

Edit 

GAMBAS

Programación visual con Software Libre



Autores

Daniel Campos
José Luis Redrejo

Prólogo
Benoît Minisini

GAMBAS

Gambas Almost Means Basic



INFORMACIÓN

Tienes en tus manos el libro **COMPLETO** de “GAMBAS programación visual con Software Libre” escrito por Daniel Campos y José Luis Redrejo bajo licencia libre.

Al libro electrónico original le faltan las páginas 34, 35, 58, 59, 104, 105, 166, 167, 190, 191, 216, 260 y 261.

Hemos escaneado e insertado estas páginas faltantes en este nuevo libro electrónico, creando así el libro electrónico completo que todo “gambero” debería leer.

Hemos modificado el libro original, añadido las páginas faltantes, unificado todos los capítulos en un único fichero y lo volvemos a hacer público conservando la misma licencia libre.

Autores:

DANIEL CAMPOS
JOSE LUIS REDREJO

Responsable editorial:
SORAYA MUÑOZ

Responsable de Marketing y Comunicación:
ÁLVARO GARCÍA

Edición:
MIRIAM MONTERO

Diseno de portada:
PICTOGRAMA

Maquetación:
ÁNGEL ESTRADA

ISBN: 978-84-934371-2-3
Depósito Legal: M-41116-2005

Edita:
©EDIT LIN EDITORIAL, S.L., 2005
Avda. Portugal, 85- local
28011 Madrid (España)
Tel.: 91 577 03 55
Fax: 91 577 06 18

www.librosdelinux.com
info@librosdelinux.com

LICENCIA

Se permite la copia y distribución de la totalidad o parte de esta obra sin ánimo de lucro. Toda copia total o parcial deberá citar expresamente el nombre del autor, nombre de la editorial e incluir esta misma licencia añadiendo, si es copia literal, la mención "copia literal".

Se autoriza la modificación y traducción de la obra sin ánimo de lucro siempre que se haga constar en la obra resultante de la modificación el nombre de la obra originaria, el autor de la obra originaria y el nombre de la editorial. La obra resultante también será libremente reproducida, distribuida, comunicada al público y transformada en términos similares a los expresados en esta licencia.

Impreso en España (Printed in Spain)

*Este libro ha sido realizado con Software Libre, concretamente con: OpenOffice.org, Evolution, Mozilla, GIMP

PRÓLOGO

La escena se desarrolla en algún lugar en el campo, en verano...

— ¡Abuelo, no te duermas! No has acabado de contarme la historia del ordenador...

— ¿Eh?... ah... sí... bueno, bueno...

El abuelo bostezó y pensó en lo que iba a decir. Ya había anochecido y sólo se oía el canto de los grillos y el rechinar de la butaca.

“Cuando mi hermano compró su primer ordenador personal, estaba compuesto de un lector de cintas, de algunos KB de memoria -no recuerdo exactamente cuántos- y de un teclado mecánico, cosa poco frecuente en aquella época. Pero, sobre todo, disponía de un lenguaje BASIC completo y de un manual que intentaba enseñar la programación paso a paso.

Algunos años más tarde fui yo el que me compré un ordenador. Los modelos eran más potentes, pero volví a encontrar el lenguaje BASIC y la posibilidad de sacar el máximo rendimiento.

En aquella época, las revistas hablaban de programación, la gente programaba e intercambiaba programas. Algunos copiaban programas que no habían creado y se les llamaba piratas... No, no, todos no tenían un parche negro en un ojo y un garfio en lugar de una mano.

Después llegó la época del IBM PC y poco a poco las cosas empezaron a cambiar. Al principio era una enorme caja gris, fea y ruidosa. Pero aumentó en potencia mientras más potencia adquiría, menos posibilidad había de sacarle el máximo rendimiento.

Gracias a un puñado de futuros millonarios teníamos pantallas más grandes, más colores, más sonido, más memoria, más capacidad de almacenamiento. Pero, al mismo tiempo, se chapuceaban los sistemas operativos, se despreciaban los principios elementales de seguridad, se frenaban las aplicaciones, se bloqueaban los formatos de datos, se ocultaba y se mentía sobre el funcionamiento interno del conjunto.

El usuario se convertía en un consumidor como cualquier otro. Campañas de publicidad gigantescas le conducían a comprar una especie de caja negra de la que seguramente había que controlar al máximo la utilización. A pesar de todo, la gente continuaba intercambiando y los piratas pirateando.

6

En cuanto a los que realmente eran capaces de sacar el máximo rendimiento a los ordenadores, tenían que comprarlos y después encerrarlos tras las puertas de campus o de una cláusula de no divulgación. Si no hacían esto, tenían que dejar de programar.

Afortunadamente, las cosas no sucedieron exactamente como estaba previsto y gracias al nacimiento de Internet, y a la mezcla de talentos de numerosos programadores, a menudo voluntarios, idealistas y pragmáticos al mismo tiempo, se impuso un movimiento de liberación y nació, entre otros, el sistema operativo GNU/Linux.

Pero esto es otra historia, te la contaré mañana..."

La idea de desarrollar Gambas sobre Linux en casa vino principalmente como reacción a la obligación de utilizar Visual Basic sobre Windows en el trabajo.

Cualquier usuario con un poco de idea, cualquier empresa, cualquier administración, necesitan sacar el mayor rendimiento posible a su ordenador, es decir, hacer

él lo que quieran hacer sin tener las competencias necesarias requeridas por los lenguajes de programación, ni el tiempo para adquirirlas.

Con un lenguaje incoherente, incompleto y repleto de fallos, Visual Basic ha respondido, a pesar de todo, a estas necesidades. Pero sus creadores, encerrados en su torre de marfil, lo han ido poco a poco abandonando. ¿Había quizás desprecio hacia esos usuarios que tenían la pretensión de sacar, como a ellos les pareciera, el máximo rendimiento de su herramienta de trabajo?

A pesar de que forma parte de la gran familia BASIC y que la interfaz de los programas se dibuja con el ratón, Gambas no tiene ningún otro punto en común con Visual Basic y es totalmente incompatible con él. Sin embargo, mi prioridad es que pueda satisfacer la misma necesidad: conseguir de la manera más simple posible sacar el máximo rendimiento de todas las funciones del sistema operativo subyacente, GNU/Linux en este caso preciso.

Gambas se inspira, sobre todo, en Java: se trata de un lenguaje orientado a objetos, procedural e interpretado. Pero su intérprete es mucho más rudimentario: no hay compilación *just-in-time*, no hay *garbage collector*. Sin embargo, he cuidado mucho su simplicidad, su coherencia, su fiabilidad y sus facultades de internacionalización.

Gambas es extensible. Es posible escribir componentes que añaden al lenguaje clases suplementarias. Estos componentes ofrecen, en general, el acceso a librerías específicas: QT, GTK+, SDL, OpenGL, y así sucesivamente...

Gambas ofrece un entorno de desarrollo integrado, moderno, que está, él mismo, escrito en Gambas. Es ése principalmente el que me permite probar el lenguaje :-). Además, es totalmente posible prescindir de él.

Finalmente, y nunca insistiré lo suficiente, Gambas no es un lenguaje creado por programadores profesionales para programadores no profesionales. Yo soy su primer usuario y otra de las razones que me ha llevado a crearlo es que todavía no deseo aprender en serio Perl o Python :-)

La primera versión estable de Gambas salió el 1 de enero de 2005, pero las cosas pondrán realmente interesantes con la segunda versión. En el momento en el que escribo estas líneas, la versión de desarrollo ofrece la posibilidad de escribir componentes directamente en Gambas, sin necesidad de dominar C o C++. Más tarde, esta posibilidad permitirá al entorno de desarrollo diseñar algo más que formularios gráficos: tablas, hojas HTML, etc.

Es difícil prever de manera precisa cuándo saldrá esta segunda versión, e incluye más en general, cómo evolucionará Gambas en el futuro. El trabajo que queda por hacer es todavía enorme y las cosas dependen mucho de los usuarios.

En cualquier caso, espero que tendréis tanto gusto en utilizarlo como yo lo he tenido en crearlo y que participaréis en su evolución. Y si, como para otros, Gambas es el factor que os lleva a utilizar por fin el Software Libre, espero que sentiréis como yo el respiro que produce esta libertad.

ÍNDICE

CAPÍTULO 1: ¿QUÉ ES GAMBAS?	17
■■■■■ I. 1 El lenguaje BASIC: su historia	18
■■■■■ I. 2 Un entorno libre	21
■■■■■ I. 3 Elementos de Gambas	23
■■■■■ I. 4 Como obtenerlo	24
■■■■■ I. 5 Compilación y dependencias	26
■■■■■ I. 6 Familiarizarse con el IDE	27
■■■■■■ El primer ejemplo	29
■■■■■■ Mejor con un ejemplo gráfico	32
■■■■■■ Un poco de magia	35
■■■■■ I. 7 Sistema de componentes	36
CAPÍTULO 2: PROGRAMACIÓN BÁSICA	41
■■■■■ 2. 1 Organización de un proyecto de Gambas	42
■■■■■■ Declaración de variables	42
■■■■■■ Subrutinas y funciones	45

■■■■■ 2.2 Tipos de datos	49
□□□□□ Conversión de tipos	50
□□□□□ Matrices	52
■■■■■ 2.3 Operaciones matemáticas	54
□□□□□ Operaciones lógicas	56
■■■■■ 2.4 Manejo de cadenas	56
■■■■■ 2.5 Control de flujo	60
□□□□□ IF... THEN... ELSE	60
□□□□□ Select	62
□□□□□ FOR	63
□□□□□ WHILE y REPEAT	64
□□□□□ Depuración en el IDE de Gambas	66
■■■■■ 2.6 Entrada y salida de ficheros	68
■■■■■ 2.7 Control de errores	72
■■■■■ 2.8 Programación orientada a objetos con Gambas	73
■■■■■ 2.9 Propiedades, Métodos y Eventos	79
CAPÍTULO 3: LA INTERFAZ GRÁFICA	81
■■■■■ 3.1 Concepto	81
□□□□□ Partiendo de la consola	84
□□□□□ El entorno de desarrollo	85
■■■■■ 3.2 Manejo básico de los controles	87
□□□□□ Posición y tamaño	87
□□□□□ Visibilidad	89
□□□□□ Textos relacionados	89
□□□□□ Colores	90
□□□□□ Ratón	92
□□□□□ Teclado	95
■■■■■ 3.3 Galería de controles	96
□□□□□ Controles básicos	96

□□□□□	Otros controles básicos misceláneos	99
□□□□□	Listas de datos	100
□□□□□	Otros controles avanzados	101
■■■■■	3.4 Diálogos	101
□□□□□	La clase Message	101
□□□□□	La clase Dialog	104
□□□□□	Dialogos personalizados	106
■■■■■	3.5 Menús	111
■■■■■	3.6 Alineación de los controles	114
□□□□□	Propiedades de la alineación	114
□□□□□	Controles con alineación predefinida	117
□□□□□	Diseño de una aplicación que aprovecha este recurso	117
■■■■■	3.7 Introducción al dibujo de primitivas	120

CAPÍTULO 4: GESTIÓN DE PROCESOS 125

■■■■■	4.1 La ayuda ofrecida por otros programas	125
■■■■■	4.2 Gestión potente de procesos	126
■■■■■	4.3 EXEC	127
□□□□□	Palabra clave WAIT	128
□□□□□	El descriptor del proceso	130
□□□□□	Redirección con TO	133
□□□□□	Matar un proceso	134
□□□□□	Redirección de la salida estandar de errores	136
□□□□□	Redirección de la salida estandar	139
□□□□□	Evento Kill() y la propiedad Value	142
□□□□□	Redirección de la entrada estandar, el uso de CLOSE	146
□□□□□	Notas finales sobre el objeto Process	148
■■■■■	4.4 SHELL	148

CAPÍTULO 5: GESTIÓN DE BASES DE DATOS 151

■■■■■ 5.1 Sistemas de bases de datos	151
■■■■■ 5.2 Bases de datos y Gambas	153
■■■■■ 5.3 Gambas-database-manager. el gestor gráfico	154
□□□□□ Crear una base	154
□□□□□ Crear una tabla	158
□□□□□ Gestionar datos de una tabla	164
□□□□□ SQL	165
■■■■■ 5.4 Programación	167
□□□□□ Modelo de bases de datos	167
□□□□□ Conectándose por código	168
□□□□□ Consulta de datos	171
□□□□□ Borrar registros	174
□□□□□ Añadir registros	176
□□□□□ Modificar registros	180
■■■■■ 5.5 Otras características	190
□□□□□ Estructura de las tablas	190
□□□□□ Más utilidades del Gestor de Bases de Datos	192

CAPÍTULO 6: RED 195

■■■■■ 6.1 Conceptos	195
■■■■■ 6.2 Creando un servidor TCP	198
■■■■■ 6.3 Un cliente TCP	205
■■■■■ 6.4 Clientes y servidores locales	211
■■■■■ 6.5 UDP	213
■■■■■ 6.6 Resolución de nombres	216
■■■■■ 6.7 Protocolo HTTP	220
■■■■■ 6.8 Protocolo FTP	225

CAPÍTULO 7: XML 227

■■■■■ 7.1 Escritura con XmlWriter	230
■■■■■ 7.2 Lectura con XmlReader	238
□□□□□ Modelos de lectura	238
□□□□□ Planteamiento inicial	238
□□□□□ Un ejemplo de lectura	240
■■■■■ 7.3 XSLT	252
□□□□□ ¿Qué es XSLT?	252
□□□□□ Una plantilla de ejemplo	252
□□□□□ Transformando el documento con Gambas	254
■■■■■ 7.4 Acerca de XML-RPC	255

CAPÍTULO 8: HERENCIA 257

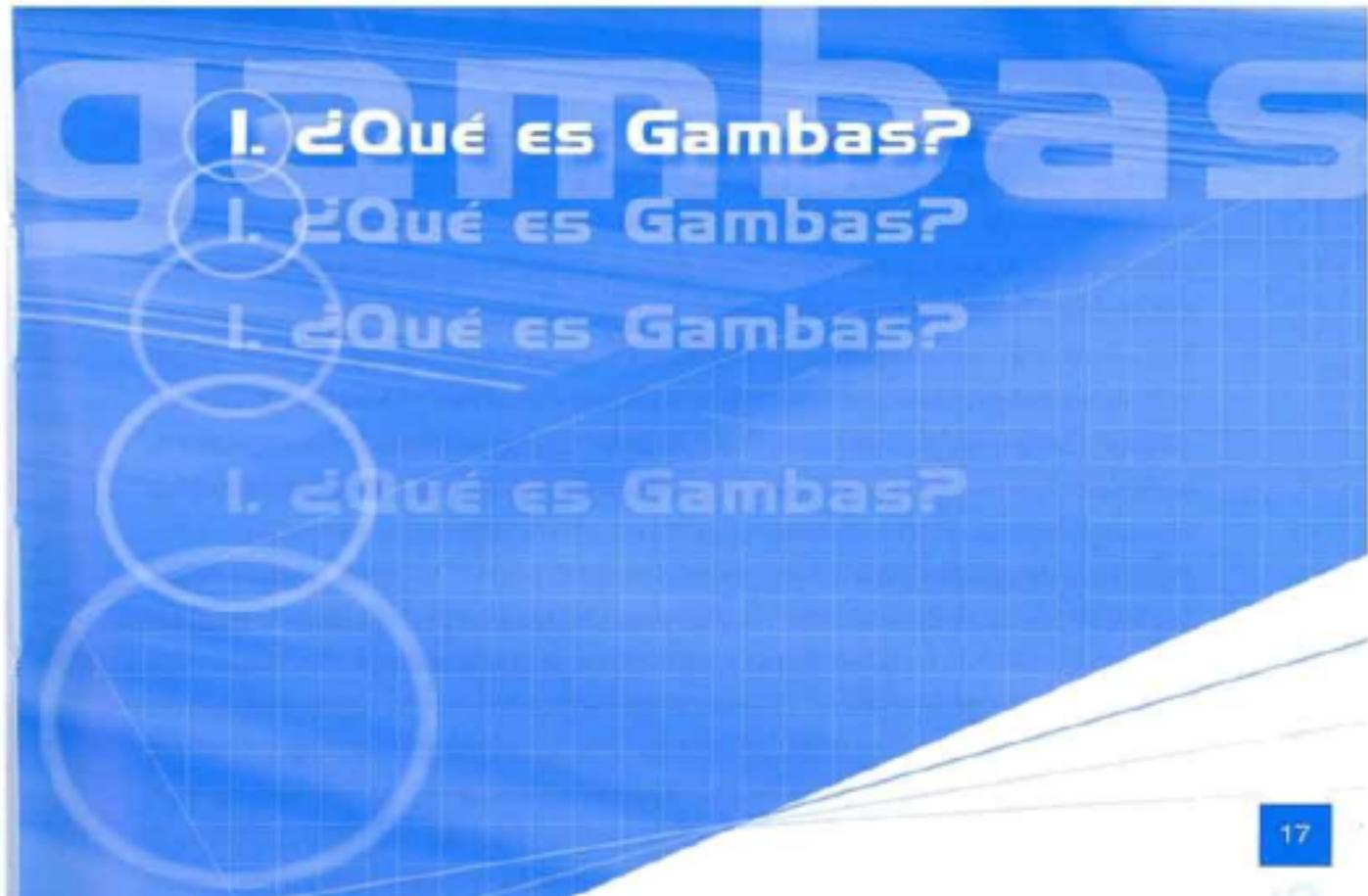
■■■■■ 8.1 Lenguajes orientados a objetos y herencia	257
■■■■■ 8.2 Conceptos necesarios	258
□□□□□ La clase padre	258
□□□□□ La clase hija. Palabra clave INHERITS	262
□□□□□ Extendiendo funcionalidades:	
Palabra clave SUPER	264
Modificando funcionalidades	265
Reemplazando métodos especiales:	
_New y _FREE	269
Limitaciones	272
■■■■■ 8.3 Crear un nuevo control con Gambas	273
□□□□□ Planteando el código	273
□□□□□ Implementación básica	274
■■■■■ 8.4 Nuevos componentes con Gambas	279
□□□□□ Preparación del código. Palabra clave EXPORT	280
□□□□□ Archivos necesarios, ubicaciones	280
□□□□□ Probando el nuevo componente	282

13

SÍNTESIS

[Índice](#)

CAPÍTULO 9: ACCESO A LA API	287
■■■■■ 9.1 Declarar una función externa	288
□□□□□ Cómo denominar la librería	289
□□□□□ El uso de los alias	290
□□□□□ Tipos de datos	291
■■■■■ 9.2 Funciones auxiliares	291
■■■■■ 9.3 Un ejemplo con libaspell	293
■■■■■ 9.4 Obtener información acerca de la librería	297
■■■■■ 9.5 Resumen	300
APÉNDICE: Marcas registradas	301



• Es obvio que para los que hablamos el idioma español, la palabra gambas nos sugiere algunas cosas, pero todas ellas bastante alejadas del mundo de los ordenadores en general y del entorno de la programación en particular. El autor original de Gambas, Benoit Minisini, no habla una palabra de nuestro idioma y su inocente ignorancia le condujo a nominar su obra con el acrónimo GAMBAS: *Gambas almost means BASIC*, es decir, 'Gambas casi quiere decir BASIC'. No es la primera vez que un nombre o una marca bautizados en otros idiomas produce estos extraños cambalaches, recordemos que Coca ColaTM tuvo que cambiar su pronunciación en China porque la primera versión de su nombre significaba *muerde el renacuajo de cera*, o algunos todavía recordamos nuestra cara de asombro al ver los anuncios de un coche al que Suzuki dio en llamar *Pajero*. Tampoco está mal recordar que ya había antecedentes de otros lenguajes de programación con nombre de animal, como Camel

o Python, aunque en esos casos el nombre estaba en inglés y en español no resultaba tan chocante.

En fin, como Benoit, que tiene los derechos de autor, no desea cambiar el nombre, nos tendremos que ir acostumbrando a que Gambas empiece a sonarnos a algo más que a buen marisco. De hecho, Gambas abre el entorno de la programación visual en Linux a todo el mundo, como lo hizo en su día Visual BasicTM en Windows. Pero como el tiempo no pasa en vano, Gambas intenta no reproducir los errores que se cometieron entonces. La ampliación del lenguaje BASIC alcanza con Gambas amplias cotas de potencia, profesionalidad y modernidad, sin abandonar nunca la sencillez y claridad de este lenguaje de programación de alto nivel. Ya nunca más se podrá decir que construir aplicaciones visuales para Linux es un proceso largo y complejo que lleva años de trabajo a gurús y maniáticos de la informática.

18

Gambas no es sólo un lenguaje de programación, es también un entorno de programación visual para desarrollar aplicaciones gráficas o de consola. Hace posible el desarrollo de aplicaciones complicadas muy rápidamente. El programador diseña las ventanas de forma gráfica, arrastra objetos desde la **Caja de Herramientas** y escribe código en BASIC para cada objeto. Gambas está orientado a eventos, lo que significa que llama automáticamente a los procedimientos cuando el usuario de la aplicación elige un menú, hace clic con el ratón, mueve objetos en la pantalla, etc.

■■■■■ I. I El lenguaje BASIC: su historia

El nombre BASIC corresponde a las siglas *Beginner's All Purpose Symbolic Instruction Code* (Código para principiantes de instrucciones simbólicas con cualquier propósito). El lenguaje fue desarrollado en 1964 en el Dartmouth College por los matemáticos John George Kemeny y Tom Kurtz. Intentaban construir un lenguaje de programación fácil de aprender para sus estudiantes de licenciatura. Debía ser un paso intermedio antes de aprender otros más potentes de aquella

época, como FORTRAN o ALGOL. Este último era el lenguaje más utilizado en aplicaciones de procesos de datos, mientras que FORTRAN era empleado en las aplicaciones científicas. Sin embargo, ambos eran difíciles de aprender, tenían gran cantidad de reglas en las estructuras de los programas y su sintaxis. El primer programa hecho en BASIC se ejecutó a las 4 de la madrugada del 1 de mayo de 1964. Debido a su sencillez, BASIC se hizo inmediatamente muy popular y se empezó a usar tanto en aplicaciones científicas como comerciales. Tuvo el mismo impacto en los lenguajes de programación que la aparición del PC sobre los grandes ordenadores.

Cuando se desarrolló BASIC eran los tiempos en los que la informática estaba recluida en universidades y grandes empresas, con ordenadores del tamaño de una habitación. Pero pronto las cosas empezaron a cambiar. En 1971 Intel fabricaba el primer microprocesador. En 1975, la empresa MITS lanzó al mercado un kit de ordenador llamado Altair 8800 a un precio de 397 dólares. Era un ordenador barato, pero no para gente inexperta, había que saber electrónica para montarlo. Además tenía sólo 256 bytes (no es una errata, solo bytes, nada de Kbytes, megas o gigas) y se programaba en código máquina a base de 0 y 1, moviendo unos interruptores que tenía en el frontal. Dos jovencitos vieron un modelo en una revista de electrónica y decidieron montarlo. Le ofrecieron al dueño de MITS, además, hacer un intérprete de BASIC para los nuevos modelos de Altair. Eran William Gates y Paul

Allen, y aquel BASIC, con un tamaño de 4 Kbytes, fue el primer producto que entregó una nueva empresa llamada Microsoft. Fue sólo el principio. A finales de los 70, Allen y Gates habían portado BASIC ya a un buen número de plataformas: Atari, Apple, Commodore... Y en 1981, cuando desarrollaron DOS para IBM y su nuevo PC, añadieron también su propio intérprete de BASIC al sistema. En posteriores años siguieron otras versiones hechas por otras compañías como Borland, pero el declive de BASIC había empezado. Las interfaces gráficas de ventanas que Apple popularizó y Microsoft adoptó con sucesivas versiones de



Figura 1. Lanzamiento del Altair 8800.

WindowsTM, se convirtieron en un estándar y BASIC no era un lenguaje preparado para estos entornos.

Sin embargo, en marzo de 1988, un desarrollador de software llamado Alan Cooper¹ intentaba vender una aplicación que permitía personalizar fácilmente el entorno de ventanas usando el ratón. El programa se llamaba Tripod y en aquellas fechas consiguió que William Gates lo viera y le encargara el desarrollo de una nueva versión a la que llamaron Ruby, y a la que añadieron un pequeño lenguaje de programación. Microsoft reemplazó ese lenguaje por su propia versión de BASIC, Quickbasic y el 20 de marzo de 1991 se lanzó al mercado con el nombre de Visual Basic. Al principio fue un verdadero fracaso de ventas, pero la versión 3 publicada en el otoño de 1996 fue un éxito total, tanto que actualmente es el lenguaje de programación más usado. Visual Basic siguió evolucionando hasta la versión 6.0. En 2002 fue integrado en la plataforma .NET de desarrollo, en lo que para muchos de los seguidores ha supuesto el abandono de Microsoft, ya que ha cambiado buena parte de la sintaxis añadiéndole complejidad en contradicción con el espíritu y el nombre del lenguaje. En cualquier caso, a día de hoy se calcula que entre el 70% y 80% de todas las aplicaciones desarrolladas en Windows se han hecho con alguna de las versiones de Visual Basic.

20

Las causas del éxito de Visual Basic son numerosas, pero entre otras se puede señalar como obvia el uso del lenguaje BASIC que fue pensado para un aprendizaje fácil. Otro de los motivos es disponer de un entorno de desarrollo cómodo, que hace un juego de niños el diseño de la interfaz gráfica de cualquier aplicación, apartando al programador de perder tiempo en escribir el código necesario para crear ventanas, botones, etc., y dejándole centrarse únicamente en la solución al problema que cualquier programa intenta resolver. Con la popularización de sistemas operativos libres como GNU/Linux, éstas y otras razones hacían prever que la aparición de un entorno equivalente libre sería un éxito y contribuiría a la presentación de muchos nuevos desarrollos que lo utilizarían. Ha habido varios intentos que no han cuajado, bien por la lentitud de su evolución, bien por su dificultad de uso o por no ser totalmente libres y no haber arrastrado a una comunidad de usuarios detrás. Finalmente, Benoit Minisini, un programador con experiencia en la escritura de compiladores

que estaba harto de luchar contra los fallos de diseño de Visual Basic, y deseaba poder usar un entorno de GNU/Linux fácil en su distribución, comenzó a desarrollar su propio entorno para Linux basado en BASIC. El 28 de febrero de 2002 puso en Internet la primera versión pública de Gambas: gambas 0.20. Benoit eliminó del diseño del lenguaje bastantes de los problemas que Visual Basic tenía, como la gestión de errores, y le añadió características comunes en los lenguajes actuales más modernos, como la orientación a objetos y la propia estructura de los programas². Como prueba de fuego, el propio entorno de desarrollo fue programado en Gambas desde la primera versión, sirviendo a un tiempo de demostración de la potencia del lenguaje y de detección de necesidades y corrección de errores que se fueron incorporando a las distintas versiones.

En enero de 2005, Benoit publicó la versión 1.0, en la que ya se incorporaba un puñado de componentes desarrollados por otros programadores que colaboraron con él: Daniel Campos, Nigel Gerrard, Laurent Carlier, Rob Kudla y Ahmad Kahmal. Esta versión se consideró suficientemente estable y cerró un ciclo. A partir de esa fecha empezó la programación de la versión 2.0. Ésta ya incluye algunas mejoras en el lenguaje, muchos más componentes y un nuevo modelo de objetos que permitirán usar Gambas en un futuro para el desarrollo de aplicaciones web con la misma filosofía y facilidad que actualmente se usa para aplicaciones de escritorio.

■ ■ ■ ■ I. 2 Un entorno libre

Gambas es un entorno de desarrollo que se distribuye con la licencia GPL GNU (*General Public Licence*³). Esto significa que se distribuye siempre con el código fuente y respeta las cuatro libertades que define la Free Software Foundation:

- La libertad de usar el programa con cualquier propósito (libertad 0).
- La libertad de estudiar cómo funciona el programa y adaptarlo a las propias necesidades (libertad 1). El acceso al código fuente es una condición previa para esto.

- La libertad de distribuir copias, con las que se puede ayudar al vecino (libertad 2).
- La libertad de mejorar el programa y hacer públicas las mejoras a los demás, de modo que toda la comunidad se beneficie (libertad 3). El acceso al código fuente es un requisito previo para esto.

Una de los engaños más comunes en el uso de Software Libre es la creencia de que este modelo de desarrollo obliga a que el trabajo se publique gratis, lo que es del todo incierto. Estas cuatro libertades permiten que, quien lo desee, venda copias de Gambas (entregando siempre el código fuente y respetando esas cuatro libertades) y, por supuesto, de cualquier aplicación desarrollada con este programa. Las aplicaciones desarrolladas con Gambas pueden o no acogerse a la licencia GPL.

También cualquier programador es libre de alterar el propio lenguaje y modificarlo a su gusto, siempre y cuando entregue el código correspondiente a esas modificaciones y respete los derechos de autor de los desarrolladores originales.

Aparte de estas libertades propias de la naturaleza de un proyecto de Software Libre sobre GNU/Linux, Gambas añade más facilidades para el programador:

- Una ayuda muy completa del lenguaje y cada uno de los componentes, algo que es muy de agradecer para los que empiezan a programar en Gambas, y que no es habitual en los proyectos de Software Libre. La ayuda que se publica está en inglés, pero existe un grupo de personas trabajando en la traducción a español⁴. Si todos nos animamos a colaborar⁵ en la traducción, pronto estará completa y disponible para el resto de usuarios.
- Una API (Interfaz para programar la aplicación) sencilla y bien documentada, lo que facilita a los programadores crear nuevos componentes para Gambas. La API no es de utilidad inmediata para quien desarrolle con este lenguaje, pero permite a los programadores avanzados que lo deseen añadir funcionalidades al entorno de desarrollo y crear nuevas herramientas para Gambas.

El lenguaje está preparado para ser independiente del gestor de ventanas que use. Esto significa que, sin cambiar una sola línea de código, una aplicación puede ser compilada para ser ejecutada en un escritorio Gnome o KDE, usando las librerías propias de ese escritorio y siendo una aplicación nativa de ese entorno. En el futuro se pueden desarrollar componentes para Windows™, Fluxbox y otros gestores de ventanas, y los programas no tendrán que modificar su código para que sean aplicaciones nativas de esos entornos. Marcando, antes de compilar, una opción en el entorno de desarrollo para elegir el componente a usar (actualmente se puede elegir entre *gtk* y *qt*, para Gnome o KDE), se generan distintas aplicaciones para distintos entornos con el mismo código fuente. Esta característica no se encuentra disponible en ningún otro lenguaje existente, lo que convierte a Gambas en un entorno único.

■ ■ ■ ■ ■ 1.3 Elementos de Gambas

Para poder desarrollar y ejecutar programas hechos con Gambas, son necesarios distintos elementos:

23

- Un **compilador**, que se encargará de transformar todo el código fuente y archivos que formen parte de un proyecto hecho en Gambas, en un programa ejecutable.
- Un **intérprete** capaz de hacer que los programas hechos en Gambas sean ejecutados por el sistema operativo.
- Un **entorno de desarrollo** que facilite la programación y diseño de las interfaces gráficas de los programas.
- **Componentes** que añaden funcionalidades al lenguaje. La palabra *componente* en Gambas tiene un significado específico, ya que no alude a partes genéricas, sino a librerías específicas que le dotan de más posibilidades. En la actualidad existen componentes para usar xml, conexiones de red, opengl, sdl, ODBC, distintas bases de datos, expresiones regulares, escritorios basados en *qt*, en *gtk*,

etc. Estos componentes son desarrollados por distintos programadores, siguiendo las directrices de la API de Gambas y la documentación publicada al efecto por Benoit Minisini.

I. 4 Cómo obtenerlo

Las nuevas versiones de Gambas se publican a través de la página web oficial del proyecto: <http://gambas.sourceforge.net>. En la actualidad existen dos ramas de Gambas: la rama estable o 1.0 y la rama de desarrollo o 1.9 que desembocará en la versión 2.0. De la rama estable sólo se publican nuevas versiones cuando es para corregir algún fallo que se haya descubierto.

24

En el momento de escribir estas líneas, la versión estable era la 1.0.11 y no se prevé que haya cambios en el futuro. La versión de desarrollo está en continuo cambio, mejora y adición de nuevos componentes. Esto la hace muy atractiva, debido a la existencia de importantes funcionalidades que no están en la versión estable (como los componentes gtk y ODBC). Sin embargo, al ser una rama en desarrollo, es muy probable que tenga fallos no descubiertos y es seguro que sufrirá cambios en un futuro próximo. En este texto trataremos todas las novedades que la versión de desarrollo contiene para que sea el usuario el que escoja con qué rama trabajar. Buena parte de las diferencias se encuentran en los nuevos componentes. Algunos de estos serán tratados en este texto, por lo que si el lector quiere trabajar con ellos deberá usar la rama de desarrollo.

Las nuevas versiones se publican siempre en forma de código fuente, para que los usuarios que lo deseen compilen el código y obtengan todas las partes que Gambas tiene. Los autores de algunos de los componentes que se han hecho para Gambas, publican de forma separada en distintos sitios web las versiones nuevas de estos, pero todas se envían a Benoit Minisini y pasan a formar parte de la publicación completa de este lenguaje de programación en la siguiente versión. De este modo, se puede decir que cuando Benoit hace pública una nueva, el paquete del código fuente contiene las últimas versiones de todo el conjunto en ese momento.

Como la compilación de Gambas y todos los componentes asociados puede ser una tarea difícil para usuarios no expertos, es común que se creen paquetes binarios con la compilación ya hecha y listos para ser instalados en distintas distribuciones de gnu/Linux. En la misma página web donde se puede bajar el código fuente se encuentran los enlaces para la descarga de los paquetes compilados para estas distribuciones. Existen actualmente paquetes disponibles para Debian, Fedora, Mandriva, Gentoo, Slackware, QiLinux y Suse. En algunos casos, como para Fedora y Debian, están disponibles tanto los paquetes de la versión estable como la de desarrollo.

En el caso de Debian, los paquetes son realizados en gnuLinEx y, posteriormente, subidos a Debian para que estén disponibles y usables en esta distribución y en todas sus derivadas, como Knoppix, Guadalinex, Progeny, Xandros, Linspire, Skolelinux, etc. Por este motivo, las últimas versiones están siempre disponibles antes en los repositorios de gnuLinEx hasta que son subidos y aprobados en Debian. Las líneas del archivo */etc/apt/sources.list* de un sistema Debian para instalar la versión más actualizada de los paquetes de Gambas son:

Para la versión estable:

```
deb http://apt.linex.org/linex gambas/
```

Para la versión de desarrollo⁶:

```
deb http://www.linex.org/sources/linex/debian/ cl gambas
```

A continuación, en cualquiera de los dos casos, para instalar todos los paquetes de Gambas, hay que ejecutar como usuario root:

```
apt-get update  
apt-get install gambas
```

Aunque el código fuente de Gambas se distribuye en un único archivo comprimido, la instalación desde paquetes compilados se hace con un buen puñado de archivos.

La razón es que no todos son necesarios para ejecutar aplicaciones hechas en Gambas. En las distribuciones de Linux se ha seguido el criterio de separar en distintos paquetes el entorno de desarrollo (paquete *gambas-ide*), el intérprete (paquete *gambas-runtime*), y se ha hecho un paquete separado para cada uno de los componentes. Si se quiere programar en Gambas son necesarios la mayoría de ellos, al menos los que el entorno de desarrollo necesita. Si se quiere ejecutar un programa hecho con este lenguaje, sólo es necesario *gambas-runtime* y un paquete por cada uno de los componentes que el programa use. Por ejemplo, si es un programa que está hecho para el escritorio Gnome y no usa ningún otro componente, sólo sería necesario instalar en el sistema los paquetes *gambas-runtime* y *gambas-gb-gtk*.

■ ■ ■ ■ I. 5 Compilación y dependencias

26

Si en lugar de instalar paquetes ya compilados para la distribución de gnu/Linux deseamos compilar Gambas desde el código fuente, deberemos seguir los pasos habituales en los sistemas GNU. Es decir, descomprimir el archivo con las fuentes y, desde el directorio que se crea al descomprimir y usando un terminal, ejecutar las siguientes instrucciones:

```
./configure  
make  
make install
```

La última de ellas debemos hacerla como root, si queremos que el programa esté disponible para todos los usuarios del ordenador. Si estamos habituados a compilar aplicaciones en sistemas GNU, disponemos ya de un compilador instalado y de bastante librerías de desarrollo. Las instrucciones anteriores tratarán de compilar e instalar todos los componentes de Gambas, que son muchos. Si no tenemos las librerías correspondientes a alguno de ellos, simplemente no se compilarán y la instrucción *./configure* nos informará de ello. Es importante saber que el entorno de desarrollo está hecho sobre las librerías gráficas *qt*, por tanto, para poder usar el entorno necesitaremos tener instalado, al menos, estas librerías de desarrollo con una versión igual

o superior a la 3.2. La versión del compilador gcc ha de ser también ésta, como mínimo. Cada uno de los componentes tiene dependencias de sus propias librerías y dependerá de la distribución de Linux que usemos, para saber el nombre del paquete que deberemos instalar antes de poder realizar la compilación.

■ ■ ■ ■ I. 6 Familiarizarse con el IDE

Aunque un programa en Gambas se podría hacer perfectamente usando un editor de texto plano cualquiera, sería un desperdicio no aprovechar uno de los mayores atractivos que el lenguaje tiene: su IDE o entorno de desarrollo. El IDE de Gambas ahorra al programador buena parte del trabajo más tedioso, le proporciona herramientas que hacen mucho más fácil su tarea, con utilidades de ayuda, de diseño de interfaces, auto-completado de instrucciones, traducción de programas, etc. En la imagen siguiente podemos ver algunas de las ventanas más importantes del entorno, que se usan durante el desarrollo de una aplicación:

27



Figura 2. Entorno de desarrollo de Gambas.

Cuando se arranca Gambas, lo primero que nos aparece es la ventana de bienvenida.

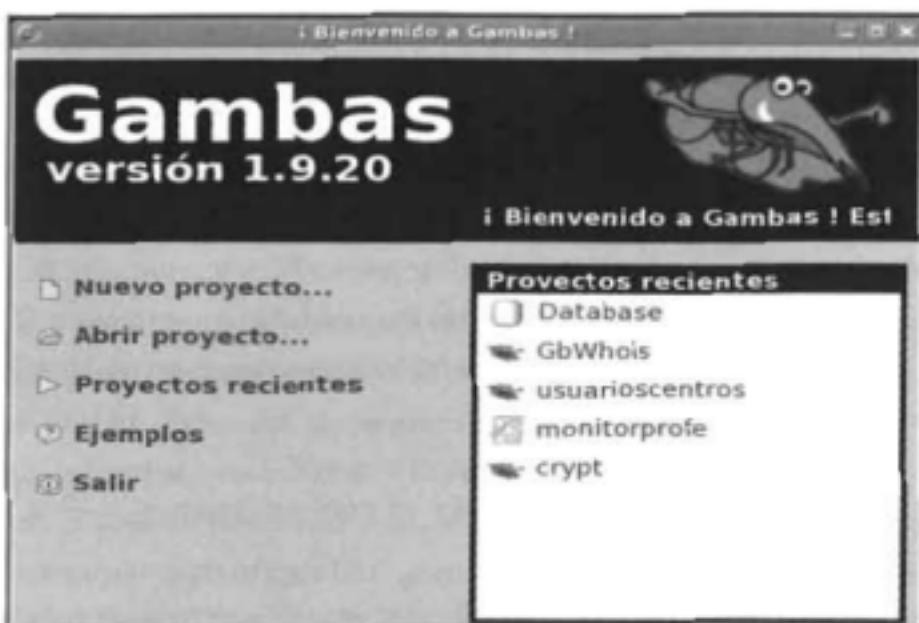


Figura 3. Ventana de bienvenida.

28

Aquí se nos ofrece la opción de comenzar un nuevo proyecto o aplicación, abrir un proyecto del que tengamos sus archivos disponibles, abrir uno usado recientemente o uno de los numerosos ejemplos que están incluidos en la ayuda de Gambas.

Antes de elegir cualquiera de estas opciones es necesario saber que todos los códigos fuente de una aplicación hecha en Gambas es lo que se denomina *proyecto*. El proyecto está formado por una serie de archivos que en Gambas están SIEMPRE situados dentro de un único directorio. En él puede haber, a gusto del desarrollador, distintos subdirectorios y organizar todo como se deseé, pero cualquier gráfico, texto y código que forme parte de la aplicación estará dentro de él. Por ello, si elegimos en esta ventana la opción Nuevo proyecto..., el asistente siempre creará un nuevo directorio con el nombre del proyecto y ahí irá introduciendo todos los archivos necesarios para el desarrollo de la aplicación. Así, para enviar a alguien el código fuente de una aplicación hecha en Gambas o cambiarla de ordenador o disco, sólo hay que transportar el directorio con el nombre del proyecto, sin tener que preocuparse de otros archivos. Del mismo modo, si desde el entorno de desarrollo escogemos un

archivo o un gráfico para integrarlo en nuestro trabajo, el archivo será copiado automáticamente al directorio del proyecto.

□ □ □ □ □ El primer ejemplo

Una de las formas más habituales de empezar a trabajar con un lenguaje de programación es haciendo un pequeño programa que muestre el mensaje *Hola Mundo*. Por tanto, empezaremos a conocer el entorno de desarrollo y el lenguaje de programación con este típico ejemplo. Comenzaremos haciendo un *hola mundo* que sea puro BASIC, es decir, que sea igual al que hubieran hecho los autores de BASIC allá por el año 1964.

En aquellos tiempos las interfaces gráficas no existían, por lo que este primer programa será un programa feo, de terminal. En el BASIC original, hacer que aparezca un mensaje en el terminal es tan simple como escribir la línea:

```
PRINT "Hola Mundo"
```

29

No es necesario ningún encabezado previo, la instrucción PRINT sirve para mostrar cualquier cadena de texto en el terminal, que en BASIC se presentan entre comillas dobles. De ahí que el programa sea tan simple como ese. Vamos a ver cómo hacerlo con el entorno de desarrollo de GAMBAS:

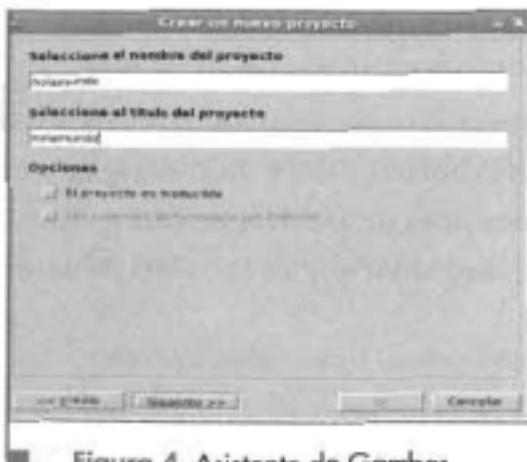


Figura 4. Asistente de Gambas.

1. Escogemos la opción Nuevo proyecto... en la ventana anterior (Figura 3). Aparecerá un asistente, en el que pulsamos Siguiente. Surgirá una nueva ventana para elegir el tipo de aplicación. Escogemos la tercera opción: Crear un proyecto de texto⁷.
2. Le damos un nombre al proyecto, por ejemplo *holamundo* y un título. Pulsamos el botón Siguiente (Figura 4).

Elegimos el directorio del disco en el que queremos crearlo; pulsamos de nuevo el botón **Siguiente** y aparecerá un resumen con los datos del proyecto (Figura 5).

3. A continuación pulsamos el botón **OK**. Se abrirá el entorno de desarrollo de Gambas listo para empezar a programar. Al ser una aplicación de terminal, que no lleva interfaz gráfica, de momento, sólo necesitamos fijarnos en la ventana de la izquierda, que en nuestro caso tendrá como título: *Proyecto – holamundo* (Figura 6).

30

En realidad ésta es, probablemente, la ventana más importante para el manejo del entorno de desarrollo. A simple vista se puede ver el menú superior, que contiene las entradas necesarias para guardar y cargar proyectos, activar las distintas ventanas del IDE, manejar la ejecución de los programas, personalizar el entorno (en Herramientas | Preferencias | Otros | Mostrar masota, podemos ocultar la animación de la gamba, que nos está mirando al abrir el proyecto), etc.

De momento nos fijamos sólo en el árbol de directorios que contiene. Podemos ver que la raíz del árbol es el nombre del proyecto: **holamundo**, y de él cuelgan tres ramas: **Clases** y **Módulos**, que son para distintos tipos de archivos de código fuente; y **Datos**, cuyo nombre indica su finalidad, almacenar ahí los archivos de datos que la aplicación requiera.

Desde el principio, Gambas nos da dos formas de realizar los programas, incluso si son tan simples como el que hemos hecho. Podemos elegir entre una programación

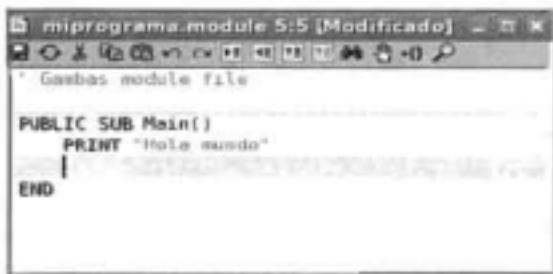


Figura 5. Datos del proyecto.



Figura 6. Proyecto – holamundo.

orientada a objetos, paradigma típico de los lenguajes de programación más potentes o una programación estructurada simple. Según ello, el archivo que contenga el código de nuestro programa será una *Clase* o un *Módulo*. Por simplicidad, de momento vamos a usar un *Módulo*. Haciendo clic con el botón derecho del ratón sobre el árbol de carpetas aparecerá un menú contextual. Elegimos las opciones Nuevo | Módulo. Surgirá una ventana en la que escribimos el nombre del módulo, por ejemplo *miprograma*, y pulsamos el botón OK, con lo que aparecerá nuestra primera ventana, con el título *miprograma.module*, donde poder escribir código.



```
miprograma.module 5.5 [Modificado] 
Gambas module file
PUBLIC SUB Main()
    PRINT "Hola mundo"
END
```

Figura 7. Ventana donde escribir el código.

Por fin podemos escribir nuestro código en BASIC. Lo haremos justo antes de la línea donde pone END, tal y como queda reflejado en la figura de la izquierda.

Cuando después de escribir el código que ya sabíamos, pulsamos la tecla INTRO, vemos que el entorno colorea el texto de una forma particular. Podemos pararnos

un instante a ver los distintos colores que se muestran (Figura 7):

- En gris aparece una línea que comienza por una comilla simple ('). Esto indica que la línea es un comentario, es decir, no se trata de ningún código de programación y el texto que sigue a la comilla no se ejecuta nunca, son comentarios que el programador puede/debería poner para facilitar que otros (o él mismo, pasado un tiempo) entiendan lo que el programa hace en ese punto.
- En azul podemos ver palabras clave del lenguaje BASIC.
- En color rosado aparece la cadena de texto.
- A la izquierda vemos un resalte amarillo al comienzo de las líneas que han sido modificadas. Esto aparecerá siempre en las líneas que contengan modificaciones que no hayan sido compiladas.

Bien, ya está listo el programa. Para comprobarlo se pulsa en el botón verde, con el símbolo del Play, que está en la pantalla del proyecto. Al hacerlo aparecerá una nueva ventana llamada *Consola* en la que se verá la salida de nuestro programa. En este caso será el simple texto *Hola Mundo*. Éste es todo el código necesario, ya se puede compilar el programa para generar un archivo ejecutable que funciona sin necesidad del entorno de desarrollo. Para ello, en el menú de la ventana del proyecto hay que escoger: **Proyecto | Crear ejecutable**. Saldrá un cuadro de diálogo para elegir el directorio en el que queremos crear el ejecutable. Pulsando OK lo generará. Al cerrar ahora el entorno de desarrollo de Gambas y abrir un terminal o pasar a una consola de Linux, podemos probar su funcionamiento. Para ello, en el directorio donde se haya creado el ejecutable, hacemos:

```
jose@a00-o04:~$ cd gambas/holamundo/  
jose@a00-o04:~/gambas/holamundo$ ./holamundo.gambas  
Hola mundo
```

32

□ □ □ □ □ Mejor con un ejemplo gráfico

El ejemplo anterior mostraba una aplicación de consola, que nos recuerda a los viejos tiempos de otros sistemas operativos o a la forma de trabajar de los hackers informáticos. En realidad, hacer ese tipo de programas no demuestra el potencial de Gambas, puesto que son realmente simples e igualmente fáciles de realizar en otros lenguajes como Python o cualquier vieja versión de BASIC.

Es mejor hacer el programa *Hola Mundo* para el entorno gráfico que inunda los escritorios de los ordenadores actuales. Para ello empezaremos igual que antes, arrancando Gambas y creando un **Nuevo proyecto** siguiendo exactamente los mismos pasos, excepto que en lugar de escoger **Crear un proyecto de texto**, cuando el asistente nos presente las distintas opciones, elegiremos **Crear un proyecto gráfico**.

Para no repetir nombre, podemos denominar al proyecto *holamundo2*. Al acabar el proceso aparecerá de nuevo la ventana de proyecto, pero en esta ocasión tendrá una rama más en el árbol: **Formularios**. Un formulario es el área donde se diseña

la interfaz gráfica de la aplicación, es decir, donde se insertan objetos como botones, cuadros de texto, listas, casillas de verificación, etc. Los formularios se corresponderán con las ventanas que la aplicación mostrará.

Haciendo clic con el botón derecho del ratón sobre el árbol del proyecto, elegimos ahora en el menú contextual que aparece: Nuevo | Formulario. Por simplicidad, en este caso ni siquiera hace falta cambiar el nombre, sólo pulsando en el botón OK aparecerá en pantalla la ventana del formulario y una ventana para escribir código BASIC casi idéntica a la del ejemplo anterior, sin nada escrito. El resultado será algo parecido a lo siguiente:



Figura 8. Pantalla del formulario.

Haciendo clic sobre la palabra **Form** en la ventana de la derecha, correspondiente a la **Caja de Herramientas**, aparecerán distintos objetos que podemos colocar en los formularios. Entre estos está el icono del botón (se distingue rápidamente por tener la palabra **OK** escrito dentro). Haciendo un clic de ratón en él, se puede dibujar en el formulario un botón, tan sólo hay que arrastrarlo con el ratón y el botón izquierdo pulsado, para dar la forma que queramos. Gambas escribe el texto *Button1* sobre el ratón, pero como ese texto no es demasiado intuitivo lo mejor es cambiarlo. Para ello hay que pinchar en el botón y después, en la ventana de propiedades, hacer un clic en la línea donde pone **Text**. Ahora se puede escribir el nuevo texto, por ejemplo **Púlsame**.

También, pulsando en la propiedad *Name* le cambiamos el nombre al objeto botón poniéndole, por ejemplo, *btnMensaje*. Siguiendo los mismos pasos dibujamos otro botón sobre el formulario, escribimos el texto **Salir** y el nombre *btnSalir*. Una vez que tenemos diseñada la interfaz gráfica con el formulario y los dos botones, pasamos a escribir el código BASIC correspondiente. Para ello, haciendo un doble clic con el ratón sobre el botón con el texto *Púlsame*, aparecerá la ventana de código con un texto ya escrito, al que añadiremos algo de forma que el código de ese botón quede así:

```
PUBLIC SUB btnMensaje_Click()
    Message.Info("Hola Mundo")
END
```

Haciendo doble clic sobre el botón Salir, escribimos el código que falta:

```
PUBLIC SUB btnSalir_Click()
    QUIT
END
```

El resultado final debe ser algo como esto:

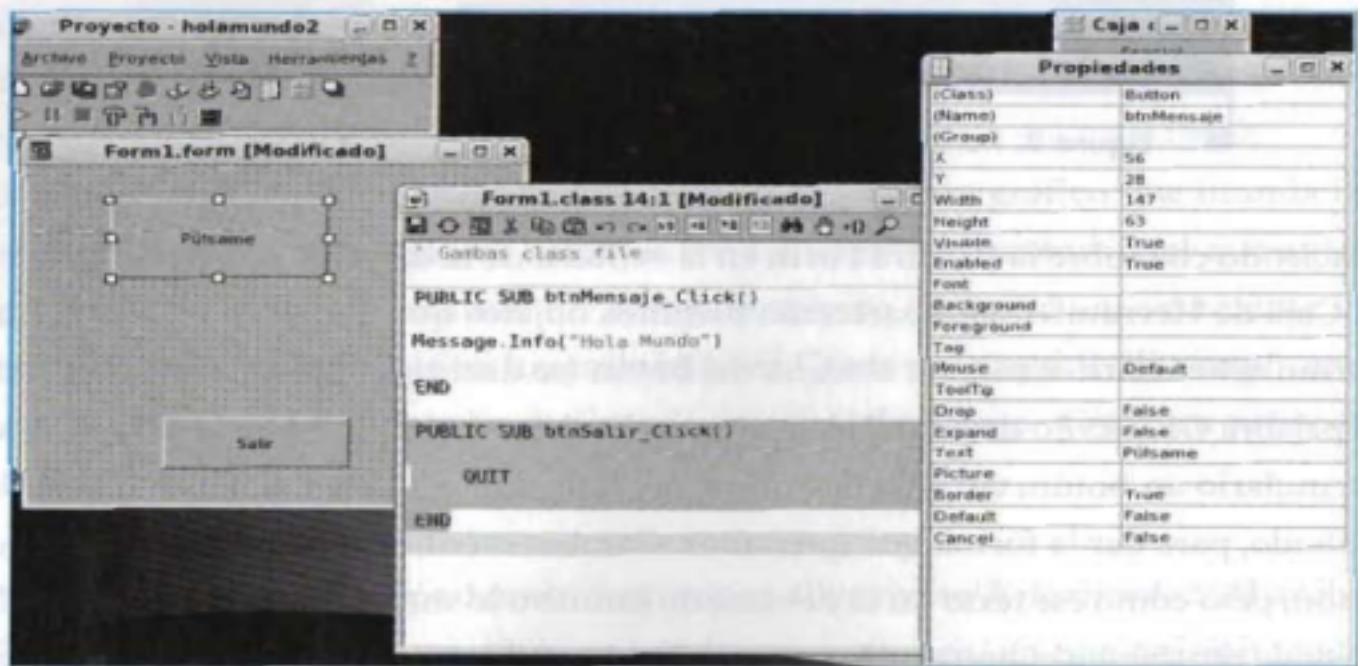


Figura 9. Resultado final de la programación de los dos botones.

Con esto ya está el programa terminado. Igual que antes, pulsando en el botón verde de la ventana de proyecto el programa se ejecutará. Sin embargo, en este caso la ejecución presenta novedades importantes: es un programa gráfico, tiene ya su ventana con la barra superior y los típicos botones de maximizar y minimizar. Además, cuenta con dos botones y la aplicación está esperando a que el usuario haga clic en alguno de ellos para actuar.

Hora de experimentar. Sería bueno, llegado este punto, que perdiéramos un poco de tiempo tocando las distintas propiedades de cada uno de los tres objetos que el programa tiene: el formulario y los dos botones, parando el programa y volviendo a arrancarlo para ver el resultado de las modificaciones tantas veces como queramos.

Aunque no es parte de este capítulo, hay un par de detalles que ya deberían empezar a observarse:

- El código BASIC se escribe entre unas líneas que empiezan por PUBLIC SUB y acaban en END.
- El texto que sigue a PUBLIC SUB lleva el nombre del objeto sobre el que se ha hecho un doble clic, seguido de la palabra Click, eso significa que el código se ejecutará cuando el usuario haga clic sobre el objeto.
- Para que aparezca Message.Info, se escribe Message y se pulsa sobre esta palabra. Aparece un menú entre cuyas opciones se encuentra la palabra Info. Moviéndonos con los cursores del teclado y pulsando la tecla de espacio o INTRO sobre esa palabra, quedaba escrito en la ventana de código. Ése es el autocompletado, que nos ayuda a escribir el código evitando tener que memorizar sentencias y mensajes extraños.

□ □ □ □ Un poco de magia

El entorno de desarrollo de Gambas está hecho con el componente *qt*, y de forma predeterminada lo escoge para las aplicaciones gráficas. Por tanto, un programa gráfico hecho en Gambas usa, en principio, estas librerías y se puede considerar

una aplicación nativa para el escritorio KDE. Sin embargo, el componente *gtk* disponible en la versión de desarrollo de Gambas, permite hacer programas enlazados con las librerías de *Gtk* para realizar aplicaciones del escritorio Gnome.

Si hemos instalado los paquetes de Gambas correspondientes a la rama de desarrollo, incluyendo el paquete *gambas-gb-gtk*, podemos recuperar la aplicación *holamundo2* del ejemplo anterior, ir a la ventana de *Proyecto* y hacer clic en el menú *Proyecto | Propiedades*, pestaña *Componentes*, y seleccionar *gb gtk*. El resultado al ejecutar la aplicación es que se trata de una nueva, pero antes de KDE y ahora de Gnome. ¡Y sin tocar una sola línea de código!

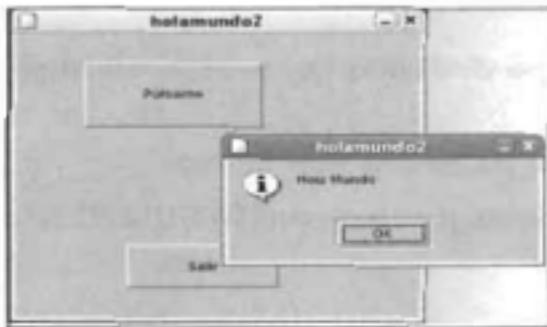


Figura 10. Aplicación *holamundo2* en KDE.

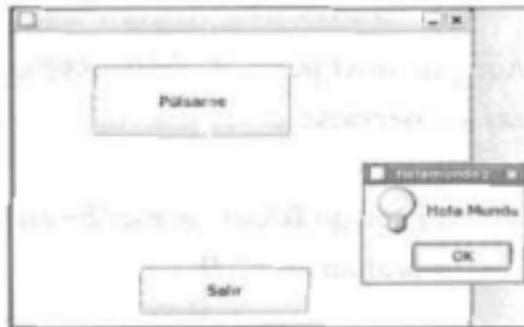


Figura 11. Aplicación *holamundo2* en Gnome.

Hasta el día de hoy no existe ningún otro lenguaje de programación con el que se pueda hacer esto.

■ ■ ■ ■ I. 7 Sistema de componentes

A lo largo de los apartados anteriores han aparecido distintas referencias a los componentes de Gambas, incluyendo su descripción en el tercer apartado de este capítulo. Estos permiten extender este lenguaje de programación. La interfaz desarrollada por Benoit para su programación hace que hayan sido varios los programadores que han colaborado con él, desarrollando nuevos componentes que se van añadiendo a las distintas versiones de Gambas en cada nueva publicación. En la versión 1.0 estable de

Gambas sólo se podían desarrollar en C y C++, pero desde la versión 1.9.4 de la rama de desarrollo también se pueden escribir componentes en Gambas, lo que abre numerosas posibilidades futuras ya que es mucho más sencillo hacerlo que en C.

El listado de componentes disponibles es amplio y se aumenta continuamente en la versión de desarrollo. Para la rama estable están fijados los siguientes:

- *gb.compress*: para compresión de archivos con protocolos *zip* y *bzip2*.
- *gb.qt*: para objetos visuales de las librerías gráficas *qt*.
- *gb.qt.ext*: para objetos visuales de las librerías gráficas *qt* que no son estándar.
- *gb.qt.editor*: un editor de texto hecho usando las librerías gráficas *qt*.
- *gb.qt.kde*: objetos visuales propios del escritorio KDE.
- *gb.qt.kde.html*: un navegador web del escritorio KDE.
- *gb.net*: objetos para conectar a servidores de red y a puertos serie de comunicaciones.
- *gb.net.curl*: objetos para construir servidores de red.
- *gb.db*: objetos de conexión a bases de datos.
- *gb.db.mysql*: driver para conectar al servidor de bases de datos MySQL.
- *gb.db.postgresql*: driver para conectar al servidor de bases de datos PostgreSQL.
- *gb.db.sqlite*: driver para usar bases de datos Sqlite 2.x.
- *gb.xml*: objetos para parsear archivos XML.

- *gb.vb*: colección de funciones para facilitar la migración desde Visual Basic.
- *gb.sdl*: objetos para reproducir, mezclar y grabar archivos de sonido.
- *gb.pcre*: objetos para usar expresiones regulares en el lenguaje.

En la rama de desarrollo existen estos componentes (algunos de los cuales han sido muy mejorados y aumentados, como el *gb.sdl*) y otros más, con una lista en continuo crecimiento. En el momento de escribir estas líneas podemos contar con:

- *gb.crypt*: objetos para encriptación DES y MD5.
- *gb.form*: objetos gráficos para formularios independientes de las librerías gráficas usadas.
- *gb.gtk*: objetos gráficos para formularios de las librerías *gtk*. Tiene los mismos objetos que el componente *gb.qt*, pero enlazan con esta otra librería de desarrollo.
- *gb.db.odbc*: driver para conectar a bases de datos a través de ODBC.
- *gb.info*: objetos que proporcionan distinta información acerca de los componentes y el sistema donde la aplicación se ejecuta.
- *gb.opengl*: objetos para dibujos tridimensionales con aceleración OpenGL.
- *gb.xml.rpc*: objetos para el uso del protocolo *rpc-xml*.
- *gb.sdl.image*: objetos para dibujos en dos dimensiones con aceleración gráfica.
- *gb.v4l*: objetos para capturar imágenes de cámaras de vídeo en Linux.
- *gb gtk pdf*: se utiliza para renderizar documentos *pdf* desde aplicaciones hechas en Gambas.

El listado de componentes disponibles para el programador puede verse en la pestaña **Componentes**, accesible a través del menú **Proyecto | Propiedades**. Cada uno de los componentes se corresponde a un paquete compilado en la distribución, de forma

que si se añade al proyecto, por ejemplo, el componente *gb.sdl*, ha de instalarse el paquete *gambas-gb-sdl* en los ordenadores donde se quiera ejecutar la aplicación compilada.

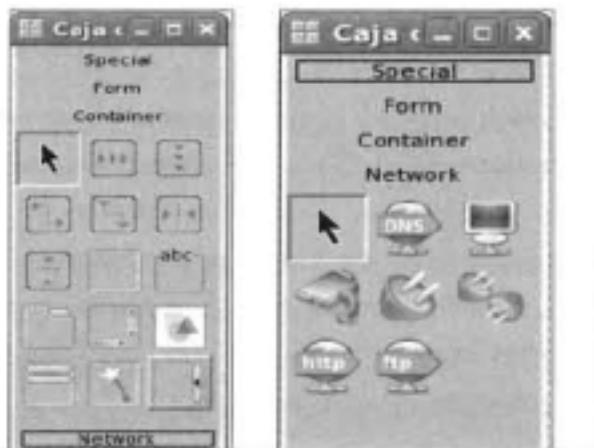


Figura 12. Componentes.

Al hacer clic sobre cada uno de los componentes aparece una pequeña descripción de su función, el nombre del autor o autores del componente, y un listado de los controles que están disponibles para el desarrollador si selecciona ese componente (Figura 12).



Los controles son clases para crear objetos útiles en la programación. Los objetos creados pueden ser visuales (como pestañas, editores de texto, etc.) u objetos de código (como servidores de red o conexiones a bases de datos). Si el componente tiene objetos visuales, estos se incorporan a alguna de las pestañas de la Caja de Herramientas del entorno de desarrollo.



Figuras 13, 14 y 15.
Caja de
Herramientas con
componentes con
objetos gráficos.

En las imágenes anteriores podemos ver algunos de los objetos gráficos que están disponibles al seleccionar los distintos componentes.

Cada componente tiene su propia documentación que se encuentra incluida en la ayuda de Gambas. En la rama estable esta documentación está siempre available; en la rama de desarrollo sólo está disponible si ha sido seleccionado para su uso en el proyecto.

Todas las cosas que se pueden hacer con Gambas, y no son parte del propio lenguaje BASIC, se programan mediante el uso de componentes. Esto significa que, por ejemplo, para hacer una aplicación de bases de datos es necesario seleccionar el componente *gb.db* o no estarán disponibles los objetos de conexión a bases de datos. Lo mismo ocurre con las conexiones de red, captura de video, etc. Estos objetos no son parte del lenguaje BASIC.

40

NOTAS

¹ http://www.cooper.com/alan/father_of_vb.html

² Diferencias entre Gambas y Visual Basic:

<http://wiki.gnulinex.org/gambas/210>

³ <http://www.fsf.org/licensing/licenses/gpl.html>. Hay una traducción (no oficial) al español en <http://gugs.sindominio.net/licencias/>

⁴ Documentación de Gambas en español: <http://wiki.gnulinex.org/gambas/>

⁵ Instrucciones para colaborar en la documentación en español:

<http://wiki.gnulinex.org/gambas/6>

⁶ Si la instalación se hace desde un sistema gnulinEx, no es necesario añadir esta linea puesto que la versión de desarrollo de Gambas es parte del repositorio oficial de gnulinEx.

⁷ Debido a la mayor extensión de la cadena **Crear un proyecto de texto** en español que en otros idiomas, es posible que ese mensaje aparezca cortado y no se vea completo. No hay que preocuparse por ello, no afecta para nada a su elección y suponemos que será corregido en posteriores versiones del IDE.



- ○ En el *Capítulo 1* se vieron ya algunos detalles acerca de la programación con Gambas.

Ahora, estudiaremos, principalmente, la sintaxis e instrucciones más importantes del lenguaje BASIC que usa Gambas. Buena parte de estas instrucciones existen en el BASIC estándar y algunas otras son extensiones propias de Gambas.

Aunque sea más espartano, en este capítulo se usarán casi siempre ejemplos de código de consola, como vimos en el apartado *El primer ejemplo* del *Capítulo 1*. Haciéndolo de este modo evitaremos las interferencias que puede suponer la explicación de conceptos relacionados con la programación gráfica y los componentes. Estos conceptos se explicarán con extensión en capítulos posteriores.

■ ■ ■ ■ ■ 2. I Organización de un proyecto de Gambas

Antes de comenzar con la programación básica hay que resumir algunos conceptos previos:

- El código fuente de los programas hechos en Gambas está compuesto de uno o más archivos que forman un proyecto. Este proyecto se archiva en un directorio del mismo nombre.
- Los archivos de código pueden ser: *Módulos* (contienen código en BASIC que se ejecuta directamente), *Clases* (contienen el código en BASIC que ejecuta un objeto de esa clase) y *Formularios* (áreas donde se diseña la interfaz gráfica de la aplicación y que se corresponden con las ventanas del programa).
- Los proyectos de texto sólo contienen *Módulos* y/o *Clases*. Las aplicaciones gráficas contienen *Formularios* y *Clases*, pero también pueden contener *Módulos*.
- El proyecto puede contener otros archivos de datos, documentos, textos, etc., sin código BASIC para ser ejecutado por la aplicación.

42

Los archivos que contienen código en BASIC (*Módulos* y *Clases*) siempre están estructurados de la siguiente manera:

- Declaración de variables.
- Subrutinas y Funciones.

□ □ □ □ □ Declaración de variables

Los programas manejan datos continuamente. Estos datos pueden ser de muchos tipos: números, letras, textos, etc., y cambiar a lo largo de la ejecución del programa (en ese caso reciben el nombre de *variables*) o permanecer con un valor fijo durante todo el tiempo (entonces se denominan *constantes*). A los datos que usa un programa se les asigna un nombre identificador.

BASIC permitía usar variables y constantes sin haberlas declarado antes, es decir, sin haber expuesto al principio del programa un listado con las variables que se iban a usar.

Eso produce programas ilegibles en cuanto empiezan a ser de tamaño medio y es una fuente constante de errores. Para evitar esto, Gambas obliga a declarar las constantes y las variables antes de utilizarlas.

Hay dos lugares donde se pueden declarar las variables, dependiendo del ámbito en el que se vayan a usar. Si se declaran dentro de una subrutina o función (en el siguiente apartado se verá con detalle qué son las subrutinas y funciones), están disponibles para ser usadas sólo dentro de esa subrutina o función. Si se declaran al principio del archivo de código (sea un *Módulo* o una *Case*) están disponibles para todo el código de ese archivo, en todas sus subrutinas.

Para declarar una variable en una subrutina o función se emplea la sintaxis:

```
Subroutine DIM nombre_variable AS tipo_variable
```

El *tipo_variable* hace referencia al tipo de dato de la variable: número entero, cadena de texto, número decimal, etc. El nombre de la variable lo elige el programador libremente. Siempre es recomendable que sea algo que indique para qué se va a usar la variable. Es decir, se debe huir de nombres como *a*, *b*, *c*, etc., y usar, por ejemplo, *edad*, *fecha_nacimiento*, *altura*, etc.

Las variables que se declaren en una subrutina o función sólo se usarán dentro de ellas. Cuando terminen se destruirán. Eso permite utilizar el mismo nombre de variable dentro de distintas subrutinas y su valor nunca se confundirá o mezclará.

Para declarar una variable al principio del *Módulo* o *Clase* usamos la sintaxis:

```
Module [STATIC] (PUBLIC | PRIVATE) nombre_variable AS  
tipo_variable
```

Si se define la variable como PRIVATE, estará disponible dentro de todo el fichero, pero no será accesible desde otros ficheros del mismo proyecto. Si se la declara como PUBLIC, se podrá acceder a la variable desde un fichero del proyecto distinto a donde se declaró.

La palabra STATIC se usa en los archivos de *Clase*, no en los *Módulos*. Sirve para definir un comportamiento especial en todos los objetos de una misma clase. En pocas palabras se podría explicar con un ejemplo: si tenemos una clase perro, tendremos algunas variables como color, raza, peso, etc., y cada objeto perro tendrá su propio valor en cada una de esas variables.

Al mismo tiempo, podemos declarar una variable que sea *número_de_patas*, de forma que si cambiamos su valor de 4 a 3, todos los objetos perros tendrán 3 patas y cada uno seguirá con su propio peso, color, etc. En este caso, la variable *número_de_patas* se declararía STATIC en el código BASIC del archivo de la clase perro. Se verá todo este comportamiento más adelante en este mismo capítulo.

Las constantes se definen sólo al principio de un archivo de *Módulo* o *Clase*, no se pueden definir dentro de las subrutinas y funciones. La sintaxis es:

```
( PUBLIC | PRIVATE ) CONST nombre_constante AS
tipo_constante = valor
```

Por tanto, al igual que las variables, pueden ser privadas o públicas.

Veamos un ejemplo de todo esto:

```
' Gambas module file
' Archivo de Módulo con el nombre: ejemploVariables
PRIVATE edad AS Date
PRIVATE altura AS Single
PRIVATE CONST Pi = 3.141592
PUBLIC calidad AS Integer
```

```

PUBLIC SUB Subrutina1()
    DIM num AS Integer
    DIM nombre AS String
    edad=30
    num = 54
END

PUBLIC SUB subrutina2()
    DIM num AS Integer
    DIM apellido AS String
    edad=32
    num = 4
END

```

En este ejemplo vemos que las variables `edad` y `altura` se pueden usar en todo el archivo. Por tanto, después de ejecutar la `Subrutina1`, `edad` valdrá 30 y, después de ejecutar la `Subrutina2` valdrá 32. Vemos también cómo la variable `num` está definida en las dos subrutinas. El valor de `num` desaparece al acabar cada una de las subrutinas y, por tanto, durante `Subrutina1` valdrá 54 y durante `Subrutina2` valdrá 4, pero después de que se ejecute el `END` de cada una de esas dos subrutinas, simplemente no existirá y si se hace referencia a `num` en cualquier otro punto del programa se producirá un error.

Finalmente, vemos que la variable `calidad` está definida como pública. Esto significa que desde cualquier otro archivo del programa se puede hacer referencia a ella mediante el nombre `ejemploVariables.calidad`, donde `ejemploVariables` es el nombre que se le ha dado al fichero donde se ha declarado `calidad`.

□ □ □ □ □ Subrutinas y funciones

Es impensable escribir todo el código de un programa sin una mínima organización. En BASIC el código se organiza dividiéndolo en procedimientos. Existen dos tipos de procedimientos: subrutinas y funciones. Una subrutina es un procedimiento que ejecuta algo, pero no devuelve ningún valor. Ejemplos de subrutinas

- serían procedimientos para dibujar algo en la pantalla, tocar un sonido, etc. Sin embargo, una función es un procedimiento que devuelve siempre un valor al terminar su ejecución. Ejemplos de función serían el cálculo de una operación matemática que devuelve un resultado, el proceso para pedir datos al usuario de la aplicación, etc.

Ya hemos visto en el capítulo anterior la sintaxis para declarar las subrutinas, puesto que se mostró cómo el entorno de desarrollo escribe automáticamente la subrutina que el programa ejecuta al hacer un clic del ratón sobre un botón. La sintaxis completa es:

```
(PUBLIC | PRIVATE) SUB nombre_de_la_subrutina (p1 As
    Tipo_de_Variable, p2 As Tipo_de_Variable, ...)
    ... código que ejecuta la subrutina
END
```

46

Las palabras PUBLIC y PRIVATE significan exactamente lo mismo que cuando se definen variables: determinan si la subrutina puede ser llamada sólo desde el archivo donde se ha codificado (PRIVATE) o desde cualquier archivo de la misma aplicación (PUBLIC).

Las variables p1, p2, etc., permiten pasarle parámetros a la subrutina, que se comportarán dentro de ella como variables declaradas dentro de la propia subrutina. Es decir, desaparecerán al ejecutar el END final. Se pueden pasar tantos parámetros como se desee a una subrutina, declarándolos todos, por supuesto.

Existen algunas subrutinas con nombres especiales en Gambas, por lo que el programador no debe usar esos nombres. Éstas son las siguientes:

- *Main*: existe en todas las aplicaciones de Gambas que sean de texto, no en las gráficas. Es el punto por el que empieza a ejecutarse el programa. Si no hubiera subrutina *Main*, Gambas daría un mensaje de error al intentar arrancar, puesto que no sabría por dónde comenzar.

- *_New* y *_free*: se ejecutan, respectivamente, al crearse y destruirse un objeto. Sólo se encuentran en los archivos de clase.
- *Objeto_Evento*: se ejecutan automáticamente cuando al Objeto le ocurre el Evento. Ya hemos visto algún ejemplo en el capítulo anterior, como *btrSalir_Click()*, que se ejecuta cuando el usuario de la aplicación hace clic con el ratón sobre el botón *btrSalir*. En las aplicaciones gráficas, el evento *Open* del formulario con el que se inicia la aplicación es la primera subrutina que el programa ejecutará. En el último apartado de este capítulo se tratarán específicamente los eventos, su significado y utilidad.

Veamos un programa de ejemplo (para probarlo, debemos crear un nuevo programa de texto siguiendo los pasos explicados en el *Capítulo 1*, apartado *El primer ejemplo*):

```
PUBLIC SUB Main()
    pinta_media(4,8)
END
PUBLIC SUB pinta_media(valor1 AS Integer, valor2 as
    Integer)
    PRINT (valor1 + valor2)/2
END
```

47

Aunque éste es un programa poco útil, sirve para expresar con simplicidad la forma de funcionar de las subrutinas. Comienza ejecutando la subrutina *Main*, en ella sólo hay una llamada a ejecutar la subrutina *pinta_media* pasándole los números enteros 4 y 8 como parámetro. La subrutina *pinta_media* muestra en la consola el resultado de hacer la media entre los dos valores que han sido pasados como parámetros.

La sintaxis para declarar una función es la siguiente:

```
(PUBLIC | PRIVATE) FUNCTION nombre_de_la_función
    (p1 As Tipo_de_Variable,p2 As Tipo_de_Variable ...)
        As Tipo_de_Dato
```

```

    ... código que ejecuta la función
    RETURN resultado_de_ejecutar la función
    END

```

La declaración es casi idéntica a la de la subrutina, añadiendo dos cosas más: el tipo de dato que la función devuelve en la primera línea y la necesidad de usar la sentencia RETURN de BASIC para indicar el valor a devolver.

Veamos otro ejemplo que produce el mismo resultado que el anterior, pero usando una función:

```

PUBLIC SUB Main()
    DIM final AS Single
    final = calcula_Media(4,8)
    PRINT final
END

PUBLIC FUNCTION calcula_Suma33(valor1 AS Integer,
    valor2 as Integer) AS Single
    RETURN (valor1 + valor2)/2
END

```

En esta ocasión es la subrutina Main la que se encarga de pintar en la consola el resultado de la operación. Es muy importante destacar la diferencia entre la forma en que se llama a una subrutina y se llama a una función. En el ejemplo anterior vimos que para llamar a la subrutina sólo se escribía su nombre con sus parámetros entre paréntesis. Sin embargo, ahora vemos que al llamar a la función se usa una asignación, final = calcula_Media(4,8). Esto debe hacerse siempre al llamar a una función: la asignación sirve para recoger el valor que se devuelve. Por motivos obvios, la variable que recoge el valor que la función retorna debe declararse del mismo tipo de dato que el que devuelve la función. En el ejemplo anterior la función devuelve un dato tipo Single (un número real con decimales) y la variable final se ha declarado, por tanto, de tipo Single.

2.2 Tipos de datos

Ya hemos visto a lo largo de los textos anteriores el uso de variables y cómo se declaran.

Hasta el momento, sólo conocemos algunos tipos de datos. Revisemos ahora todos los que soporta Gambas:

- *Boolean*: es un tipo de dato que suele ser el resultado de una comparación. Sólo acepta dos valores: *False* y *True* (Falso y Verdadero en español).
- *Byte*: representa un número entero positivo entre 0 y 255.
- *Short*: representa un número entero con valores posibles entre -32.768 y +32.767.
- *Integer*: representa un número entero con valores posibles entre -2.147.483.648 y +2.147.483.647.
- *Long*: representa un número entero con valores posibles entre -9.223.372.036.854.775.808 y +9.223.372.036.854.775.807.
- *Single*: representa un número real, con decimales, con valores posibles entre -1,7014118E+38 y +1,7014118E+38.
- *Float*: representa un número real, con decimales, con valores posibles entre -8,98846567431105E+307 y +8,98846567431105E+307.
- *Date*: sirve para almacenar un valor de fecha y hora. Internamente, la fecha y hora se almacena en formato UTC, al devolver el dato se representa en el formato local, según la configuración del ordenador.
- *String*: se usa para almacenar una cadena de texto. En BASIC las cadenas de texto se asignan mediante dobles comillas.

- *Variant*: significa *cualquier tipo de dato*, es decir, puede almacenar un *Integer*, *Single*, *String*, etc. Se debe evitar su uso porque ocupa más memoria que los anteriores y los cálculos con *Variant* son mucho más lentos.
- *Object*: representa cualquier objeto creado en Gambas.

Será el programador el que elija el tipo de dato con el que debe ser declarada una variable. Siempre se ha de tender a usar los tipos de datos más pequeños, puesto que ocupan menos memoria y el microprocesador los maneja con más velocidad. Sin embargo, eso puede limitar las opciones de la aplicación, por lo que a menudo se opta por tipos mayores para no cerrar posibilidades.

Por ejemplo, si se va a usar una variable para definir la edad de una persona, lo lógico es utilizar un dato de tipo *byte* (el valor máximo es 255). Sin embargo, si la edad a guardar es la de un árbol, es conveniente usar un tipo *Short*, ya que puede haber árboles con más de 255 años, pero no se conocen con más de 32.767.

60

□ □ □ □ □ Conversión de tipos

Gambas permite distintas conversiones entre tipos de datos. La forma de hacer la conversión puede ser implícita o explícita. Son conversiones implícitas cuando el propio intérprete de Gambas se encarga de gestionarlas. Por ejemplo:

```

    DIM resultado as Single
    DIM operando1 as Single
    DIM operando2 as Integer
    operando1= 3.5
    operando2=2
    resultado = operando1 * operando2

```

En este caso, para poder realizar la multiplicación, el intérprete convierte, de forma transparente para el programador, el *operando2* a un valor *Single*. Son conversiones explícitas las que debe hacer el programador al escribir el código para poder realizar operaciones, mezclar datos de distintos tipos, etc.

Estas conversiones se hacen mediante unas funciones que están incluidas en el BASIC de Gambas. Evidentemente, la conversión se hará siempre que sea posible, si no lo es, producirá un error. Éste es el listado de funciones de conversión existente:

- *CBool(expresión)*: convierte la expresión a un valor booleano, verdadero o falso. El resultado será falso si la expresión es falsa, el número 0, una cadena de texto vacía o un valor nulo. Será verdadero en los demás casos. Por ejemplo:
 - Devuelven *False* las siguientes operaciones: *Cbool("")*, *Cbool(0)*, *Cbool(NULL)*.
 - Devuelven *True* las operaciones: *Cbool(1)*, *Cbool("Gambas")*.
- *CByte(expresión)*: convierte la expresión en un valor tipo *Byte*. Primero se convierte la expresión a un número binario de 4 bytes. Si este número es mayor de 255, se corta recogiendo los 8 bits de menor peso. Por ejemplo, *Cbyte("17")* devuelve 17, pero *Cbyte(100000)* devuelve 160. El valor 160 es debido a que el número 100.000 en binario es 1100001101010000, sus 8 últimos bits son 10100000, que pasado de binario a decimal da lugar a 160. Si no sabemos operar con números binarios lo mejor que se puede hacer es evitar este tipo de conversiones que resultan en valores tan "sorprendentes".
- *CShort(expresión)*, *CIInt(expresión)* o *CIInteger(expresión)*, y *CLong(expresión)*: convierten, respectivamente, la expresión en un número de tipo *Short*, *Integer* y *Long*. En el caso de *CShort* la conversión se realiza igual que para *CByte*, pudiendo producirse resultados extraños igualmente si la expresión resulta en un número mayor de 32.767.
- *CDate(expresión)*: convierte la expresión en una fecha, pero hay que tener cuidado porque no admite el formato de fecha local, sólo el formato inglés *mes/día/año horas:minutos:segundos*. Es decir, *CDate("09/06/1972 01:45:12")* es el dia 6 de septiembre de 1972.

- *CSingle(expresión)* o *CSng(expresión)* y *CFloat(expresión)* o *Cflt(expresión)*: convierten, respectivamente, la expresión en un número del tipo *Single* y *Float*. La expresión debe usar el punto (.) y no la coma (,) como separador decimal.
- *CStr(expresión)*: convierte la expresión en una cadena de texto sin tener en cuenta la configuración local. Por tanto, *Cstr(1.58)* devuelve la cadena de texto *1.58*, independientemente de si la configuración local indica que el separador decimal es el punto o la coma, o *CStr(CDate("09/06/1972 01:45:12"))* devuelve "09/06/1972 01:45:12".
- *Str\$(expresión)*: convierte la expresión en una cadena de texto, teniendo en cuenta la configuración local. Por tanto, *Str\$(1.58)* devuelve la cadena de texto *1,58*, si la configuración local está en español. Del mismo modo, *Str\$(CDate("09/06/1972 01:45:12"))* devuelve "06/09/1972 01:45:12" si la configuración local está en español, puesto que en este idioma la costumbre es escribir las fechas en la forma día/mes/año.
- *Val(expresión)*: convierte una cadena de texto en un tipo *Boolean*, *Date* o alguno de los tipos numéricos, dependiendo del contenido de la expresión. *Val* es la función opuesta de *Str\$* y, por tanto, también tiene en cuenta la configuración local del ordenador en el que se ejecuta. Intenta convertir la expresión en un tipo *Date*, si no puede en un número con decimales, si tampoco puede en un número entero y si, finalmente, tampoco puede la convierte en un tipo *Boolean*.

□ □ □ □ □ Matrices

En numerosas ocasiones, cuando se intenta resolver un problema mediante la programación, surge la necesidad de contar con la posibilidad de almacenar distintos datos en una misma variable. La solución más simple para este problema son las *Matrices* o *Arrays*. Se pueden definir matrices que contengan cualquier tipo de datos, pero con la condición de que todos los elementos de la matriz sean del mismo tipo. No hay más límite en la dimensión de la matriz que la memoria del ordenador y la capacidad del programador de operar con matrices de dimensiones grandes.

La sintaxis para trabajar con matrices es la misma que para las variables, añadiendo entre corchetes las dimensiones de la matriz. Algunos ejemplos:

```
DIM Notas[2, 3] AS Single  
DIM Edades[40] AS Integer  
PRIVATE Esto_no_hay_quien_lo_maneje[4, 3, 2, 5, 6, 2]  
AS String
```

Para acceder al valor de cada una de las celdas de la matriz, hay que referirse siempre a su índice. En una matriz de dos dimensiones lo podemos identificar fácilmente por filas y columnas.

Hay que tener en cuenta que el índice comienza a contar en el 0, no en el 1. Es decir, en la matriz *Listado[3]* existirán los valores correspondientes a *Listado[0]*, *Listado[1]* y *Listado[2]*. Si se intenta acceder a *Listado[3]* dará un error *Out of Bounds*, fuera del límite. Por ejemplo:

```
Dim Alumnos[4,10] AS String  
Dim columna AS Integer  
Dim fila AS Integer  
Columna= 2  
Fila= 6  
Alumno[Columna, Fila] = "Manolo Pérez Rodríguez"
```

Hemos visto que hay que declarar estas matrices con las dimensiones máximas que van a tener. Eso supone que el intérprete de Gambas reserva la memoria necesaria para ellas al comenzar el uso del programa. Sin embargo, hay veces en que, por las características de la aplicación, se desconoce la dimensión que hará falta para la matriz. Para solventar esta posibilidad Gambas tiene predefinidas matrices unidimensionales dinámicas de todos los tipos de datos, excepto *Boolean*. Con estas matrices se trabaja de forma distinta a las anteriores, ya que hacen falta funciones para añadir nuevos elementos a la matriz, borrarlos o saber el número de elementos que tiene ésta.

Con este ejemplo veremos el uso y funciones más usuales al trabajar con matrices dinámicas:

```

DIM nombres AS String[]

' La siguiente instrucción inicializa nombres para
poder usarlo.

' Es un paso previo obligado:
nombres = NEW String[]

' Así podemos añadir valores a la matriz:
nombres.Add("Manolo")
nombres.Add("Juan")
nombres.Add("Antonio")

'Count devuelve el número de elementos de la matriz.

'La siguiente instrucción pintará 3 en la consola
PRINT nombres.Count

'La siguiente instrucción borrará la fila de "Juan":
nombres.Remove(1)

PRINT nombres.Count 'pintará 2

PRINT nombres[1] 'pintará "Antonio"

'La siguiente instrucción vaciará nombres:
nombres.Clear

PRINT nombres.Count 'pintará 0

```

2.3 Operaciones matemáticas

Cuando se trata de trabajar con números, Gambas tiene las operaciones habituales en casi todos los lenguajes de programación:

- +, -, * y / se usan, respectivamente, para la suma, resta, multiplicación y división.
- ^ es el operador de potencia, por ejemplo, $4^3 = 64$.

- Para la división hay dos operadores adicionales, \ o DIV y MOD, que devuelven, respectivamente, la parte entera del resultado de la división y el resto. Es decir, $9 \text{ DIV } 2 = 4$, $9 \backslash 2 = 4$ y $9 \text{ MOD } 4 = 1$.

Aparte de estos operadores existen las siguientes funciones matemáticas para realizar cálculos más complejos:

- *Abs(número)*: devuelve el valor absoluto de un número.
- *Dec(número)*: decrementa un número.
- *Frac(número)*: devuelve la parte decimal de un número.
- *Inc(número)*: incrementa un número.
- *Int(número)*: devuelve la parte entera de un número.
- *Max(número1, número2, ...)*: devuelve el número mayor.
- *Min(número1, número2, ...)*: devuelve el número menor.
- *Round(número, decimales)*: redondea un número con los decimales deseados.
- *Sgn(número)*: devuelve el signo de un número.
- *Rnd([mínimo],[máximo])*: devuelve un número aleatorio comprendido entre *mínimo* y *máximo*. Si no se expresa ningún valor para *mínimo* y *máximo*, el número estará comprendido entre 0 y 1. Si sólo se expresa un valor, el número estará comprendido entre el 0 y ese valor. Muy importante: antes de usar *Rnd* es necesario ejecutar la instrucción *Randomize* que inicializa el generador de números aleatorios. Si no se hace, se obtendrá el mismo número en sucesivas ejecuciones de *Rnd*.

■■■■■ Operaciones lógicas

Para realizar operaciones entre variables de tipo *Boolean* o expresiones cuyo resultado sea *Boolean*, existen algunas instrucciones similares a las que podemos ver en casi todos los lenguajes de programación. Se trata *AND*, *OR*, *NOT* y *XOR*. Si tenemos conocimientos de lógica y números binarios, no nos resultará difícil identificarlas y saber su comportamiento al tratarse de las operaciones binarias más básicas. En caso contrario, será fácil usarlas y entender su funcionamiento con una simple traducción al español de las tres primeras: *Y*, *O*, *NO*. Es decir, sirven para unir condiciones del tipo: color es verde y no es marrón, que se escribiría:

```
Color="Verde" AND NOT Color="Marrón"
```

El caso de *XOR* es más difícil de entender puesto que es una operación algo especial llamada *OR exclusivo*. El resultado de una operación *XOR* es verdadero cuando los dos operandos son distintos y, falso, cuando los dos operandos son iguales. En la práctica, esta operación se utiliza en cálculos con números binarios, en cuyo caso seguro que conocemos perfectamente su funcionamiento.

■■■■■ 2. 4 Manejo de cadenas

Una de las tareas más habituales en los programas informáticos es el uso de cadenas de texto, tanto si se trata de aplicaciones de bases de datos, como para la simple salida de mensajes en pantalla. En Gambas se han implementado todas las funciones de cadenas de texto del BASIC estándar más las que están presentes en Visual Basic. Antes de proceder a su listado, destacar que existe un "operador" de cadenas de texto que permite concatenarlas directamente, se trata del símbolo &. Veamos un ejemplo de su uso:

```
Dim Nombre AS String  
Dim Apellidos AS String  
Nombre = "Manuel"  
Apellidos = "Álvarez Gómez"  
PRINT Apellidos & ", " & Nombre
```

La salida en consola será:

Álvarez Gómez, Manuel

Veamos ahora el listado de las funciones disponibles para manejar cadenas de texto:

- *Asc(Cadena,[Posición]):* devuelve el código ASCII¹ del carácter que está en la posición indicada en la cadena dada. Si no se da la posición, devuelve el código del primer carácter.
- *Chr\$:* devuelve un carácter a partir de su código ASCII. Esta función es útil para añadir caracteres especiales a una cadena de texto, por ejemplo:

```
PRINT "Manuel" & Chr$(10) & "Antonio"
```

insertará una tabulación entre los dos nombres, ya que en la tabla ASCII el código 10 corresponde a un avance de línea (*Line Feed*).

57

- *InStr (Cadena, Subcadena [, Inicio]):* busca la subcadena dentro de la cadena y devuelve un número con la posición donde la encontró. Si se da el valor *Inicio*, la búsqueda empezará en esa posición. Por ejemplo:

```
PRINT Instr("Gambas es basic", "bas", 5)
```

devuelve un 11, mientras que:

```
PRINT Instr("Gambas es basic", "bas")
```

devuelve un 4.

- *RinStr (Cadena, Subcadena [, Inicio]):* funciona igual que *InStr*, sólo que empieza a buscar de derecha a izquierda en la cadena.

- *Lcase\$(Cadena)*: convierte una cadena a minúsculas.
- *Ucase\$(cadena)*: convierte una cadena a mayúsculas.
- *Left\$(cadena,n)*: devuelve los primeros *n* caracteres de una cadena.
- *Right\$(cadena,n)*: devuelve los últimos *n* caracteres de una cadena.
- *Mid\$(Cadena, posición [,n])*: devuelve una cadena que empieza en *posición* y mide *n* caracteres. Si no se especifica *n*, la cadena devuelta contendrá todos los caracteres a partir de *posición*.
- *Replace\$(Cadena, patrón, reemplazo)*: reemplaza, dentro de *Cadena*, cada vez que encuentra el texto *patrón* por el texto *reemplazo*. Por ejemplo:

```
PRINT Replace$("LinEx es una distribución de Linux",
               "Lin", "gnu/Lin")
```

pinta el texto:

"gnu/LinEx es una distribución de gnu/Linux"

- *Subst\$(Patrón, reemplazo1[, reemplazo2,...])*: sustituye en la cadena *patrón* los símbolos $\&1$, $\&2$... por las cadenas *reemplazo1*, *reemplazo2*... Por ejemplo:

```
PRINT Subst$("El alumno &1 tiene &2 años", "Manuel", "8")
```

da como salida la cadena:

"El alumno Manuel tiene 8 años".

Evidentemente, el potencial de esta instrucción se muestra cuando las cadenas *reemplazo* son variables de tipo *String*.

- *Len(cadena)*: devuelve la longitud de una cadena.
- *Ltrim\$(cadena)*: elimina los espacios en blanco de la parte inicial de una cadena.
- *Rtrim\$(cadena)*: elimina los espacios en blanco de la parte final de una cadena.
- *Trim\$(cadena)*: elimina todos los espacios en blanco de una cadena.
- *Space\$(n)*: devuelve una cadena que contiene *n* espacios en blanco.
- *String\$(n, patrón)*: devuelve una cadena que contiene el *patrón* concatenado *n* veces.
- *Split(Cadena [, Separadores , Escape , Ignorar_nulos])*: devuelve una matriz de cadenas de texto. Las partes de esta matriz se obtienen al trocear la *Cadena* cada vez que se encuentra uno de los *Separadores*. Si no se especifica ninguno, el separador se considera la coma (,). Si en algún caso se quiere incluir el *Separador* en alguna de las cadenas separadas, habría que colocar el *Escape* marcando el principio y final de la parte en la que debe ignorarse el separador.
- *Ignorar_nulos* es un valor *True* o *False*, e indica si se deben ignorar o no las cadenas que resulten vacías como consecuencia del troceado. Veamos un ejemplo que explique mejor el funcionamiento:

```
DIM texto as String[]  
DIM texto2 as String[]  
texto = Split("Había una vez un circo", " ")  
texto2 = Split("Había una vez iun circoi", " ", "i")  
PRINT texto[0] & "," & texto[1] & "," & texto[2] & ","  
& texto[3]  
PRINT texto2[0] & "," & texto2[1] & "," & texto2[2]  
& "," & texto2[3]
```

devuelve:

```
Había,una,vez,un  
Había,una,vez,un circo
```

En el segundo caso se puede ver cómo, aunque el separador es el espacio en blanco, no se ha separado el texto **un circo** al estar rodeado del carácter de escape.

2.5 Control de flujo

60

Son muchas las ocasiones en que el flujo en que se ejecutan las instrucciones en un programa no es adecuado para resolver problemas.

Todo el código BASIC que hemos visto hasta ahora ejecuta sus instrucciones de arriba abajo, según se las va encontrando. Sin embargo, con frecuencia, habrá que volver atrás, repetir cosas, tomar decisiones, etc. Este tipo de acciones se denomina *control de flujo* y el BASIC de Gambas proporciona un buen juego de sentencias para manejarlo.

IF ... THEN ... ELSE

Es la sentencia más común para tomar una decisión: si se cumple una condición, entonces se ejecuta algo, en caso contrario, se ejecuta otra cosa.

Su forma más básica es:

```
IF Expresión THEN  
    ...  
ENDIF
```

O si lo que se ejecuta es una sola instrucción:

```
IF Expresión THEN sentencia_a_ejecutar
```

La sintaxis completa de la instrucción es:

```
IF Expresión [ { AND IF | OR IF } Expresión ... ] THEN  
    ...  
    [ ELSE IF Expresión [ { AND IF | OR IF } Expresión ... ]  
    THEN  
        ... ]  
    [ ELSE  
        ... ]  
ENDIF
```

Algunos ejemplos de uso:

```
DIM Edad AS Integer  
.....  
IF Edad > 20 THEN  
    PRINT "Adulto"  
ENDIF  
IF Edad > 20 THEN PRINT "Adulto"  
IF Edad < 2 AND Edad >0 THEN  
    PRINT "Bebé"  
ELSE IF Edad < 12 THEN  
    PRINT "Niño"  
ELSE IF Edad < 18 THEN  
    PRINT "Joven"  
ELSE  
    PRINT "Adulto"  
ENDIF
```

61

Dependiendo del valor que tuviera la variable Edad al llegar al primer IF, se imprimirá un resultado distinto.

□ □ □ □ □ Select

En el caso anterior vimos que en ocasiones el flujo del programa necesita revisar varias condiciones sobre una misma variable, produciendo un *IF* dentro de otro (*IF* anidados). Esa estructura no es cómoda de leer ni produce un código limpio. Para estos casos existe la sentencia SELECT, que es mucho más apropiada. Su sintaxis es:

```
SELECT [ CASE ] Expresión
  [ CASE Expresión [ TO Expresión #2 ] [ , ... ] ]
  ...
  [ CASE Expresión [ TO Expresión #2 ] [ , ... ] ]
  ...
  [ { CASE ELSE | DEFAULT }
  ...
END SELECT
```

Véamnos cómo se aplica al mismo ejemplo anterior de las edades:

62

```
DIM Edad AS Integer
....
SELECT CASE edad
CASE 0 TO 2
  PRINT "Bebé"
CASE 2 TO 12
  PRINT "Niño"
CASE 18
  PRINT "Bingo, ya puedes votar"
CASE 13 TO 17
  PRINT "Joven"
CASE ELSE
  PRINT "Adulto"
END SELECT
```

Se trata de un código mucho más fácil de leer que el anterior.

□ □ □ □ □ FOR

Cuando se hace necesario contar o realizar una acción un número determinado de veces, la sentencia FOR es la solución:

```
FOR Variable = Expresión TO Expresión [ STEP Expresión ]  
    ...  
NEXT
```

El bucle incrementa la Variable de uno en uno, a no ser que se especifique un valor a STEP. Se pueden especificar valores negativos, de forma que se convertiría en una cuenta atrás. Por ejemplo:

```
DIM n AS Integer  
FOR n = 10 TO 1 STEP -1  
    PRINT n  
NEXT
```

63

Si se quiere interrumpir un bucle en algún punto, se puede usar la sentencia BREAK:

```
DIM n AS Integer  
FOR n = 10 TO 1 STEP -1  
    IF n=3 THEN BREAK  
    PRINT n  
NEXT
```

El bucle acabaría cuando n valiera 3 y no se escribirían los últimos 3 números. También se dispone de la sentencia CONTINUE, que permite saltarse pasos en el bucle:

```
DIM n AS Integer  
FOR n = 1 TO 4  
    IF n=2 THEN CONTINUE  
    PRINT n  
NEXT
```

Se saltaría el 2 al escribir los valores de **n**. Existe una variante del bucle FOR que se usa al recorrer elementos de una colección, como una matriz. La sintaxis en este caso es:

```
FOR EACH Variable IN Expresión  
...  
NEXT
```

Pongamos un ejemplo usando las matrices dinámicas que ya hemos visto en este capítulo:

```
DIM Matriz AS String[]  
DIM Elemento AS String  
  
Matriz = NEW String[]  
Matriz.Add("Azul")  
Matriz.Add("Rojo")  
Matriz.Add("Verde")  
  
FOR EACH Elemento IN Matriz  
    PRINT Elemento;  
NEXT
```

64

Escribiría como salida: *AzulRojoVerde*.

□ □ □ □ WHILE y REPEAT

Cuando se quiere repetir la ejecución de una porción de código en varias ocasiones dependiendo de una condición, tenemos dos instrucciones distintas: WHILE y REPEAT.

Su comportamiento es casi idéntico. La diferencia estriba en que si la condición necesaria para que se ejecute el código es falsa desde el principio, con REPEAT se ejecutará una vez y con WHILE no se ejecutará nunca. La sintaxis de ambas es:

```
WHILE Condición  
    ... instrucciones  
WEND
```

y

```
REPEAT  
    ...instrucciones  
UNTIL Condición
```

En el caso del bucle WHILE existe una variante de la sintaxis consistente en sustituir WHILE por DO WHILE y WEND por LOOP. Es exactamente lo mismo; depende del programador elegir un formato u otro. Veamos un ejemplo:

```
DIM a AS Integer  
a = 1  
WHILE a <= 10  
    PRINT "Hola Mundo "; a  
    INC a  
WEND  
a = 1  
  
REPEAT  
    PRINT "Hola Mundo "; a  
    INC a  
UNTIL a > 10
```

65

Este ejemplo producirá el mismo resultado en la ejecución del bucle WHILE que en el del REPEAT, en ambos casos escribirá diez veces “Hola Mundo” junto al valor de la variable a que se irá incrementando de 1 a 10. El uso de estas estructuras puede ser peligroso. Si durante la ejecución del bucle no hay forma de que la condición pase de ser verdadera a falsa, estaríamos ante un bucle infinito y el programa entraña en situación de bloqueo.

■ ■ ■ ■ ■ Depuración en el IDE de Gambas

Una vez escrito parte del código de un programa, lo usual es comprobar si funciona, pero lo habitual es que la mayor parte de las veces no lo haga en el primer intento. Tanto si se es un programador experimentado como si no, los fallos son parte de la rutina. Saber encontrarlos y corregirlos es lo que se denomina depuración, y es una de las tareas más importantes a realizar. Cuando son fallos de sintaxis, el entorno de desarrollo suele darnos mensajes indicativos del problema, parando la ejecución en la línea en que se produce.

Cuando se adquiere una cierta soltura con el lenguaje, los fallos de sintaxis son cada vez menos comunes, pero los fallos en la lógica de la ejecución del programa se producirán siempre. Cuando esa lógica pasa, además, por instrucciones de control de flujo como las que hemos visto en este capítulo, la dificultad en encontrar los errores es mayor, puesto que la aplicación transcurre en distintas ocasiones por la misma porción de código y es posible que el fallo no se produzca la primera vez que se ejecute ese código.

66

Para facilitar esta tarea, el IDE de Gambas dispone de distintas herramientas de depuración. La primera de ellas es la posibilidad de fijar puntos de interrupción. Es decir, señalar sitios en los que el programa se parará para permitir ver el estado de las variables y en qué punto de la ejecución se encuentra.

Para fijar un punto de interrupción en una línea de código, tan sólo hay que colocar el cursor del ratón en esa línea y pulsar la tecla F9 o el botón con el símbolo de la mano levantada, que está en la parte superior de la ventana de código. Las líneas en las que se fija un punto de interrupción tienen el fondo en rojo.

```

END
PUBLIC SUB btnRun_Click()
    DIM rData AS Result
    DIM hForm AS FRequest
    rData = $hConn.Exec(txtRequest.Text)
    hForm = NEW FRequest($hConn, rData)
    hForm.Show
CATCH
    Message.Error(Error.Text)
END

```

Figura 1. Punto de interrupción en la línea de código.



Figura 2. Ejecución del programa desde la ventana Proyecto.

La misma operación que crea el punto la elimina, es decir, pulsando F9 de nuevo el fondo rojo desaparecerá. La ejecución del programa, como se explicó en el capítulo anterior, se arranca pulsando en el símbolo verde de Play de la ventana de Proyecto (o pulsando la tecla de función F5). Junto al botón verde se encuentra un botón de Pausa, que permite parar la ejecución, y otro más de Stop que permite detenerla en cualquier momento.

Si se quiere correr la aplicación ejecutando una a una las instrucciones para ir observando por dónde transcurre el flujo del programa, se puede pulsar la tecla de función F8 o cualquiera de los dos botones que se encuentran a la derecha del símbolo de Stop. Haciendo esto, el entorno de desarrollo saltará a la primera línea que se deba correr e irá ejecutando línea a línea cada vez que pulsemos F8, o el ícono que muestra la flecha entrando entre las llaves, mencionado anteriormente. El botón que está justo a la derecha del Stop y que muestra una flecha saltando por encima de las llaves, parece producir el mismo efecto (su tecla rápida es Mayúsculas + F8), pero no es así: el comportamiento cambia cuando el programa llegue a una llamada, a una subrutina o a una función. En este caso, este ícono ejecutará la llamada y todo lo que tenga que hacer la función o subrutina en un solo paso, sin que veamos el flujo del programa por el procedimiento. Si hubiéramos pulsado F8 habríamos entrado en la subrutina y visto también paso a paso cómo se ejecutan las instrucciones. Por tanto, con estos dos botones podemos elegir cuando lleguemos a la llamada a un procedimiento, si queremos depurar también ese procedimiento o simplemente ejecutarlo y pasar a la siguiente línea de código.

Finalmente, cuando pausamos la ejecución del programa, aparecen tres nuevas pestañas en la parte inferior de la ventana de proyecto. En la pestaña Local se ven todas las variables del procedimiento que se está ejecutando y el valor que tienen en ese momento. En la pestaña Pila se ve el listado de llamadas entre procedimientos que se han producido hasta llegar a ese punto del programa. Así, podemos saber a través de

qué pasos se ha llegado a esa instrucción. Finalmente, en la pestaña Observar podemos introducir cualquier expresión en BASIC, incluyendo operaciones con las variables que el programa tenga declaradas para ver cuál es el resultado o el valor que tienen en el momento de la pausa.

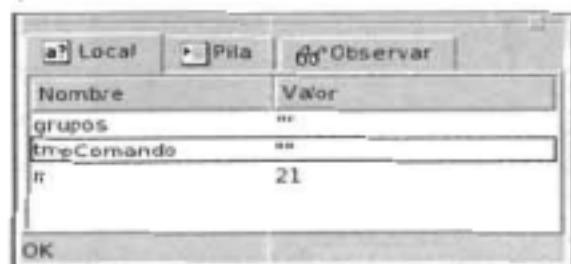


Figura 3. Pestañas Local, Pila y Observar de la ventana Proyecto.

2. 6 Entrada y salida: ficheros

68

En este apartado veremos la forma más común de trabajar con ficheros en Gambas. Gambas trata los ficheros como un flujo de datos (la palabra exacta para esto es *Stream*), lo que tiene una implicación muy cómoda: todos los flujos de datos se tratan de igual manera, con lo que el código para manejar un fichero es igual al código para manejar una conexión de red o un puerto de comunicaciones, puesto que todos son objetos de tipo *Stream*. Las operaciones tradicionales con un fichero son abrirlo, crearlo, escribir y leer datos. Veamos cómo se usan:

```
Archivo = OPEN Nombre_de_Archivo FOR [ READ | INPUT ]
[ WRITE | OUTPUT ] [ CREATE | APPEND ] [ WATCH ]
```

Abrimos un archivo con distintas finalidades:

- **READ** o **INPUT**: para leer datos, en el primer caso no se usa buffer de datos, con **INPUT** sí que hay un buffer intermedio.
- **WRITE** o **OUTPUT**: para escribir datos, con **WRITE** no hay buffer de datos, con **OUTPUT** sí se usa.
- **CREATE**: si el fichero no existe se crea. Si no se usa esta palabra, el fichero debe existir antes de abrirlo o dará un error.

- APPEND: los datos se añaden al final del fichero.
- WATCH: si se especifica esta palabra, Gambas lanzará los eventos (que veremos más adelante) *File_Read* y *File_Write* en caso de que se pueda leer y escribir en el archivo.

Ahora, cerremos un Archivo que ha sido abierto con la sentencia OPEN.

CLOSE [#] Archivo

Escribimos, convirtiendo en cadena de texto, la Expresión en el Archivo abierto anteriormente.

PRINT [#Archivo ,] Expresión]

Si no se especifica ningún archivo, tecleamos la expresión en la consola, como se ha visto en distintos ejemplos a lo largo de todo este capítulo. La instrucción PRINT admite un cierto control de cómo se colocan las expresiones, dependiendo de algunos signos de puntuación que se pueden colocar al final de la sentencia:

- Si no hay nada después de la *Expresión*, se añade una nueva línea al final. Por tanto, la salida de la siguiente instrucción *PRINT* será en una línea nueva.
- Si hay un punto y coma detrás de la *Expresión*, la siguiente instrucción *PRINT* se escribirá justo detrás de la salida anterior, sin espacios, líneas o tabulaciones intermedias.
- Si hay un punto y coma doble, se añade un espacio entre las expresiones, lo que permite concatenar expresiones en una misma línea usando distintas sentencias *PRINT*.
- Si se utiliza una coma en lugar de un punto y coma, se añade una tabulación, con lo que también se pueden concatenar expresiones en una misma línea.

Seguidamente, escribimos, sin convertir en cadena de texto, la Expresión en el Archivo abierto anteriormente. Es una instrucción que suele usarse en lugar de *PRINT* cuando los datos a escribir no son cadenas de texto, como en el caso de archivos binarios.

```
WRITE [ #Archivo , ] Expresión [ , Longitud ]
```

En cualquier caso, también puede usarse con cadenas de texto y permite indicar, con el parámetro **Longitud**, el número de caracteres que se desean sacar en cada operación de escritura. Al contrario que con *PRINT*, no se pueden usar signos de puntuación para controlar la posición de la escritura.

```
INPUT [ #Archivo , ] Variable1 [ , Variable2 ... ]
```

Leemos, desde un Archivo, un dato y asignamos su valor a **Variable1**, **Variable2**, etc. Los datos deben estar separados en el archivo por comas o en distintas líneas. Si no se especifica el Archivo, leemos los datos desde la consola, esperando que el usuario de la aplicación los introduzca.

```
READ [ #Archivo , ] Variable [ , Longitud ]
```

Se puede decir que es la instrucción opuesta a *WRITE*. Leemos del Archivo datos binarios y les asignamos su valor a la **Variable**. Si ésta es una cadena de texto, se puede fijar la **Longitud** de la cadena a leer.

```
LINE INPUT [ #Archivo , ] Variable
```

En una línea de texto entera del Archivo, le asignamos a la **Variable**. No se debe usar para leer de flujos binarios.

```
Eof ( Archivo )
```

Devuelve *True* (verdadero) cuando se llega al final del Archivo y *False* (falso) en caso contrario.

Lof (Flujo)

Si el Flujo de datos es un archivo, devuelve su tamaño. Si en lugar de un archivo es un *Socket* de una conexión de red o un objeto *Process*, devuelve el número de bytes que pueden leerse de una sola vez. Una vez vistas las sentencias más comunes para el manejo de flujos de datos, veamos algunos ejemplos de uso:

```
' Leer datos de un puerto serie:  
' (Requiere seleccionar el componente gb.net en el  
proyecto)  
DIM Archivo AS File  
Archivo = OPEN "/dev/ttyS0" FOR READ WRITE WATCH  
...  
' El evento File_Read se produce cuando haya datos  
que leer:  
PUBLIC SUB File_Read()  
    DIM iByte AS Byte  
    READ #Archivo, iByte  
    PRINT "Tengo un byte: "; iByte  
END
```

71

```
'Lee el contenido del fichero /etc/passwd y lo  
muestra en la consola:  
DIM Archivo AS File  
DIM Linea AS String  
Archivo = OPEN "/etc/passwd" FOR INPUT  
  
WHILE NOT Eof(Archivo)  
    LINE INPUT #Archivo, Linea  
    PRINT Linea  
WEND  
CLOSE Archivo
```

2.7 Control de errores

Es seguro que, en algún momento, en la mayor parte de los programas, bien por acciones del usuario del programa, bien por el propio flujo de la ejecución, se producen errores, como intentar borrar un fichero que no existe, hacer una división por cero, conectar a un servidor web que no responde, etc. En todos estos casos, Gambas muestra un mensaje en pantalla y, a continuación, el programa se rompe y deja de funcionar. Es evidente que ése es un comportamiento que el desarrollador no desea y debe poner las medidas oportunas para evitarlo. La forma de hacerlo es implementando un control de errores para que la aplicación sepa lo que debe hacer en esos casos. Gambas implementa las instrucciones necesarias para capturar los errores y procesarlos según los deseos del programador. Las instrucciones para ello son:

72

- **TRY Sentencia:** ejecuta la sentencia sin lanzar el error aunque se produzca, el programa continúa por la instrucción que haya después del TRY, tanto si hay error como si no. Se puede saber si existe error consultando la sentencia *ERROR* que valdrá verdadero o falso. Por ejemplo:

```
' Borrar el archivo aunque no exista
TRY KILL "/tmp/prueba/"
' Comprobar si ha tenido éxito
IF ERROR THEN PRINT "No fue posible eliminar el archivo"
```

- **FINALLY:** se coloca al final de un procedimiento. Las instrucciones que siguen a continuación se ejecutan siempre, tanto si ha habido un error en el procedimiento como si no lo ha habido.
- **CATCH:** se coloca al final de un procedimiento. Las instrucciones que siguen a continuación se ejecutan sólo si se ha producido un error en la ejecución del procedimiento (incluyendo si el error se ha producido en subrutinas o funciones llamadas desde el mismo procedimiento). Si existe una instrucción *FINALLY*, ha de colocarse delante de *CATCH*.

Veamos un ejemplo de FINALLY y CATCH:

```
' Mostrar un archivo en la consola
SUB PrintFile(Nombre_Archivo AS String)
    DIM Archivo AS File
    DIM Linea AS String

    OPEN Nombre_Archivo FOR READ AS #Archivo

    WHILE NOT EOF(Archivo)
        LINE INPUT #Archivo, Linea
        PRINT Linea
    WEND

    FINALLY ' Siempre se ejecuta, incluso si hay error
        CLOSE # Archivo
        .
    CATCH ' Se ejecuta sólo si hay error
        PRINT "Imposible mostrar el archivo "; Nombre_Archivo
    END
```

73

2.8 Programación orientada a objetos con Gamas

BASIC es un lenguaje de programación estructurada. Esto significa que los programas tienen el flujo que hemos visto hasta ahora: empiezan en un punto de una subrutina y van ejecutando las instrucciones de arriba a abajo, con los saltos correspondientes a las llamadas a procedimientos y distintas funciones de control del flujo como bucles, condiciones, etc.

Este tipo de programación permite resolver la mayor parte de los problemas y, de hecho, se ha estado usando durante muchos años. A día de hoy, se siguen desarrollando

aplicaciones con lenguajes, como BASIC o C, de programación estructurada. Sin embargo, desde finales de los años 70 se ha venido trabajando también en otro paradigma de programación: la programación orientada a objetos. Los lenguajes que adoptan este paradigma, como Smalltalk, Java o C++, intentan modelar la realidad. La ejecución de estos programas se basa en la interacción de los distintos objetos que definen el problema, tal y como ocurre en la vida normal, en la que los objetos, personas y animales nos movemos, enviamos mensajes unos a otros y ejecutamos acciones. La programación orientada a objetos se usa cada día con más frecuencia porque permite una mejor división de las tareas que un programa debe hacer, facilita la depuración y la colaboración entre distintos programadores en los proyectos que son de gran tamaño y, en muchos aspectos, tiene un potencial mucho mayor que la programación estructurada.

Por todas estas razones, es común que se añadan características de programación orientada a objetos a lenguajes que, como BASIC, en un principio no la tenían. Gambas permite hacer programas estructurados a la vieja usanza, escribiendo el código en *Módulos*, como hemos visto. Y también posibilita la programación orientada a objetos mediante el uso de *Clases*. Es más, para la programación gráfica y el uso de la mayor parte de los componentes que se añaden al lenguaje, no hay más alternativa que el uso de objetos.

Podemos ver cómo funciona todo esto en Gambas pensando en el caso de que tuviéramos que escribir un programa que simulara el comportamiento de un coche. Usando la programación orientada a objetos, definiríamos cada una de las partes del coche mediante un archivo de clase en el que escribiríamos el código BASIC necesario para definir las características de esa parte y la interfaz con la que se comunica con el mundo real. Por ejemplo, definiríamos cómo es un volante, cómo es una rueda, cómo es un asiento, un acelerador, un motor, etc. Después, y basándonos en esa definición, crearíamos cada uno de los objetos para crear el coche: un volante, cuatro ruedas, varios asientos, un motor, un acelerador, etc. Cada uno de esos objetos respondería a ciertos mensajes. Por ejemplo, el volante respondería a un giro actuando sobre el eje de las ruedas, el motor respondería incrementando sus revoluciones si recibe una presión del acelerador, etc.

Cada vez que creamos un objeto basándonos en el archivo de clase que lo ha definido, decimos que el objeto ha sido *instanciado*. Se pueden instanciar tantos objetos como se desee a partir de una clase y una vez creados tienen vida propia, son independientes unos de otros con sus propias variables internas y respondiendo a las distintas acciones según hayan sido definidos todos en la clase.

Otra de las características de la programación orientada a objetos es la *Herencia*. Cuando un objeto hereda de otro objeto significa que es del mismo tipo, pero que puede tener características añadidas. Por ejemplo, supongamos que definimos la clase *cuadro_de_texto* con ciertas características como tamaño de texto, longitud, etc. A continuación, podemos crear objetos de esa clase y son *cuadros_de_texto*. Con Gambas podemos, además, crear una nueva clase, por ejemplo: *cuadro_de_texto_multilínea* que herede de *cuadro_de_texto*. Eso significaría que un *cuadro_de_texto_multilínea* es un *cuadro_de_texto* al que se le añaden más cosas. Todo el comportamiento y propiedades del *cuadro_de_texto* ya están codificados y no hay que volver a hacerlo.

Veamos un ejemplo sencillo que aclare algunos conceptos, para lo que vamos a crear en el entorno de desarrollo un nuevo proyecto de programa de texto. A continuación, le añadimos un *Módulo* con un nombre cualquiera y dos archivos de clase, a uno lo llamamos *SerVivo* y a otro *Hombre*.

75

Este es el código que debemos escribir en el archivo de clase *SerVivo.cls*:

```
' Gambas class file
PRIVATE patas AS Integer
PRIVATE nacimiento AS Integer
PUBLIC SUB nacido(fecha AS Date)
    nacimiento = Year(fecha)
END

PUBLIC SUB PonePatas(numero AS Integer)
    Patas = numero
END
```

```

PUBLIC FUNCTION edad() AS Integer
    RETURN Year(Now) - nacimiento
END

PUBLIC FUNCTION dicePatas() AS Integer
    RETURN patas
END

```

Este es el código a escribir en el archivo de clase *Hombre.cls*:

```

' Gambas class file
INHERITS SerVivo
PRIVATE Nombre AS String
PRIVATE Apellido AS String

PUBLIC SUB PoneNombre(cadena AS String)
    Nombre = cadena
END

PUBLIC SUB PoneApellido(cadena AS String)
    Apellido = cadena
END

PUBLIC FUNCTION NombreCompleto() AS String
    RETURN Nombre & " " & Apellido
END

```

Y éste el código a escribir en el Módulo que hayamos creado:

```

' Gambas module file
PUBLIC SUB Main()
DIM mono AS SerVivo
DIM tipejo AS Hombre

```

```

mono = NEW SerVivo
mono.nacido(CDate("2/2/1992"))
mono.PonePatas(3)
PRINT mono.edad()
PRINT mono.dicePatas()

tipejo = NEW hombre
tipejo.nacido(CDate("2/18/1969"))
tipejo.PoneNombre("Vicente")
tipejo.PoneApellido("Pérez")
PRINT tipejo.edad()
PRINT tipejo.NombreCompleto()
END

```

Veamos los tres archivos, empezando con el de clase *SerVivo.cls*. Tiene un código muy simple: declara un par de variables, un par de subrutinas con las que se puede asignar el valor a esas variables y dos funciones con las que devuelve valores de algunos cálculos realizados sobre las variables y la fecha actual.

El archivo *Hombre.cls* es muy similar, pero con una sentencia nueva: al inicio del fichero se ha colocado la instrucción INHERITS *SerVivo*. Al haber hecho eso estamos diciendo que todos los objetos que se instancien de la clase *Hombre* serán objetos *SerVivo* y, por tanto, tendrán también las mismas funciones que un *SerVivo* y podrán realizar las mismas operaciones.

Si hubieran sido necesarias algunas labores de inicialización de los objetos al ser creados, se harían añadiendo al archivo de clase una subrutina con la sintaxis: PUBLIC SUB *_New()*. Todo el código que se escriba ahí se ejecutará cada vez que se cree un objeto.

Y, finalmente, el módulo y su procedimiento Main. Ahí se declaran dos variables: *mono* y *tipejo*, pero en lugar de declararlos como uno de los tipos de datos que vimos en los primeros apartados de este capítulo, se declaran como objetos de la

clase *SerVivo* y *Hombre*. Por tanto, hablando con propiedad, *mono* y *tipejo* no son variables, sino objetos. Las clases corresponden a los nombres de los dos archivos de clase que hemos definido. Además, vemos que para crear un objeto de una clase es necesario usar siempre la palabra NEW, lo que provocará además que se ejecute el código de la subrutina *_New* del archivo de clase, si esta subrutina existiera. Una vez que el objeto ha sido creado, podemos acceder a sus subrutinas y funciones escribiendo el nombre de un objeto y el procedimiento separados por un punto. Sólo podemos acceder a los procedimientos que hayan sido declarados en el archivo de clase como PUBLIC.

A parte del ahorro de código obvio que supone para la clase *Hombre* no tener que escribir las funciones que hace un *SerVivo*, hay algunas consecuencias más importantes que se deducen de este ejemplo. Tal y como está escrito, se podría alterar el código de la clase *SerVivo*, modificando y cambiando variables, y el programa seguiría funcionando igual, sin tener que tocar en absoluto la clase *Hombre*.

78

Es decir, se puede, por ejemplo, mejorar el método para calcular la edad, teniendo en cuenta el día de nacimiento dentro del año. También se puede cambiar el nombre de las variables y no afecta a la clase *Hombre* y al resto del programa. Otra de las posibilidades que existe es el uso de los archivos de clase tal y como están en otro proyecto donde, del mismo modo, se necesiten seres vivos, reutilizando de forma sencilla el código ya escrito.

Se han elaborado multitud de libros y artículos sobre la programación orientada a objetos. La pretensión de este apartado es únicamente servir de mera introducción general y explicar la sintaxis necesaria para su uso con Gambas. La mejor forma de aprender, disfrutar de ella y obtener todo su potencial es probando y viendo ejemplos que la usen extensivamente.

El código fuente del entorno de desarrollo de Gambas hace un uso muy amplio de clases y objetos, con lo que es una fuente completa de ejemplos. Está disponible dentro del directorio *apps/src/gambas2* del fichero comprimido que contiene los fuentes de Gambas.

2.9 Propiedades, Métodos y Eventos

Los componentes que extienden las posibilidades de Gambas son de distintos tipos, pero todos ellos contienen clases que definen distintos objetos. En el caso de los componentes que sirven para crear interfaces gráficas, estos objetos incorporan un ícono a la Caja de Herramientas del entorno de desarrollo. No hay que crear este tipo de objetos escribiendo el código correspondiente y la palabra *NEW*, sino sólo dibujándolos sobre un formulario después de haber pulsado el botón con el ícono respectivo. Es el entorno de desarrollo el que se encarga de escribir el código necesario para crearlos.

Cualquiera de los objetos que se crean a partir de los componentes tienen ya un comportamiento definido en su clase. Este comportamiento incluye la interfaz con la que los objetos se van a comunicar unos con otros en el programa, y lo necesario para que el programador pueda definirlos y hacerlos actuar a su conveniencia. Para facilitar esas tareas, estos objetos disponen de *Propiedades, Métodos y Eventos*. Las *propiedades* permiten cambiar parámetros del objeto. En realidad, al darle un valor a una propiedad no estamos sino asignando valores a algunas variables del objeto que éste interpreta internamente para producir un efecto. Por ejemplo, podemos asignarle a un formulario un valor numérico a la propiedad *Background*, y con ello le cambiaremos el color del fondo. Para modificar las propiedades disponemos de la ventana de *Propiedades* en el IDE (visible al pulsar **F4** o desde el menú de la ventana de Proyecto: **Vista | Propiedades**), que nos permite hacerlo con una interfaz gráfica sencilla. También podemos cambiar una propiedad mediante código haciendo referencia al nombre del objeto y a la propiedad, separados por un punto. Por ejemplo: *Form1.Background=0*, dejaría el fondo del formulario *Form1* en color negro.

Los *métodos* son las funciones que el objeto puede realizar. Sólo pueden asignarse mediante código y cada objeto tiene su propia colección de métodos. Por ejemplo: *btnSalir.Hide* haría que el botón *btnSalir* se ocultara. Finalmente, los *eventos* son subrutinas que se ejecutan para indicar algo que le ha ocurrido al objeto. Un ejemplo que ya vimos en el capítulo anterior es el evento *Click* del botón que se ejecuta cuando hacemos clic con el ratón. Los eventos son, en las aplicaciones gráficas, los que marcan el flujo del programa. De hecho, con frecuencia se dice que Gambas es

un lenguaje de programación orientado a eventos. Es el usuario de la aplicación al interaccionar, el que obliga al programa a ejecutar código respondiendo a sus acciones. Los objetos en Gambas avisan con los eventos cada vez que se produce una acción sobre ellos. Es el programador el encargado de escribir en las subrutinas que tratan los eventos el código necesario para responder a ellos.

El entorno de desarrollo proporciona varias formas de conocer los métodos, eventos y propiedades que un objeto puede entender. La primera y más obvia es usando la ayuda, donde se puede encontrar el listado de todas las clases de objetos disponibles, con todas las explicaciones y el listado completo de las propiedades. El editor de código también suministra información, puesto que al escribir el nombre de un objeto y pulsar un punto, aparece un menú contextual con el listado de las propiedades (en color violeta) y métodos (en color verde) que el objeto tiene disponibles.

Para los objetos gráficos hay algunas ayudas más.

El listado de propiedades disponible se puede ver en la ventana de **Propiedades**.

Para los eventos podemos hacer clic con el botón derecho sobre un objeto gráfico en su formulario. Aparecerá un menú donde, entre otras, vemos la opción **Evento**. Podemos elegir el evento que queramos y, automáticamente, el entorno ya escribirá el código correspondiente al principio y al final del procedimiento.

80

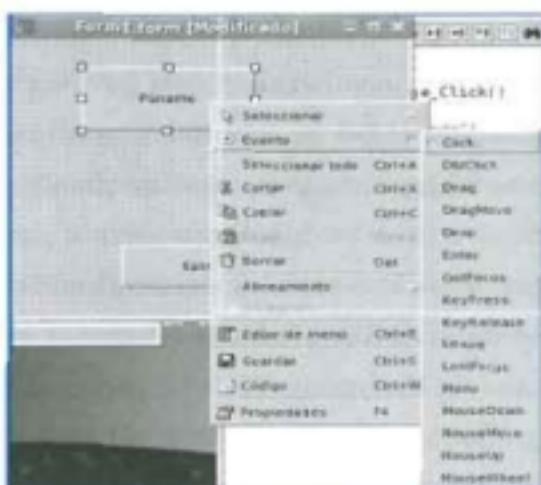


Figura 4. Evento Click.

NOTAS

¹ Los ordenadores sólo entienden de números, no de letras. Para poder usar caracteres se aplican tablas de conversión que asignan un número o código a cada carácter. La tabla de conversión usada normalmente se llama ASCII.

En <http://es.wikipedia.org/wiki/Ascii> se puede ver la tabla completa.

3. La interfaz gráfica

81

3. I Concepto

Si bien Gambas, como cualquier otro lenguaje de programación bien diseñado, puede trabajar perfectamente desligado de toda librería gráfica, creando programas de consola, uno de sus puntos fuertes es la sencillez para crear interfaces gráficas de usuario.

En el mundo GNU/Linux ha existido, a lo largo de la historia, diversas librerías que facilitan la creación de dichas interfaces. Al nivel más bajo, el tradicional sistema X-Window tan sólo proporciona una API para mostrar ventanas, dibujar líneas, copiar mapas de bits y poco más. Sobre dicho sistema, una de las primeras librerías de apoyo que proporcionaba controles completos, tales como botones o etiquetas, fue la Motif™, tradicional de sistemas UNIX™. Su clon libre, Lesstif, probablemente llegó demasiado tarde al escenario para tener un papel relevante.

Las interfaces creadas con estas dos librerías han tenido siempre fama, además, de ser bastante feas o incómodas, especialmente a ojos de los usuarios de Windows™, y no hay que olvidar que de este sistema propietario proviene buena parte de los actuales usuarios de escritorio GNU/Linux.

La compañía TrollTech™, por su parte, creó las librerías QT, para el desarrollo de aplicaciones gráficas con C++. Estas librerías, en principio, se distribuían bajo una licencia, la QPL, no totalmente compatible con el proyecto de la Free Software Foundation. Su inclusión como base del proyecto de escritorio KDE generó un gran revuelo, y el rechazo de un sector de la comunidad hacia ambos proyectos (QT y KDE). En la actualidad, las librerías QT en su edición no comercial, se distribuyen bajo licencia GPL, lo cual implica que los programas desarrollados y compilados con QT como base, han de ser también Software Libre compatible con la GPL. Un programa que no cumpla esta norma, habría de ser compilado sobre la versión comercial de QT, que la compañía antes citada vende y soporta.

82

En parte como rechazo al tandem QT/KDE, y en parte para crear una alternativa al popular escritorio KDE, surgió el proyecto GNOME, que está basado en las librerías GTK+.

GTK+ se desarrolló al principio como una librería gráfica escrita únicamente para el popular programa de dibujo The Gimp (de hecho, GTK significa *Gimp Tool Kit*). Pero más tarde se escindió de este proyecto para convertirse en una librería de propósito general, especialmente diseñada para desarrollos en lenguaje C. Hoy en día, todo el proyecto GTK+ está dividido en varios bloques y niveles: Glib, utilidades de carácter general sin relación con la interfaz gráfica; Gobject para dotar, de cierta orientación a objetos al lenguaje C; Atk, que es un kit de accesibilidad; Pango, para la gestión de fuentes; Gdk, para el dibujo de bajo nivel; y, finalmente, GTK, que proporciona los elementos de la interfaz gráfica habitual. La licencia de GTK+ es LGPL, por lo que ha sido utilizada, además de en muchos proyectos de Software Libre, en programas gráficos privativos que no desean contar con el soporte de la versión comercial de QT porque la han visto como una alternativa más cómoda en sus desarrollos, o han decidido reducir los costes al no tener que pagar por la librería gráfica.

Al margen de estos pesos pesados, hay nombres como FOX, FLTK o WxWidgets, que también resuenan como alternativas para el desarrollo de programas gráficos.

Por tanto, hay muchas alternativas (toolkits) para desarrollar interfaces, y al menos dos de ellas (QT y GTK+) sirven como base para los dos escritorios más comunes (KDE y GNOME), que a su vez aportan un aspecto y funcionalidades diferentes. No obstante, una aplicación KDE puede funcionar en un entorno GNOME y viceversa, a costa, quizás, de perder homogeneidad en el entorno.

Gambas ha decidido ser neutral al respecto. Aporta una interfaz de alto nivel, sencilla para el diseño y programación habituales, que no está ligada a los conceptos de QT, GTK+, ni a ninguna otra librería gráfica subyacente.

No obstante, a la hora de implementar dicha interfaz sí que es necesario emplear alguna librería de sustento por debajo, escrita en C o C++. Por tanto, existen dos componentes gráficos que proporcionan al programador libertad de elección: *gb.qt* y *gb.gtk*. Como se puede suponer por sus nombres, el primero utiliza código compilado con QT y el segundo código compilado con GTK+.

La particularidad de Gambas es que el código escrito para *gb.qt* funcionará exactamente igual si reemplazamos este componente por *gb.gtk* y viceversa. Por lo tanto, el programador en cada momento puede elegir el que más se adapte a sus necesidades por diversos motivos, por ejemplo:

- Integración con KDE, GNOME o XFCE (este último, un escritorio ligero basado en GTK+).
- Aspecto final. Algunos programadores y usuarios se sienten más a gusto con una aplicación QT, otros con GTK+.
- Cuestiones de rendimiento, uso de recursos.
- Licencia, costes en software comercial.

- Necesidades especiales. GTK+ se puede compilar sobre DirectFB, un sistema gráfico alternativo a X-Window, ligero y apto para sistemas empotrados. Tal vez *gb.qt* se pueda compilar en el futuro sobre Qtopia, la librería de TrollTech™ generalmente empleada en PDAs.

□ □ □ □ □ Partiendo de la consola

Vamos a empezar por el camino difícil para comprender cómo la interfaz gráfica no es más que otro accesorio de Gambas, que al igual que el resto de componentes, aporta clases a partir de las cuales crearemos objetos, en este caso *Controles* o *Widgets*.

Crearemos un proyecto de consola, (sí, de consola), llamado Ventana. Añadiremos un módulo *modMain* y una referencia al componente *gb.qt*.

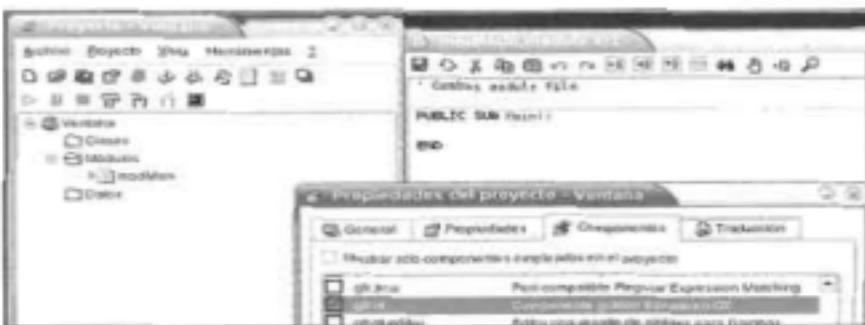


Figura 1. Proyecto Ventana.

Escribiremos ahora el siguiente código:

```
PUBLIC SUB Main()

    DIM hWin AS Window

    hWin = NEW Window
    hWin.Show()

END
```

Al ejecutarlo, aparece una ventana solitaria en la pantalla, que desaparecerá cuando pulsemos el botón Cerrar del gestor de ventanas.

Las ventanas son contenedores de primer nivel. Un *contenedor* es un control que permite situar otros en su interior (botones, cuadros de texto, etc.). Y el adjetivo de *primer nivel* se refiere a que no tiene un *padre*, es decir, que no cuelga de otro control de nuestra aplicación, sino directamente del escritorio.

En este programa podemos ver varios efectos, a primera vista, extraños. El primero es que hemos declarado `hWin` como una variable local, así pues, parece que al finalizar la función `Main` el objeto debería destruirse. Esto no así ya que la ventana, al ser un control, mantiene una referencia interna (los objetos Gambas sólo se destruyen si no existe una referencia en todo el programa). Dicha referencia podemos decir que se corresponde con la ventana que está dibujada en el servidor gráfico, con la intermediación de la librería gráfica (en este caso QT). Por otra parte, el programa debería haber finalizado al terminar la función `Main()` y no ha sido así.

85

El segundo efecto se debe a que tanto el componente `gb.qt` como el componente `gb.gtk` son llamados automáticamente tras el método `Main()` del programa, y queda en el lazo principal de la librería gráfica, esperando a que se produzcan eventos gráficos (por ejemplo, una pulsación del ratón sobre un control).

En la práctica, para crear un programa gráfico indicaremos directamente en el asistente que deseamos crear un proyecto gráfico, de modo que el entorno de desarrollo incluye ya, por defecto, el componente `qb.qt`, y nos permite crear un *formulario de inicio*. Un formulario de inicio es una clase de inicio que hereda las características de la clase `Form`, que no necesita de un método `Main` en el programa porque el intérprete llama directamente al formulario para mostrarlo, y entra en el lazo de gestión de eventos.

□ □ □ □ □ **El entorno de desarrollo**

Para crear un nuevo formulario nos situaremos en la ventana principal del IDE y, tras pulsar el botón derecho, elegiremos Nuevo | Formulario.

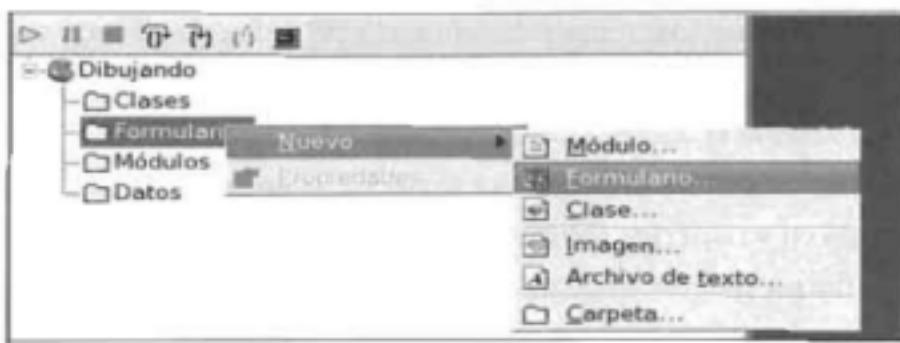


Figura 2. Pasos para crear un nuevo formulario.

Ahora pasaremos a situar los diferentes controles, para lo cual los seleccionaremos de la Caja de herramientas y los posicionaremos sobre los formularios manteniendo el botón izquierdo pulsado, mientras le damos el tamaño deseado. El código se irá escribiendo en función de los eventos que generen los controles, por ejemplo, el evento

Click de los botones o los eventos de ratón y teclado de cada control.

Al pulsar con doble clic sobre un control, se genera, automáticamente, el encabezado del código correspondiente al evento por defecto (el más característico) del control seleccionado.

Además, pulsando con el botón derecho se puede seleccionar la lista de eventos disponibles para un control.



Figura 3. Controles disponibles en la Caja de herramientas.

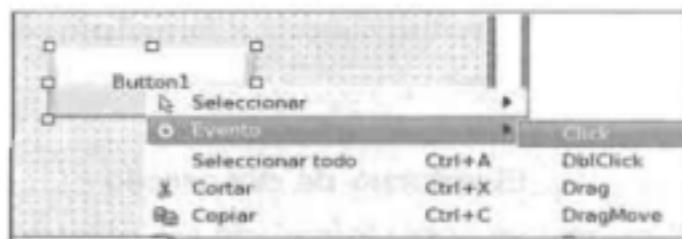


Figura 4. Eventos del control Button1.

3. 2 Manejo básico de los controles

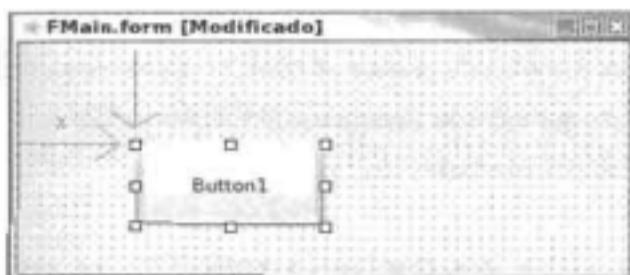
A pesar de que cada control ha sido diseñado para cumplir una función específica, comparten buena parte de su interfaz de programación, de modo que aprender a manejar un nuevo control sea tarea sencilla para un programador que ya ha trabajado con otros controles.

Los controles heredan métodos, propiedades y eventos de la clase *Control*, por lo que todas las características heredadas son aplicables a todos ellos.

En cuanto a los contenedores, proceden de la clase *Container* (que a su vez procede de *Control*) e igualmente tienen muchas características comunes.

□ □ □ □ □ Posición y tamaño

Todos los controles disponen de una serie de propiedades que permiten conocer y modificar su posición y tamaño dentro de su contenedor, o del escritorio en el caso de las ventanas:



■ Figura 5. Propiedades X e Y del control.

escritorio. Los controles disponen de otras dos propiedades, *Left* y *Top*, que son sinónimas de *X* e *Y*, respectivamente. Usar una u otra se deja a la elección del programador.

- Propiedades *X* e *Y*: son de lectura y escritura, y determinan la posición del control, es decir, su punto superior izquierdo. En los controles comunes, la posición indicada es relativa a su contenedor, y en el caso de las ventanas es relativa a la esquina superior izquierda del

- Propiedades *W* y *H*: son de lectura y escritura, y determinan el ancho y alto del control, respectivamente. Dispone de dos propiedades sinónimas. *Width* y *Height* con el mismo significado.

- **Propiedades ScreenX y ScreenY:** son de sólo lectura, y permiten conocer la posición de cualquier control relativa al escritorio, en lugar de a su contenedor *padre*.

Los contenedores disponen, además, de las propiedades *ClientX*, *ClientY*, *ClientWidth* y *ClientHeight*, que determinan, respectivamente, el inicio y la dimensión del área útil para contener controles *hijo*. Por ejemplo, el control *TabStrip*, que dispone de unas pestañas en la parte superior, sitúa a sus *hijos* por debajo de ellas; y *ScrollView*, que puede mostrar barras de Scroll, ve su área útil reducida por dichas barras.

Existen también una serie de métodos para modificar los controles:

- **Método Resize(W,H):** con él podemos cambiar el tamaño de un control modificando su alto y ancho de una sola vez, en lugar de hacerlo en dos pasos modificando las propiedades *W* y *H*, lo que mejora el efecto gráfico de la redimension ante el usuario.
- **Método Move(X,Y):** mueve de una sola vez el control a la posición indicada, en lugar de hacerlo en dos pasos. También dispone de dos parámetros adicionales, *Move(X,Y,W,H)*, con los cuales además de mover el control se puede redimensionar, todo ello en un solo paso, generando una transición más suave ante el usuario, que si modificáramos las propiedades una por una.
- **Métodos MoveRelative y ResizeRelative:** son similares a *Move* y *Resize*, respectivamente, pero en este caso las unidades no son píxeles, sino unidades relativas al tamaño de la fuente por defecto del escritorio. Con esta capacidad, el aspecto del formulario será similar para usuarios que utilicen distintas configuraciones de fuentes (por ejemplo, grandes en un escritorio a 1024x768 o más pequeñas en un escritorio a 800x600).

Las ventanas (controles *Window* y *Form*) disponen, por su parte, de varios eventos relacionados con la posición y tamaño. El evento *Resize* se genera cada vez que el usuario redimensiona una ventana, y *Move* cuando se mueve.

