Teclado	KeyListener	addKeyListener	KeyAdapter	KeyEvent
Ratón	MouseListener	addMouseListener	MouseAdapter	MouseEvent

Manejo de Eventos con Delegados en C#

Los delegados son clases especiales en .NET que manejan internamente una referencia a un método de una clase, de manera que puede llamarla directamente. Es el equivalente a un puntero a función de C o C++, pero sin permitir un acceso directo a dicho puntero o referencia. Los delegados tienen un formato especial de declaración:

```
[Modificadores] delegate <tipo de retorno> <nombre>([Lista de Parámetros.]);
```

La declaración de un delegado es muy similar a la de un método, pero con la palabra “delegate” entre los modificadores y el tipo de valor de retorno. Es importante tener en cuenta que esta declaración corresponde a un “tipo de dato”, no a una variable. Dado que esta declaración corresponde a un tipo de dato más, la declaración de variables de este tipo y su inicialización siguen las mismas reglas conocidas para las clases. El siguiente ejemplo muestra la creación de un objeto delegado y su uso.

```
using System;

class OtraClase {
    public static int Metodo3(string sMensaje) {
        Console.WriteLine("    OtraClase.Metodo3 : " + sMensaje);
        return 3;
    }
    public int Metodo4(string sMensaje) {
        Console.WriteLine("    OtraClase.Metodo4 : " + sMensaje);
        return 4;
    }
}

class Principal {

    private delegate int MiDelegado(string sMensaje);
    static event MiDelegado evento;

    private static int Metodo1(string sMensaje) {
        Console.WriteLine("    Principal.Metodo1 : " + sMensaje);
        return 1;
    }
    private int Metodo2(string sMensaje) {
        Console.WriteLine("    Principal.Metodo2 : " + sMensaje);
        return 2;
    }

    public static void Main(string[] args) {
        //////////////////////////////////////
        // Prueba con delegados

        Console.WriteLine("Prueba con delegados");
        MiDelegado delegado;

        delegado = new MiDelegado(Metodo1);
        Console.WriteLine("Llamando al delegado ...");
        Console.WriteLine("    retorno = " + delegado("mensaje1"));

        Principal refPrincipal = new Principal();
        delegado = new MiDelegado(refPrincipal.Metodo2);
    }
}
```

```
Console.WriteLine("Llamando al delegado ...");
Console.WriteLine("    retorno = " + delegado("mensaje2"));

delegado = new MiDelegado(OtraClase.Metodo3);
Console.WriteLine("Llamando al delegado ...");
Console.WriteLine("    retorno = " + delegado("mensaje3"));

OtraClase refOtraClase = new OtraClase();
delegado = new MiDelegado(refOtraClase.Metodo4);
Console.WriteLine("Llamando al delegado ...");
Console.WriteLine("    retorno = " + delegado("mensaje4"));

//////////
// Prueba con eventos

Console.WriteLine("Prueba con eventos");

evento += new MiDelegado(Metodo1);
evento += new MiDelegado(refPrincipal.Metodo2);
evento += new MiDelegado(OtraClase.Metodo3);
evento += new MiDelegado(refOtraClase.Metodo4);
Console.WriteLine("Llamando al evento ...");
Console.WriteLine("    retorno = " + evento("mensaje del evento"));
}
}
```

La salida de este programa es:

```
Prueba con delegados
Llamando al delegado ...
    Principal.Metodo1 : mensaje1
    retorno = 1
Llamando al delegado ...
    Principal.Metodo2 : mensaje2
    retorno = 2
Llamando al delegado ...
    OtraClase.Metodo3 : mensaje3
    retorno = 3
Llamando al delegado ...
    OtraClase.Metodo4 : mensaje4
    retorno = 4

Prueba con eventos
Llamando al evento ...
    Principal.Metodo1 : mensaje del evento
    Principal.Metodo2 : mensaje del evento
    OtraClase.Metodo3 : mensaje del evento
    OtraClase.Metodo4 : mensaje del evento
    retorno = 4
```

La declaración de una variable tipo delegado y su inicialización es similar a la de cualquier otra clase, excepto por el parámetro de su constructor. Note que si el método pasado como parámetro no es estático, se requiere contar con una referencia a un objeto de la clase que contiene dicho método, al que se desea llamar.

El método pasado como parámetro, al crear un objeto delegado, debe tener el mismo formato de declaración del delegado. Para el ejemplo anterior, tanto los métodos llamados, como la declaración del tipo delegado, reciben como parámetro una referencia a un objeto System.String y retornan un valor System.Int32.

Luego de inicializar una variable de tipo delegado, su uso es igual al de un puntero a función de C o C++, utilizando el nombre de la variable como si se tratara del nombre de un método.

Los objetos delegados utilizados en el código anterior, sólo nos permiten llamar a un único método a la vez. Sin embargo, es posible utilizar un objeto delegado enlazado con otros objetos delegados, de forma que se pueda llamar a más de una función. A este tipo de delegado se le

llama **MulticastDelegate**. El uso de este tipo de delegados cae fuera del alcance del presente curso.

Sin embargo, como puede verse en el código anterior, una forma simple de llamar a un grupo de métodos es utilizando la palabra clave event. El formato de declaración de una variable event es:

```
[Modificadores] event <Nombre del Delegado> <Nombre>;
```

Esta declaración sólo puede ir en el ámbito de una clase, no dentro de un método. Esto se debe a que, a pesar de parecerse a la declaración de una variable, al agregarle la palabra event a la declaración, la sentencia se expande al ser compilado el código, generándose la declaración de una variable delegado, del tipo puesto en la declaración, y dos métodos, add_XXX y remove_XXX (donde XXX es el nombre del tipo del delegado). Para el código anterior, esta expansión sería de la forma:

```
private static MiDelegado evento = null;  
private static MiDelegado add_MiDelegado(...) {}  
private static MiDelegado remove_MiDelegado(...) {}
```

Estos métodos add y remove son llamados automáticamente cuando se utilizan los operadores '+' y '-', respectivamente, con la variable evento. Debido a esto, la variable declarada con la palabra event no requiere ser inicializada.

En .NET se utilizan los conceptos de delegado y evento para manejar la respuesta a la interacción del usuario con las ventanas del programa. Para cada tipo de evento que el usuario pueda generar, existen en las clases de .NET propiedades que encapsulan variables event, así como los tipos de delegados correspondientes. El siguiente código muestra un ejemplo del uso de estos eventos y delegados:

```
using System;  
// Al siguiente espacio de nombres pertenecen:  
// Form, KeyPressEventHandler, KeyPressEventArgs,  
// MouseEventHandler, MouseEventArgs, PaintEventHandler, PaintEventArgs.  
using System.Windows.Forms;  
  
class Ventana : Form {  
    public Ventana() {  
        this.Size = new System.Drawing.Size(400, 400);  
        this.Text = "Titulo de la Ventana";  
        this.Visible = true;  
        this.MouseUp += new MouseEventHandler(this.Ventana_MouseUp);  
        this.MouseLeave += new EventHandler(this.Ventana_MouseLeave);  
        this.KeyPress += new KeyPressEventHandler(this.Ventana_KeyPress);  
    }  
    private void Ventana_MouseUp(object sender, MouseEventArgs e) {  
        MessageBox.Show("Evento MouseUp");  
    }  
    private void Ventana_MouseLeave(object sender, System.EventArgs e) {  
        MessageBox.Show("Evento MouseLeave");  
    }  
    private void Ventana_KeyPress(object sender, KeyPressEventArgs e) {  
        MessageBox.Show("Evento KeyPress");  
    }  
}  
  
class Eventos_Ventana {  
    public static void Main(string[] args) {  
        Ventana refVentana = new Ventana();  
        Application.Run(refVentana);  
    }  
}
```

En el código anterior, se hace uso de las propiedades MouseUp, MouseLeave, KeyPress y Paint, las que nos permiten acceder a datos miembros internos declarados como event para los tipos

de delegado `MouseEventHandler`, `EventHandler`, `KeyPressEventHandler` y `PaintEventHandler` respectivamente.

Es importante tener en cuenta que las variables declaradas como `event` son únicamente una forma de simplificar el trabajo con los delegados, cuando se desea tener la posibilidad de llamar a más de un método cuando un evento sucede.

Aunque enfocado en forma distinta a Java, el uso de delegados es realmente la implementación del mismo patrón de diseño, el patrón Observador. La única diferencia está en que, mientras en Java el objeto observador implementa una interfaz para que el sujeto observable le notifique de un evento llamando a los métodos de ésta, en .NET se utiliza un puntero a función encapsulado en una clase especial.

Tipos de Eventos

Los eventos con los que interactúa un programa son comúnmente los producidos como consecuencia de un mensaje del sistema operativo, en particular los relacionados con el ratón, el teclado y con el manejo de la ventana.

Un programa también puede definir sus propios eventos o simular los ya existentes como una manera de independizar el programa de las capacidades del sistema operativo subyacente (como es el caso de Java para muchas de sus clases, del paquete `Swing`, con representación visual).

Un programa también puede disparar eventos al detectar que sucesos no visuales se producen en su entorno, como por ejemplo: El arribo de un paquete de datos por red, la recepción de un mensaje enviado desde otro programa, la baja del nivel de algún recurso por debajo del límite crítico (como la memoria), etc.

Gráficos en 2D

El manejo de los dispositivos gráficos en Windows se realiza mediante la librería `GDI32` (Graphics Device Interface). Esta librería aísla las aplicaciones de las diferencias que existen entre los dispositivos gráficos con los que puede interactuar un computador.

El siguiente diagrama muestra el flujo de comunicación entre las aplicaciones en ejecución, la librería gráfica de Windows, las librerías por cada dispositivo y los dispositivos mismos.

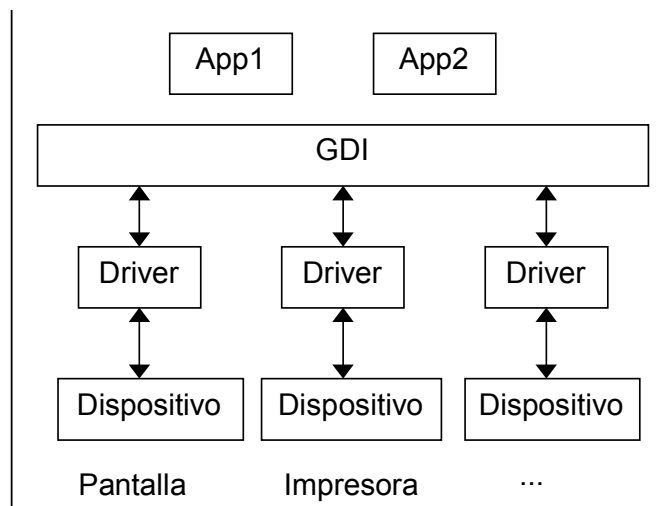


Figura 6 - 8 Flujo de comunicación entre aplicaciones y la librería GDI

Como se puede apreciar, las aplicaciones interactúan únicamente con la librería GDI. Una vez que una aplicación le indica a GDI con qué tipo de dispositivo desea interactuar, dicha interacción se realiza en forma independiente al dispositivo elegido. Las librerías por cada dispositivo se conocen como Drivers, y deben cumplir la especificación requerida por GDI para que puedan interactuar con él. De esta forma, cada fabricante de un nuevo dispositivo, que desee que éste sea utilizado desde Windows deberá proveer un Driver que sepa cómo manejar su dispositivo y que cumpla las especificaciones de GDI.

Un punto importante es ¿qué sucede cuando una aplicación le indica a GDI con qué tipo de dispositivo desea interactuar? La GDI busca si dicho dispositivo (es decir, su driver) existe, lo carga a memoria si no estuviese ya cargado, crea una entrada en la tabla de recursos para el nuevo recurso a utilizar (el dispositivo), llena dicha entrada y retorna al programa un handle al nuevo dispositivo creado. Dicha entrada en la tabla de recursos contiene:

- Información sobre el dispositivo mismo, su tipo, sus capacidades, etc.
- Información sobre su estado actual, lo que puede incluir referencias a otros recursos utilizados cuando la aplicación desea interactuar con el dispositivo.

A dicha información en conjunto se le llama Contexto del Dispositivo (Device Context), por lo que el tipo de su handle relacionado es HDC (Handle Device Context).

De lo anterior se resume que, para que una aplicación pueda interactuar con un dispositivo debe de obtener un HDC adecuado para dicho dispositivo. Al igual que con otros handles, la aplicación deberá liberar dicho HDC cuando ya no requiera trabajar más con él.

Existen 5 formas de dibujo bastante comunes (no son las únicas):

- Síncrono, donde las acciones de dibujo se realizan en cualquier lugar de la aplicación.
- Asíncrono, donde las acciones de dibujo se realizan en un lugar bien definido de la aplicación.
- Sincronizado, cuando el dibujo asíncrono se “sincroniza” con otras acciones en cualquier lugar de la aplicación.
- En Memoria, donde las acciones de dibujo se realizan en un lugar de la memoria distinta a la memoria de video.
- En Impresora, donde las acciones de dibujo se traducen en comandos enviados a una impresora.

En los tres primeros casos, los HDC que se obtendrían corresponden al área cliente de una ventana, siendo el dispositivo gráfico un monitor de computadora. En el penúltimo caso, el dispositivo gráfico es un espacio de memoria fuera de la memoria de video.

A continuación veremos cómo se realizan los distintos tipos de dibujo para los diferentes lenguajes utilizados. Es importante mantener siempre presente la idea de que, sin importar en qué lenguaje se trabaje, siempre se debe utilizar un HDC para dibujar sobre un dispositivo gráfico.

Dibujo con API de Windows

En esta sección se verá las diferentes formas de dibujo que permite la librería de Windows. Es importante tener en consideración que la implementación en Windows de las librerías de dibujo tanto en Java como C# utilizan internamente esta API para realizar su trabajo.

Funciones de Dibujo

A continuación se explica algunas de las funciones de dibujo de la librería GDI:

La función “DrawText” utiliza la fuente de letra, color de texto y color de fondo actual del DC, para dibujar un texto. Su prototipo es:

```
int DrawText(  
    HDC hDC,          // handle al DC  
    char* texto,      // texto a dibujar  
    int longitud,     // longitud del texto  
    RECT* area,       // rectángulo dentro del que se dibujará el texto  
    UINT opciones     // opciones de dibujo del texto  
);
```

Un ejemplo de uso sería:

```
RECT rc = {20, 50, 0, 0};  
DrawText( hDC, "hola", -1, &rc, DT_NOCLIP);
```

El código anterior dibujaría la cadena “hola” sobre el DC referido con el handle “hDC”. Utilizamos la opción DT_NOCLIP dado que no nos interesa restringir la salida del texto a un rectángulo específico. Por el mismo motivo sólo especificamos la posición X e Y inicial del texto en la variable “rc”. La estructura RECT se define como:

```
struct RECT { long left, top, right, bottom; };
```

La función MoveToEx establece el punto inicial de dibujo de líneas sobre un DC. A partir de dicha posición se realizará dibujos de líneas hacia otras posiciones, con funciones como LineTo. Cada nueva línea dibujada actualiza la posición actual de dibujo de líneas. Los prototipos de MoveToEx y LineTo son:

```
BOOL MoveToEx(  
    HDC hdc,          // handle al DC  
    int Xinicial,     // coordenada-x de la nueva posición  
                    // (la que se convertirá en la actual)  
    int Yinicial,     // coordenada-y de la nueva posición  
                    // (la que se convertirá en la actual)  
    POINT* PosAntigua // recibe los datos de la antigua posición actual  
);  
  
BOOL LineTo(  
    HDC hdc,          // handle al DC  
    int Xfinal,       // coordenada-x del punto final  
    int Yfinal        // coordenada-y del punto final  
);
```

Un ejemplo de uso sería:

```
MoveToEx( hDC, 10, 10, NULL );  
LineTo( hDC, 40, 10 );  
LineTo( hDC, 40, 40 );  
LineTo( hDC, 10, 10 );
```

El código anterior dibujaría un triángulo con vértices (10,10), (40,10) y (40,40). La posición actual de dibujo, para subsiguientes dibujos de líneas, quedaría en la coordenada (10,10).

Es importante señalar que todas las acciones de dibujo sobre un DC se realizan en base al sistema de coordenadas del mismo. En el caso de un DC relativo al área cliente de una ventana, el origen de su sistema de coordenadas en la esquina superior izquierda del área cliente, con el eje positivo X avanzando hacia la derecha, y el eje positivo Y avanzando hacia abajo.

Dibujo Asíncrono

Las acciones de dibujo se centralizan en el procesamiento del mensaje WM_PAINT. El siguiente código muestra un esquema típico de procesamiento de este mensaje:

```
case WM_PAINT:  
    hDC = BeginPaint(hWnd, &ps);  
    // Aquí van las acciones de dibujo  
    EndPaint(hWnd, &ps);  
    break;
```

La función BeginPaint retorna un HDC adecuado para dibujar sobre el área cliente invalidada de una ventana. La función EndPaint libera el HDC obtenido. Para entender el concepto de invalidación, imagine el siguiente caso:

- Se tiene en un momento dado dos ventanas mostradas en pantalla. La primera oculta parte de la segunda, como se muestra en la siguiente figura:

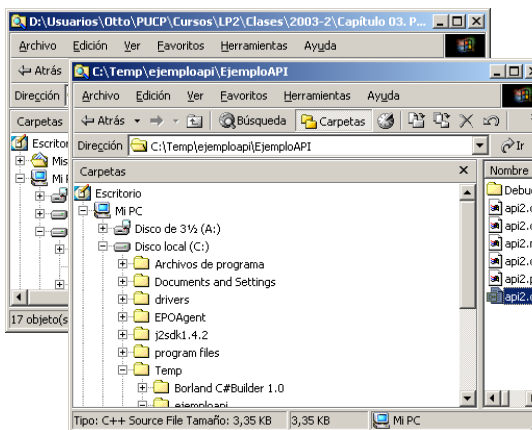


Figura 6 - 9 Dibujo asíncrono: Ventana ocultando otra ventana

- Luego se mueve la primera ventana de forma que descubre parte o toda el área oculta de la segunda ventana, como se muestra en la siguiente figura:

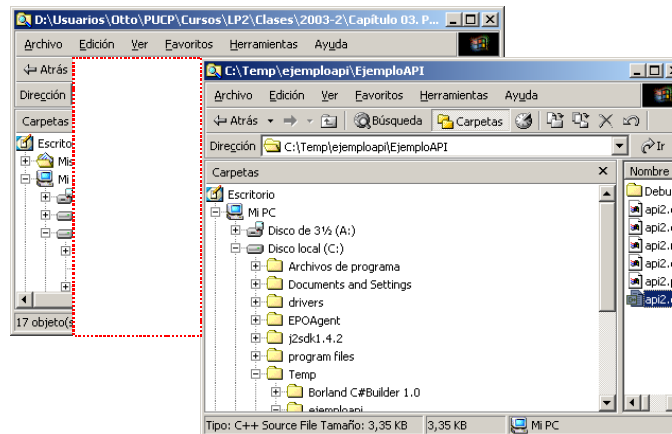


Figura 6 - 10 Dibujo asíncrono: Ventana descubriendo otra ventana

- El dibujo actual del área descubierta ya no es válido y debe de ser redibujado por el código de la aplicación correspondiente a la segunda ventana, dado que Windows no tiene forma de saber cómo se debe dibujar cada ventana de cada aplicación, sólo las

aplicaciones mismas lo saben. Sin embargo, Windows sí reconoce que esta invalidación ha ocurrido, dado que sabe dónde se encuentra cada ventana y cuál está frente a cual, por lo que genera un mensaje WM_PAINT, con la información acerca del rectángulo invalidado, y lo deposita en la cola de mensajes de la aplicación a la que le corresponde dicha ventana invalidada.

- Finalmente, cuando el bucle de procesamiento de mensajes correspondiente extraiga y mande a procesar dicho mensaje WM_PAINT, la función de ventana de la ventana invalidada repintará el área de dibujo inválido.

La estructura PAINTSTRUCT es llenada por BeginPaint con información acerca del área invalidada. Esta información podría ser utilizada por el programa para aumentar la eficiencia del código de dibujo, dado que podría repintar solamente el área invalidada y no repintar toda el área cliente. Uno de los datos miembros de dicha estructura es el mismo valor retornado por BeginPaint. La función EndPaint utiliza dicho dato para eliminar el DC.

Los mensajes WM_PAINT son generados en forma automática por el sistema operativo cuando éste sabe que el área cliente de una ventana requiere repintarse. Si al generarse un mensaje WM_PAINT para una ventana, ya existe en la cola de mensajes otro WM_PAINT para la misma ventana, se juntan ambos mensajes en uno solo para un área invalidada igual a la combinación de las áreas invalidadas de ambos mensajes.

También es posible generar un mensaje WM_PAINT manualmente y colocarlo en la cola de mensajes respectiva de forma que se repinte la ventana. La función que hace ésto es:

```
BOOL InvalidateRect(  
    HWND hWnd,          // handle de la ventana  
    CONST RECT* lpRect, // rectángulo a invalidar  
    BOOL bErase         // flag de limpiado  
);
```

El segundo parámetro es el rectángulo, dentro del área cliente, que deseamos invalidar. El tercer parámetro le sirve a la función BeginPaint. Si dicho parámetro es 1, BeginPaint pinta toda el área invalidada utilizando la brocha con la que se creó la ventana (dato miembro hbrBackground de la estructura WNDCLASS) antes de finalizar y retornar el HDC (de forma que se comience con un área de dibujo limpia). Si dicho parámetro es 0, BeginPaint no realiza este limpiado.

El siguiente código muestra el uso de esta función:

```
...  
case WM_LBUTTONDOWN:  
    iContador++;  
    InvalidateRect(hWnd, NULL, TRUE);  
    break;  
case WM_RBUTTONDOWN:  
    for(iBucle = 0; iBucle < 10; iBucle++) {  
        iContador++;  
        InvalidateRect(hWnd, NULL, TRUE);  
    }  
    break;  
case WM_PAINT:  
    hDC = BeginPaint(hWnd, &ps);  
    sprintf(szMensaje, "Contador=%d", iContador);  
    DrawText(hDC, szMensaje, -1, &rc, DT_NOCLIP);  
    EndPaint(hWnd, &ps);  
    break;  
...
```

El fragmento de código anterior pertenece a una función de ventana que utiliza InvalidateRect. Cuando se presiona con el botón izquierdo del ratón se modifica un contador de forma que el dibujo actual ya no es correcto y debe repintarse. Para el botón derecho se desea que se muestre,

como una secuencia animada, cómo se va modificando el contador. Sin embargo, esta secuencia no se muestra, y sólo se ve el valor final del contador en la ventana. Esto se debe al hecho que `InvalidateRect` “no es” una llamada directa al código en `WM_PAINT`, sino la colocación de un mensaje `WM_PAINT` en la cola de mensajes, por lo que sólo al retornar del procesamiento del mensaje actual, `WM_RBUTTONDOWN` en este caso, el bucle de procesamiento de mensajes podrá retirar el `WM_PAINT` de la cola de mensajes y procesarlo.

Como puede apreciarse, el dibujo realizado en `WM_PAINT` es asíncrono respecto a la solicitud de dibujo realizada en `WM_RBUTTONDOWN`. Este tipo de dibujo es adecuado para dibujos estáticos o de fondo, pero no para secuencias de animación.

Dibujo Síncrono

Para realizar dibujos desde cualquier otro lugar fuera del procesamiento del mensaje `WM_PAINT` se utiliza la combinación de funciones:

```
// Obtener un DC para el área cliente de la ventana referida con el handle hWnd
HDC GetDC( HWND hWnd );
// Liberar el DC obtenido con GetDC
int ReleaseDC( HWND hWnd, HDC hDC );
```

Para obtener un `HDC` relativo al área cliente de una ventana se utiliza la función `GetDC`. Una vez que se han finalizado las acciones de dibujo sobre la ventana, se debe liberar el `HDC` obtenido llamando a `ReleaseDC`.

El siguiente segmento de un programa muestra el uso de esta técnica:

```
...
case WM_LBUTTONDOWN:
    hDC = GetDC(hWnd);
    iContador++;
    sprintf(szMensaje, "Contador=%d", iContador);
    DrawText(hDC, szMensaje, -1, &rc, DT_SINGLELINE);
    ReleaseDC(hWnd, hDC);
    break;
case WM_RBUTTONDOWN:
    hDC = GetDC(hWnd);
    for(iBucle = 0; iBucle < 10; iBucle++) {
        iContador++;
        sprintf(szMensaje, "Contador=%d", iContador);
        DrawText(hDC, szMensaje, -1, &rc, DT_SINGLELINE);
        Sleep(100);
    }
    ReleaseDC(hWnd, hDC);
    break;
...
```

A diferencia del caso anterior, el `HDC` creado puede utilizarse en cualquier lugar del programa. Este `HDC` debe ser liberado cuando ya no sea requerido.

Dibujo Sincronizado

El dibujo sincronizado permite realizar modificaciones al estado del dibujo en cualquier parte del programa, concentrando el trabajo de dibujo dentro del mensaje de pintado `WM_PAINT`.

El siguiente segmento de un programa muestra el uso de esta técnica:

```
...
case WM_LBUTTONDOWN:
    iContador++;
    InvalidateRect(hWnd, NULL, TRUE);
    UpdateWindow(hWnd); // acá se fuerza el procesamiento del mensaje WM_PAINT
                        // colocado en la cola de mensajes por InvalidateRect
    break;
case WM_RBUTTONDOWN:
    for(iBucle = 0; iBucle < 10; iBucle++) {
```

```
        iContador++;
        InvalidateRect(hWnd, NULL, TRUE);
        UpdateWindow(hWnd);
        Sleep(100);    }
    break;
case WM_PAINT:
    HDC = BeginPaint(hWnd, &ps);
    sprintf(szMensaje, "Contador=%d", iContador);
    DrawText(hDC, szMensaje, -1, &rc, DT_SINGLELINE);
    EndPaint(hWnd, &ps);
    break;
...
```

Dibujo en Memoria

Cuando el dibujo se debe realizar en varios pasos, el realizarlo directamente sobre la ventana provoca que el usuario del programa vea todo el proceso de dibujo, produciéndose en algunos casos el parpadeo de la imagen. En estos casos es posible realizar la composición del dibujo en memoria, para luego pasar la imagen final a la ventana en una sola acción.

El siguiente segmento de un programa muestra el uso de esta técnica:

```
// Luego de creada la ventana

hBM_Fondo = (HBITMAP)LoadImage(hInstApp, "Fondo.bmp",
    IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE | LR_DEFAULTSIZE);
hDC_Fondo = CreateCompatibleDC(hDC);
hBM_Original = (HBITMAP)SelectObject(hDC_Fondo, hBM_Fondo);
szMensaje = "Mensaje de Texto";
TextOut(hDC_Fondo, 20, 20, szMensaje, strlen(szMensaje));

ShowWindow( hWnd, iShowCmd );
UpdateWindow( hWnd );

// Luego del bucle de procesamiento de mensajes

SelectObject(hDC_Fondo, hBM_Original);
DeleteDC(hDC_Fondo);
DeleteObject(hBM_Fondo);

// En la función de ventana

HBITMAP hBM_Fondo = 0;
HDC hDC_Fondo = 0;

LRESULT CALLBACK FuncionVentana( ... ) {
    HDC hDC; PAINTSTRUCT ps; char * szMensaje;

    switch( uMsg ) {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &ps);
        if(hBM_Fondo != 0 && hDC_Fondo != 0)
            BitBlt(hDC, 0, 0, 800, 600, hDC_Fondo, 0, 0, SRCCOPY);
        else {
            szMensaje = "Error al cargar la imagen";
            TextOut(hDC_Fondo, 20, 20, szMensaje, strlen(szMensaje));
        }
        EndPaint(hWnd, &ps);
        break;
        ...
    }
    return 0;
}
```

Luego de creada la ventana se realiza lo siguiente:

- Crear un DC en memoria en base a otro DC, típicamente, en base al DC de una ventana.
- Crear un área de dibujo en memoria, ésto es, un BITMAP.

- Seleccionar el bitmap en el DC en memoria.
- Realizar los dibujos respectivos en el DC en memoria.

Luego del bucle de mensajes, donde la ventana ya fue destruida, se realiza lo siguiente:

- Se restaura el DC en memoria seleccionándole el bitmap con el que se creó.
- Destruir el DC de memoria.
- Destruir el bitmap.

Dado que el DC en memoria y el bitmap se utilizarán en la función de ventana, es conveniente definir sus variables como globales.

Dentro del procesamiento del mensaje WM_PAINT la secuencia de pasos suele ser la siguiente:

- Se obtiene un handle a un DC para el área cliente de la ventana: BeginPaint
- Utilizando dicho handle se llaman a las funciones de dibujo del API de Windows para dibujar sobre la ventana.
- Se libera el DC: EndPaint

Dibujo en Java

Java provee, a partir de su versión 1.2, el paquete SWING que simplifica significativamente el trabajo con ventanas. Esta librería está formada en su mayoría por componentes ligeros, de forma que se obtenga la máxima portabilidad posible de las aplicaciones con ventanas hacia las diferentes plataformas que soportan Java.

El paquete SWING se basa en el paquete AWT que fue desarrollado con las primeras versiones de Java. Para los ejemplos en las siguientes secciones, se realizará dibujo en 2D sobre la clase base para ventanas JFrame de SWING.

Dibujo Asíncrono

El dibujo en una ventana requiere únicamente sobrescribir el método “paint” de la clase “JFrame”. Dentro de este método se llama a la implementación de la clase base de paint y luego se realizan acciones de dibujo utilizando la referencia al objeto Graphics recibida. La clase Graphics contiene métodos adecuados para realizar:

- Dibujo de texto.
- Dibujo de figuras geométricas con y sin relleno.
- Dibujo de imágenes.

El escribir instrucciones de dibujo dentro del método paint equivale a hacerlo dentro del “case WM_PAINT” de la función de ventana en API de Windows. En su implementación para Windows, es de esperarse que la clase Graphics maneje internamente un HDC, obtenido mediante una llamada a “BeginPaint”. El siguiente programa muestra el dibujo asíncrono.

```
class Ventana extends JFrame {
    int iContador =0;

    public Ventana() {
        . . .
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent evt) {
                if(evt.getButton() == 1){
                    iContador++;
                }
            }
        });
    }
}
```

```
        repaint();
    }
    if(evt.getButton() == 3)
        for(int iBucle = 0; iBucle < 10; iBucle++) {
            iContador++;
            repaint();
        }
    });
}
public void paint(Graphics g) {
    super.paint(g);
    g.drawString("Contador=" + iContador, 100, 100);
    System.out.println(iContador);
    g.dispose();
}
}
...

```

Dibujo Síncrono

Para el dibujo síncrono se obtiene un objeto Graphics utilizando el método getGraphics de JFrame. Es importante que, si dicho método es llamado muy seguido, se libere los recursos del objeto Graphics obtenido (por ejemplo, el HDC obtenido mediante un GetDC para la implementación en Windows de esta clase) llamando al método “dispose” (que es de suponer debería llamar a ReleaseDC). Éste es un claro ejemplo donde el usuario debe preocuparse por liberar explícitamente los recursos dado que el recolector de basura puede no hacerlo a tiempo.

El siguiente programa muestra el dibujo síncrono.

```
class Ventana extends JFrame {
    int iContador =0;

    public Ventana() {
        ...
        addMouseListener(new MouseAdapter() {
            public void mousePressed(java.awt.event.MouseEvent evt) {
                Graphics g = getGraphics();
                if(evt.getButton() == 1) {
                    iContador++;
                    g.clearRect(0, 0, 400, 400);
                    g.drawString("Contador=" + iContador, 100,100);
                }
                if(evt.getButton() == 3)
                    for(int iBucle = 0; iBucle < 10; iBucle++) {
                        iContador++;
                        g.clearRect(0, 0, 400, 400);
                        g.drawString("Contador=" + iContador, 100,100);
                        System.out.println(iContador);
                        try{ Thread.sleep(800); } catch(Exception e) {}
                    }
                g.dispose();
            }
        });
    }
}
...

```

Dibujo Sincronizado

Para sincronizar un dibujo se llama al método “update” de la clase JFrame y se concentra todo el dibujo en la sobrescritura del método “paint” de la ventana. El siguiente programa muestra el uso de “update”.

```
class Ventana extends JFrame {
    int iContador =0;

    public Ventana() {
        ...
    }
}

```

```
addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent evt) {
        Graphics g = getGraphics();
        if(evt.getButton() == 1) {
            iContador++;
            update(g);
        }
        if(evt.getButton() == 3)
            for(int iBucle = 0; iBucle < 10; iBucle++) {
                iContador++;
                update(g);
                try{ Thread.sleep(1000); } catch(Exception e) {}
            }
        g.dispose();
    }
});
}
public void paint(Graphics g) {
    super.paint(g);
    g.drawString("Contador=" + iContador, 100, 100);
    System.out.println("Dibujando " + iContador);
    g.dispose();
}
}
. . .
```

Dibujo en Memoria

Existen varias estrategias para realizar dibujo en memoria en Java, para cada cual un conjunto de clases adecuadas. Una de estas estrategias consiste en utilizar un objeto `BufferedImage`, el cual crea un espacio en la memoria sobre la cual se puede realizar un dibujo. Esta clase provee un método “`getGraphics`” que permite obtener un objeto `Graphics` adecuado para dibujar en esta memoria. Luego de compuesta la imagen en memoria, se puede utilizar el método “`drawImage`” del objeto `Graphics` en el método “`paint`” para dibujar dicha imagen en la ventana.

El siguiente programa muestra esta estrategia de dibujo.

```
class Ventana extends JFrame{
    Image imgDibujar, imgFondo;
    boolean dibujoListo;
    String mensaje = "Cargando imagen ...";

    public Ventana() {
        . . .
        Toolkit tk = getToolkit();
        imgFondo = tk.createImage("Fondo.gif");
        dibujoListo = true;
    }

    public void paint(Graphics g) {
        super.paint(g);
        if(dibujoListo){
            g.drawImage(imgFondo, 0, 0, this);
        }
        else
            g.drawString("falta dibujo", 50, 50);
        g.dispose();
    }
}
. . .
```

Dibujo en C#

En .NET las clases relacionadas con el dibujo sobre ventanas se encuentran dentro del espacio de nombres `System.Drawing`.

Dibujo Asíncrono

Realizar un dibujo síncrono sobre una ventana consiste en agregar un nuevo delegado al evento “Paint” de la clase Form. Dicho delegado deberá hacer referencia a un método que reciba como parámetro una referencia “object” y una referencia “PaintEventArgs”. Ésta última contiene las propiedades y métodos necesarios para realizar un dibujo sobre la ventana.

El siguiente programa muestra un ejemplo de este tipo de dibujo.

```
class Ventana : Form {
    int iContador =0;

    public Ventana() {
        . . .
        this.Paint += new PaintEventHandler(this.Ventana_Paint);
    }
    private void Ventana_MouseDown(object sender, MouseEventArgs e) {
        if(e.Button == MouseButtons.Left) {
            iContador++;
            Invalidate();
        }
        if(e.Button == MouseButtons.Right)
            for(int iBucle = 0; iBucle < 10; iBucle++) {
                iContador++;
                Invalidate();
            }
    }
    private void Ventana_Paint(object sender, PaintEventArgs e) {
        Graphics g = e.Graphics;
        Font f = this.Font;
        Brush b = Brushes.Black;
        Console.WriteLine(iContador);
        g.DrawString("Contador=" + iContador, f, b, 100, 100);
        g.Dispose();
    }
}
. . .
```

Dibujo Síncrono

Para el dibujo síncrono se utiliza el método CreateGraphics de la clase Form desde cualquier punto del programa. Este método retorna un objeto Graphics (podemos suponer que internamente llama a GetDC) con el cual se puede dibujar sobre la ventana. Cuando ya no se requiera utilizar este objeto, el programa debe llamar al método “Dispose” del mismo, de forma que se liberen los recursos reservados por éste en su creación (podemos suponer que libera el HDC interno que maneja mediante un ReleaseDC).

El siguiente programa muestra el uso del dibujo síncrono.

```
class Ventana : Form {
    int iContador =0;

    public Ventana() {
        . . .
    }
    private void Ventana_MouseDown(object sender, MouseEventArgs e) {
        Graphics g = CreateGraphics();
        Font f = this.Font;
        Brush b = Brushes.Black;
        if(e.Button == MouseButtons.Left) {
            iContador++;
            g.Clear(Color.LightGray);
            g.DrawString("Contador=" + iContador, f, b, 100, 100);
        }
        if(e.Button == MouseButtons.Right)
            for(int iBucle = 0; iBucle < 10; iBucle++) {
                iContador++;
            }
    }
}
```

```
        g.Clear(Color.LightGray);
        System.Console.WriteLine(iContador);
        g.DrawString("Contador=" + iContador, f, b, 100, 100);
        Thread.Sleep(800);
    }
    g.Dispose();
}
...
}
```

Dibujo Sincronizado

Para sincronizar un dibujo se llama al método “Refresh” de la clase Form y se concentra todo el dibujo en la sobrescritura del método donde se concentra el trabajo de dibujo asíncrono. El siguiente programa muestra el uso de “Refresh”. El siguiente programa muestra el uso del dibujo sincronizado.

```
class Ventana : Form {
    int iContador = 0;

    public Ventana() {
        ...
        Paint += new PaintEventHandler(this.Ventana_Paint);
    }
    private void Ventana_MouseDown(object sender, MouseEventArgs e) {
        if(e.Button == MouseButtons.Left){
            iContador++;
            Refresh();
        }
        if(e.Button == MouseButtons.Right)
            for(int iBucle = 0; iBucle < 10; iBucle++) {
                iContador++;
                Refresh();
                Thread.Sleep(100);
            }
    }
    private void Ventana_Paint(object sender, PaintEventArgs e) {
        Graphics g = e.Graphics;
        Font f = this.Font;
        Brush b = Brushes.Black;
        g.DrawString("Contador=" + iContador, f, b, 100, 100);
    }
    ...
}
```

Dibujo en Memoria

Al igual que en Java, existen muchas estrategias de dibujo en memoria. El siguiente ejemplo crea un objeto de la clase “Image” que inicialmente contiene un dibujo guardado en un archivo. Dicho objeto crea un área en memoria, inicializada con la imagen leída del archivo, a la que puede accederse mediante un objeto Graphics creado mediante el método estático “FromImage” de la misma clase Graphics. Cuando dicho objeto Graphics ya no se requiera, el programa debe liberar sus recursos llamando al método “Dispose”. El siguiente programa muestra un ejemplo de este tipo de dibujo.

```
class Ventana : Form {
    Image img;

    public Ventana() {
        this.Size = new System.Drawing.Size(400, 400);
        this.Text = "Título de la Ventana";
        this.Visible = true;
        this.Paint += new PaintEventHandler(Ventana_Paint);

        img = Image.FromFile("Fondo.bmp");
        Graphics g = Graphics.FromImage(img);
        Font f = this.Font;
        Brush b = Brushes.Black;
    }
}
```



```
50, 30);    g.DrawString("Es injusto que el coyote nunca alcance al correcaminos", f, b,
            g.Dispose();
        }
        public void Ventana_Paint(object sender, PaintEventArgs args) {
            Graphics g = args.Graphics;
            g.DrawImage(img, 0, 0);
            g.Dispose();
        }
    }
    . . .
```

Manejo de Elementos GUI

Las ventanas tienen un conjunto de elementos visuales que ocupan parte (o toda) su área cliente y cuyo comportamiento está bien estandarizado y es común a todos los programas que se ejecutan utilizando una misma librería gráfica. Algunos de estos elementos son: botones, cajas de texto, etiquetas, listas de selección, agrupadores, etc.

En esta sección veremos cómo se crean los elementos GUI más comunes, cómo se distribuyen en el área cliente y cómo se manejan los eventos que producen al interactuar el usuario con ellos.

Elementos GUI del API de Windows

En API de Windows, todos los elementos GUI son ventanas. Cada elemento se crea utilizando una clase de ventana preregistrada, y por consiguiente posee una función de ventana ya implementada en alguna de las librerías del API. El siguiente código muestra un ejemplo simple de creación de una ventana con una etiqueta, una caja de texto y un botón.

```
#include <windows.h>
#define ID_TEXTO 101
#define ID_BOTON 102

LRESULT CALLBACK FuncionVentana( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam ) {
    switch( uMsg ) {
        case WM_COMMAND:
            if( LOWORD( wParam ) == ID_BOTON ) {
                char szNombre[ 100 ];
                HWND hWndBoton = ( HWND )lParam;
                GetDlgItemText(hWnd, ID_TEXTO, szNombre, 100);
                MessageBox(hWnd, szNombre, "Hola", MB_OK );
            }
            break;
        // Aquí va el resto del switch
        ...
    }
    return 0;
}

BOOL RegistrarClaseVentana( HINSTANCE hIns ) {
    ...
}

HWND CrearInstanciaVentana( HINSTANCE hIns ) {
    ...
}

int WINAPI WinMain( HINSTANCE hIns, HINSTANCE hInsPrev, LPSTR lpCmdLine, int iShowCmd ) {
    // Creo y registro la ventana principal.
    ...

    // Creo una etiqueta, una caja de texto y un botón.
    HWND hWndLabel = CreateWindow(
        "STATIC", "Ingrese un nombre:", WS_CHILD | WS_VISIBLE, 10, 10, 150, 20,
        hWnd, NULL, hIns, NULL );
```

```
HWND hWndTextBox = CreateWindow(
    "EDIT", "", WS_CHILD | WS_VISIBLE | WS_BORDER, 170, 10, 100, 20,
    hWnd, (HMENU)ID_TEXTO, hIns, NULL );
HWND hWndButton = CreateWindow(
    "BUTTON", "OK", WS_CHILD | WS_VISIBLE, 10, 40, 100, 20,
    hWnd, (HMENU)ID_BOTON, hIns, NULL );

// Muestro la ventana.
...

// Se realiza un bucle donde se procesen los mensajes de la cola de mensajes.
MSG Mensaje;
while( GetMessage( &Mensaje, NULL, 0, 0 ) > 0 )
    if( TranslateMessage( &Mensaje ) == FALSE )
        DispatchMessage( &Mensaje );

return 0;
}
```

A los elementos GUI del API de Windows se les llama controles. Como puede observarse, los controles no son más que ventanas cuyos nombres de clase (STATIC, EDIT y BUTTON) corresponden a clases de ventana preregistradas. Dado que estas ventanas deben dibujarse dentro del área cliente de nuestra ventana principal, tenemos que asignarles la constante de estilo WS_CHILD y el octavo parámetro debe ser el identificador de esta ventana padre. El estilo WS_VISIBLE evita que tengamos que ejecutar ShowWindow para cada una de estas ventanas hijas.

El noveno parámetro de CreateWindow puede, opcionalmente, ser un número identificador que distinga dicha ventana hija de sus ventanas hermanas (hijas de la misma ventana padre). Este parámetro es aprovechado en la función de ventana de nuestra ventana principal. En dicha función se agrega una sentencia CASE para el mensaje WM_COMMAND, el cual es generado por diferentes objetos visibles cuando el usuario interactúa con ellos. En particular, cuando presionamos el botón creado, se agrega a la cola de mensajes del programa, un mensaje WM_COMMAND con los dos bytes menos significativos del parámetro wParam iguales al identificador del botón, ID_BOTON. También utilizamos el identificador de la caja de texto, ID_TEXTO, para poder obtener el texto ingresado llamando a la función GetDlgItemText.

En general, la interacción con los controles estándar del API de Windows, así como otros objetos visuales de una ventana, generan mensajes que son enviados a la ventana padre para su procesamiento. De igual forma, dicho procesamiento suele incluir el envío de nuevos mensajes a los objetos visibles, para obtener más información o para reflejar el cambio de estado del programa (internamente, GetDlgItemText envía un mensaje directamente a la función de ventana del control hijo, de manera que ésta devuelva el texto ingresado), en otras palabras, todo se realiza enviando y recibiendo mensajes. Esto tiene la ventaja de unificar la forma de trabajo con ventanas a un modelo simple de envío-recepción de mensajes, pero con la desventaja de limitar el procesamiento a una sola ventana (comunmente, solo la ventana padre). Esto tiene sentido bajo el enfoque de que, todos los elementos manejados son ventanas, por tanto es de esperarse que sólo la ventana padre esté interesada en los mensajes de sus hijas. El problema sucede cuando queremos encapsular la funcionalidad de una ventana a sí misma para ciertos trabajos, es decir, ¿cómo hacer para que una caja de texto maneje por sí mismo los mensajes que sólo le competen a él, y envíe a la ventana padre el resto?. Veremos que Java y C# solucionan, de diferente forma, estas carencias del enfoque del API de Windows.

Elementos GUI de Java

En Java, los elementos GUI se denominan componentes. El siguiente código muestra una ventana equivalente en Java al código anterior en API de Windows.

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

class VentanaDePrueba extends JFrame {
    private JTextField txt;

    public VentanaDePrueba(String titulo) {
        super(titulo);

        JLabel lbl = new JLabel("Ingrese un nombre:");
        txt = new JTextField(20);
        JButton btn = new JButton("OK");
        btn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(VentanaDePrueba.this,
                    txt.getText(), "Hola", JOptionPane.INFORMATION_MESSAGE);
            }
        });

        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(lbl);
        cp.add(txt);
        cp.add(btn);
    }
}

public class Componentes {
    public static void main(String args[]) {
        VentanaDePrueba ventana = new VentanaDePrueba(
            "Ventana de prueba de componentes");
        ventana.setSize(400, 300);
        ventana.setVisible(true);
        ventana.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}
```

En el programa anterior, `JTextField`, `JLabel` y `JButton` son componentes que representan una caja de texto, una etiqueta y un botón respectivamente, al igual que las clases de ventana `EDIT`, `STATIC` y `BUTTON` del API de Windows.

Los elementos GUI de Java se denominan **componentes**. Como puede observarse, los componentes en Java son clases que se agregan a una ventana para ser visualizados. Es importante en este punto hacer una distinción entre lo que son internamente estos componentes, contra lo que son los controles del API de Windows. Mientras que los controles del API de Windows son ventanas, los componentes de Java (a partir de la versión 1.2 del JDK, denominada Java 2) se dividen en dos categorías:

- Los **componentes pesados**. Su comportamiento está supeditado a las capacidades de la plataforma subyacente, en este caso particular, Windows. Estos componentes son `JFrame`, `JDialog` y `JApplet`. Estos componentes crean ventanas de Windows (o utilizan directamente una ya creada) y las administran internamente, ofreciendo al programador una interfaz más amigable. En otras palabras, por ejemplo, cuando realizamos click sobre una ventana creada con un objeto que deriva de `JFrame`, el evento que se genera es un mensaje `WM_LBUTTONDOWN`, el cual es sacado de la cola de mensajes y enviado a una función de ventana que llama a un método de nuestro objeto `JFrame`, el cual se encarga de llamar al método respectivo de `MouseListener` para todos los objetos registrados con un llamado a `addMouseListener`. Además de lo anterior, el dibujo de la ventana, el efecto de maximizado y minimizado, la capacidad de redimensionar la

ventana y todos los efectos visuales posibles, son gestionados por las funciones del API de Windows, así como las capacidades ofrecidas por la propia clase JFrame, por ejemplo, al llamar al método setVisible se estaría llamando internamente a ShowWindow del API de Windows. Debido a esta dependencia, estos componentes son denominados pesados.

- Los **componentes ligeros**. Su comportamiento está supeditado a las capacidades ofrecidas por un componente pesado, del cual heredan o dentro del cual se dibujan, y no de la plataforma subyacente, en este caso particular, Windows. Si bien los mensajes producidos por la interacción del usuario siguen siendo producidos por el sistema operativo, el manejo de éstos (en forma de eventos), el dibujo de los componentes y de sus efectos visuales relacionados, están codificados completamente en Java. Estos componentes definen además sus propios eventos. Debido a esta independencia, estos componentes son denominados ligeros. Ejemplos de estos componentes son JButton, JLabel y JTextField. Los componentes ligeros se dibujan sobre el área cliente de componentes pesados y simulan el “Look And Feel” correspondiente, es decir, no crean una ventana child. Este tipo de ventanas, child, se verá mas adelante (sección “Tipos de Ventana”)

Por otro lado, la clase JFrame no administra directamente el área cliente de su ventana, sino que delega dicho trabajo a un objeto Contenedor, derivado de la clase Container. Un Contenedor es básicamente un Componente Java con la capacidad adicional de poder mostrar otros Componentes dentro de su área de dibujo. Es por esto que es necesario obtener una referencia a dicho contenedor de la ventana, llamando al método getContentPane, dado que es a dicho contenedor al que deberemos agregarle los componentes que deseamos visualizar.

Los componentes, al igual que una ventana, pueden generar mensajes como respuesta a la interacción del usuario con ellos. En el código anterior, un botón creado con la clase JButton genera el evento Action cuando el usuario, con el ratón o el teclado, presiona dicho botón. Para procesar dicho evento, definimos una clase inner anónima que implementa la interfaz ActionListener, instanciando dicha clase y pasándole la referencia a esta instancia al método addActionListener del botón.

Es interesante notar el hecho de que una clase inner anónima puede ser creada realmente a partir de una clase o de una interfaz, siempre que en éste último caso se implementen todos sus métodos, que para este caso, es uno sólo, actionPerformed. De igual manera, es interesante notar que se ha escogido configurar el objeto observador del evento de cerrado de la ventana desde el método main, no desde el constructor de la ventana. Ambos enfoques son equivalentes.

Dentro del método actionPerformed se hace uso del dato miembro “txt” de tipo JTextField para poder mostrar el mensaje respectivo mediante el método estático showMessageDialog de la clase JOptionPane. Note que el primer parámetro de este método debe ser una referencia a la ventana padre de la ventana que se mostrará, y que dicho parámetro se pasa utilizando la expresión “Ventana.this”. Esto se debe a que si pasáramos únicamente this, nos estaríamos refiriendo al objeto anónimo que implementa la interfaz ActionListener.

Finalmente para este código, note que antes de agregar los componentes al content pane, se llama al método setLayout. Esto permite determinar la forma en que los componentes son distribuidos dentro del área cliente del contenedor. El manejo del diseño (layout) de una ventana se tratará más adelante.

Sumarizando las diferencias entre API de Windows y Java, mientras los objetos visuales comunes llamados controles, preimplementados en dicha librería, son básicamente ventanas y

las acciones son manejadas mediante mensajes enviados por estos controles a sus ventanas padres, en Java se trabajan con clases llamadas componentes, las cuales se agregan al contenedor `ContentPane` de la ventana, y sus acciones son manejadas mediante interfaces. Mientras que los controles del API de Windows tienen una posición fija respecto a la esquina superior izquierda del área cliente de su ventana padre y sus dimensiones son fijas, los componentes de Java no tienen una posición ni dimensión fija, y esto es manejado mediante objetos `Layout` que asisten en el diseño de la ventana. Mientras que en API de Windows los mensajes sólo pueden ser recibidos por ventanas y sólo una ventana, generalmente la ventana padre, es la que recibe los mensajes de los controles, en Java cualquier objeto de cualquier clase que implemente la interfaz correspondiente a un evento puede recibir la notificación del mismo.

Elementos GUI de C#

Los elementos GUI en C# se denominan controles. El siguiente código muestra una ventana equivalente en C# al código anterior en API de Windows.

```
using System;
using System.Windows.Forms;
using System.Drawing;

class Ventana : Form {
    private TextBox txt;

    public Ventana() {
        this.Text = "Prueba de Controles";
        this.Size = new Size(300, 300);

        Label lbl = new Label();
        lbl.AutoSize = true;
        lbl.Text = "Ingrese un nombre:";
        lbl.Location = new Point(10, 10);

        txt = new TextBox();
        txt.Size = new Size(100, lbl.Height);
        txt.Location = new Point(10 + lbl.Width + 10, 10);

        Button btn = new Button();
        btn.Text = "OK";
        btn.Size = new Size(100, lbl.Height + 10);
        btn.Location = new Point(10, 10 + lbl.Height + 10);
        btn.Click += new EventHandler(Btn_Click);

        this.Controls.Add(lbl);
        this.Controls.Add(txt);
        this.Controls.Add(btn);
    }

    private void Btn_Click(object sender, EventArgs args) {
        MessageBox.Show(txt.Text, "Hola");
    }
}

class Principal {
    public static void Main(string[] args) {
        Application.Run(new Ventana());
    }
}
```

A diferencia de Java, dichos controles sí se crean en base a ventanas de Windows y su posición en el área cliente de la ventana padre sí se determina explícitamente. Al igual que Java, existe un objeto que administra la colección de controles dibujados en una ventana: `Controls`. Al igual que Java, cualquier objeto puede ser notificado de un evento, utilizando un objeto delegado del tipo del evento y agregándolo al evento correspondiente del componente.

Manejo Asistido del Diseño

El diseño o layout de una ventana es la distribución del espacio de su área cliente entre los elementos GUI que contiene y que componen la interfaz que ofrece al usuario. El API de Windows no ofrece herramientas para asistir al programa en el manejo del diseño. Java y C# sí lo ofrecen.

En Java todo contenedor mantiene una referencia a un objeto que implemente la interfaz `LayoutManager` y que se encarga de determinar la posición y dimensiones de todos los componentes agregados al `ContentPane` del contenedor. Esto le quita la tarea al programador de especificar por código estos datos por cada componente. Algunos de los tipos de Layout predefinidos son:

- **BorderLayout:** El área del content pane se divide en 5 regiones (norte, sur, este, oeste y centro) colocando un componente agregado en cada región, por lo que sólo se permite mostrar hasta 5 componentes a la vez, independientemente de que se agreguen más.
- **FlowLayout:** Los componentes son colocados en líneas, uno después del otro, como si cada componente fuese una palabra, continuando con la siguiente línea cuando no queda espacio en la línea actual para visualizar completamente dicho componente. Las dimensiones de cada componente son obtenidas de los valores por defecto que cada uno tiene o de las especificadas por el programa, el layout no modifica estas dimensiones.
- **GridLayout:** El área del ContentPane se divide en cuadrícula o grilla, donde cada celda tiene las mismas dimensiones. Los componentes son colocados dentro de estas celdas, modificándoles sus dimensiones para que la ocupen completamente.

El siguiente código muestra el uso de estos layouts en una ventana que utiliza un tipo u otro según un parámetro pasado en su constructor.

```
public Ventana(String Nombre) {
    Container cp = getContentPane();
    if(Nombre.equals("BorderLayout")) {
        cp.setLayout(new BorderLayout());
        cp.add(new JButton("CENTER"), BorderLayout.CENTER);
        cp.add(new JButton("EAST"), BorderLayout.EAST);
        cp.add(new JButton("WEST"), BorderLayout.WEST);
        cp.add(new JButton("NORTH"), BorderLayout.NORTH);
        cp.add(new JButton("SOUTH"), BorderLayout.SOUTH);
    } else if(Nombre.equals("FlowLayout")) {
        cp.setLayout(new FlowLayout(FlowLayout.CENTER));
        for(int i = 0; i < 10; i++)
            cp.add(new JButton("Boton-" + i));
    } else if(Nombre.equals("GridLayout")) {
        cp.setLayout(new GridLayout(3, 2));
        for(int i = 0; i < 10; i++)
            cp.add(new JButton("Boton-" + i));
    }
}
```

En C# el manejo del diseño se realiza mediante anclas (anchors) y muelles (docks). Todo control posee las siguientes propiedades:

- **Anchor:** Determina a qué borde del contenedor se anclará el control. Por ejemplo, si el control se coloca inicialmente a una distancia de 100 píxeles del borde inferior de su ventana, al redimensionarse el anchor modificará automáticamente la posición del control de forma que conserve dicha distancia.

- **Dock:** Determina a qué borde del contenedor se adosará el control. Por ejemplo, si el control se adosa al borde izquierdo de su ventana, su ancho inicial se conserva pero su alto se modifica de forma que coincida con el alto del área cliente de su ventana. Su posición también se modifica de forma que su esquina superior izquierda coincida con la del área cliente de su ventana.
- **DockPadding:** Se establece en el contenedor, por ejemplo, una clase que hereda de Form. Determina la distancia a la que los componentes, adosados a sus bordes, estarán de los mismos.

El siguiente código muestra el uso de estas propiedades sobre un botón que es agregado a una ventana.

```
Boton = new Button();
Boton.Text = "Boton1";
Boton.Dock = DockStyle.Top;
Controls.Add( Boton );

Boton = new Button();
Boton.Text = "Boton2";
Boton.Location = new System.Drawing.Point(100, 100);
Boton.Size = new System.Drawing.Size(200, 50);
Boton.Anchor = AnchorStyles.Bottom | AnchorStyles.Right;
Controls.Add( Boton );
```

Tipos de Ventana

En un sistema gráfico con ventanas, dichas ventanas pueden estar relacionadas. Estas relaciones determinan los tipos de ventanas que se pueden crear.

En Windows, existen dos tipos de relaciones entre ventanas:

- La **relación de pertenencia**. Cuando dos ventanas tienen esta relación, una ventana (llamada owned) le pertenece a la otra ventana (llamada owner), lo que significa que:
 - ⇒ La ventana owned siempre se dibujará sobre su ventana owner. A la ubicación de una ventana con respecto a otra en un eje imaginario Z que sale de la pantalla del computador, se le conoce como orden-z.
 - ⇒ La ventana owned es minimizada y restaurada cuando su ventana owner es minimizada y restaurada.
 - ⇒ La ventana owned es destruida cuando su ventana owner es destruida.
- La **relación padre-hijo**. Cuando dos ventanas tienen esta relación, una ventana (llamada hija) se dibujará dentro del área cliente de otra (llamada padre).

La relación de pertenencia se establece con el octavo parámetro de la función CreateWindow, hWndParent. La relación padre-hijo se establece con el tercer parámetro de la función CreateWindow, escogiendo como bit-flag WS_CHILD o WS_POPUP.

Una ventana popup tiene como área de dibujo el escritorio de Windows (Windows Desktop) y puede tener un botón relacionado en la barra de tareas (Windows TaskBar). Como contraparte, una ventana Child tiene como área de dibujo el área cliente de otra ventana, la que puede ser de tipo Popup o Child, y no puede tener un botón relacionado en la barra de tareas.

Las ventanas popup pueden o no tener una ventana owner. Cuando no la tienen se les llama OVERLAPPED. Un ejemplo de una ventana OVERLAPPED es la ventana principal de todo programa con ventanas de Windows. Las ventanas child siempre tienen una ventana owner.

En resumen, las ventanas popup pueden ser owner o owned, mientras que las ventanas child son siempre owned.

Una ventana que contiene una o más ventanas child del mismo tipo, cada una con su propia barra de título y botones de sistema, se le conoce como ventana **MDI** (Multiple Document Interface), donde cada ventana hija suele ser utilizada para manipular un documento. Ejemplos de estas ventanas son los programas editores de texto como Word. Las ventanas no-MDI son conocidas como ventanas SDI (Single Document Interface).

A las ventanas cuyos elementos permiten mostrar e ingresar información, éste es, establecer un diálogo con el usuario se les conoce como Cajas De Dialogo. Las cajas de diálogo no son estrictamente un tipo de ventana, su tipo real es Popup, más bien su concepto corresponde a “una forma de manejo” de una ventana. Existen dos formas de mostrar una caja de diálogo: Modal y amodalmente. Una caja de diálogo modal “detiene”, por así decirlo, la ejecución del código desde donde se le muestra, una caja de diálogo amodal no. Por ello las cajas de diálogo modales son adecuadas cuando, dentro de un bloque de instrucciones, se requiere pedir al usuario que ingrese alguna información necesaria para seguir con la ejecución del algoritmo implementado por el bloque. Un ejemplo son las ventanas mostradas por los programas al momento de imprimir. En estos casos “no es conveniente” que el usuario del programa pueda interactuar con la ventana principal de forma que modifique los datos que se imprimirán mientras se están imprimiendo, por lo que resulta imprescindible que el procesamiento de los eventos de la ventana principal sea bloqueado. Las cajas de diálogo amodales son adecuadas para mostrar e ingresar información mientras se sigue interactuando con otra ventana, comunmente la ventana principal del programa. Un ejemplo son las barras de herramientas de algunos programas gráficos como CorelDraw, el editor ortográfico de Word, etc.

El API de Windows implementa un conjunto de cajas de diálogo para acciones comunes como seleccionar un color, abrir un archivo, imprimir, etc. A estas cajas de diálogo se le conoce como Cajas de Diálogo Comunes.

En las siguientes secciones se detallará las capacidades del API de Windows manejado desde C/C++, de Java y de la plataforma .NET programada desde C#, para crear los diferentes tipos de ventanas.

Ventanas con API de Windows

Para crear una ventana owner se utiliza el estilo WS_POPUP (o algún estilo que lo incluya, como WS_OVERLAPPED o WS_OVERLAPPEDWINDOW) y se pasa NULL como el manejador de su ventana owner, lo que equivale a decir que no tiene ventana owner. El siguiente código crea una ventana owner:

```
hWndPopupOwner = CreateWindow(  
    "ClaseVentanaPopup",  
    "Título de la Ventana Owner",  
    WS_POPUP | WS_CAPTION,  
    100, 100, 200, 200,  
    NULL, // no tiene ventana owner  
    NULL, hIns, NULL  
);
```

Para crear una ventana owned se utiliza el estilo WS_POPUP y se pasa un manejador válido de su ventana owner. El siguiente código crea una ventana owned:

```
hWndPopupOwned = CreateWindow(  
    "ClaseVentanaPopup",  
    "Título de la Ventana Popup Owned",  
    WS_POPUP | WS_CAPTION,  
    100, 100, 200, 200,
```



```
hWndPopupOwner, // ventana owner
NULL, hIns, NULL
);
```

Para crear una ventana child se utiliza el estilo `WS_CHILD` y se pasa un manejador válido de su ventana owner. El siguiente código crea una ventana owned:

```
hWndChild = CreateWindow(
    "ClaseVentanaHija",
    "Título de la Ventana Hija",
    WS_CHILD | WS_BORDER,
    100, 100, 200, 200,
    hWnd, // debe tener una ventana owner
    NULL, hIns, NULL
);
```

La creación y manejo de cajas de diálogo y ventanas MDI con API de Windows va más allá de los alcances del presente documento.

Ventanas en Java

En Java cada nueva herencia de las clases base para la creación de ventanas (`JFrame` y `JDialog`) determina una nueva clase de ventana. Un programa puede definir y crear una o más ventanas, de igual o distinto tipo. Sin embargo, al crear una nueva ventana no se establece una relación de parentesco entre ellas, todas se comportan como `popup owner`.

El siguiente código muestra la creación de una ventana popup en Java desde la ventana principal del programa.

```
class VentanaPopup extends JFrame { ... }
class VentanaPrincipal extends JFrame {
    public VentanaPrincipal() {
        JButton boton = new JButton("Crear Ventana");
        boton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                VentanaPopup vp = new VentanaPopup();
            }
        }); ...
    } ...
}
```

Java soporta la creación de aplicaciones MDI. La ventana MDI es llamada “backing window” y consiste en una ventana popup con un “Content Pane” del tipo “`JDesktopPane`”. Las “pseudo-ventanas child” son implementadas con la clase “`JInternalFrame`”, la que hereda de “`JComponent`” por lo que, como puede deducirse, no son realmente ventanas. El siguiente código muestra un ejemplo de este uso:

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

class Ventanas1 extends JFrame {
    public Ventanas1() {
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                dispose();
                System.exit(0);
            }
        });

        JDesktopPane desktop = new JDesktopPane();
        setContentPane(desktop);

        JInternalFrame child = new JInternalFrame();
        child.setSize(100, 100);
        child.setTitle("Ventana hija");
        child.setVisible(true);
    }
}
```

```
        child.setResizable(true);
        child.setClosable(true);
        child.setMaximizable(true);
        child.setIconifiable(true);
        desktop.add(child);
    }

    public static void main(String args[]) {
        System.out.println("Starting Ventanas1...");
        Ventanas1 mainFrame = new Ventanas1();
        mainFrame.setSize(400, 400);
        mainFrame.setTitle("Ventanas1");
        mainFrame.setVisible(true);
    }
}
```

Las cajas de diálogo en Java se crean mediante clases que heredan de `JDialog`. El siguiente código muestra el uso de esta clase:

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

class CajaDeDialogo extends JDialog {
    JTextField texto;
    public CajaDeDialogo(JFrame padre, boolean EsModal) {
        super(padre, EsModal);
        setSize(300,100);

        texto = new JTextField(20);
        JButton boton = new JButton("OK");
        boton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                setVisible(false);
            }
        });

        Container cp = getContentPane();
        cp.setLayout(new GridLayout(3,1));
        cp.add(new JLabel("Ingrese un texto"));
        cp.add(texto);
        cp.add(boton);
    }
    public String ObtenerResultado() {
        return texto.getText();
    }
}

class Ventanas2 extends JFrame {
    JLabel etiqueta;

    public Ventanas2() {
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                dispose();
                System.exit(0);
            }
        });

        etiqueta = new JLabel("Resultado = ...");
        JButton boton1 = new JButton("Mostrar como modal");
        boton1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                CajaDeDialogo dialogo = new CajaDeDialogo(Ventanas2.this,
true);
                dialogo.setVisible(true);
                etiqueta.setText("Resultado = " +
dialogo.ObtenerResultado());
            }
        });
        JButton boton2 = new JButton("Mostrar como amodal");
        boton2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {

```

```
        CajaDeDialogo dialogo = new CajaDeDialogo(Ventanas2.this,
false);
        dialogo.setVisible(true);
        etiqueta.setText("Resultado = " +
dialogo.ObtenerResultado());
    }
    });

    Container cp = getContentPane();
    cp.setLayout(new GridLayout(3,1));
    cp.add(boton1);
    cp.add(boton2);
    cp.add(etiqueta);
}

public static void main(String args[]) {
    System.out.println("Starting Ventanas2...");
    Ventanas2 mainFrame = new Ventanas2();
    mainFrame.setSize(300, 200);
    mainFrame.setTitle("Ventanas2");
    mainFrame.setVisible(true);
}
}
```

El programa anterior muestra la diferencia entre utilizar una caja de diálogo modal y una amodal. También muestra una forma de pasar datos desde la caja de diálogo y la ventana que la crea.

Las cajas de diálogo de Java también son llamadas “ventanas secundarias”, mientras que las pseudo-ventanas hijas creadas con `JInternalFrame` son llamadas “ventanas primarias”.

Adicionalmente Java provee la clase `JOptionPane`, la que permite crear cajas de diálogo con funcionalidad común, como por ejemplo, cajas de diálogo con un texto como mensaje y botones YES, NO y CANCEL.

Ventanas en C#

Al igual que en Java, cada nueva herencia de las clases base para la creación de ventanas, `Form`, determina una nueva clase de ventana. Un programa puede definir y crear una o más ventanas, de igual o distinto tipo. A diferencia de Java, se pueden crear ventanas `popup` `owner` y `owned`.

El siguiente código muestra la creación de dos ventanas `popup`, una `owner` y la otra `owned`.

```
class VentanaPopup : Form {
    public VentanaPopup( Form OwnerForm ) {
        ...
        this.Owner = OwnerForm;
        this.Visible = true;
    }
}
class VentanaPrincipal : Form {
    public VentanaPrincipal() {
        ...
        VentanaPopup vp1 = new VentanaPopup( null );
        VentanaPopup vp2 = new VentanaPopup( this );
    }
    ...
}
```

C# maneja ventanas `child` sólo como ventanas hijas de una ventanas MDI. La ventana MDI consiste en una ventana con la propiedad `IsMdiContainer` puesta en “true”. Para que una ventana sea `child` de otra, se establece su propiedad `MdiParent` con la referencia de una ventana MDI. El siguiente código muestra un ejemplo de este uso:

```
using System;
using System.Windows.Forms;

class VentanaHija : Form {
```

```
public VentanaHija(Form padre) {
    this.SuspendLayout();
    this.Text = "Ventana Hija";
    this.Location = new System.Drawing.Point(10,10);
    this.Size = new System.Drawing.Size(100, 100);
    this.MdiParent = padre;
    this.Visible = true;
    this.ResumeLayout(false);
}
}

class VentanaPrincipal : Form
{
    public VentanaPrincipal()
    {
        InitializeComponent();
        this.IsMdiContainer = true;
        VentanaHija hija = new VentanaHija(this);
    }

    void InitializeComponent()
    {
        this.SuspendLayout();
        this.Name = "MainForm";
        this.Text = "Esta es la Ventana Principal";
        this.Size = new System.Drawing.Size(300, 300);
        this.ResumeLayout(false);
    }

    public static void Main(string[] args)
    {
        Application.Run(new VentanaPrincipal());
    }
}
```

Las cajas de diálogo en C# son ventanas con la propiedad `FormBorderStyle` puesta a `FixedDialog`. El siguiente código muestra el uso de una caja de diálogo.

```
using System;
using System.Windows.Forms;

class CajaDeDialogo : Form
{
    private TextBox texto;
    public CajaDeDialogo() {
        this.Text = "Caja de Dialogo";
        this.FormBorderStyle = FormBorderStyle.FixedDialog;

        Label etiqueta = new Label();
        etiqueta.Text = "Ingrese un texto";
        etiqueta.Location = new System.Drawing.Point(24, 16);

        texto = new TextBox();
        texto.Size = new System.Drawing.Size(128, 32);
        texto.Location = new System.Drawing.Point(24, 64);

        Button boton = new Button();
        boton.Text = "OK";
        boton.Size = new System.Drawing.Size(128, 32);
        boton.Location = new System.Drawing.Point(24, 112);
        boton.Click += new System.EventHandler(this.buttonClick);

        Controls.Add(etiqueta);
        Controls.Add(texto);
        Controls.Add(boton);
    }
    void buttonClick(object sender, System.EventArgs e)
    {
        //Visible = false;
        Close();
    }
    public string ObtenerResultado() {
        return texto.Text;
    }
}
```

```
}  
}  
  
class VentanaPrincipal : Form  
{  
    private System.Windows.Forms.Label label;  
    private System.Windows.Forms.Button button2;  
    private System.Windows.Forms.Button button;  
    public VentanaPrincipal()  
    {  
        InitializeComponent();  
    }  
  
    void buttonClick(object sender, System.EventArgs e)  
    {  
        // caso de una caja de dialogo modal  
        CajaDeDialogo dialogo = new CajaDeDialogo();  
        dialogo.ShowDialog(this);  
        // dado que la ejecución de la instrucción anterior se detiene hasta que la  
        // caja de diálogo se cierre, es posible recuperar el valor ingresado  
        label.Text = "Resultado = " + dialogo.ObtenerResultado();  
    }  
  
    void button2Click(object sender, System.EventArgs e)  
    {  
        // caso de una caja de dialogo amodal  
        CajaDeDialogo dialogo = new CajaDeDialogo();  
        dialogo.Show();  
        // dado que la ejecución de la instrucción anterior "NO" se detiene hasta que  
        // la caja de diálogo se cierre, el valor recuperado sera el valor que  
        // inicialmente tiene la caja de texto de dicha caja de dialogo al crearse,  
        // esto es, una cadena vacia  
        label.Text = "Resultado = " + dialogo.ObtenerResultado();  
    }  
  
    void InitializeComponent() {  
        this.button = new System.Windows.Forms.Button();  
        this.button2 = new System.Windows.Forms.Button();  
        this.label = new System.Windows.Forms.Label();  
        this.SuspendLayout();  
        //  
        // Primer botón  
        //  
        this.button.Location = new System.Drawing.Point(24, 16);  
        this.button.Name = "button";  
        this.button.Size = new System.Drawing.Size(128, 32);  
        this.button.TabIndex = 0;  
        this.button.Text = "Mostrar Como Modal";  
        this.button.Click += new System.EventHandler(this.buttonClick);  
        //  
        // Segundo botón  
        //  
        this.button2.Location = new System.Drawing.Point(24, 64);  
        this.button2.Name = "button2";  
        this.button2.Size = new System.Drawing.Size(128, 32);  
        this.button2.TabIndex = 1;  
        this.button2.Text = "Mostrar Como Amodal";  
        this.button2.Click += new System.EventHandler(this.button2Click);  
        //  
        // Etiqueta  
        //  
        this.label.Location = new System.Drawing.Point(24, 112);  
        this.label.Name = "label";  
        this.label.Size = new System.Drawing.Size(128, 24);  
        this.label.TabIndex = 2;  
        this.label.Text = "Resultado = ...";  
        //  
        // Agrego los controles  
        //  
        this.ClientSize = new System.Drawing.Size(248, 165);  
        this.Controls.AddRange(new System.Windows.Forms.Control[] {  
            this.label,  
            this.button2,  
            this.button  
        });  
    }  
}
```

```
        this.button));  
        this.Text = "Prueba con Cajas de Diálogo";  
        this.ResumeLayout(false);  
    }  
  
    public static void Main(string[] args)  
    {  
        Application.Run(new VentanaPrincipal());  
    }  
}
```

El programa anterior muestra la diferencia entre utilizar una caja de diálogo modal, con `ShowDialog`, y una amodal, con `Show`. También muestra una forma de pasar datos desde la caja de diálogo y la ventana que la crea.

Dado que es común validar la forma en que fue respondida una caja de diálogo modal, se provee la propiedad `DialogResult`, cuyo tipo es el enumerado `DialogResult` con los siguientes valores: `Abort`, `Cancel`, `Ignore`, `No`, `None`, `OK`, `Retry`, `Yes`. Esta propiedad se establece automáticamente en algunos casos (cuando se cierra la ventana, se establece a `Cancel`) o manualmente desde eventos programados.

Notas sobre Localización de Archivos

Los programas en ejecución (llamados procesos) tienen siempre un directorio de trabajo. Este directorio sirve para poder ubicar, de forma relativa, otros archivos (por ejemplo, para abrir dichos archivos). De esta forma un proceso no requiere utilizar siempre el directorio absoluto para acceder a archivos que se encuentran en su mismo directorio de trabajo o en algún otro directorio cercano a éste, como sus subdirectorios.

Por defecto, un proceso obtiene su directorio de trabajo heredándolo de su proceso padre (el que lo arrancó), a menos que este último indique explícitamente otro directorio. Por ejemplo, la siguiente línea de comando corresponde a una ventana de comandos (o shell) desde donde se arranca un programa `Abc`:

```
c:\prueba> c:\temp\Abc.exe
```

Note que el programa se encuentra en el directorio “`c:\temp`” mientras que el directorio de trabajo del shell es “`c:\prueba`”. Luego, el nuevo proceso `Abc` creado tendrá como directorio de trabajo “`c:\prueba`”, pues lo hereda del shell. Si se desea arrancar un programa con un directorio de trabajo distinto al del shell, se puede utilizar el comando `start`:

```
c:\prueba> start /Dd:\otroDirectorio c:\temp\Abc.exe
```

Es fácil ver cual es el directorio de trabajo actual del shell (en Windows lo indica el mismo prompt, en Linux se puede consultar con un comando, por ejemplo `pwd`) y cambiarlo (utilizando un comando como `cd`). De igual forma, utilizando las llamadas a las funciones adecuadas del API del sistema operativo, cualquier proceso puede modificar su directorio de trabajo durante su ejecución.

Para programas diferentes a los shells, desde donde también es posible arrancar otros programas, el directorio de trabajo actual puede no ser claro, por lo que heredarlo puede no ser lo que el usuario espera. Por ejemplo, al arrancar un programa desde el explorador de Windows haciendo un doble-click sobre el nombre del archivo ejecutable, el usuario espera que se inicie dicho programa teniendo como directorio de trabajo inicial el mismo directorio donde se encuentra el archivo ejecutable, independientemente de cual sea actualmente el directorio de trabajo del explorador de Windows. Este comportamiento puede modificarse creando accesos

directos a dichos programas ejecutables y configurándolos para especificar un directorio distinto como directorio de trabajo inicial.

No todos los archivos a los que un proceso requiere acceder se ubican utilizando el directorio de trabajo como referencia. Por ejemplo, en el caso de las librerías, se suelen utilizar variables de entorno para definir conjuntos de directorios de búsqueda. Por ejemplo, para las DLL nativas de Windows, se utiliza la variable de entorno PATH, mientras el compilador e intérprete de Java utilizan la variable de entorno CLASSPATH.

Programación Concurrente

El objetivo de este capítulo es dar una base teórica y práctica al lector sobre el manejo de la concurrencia en un programa así como sus ventajas y desventajas.

Procesos e Hilos

Un **proceso** es todo aquello que el computador necesita para ejecutar una aplicación. Técnicamente, un proceso es una colección de:

- Espacio de memoria virtual.
- Código.
- Datos (como las variables globales y la memoria del montón).
- Recursos del sistema (como los descriptores de archivos y los manejadores o *handles*).

Un **hilo** es la instancia de ejecución dentro de un proceso, es código que está siendo ejecutado *serialmente* dentro de un proceso. Técnicamente, un hilo es una colección de:

- Estados de la pila.
- Información del contexto de ejecución (como el estado de los registros del CPU)

Un procesador ejecuta hilos, no procesos, por lo que una aplicación tiene al menos un proceso, y un proceso tiene al menos un hilo de ejecución. El primer hilo de un proceso, el que comienza su ejecución, se le llama *hilo primario*. A partir del hilo primario pueden crearse *hilos secundarios*, y a su vez crearse de éstos. Un hilo puede a su vez crear otros procesos. Cada hilo puede ejecutar secciones de código distintas, o múltiples hilos pueden ejecutar la misma sección de código. Los hilos de un proceso tienen pilas independientes, pero comparten los datos y recursos de su proceso.

A un sistema operativo capaz de “ejecutar” más de un hilo, pertenecientes al mismo o a un distinto proceso, se le llama **multitarea**. Es importante no perder de vista que el sistema operativo mismo es un conjunto de procesos con atribuciones especiales. El sistema operativo (S.O.) utiliza elementos del hardware del computador, como veremos más adelante, para asistirse en el control de la multitarea.

La aparente ejecución simultánea de más de un hilo es sólo un efecto del rápido paso de la ejecución de un hilo a otro, por turnos, por parte del computador. Un computador con un único procesador es capaz de ejecutar el código de un único hilo por vez. Dado que el hilo es la unidad de ejecución de las aplicaciones, es también la unidad de planificación del orden de ejecución de las mismas. El componente del sistema operativo encargado de determinar cuándo y qué tan frecuentemente un hilo debe ejecutarse se le llama **planificador** (scheduler). Cuando uno de los hilos de un proceso está en ejecución, se dice que su proceso está en ejecución. Sin embargo, vale insistir, el decir que un proceso se ejecuta significa que uno de sus hilos se está ejecutando. El computador ejecuta hilos, no procesos.

Para sacar a un hilo de ejecución e ingresar otro, se realiza un **cambio de contexto**. Este cambio se realiza mediante una interrupción de hardware, la que ejecuta una rutina instalada por el S.O. cuando se arrancó el computador (como es de suponerse, el planificador) y que realiza lo siguiente:

- Guarda la información del estado de ejecución del hilo saliente.
- Carga la información del estado de ejecución del hilo entrante.
- Finaliza la rutina de la interrupción, por lo que “continúa” la ejecución del hilo configurado, es decir, el hilo entrante.

Cuando el cambio de contexto es entre hilos de un mismo proceso, éste se realiza en forma rápida, dado que la información del contexto de ejecución relacionada con el proceso, como es el espacio de direccionamiento, no requiere ser modificada, todos los hilos de un proceso comparten el mismo espacio de direccionamiento. Cuando el cambio de contexto es entre hilos de distintos procesos, éste es más lento debido al número de cambios que es necesario realizar. Éste es el motivo por el cual los S.O. multitarea modernos separan los conceptos de proceso e hilo, dado que técnicamente un S.O. podría trabajar únicamente con procesos como unidad de ejecución. El uso de los hilos le permite a los programas que requieren realizar varias tareas en paralelo o concurrentemente, usar hilos en lugar de crear nuevos procesos, mejorando el desempeño general del computador.

Espacio de Memoria Virtual

Los procesos establecen un nivel de aislamiento entre una aplicación y otra, donde los hilos de cada uno son ejecutados como si sólo existiese un único proceso en el computador. Esto se consigue manejando por cada proceso un espacio de memoria virtual. Este espacio de memoria se obtiene con un manejo lógico de la memoria física del computador, de forma que cada proceso tenga la “sensación” de que cuenta con un espacio de memoria mayor del que físicamente posee la memoria RAM del computador. Cuando un hilo de un proceso quiere acceder a una posición de memoria virtual se produce un “mapeo” de la dirección que maneja el código del programa, a la dirección física. Si el hilo intenta trabajar con más memoria de la existente en la RAM, el S.O. juega con la memoria en disco, bajando y subiendo bloques de memoria de la RAM en un proceso llamado **swapping**.

El rango de valores para las direcciones de memoria que los hilos de un proceso pueden usar se le llama **espacio de direccionamiento**. Dado que dichas direcciones son virtuales y que los procesos deben ejecutarse independientemente unos de otros, la misma dirección para hilos de procesos distintos se referirá a posiciones de memoria distintas.

Dado que cada proceso maneja su propio espacio de direccionamiento, se evita que un hilo de un proceso pueda acceder inadvertidamente a la memoria de otro hilo, produciendo errores.

Esto permite una gran seguridad al trabajar con varios procesos a la vez, pero dificulta la comunicación entre procesos.

Ciclo de vida de un Hilo

Desde el momento en que la información utilizada por el computador para poder ejecutar un hilo es creada por el S.O., el hilo pasa por una secuencia de etapas hasta que es eliminado. La Figura 8 - 1 muestra un diagrama de estados general para un hilo. Los cambios de estado en el ciclo de vida de un hilo son producidos por el sistema operativo o por una llamada a algún método.

Cuando un hilo se crea no se ejecuta en “ese” momento, su estado es *creado* y deberá pasar al estado *listo* para ser elegible por el planificador y ejecutarse. El estado *listo* significa “listo para ser ejecutado”. En todo momento del trabajo de un computador, el planificador de tareas mantiene una “lista de los hilos listos”, de forma que cuando el hilo en ejecución actual termine de ejecutar, el planificador escoja de la “lista de listos” un nuevo hilo para ser ejecutado. Cuando un hilo entra a ejecutarse, su estado pasa a *en ejecución*.

Note que un hilo puede pasar del estado *listo* al de *en ejecución* y nuevamente al estado *listo* repetidas veces. Estos cambios se basan en la forma en que el S.O. planifica la ejecución de los hilos listos.

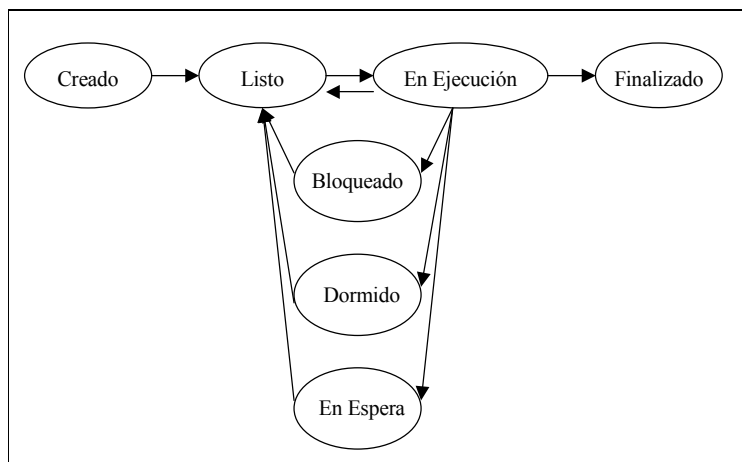


Figura 8 - 1 Ciclo de vida de un hilo

Note que un hilo en ejecución puede pasar a un estado de latencia en el que no se le asignará el CPU hasta que ocurra el evento adecuado que lo saque de dicho estado, pasándolo al estado de *listo*. Estos estados de latencia son: *Bloqueado*, *dormido*, *en espera*.

Cuando un hilo ha finalizado (o muerto), la información que el S.O. guarda de él no ha sido aún eliminada. Esto permite que otro hilo en ejecución pueda verificar dicho estado y pueda tomar alguna acción, lo que puede servir para sincronizar las acciones de los hilos.

Planificación

Un hilo se ejecuta hasta que muere, le cede la ejecución a otros hilos o es interrumpido por el planificador. Cuando el planificador tiene la capacidad de interrumpir un hilo, cuando su tiempo de ejecución vence u otro hilo de mayor prioridad está listo para ser ejecutado, se le llama **preemptivo**. Cuando debe esperar a que dicho hilo ceda voluntariamente el CPU a otros hilos, se le llama **no preemptivo**. Éste último esquema ofrece mayor control al programador sobre la

planificación de su programa, por consiguiente es más propenso a las fallas, por lo que los S.O. multitarea modernos son preentivos. En los S.O. preentivos, una forma de determinar cuándo un hilo debe salir de ejecución es dándole un tiempo límite, al que se le llama **quantum**. Windows 95 y sus predecesores, así como Windows NT y sus predecesores, son sistemas operativos preentivos por quantums.

Cuando el quantum de un hilo vence, el planificador debe elegir cuál será el nuevo hilo a ejecutar. Uno de los elementos que se utiliza para esto es la prioridad del hilo. Todo hilo tiene una prioridad relacionada, la que está relacionada con la prioridad de su proceso. El planificador maneja una lista con prioridad de “listas de hilos listos”, como se muestra en la Figura 8 - 2. El planificador suele escoger un hilo dentro de la lista de mayor prioridad.

El trabajo con prioridades permite ejecutar más continuamente los hilos cuyos trabajos se consideran más prioritarios. Sin embargo, esto trae consigo dos problemas:

- **Bloqueos muertos** (deadlocks), ocurren cuando un hilo espera a que otro hilo, con menor prioridad, realice algo para poder continuar. Dado que el hilo esperado tiene menor prioridad que quien espera, nunca será elegido para ejecutarse, por lo que la condición nunca se cumplirá.
- **Condiciones de carrera** (race conditions), ocurren cuando un hilo acaba antes que otro, del cual depende para modificar los recursos compartidos, por lo que accede a valores errados.

Existen técnicas que los S.O. y las aplicaciones utilizan para determinar estas condiciones y evitarlas o solucionarlas.

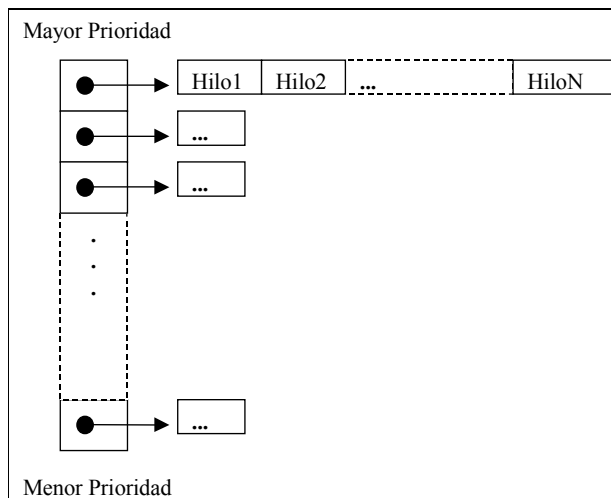


Figura 8 - 2 Lista con prioridad de los hilos "listos" del planificador.

Soporte para la Programación Concurrente

C y C++ no ofrecen soporte nativo (como parte de la especificación del lenguaje) para la programación concurrente, por lo que requieren hacer uso directo de las librerías que el sistema operativo subyacente provea para dicho fin.

Java y C# proveen soporte nativo para la programación concurrente, con librerías de clases para dicho fin que forman parte de la especificación del lenguaje. Ambos hacen uso de un conjunto reducido de características básicas de las librerías del S.O. subyacente. En el caso de Java, esto

origina que sólo se permita la programación multihilos pero como contraparte se pueda utilizar las mismas librerías de clases y un comportamiento “bastante” similar desde distintos S.O. C# hace una implementación interna de la mayor parte de las características de la programación concurrente, como una forma de solucionar muchos de los problemas más comunes de esta programación sin modificar o depender del S.O.

Manejo de Hilos

En esta sección revisaremos las técnicas más simples de manejo de hilos vistas desde C/C++, Java y C#. No se verá el tema de la programación con procesos.

Creación

Una aplicación crea un hilo en Windows haciendo uso de la función `CreateThread`. El siguiente código en C++ muestra el uso de esta función:

```
#include <windows.h>
#include <stdio.h>

DWORD __stdcall FuncionHilo(void* dwData) {
    char* DatosHilo = (char*) dwData;
    printf("Ejecutado hilo con mensaje %s\n", DatosHilo);
}

int main(int argc, char *argv[]) {
    for(int i = 0; i < 10; i++) {
        char* DatosHilo = new char[20];
        sprintf(DatosHilo, "Hilo-%d", i);
        DWORD dwID;
        HANDLE hHilo = CreateThread(
            NULL,           // dirección de una estructura SECURITY_ATTRIBUTES
            0,              // tamaño inicial de la pila del hilo
            FuncionHilo,     // punto de entrada del hilo
            (void*) DatosHilo, // parámetro del punto de entrada
            0,              // banderas con opciones. CREATE_SUSPENDED => ResumeThread
            &dwID           // recibe el identificador del hilo
        );
    }
    getchar();
    return 0;
}
```

Al igual que la función `main` o `WinMain` que son los puntos de entrada de nuestros programas, cuando se crea un hilo es necesario especificar una función de entrada, de forma que el S.O. sepa que cuando el hilo termina de ejecutar dicha función, finaliza. El ejemplo anterior crea 10 hilos con la misma función de entrada, `FuncionHilo`, pasándole como parámetro una cadena de texto, cuarto parámetro de **CreateThread**, que es impresa por cada hilo al ejecutar `FuncionHilo`. Si bien en este ejemplo pasamos un único dato a la función de entrada, utilizando un puntero a una estructura o clase es posible pasar toda la información requerida a la función de entrada de un hilo. Esta función de entrada podrá tener cualquier nombre pero la misma firma (tipos del valor de retorno y los parámetros) que `FuncionHilo`. El significado de la palabra reservada `__stdcall` está relacionado con un requisito en la declaración de las funciones de entrada de los hilos del API de Windows. Su significado se verá más adelante.

El ejemplo anterior termina con un `getchar` al final de `main` debido a que, si `main` termina, el proceso termina. Eso se debe a que, si bien técnicamente un proceso no finaliza hasta que todos sus hilos finalizan, el hilo primario tiene un tratamiento especial en el sentido de que, si éste finaliza, se fuerza la finalización de los hilos secundarios y el proceso termina. Podemos probar esto quitando el `getchar` de `main` y corriendo el programa.

Un ejemplo igualmente sencillo en Java sería el siguiente:

```
class Hilo extends Thread {
    public Hilo( String nombre ) {
        super( nombre );
    }
    public void run() {
        System.out.println( "Hilo " + getName() + " ejecutado" );
    }
}

class Hilos0 {
    public static void main(String args[]) {
        for(int i = 0; i < 10; i++) {
            Hilo hilo = new Hilo("Hilo" + i);
            hilo.start();
        }
    }
}
```

En Java se hace uso de la clase **Thread**. Para crear un hilo se suele heredar de dicha clase y sobrescribir el método *run*, colocando dentro de él todo el código que deseamos que el hilo ejecute. El hilo se crea en estado *creado* y pasa al estado *listo* al llamar al método *start*. El método *run* es el equivalente a la función de entrada del hilo en API de Windows, con la ventaja de ser un método, por lo que todos los datos que se hubiera deseado pasar a su equivalente en API de Windows, se pasan al crear el objeto de la clase derivada de *Thread* o bien llamando a sus métodos y guardándolos como datos miembros del objeto. Esto ofrece un enfoque más *orientado a objetos* para el manejo de hilos. Note que se llama a *super* en el constructor pasándole una cadena, lo que establece un nombre al objeto, lo que es útil para efectos de depuración. Dicho nombre se puede modificar y obtener con los métodos **setName** y **getName** respectivamente.

A diferencia del ejemplo de API de Windows, no se requiere de ningún artificio al final de *main* para evitar que el hilo primario finalice sin darle oportunidad a los secundarios a terminar su trabajo. Esto se debe a que cuando se retorna de *main* el intérprete de Java, quien es el que llama a *main*, se encarga de esperar a todos los hilos que fueron creados hasta que finalicen y recién allí finalizar el hilo primario y con él, el proceso.

En conclusión, el intérprete de Java realiza el siguiente algoritmo:

```
main(...);
if(hay_hilos_pendientes)
    joinAll();
return;
```

Un ejemplo igualmente sencillo en C# sería el siguiente:

```
using System;
using System.Threading;

class Hilos0 {
    public static void Ejecutar() {
        Thread hiloActual = Thread.CurrentThread;
        Console.WriteLine("Hilo " + hiloActual.Name + " ejecutado");
    }
    public static void Main() {
        for(int i = 0; i < 10; i++) {
            Thread hilo = new ThreadStart(Ejecutar);
            hilo.Name = "Hilo" + i;
            hilo.Start();
        }
    }
}
```

En C# se utiliza la clase **Thread**, de la cual “no se puede heredar”, dado que ha sido declarada como *sealed*. Lo que se hace con ella es instanciarla pasándole como parámetro un objeto

delegado que será utilizado para llamar a su método cuando el hilo se ejecute. El método relacionado con el objeto delegado hace el papel de la función de entrada del hilo. Este esquema ofrece la ventaja de evitar una herencia y poder hacer que cualquier método, estático o no, pertenecientes a la misma clase o a distintas, sean ejecutados por hilos distintos, todo esto sin perder el enfoque orientado a objetos.

Al igual que Java, existe código que se ejecuta después de finalizado el método Main y que realiza una espera hasta que todos los hilos secundarios finalicen, para luego finalizar el primario.

Clases para el Manejo de Hilos

C#

Las clases relacionadas con el manejo de hilos se encuentran en el espacio de nombres **System.Threading**. Dentro de ésta existen dos clases importantes: **Thread** y **Monitor**.

Tabla 8 - 1 Métodos de la clase Thread

Método	Descripción
Thread(ThreadStart)	Constructor que crea el hilo en estado no-iniciado, denominado “Unstarted”. La documentación no indica si este estado corresponde a un hilo creado en estado “suspendido”, o bien significa que el hilo aún no se ha creado, tan solo se ha creado el objeto Thread que permitirá manejarlo.
Start()	Coloca el hilo en estado “listo”, denominado “Running”. Tanto un hilo “listo” como uno “en-ejecución” tienen en .NET el estado Running.
Abort()	Causa un ThreadAbortException en el hilo. La excepción puede ser capturada por un bloque catch en el código ejecutado por el hilo, pero al finalizar dicho bloque catch la excepción sera automáticamente relanzada. Debido a esto la ejecución inevitablemente terminará saliendo del método que es punto de entrada del hilo, por lo que el hilo finalizará, con estado “Stopped”. Si nadie refiere al objeto Thread, será recolectado en algún momento.
Sleep(mlseg)	Coloca al hilo en estado de bloqueo, denominado “WaitSleepJoin”. Es el S.O. quien lo saca luego de un tiempo. El parámetro mlseg especifica dicho tiempo en milisegundos.
Interrupt()	Saca al hilo de un estado “WaitSleepJoin” y lo pone en estado “Running”.
Join()	Permiten que un hilo espere a que otro hilo finalice su ejecución.
Suspend()	Permite que un hilo suspenda a otro hilo.
Resume()	Permite que un hilo saque del estado de suspensión a otro hilo.

El estado *WaitSleepJoin* agrupa a varias condiciones de bloqueo: La espera de un objeto de bloqueo que no sea un hilo, la espera a que un tiempo transcurra y la espera a que un hilo finalice.

Tabla 8 - 2 Métodos de la clase Monitor

Método	Descripción
Wait()	Coloca al hilo en estado “WaitSleepJoin”.
Pulse() PulseAll()	Sacan del estado “WaitSleepJoin” a los hilos inactivos por dicho monitores.

Java

Todos los programas que se han mostrado como ejemplos hasta ahora han utilizado más de un hilo, dado que, en paralelo con el hilo principal que corre nuestro programa, el intérprete de Java dispara un hilo adicional, el **garbage collector**.

Sin embargo, a pesar de que uno de los objetivos principales del lenguaje Java es la portabilidad, la implementación de hilos en Java no llega a ser, hasta ahora, independiente de la plataforma. Esto se debe a las significativas diferencias entre los diversos sistemas operativos en la forma en que implementan la multitarea.

La Clase Thread

Es la clase base para la creación de hilos. Cuando se crea un hilo se define una nueva clase que deriva de **Thread**. Una vez que un nuevo objeto hilo ha sido creado, se utilizan los métodos heredados de Thread para administrarlo.

Cuando se instancia un objeto de una clase de hilo su estado es *creado* (llamado también *nacido*). Esto es equivalente a crear en C para Windows un hilo en estado *suspendido*.

Para que el hilo comience a correr, se debe de llamar a su método *start*, heredado de *Thread*. El método *start* arranca el hilo, lo que en su momento llama al método *run* de *Thread*, cuya implementación no hace nada dado que se espera que la clase derivada, en base a la que se creó el objeto hilo, la sobrescriba. Es en *run* donde se debe de colocar el código que debe de ejecutar las tareas del hilo. El método *run* es el equivalente a la función del hilo que se implementa en C para Windows.

Al igual que en C para Windows, mientras se ejecuta el método *run*, el hilo pasa del estado *listo* al estado *en ejecución* y nuevamente al estado *listo* repetidamente, hasta que finaliza este método, lo que coloca al hilo en estado *finalizado* (también llamado *muerto*).

Tabla 8 - 3 Métodos de la clase Thread

Método	Descripción
Thread() Thread(String name)	Constructores. El parámetro name le da un nombre al hilo. El constructor por defecto asigna un nombre de forma automática.
String getName() void setName(String name)	Permiten obtener y modificar el nombre de un objeto hilo.

<code>int getPriority()</code> <code>void setPriority(int newPriority)</code>	Permiten obtener y modificar la prioridad de ejecución de un objeto hilo.
<code>void start()</code>	Coloca el estado del hilo en listo para ejecutarse.
<code>void run()</code>	Este método debe contener el código que ejecute las tareas del hilo.
<code>static void sleep(long millis)</code> <code>static void sleep(long millis, int nanos)</code>	Pone a dormir a un hilo. El parámetro <code>millis</code> especifica un tiempo en milisegundos. El parámetro <code>nanos</code> especifica un tiempo en nanosegundos.
<code>static void yield()</code>	Permite que un hilo voluntariamente renuncie a seguir siendo ejecutado si es que existe algún otro hilo en estado listo.
<code>void interrupt()</code> <code>static boolean interrupted()</code> <code>boolean isInterrupted()</code>	Permiten manejar la interrupción de un hilo.
<code>void join()</code> <code>void join(long millis)</code> <code>void join(long millis, int nanos)</code>	Permiten que un hilo espere a que otro hilo finalice su ejecución. Al igual que <code>sleep</code> , los parámetros permiten especificar un tiempo, en este caso, un tiempo máximo de espera. Si no se especifica un tiempo o éste es cero, se espera indefinidamente.
<code>static Thread currentThread()</code>	Permite obtener una referencia al hilo actual.
<code>static void dumpStack()</code>	Permite imprimir, en la salida estándar, un reporte de la pila de llamadas a métodos hasta el punto de ejecución actual.
<code>String toString()</code>	Permite obtener una descripción textual del hilo: Su nombre, su prioridad y el grupo al que pertenece.

Los métodos estáticos están pensados para ser llamados desde dentro de la ejecución del hilo, esto es, para realizar acciones sobre el hilo actual *en ejecución*. Estos métodos se pueden llamar desde `run` o desde alguno de los métodos (de la misma clase o de otras clases) que éste llame. Tome en cuenta que el método `main` es llamado por el método `run` del hilo primario.

El nombre de un hilo permite identificarlo dentro de un grupo de hilos, aunque dicha característica puede no ser utilizada. Java maneja el concepto de **grupo de hilos**. El manejo de los grupos de hilos así como sus semejanzas y diferencias con el concepto de proceso va más allá de los alcances del curso.

Todo hilo tiene una **prioridad**. Java maneja prioridades en el rango de `Thread.MIN_PRIORITY` (constante igual a 1) a `Thread.MAX_PRIORITY` (constante igual a 10). Por defecto, el hilo primario de un programa (el que llama al método `main` de nuestra clase ejecutable) se crea con la prioridad `Thread.MIN_NORMAL` (constante igual a 5). Cada nuevo objeto hilo que se cree

hereda la prioridad del objeto hilo desde donde se instanció. Ahora bien, es importante recordar que Java se basa en las capacidades multitarea de la plataforma actual de ejecución, por lo que el efecto de la modificación de las prioridades sobre el orden de ejecución de los hilos es dependiente de la plataforma.

Un hilo *en ejecución* puede pasar a estar *bloqueado*, *dormido* o *en espera*. Estos estados pueden ser interrumpidos por el mismo sistema operativo o por otro hilo (método *interrupt*). Cuando estos estados se interrumpen, el hilo pasa al estado *listo*, de forma que pueda tomar alguna acción, como consecuencia de la interrupción, cuando el sistema operativo lo pase al estado *en ejecución*. El hilo guarda en un dato miembro interno el estado interrumpido, de forma que se pueda averiguar este hecho. Un hilo puede averiguar si fue interrumpido llamando el método *isInterrupted*. Este método no modifica el valor de este flag, a diferencia del método *interrupted*, que sí lo hace, colocando el flag a *false*. Esto significa que dos llamadas consecutivas a *interrupted*, luego de que el hilo fue interrumpido, devolverán *true* y *false* respectivamente, a menos que entre llamada y llamada se haya interrumpido nuevamente al hilo. Este flag puede servir como una forma de sincronizar el trabajo de un hilo con otro hilo.

Otra forma de sincronización de hilos es hacer que uno quede en estado de espera hasta que otro haya finalizado su trabajo, esto es, pase al estado *finalizado*. Esto se consigue mediante los métodos *join*.

Por último, el acceso a recursos compartidos y sincronización del trabajo entre los hilos se realiza mediante los objetos **monitores**. El uso de estos se verá mediante ejemplos.

Aspectos Generales de la Ejecución de un Hilo

Un hilo en ejecución, puede dejar de ejecutarse por diversos motivos:

- Si el hilo finaliza, entonces pasa al estado finalizado.
- El sistema operativo concede intervalos de tiempo a cada hilo en estado listo para que se ejecute, por lo tanto, si el intervalo de tiempo del hilo actual en ejecución vence, el hilo pasa a estado listo.
- Si el sistema operativo es preentivo, y determina que un hilo de mayor prioridad ha pasado al estado listo, entonces el hilo actual pasa al estado listo y el de mayor prioridad al estado en ejecución.
- El hilo queda en un estado de bloqueo. Casos comunes (y por lo mismo caracterizados con un nombre especial) son los estados dormido (*sleep*) y en espera (*wait*). Para el primer caso, cuando el sistema operativo advierte que el tiempo de dormir venció, pasa al hilo al estado listo. Para el segundo, un ejemplo es una operación de E/S a disco: Dado que no es el hilo quien realiza la lectura del disco, se bloquea esperando a que el sistema operativo complete la lectura solicitada. Cuando eso sucede, el sistema operativo desbloquea al hilo.

Trabajo con Hilos en C++

Se debe seguir los siguientes pasos:

16. Definir una función de entrada del hilo.
17. Crear el hilo invocando a la función *CreateThread*.

18. Arrancar la ejecución de los hilos.

A continuación se muestra la forma de invocar a la función *CreateThread* y la especificación de sus parámetros:

```
HANDLE hHilo = CreateThread(NULL, 0, FuncionHilo, (void*)Nombre, 0, &dwID);
```

Tabla 8 - 4 Parámetros de la función CreateThread

Parámetro	Descripción
1ero	Dirección de una estructura SECURITY_ATTRIBUTES.
2do	Tamaño inicial de la pila del hilo.
3ro	Punto de entrada del hilo.
4to	Parámetro del punto de entrada.
5to	Banderas que determinan opciones. CREATE_SUSPENDED => ResumeThread.
6to	Recibe el identificador del hilo.

El siguiente programa muestra un ejemplo sencillo de la creación y uso de un hilo.

```
#include <windows.h>
#include <stdio.h>
#include <time.h>

struct DatosHilo {
    char Nombre[20];
    DatosHilo(int i) {
        sprintf(Nombre, "Hilo%d", i);
    }
};

DWORD __stdcall FuncionHilo(void* dwData) {
    DatosHilo* Datos = (DatosHilo*) dwData;
    printf("Hilo %s se va a dormir\n", Datos->Nombre);
    Sleep(rand() % 5000);
    printf("Hilo %s se despertó\n", Datos->Nombre);
    delete Datos;
    return 0;
}

int main(int argc, char *argv[]) {
    srand( (unsigned)time( NULL ) );
    DWORD dwID;
    HANDLE hHilo1, hHilo2, hHilo3, hHilo4;

    hHilo1=CreateThread(NULL, 0, FuncionHilo, new DatosHilo(1), CREATE_SUSPENDED, &dwID);
    hHilo2=CreateThread(NULL, 0, FuncionHilo, new DatosHilo(2), CREATE_SUSPENDED, &dwID);
    hHilo3=CreateThread(NULL, 0, FuncionHilo, new DatosHilo(3), CREATE_SUSPENDED, &dwID);
    hHilo4=CreateThread(NULL, 0, FuncionHilo, new DatosHilo(4), CREATE_SUSPENDED, &dwID);

    printf("Arrancando los hilos ...\n");
    ResumeThread(hHilo1);
    ResumeThread(hHilo2);
    ResumeThread(hHilo3);
    ResumeThread(hHilo4);

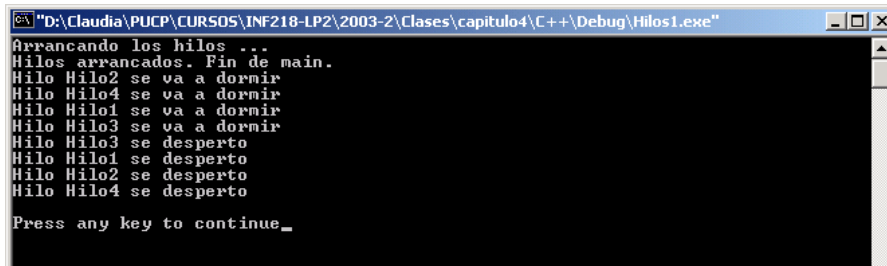
    printf("Hilos arrancados. Fin de main.\n");
    getchar();
    return 0;
}
```

```
}
```

Al correr el programa la salida será parecida a de la figura 8.3. Note que el hilo primario, correspondiente al método main, termina su trabajo antes de que cualquier hilo comience a ejecutarse. Sin embargo, esto no necesariamente es así.

Note que, a diferencia de Java y C# (como se verá más adelante) la “aparente” finalización del hilo primario sí finaliza el programa. Esto se debe a que en C++ los programas generados no agregan ningún código especial que permita al hilo primario esperar hasta que todos los secundarios hayan finalizado. Si se desea hacer esto tendrá que agregar a la función main (o a funciones llamadas desde ésta) código explícito para esta tarea.

Si bien en la Figura 8 - 3 se muestra que los hilos inician su trabajo con cierto orden, mostrando su mensaje, esto no necesariamente es así, por lo que no podemos asumir que el orden en que se arrancan los hilos será el orden en que comiencen a ejecutarse. En general, el orden de ejecución de los hilos es una decisión del sistema operativo en base a las políticas que implemente su planificador.



```
"D:\Claudia\PUCP\CURSOS\INF218-LP2\2003-2\Clases\capitulo4\C++\Debug\Hilos1.exe"
Arrancando los hilos ...
Hilos arrancados. Fin de main.
Hilo Hilo2 se va a dormir
Hilo Hilo4 se va a dormir
Hilo Hilo1 se va a dormir
Hilo Hilo3 se va a dormir
Hilo Hilo3 se despierto
Hilo Hilo1 se despierto
Hilo Hilo2 se despierto
Hilo Hilo4 se despierto
Press any key to continue_
```

Figura 8 - 3 Salida del Ejemplo de Hilos en C++

Sincronización

En C++ utilizaremos **secciones críticas** para la sincronización de hilos. La *sección crítica* se define dentro de una función delimitando las instrucciones que se quiere que solamente un hilo a la vez las ejecute.

Cuando un grupo de instrucciones están delimitadas por una *sección crítica* y son ejecutadas desde un hilo, ningún otro hilo puede acceder a éstas. Cuando un hilo entra a ejecutar una *sección crítica*, ésta queda bloqueada. Esto implica que si un hilo intenta ejecutar esta *sección crítica*, quedará en estado de *espera* hasta que ésta se desbloquee. Cuando un hilo sale de la *sección crítica*, ésta se desbloquea, permitiendo que otros hilos accedan a ella. Si hubiese algún hilo esperando una llamada pendiente a la *sección crítica*, el hilo pasa al estado *listo*, de forma que cuando entre al estado *en ejecución* pueda ejecutar el código de la *sección crítica*.

El siguiente ejemplo muestra el uso de una *sección crítica*.

```
#include <windows.h>
#include <stdio.h>
#include <time.h>

class Productor {
    int Trabajo;
    CRITICAL_SECTION cs;
public:
    Productor() {
        Trabajo = 0;
        InitializeCriticalSection(&cs);
    }
    ~Productor() {
        DeleteCriticalSection(&cs);
    }
};
```

```
    }
    int SiguienteTrabajo() {
        EnterCriticalSection(&cs);
        int Nuevo = Trabajo++;
        Sleep( rand() % 1000 );
        LeaveCriticalSection(&cs);
        return Nuevo;
    }
};

class DatosConsumidor {
    char Nombre[20];
    Productor* p;
public:
    DatosConsumidor(char* Nombre, Productor* p) {
        strcpy(this->Nombre, Nombre);
        this->p = p;
    }
    void Ejecutar() {
        for( int i = 0; i < 5; i++ ) {
            int iTrabajo = p->SiguienteTrabajo();
            printf("%s: Obtenido trabajo %d\n", Nombre, iTrabajo);
            Sleep(rand() % 1000);
            printf("%s: Trabajo %d completado\n", Nombre, iTrabajo);
        }
    }
};

DWORD __stdcall FuncionHilo(void* dwData) {
    DatosConsumidor* Datos = (DatosConsumidor*) dwData;
    Datos->Ejecutar();
    delete Datos;
    return 0;
}

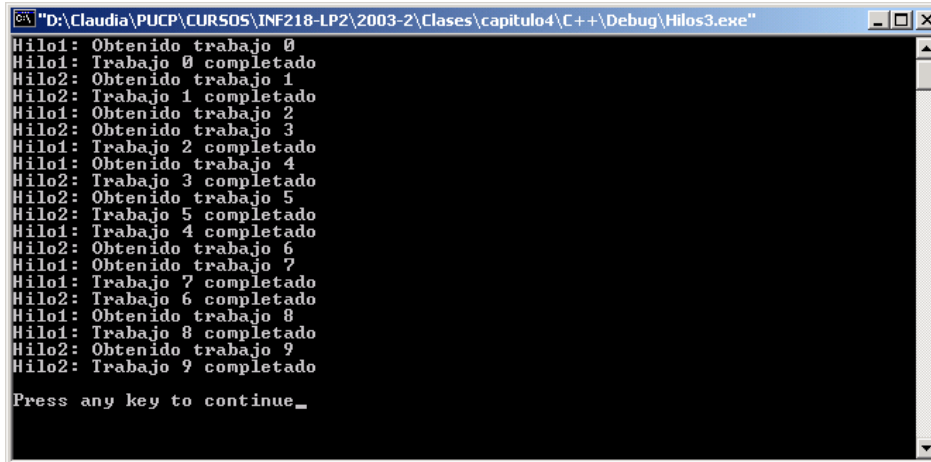
int main(int argc, char *argv[]) {
    srand( (unsigned)time( NULL ) );

    DWORD dwID;
    Productor* p = new Productor();
    CreateThread(NULL, 0, FuncionHilo, new DatosConsumidor("Hilo1", p), 0, &dwID);
    CreateThread(NULL, 0, FuncionHilo, new DatosConsumidor("Hilo2", p), 0, &dwID);

    getchar();
    //delete p;      // este delete es riesgoso, hay un problema de sincronización
    return 0;
}
```

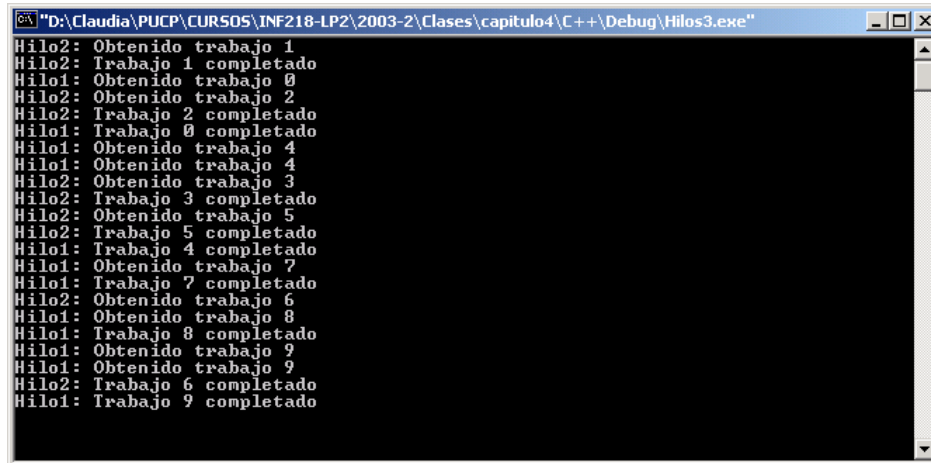
Este programa, cuya ejecución se muestra en la Figura 8 - 4, simula un proceso “productor-consumidor”. El productor devuelve un trabajo a demanda del consumidor que se lo solicita. Cuando el productor calcula un nuevo trabajo le toma un tiempo aleatorio entregarlo. Sin embargo, dado que la función de entrega de un trabajo, *SiguienteTrabajo*, cuenta con una sección crítica delimitando sus instrucciones, sólo un hilo podrá obtener un trabajo a la vez, por lo que no habrá pérdida de trabajos.

Para ejemplificar mejor esta sincronización, la salida de la Figura 8 - 5 muestra lo que podría ocurrir si la función *SiguienteTrabajo* no se sincronizara.



```
D:\Claudia\PUCP\CURSOS\INF218-LP2\2003-2\Clases\capitulo4\C++\Debug\Hilos3.exe
Hilo1: Obtenido trabajo 0
Hilo1: Trabajo 0 completado
Hilo2: Obtenido trabajo 1
Hilo2: Trabajo 1 completado
Hilo1: Obtenido trabajo 2
Hilo2: Obtenido trabajo 3
Hilo1: Trabajo 2 completado
Hilo1: Obtenido trabajo 4
Hilo2: Trabajo 3 completado
Hilo2: Obtenido trabajo 5
Hilo2: Trabajo 5 completado
Hilo1: Trabajo 4 completado
Hilo2: Obtenido trabajo 6
Hilo1: Obtenido trabajo 7
Hilo1: Trabajo 7 completado
Hilo2: Trabajo 6 completado
Hilo1: Obtenido trabajo 8
Hilo1: Trabajo 8 completado
Hilo2: Obtenido trabajo 9
Hilo2: Trabajo 9 completado
Press any key to continue_
```

Figura 8 - 4 Salida del ejemplo de sincronización de hilos en C++



```
D:\Claudia\PUCP\CURSOS\INF218-LP2\2003-2\Clases\capitulo4\C++\Debug\Hilos3.exe
Hilo2: Obtenido trabajo 1
Hilo2: Trabajo 1 completado
Hilo1: Obtenido trabajo 0
Hilo2: Obtenido trabajo 2
Hilo2: Trabajo 2 completado
Hilo1: Trabajo 0 completado
Hilo1: Obtenido trabajo 4
Hilo1: Obtenido trabajo 4
Hilo2: Obtenido trabajo 3
Hilo2: Trabajo 3 completado
Hilo2: Obtenido trabajo 5
Hilo2: Trabajo 5 completado
Hilo1: Trabajo 4 completado
Hilo1: Obtenido trabajo 7
Hilo1: Trabajo 7 completado
Hilo2: Obtenido trabajo 6
Hilo1: Obtenido trabajo 8
Hilo1: Trabajo 8 completado
Hilo1: Obtenido trabajo 9
Hilo1: Obtenido trabajo 9
Hilo2: Trabajo 6 completado
Hilo1: Trabajo 9 completado
```

Figura 8 - 5 Salida del ejemplo “Productor-Consumidor” sin sincronización de Hilos en C++

El siguiente ejemplo muestra el uso de la función *WaitForSingleObject*.

```
#include <windows.h>
#include <stdio.h>
#include <time.h>

class DatosHilo {
    bool Finalizar;
public:
    DatosHilo() {
        Finalizar = false;
    }
    void Finaliza() {
        Finalizar = true;
    }
    void Ejecutar() {
        int Trabajo = 0;
        while( !Finalizar ) {
            Trabajo++;
            printf("Hilo: Inicio de trabajo %d\n", Trabajo );
            Sleep(rand() % 3000);
            printf("Hilo: Finalizo trabajo %d\n", Trabajo );
        }
    }
};
```

```
DWORD __stdcall FuncionHilo(void* dwData) {
    DatosHilo* Datos = (DatosHilo*) dwData;
    Datos->Ejecutar();
    delete Datos;
    return 0;
}

int main(int argc, char *argv[]) {
    srand( (unsigned)time( NULL ) );

    DWORD dwID;
    DatosHilo* Datos = new DatosHilo();
    HANDLE hHilo = CreateThread(NULL, 0, FuncionHilo, Datos, 0, &dwID);

    printf("Hilo primario se va a dormir ...\n" );
    Sleep(rand() % 3000);
    printf("Hilo primario se despierta. Esperando a que finalice el hilo "
           "secundario ...\n");
    Datos->Finaliza();
    WaitForSingleObject(hHilo, INFINITE);
    printf("Función main finalizó\n");
    getchar();
    return 0;
}
```

A continuación en la figura 8.6 se muestra la salida del programa. Note que el hilo primario, el que ejecuta al método main, espera a que el nuevo hilo creado finalice para continuar.

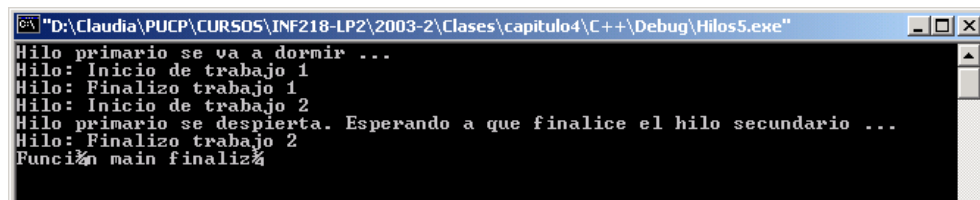


Figura 8 - 6 Salida del ejemplo del uso de la función WaitForSingleObject en C++

Trabajo con Hilos en C#

Las clases para hilos están en el espacio de nombres **System.Threading**. La clase *Thread* es *sealed*, es decir, nadie puede heredar de ella.

El procedimiento para crear un hilo es:

35. Crear un objeto delegado ThreadStart, pasándole como parámetro el método que ejecutará el hilo.
36. Crear un objeto Thread, pasándole como parámetro el objeto delegado.
37. Iniciar la ejecución del hilo llamando al método Start.

El siguiente programa muestra un ejemplo sencillo de creación y uso de un hilo.

```
using System;
using System.Threading;

class DatosHilo {
    private static Random r = new Random();
    public void Ejecutar() {
        Thread HiloActual = Thread.CurrentThread;
        int tiempo = r.Next(5000);
        Console.WriteLine("Hilo " + HiloActual.Name + " se va a dormir");
        Thread.Sleep(tiempo);
        Console.WriteLine("Hilo " + HiloActual.Name + " se despertó");
    }
}
```



```
class Hilos1 {
    public static void Main() {
        DatosHilo datos1, datos2, datos3, datos4;
        Thread hilo1, hilo2, hilo3, hilo4;
        datos1 = new DatosHilo();
        datos2 = new DatosHilo();
        datos3 = new DatosHilo();
        datos4 = new DatosHilo();
        hilo1 = new Thread(new ThreadStart(datos1.Ejecutar));
        hilo2 = new Thread(new ThreadStart(datos2.Ejecutar));
        hilo3 = new Thread(new ThreadStart(datos3.Ejecutar));
        hilo4 = new Thread(new ThreadStart(datos4.Ejecutar));
        hilo1.Name = "Hilo1";
        hilo2.Name = "Hilo2";
        hilo3.Name = "Hilo3";
        hilo4.Name = "Hilo4";
        Console.WriteLine( "Arrancando los hilos ..." );
        hilo1.Start();
        hilo2.Start();
        hilo3.Start();
        hilo4.Start();
        Console.WriteLine( "Hilos arrancados. Fin de main.\n" );
    }
}
```

Al correr el programa la salida será parecida a de la figura 8.7. Note que el hilo primario, correspondiente al método main, termina su trabajo antes de que cualquier hilo comience a ejecutarse. Sin embargo, esto no necesariamente es así.

Note que la finalización del hilo primario no finaliza el programa. El programa finaliza cuando todos los hilos creados finalizan.

Si bien en la Figura 8 - 7 se muestra que los hilos inician su trabajo con cierto orden, mostrando su mensaje, esto no necesariamente es así, por lo que no podemos asumir que el orden en que se arrancan los hilos será el orden en que comiencen a ejecutarse. En general, el orden de ejecución de los hilos es una decisión del sistema operativo en base a las políticas que implemente su planificador.

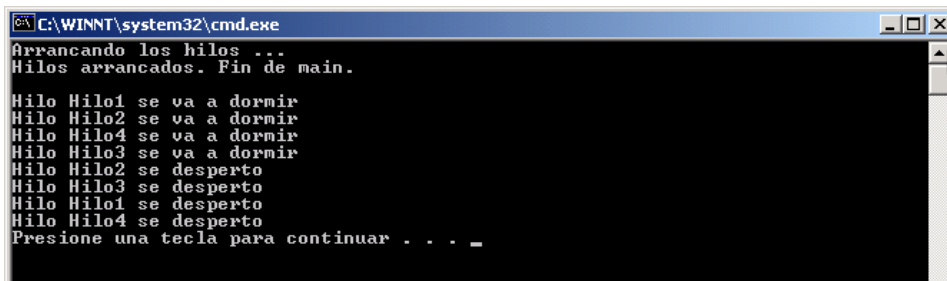


Figura 8 - 7 Salida del ejemplo de hilos en C#

Como se explicó, un hilo *bloqueado*, *dormido* o *en espera* puede ser interrumpido y sacado de este estado. Debido a esto, las llamadas a los métodos que producen estos estados disparan excepciones adecuadas para los casos en que estos estados son interrumpidos. El manejo de estos cambios de estado y otras acciones con hilos utilizan ampliamente las excepciones para su control. En el caso del método `Interrupt` se produce la excepción `System.Threading.ThreadInterruptedException`.

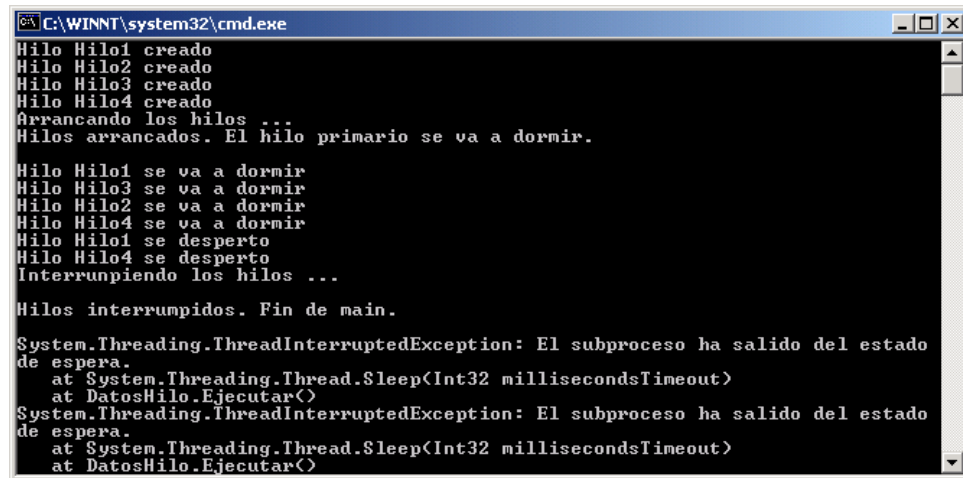
El siguiente programa muestra la interrupción de los hilos.

```
using System;
using System.Threading;
```

```
class DatosHilo {
    private static Random r = new Random();
    private Thread hilo;
    public Thread Hilo {
        get { return hilo; }
    }
    public DatosHilo(string Nombre) {
        hilo = new Thread(new ThreadStart(Ejecutar));
        hilo.Name = Nombre;
        Console.WriteLine( "Hilo " + hilo.Name + " creado" );
    }
    public void Ejecutar() {
        int Tiempo = r.Next(5000);
        Console.WriteLine("Hilo " + hilo.Name + " se va a dormir");
        Thread.Sleep(Tiempo);
        Console.WriteLine("Hilo " + hilo.Name + " se despierto");
    }
}

class Hilos2 {
    public static void Main() {
        DatosHilo datos1, datos2, datos3, datos4;
        datos1 = new DatosHilo( "Hilo1" );
        datos2 = new DatosHilo( "Hilo2" );
        datos3 = new DatosHilo( "Hilo3" );
        datos4 = new DatosHilo( "Hilo4" );
        Console.WriteLine("Arrancando los hilos ...");
        datos1.Hilo.Start();
        datos2.Hilo.Start();
        datos3.Hilo.Start();
        datos4.Hilo.Start();
        Console.WriteLine("Hilos arrancados. El hilo primario se va a dormir.\n");
        Thread.Sleep( 2500 );
        Console.WriteLine("Interrunpiendo los hilos ...");
        datos1.Hilo.Interrupt();
        datos2.Hilo.Interrupt();
        datos3.Hilo.Interrupt();
        datos4.Hilo.Interrupt();
        Console.WriteLine("\nHilos interrumpidos. Fin de main.\n");
        Console.ReadLine();
    }
}
```

Al correr el programa la salida será parecida a la mostrada en la Figura 8 - 8.



```
C:\WINNT\system32\cmd.exe
Hilo Hilo1 creado
Hilo Hilo2 creado
Hilo Hilo3 creado
Hilo Hilo4 creado
Arrancando los hilos ...
Hilos arrancados. El hilo primario se va a dormir.

Hilo Hilo1 se va a dormir
Hilo Hilo3 se va a dormir
Hilo Hilo2 se va a dormir
Hilo Hilo4 se va a dormir
Hilo Hilo1 se despierto
Hilo Hilo4 se despierto
Interrunpiendo los hilos ...

Hilos interrumpidos. Fin de main.

System.Threading.ThreadInterruptedException: El subproceso ha salido del estado
de espera.
   at System.Threading.Thread.Sleep(Int32 millisecondsTimeout)
   at DatosHilo.Ejecutar()
System.Threading.ThreadInterruptedException: El subproceso ha salido del estado
de espera.
   at System.Threading.Thread.Sleep(Int32 millisecondsTimeout)
   at DatosHilo.Ejecutar()
```

Figura 8 - 8 Salida del ejemplo de interrupción de hilos en C#

Sincronización

C# utiliza la clase `Monitor` para la sincronización de hilos. La clase `Monitor` mediante sus métodos estáticos delimita una sección crítica. Esto implica que en cualquier método se puede definir una zona crítica.

Cuando un grupo de instrucciones están delimitadas por una *sección crítica* y son ejecutadas desde un hilo, ningún otro hilo puede acceder a éstas. Cuando un hilo entra a ejecutar una *sección crítica*, ésta queda bloqueada. Esto implica que si un hilo intenta ejecutar esta *sección crítica*, quedará en estado de *espera* hasta que ésta se desbloquee. Cuando un hilo sale de la *sección crítica*, ésta se desbloquea, permitiendo que otros hilos accedan a ella. Si hubiese algún hilo esperando una llamada pendiente a la *sección crítica*, el hilo pasa al estado *listo*, de forma que cuando entre al estado *en ejecución* pueda ejecutar el código de la *sección crítica*.

El siguiente ejemplo muestra el uso de una *sección crítica*.

```
using System;
using System.Threading;

class Productor {
    public static Random Rand = new Random();
    private int Trabajo = 0;

    public int SiguienteTrabajo() {
        Monitor.Enter(this);
        Trabajo++;
        Thread.Sleep(Rand.Next(1000));
        Monitor.Exit(this);
        return Trabajo;
    }
}

class DatosConsumidor {
    public readonly Thread Hilo;
    private Productor P;

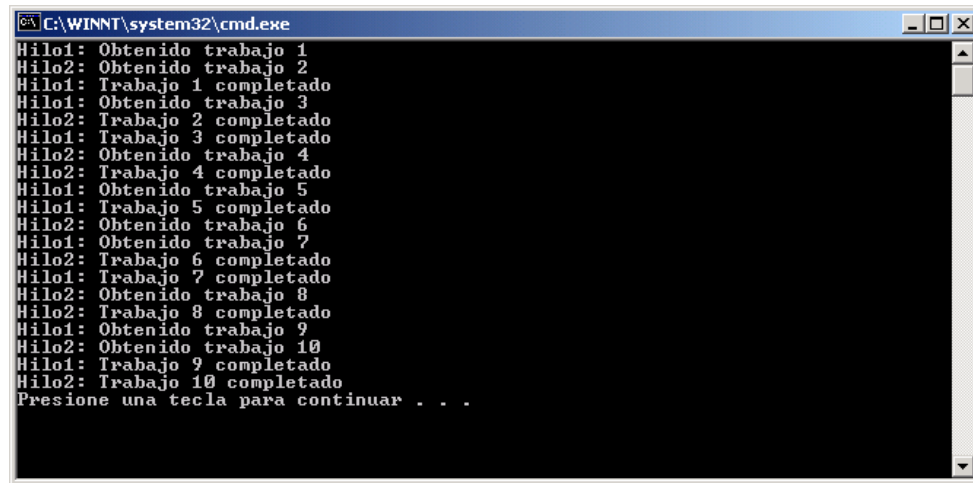
    public DatosConsumidor(string Nombre, Productor P) {
        this.P = P;
        Hilo = new Thread(new ThreadStart(Ejecutar));
        Hilo.Name = Nombre;
    }

    public void Ejecutar() {
        for( int i = 0; i < 5; i++ ) {
            int Trabajo = P.SiguienteTrabajo();
            Console.WriteLine(Hilo.Name + ": Obtenido trabajo " + Trabajo);
            Thread.Sleep(Productor.Rand.Next(1000));
            Console.WriteLine(Hilo.Name + ": Trabajo " + Trabajo + " completado"
);
        }
    }
}

public class Hilos3 {
    public static void Main() {
        Productor p = new Productor();
        DatosConsumidor datos1 = new DatosConsumidor("Hilo1", p);
        DatosConsumidor datos2 = new DatosConsumidor("Hilo2", p);
        datos1.Hilo.Start();
        datos2.Hilo.Start();
    }
}
```

Este programa, cuya ejecución se muestra en la Figura 8 - 9, simula un proceso “productor-consumidor”. El productor devuelve un trabajo a demanda al consumidor que se lo solicita. Cuando el productor calcula un nuevo trabajo le toma un tiempo aleatorio entregarlo. Sin embargo, dado que la función de entrega de un trabajo, *SiguienteTrabajo*, cuenta con una sección

crítica delimitando sus instrucciones, sólo un hilo podrá obtener un trabajo a la vez, por lo que no habrá pérdida de trabajos.



```
C:\WINNT\system32\cmd.exe
Hilo1: Obtenido trabajo 1
Hilo2: Obtenido trabajo 2
Hilo1: Trabajo 1 completado
Hilo1: Obtenido trabajo 3
Hilo2: Trabajo 2 completado
Hilo1: Trabajo 3 completado
Hilo2: Obtenido trabajo 4
Hilo2: Trabajo 4 completado
Hilo1: Obtenido trabajo 5
Hilo1: Trabajo 5 completado
Hilo2: Obtenido trabajo 6
Hilo1: Obtenido trabajo 7
Hilo2: Trabajo 6 completado
Hilo1: Trabajo 7 completado
Hilo2: Obtenido trabajo 8
Hilo2: Trabajo 8 completado
Hilo1: Obtenido trabajo 9
Hilo2: Obtenido trabajo 10
Hilo1: Trabajo 9 completado
Hilo2: Trabajo 10 completado
Presione una tecla para continuar . . .
```

Figura 8 - 9 Salida del ejemplo de sincronización de hilos en C#

Ahora bien, que pasaría en el caso de tener un productor que produce indiferentemente de que existan consumidores que consuman dichos trabajos, y a la vez consumidores que consuman indiferentemente de que exista algo que consumir. Si el productor elimina un trabajo para crear uno nuevo antes de que el anterior sea consumido, se perderá dicho trabajo. Si el consumidor consume un trabajo antes de que se haya producido uno nuevo, un trabajo será realizado dos o más veces. En este caso, ambas labores, la de producir y la de consumir deben de sincronizarse una con otra, esto es, el productor no debe seguir produciendo mientras que no se haya consumido el trabajo anterior y el consumidor debe esperar a que se produzca un nuevo trabajo antes de consumirlo.

En este tipo de situación, la sincronización utilizando la clase *Monitor* por sí sola no es suficiente. Se necesita que el hilo productor espere y notifique al consumidor, y éste a su vez espere y notifique al productor.

Para esto, la clase *Monitor* provee los métodos *Wait*, *Pulse* y *PulseAll*. El método *Wait* coloca al hilo en estado de espera y desbloquea al monitor. El hilo saldrá de dicho estado cuando otro hilo acceda y llame a *Pulse* o *PulseAll*. Es ese momento el hilo será colocado en estado *listo* y competirá con el resto de hilos que intentan acceder a algún *sección crítica* de algún método (o que también hayan salido del estado de espera) para bloquearlo y ejecutar su código.

El método *Pulse* saca al primero de los hilos que esté en la lista de espera, pasándolo del estado *esperando* al estado *listo*. De esta forma el hilo vuelve a entrar a competir por el acceso al objeto monitor.

El método *PulseAll* saca a todos los hilos que esté en la lista de espera, pasándolos del estado *esperando* al estado *listo*.

El siguiente ejemplo muestra el uso de los métodos *Wait* y *Pulse*.

```
using System;
using System.Threading;

class Trabajo {
    private int NroTrabajo = -1;
    private bool PuedoCrear = true; // determina si se puede crear o consumir
```

```
        public void CrearNuevo( int NroTrabajo ) {
            Monitor.Enter(this);
            while ( !PuedoCrear ) { // aún no se puede producir
                try {
                    Monitor.Wait(this);
                }
                catch ( Exception ) {
                }
            }

            Console.WriteLine(Thread.CurrentThread.Name+" creo el trabajo "+ NroTrabajo);
            this.NroTrabajo = NroTrabajo;
            PuedoCrear = false;
            Monitor.Pulse(this); // notifico que existe un trabajo listo
            Monitor.Exit(this);
        }

        public int ObtenerTrabajo() {
            Monitor.Enter(this);
            while ( PuedoCrear ) { // aún no se puede obtener
                try {
                    Monitor.Wait(this);
                }
                catch ( Exception ) {
                }
            }

            PuedoCrear = true;
            Monitor.Pulse(this); // notifico que ya se obtuvo el trabajo actual
            Console.WriteLine( Thread.CurrentThread.Name + " obtuvo el trabajo " +
                NroTrabajo );
            Monitor.Exit(this);
            return NroTrabajo;
        }
    }

    class Productor{
        private Trabajo t;
        public Thread Hilo;
        public Productor( Trabajo t ) {
            Console.WriteLine("Productor");
            this.t = t;
            Hilo = new Thread(new ThreadStart(run));
            Hilo.Name = "Productor";
        }

        public void run() {
            for ( int Contador = 1; Contador <= 5; Contador++ ) {
                // simulo un tiempo de demora aleatorio
                // en la creación del nuevo trabajo
                Random random = new Random();
                try {
                    Thread.Sleep(random.Next(1000));
                }
                catch( Exception ) {
                }

                t.CrearNuevo( Contador );
            }

            Console.WriteLine( Hilo.Name + " finalizo la produccion de trabajos" );
        }
    }

    class Consumidor {
        private Trabajo t;
        public Thread Hilo;

        public Consumidor ( Trabajo t ) {
            this.t = t;
            Hilo = new Thread(new ThreadStart(run));
            Hilo.Name = "Consumidor";
            Console.WriteLine("Consumidor");
        }
    }
```

```
public void run() {
    int iTrabajo;
    do {
        iTrabajo = t.ObtenerTrabajo();

        // simulo un tiempo de demora aleatorio
        // en el procesamiento del trabajo
        Random random = new Random();
        try {
            Thread.Sleep(random.Next(1000));
        }
        catch( Exception) {
        }
    } while ( iTrabajo != 5 );

    Console.WriteLine( Hilo.Name + " termino de procesar los trabajos" );
}
}
public class Hilos4 {
    public static void Main( String []args) {
        Trabajo t = new Trabajo();
        Productor p = new Productor( t );
        Consumidor c = new Consumidor( t );

        p.Hilo.Start();
        c.Hilo.Start();
        Console.ReadLine();
    }
}
```

Al correr el programa la salida mostrada en la Figura 8 - 10.

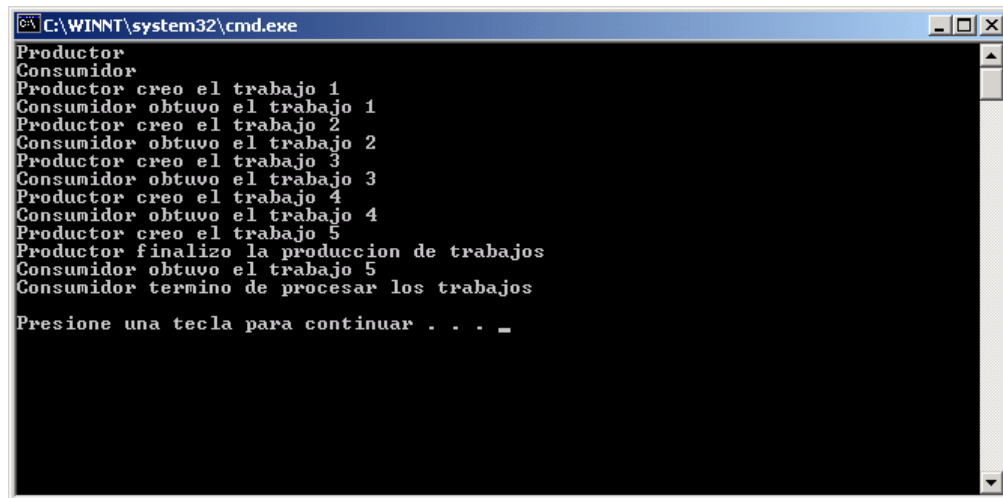


Figura 8 - 10 Salida del ejemplo de sincronización con Wait y Pulse en C#

También es posible hacer que un hilo quede bloqueado hasta que otro hilo pase a estado *finalizado*. Para esta labor se utiliza el método *Join*. El siguiente ejemplo muestra el uso de dicho método.

```
using System;
using System.Threading;

class DatosHilo {
    public static Random Rand = new Random();
    public readonly Thread Hilo;
    private bool Finalizar = false;

    public DatosHilo(string Nombre) {
        Hilo = new Thread(new ThreadStart(Ejecutar));
    }
}
```

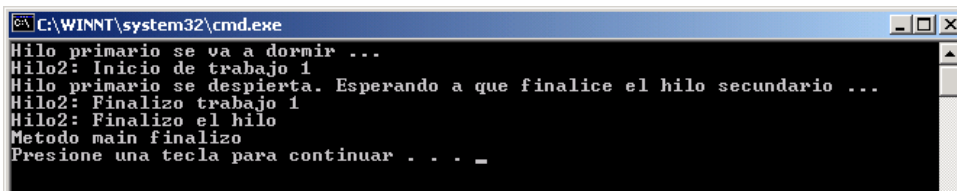
```
Hilo.Name = Nombre;
}
public void Finaliza() {
    Finalizar = true;
}
public void Ejecutar() {
    int Trabajo = 0;
    while( !Finalizar ) {
        Trabajo++;
        Console.WriteLine(Hilo.Name + ": Inicio de trabajo " + Trabajo );
        Thread.Sleep(Rand.Next(3000));
        Console.WriteLine(Hilo.Name + ": Finalizo trabajo " + Trabajo );
    }
    Console.WriteLine(Hilo.Name + ": Finalizo el hilo");
}
}

public class Hilos5 {
    public static void Main() {
        DatosHilo datos = new DatosHilo("Hilo2");
        datos.Hilo.Start();
        Console.WriteLine("Hilo primario se va a dormir ...");
        Thread.Sleep(DatosHilo.Rand.Next(3000));

        Console.WriteLine(
            "Hilo primario se despierta. Esperando a que finalice el hilo
secundario ...");
        datos.Finaliza();
        datos.Hilo.Join();

        Console.WriteLine("Metodo main finalizo");
    }
}
```

Al ejecutar este programa se obtendrá una salida como la que se muestra en la Figura 8 - 11.



```
C:\WINNT\system32\cmd.exe
Hilo primario se va a dormir ...
Hilo2: Inicio de trabajo 1
Hilo primario se despierta. Esperando a que finalice el hilo secundario ...
Hilo2: Finalizo trabajo 1
Hilo2: Finalizo el hilo
Metodo main finalizo
Presione una tecla para continuar . . . _
```

Figura 8 - 11 Salida del ejemplo de Join en C#

Trabajo con Hilos en Java

Se debe seguir los siguientes pasos:

38. Definir una nueva clase que derive de *Thread*.
39. Sobrescribir el método run.
40. Crear e inicializar variables de la nueva clase de hilo.
41. Arrancar la ejecución de los hilos llamando al método start.

El siguiente programa muestra un ejemplo sencillo de creación y uso de un hilo.

```
class MiHilo extends Thread
{
    private int sleepTime;

    // El constructor asigna un nombre al hilo
    // llamando al constructor apropiado de Thread
    public MiHilo( String name )
    {
        super( name );
        // Calculo un tiempo aleatorio para dormir entre 0 y 5 segundos
    }
}
```

```
        sleepTime = (int) ( Math.random() * 5000 );
        // Imprimo en la salida estándar los datos
        // del nuevo hilo creado
        System.out.println( "Nombre: " + getName() +
            "; tiempo a dormir: " + sleepTime );
    }

    // Código de ejecución del hilo
    public void run()
    {
        System.out.println( getName() + " going to sleep" );

        // pongo a dormir el hilo
        try
        {
            sleep( sleepTime );
        }
        catch ( InterruptedException exception )
        {
            System.out.println( exception.toString() );
        }

        // Indico que el hilo finalizo su trabajo
        System.out.println( getName() + " done sleeping" );
    }
}

public class Ejemplo1
{
    public static void main( String args[] )
    {
        MiHilo hilo1, hilo2, hilo3, hilo4;

        hilo1 = new MiHilo( "Hilo1" );
        hilo2 = new MiHilo( "Hilo2" );
        hilo3 = new MiHilo( "Hilo3" );
        hilo4 = new MiHilo( "Hilo4" );

        System.out.println( "Arrancando los hilos ..." );

        hilo1.start();
        hilo2.start();
        hilo3.start();
        hilo4.start();

        System.out.println( "Hilos arrancados\n" );
    }
}
```

Al correr el programa la salida será parecida a la mostrada en la Figura 8 - 12.

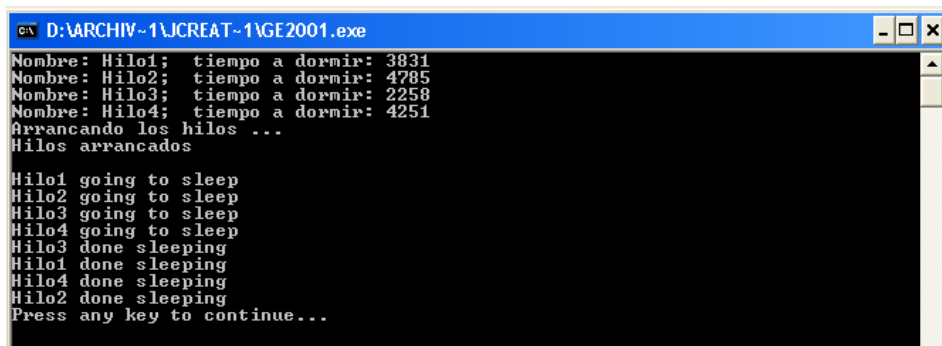
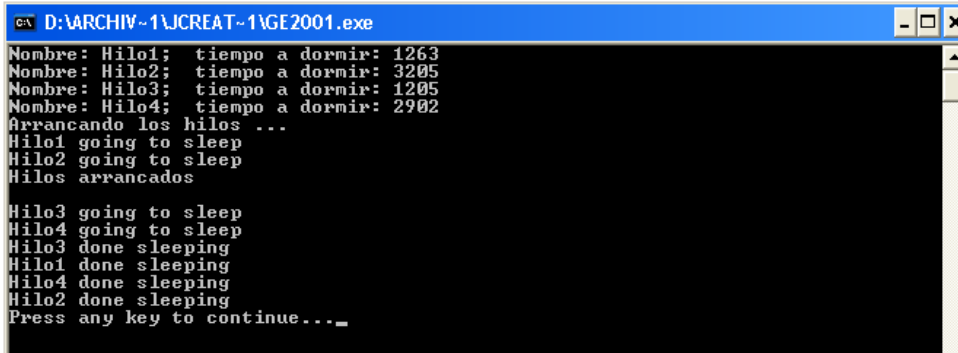


Figura 8 - 12 Salida del ejemplo de hilos en Java

Note que el hilo primario, correspondiente al método *main*, termina su trabajo antes de que cualquier hilo comience a ejecutarse. Sin embargo, esto no necesariamente es así. Otra ejecución podría arrojar lo mostrado en la Figura 8 - 13.



```
Nombre: Hilo1; tiempo a dormir: 1263
Nombre: Hilo2; tiempo a dormir: 3205
Nombre: Hilo3; tiempo a dormir: 1205
Nombre: Hilo4; tiempo a dormir: 2902
Arrancando los hilos ...
Hilo1 going to sleep
Hilo2 going to sleep
Hilos arrancados
Hilo3 going to sleep
Hilo4 going to sleep
Hilo3 done sleeping
Hilo1 done sleeping
Hilo4 done sleeping
Hilo2 done sleeping
Press any key to continue....
```

Figura 8 - 13 Segunda salida del ejemplo de hilos en Java

Note que la finalización del hilo primario no finaliza el programa. El programa finaliza cuando todos los hilos creados finalizan.

Si bien en ambos ejemplos los hilos inician su trabajo en orden, mostrando su mensaje *going to sleep*, esto no necesariamente es así, por lo que no podemos asumir que el orden en que se arrancan los hilos será el orden en que comiencen a ejecutarse. En general, el orden de ejecución de los hilos es una decisión del sistema operativo en base a las políticas que implemente su *planificador*.

Note que la llamada al método *sleep* se coloca dentro de un bloque *try*, dado que este método puede disparar la excepción *InterruptedException*. Como se explicó en la sección anterior, un hilo *bloqueado*, *dormido* o *esperando* puede ser interrumpido y sacado de este estado. Debido a esto, las llamadas a los métodos que producen estos estados disparan excepciones adecuadas para los casos en que estos estados son interrumpidos. El manejo de estos cambios de estado y otras acciones con hilos utilizan ampliamente las interrupciones para su control. Estas excepciones son *No-RunTime*, por lo que el no manejarlas (con bloques *try* o indicándolas en la lista *throws* del método) provocarían un error en tiempo de compilación.

El siguiente programa muestra la interrupción de los hilos.

```
class MiHilo extends Thread
{
    private int sleepTime;

    // El constructor asigna un nombre al hilo
    // llamando al constructor apropiado de Thread
    public MiHilo( String name )
    {
        super( name );

        // Calculo un tiempo aleatorio para dormir entre 0 y 5 segundos
        sleepTime = (int) ( Math.random() * 5000 );
        // Imprimo en la salida estándar los datos
        // del nuevo hilo creado
        System.out.println( "Nombre: " + getName() +
            "; tiempo a dormir: " + sleepTime );
    }

    // Código de ejecución del hilo
    public void run()
    {
        // pongo a dormir el hilo
        try
        {
            System.out.println( getName() + " going to sleep" );

            Thread.sleep( sleepTime );
        }
    }
}
```

```
        // Indico que el hilo finalizo su trabajo satisfactoriamente
        System.out.println( getName() + " done sleeping" );
    }
    catch ( InterruptedException exception )
    {
        System.out.println( exception.toString() );
    }
}

}

public class Ejemplo2
{
    public static void main( String args[] ) throws InterruptedException
    {
        MiHilo hilo1, hilo2, hilo3, hilo4;

        hilo1 = new MiHilo( "Hilo1" );
        hilo2 = new MiHilo( "Hilo2" );
        hilo3 = new MiHilo( "Hilo3" );
        hilo4 = new MiHilo( "Hilo4" );

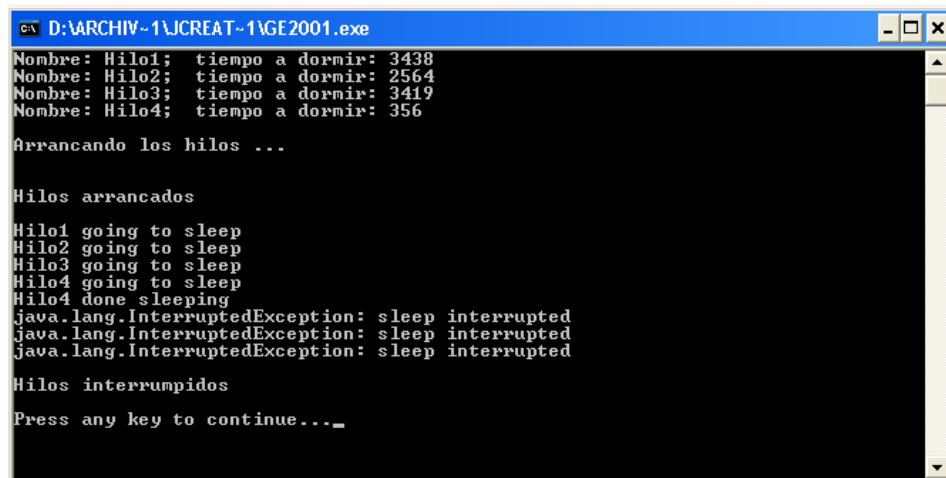
        System.out.println( "\nArrancando los hilos ...\n" );

        hilo1.start();
        hilo2.start();
        hilo3.start();
        hilo4.start();

        System.out.println( "\nHilos arrancados\n" );
        // Les doy un plazo de tiempo para que completen su trabajo
        Thread hiloPrimario = Thread.currentThread();
        hiloPrimario.sleep( 2500 );

        // Interrumpo los hilos
        hilo1.interrupt();
        hilo2.interrupt();
        hilo3.interrupt();
        hilo4.interrupt();
        System.out.println( "\nHilos interrumpidos\n" );
    }
}
```

Al correr el programa la salida será la mostrada en la figura 8.14.



```
D:\ARCHIV~1\JCREAT~1\GE2001.exe
Nombre: Hilo1; tiempo a dormir: 3438
Nombre: Hilo2; tiempo a dormir: 2564
Nombre: Hilo3; tiempo a dormir: 3419
Nombre: Hilo4; tiempo a dormir: 356

Arrancando los hilos ...

Hilos arrancados
Hilo1 going to sleep
Hilo2 going to sleep
Hilo3 going to sleep
Hilo4 going to sleep
Hilo4 done sleeping
java.lang.InterruptedException: sleep interrupted
java.lang.InterruptedException: sleep interrupted
java.lang.InterruptedException: sleep interrupted

Hilos interrumpidos
Press any key to continue....
```

Figura 8 - 14 Ejemplo de interrupción de hilos en Java

Note que sólo el hilo “Hilo4” logra finalizar su trabajo antes de que sea interrumpido, los demás hilos son interrumpidos antes. En este caso en particular, dado que no se crea ningún recurso

dentro del bloque *try* que no pueda ser liberado automáticamente por el intérprete (o alguna otra acción importante), no es necesario definir un bloque *finally* que se encargue de ello.

Note que el método *main* obtiene una referencia al hilo que lo ejecuta utilizando el método estático *currentThread* de la clase *Thread*. Con esta referencia llama al método *sleep*. Sin embargo, dado que este método es estático, también se pudo utilizar:

```
Thread.sleep( 2500 );
```

El tiempo que se pasa como parámetro al método *sleep*, y a otros métodos de *Thread* que requieran una especificación de tiempo, se dan en milisegundos.

Note que el método *main* no desea manejar la interrupción que pueda disparar *sleep*, por lo que declara dicha interrupción en la lista *throws* de su declaración.

Sincronización

Java utiliza objetos **monitores** para la sincronización de hilos. Un objeto *monitor* es aquel que contiene métodos declarados con el modificador **synchronized**, sin importar de qué clase sea el objeto. Esto implica que cualquier objeto es susceptible de ser utilizado como un *monitor*.

Cuando un método *synchronized* de un objeto es llamado desde un hilo, ningún otro hilo puede acceder a éste u otro método *synchronized* del mismo objeto. Cuando un hilo entra a ejecutar un método *synchronized* de un objeto, el objeto queda bloqueado. Esto implica que si un hilo llama a un método *synchronized* de un objeto bloqueado, quedará en estado de espera hasta que éste se desbloquee. Cuando un hilo sale de la ejecución de un método *synchronized* de un objeto, éste se desbloquea, permitiendo que otros hilos accedan a él. Si hubiese algún hilo esperando una llamada pendiente a un método *synchronized* de este objeto, el hilo pasa al estado *listo*, de forma que cuando entre al estado *en ejecución* pueda ejecutar el código del método.

En resumen, un hilo que accede a ejecutar un método *synchronized* de un objeto, bloquea a dicho objeto para el resto de los hilos. Sólo un método *synchronized* de un objeto puede ser ejecutado a la vez por un hilo. Sin embargo, el resto de hilos sí pueden acceder a los métodos de dicho objeto que no sean *synchronized*. Esto permite una fácil sincronización en el acceso a los recursos del objeto monitor. Esta técnica es equivalente al uso de *secciones críticas* en la programación en C para Windows.

El siguiente ejemplo muestra el uso de un *monitor*.

```
class Productor
{
    private int iTrabajo = 0;

    public synchronized int SiguienteTrabajo()
    {
        iTrabajo++;

        // simulo un tiempo de demora en la entrega del trabajo
        try
        {
            Thread.sleep( ( int ) ( Math.random() * 1000 ) );
        }
        catch( InterruptedException e )
        {
        }

        return iTrabajo;
    }
}

class HiloConsumidor extends Thread
{

```

```
Productor p;

HiloConsumidor( Productor p )
{
    this.p = p;
}

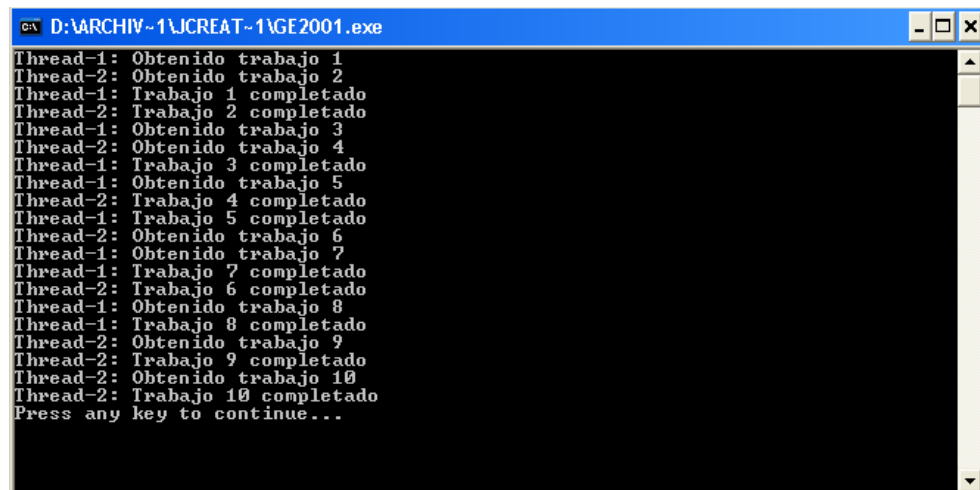
public void run()
{
    for( int i = 0; i < 5; i++ )
    {
        int iTrabajo = p.SiguienteTrabajo();
        System.out.println( getName() + ": Obtenido trabajo " + iTrabajo );

        try
        {
            // simulo un tiempo de trabajo
            sleep( ( int )( Math.random() * 1000 ) );

            // reporto que se finalizo el trabajo
            System.out.println( getName() + ": Trabajo " + iTrabajo +
                               " completado" );
        }
        catch( InterruptedException e )
        {
            System.out.println( getName() + ": Trabajo " + iTrabajo +
                               " interrumpido" );
        }
    }
}

public class Ejemplo3
{
    public static void main( String args[] )
    {
        Productor p = new Productor();
        HiloConsumidor hilo1 = new HiloConsumidor( p );
        HiloConsumidor hilo2 = new HiloConsumidor( p );
        hilo1.start();
        hilo2.start();
    }
}
```

Al correr el programa la salida será la mostrada en la Figura 8 - 15.



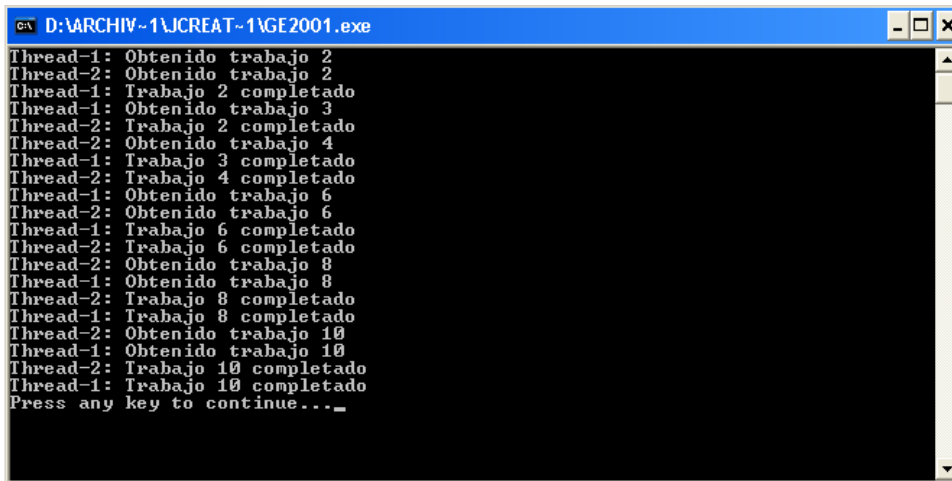
```
D:\ARCHIV~1\JCREAT~1\GE2001.exe
Thread-1: Obtenido trabajo 1
Thread-2: Obtenido trabajo 2
Thread-1: Trabajo 1 completado
Thread-2: Trabajo 2 completado
Thread-1: Obtenido trabajo 3
Thread-2: Obtenido trabajo 4
Thread-1: Trabajo 3 completado
Thread-1: Obtenido trabajo 5
Thread-2: Trabajo 4 completado
Thread-1: Trabajo 5 completado
Thread-2: Obtenido trabajo 6
Thread-1: Obtenido trabajo 7
Thread-1: Trabajo 7 completado
Thread-2: Trabajo 6 completado
Thread-1: Obtenido trabajo 8
Thread-1: Trabajo 8 completado
Thread-2: Obtenido trabajo 9
Thread-2: Trabajo 9 completado
Thread-2: Obtenido trabajo 10
Thread-2: Trabajo 10 completado
Press any key to continue...
```

Figura 8 - 15 Salida del ejemplo de sincronización en Java

Este programa simula un proceso “productor-consumidor”. El productor devuelve un trabajo a demanda al consumidor que se lo solicita. Cuando el productor calcula un nuevo trabajo le toma

un tiempo aleatorio entregarlo. Sin embargo, dado que el método de entrega de un trabajo, `SiguienteTrabajo`, es declarado `synchronized`, sólo un hilo podrá obtener un trabajo a la vez, por lo que no habrá pérdida de trabajos.

Para ejemplificar mejor esta sincronización, la salida en la Figura 8 - 16 muestra lo que podría ocurrir si el método `SiguienteTrabajo` no se sincronizara:



```
D:\ARCHIV-1\JCREAT-1\GE2001.exe
Thread-1: Obtenido trabajo 2
Thread-2: Obtenido trabajo 2
Thread-1: Trabajo 2 completado
Thread-1: Obtenido trabajo 3
Thread-2: Trabajo 2 completado
Thread-2: Obtenido trabajo 4
Thread-1: Trabajo 3 completado
Thread-2: Trabajo 4 completado
Thread-1: Obtenido trabajo 6
Thread-2: Obtenido trabajo 6
Thread-1: Trabajo 6 completado
Thread-2: Trabajo 6 completado
Thread-2: Obtenido trabajo 8
Thread-1: Obtenido trabajo 8
Thread-2: Trabajo 8 completado
Thread-1: Trabajo 8 completado
Thread-2: Obtenido trabajo 10
Thread-1: Obtenido trabajo 10
Thread-2: Trabajo 10 completado
Thread-1: Trabajo 10 completado
Press any key to continue...
```

Figura 8 - 16 Salida del ejemplo sin sincronización en Java

Note como algunos trabajos se pierden y otros se realizan dos veces. Esto se debe a que la instrucción:

```
iTrabajo++;
```

Es ejecutada, en ciertas ocasiones, por más de un hilo antes de que el método `SiguienteTrabajo` retorne.

Note el nombre que asigna el constructor por defecto de `Thread` a los hilos, dado que no se ha llamado a algún constructor con parámetros de `Thread` para darle uno.

Ahora bien, que pasaría en el caso de tener un productor que produce indiferentemente de que existan consumidores que consuman dichos trabajos, y a la vez consumidores que consuman indiferentemente de que exista algo que consumir. Si el productor elimina un trabajo para crear uno nuevo antes de que el anterior sea consumido, se perderá dicho trabajo. Si el consumidor consume un trabajo antes de que se haya producido uno nuevo, un trabajo será realizado dos o más veces. En este caso, ambas labores, la de producir y la de consumir deben de sincronizarse una con otra, esto es, el productor no debe de seguir produciendo mientras que no se haya consumido el trabajo anterior y el consumidor debe de esperar a que se produzca un nuevo trabajo antes de consumirlo.

En este tipo de situación, la sincronización utilizando métodos `synchronized` por sí sola no es suficiente. Se necesita que el hilo productor espere y notifique al consumidor, y éste a su vez espere y notifique al productor.

Para esto, la clase `Object`, de la que hereda cualquier objeto que se utilice como monitor, provee los métodos `wait`, `notify` y `notifyAll`. Estos métodos sólo pueden ejecutarse dentro de un método declarado `synchronized`, o en algún método llamado desde uno declarado `synchronized`. El no hacerlo así provocaría un error en tiempo de ejecución. Esto significa que estos métodos sólo pueden ser ejecutados desde un objeto monitor, por el hilo que haya bloqueado a este objeto.

El método *wait* coloca al hilo en estado de espera y desbloquea al objeto *monitor*. El hilo saldrá de dicho estado cuando otro hilo acceda a algún método *synchronized* del mismo objeto *monitor* y llame a *notify* o *notifyAll*. En ese momento el hilo será colocado en estado *listo* y competirá con el resto de hilos que intentan acceder a algún método *synchronized* de dicho objeto *monitor* (o que también hayan salido del estado de espera) para bloquearlo y ejecutar su código.

El método *notify* saca al primero de los hilos que esté en la lista de espera del objeto *monitor*, del estado *esperando* y lo coloca en estado *listo*. De esta forma el hilo vuelve a entrar a competir por el acceso al objeto *monitor*.

El método *notifyAll* saca a todos los hilos que esté en la lista de espera del objeto *monitor*, del estado *esperando* y los coloca en estado *listo*.

Tome en cuenta que sólo un hilo puede acceder a ejecutar el código de un método *synchronized* de un objeto *monitor*. Luego, aún cuando exista más de un hilo esperando a ejecutar un método *synchronized* y otros tantos que hayan sido sacados del estado de espera para el mismo objeto *monitor*, el primero que entre a en ejecución bloqueará al objeto *monitor*, lo que colocará a todos los demás hilos en estado *esperando*. Cuando el hilo ganador termine la ejecución de dicho método, todos los hilos que están esperando volverán al estado *listo*, de forma que vuelvan a competir por el acceso al objeto *monitor*.

El siguiente ejemplo muestra el uso de los métodos *wait* y *notify*.

```
class Trabajo
{
    private int NroTrabajo = -1;
    private boolean PuedoCrear = true; // determina si se puede crear o consumir

    public synchronized void CrearNuevo( int NroTrabajo )
    {
        while ( !PuedoCrear ) { // aún no se puede producir
            try {
                wait();
            }
            catch ( InterruptedException e ) {
            }
        }

        System.out.println( Thread.currentThread().getName() +
            " creo el trabajo " + NroTrabajo );

        this.NroTrabajo = NroTrabajo;

        PuedoCrear = false;
        notify(); // notifico que existe un trabajo listo
    }

    public synchronized int ObtenerTrabajo()
    {
        while ( PuedoCrear ) { // aún no se puede obtener
            try {
                wait();
            }
            catch ( InterruptedException e ) {
            }
        }

        PuedoCrear = true;
        notify(); // notifico que ya se obtuvo el trabajo actual

        System.out.println( Thread.currentThread().getName() +
            " obtuvo el trabajo " + NroTrabajo );

        return NroTrabajo;
    }
}
```

```
class Productor extends Thread
{
    private Trabajo t;

    public Productor( Trabajo t )
    {
        this.t = t;
    }

    public void run()
    {
        for ( int Contador = 1; Contador <= 5; Contador++ )
        {
            // simulo un tiempo de demora aleatorio
            // en la creación del nuevo trabajo
            try
            {
                sleep( (int) ( Math.random() * 1000 ) );
            }
            catch( InterruptedException e )
            {
            }

            t.CrearNuevo( Contador );
        }

        System.out.println( getName() + " finalizo la produccion de trabajos" );
    }
}

class Consumidor extends Thread
{
    private Trabajo t;

    public Consumidor ( Trabajo t )
    {
        this.t = t;
    }

    public void run()
    {
        int iTrabajo;
        do {
            iTrabajo = t.ObtenerTrabajo();

            // simulo un tiempo de demora aleatorio
            // en el procesamiento del trabajo
            try
            {
                sleep( (int) ( Math.random() * 1000 ) );
            }
            catch( InterruptedException e )
            {
            }
        } while ( iTrabajo != 5 );

        System.err.println( getName() + " termino de procesar los trabajos" );
    }
}

public class Ejemplo4
{
    public static void main( String args[] )
    {
        Trabajo t = new Trabajo();
        Productor p = new Productor( t );
        Consumidor c = new Consumidor( t );

        p.start();
        c.start();
    }
}
```

Al correr el programa la salida será la mostrada en la figura 8.17.

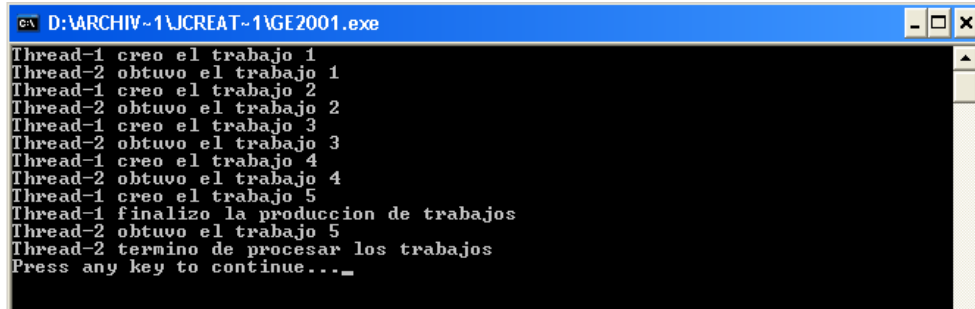


Figura 8 - 17 Salida del ejemplo de uso de “wait” y “notify” en Java

Note que a pesar de que tanto el productor como el consumidor tratan de crear y obtener respectivamente un trabajo en distinto orden, éstos son producidos y consumidos en el orden en que son creados gracias a la sincronización mediante *wait* y *notify*.

Note además que en algún momento el productor, por ejemplo, podría utilizar su propia notificación en su llamada a *wait*. Sin embargo, dado que se utiliza un flag, *PuedoCrear*, el hilo del productor nunca saldrá del bucle de espera hasta que efectivamente el trabajo haya sido retirado por el consumidor.

En ocasiones es necesario que un hilo verifique que otro haya finalizado para poder continuar con su trabajo. En este caso se utiliza el método *join*. Un hilo que llame al método *join* utilizando la referencia de otro hilo, pasará al estado *esperando* hasta que dicho hilo termine, esto es, pase al estado *finalizado*. Cuando esto sucede, el hilo que espera pasa al estado *listo*.

El siguiente ejemplo muestra el uso del método *join*.

```
class MiHilo extends Thread
{
    private boolean bFinalizar = false;

    public void Finaliza()
    {
        bFinalizar = true;
    }

    public void run()
    {
        int iTrabajo = 0;

        while( !bFinalizar )
        {
            iTrabajo++;

            // reporto que se inicio el trabajo
            System.out.println( getName() + ": Inicio de trabajo " + iTrabajo );

            // simulo un tiempo de trabajo
            try
            {
                sleep( ( int )( Math.random() * 3000 ) );
            }
            catch( InterruptedException e )
            {
            }

            // reporto que se finalizo el trabajo
            System.out.println( getName() + ": Finalizo trabajo " + iTrabajo );
        }
        // reporto que finalizo el hilo
        System.out.println( getName() + ": Finalizo el hilo" );
    }
}
```



```
}  
}  
  
public class Ejemplo5  
{  
    public static void main( String args[] )  
    {  
        MiHilo hilo = new MiHilo();  
        hilo.start();  
  
        try  
        {  
            // simulo un tiempo de trabajo  
            Thread.sleep( ( int )( Math.random() * 3000 ) );  
  
            // le aviso al hilo que finalice y lo espero  
            hilo.Finaliza();  
            hilo.join();  
        }  
        catch( InterruptedException e )  
        {  
        }  
  
        // reporto el fin  
        System.out.println( "Metodo main finalizo" );  
    }  
}
```

Al correr el programa se tendrá una salida como la mostrada en la Figura 8 - 18.

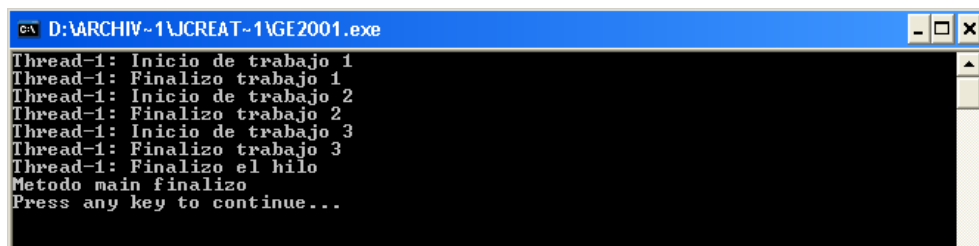


Figura 8 - 18 Salida del ejemplo de uso de "join" en Java

Note que el hilo primario, el que ejecuta al método *main*, espera a que el nuevo hilo creado finalice para continuar. Al igual que *sleep*, *join* también puede ser interrumpido y arrojar una excepción, por lo que se le ha colocado dentro del bloque *try*.

Note también que no es necesario definir un constructor en una clase hilo, dado que *Thread* posee un constructor por defecto. El único método que siempre debe de implementarse es *run*.

Aunque todos los ejemplos mostrados son aplicaciones de consola, el uso de hilos se aplica a todo tipo de programa en Java.

Referencias

- Libro “Programación Orientada a Objetos con C++”; autor Francisco Javier Cevallos; editorial Alfaomega; edición 3ra; año 2004; capítulo 6.
 - Libro “Essential C# 2.0”; autor Mark Michaelis; editorial Addison-Wesley Professional; edición 1ra, año 2006.
 - Libro “C++ Como Programar”; autores Harvey M. Deitel y Paul J. Deitel; editorial Pearson; edición 4ta; año 2003; capítulos 11 y 21.
 - Libro en-línea gratuito “Introduction to Programming Using Java”; autor David J. Eck; versión 4.0; fecha julio 2002; sección 12.1.
 - Documentación de Microsoft Visual Studio 2005.
 - Documentación de la Api de Java.
<http://www.faqs.org/docs/javap/c12/s1.html>
 - Página Web “Introduction to C# Generics Tutorial”.
http://www.deitel.com/articles/csharp_tutorials/20051111/Intro_CSharpGenerics.html
 - Página Web “The Collections Framework”.
<http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html>
 - Página Web “Generics”; tutorial de Sun.
<http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html>
 - Página Web “Generics in C#, Java, and C++: A Conversation with Anders Hejlsberg, Part VII”; artículo; autores Bill Venners y Bruce Eckel; fecha enero 2004.
<http://www.artima.com/intv/generics.html>
 - Página Web “C++ Notes: Table of Contents”.
<http://www.fredosaurus.com/notes-cpp/index.html>
 - Página Web “A modest STL tutorial”.
<http://www.cs.brown.edu/people/jak/proglang/cpp/stltut/tut.html>
 - Página Web “The C++ Standard Library - A Tutorial and Reference”.
<http://www.josuttis.com/libbook/idx.html>
 - Página Web “Overview of Generics in the .NET Framework”.
<http://windowssdk.msdn.microsoft.com/en-us/library/ms172193.aspx>
 - Página Web “Generic Programming”; autor David R. Musser.
<http://www.cs.rpi.edu/~musser/gp/>
-