

4TH EDITION

INTRODUCTION TO PROGRAMMING WITH XOJO



RHINE, LEFEBVRE, PERLMAN

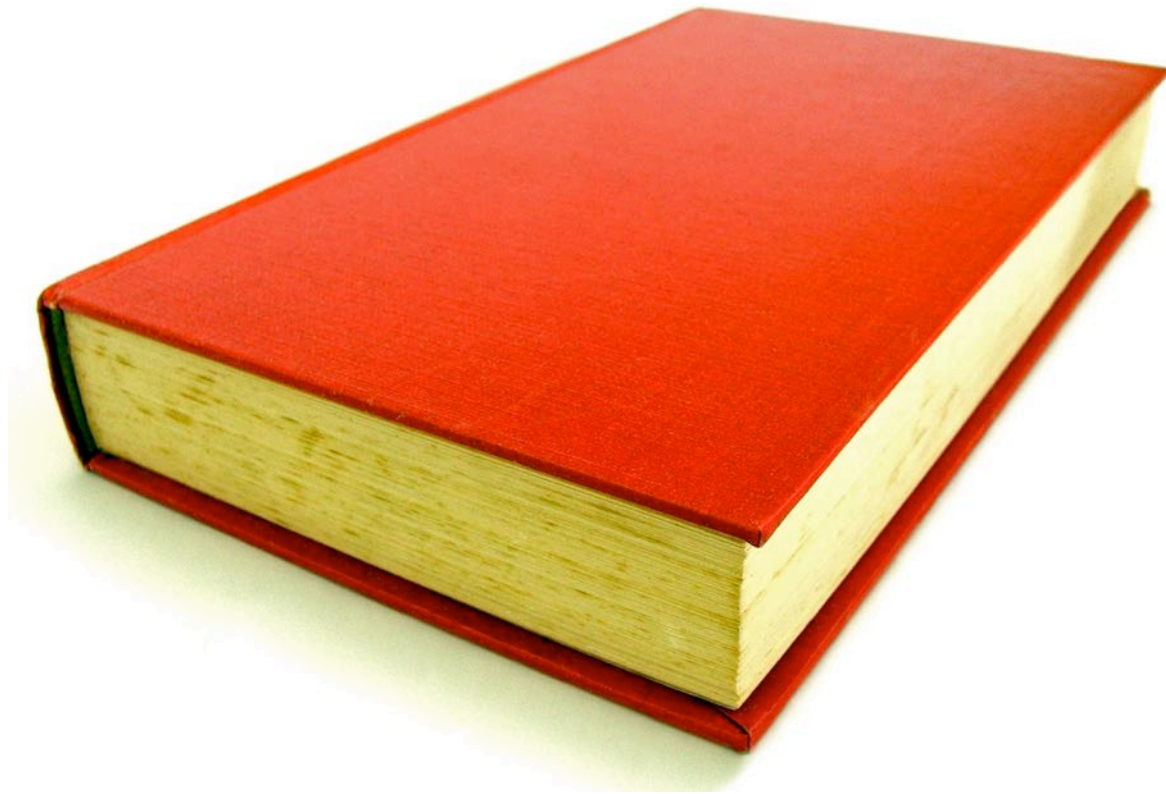
Introducción

CONTENTS

1. Antes de Empezar...
2. Agradecimientos
3. Convenciones
4. Copyright y Licencia



Antes de empezar...



Cuando termines de leer este libro, no serás un desarrollador experto pero tendrás unos fundamentos sólidos sobre los bloques básicos de construcción a la hora de crear tus propias apps. Mi esperanza es que leyendo “Introducción a la programación con Xojo” te motive a aprender más de Xojo o de cualquier otro lenguaje de programación.

El lenguaje de programación más difícil de aprender siempre es el primero. Este libro se centra en Xojo - porque es más fácil de aprender en comparación con muchos otros lenguajes. Una vez que has aprendido un lenguaje, los otros resultan más fáciles de

aprender porque ya has aprendido los conceptos básicos involucrados. Por ejemplo, una vez que sabes como escribir código en Xojo, aprender Java resulta mucho más sencillo, no sólo porque los lenguajes son similares y ya sabes sobre arrays, bucles, variables, clases, depuración, etc. Después de todo, un bucle es un bucle en cualquier lenguaje.

Por tanto, si bien este libro se centra en Xojo, los conceptos presentados son aplicables a muchos otros lenguajes de programación. Cuando es apropiado se reseñan las posibles diferencias en las anotaciones.

Antes de comenzar, necesitarás descargar e instalar Xojo en tu ordenador. Para ello, visita <http://www.xojo.com> y haz clic en el enlace de descarga. Puedes descargar, usar y probar Xojo de forma gratuita; sólo es necesario comprar una licencia de Xojo cuando quieras compilar tus apps.

Agradecimientos

Agradecimiento especial a Brad Rhine, quien escribió las versiones originales de este libro con la ayuda de Geoff Perlman (CEO de Xojo, Inc).

También queremos dar nuestro agradecimiento a la comunidad Xojo. Escuchamos con frecuencia el modo en el que Xojo os ha facilitado el aprendizaje de la programación o la creación de una app o incluso crear vuestro propio negocio Vosotros habéis hecho posible Xojo durante estos más de 20 años.

Convenciones

Dado que Xojo puede funcionar sobre diferentes sistemas operativos y crear apps para diferentes sistemas operativos, algunas de las capturas de pantalla en este libro se han realizado en Windows y otras en macOS. Una de las aplicaciones de ejemplo está basada en web, de modo que verás que sus capturas de pantalla se han realizado en un navegador web.

A medida que avances en el libro también advertirás diferentes formatos de texto.

Uno de los formatos se corresponde con ejemplos de código. Todo lo encontrado en un ejemplo de código ha de ser tecleado en Xojo tal y como aparece en la página. Un ejemplo de código tiene el siguiente aspecto:

```
Var x As Integer
Var y As Integer
Var z As Integer
x = 23
y = 45
z = y * x
MessageDialog.Show(z.ToString)
```

Algunas veces verás el símbolo `↵` en un ejemplo de código. El símbolo `↵` indica un recorte de línea. Esto no significa que debas pulsar la tecla retorno y continúes en una nueva línea; sólo significa que los márgenes del libro no proporcionan la misma

cantidad de espacio que la pantalla del ordenador, de modo que hemos tenido que recortar la línea para que quepa en la página. De modo que no has de teclear el símbolo `↵` sino que sólo has de continuar tecleando lo que aparece en la siguiente línea indentada. De modo que cuando veas código como el siguiente:

```
theMessage = theMessage + myString ↵
           + EndOfLine
```

Deberás de teclearlo todo en una línea como aquí:

```
theMessage = theMessage + myString + EndOfLine
```

Otro formato que verás son los pasos. Un paso tiene el siguiente aspecto:

- 1) **Esto es algo que debes hacer. Puede que se trate de que debas definir “Algún Texto” como la etiqueta de algo. Si esto ocurre, teclea el contenido que se encuentra entrecomillado, pero no las comillas propiamente dichas.**

Esta es una explicación más detallada sobre el paso indicado.

Probablemente indicará más detalles sobre la tarea en la que estás trabajando.

La mayoría de los ejemplos de código en este libro están acompañados por una serie de pasos explicando como funcionan las cosas.

Por último, puede que veas una nota. Una nota tiene el siguiente aspecto:

Esta es una nota. El texto de la nota no es absolutamente esencial, pero puede proporcionar información adicional sobre el asunto tratado.

Copyright y Licencia

Este trabajo está protegido mediante copyright © 2012-2021 por Xojo, Inc.

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. Esto significa que eres libre de compartir el material de este libro siempre y cuando no se obtenga beneficio económico y que se mantiene la atribución a Xojo, Inc.

Si quieres imprimir copias de este libro, tienes permiso para hacerlo siempre y cuando todos los contenidos permanezcan intactos, incluyendo el enlace al PDF disponible de forma pública y gratuita: <http://www.xojo.com/learn/>

¡Hola, Mundo!

CONTENIDOS

1. Información del Capítulo
2. Primeros Pasos
3. Ejecutar y Compilar
4. ¡Hola Mundo!
5. Cazando Bugs



1.1 Información del Capítulo

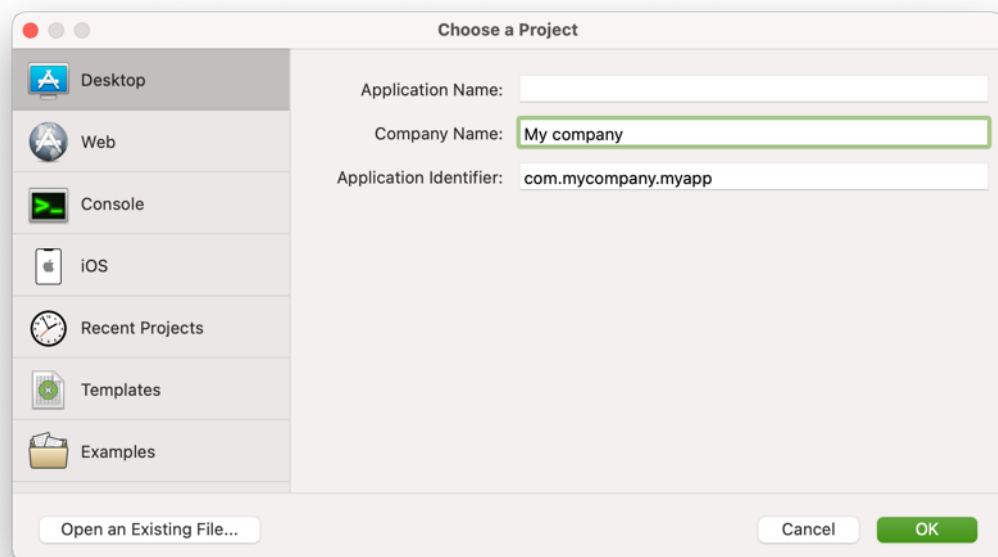
Bienvenido a la Introducción a la Programación con Xojo.

Xojo es un entorno de desarrollo integrado, o IDE, utilizado para diseñar y crear otras aplicaciones de software. Utiliza un lenguaje de programación que también se llama Xojo. En términos muy generales, como programador o desarrollador introduces tu código Xojo en el IDE Xojo, quien se encarga de compilar tu código en una app nativa que puede ejecutarse, independientemente de Xojo, en tu ordenador o en el de algún otro.

Este capítulo presenta el IDE. Aprenderás como navegar por el IDE, como personalizarlo, como organizar tus proyectos y como ejecutar y compilar tus propias aplicaciones. Puede que algunos de los conceptos presentados en este capítulo no tengan mucho sentido por ahora, pero se explicarán en mayor detalle en capítulos posteriores.

1.2 Primeros pasos

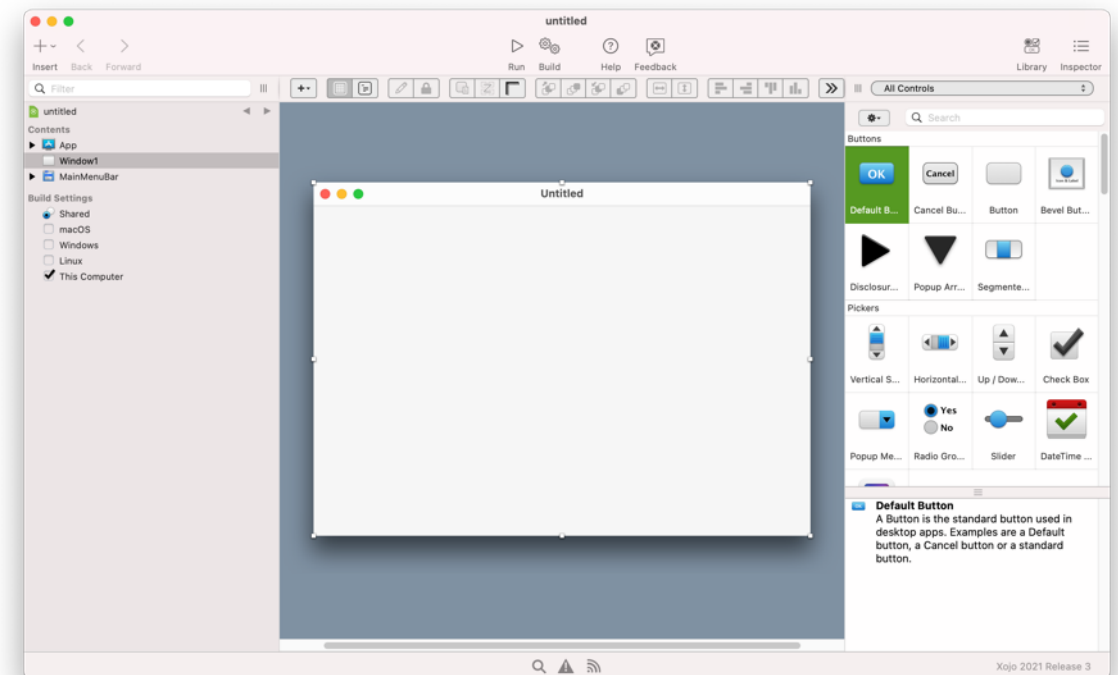
Comienza abriendo la aplicación Xojo. Xojo se ejecutará con una ventana inicial Getting Started mostrando algunos de los recursos disponibles para ti a medida que aprendes Xojo. Cuando cierres dicha ventana, llegarás a la ventana Selector de Proyecto (Project Chooser, en inglés). Esta te pedirá que selecciones una plantilla para un nuevo proyecto. Selecciona “Desktop” y pulsa el botón OK.



Xojo puede crear diferentes tipos de apps, como apps de Escritorio, web, de consola (o de línea de comandos), apps iOS, apps para Raspberry Pi y pronto también para Android.

Una vez que hayas seleccionado una plantilla, Xojo creará un proyecto en blanco basado en dicha plantilla. El proyecto es el

archivo que almacena todo el código fuente, los diseños de interfaz de usuario y la información sobre la app que estás desarrollando. El proyecto vacío por omisión para Desktop tiene el siguiente aspecto:



Por omisión verás una ventana con tres áreas diferenciadas. El área más grande, conocida como el tapete, contiene una ventana vacía. Aquí es donde puedes comenzar a diseñar la interfaz de usuario (UI) para tu aplicación. El panel de la izquierda, conocido como el Navegador, contiene una lista de los elementos utilizados en tu proyecto. El panel de la derecha es la Librería (Library, en inglés), correspondiéndose con una lista de todos los controles que puedes añadir a tu interfaz de usuario (mediante arrastrar y soltar). Por defecto, no verás más que los elementos App,

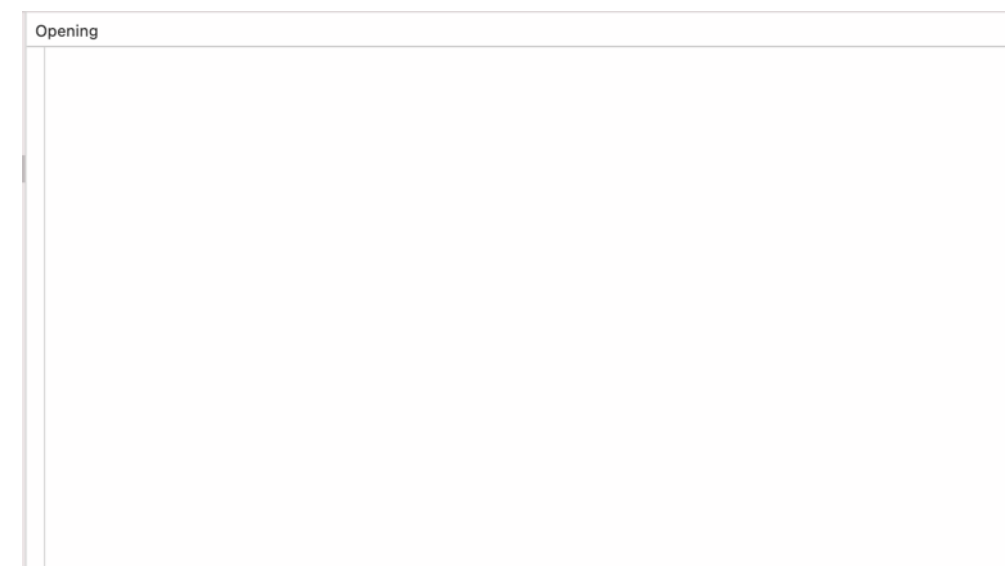
Window1, y MainMenuBar. Bajo Contents verás algunas opciones para los Ajustes de Compilación (Build Settings, en inglés). Los Ajustes de Compilación te permiten cambiar los ajustes de la aplicación que se va a crear, como por ejemplo su nombre, número de versión e icono.

Sobre el tapete, en la parte superior de la ventana, se encuentra la barra de herramientas. Los botones situados más a la derecha conmutarán la visibilidad entre la Librería y el Inspector. Recuerda que la Librería es la lista de controles, y el Inspector te permite modificar las propiedades de aquello que esté seleccionado en el tapete o el Navegador. Por ejemplo, si arrastras y sueltas un botón desde la Librería a tu interfaz, puedes seleccionar dicho botón y utilizar el Inspector para cambiar su texto o su tamaño.

Todo ello compone la vista denominada Vista de Diseño (Layout Editor, en inglés). En la Vista de Diseño puedes componer visualmente el aspecto de tu aplicación. Otra vista importante es la Vista de Código (Code Editor, en inglés). Aquí es donde puedes introducir el código fuente Xojo que controla el comportamiento y funcionalidad de tu app. La mejor forma de comenzar en el Editor de Código es usando el Menu Insert para insertar un Event Handler. Los eventos se tratarán con mayor detalle en un capítulo posterior; por ahora, sólo necesitas saber que los eventos permiten que tu aplicación reaccione a las acciones realizadas por los usuarios finales de tu aplicación, así como a las acciones que puedan causar tanto el ordenador o el sistema operativo

durante el funcionamiento de la aplicación. Selecciona Event Handler en el Menú Insert y elige la opción Opening en el listado de eventos mostrados, haciendo a continuación clic sobre el botón OK

Observa como cambia la interfaz. Desaparece el tapete y se sustituye por el Editor de Código, mientras que el Navegador y el Inspector continúan visibles. El nombre del evento que estás editando estará visible en la parte superior del editor de código.



A medida que añadas componentes a tus proyectos los verás en el Navegador, tanto si estás en el Editor de Diseño o en el Editor de Código. Puedes hacer doble clic sobre un item en la lista para editarlo. También puedes arrastrar estos ítems para situarlos en el orden que desees. El orden en el que coloques estos elementos no incidirán en el rendimiento o funcionalidad de la aplicación creada; depende de ti organizar tu proyecto del modo que tenga más sentido para ti. Incluso puedes añadir carpetas y

subcarpetas si deseas organizar tu proyecto de tal modo. Para añadir una carpeta, dirígete al Menú Insert y selecciona la opción Folder. Luego puedes arrastrar y soltar otros elementos de proyecto sobre la carpeta.

El menú Insert es uno de los que utilizarás con más frecuencia a medida que crees más y más aplicaciones complejas. Además de para las carpetas, también puedes utilizarlo para añadir clases, ventanas y otros componentes a tus proyectos. Una vez más, estos conceptos se explicarán en siguientes capítulos.

1.3 Ejecutar y Compilar

Uno de los botones que aparece en la barra de herramientas es el botón Run. Haciendo clic sobre dicho botón le indicará a Xojo que cree una copia temporal de tu proyecto y que lo ejecute. También puedes ejecutar tu proyecto seleccionado Run en el menú Project. Aunque todavía no has añadido código a tu proyecto, vamos allá y ejecútalo ahora.

Se presentará una ventana vacía. Si bien puede que no resulte muy impresionante, lo cierto es que lo es. Tu proyecto se ha convertido de un archivo de proyecto Xojo a una app que puede ejecutarse en tu ordenador. Además, puede responder a los comandos de menú y atajos de teclado, reaccionando en consecuencia. Por ejemplo, si pulsas Comando + Q en Mac o Alt-F4 en Windows, la app se cerrará y volverás a Xojo.

Ejecutar la app así permite acceder al depurador de Xojo, aspecto tratado en un capítulo posterior. Lo que no crea es una app que puedas compartir con otros. La app creada así es temporal y dirigida a su uso en pruebas y tareas de depuración.

Para crear una app que puedas compartir, has de compilarla. El botón Build está a la derecha del botón Run en la barra de herramientas; o bien puedes elegir Build Application en el menú Project.

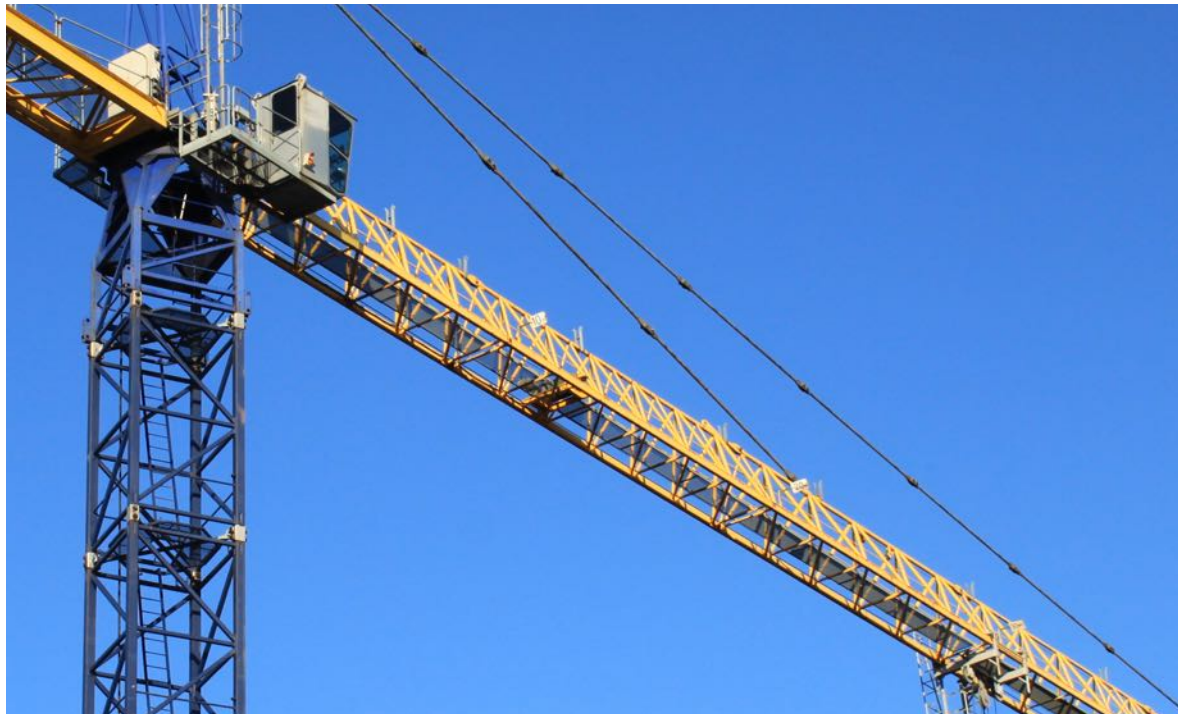
Recuerda, puedes ejecutar tus proyectos con la versión gratuita de Xojo, pero has de comprar una licencia Xojo para Compilar tus apps.

Si compilas tu proyecto ahora tendrás una app que puede ejecutarse, pero tendrá un nombre e icono genéricos, y tampoco hará nada interesante en este punto.



Para cambiar esto, selecciona el ítem App en el Navegador y selecciónalo. Este elemento, denominado clase, te permite definir propiedades que se aplicarán a la app como un todo. Con la clase App seleccionada, verás como el panel derecho cambia para mostrar el Inspector. Si no es así, haz clic en el botón Inspector en la barra de herramientas.

El Inspector está agrupado en varias secciones de propiedades. No cubriremos todas estas secciones ahora. Observa que hay un grupo separado de ajustes de compilación para cada plataforma para la que quieras compilar: Windows, Linux y macOS. Ubica el ajuste correspondiente a tu plataforma y cambia la propiedad App Name.



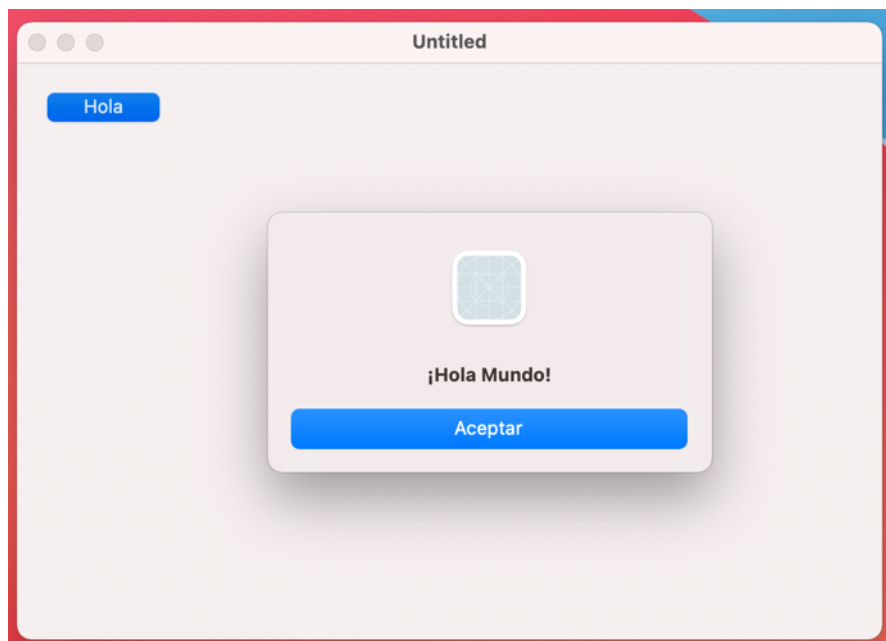
Haz clic en Shared Settings en el área de Build Settings en el Navegador. Verás más grupos de ajustes, incluido uno denominado Version. Para una aplicación vacía como esta no es importante, pero cuando compiles apps más complejas y las actualices necesitarás mantener estos ajustes actualizados. En general, deberías incrementar MajorVersion cuando hayas añadido nuevas características o funcionalidad principales a tu aplicación. Debería aumentarse MinorVersion cuando se añadan

pequeñas características, e incrementar BugVersion cuando se compilen nuevas versiones de la aplicación en las que se corrijan fallos. Si se introduce 3 para MajorVersion, 1 para MinorVersion y 4 para BugVersion, la aplicación se presentará al sistema operativo como la versión 3.1.4.

Bajo BugVersion se encuentra StageCode ofreciendo cuatro opciones: Development, Alpha, Beta y Final. A medida que trabajes añadiendo código a tu proyecto, el StageCode debería configurarse a Development. Una vez que tu aplicación contenga todas las características, deberías cambiar StageCode a Alpha. StageCode debería cambiarse a Beta una vez que tu aplicación esté prácticamente completa y también se haya realizado la mayor parte del testeo interno. Antes de que publiques una compilación de tu aplicación y la ofrezcas al público general, deberías configurar StageCode a Final.

1.4 ¡Hola, Mundo!

Si bien la programación de ordenadores está lejos de ser una ciencia antigua, también tiene sus tradiciones. Una de estas tradiciones es la app Hola Mundo (Hello World, en inglés). Cuando un desarrollador está aprendiendo un nuevo lenguaje, es tradicional comenzar con una app muy simple que anuncia su existencia declarando “¡Hola Mundo!”. No romperemos dicha tradición, de modo que esta sección te ayudará a crear tu propia app Hello World. Esta es la captura de pantalla de la aplicación final.



En el Navegador, encuentra el ítem llamado Window1. Window1 representa la vista por defecto que muestra tu app al usuario final (esto puede cambiarse, tal y como veremos en posteriores

capítulos). Haz clic en Window1 para abrirla en el Layout Editor. Esta es la vista que utilizarás para diseñar la interfaz de usuario para tus proyectos.

Tal y como se ha indicado anteriormente, hay tres áreas principales que verás en el Layout Editor. El área de mayor tamaño, ubicada en el centro, se denomina el tapete. En el tapete verás una ventana vacía. A la izquierda de la ventana está el Navegador conteniendo los ítems de tu proyecto. A la derecha puede estar tanto la Librería como el Inspector. La Librería contiene elementos de interfaz de usuario que puedes añadir a la ventana (o bien situar en cualquier parte del tapete).

1) Encuentra el Botón por Defecto en el listado de controles y arrástralo sobre la ventana en el tapete.

En primer lugar, pon tu atención en la Librería. A medida que recorres el listado verás varios elementos de interfaz de usuario, algunos de los cuales te resultarán familiares, como por ejemplo el CheckBox, varios Botones y la ScrollBar. La utilidad de otros controles, como por ejemplo el Canvas, el PagePanel y el Timer pueden no resultar muy evidentes a primera vista. Estos controles y otros se cubrirán en el Capítulo Seis. Para la aplicación Hello World, el único control requerido es el Botón por Defecto. Pasa sobre los controles y verás más detalles mostrados en la sección inferior de la Librería. Encuentra el Botón por Defecto en el listado de controles y arrástralo sobre la ventana en el Layout Editor. Arrastrar controles sobre la ventana es una acción que realizarás con frecuencia a medida que creas tu app. Una vez que hayas soltado el control sobre la ventana, puedes arrastrarlo para cambiar su posición y también puedes utilizar las propiedades del Inspector para modificarlo.

2) Con el Botón por Defecto recién añadido y seleccionado, haz clic en el Botón del Inspector en la barra de herramientas.

Aquí cambiarás tan sólo unas pocas propiedades para tu aplicación Hello World, y en realidad son todas opcionales. Sin embargo, querrás empezar incluso ahora a desarrollar hábitos que te llevarán a una mayor productividad a lo largo del camino. Con esto en mente, la primera propiedad que editarás es la primera propiedad en la lista: Name.

3) Cambia el nombre de tu PushButton a “HelloButton”.

El nombre de un control es el modo en el que te refieres a este desde el código. El nombre de un control no puede contener espacios o signos de puntuación, más allá del guión bajo. Es muy recomendable dar a cada control un nombre del que puedas recordar su propósito. En este caso, el nombre HelloButton te recordará que este botón dirá “Hola”. El lenguaje Xojo no diferencia entre mayúsculas y minúsculas, de modo que, si lo prefieres, puedes indicar el nombre del control usando todos los caracteres en minúsculas; Xojo no pondrá objeciones.

4) Cambia la etiqueta de HelloButton a “Hola” en el Inspector.

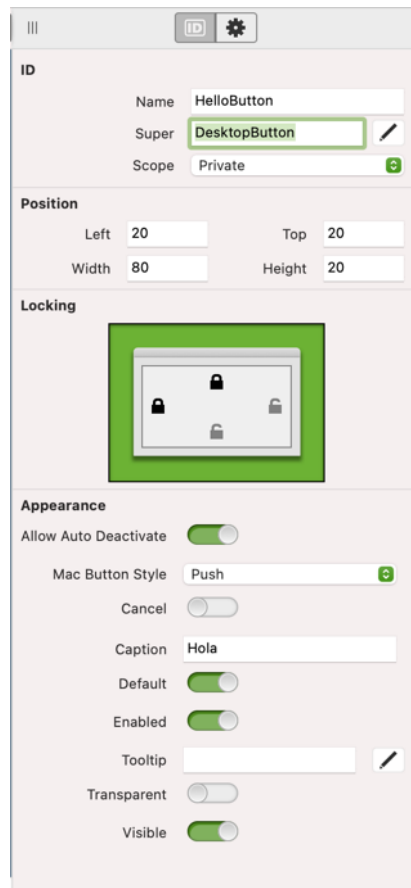
La segunda propiedad que modificarás es la etiqueta del PushButton. Si bien el nombre del control es el modo en el que tu, como desarrollador, te referirás al control, la etiqueta es el texto que verá el usuario de tu aplicación. En el caso de un Botón, la propiedad Caption será el texto visible en el botón propiamente dicho. Sitúa el apuntador sobre el Botón para encontrar y seleccionar el icono Caption. Cambia la etiqueta del botón para que sea “Hola” en la sección Appearance del Inspector. La etiqueta puede incluir espacios y signos de puntuación.

5) Cambia la posición del Botón en la ventana modificando sus propiedades Left y Top en la sección Position del Inspector.

Si prefieres modificar la posición del botón visualmente, puedes arrastrarlo manualmente a la posición y utilizar las líneas de guía que aparecen para ajustar su posición. No te preocupes mucho sobre la posición específica del botón; utiliza tu propio juicio para diseñar la ventana. Tu ventana puede parecerse a esta:



Y esta es una vista parcial (más aproximada) del Inspector:



6) Ejecuta tu proyecto.

Deberá aparecer tu ventana, completa con tu botón “Hola”. Sin embargo, al hacer clic sobre el botón no mostrará el mensaje “¡Hola Mundo!”. Para ello, has de añadir algo de código al Botón.

7) Sal de la app y vuelve a Xojo.

Asegúrate de guardar los proyectos creados a lo largo del libro. Volverás a su desarrollo a medida que avances en tu aprendizaje.

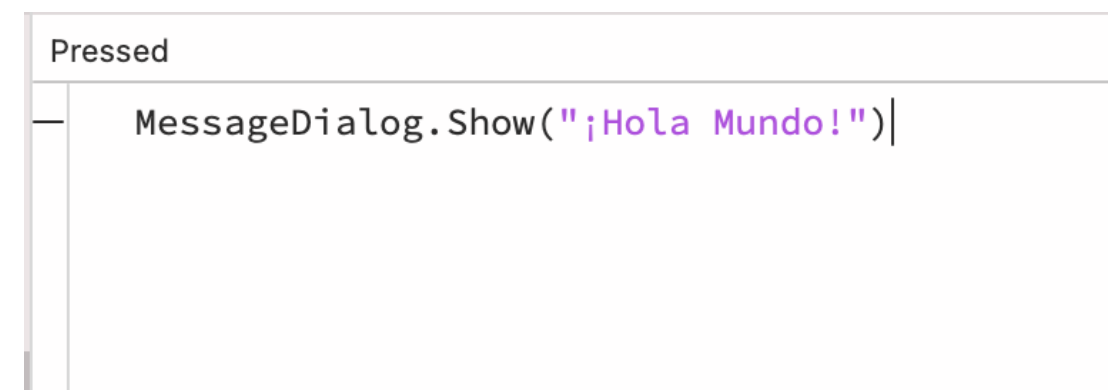
Cuando veas de nuevo el Layout Editor, haz doble clic en HelloButton. Xojo presentará una lista de eventos sobre los que puedes añadir código.

Selecciona el evento Action (el código del evento Action se ejecuta cada vez que se pulsa en el Botón). También verás otros eventos en el listado, como por ejemplo KeyDown, FocusLost, Opening, y otros, pero por ahora sólo necesitarás añadir el evento Pressed (los eventos se explican con mayor detalle cuando se traten los controles en el Capítulo Seis).

8) Con el evento Pressed seleccionado, haz clic en OK para mostrar el Editor de Código. Este es el código que usarás:

```
MessageBox.Show("¡Hola Mundo!")
```

Tu Editor de Código debería tener el siguiente aspecto:



A medida que escribas código en el Editor de Código observarás que Xojo te ofrecerá el autocompletado a medida que escribes. El autocompletado de Xojo es una excelente forma de aprender más sobre el lenguaje, dado que puedes empezar escribiendo unas pocas letras para ver cuáles son las sugerencias mostradas. Además, también se mostrará ayuda en la parte inferior de la ventana, ofreciendo información sobre el método o función sobre la que esté situada el apuntador del ratón. Este es otro modo excelente de aprender más sobre el lenguaje Xojo.

En cuanto a lo que has escrito, has introducido dos cosas: un método y un parámetro. Estos términos se explican con mayor detalle en el Capítulo

Tres; pero, por ahora, sólo has de saber que un método en Xojo es simplemente el modo de decirle al ordenador que haga algo. Un parámetro asociado con dicho método proporciona al ordenador detalles adicionales sobre la forma en la que quieres hacerlo. En esencia, el método es lo que quieres hacer, y los parámetros son cómo hacerlo.

El método que estás ejecutando se llama Show y pertenece a la clase MessageDialog. Toma un fragmento de texto y lo muestra al usuario final de tu app. El fragmento de texto en cuestión para este ejemplo es “¡Hola Mundo!”.

9) Ejecuta de nuevo tu aplicación.

Una vez más, debería aparecer tu ventana, completa con tu botón “Hola”. En esta ocasión, cuando se hace clic en el botón sí muestra un mensaje “¡Hola Mundo!”. ¡Enhorabuena! Has creado y ejecutado tu primera app Xojo.

10) Sal de la aplicación.

Si estás familiarizado con otros lenguajes de programación, puedes observar algunas diferencias entre estos y Xojo. En primer lugar, muchos lenguajes requieren del uso de punto y coma al final de cada línea, en vez de simplemente utilizar un espacio o retorno de línea para indicar el final de línea. el código Xojo nunca debe finalizar con un punto y coma - de hecho Xojo utilizará el área de Ayuda de Sintáxis en la parte inferior de la ventana para advertirte de ello si escribes un punto y coma como parte de tu hábito.

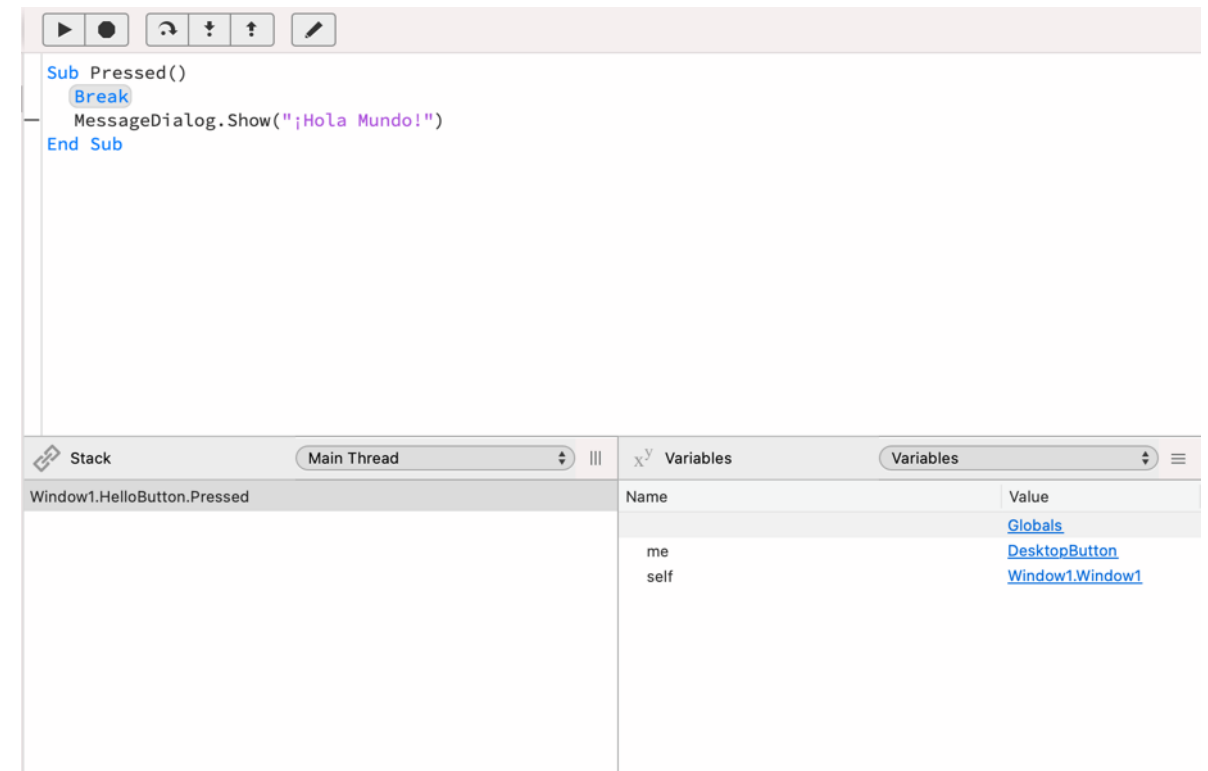
1.5 Cazando Bugs

Ahora que está funcionando correctamente tu primera aplicación, esta sección te proporcionará una breve introducción al depurador de Xojo. Observa que si bien el depurador forma una parte sustancial de Xojo, no se dedica un capítulo específico sobre el mismo. En vez de ello, y a medida que se introducen otros conceptos, irás aprendiendo sobre el depurador de forma gradual. Por ahora, verás dos formas de acceder al depurador: una de forma intencionada y otra de forma accidental.

En primer lugar, la forma intencionada: vuelve al Editor de Código, ubica la posición en la que has introducido la línea `MessageBox.Show` (si tu aplicación Hello World aun está funcionando, tendrás que salir de ella). Cambia el código en el evento Action de HelloButton para que sea así:

```
Break
MessageBox.Show("Hello, World!")
```

La palabra clave `Break` causa que tu app se pause, pero que no se detenga, y muestra el depurador de Xojo. Ejecuta tu proyecto ya con la palabra clave `Break` incluida. Cuando hagas clic en HelloButton, verás una pantalla similar a la mostrada a continuación:



Este es el depurador de Xojo. Con tu actual proyecto no hay mucho que ver, pero a medida que tus proyectos sean más complejos, el depurador te proporcionará una buena cantidad de información sobre tu aplicación a medida que se está ejecutando. Advierte que en la anterior captura de pantalla está seleccionada la línea que se está ejecutando (la palabra clave `Break`).

El panel en la parte inferior derecha de la ventana proporciona una vista jerárquica con las variables y propiedades de tu app.

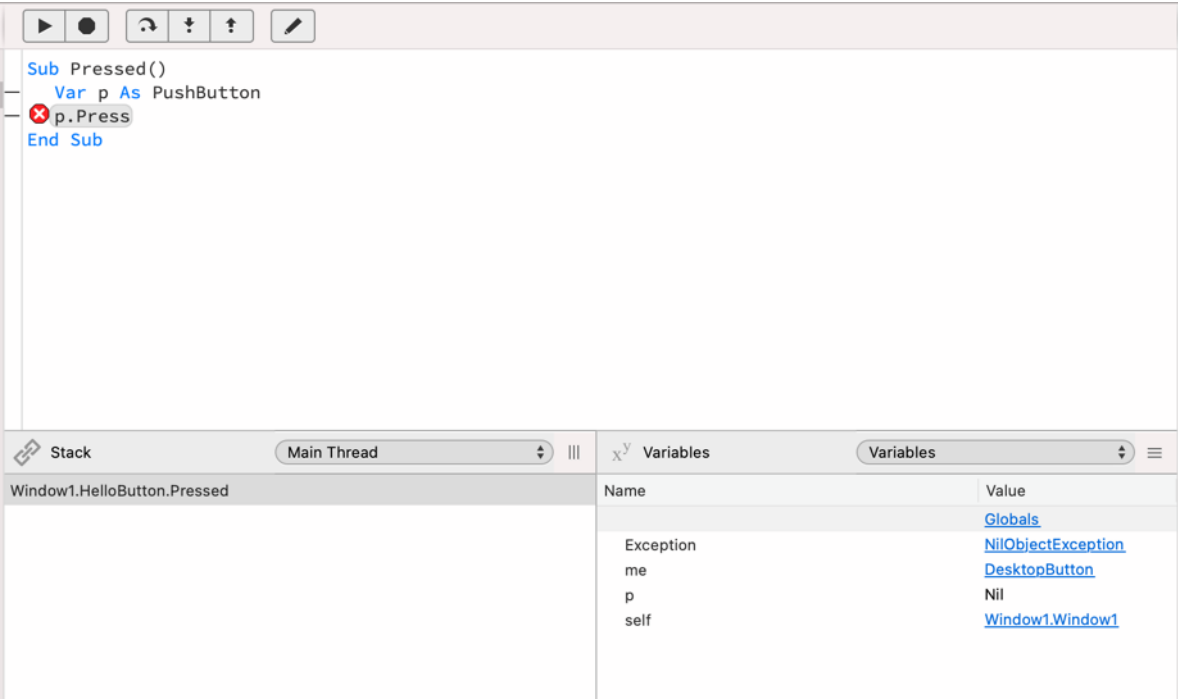
Pulsa el botón `Stop` en la Barra del Editor para detener la ejecución de tu app y volver a la edición de tu proyecto.

Ahora que has visto como acceder al depurador intencionadamente, veamos como hacerlo de forma accidental.

Este ejemplo es forzado, pero debería mostrarte un aspecto importante en el uso del depurador. Cambia el código en el evento Pressed de HelloButton con las siguientes dos líneas:

```
Var p As PushButton
p.Press
```

Por ahora, no tienes que preocuparte ni siquiera sobre lo que el anterior código intenta hacer. Ejecuta simplemente de nuevo tu proyecto y pulsa HelloButton. Debería de aparecer nuevamente el depurador, pero con un aspecto ligeramente distinto:



El icon con la “X” de color rojo indica que ha ocurrido un error. Un vistazo al panel de variables muestra que P, el PushButton, es Nil. El significado de esto será más claro en capítulos posteriores,

pero en esencia has intentado acceder a algo que sencillamente aun no existe.

En términos de programación, un objeto no existente se denomina Nil. Cuando Xojo encuentra un objeto Nil, o uno de los múltiples tipos de excepciones, el depurador se mostrará por omisión.

¿Bugs? Es posible que te preguntes por qué los errores de programación se llaman bugs. Es muy conocido que en los primeros años de la informática, los ordenadores como el Mark II utilizaban tubos de vacío que generaban calor y luz, y que atraían a bichos que podían interferir con el funcionamiento del ordenador. Pero las anotaciones de Thomas Edison, en la época de los 1870, mucho antes del Mark II, ya mostraban la palabra bug para referirse a los fallos de las máquinas.

En términos generales, una excepción tiene lugar cuando ocurre algo en una aplicación en ejecución y que el código no se espera. Esto puede ser un número fuera del rango esperado o, como en el anterior ejemplo, un intento de acceder a un objeto que no existe. La escritura de código defensivo para prevenir excepciones forma una gran parte en el desarrollo de software.

Preséntate a ti mismo

CONTENIDOS

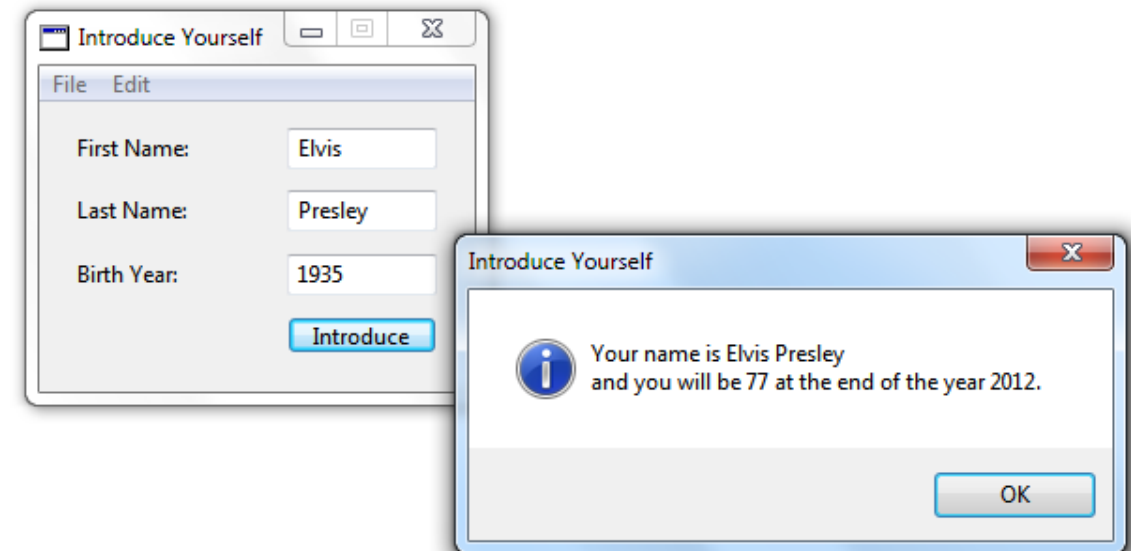
1. **Introducción al Capítulo**
2. **Un Lugar para tus Cosas**
3. **Tipos de Datos Comunes**
4. **Variables en la Práctica**
5. **Trabajando con las Variables**
6. **Indentificación de Bugs avanzada**

2.1 Introducción del Capítulo

Este capítulo introducirá el concepto de las variables. Una variable es un nombre dado a una posición en la memoria del ordenador que contiene un valor. Desde el punto del desarrollador, una variable tiene tres propiedades: un nombre, un valor y el tipo de dato que contiene.

Por ejemplo, puedes tener una variable con el nombre `MyAge`. Su tipo puede ser un número entero (integer, en inglés), y su valor puede ser 16.

El tipo de dato de una variable puede ser prácticamente cualquier cosa. Este capítulo se centrará en los tipos de datos más comunes y cómo utilizarlos en tu código. También indicará algunas buenas prácticas a la hora de nombrar tus variables, así como a la hora de asignarles valores. Por último, crearás una pequeña aplicación llamada “Introduce Yourself”. Esta aplicación realizará unas pocas preguntas al usuario final y proporcionará como resultado alguna información al usuario.



2.2 Un Lugar para tus cosas

Una variable es un modo en el que puedes referirte a una posición en la memoria del ordenador que contiene un valor al que puedes acceder, ya sea para leerlo o cambiarlo. Para referirte a esta posición de memoria, comenzarás eligiendo un nombre para tu variable. Elegir un buen nombre para la variable en los inicios de tu experiencia como programador te será de gran ayuda a lo largo del camino. Asignar un nombre a las variables puede resultar difícil, pero si mantienes unas cuantas normas en mente podrás llegar a tus propias convenciones.

Recuerda que estas son unas guías simples. Cada persona necesita desarrollar sus propias convenciones a la hora de crear nombres de variables y que mejor encaje con su estilo de programación.

En primer lugar, elige nombres de variables que sean específicas y descriptivas. Por ejemplo, el proyecto de ejemplo que crearás en este capítulo tendrá una variable que necesita almacenar el nombre y apellido del usuario, de modo que la variable se llamará `fullName`. Si necesitas diferenciar entre distintos tipos de nombres, entonces puede que tengas variables denominadas `employeeFullName` y `supervisorFullName`. No te preocupes de utilizar nombres largos para tus variables, dado que el

autocompletado de Xojo puede ser de ayuda cuando tengas que teclearlos posteriormente. Evita el uso de nombres genéricos. Cuando trabajas con una fecha tope puedes sentirte tentado de utilizar sólo una letra como nombre de la variable. Si bien esto puede ser apropiado para ciertas funciones de conteo y bucles (algo que veremos en posteriores capítulos), en general resultan más prácticos los nombres concretos. Un nombre de variable debería indicar su propósito a simple vista.

En segundo lugar, cuando se trata del uso de mayúsculas y minúsculas, no hay un único modo correcto. Algunos desarrolladores prefieren el estilo `camelCase` (también conocido como mayúsculas intermedias), algunos prefieren `nombre_con_guiones_bajos`, y algunos utilizan `cajabajasinpuntuacion`. Con independencia de la opción que elijas, sé lógico y, lo más importante, sé consistente. En este libro usaremos nombres de variables con estilo `camelCase`.

En tercer lugar, algunos desarrolladores prefieren utilizar el nombre de variable para que indique su tipo, aunque está lejos de ser algo universal. Algunos ejemplos podrían ser `nameString`, `birthDate` y `favoriteColor`. No es parte de la norma que seguiremos en este libro, pero siéntete libre de seguirlo en tu propio código y proyectos.

Por último, observa que a medida que amplías tu conocimiento de Xojo, podrás aplicar estas mismas reglas y convenciones a

otros aspectos de tu código, como es el nombre de las funciones y también de tus propias clases.

Ahora que tienes alguna idea sobre cómo nombrar variables, es el momento de echar un vistazo sobre cómo indicar a Xojo estos nombres. Para ello utilizarás la palabra clave `Var`.

Para utilizar uno de los ejemplos vistos, puedes tener una variable llamada `fullName` que utilizarás para almacenar el nombre de alguien. Su tipo de dato será `String` (el cual se explicará en la Sección 2.3). Para crear esta variable, introduce esta línea en Xojo:

```
Var fullName As String
```

Esta simple línea hace mucho. Tal y como hemos explicado anteriormente, reserva espacio en la memoria del ordenador para la información que quieras almacenar. También utiliza un nombre específico y fácil de recordar a la hora de usarlo. Por último, indica a Xojo qué tipo de datos se guardarán en dicha variable (en el ejemplo anterior, una `String`, o datos de texto).

“Var”, como ya habrás imaginado, es una abreviatura de “Variable.”

Indicar a Xojo el tipo de dato que estarás usando es algo crítico. Xojo es un lenguaje de programación de *tipado fuerte*. Esto significa que cada variable tiene un tipo conocido, y que se

espera que tu, como desarrollador, trates dicha variable con su tipo declarado. Por ejemplo, puedes realizar operaciones matemáticas sobre datos numéricos, pero no sobre el texto. Xojo te ofrecerá sugerencias y advertencias si intentas usar una variable que no está soportada por su tipo de dato.

Es preciso instanciar la mayor parte de los tipos de datos. Mientras que al usar la palabra clave `Var` se reserva espacio para una variable, la palabra clave `New` *instancia* crea una nueva instancia de tu variable. Considera las siguientes líneas:

```
Var holiday As DateTime
```

Con esta línea has reservado espacio en la memoria para tu variable, le has dado un nombre (`holiday`) y has dicho a Xojo con qué tipo de dato trabajará (un valor de fecha y hora). Lo que no has hecho, sin embargo, es crear el objeto `DateTime` propiamente dicho. De modo que si intentas acceder a una de las propiedades de `holiday`, como pueda ser la propiedad `Year`, antes de continuar, te encontrarás con una Excepción de tipo `NilObjectException`, como la mostrada al final del Capítulo Uno.

Para instanciar tu variable, utiliza la palabra clave `New`:

```
holiday = New DateTime(2020, 5, 25)
```

Una vez que se ha instanciado tu variable, puedes acceder con certeza o modificar sus datos y propiedades. Este es un ejemplo

más completo en la creación de una variable y el acceso a sus propiedades:

```
Var holiday As DateTime  
holiday = New DateTime(2020, 5, 25)  
MessageBox.Show(holiday.Year.ToString)
```

En este ejemplo breve has creado una variable, la has instanciado y has ofrecido al usuario final un diálogo de mensaje conteniendo la fecha actual en un formato legible.

2.3 Tipos de Datos Comunes

Habrás advertido “tipos de datos” mencionado con frecuencia en las anteriores páginas; pero, ¿qué es un tipo de dato? De forma simple, un tipo de dato es una forma de información que se comporta en un modo determinado y que tiene ciertas características. En esta sección aprenderás sobre los tipos de datos más comunes. La mayoría de los lenguajes de programación utilizan el mismo conjunto básico de tipos de datos.

STRING (CADENAS DE TEXTO)

Uno de los tipos de datos más comunes que encontrás es el String. Un string es un fragmento de texto. Puede ser de cualquier longitud (con el tamaño máximo de una string limitado únicamente por la memoria del ordenador), y puede contener cualquier dato que pueda ser representado mediante letras (de cualquier idioma), números y signos de puntuación. Para crear una cadena en Xojo, utiliza simplemente la palabra clave Var:

```
Var myName As String
```

El tipo de datos String se encuentra entre uno de los pocos tipos de datos que no necesitan ser instanciados. Tan pronto como se

ejecute la línea de código que contiene la palabra clave Var, se crea la String en memoria, si bien estará vacía.



Siempre que introduzcas datos de cadena de texto (string) en Xojo, estos deben estar contenidos entre comillas dobles:

```
myName = "Elvis Presley"
```

Si tienes una cadena que ya contiene comillas dobles, debes “escapar” las comillas duplicándolas:

```
myName = "Elvis ""The King"" Presley"
```

Puede que una cadena con dobles dobles comillas tenga un aspecto raro, pero las dobles comillas adicionales indican a Xojo que es parte de los datos de cadena y no indican el final de la misma.

Las cadenas pueden resultar a veces una fuente de confusión para los desarrolladores que empiezan. Dado que pueden contener datos textuales, sus datos pueden parecer en ocasiones como datos de otro tipo. Considera lo siguiente:

```
Var myAge As String  
myAge = "18"
```

Si observas el valor de la cadena myAge, nos parece como humanos que se trata de un dato numérico: 18. Sin embargo, para el ordenador (y el compilador Xojo), no es así. Es simplemente una serie de bytes que representan un 1 seguido de un 8. Antes de usar estos datos, que parecen numéricos, como parte de una función matemática, necesitarás realizar un paso intermedio. Utiliza la función ToInteger para recuperar su valor como un número que el ordenador pueda reconocer como tal:

```
Var myAge As String  
Var myNumericAge As Integer  
myAge = "18"  
myNumericAge = myAge.ToInteger
```

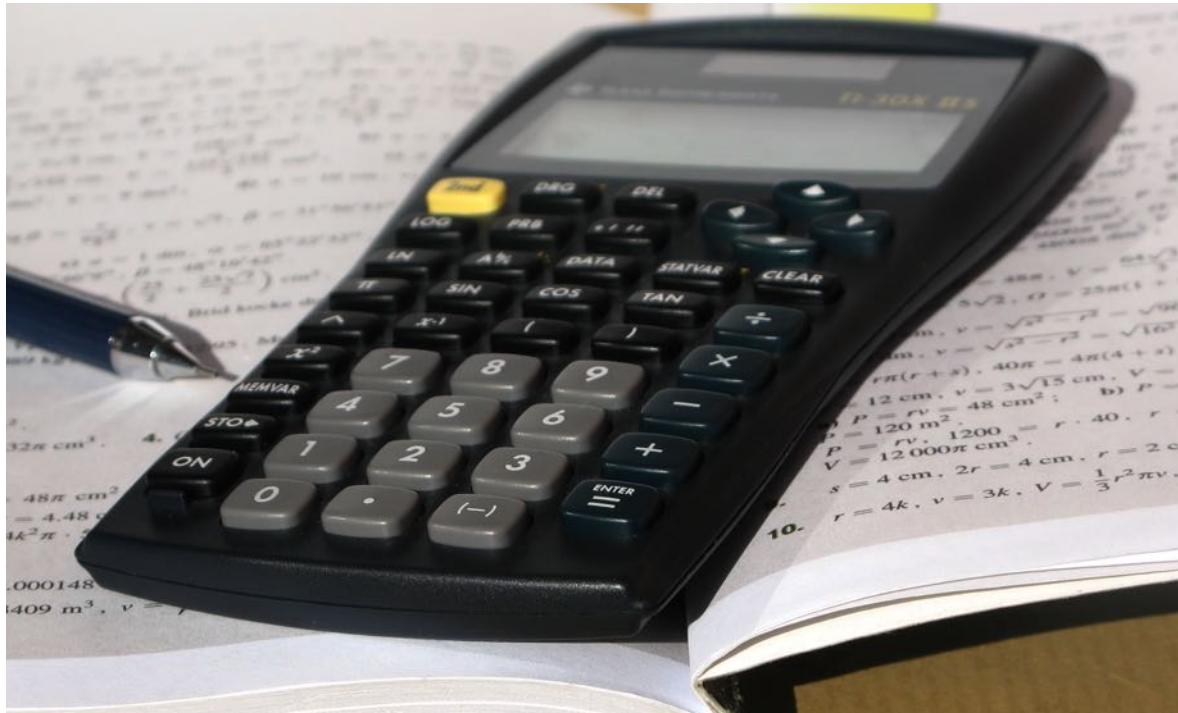
ToInteger es un método perteneciente a las cadenas y que convierte el valor de cadena a un entero y devuelve dicho valor. ¿Qué es un Integer? Un Integer es un número entero.

NÚMEROS ENTEROS (INTEGER) Y REALES (DOUBLE)

Un Integer es un número entero, sin parte fraccional; de modo que no contiene precisión más allá de la coma decimal. Generalmente un entero puede ser positivo o negativo.

Los valores máximo y mínimo de un entero dependen del ordenador. En un sistema de 32 bits, un entero puede ser tan bajo como -2.147.483.648 y tan alto como 2.147.483.647.

Esto es, un rango de 4.294.967.295 posibles valores. La mayoría de los ordenadores modernos son de 64 bits. Un ordenador de 64 bits puede contener un entero tan bajo como -2^{63} y tan alto como uno por debajo de 2^{63} .



Hay varias formas de declarar una variable de entero. La primera y más simple es:

```
Var myNumber As Integer
```

Si estás en un ordenador de 32 bits, esto te dará un entero de 32 bits; y si estás en un ordenador de 64 bits, esto te dará un entero de 64 bits.

Si quieres ser más específico sobre el tipo de entero que necesitas, puedes hacerlo:

```
Var myHugeNumber As Int32  
Var myReallyHugeNumber As Int64
```

Las dos líneas anteriores te darán un entero de 32 bits y uno de 64 bits. Salvo que tengas una necesidad concreta de hacerlo de otro modo, el modo más sencillo es declarar tus variables es como Integer.

Tal y como se ha indicado anteriormente, puede que un entero no sea tan preciso como necesitas. Dado que los enteros no cuentan con parte fraccional, no puedes usarlos para almacenar números más precisos, como pueda ser 3,14. Si intentas almacenar 3,14 como entero, resultará en un valor de 3.

Para almacenar un número más preciso, puedes utilizar el tipo de dato Double. Un Double es un número real de coma flotante; también conocido como número de precisión doble. Un Double puede tener parte decimal y puede contener cualquier cantidad de dígitos significativos más allá de la coma decimal.

El rango de un número de precisión doble es realmente amplio. Su rango es tan amplio que probablemente nunca tendrás que preocuparte de excederlo.

Dado que tanto los enteros como los dobles son tipos de datos numéricos, podrás utilizarlos en operaciones matemáticas sin problemas:

```
Var oneNumber As Integer  
Var anotherNumber As Integer  
Var theResult As Integer  
oneNumber = 5
```



```
anotherNumber = 10
theResult = oneNumber + anotherNumber
```

En el anterior código, theResult contiene el valor 15.

Por supuesto, puede hacerse lo mismo también con doubles:

```
Var oneNumber As Double
Var anotherNumber As Double
Var theResult As Double
oneNumber = 5.3
anotherNumber = 10.2
theResult = oneNumber + anotherNumber
```

En este código, theResult contendrá el valor 15,5.

En ambos ejemplos has tecleado de más. Observa como las tres variables se han declarado en tres líneas separadas. Esto es perfectamente aceptable, pero hay una forma abreviada de hacerlo:

```
Var oneNumber, anotherNumber, theResult As Double
```

El sistema que utilices dependerá de tus propias preferencias y de cómo resulte más legible para ti.

Los enteros y los números de coma flotante no necesitan ser instanciados, tal y como también ocurre con las String. Cuando los declaras, estos pasan a existir inmediatamente con un valor de cero.

BOOLEANOS

Un booleano es un tipo de dato usado para almacenar valores de *la verdad*. Su valor puede ser True (cierto) o False (falso):

```
Var thisIsAwesome As Boolean
```

Tal y como ocurre con las String, enteros y los números de coma flotante, no es necesario instanciar los booleanos. Una vez declarado, un booleano tendrá el valor falso (False) excepto que se indique lo contrario.

Así, en el anterior código tienes un booleano llamado thisIsAwesome cuyo valor es False.

Como la programación es genial, cambiemos el valor:

```
Var thisIsAwesome As Boolean
thisIsAwesome = True
```

Si posteriormente necesitas cambiar un booleano nuevamente a falso (False), puedes hacerlo así:

```
thisIsAwesome = False
```



FECHAS Y HORAS

Las fechas y las horas pueden ser un tipo de dato quisquilloso a la hora de utilizarlo en Xojo. Un objeto DateTime puede almacenar todos los detalles relevantes con una fecha y hora particulares. Para crear un objeto DateTime, utiliza la palabra clave Var e instancia la variable (a diferencia de los tipos de datos tratados hasta ahora, es necesario instanciar una fecha mediante la palabra clave New):

```
Var today As DateTime
```

Sin embargo, cuando creas una instancia un DateTime, debes indicarle con qué fecha y hora se empieza. El modo más sencillo de hacerlo es con la función Now que forma parte de DateTime:

```
Var today As DateTime = DateTime.Now
```

Se puede acceder a cada “elemento” de la fecha y la hora mediante sus propiedades. Sin embargo no es posible modificar la mayoría de ellas. Esto se debe al hecho de que puedes crear fácilmente una fecha que no exista ajustando, por ejemplo, Month (mes) a 13. En vez de ello, cuando creas un DateTime le pasas el año, el mes y el día a los cuales quieres ajustar la fecha:

```
Var someDay As New DateTime(1944, 6, 6)
```

También puedes pasar una hora específica. En este ejemplo, hemos añadido 10AM:

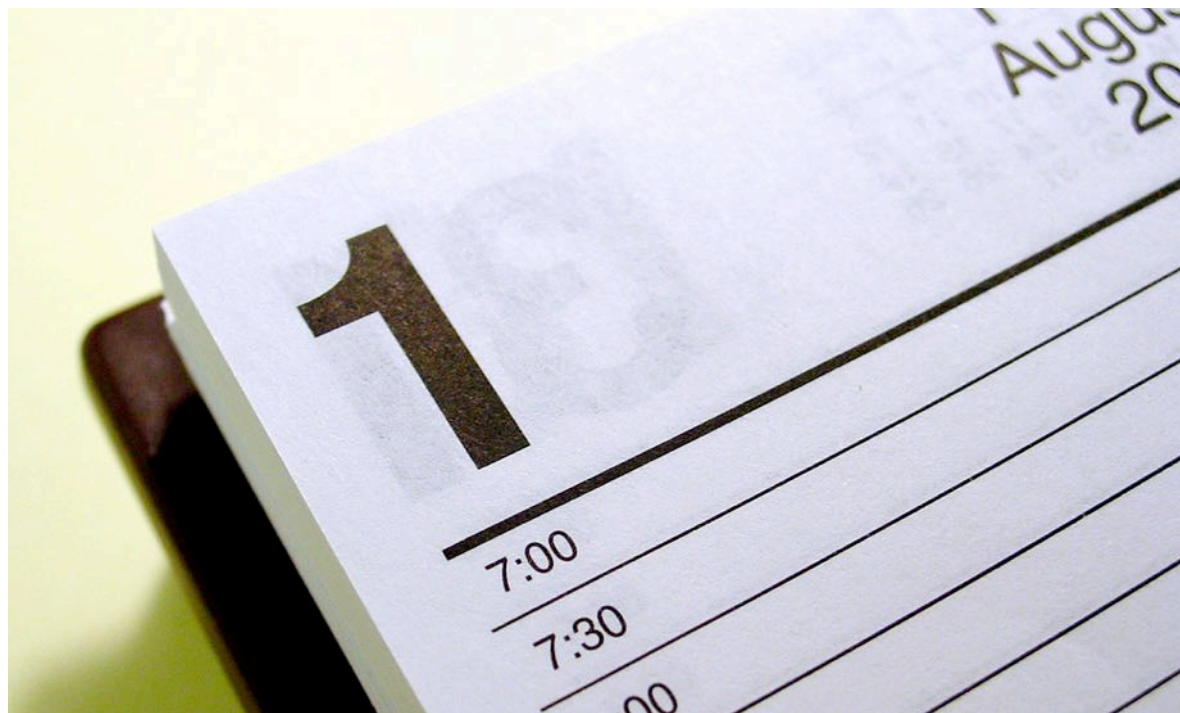
```
Var dDay As New DateTime(1944, 6, 6, 10, 0, 0)
```

Estos valores se almacenan en varias propiedades que puedes recuperar:

```
Var today As New DateTime = DateTime.Now  
Var thisMonth As Integer  
thisMonth = today.Month
```

En este código tienes ahora un entero llamado ThisMonth que contiene una representación numérica del mes actual (Enero = 1, Febrero = 2, etc).

Un objeto DateTime tiene otras propiedades que pueden ser leídas pero no modificadas (en términos de programación, puedes “obtenerlas” pero no “definirlas”). DayOfWeek es un entero que representa la posición del día en la semana (1 = Domingo, 7 = Sábado). DayOfYear es, de igual modo, un entero que representa la posición del día en el año (un objeto de fecha representado el 1 de febrero tendrá un valor de 32 para DayOfYear). También hay una propiedad WeekOfYear.



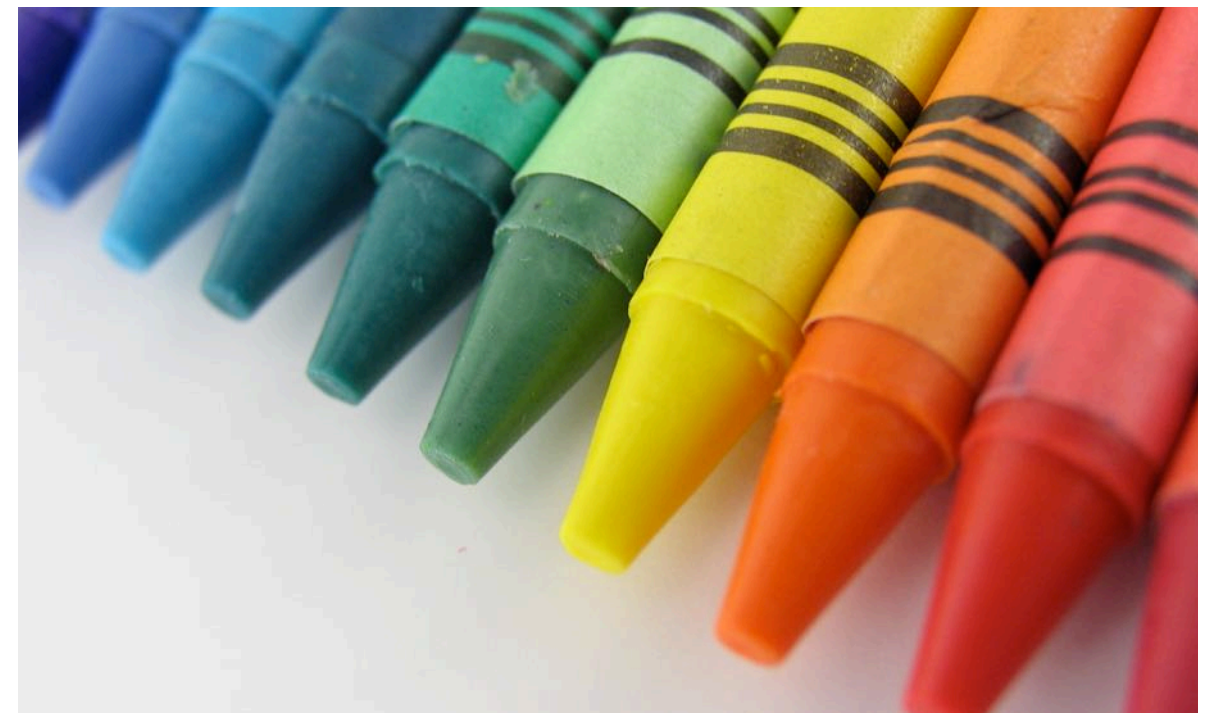
Utiliza la función ToString de DateTime para ver el DateTime como una cadena de texto. Incluso puedes elegir entre varios formatos para mostrar la fecha y hora como texto. Como aparezca el DateTime como cadena variará de los ajustes regionales establecidos en el ordenador del usuario.

COLORES

Puede resultar extraño pensar en un color como una variable, pero Xojo lo permite. Para crear una variable de color:

```
Var myColor As Color
```

Al igual que ocurre con las cadenas, los enteros y los números en coma flotante, no es necesario instanciar los colores. Cuando se crea, el color por defecto es el negro. Puedes cambiar un color en código o bien pidiendo al usuario que seleccione un color. Verás ambos métodos.



Para cambiar un color en código, debes entender algunos fundamentos sobre como se almacenan los colores en Xojo. Un color tiene tres propiedades que se cubrirán aquí: Red, Green y

Blue. Cada una de estas es un entero que puede estar entre cero y 255. Cuanto más alto sea el valor, mayor presencia tendrá ese componente del color. Como referencia, el color negro tendrá el valor cero en cada una de estas propiedades, mientras que el color blanco tendrá el valor 255 para cada una de estas propiedades. El rojo puro tendrá 255 para Red, cero para Green y cero para Blue. El color púrpura, tendrá 255 para Rojo, cero para Verde y 255 para Blue.

Para definir estas propiedades en código, sólo has de asignar un valor a cada uno de ellos con el método RGB:

```
Var thisIsWhite As Color
thisIsWhite = RGB(255,255,255)
Var thisIsPurple As Color
thisIsPurple = RGB(255,0,255)
```

O puedes pedir al usuario final que seleccione un color utilizando la función Color.SelectedFromDialog:

```
Var myFavoriteColor As Color
If Color.SelectdFromDialog =
(myFavoriteColor, "Pick A Color:") Then
    //Ahora tenemos el color del usuario
End If
```

Puede resultar útil añadir anotaciones a tu código como recordatorio de cómo funciona algo o por qué has elegido codificarlo del modo que lo has hecho. Esto se denomina “comentar”. Para comentar el código, inicial la línea con // o el caracter ‘ seguido por tus comentarios. Esto indica a Xojo que puede ignorarlo y no lo tratará como código.

La función Color.SelectedFromDialog toma dos parámetros. El primero es una variable de color, y el segundo es la cadena presentada al usuario (en el código “Pick A Color:”). La función devuelve un booleano (True o False) en función de que el usuario haya seleccionado un color o bien que haya cancelado la selección. Por ello es por lo que la segunda línea del código anterior comienza con un “if” —el resultado está basado en las acciones del usuario, las cuales no puedes controlar o saber por adelantado. Todo el código entre la línea “If” y la línea “End If” se ejecutará si el usuario ha seleccionado un color. Si el usuario ha cancelado la selección, entonces no ocurrirá nada. Por ahora, puedes dejar la sección intermedia del código en blanco, pues aprenderás sobre las condicionales y la lógica en el Capítulo Tres.

Si estás familiarizado con HTML o CSS, o si tienes experiencia trabajando con colores hexadecimales (donde rojo, verde y azul tienen un rango de 00 a FF), entonces también puedes definir colores usando "&c" (conocido como un literal):

```
Var ThisIsBlue As Color  
ThisIsBlue = &c0000FF
```

Si no estás familiarizado con matemática hexadecimal, puedes continuar usando el método RGB para empezar.

2.4 Variables en la Práctica

Ahora que conoces los fundamentos de la creación de variables, es el momento de obtener, definir y comparar sus valores.

En la mayoría de los casos, para definir el valor de una variable utiliza el signo igual. En función del tipo de dato puede que tengas que utilizar o no las comillas. Se requieren en caso de los datos de cadena, pero deben omitirse para los tipos de datos numéricos o booleanos.

```
Var meaningOfLife As Integer
Var chapterTitle As String
Var thisIsEasy As Boolean
meaningOfLife = 42
chapterTitle = "Introduce Yourself"
thisIsEasy = True
```

Como se ha indicado anteriormente, un `DateTime` o los colores son una historia diferente. No se puede definir directamente un valor de fecha (salvo que tengas otro objeto `DateTime` ya creado, en cuyo caso aun tendrás que encargarte de que el `DateTime` original tenga un valor inicial). Para definir el valor de un `DateTime` necesitas algo denominado un *Constructor*.

Un Constructor es un método especial que se ejecuta cuando se instancia un nuevo objeto. Si conoces los valores de fecha

puedes asignarlos cuando instancias una fecha, utilizando para ello la siguiente sintaxis:

```
Var nineEleven As DateTime(2001, 9, 11)
```

El constructor de fecha puede tomar hasta siete parámetros. En el ejemplo anterior sólo has utilizado tres: año, mes y día. Ha de especificarse el año, pero puede prescindirse del mes y del día, en cuyo caso se asume que son 1; cualquier otro valor no indicado se asume como cero. Los parámetros, en orden, son: año como `Integer`, mes como `Integer`, día como `Integer`, hora como `Integer`, minuto como `Integer`, segundo como `Integer`, nanosegundo como `Integer` y `timeZone` como `TimeZone` (por ahora podemos obviar la zona horaria).

Pueden definirse los colores asignando cada una de las propiedades RGB del color:

```
Var logoBackgroundColor As Color
logoBackgroundColor = Color.RGB(255, 0, 255)
```

Por supuesto, programar es generalmente más complicado que simplemente asignar valores tal y como has visto en los ejemplos hasta ahora. Con frecuencia, también implica realizar cálculos. En el caso de datos numéricos, Xojo soporta las operaciones matemáticas habituales que podrías esperar. La suma se realiza con el operador `+` (el signo más); la resta con el operador `-` (el signo menos), y la multiplicación se realiza con el operador `*` (el

signo asterisco). Si has realizado cualquier cálculo matemático en un ordenador, estos deberían de resultarte ya familiares. Estos son algunos códigos de ejemplo:

```
Var unitCost, quantity, totalCost As Double
unitCost = 25
quantity = 4
totalCost = unitCost * quantity
//totalCost is now 100
```

Un fragmento de código que ejecuta una fórmula o función se denomina una expresión. Las expresiones también pueden ser algebraicas y utilizan la misma variable múltiples veces:

```
Var theAnswer As Integer
theAnswer = 25 + 35
//theAnswer is now 60
theAnswer = theAnswer + 10
//theAnswer is now 70
```

La división es ligeramente más complicada. Hay tres operadores relacionados con la división: / (barra), \ (barra invertida) y Mod. El utilizado con más frecuencia es la barra, utilizada para lo que se conoce como división de coma flotante:

```
Var exactAnswer As Double
exactAnswer = 5 / 2
//exactAnswer is now 2.5
```

Con la barra invertida, Xojo realiza una división entera en la que se desprecia la parte fraccional de los valores:

```
Var roundedAnswer As Integer
roundedAnswer = 5 \ 2
//roundedAnswer is now 2
```

Por último, Mod se utiliza para calcular el resto en una operación de división:

```
Var leftoverValue As Double
leftoverValue = 5 Mod 2
//leftoverValue is now 1,
//since 5 divided by 2 is 2 with a remainder of 1
```

Mod también puede resultar útil para saber si un entero es par o impar. Si tu entero Mod 2 es igual a 1, entonces el número es impar. Si la respuesta es cero, el número es par.

El operador más también se puede utilizar con cadenas para concatenarlas (unirlas):

```
Var presidentName As String
presidentName = "Abraham" + " " + "Lincoln"
//presidentName is now "Abraham Lincoln"
```

En el anterior ejemplo, advierte el espacio añadido entre el nombre y el apellido. La omisión del espacio es un error común en la concatenación de cadenas.

Tal y como ocurre con las operaciones matemáticas, el operador más puede utilizar expresiones que se refieran múltiples veces a las mismas variables:

```
Var presidentName As String
presidentName = "Abraham" + " " + "Lincoln"
//presidentName is now "Abraham Lincoln"
presidentName = "President" + " " + presidentName
//presidentName is now "President Abraham Lincoln"
```

Obtener el valor de una variable es generalmente una cuestión sencilla. Cuando se depura, un modo fácil de mostrar el valor de una variable es mediante el uso del método `MessageBox.Show` que ya has utilizado en el Capítulo Uno. Dado que `MessageBox.Show` toma una cadena como su único parámetro requerido, mostrar el valor de una cadena es realmente simple:

```
Var theGreeting As String
theGreeting = "Hello!"
MessageBox.Show(theGreeting)
```

En el caso de las variables numéricas es algo más quisquilloso, dado que el método `MessageBox.Show` no puede tomar directamente un entero o valor de coma flotante (`Double`). Xojo proporciona varias formas para convertir datos numéricos a cadenas, pero el modo más sencillo de hacerlo es con la función `ToString`.

Para usarla, llámala sobre tu variable de entero o `Double`, y mostrará como cadena el valor de dicho número:

```
Var myAgeAsANumber As Integer
Var myAgeAsString As String
myAgeAsANumber = 16
myAgeAsString = myAgeAsANumber.ToString
MessageBox.Show(myAgeAsString)
```

El código anterior almacena el valor de un entero en una cadena y muestra luego dicha cadena al usuario en un diálogo.

Salvo que indiques lo contrario, tu valor numérico no tendrá formato. Por ejemplo, un número elevado puede mostrarse en notación científica, o un valor decimal puede mostrar más posiciones de las que querrías. En estos casos, se puede pasar un formato a la función `ToString`.

La especificación de formato es una cadena que describe cómo debería mostrarse el número. Por ejemplo, si tienes el valor `.25` y quieres mostrarlo como un porcentaje, deberías de utilizar `"#%"` como la especificación de formato:

```
Var myPercent As Double
myPercent = 0.25
MessageBox.Show(myPercent.ToString("#%"))
```

A simple vista puede parecer como algo sin sentido, pero existen unas cuantas reglas para la especificación de formato. En primer

lugar, el signo almohadilla (#) representa el número que quieres formatear. En el ejemplo anterior, el símbolo de porcentaje (%) indica a Xojo que muestre el número como porcentaje, de modo que se multiplica el número por 100 y se muestra a continuación el signo de porcentaje. La siguiente tabla muestra algunas de las posibilidades ofrecidas para la especificación de formato.

Caracter	Descripción
#	Marcador de posición que muestra el dígito del valor si está presente. Si se utilizan menos caracteres de marcador de posición que el número pasado, entonces se redondea el resultado.
0	Marcador de posición que muestra el dígito del valor si está presente. Si el dígito no está presente, se muestra el 0 (cero) en su lugar.
.	Marcador de posición para la coma decimal.
,	Marcador de posición que indica que el número debe formatearse con separadores de miles.
%	Muestra el número multiplicado por 100.
+	Muestra el signo más a la izquierda del número si el número es positivo o menos si el número es negativo.
-	Muestra el signo menos a la izquierda del número si es negativo. No hay efecto para los números positivos.
E or e	Muestra el número usando notación científica.

La siguiente tabla muestra algunos ejemplos poniendo en práctica algunas de estas especificaciones de formato. Una nota rápida: una especificación de formato puede formarse de una a

tres partes separadas por punto y coma. La primera parte es la especificación de formato para números positivos, la segunda parte para números negativos y la tercera parte es para cero. Si sólo se indica una especificación, será la utilizada para todos los números.

Formato	Número	Cadena formateada
###	1,786	1,79
#.0000	1,3	1,3000
0000	5	0005
##%	0,25	25%
###,###.##	145678,5	145.678,5
###e	145678,5	1.46e+05
-#.##	-3,7	-3,7
+#.##	3,7	+3.7

Este es un ejemplo de código con la tabla anterior:

```
Var stickerPrice As Double
stickerPrice = 24995.9
ProgressDialog.Show ▾
(stickerPrice.ToString("###,###.00"))
```

Este debería mostrar “24.995,90” al usuario final.

Hasta ahora has visto como convertir números en cadenas para su visualización, pero también puedes convertir cadenas en números para su uso en cálculos. Como se mencionaba

anteriormente en este capítulo, el mejor modo de hacerlo es usando la función `ToInteger` (o `ToDouble`):

```
Var myAge As String
Var myNumericAge As Integer
Var myAgeInTenYears As Integer
myAge = "18"
myNumericAge = myAge.ToInteger
myAgeInTenYears = myNumerictheMessageAge + 10
//myAgeInTenYears is now 28
```

Ahora que has aprendido algunas cosas sobre las variables así como el modo de obtener y definir sus valores, es el momento de crear el proyecto de ejemplo de este capítulo, “Preséntate a ti mismo”.

2.5 Trabajando con las Variables

Si aún no lo has hecho, abre Xojo y crea un nuevo proyecto Desktop.

1) Haz clic en Window1 en el Navegador.

Aparecerá el Editor de Diseño para Window1.

En el Capítulo Uno utilizaste el Inspector para definir las propiedades de un botón en la ventana. También puedes utilizar el Inspector para definir las propiedades de la ventana propiamente dicha. Sin objetos en la ventana seleccionada, el Inspector modificará las propiedades de la ventana. Vamos a definir tres propiedades de la ventana: title (título), width (ancho) y height (altura).

2) Define el título de Window1 a “Introduce Yourself”.

Advierte que el título (tittle) y nombre (name) de la ventana son diferentes. El título es lo que se muestra a tu usuario final en la parte superior de la ventana cuando se esté ejecutando la aplicación. El nombre es el modo en el que tu, como desarrollador, te refieres a la ventana desde tu código. En una aplicación más compleja siempre es recomendable dar a tus ventanas nombre significativos, pero para el propósito de este proyecto será suficiente con simplemente definir su título.

- 3) Desde la librería, arrastra una Label sobre Window1.
- 4) Cambia al Inspector y define la propiedad Value de Label a “First Name:” y su nombre a “FirstNameLabel”.
- 5) Cambia a la Librería y arrastra un TextField sobre Window1.
- 6) Cambia al Inspector y define el nombre del TextField a “FirstNameField”.
- 7) Añade dos etiquetas más, dos TextField más, y un botón a la ventana, utilizando las propiedades de la siguiente tabla.

Control	Nombre	Texto	Etiqueta
Label	LastNameLabel	Last Name:	N/A
Text Field	LastNameField	N/A	N/A
Label	BirthYearLabel	Birth Year:	N/A
Text Field	BirthYearField	N/A	N/A
Default Button	IntroduceButton	N/A	Introduce

8) Posiciona y ajusta el tamaño de tus controles según tu propia creatividad.

Cuando se hayan añadido todos tus controles a Window1, tu ventana será similar a la siguiente captura de pantalla:



Ahora que la interfaz está completa, tendrás que escribir código para que la app pueda hacer algo. Para añadir código, haz doble clic en IntroduceButton y selecciona el evento Pressed para abrir el Editor de Código.

```

Pressed
-
Var fullName As String
Var currentAge As Integer
Var today As DateTime = DateTime.Now
Var theMessage As String
fullName = FirstNameField.Text + " " + LastNameField.Text
currentAge = today.Year - BirthYearField.Text.ToInteger
theMessage = "Your name is " + fullName + EndOfLine
theMessage = theMessage + "and you will be "
theMessage = theMessage + currentAge.ToString
theMessage = theMessage + " at the end of the year "
theMessage = theMessage + today.Year.ToString + "."
MessageBox(theMessage)

```

Al inicio del capítulo, vimos lo que hará esta aplicación. Como recordatorio, calculará el nombre completo del usuario final y su edad a finales de año, mostrando dicha información al usuario. Usaremos cuatro variables para ello.

1) En el Editor de código, añade las siguientes variables:

```

Var fullName As String
Var currentAge As Integer
Var today As DateTime = DateTime.Now
Var theMessage As String

```

```

Sub Pressed()
    Var fullName As String
    Var currentAge As Integer
    Var today As DateTime = DateTime.Now
    Var theMessage As String
    fullName = FirstNameField.Text + " " + LastNameField.Text
    currentAge = today.Year - BirthYearField.Text.ToInteger
    theMessage = "Your name is " + fullName + EndOfLine
    theMessage = theMessage + "and you will be "
    theMessage = theMessage + currentAge.ToString
    theMessage = theMessage + " at the end of the year "
    theMessage = theMessage + today.Year.ToString + "."
    MessageBox(theMessage)
End Sub

```

Stack	Main Thread	Variables																
Window1.HelloButton.Pressed																		
		<table border="1"> <thead> <tr> <th>Name</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td colspan="2">Globals</td> </tr> <tr> <td>currentAge</td> <td>0</td> </tr> <tr> <td>fullName</td> <td></td> </tr> <tr> <td>me</td> <td>DesktopButton</td> </tr> <tr> <td>self</td> <td>Window1.Window1</td> </tr> <tr> <td>theMessage</td> <td></td> </tr> <tr> <td>today</td> <td>DateTime</td> </tr> </tbody> </table>	Name	Value	Globals		currentAge	0	fullName		me	DesktopButton	self	Window1.Window1	theMessage		today	DateTime
Name	Value																	
Globals																		
currentAge	0																	
fullName																		
me	DesktopButton																	
self	Window1.Window1																	
theMessage																		
today	DateTime																	

FullName es la cadena que utilizarás para almacenar el nombre y apellido concatenados del usuario. CurrentAge almacenará la edad del usuario en años. Today contendrá la fecha actual (observa que el código ya lo ha instanciado, de modo que contiene datos sobre la fecha actual). Por último, theMessage es una cadena que reúne todo para mostrarlo al usuario final.

2) Introduce esta línea de código:

```

fullName = FirstNameField.Text +
    " " + LastNameField.Text

```


Este código concatena el nombre y apellido del usuario. Para ello, tendrás que acceder a las propiedades Text de los TextField. La propiedad Text contiene lo que se muestra en el TextField. Se accede a ello mediante lo que se conoce como notación por punto; algo que ya has utilizado al obtener y definir los valores para las fechas y los colores. La notación por punto es un modo de acceder a las propiedades utilizando el nombre del objeto, seguido de un punto, y seguido a continuación del nombre de la propiedad; como por ejemplo FirstNameField.Text. Esto debería resultar familiar de cuando vimos la concatenación de cadenas al inicio del capítulo.

3) Para determinar los años del usuario, introduce este código:

```
currentAge = Today.Year -  
BirthYearField.Text.ToInteger
```

Anotación: para los propósitos de este proyecto ignoraremos el mes de nacimiento y simplemente calcularemos la edad en años al final del año en curso; tomar los meses en cuenta no es difícil, pero implica unos conocimientos que aun no hemos tratado. Recuperarás los datos introducidos por el usuario en BirthYearField. Recuerda, sin embargo, que BirthYearField.Text te dará una string, que tendrás que convertir a un dato numérico usando ToInteger. Restarás dicho número del año actual para determinar la edad del usuario.

4) Añade esta línea de código:

```
theMessage = "Your name is " + fullName + EndOfLine
```

Ahora tienes todos los datos necesarios para mostrarlos al usuario final. El siguiente paso es unirlos en un mensaje para mostrarlo al usuario final. Harás esto nuevamente mediante la concatenación de cadenas. Para que

tu mensaje resulte más agradable a la vista, también añadirás un salto de línea tras el nombre completo del usuario. Esto se realiza mediante la clase EndOfLine.

Por motivos históricos, macOS, UNIX, y Windows indican el final de línea de forma diferente. Para solucionarlo, la clase EndOfLine de Xojo utilizará el final de línea correcto automáticamente para la plataforma en curso, si bien también es posible utilizar finales de línea específicos cuando sea necesario. En la mayoría de los casos, simplemente deberás de utilizar EndOfLine y estará bien.

5) Introduce este código:

```
theMessage = theMessage + "and you will be "  
theMessage = theMessage + currentAge.ToString  
theMessage = theMessage + " at the end of the year "  
theMessage = theMessage + today.Year.ToString + "."
```

En este punto, theMessage contiene "Your name is ", seguido por el nombre completo del usuario, que ya has calculado, seguido de un salto de línea. Estas líneas de código formarán el resto de theMessage, incluyendo la edad del usuario.

6) Muestra theMessage al usuario final usando esta línea:

```
MessageBox(theMessage)
```

Todo junto, tu código será el siguiente:

```
Var fullName As String  
Var currentAge As Integer
```

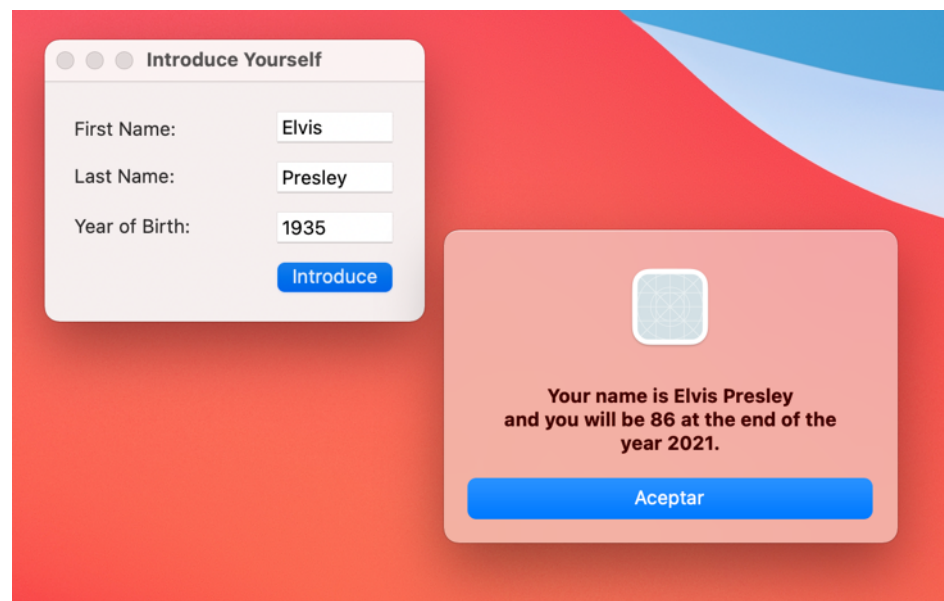
```

Var today As DateTime = DateTime.Now
Var theMessage As String
fullName = FirstNameField.Text & " " & LastNameField.Text
currentAge = today.Year - &#160;
    BirthYearField.Text.ToIntInteger
theMessage = "Your name is " & fullName & EndOfLine
theMessage = theMessage & "and you will be "
theMessage = theMessage & currentAge.ToString
theMessage = theMessage & " at the end of the year "
theMessage = theMessage & today.Year.ToString & "."
MessageBox(theMessage)

```

- 7) **Ejecuta tu proyecto.**
- 8) **Rellena el formulario y pulsa IntroduceButton.**

Deberías ver algo como esto:



- 9) **Sal de tu aplicación.**

2.6 Identificación de Bugs avanzada

En el Capítulo Uno vimos brevemente el depurador de Xojo utilizando la palabra clave Break. Con el proyecto actual “Introduce Yourself” es buen momento para echar un vistazo más en profundidad a los puntos de parada (breakpoints).

Un punto de parada indica un lugar donde quieres que Xojo detenga la ejecución de tu código para acceder al depurador.

Para añadir un punto de parada, ubica el guión de color gris en el margen izquierdo asociado a una línea de código y haz clic sobre él. Este cambiará a un punto de color rojo. Añade un punto de parada en la línea CurrentAge de tu código.

Esto indicará a Xojo que detenga la ejecución de tu aplicación antes de que se ejecute la línea CurrentAge. Para demostrarlo, ejecuta tu proyecto ahora y rellena tu nombre y año de nacimiento. Cuando pulses IntroduceButton, verás el depurador.

Como viste en el Capítulo Uno, se destacará en gris la línea actual de código y el panel derecho nos proporcionará un listado de variables. Este es un detalle de tus variables:

Variables	
Name	Value
Globals	
currentAge	0
fullName	
me	DesktopButton
self	Window1.Window1
theMessage	
today	DateTime

La cadena FullName ya está compuesta, pero CurrentAge tiene el valor cero. Esto se debe a que esta línea de código:

```
currentAge = today.YearBirthYearField.¬
Value.ToInteger
```

aun no se ha ejecutado. Cuando defines un punto de parada, ten cuidado sobre su ubicación, o puede que no veas los resultados esperados.

En la barra de herramientas del depurador, ubica el botón Step.



Haz clic en dicho botón para saltar a la siguiente línea de código. Ahora se destacará la siguiente línea en gris, y el panel de variables cambiará para mostrar el valor calculado de Currentage. Puedes continuar pulsando el botón Step para caminar por la ejecución del código. Si estás listo para volver a la aplicación en funcionamiento, pulsa el botón Resume en la barra de herramientas del depurador.



Si algo ha ido tan mal que necesitas volver a tu código, pulsa el botón Stop.



Cuando se desarrolla una aplicación compleja es frecuente que te veas usando decenas de puntos de parada repartidos por tu código. Si quieres eliminarlos todos de una vez, dirígete al menú Project y selecciona Breakpoint->Clear All.

¿Dónde vamos ahora?

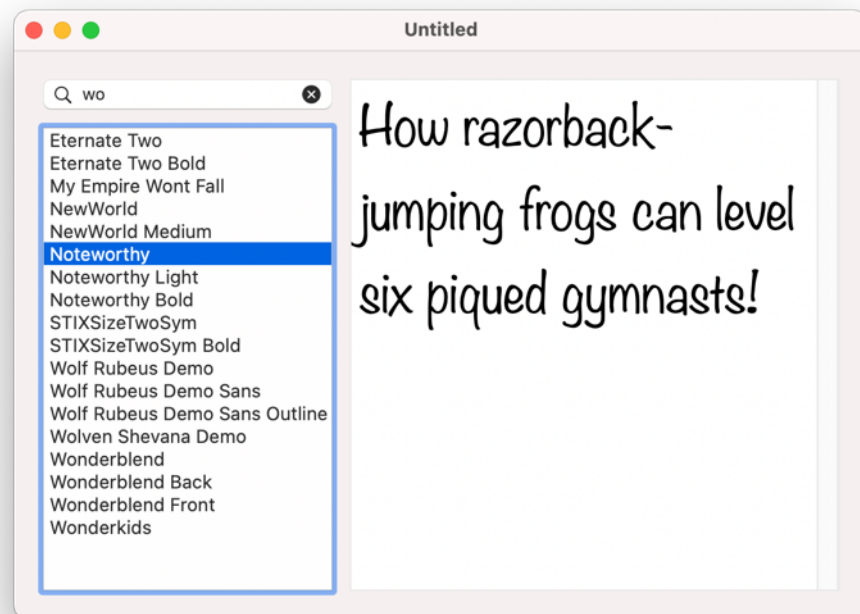
CONTENIDOS

1. **Introducción del Capítulo**
2. **If... Then**
3. **Select Case**
4. **For... Next**
5. **Do... Loop**
6. **While... Wend**
7. **Exit y Continue**
8. **Bucles en la Práctica**

3.1 Introducción del Capítulo

Hemos visto como obtener y definir valores. En una aplicación compleja tendrás que aplicar bastante lógica. En este capítulo aprenderás como comprobar y comparar variables, respondiendo adecuadamente en tu código, y controlando el flujo.

Como proyecto de este capítulo crearemos un previsualizador de fuentes. Listará todas las fuentes instaladas en el ordenador. Cuando una fuente está seleccionada, tu aplicación mostrará una vista previa de dicha fuente. Tu aplicación también nos permitirá buscar fuentes concretas. Esta captura muestra el resultado:



3.2 If...Then

El problema lógico más común con el que te encontrarás como desarrollador es el de comprobar si un valor coincide con otro valor. Por ejemplo, puede que necesites comprobar si un booleano es verdadero o falso, o puede que necesites comprobar si un entero es igual o mayor que 100. También es posible que necesites comprobar si una cadena contiene otra cadena. Este tipo de lógica se realiza utilizando If.

La mayoría de los lenguajes de programación, como C#, Swift, C, C++, PHP, Java, incluyen la palabra clave If. Algunos lenguajes usan una sintaxis ligeramente diferente, pero el resultado es el mismo. Una vez que aprendas como usar If...Then en Xojo, podrás utilizarlo también en cualquier otro lenguaje.

Para usar uno de los anteriores ejemplos, imagina que quieres comprobar si un booleano es cierto o falso:

```
Var theLightsAreOn As Boolean
theLightsAreOn = True
If theLightsAreOn = True Then
    //React here to the fact that the lights are on
End If
```

¿Qué significa theLightsAreOn? theLightsAreOn es una variable booleana, tal y como has aprendido en el capítulo anterior. Su

valor puede proceder de cualquier parte en tu código, o bien puede proceder del usuario haciendo clic sobre una casilla de verificación o sobre un botón de radio.

Advierte como la instrucción If sigue cierta estructura. La línea siempre empieza con If, seguido por la expresión a evaluar, seguido por Then. La siguiente sección de código siempre estará *indentada automáticamente* por Xojo, y se ejecutará sólo si la expresión se evalúa como cierta. Siguiendo a dicho código está la línea final: End If.

Observa que si estás comprobando el valor de un booleano, puedes omitir la parte “ = True” o “ = False” de la expresión:

```
Var theLightsAreOn As Boolean
theLightsAreOn = True
If theLightsAreOn Then
    //React here to the fact that the lights are on
End If
```

Si has utilizado cualquier lenguaje basado en C en el pasado, observarás algo interesante sobre Xojo. En la mayoría de los lenguajes basados en C, debes utilizar un doble signo de igualdad para comparar valores (If MyVar1 == MyVar2), mientras que aquí sólo utilizas un signo de igualdad. Xojo sobrecarga el operador de igualdad de modo que pueda utilizarse tanto para comparar como para asignar valores. De hecho, si usas “==” en Xojo, este te advertirá de ello.

En el anterior ejemplo sólo indicas a Xojo qué hacer si theLightsAreOn es cierto. Hay ocasiones en las que necesitas reaccionar también si el valor es falso. Para ello, utiliza la palabra clave Else:

```
Var theLightsAreOn As Boolean
theLightsAreOn = True
If theLightsAreOn Then
    //React here to the fact that the lights are ON
Else
    //React here to the fact that the lights are OFF
End If
```

Por supuesto, no estás limitado a comprobar sólo valores booleanos. Si necesitas comprobar un entero también puedes hacerlo:

```
Var myAge As Integer
myAge = 16
If myAge > 17 Then
    MessageBox("You can vote.")
Else
    MessageBox("You cannot vote yet.")
End If
```

En el anterior ejemplo estás utilizando el símbolo mayor que en vez del signo igual. Cuando se utiliza mayor que o menor que, recuerda que no incluyen el número que estás comparando. En otras palabras, en este ejemplo si myAge fuese igual a 17 el mensaje sería “You cannot vote yet.”

Si quieres que las comparaciones sean inclusivas puedes utilizar en tus expresiones los operadores para “mayor que o igual a” y “menor que o igual a”:

```
If myAge >= 17 Then
    MessageBox("You can vote.")
Else
    MessageBox("You cannot vote yet.")
End If
```

Puedes usar cualquier tipo con If. Este ejemplo usa cadenas:

```
Var myString As String
myString = "Hello"
If myString = "Hello" Then
    MessageBox("We have a match!")
Else
    MessageBox("We do not have a match!")
End If
```

Las comparaciones de cadena en Xojo no son sensibles a mayúsculas/minúsculas. El anterior ejemplo podría reescribirse como sigue, obteniendo la misma funcionalidad:

```
Var myString As String
myString = "Hello"
If myString = "hello" Then
    MessageBox("We have a match!")
Else
    MessageBox("We do not have a match!")
End If
```


En un sistema que no diferencia entre mayúsculas/minúsculas, una “a” minúscula será igual a la “A” mayúscula; se considera que los caracteres son iguales. Si necesitas realizar una comparación de cadenas que sea sensible a mayúsculas/minúsculas, puedes utilizar la función `String.Compare`.



A menudo necesitarás determinar si una cadena contiene otra cadena. Por ejemplo, puede que necesites comprobar si un dato introducido por el usuario contiene una palabra clave en particular. Esto puede realizarse con la función `IndexOf`, la cual (como `Compare`) es un método de `String`. `IndexOf` puede tomar hasta cuatro parámetros y devuelve un entero. El primer parámetro es un entero y es opcional. Indica la posición en la que se debe comenzar la búsqueda de una cadena dentro de otra. Por defecto este valor es cero. El segundo parámetro es la

cadena a buscar. El valor devuelto por `IndexOf` es un entero que indica la posición en la cadena fuente a partir de la cual se ha encontrado la cadena buscada. Si no se encuentra, `IndexOf` devolverá cero. Cualquier valor distinto de cero indica que se ha encontrado la cadena.

```
Var alphabet, part As String
Var location As Integer
alphabet = "abcdefghijklmnopqrstuvwxyz"
part = "def"
location = alphabet.IndexOf(part)
//The part was found and location = 4
part = "foo"
location = alphabet.IndexOf(part)
//The part was not found and location = 0
```

Por ejemplo, asumamos que has preguntado al usuario por una breve descripción acerca de su currículum y que necesitas mostrar un mensaje de error si el usuario incluye la palabra “ninja” en cualquier parte de su bio. La interfaz de tu app incluiría un `TextField` llamado `BioField`, donde el usuario introduciría su información biográfica.

```
If BioField.Text.IndexOf("ninja") > 0 Then
    //Issue a ninja alert!
Else
    //This is a ninja-free zone
End If
```

Si recuerdas en el proyecto Introduce Yourself del Capítulo Dos, se solicitaba al usuario que introdujese su año de nacimiento en un TextField. Entonces tomabas dicho valor y lo convertías a un entero. Lo que no hacías, sin embargo, es verificar que dicho dato era efectivamente un número. Xojo tiene una función llamada IsNumeric que puede comprobar si el valor de una string puede convertirse a un dato numérico. Su uso es sencillo:

```
Var numberText As String
numberText = "123"
If IsNumeric(numberText) Then
    MessageBox(numberText + " can be converted")
    //This is the message box that will appear
Else
    MessageBox(numberText + " can't be converted")
End If
numberText = "Marbles"
If IsNumeric(numberText) Then
    MessageBox(numberText + " can be converted")
Else
    MessageBox(numberText + " can't be converted")
    //This is the message box that will appear
End If
```

IsNumeric no está limitado a enteros. También interpretará correctamente números en coma flotante e incluso notación científica.

Si necesitas comparar valores DateTime, el mejor modo de hacerlo es comparando sus propiedades SecondsFrom1970:

```
If today.SecondsFrom1970 > ¬
yesterday.SecondsFrom1970 Then
    //Time is moving forward
End If
```

También puedes utilizar otras propiedades de fecha para la comparación, como el año, mes, día, etc. O si necesitas comprobar una fecha para ver si es parte de un año dado:

```
If today.Year = 2012 Then
    //The world may have ended.
    //Be cautious.
End If
```

¿Qué pasa con el fin del mundo? Algunas personas creían que la antigua profecía Maya predecía que el mundo terminaría en Diciembre de 2012. Obviamente, no lo hizo.

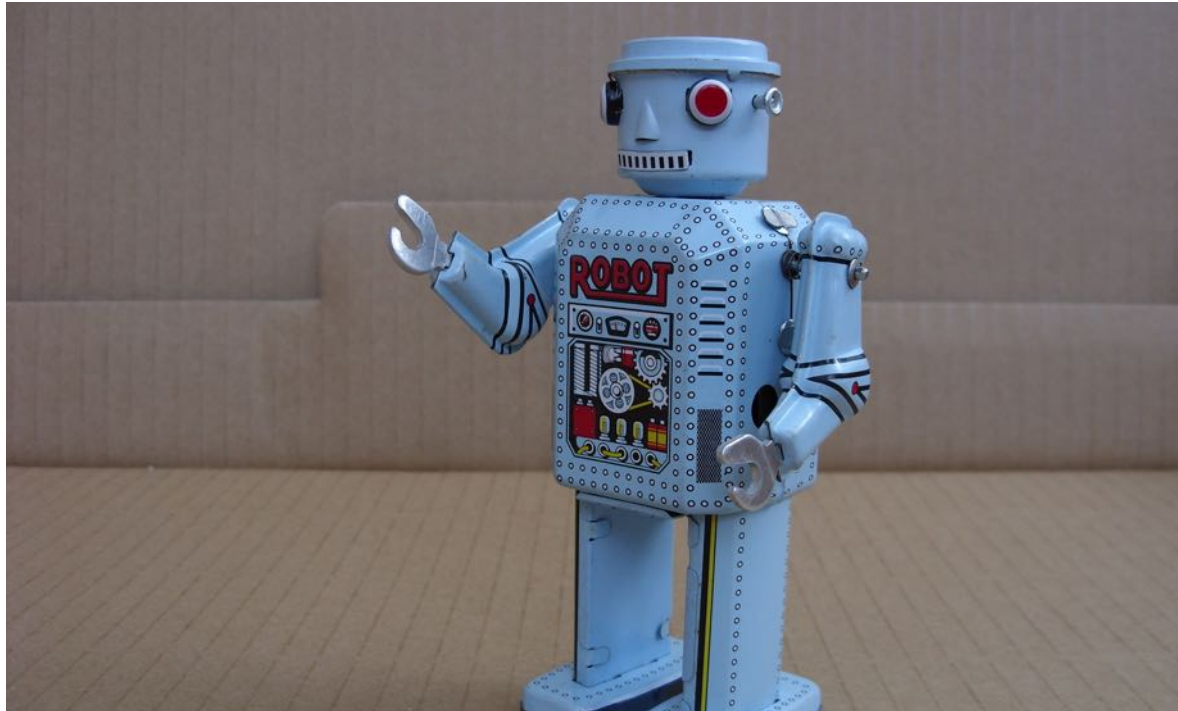
También puedes comprobar más de una posibilidad:

```
If today.Year = 9999 Or today.Year = 10000 Then
    //Be aware of potential Y10K problems.
End If
```

El código anterior utiliza “Or” para separar tus condiciones, de modo que la expresión se evaluará como cierta si se cumple cualquier condición. También puedes utilizar “And” para separar las condiciones, en cuyo caso es necesario que se cumplan

ambas condiciones; si sólo es cierta una de ellas, entonces la expresión se evaluará como falsa.

```
If today.Year = 2012 And today.Month = 12 Then
    //The world may be ending really soon. Be
    cautious.
End If
```



Hay momentos en los que tendrás que comprobar múltiples condiciones y reaccionar en consecuencia. Por ejemplo, unos párrafos antes comprobaste por “ninjas”. ¿Qué pasa si tu código también necesita identificar piratas, zombies o robots? Puedes añadir Elseif entre el If y el End If; o bien puedes añadir varias instrucciones Elseif:

```
If BioField.Value.IndexOf("ninja") > 0 Then
```

```
    //Issue a ninja alert!
ElseIf BioField.Value.IndexOf("robot") > 0 Then
    //Issue a robot alert!
ElseIf BioField.Value.IndexOf("pirate") > 0 Then
    //Issue a pirate alert!
ElseIf BioField.Value.IndexOf("zombie") > 0 Then
    //Issue a zombie alert!
Else
    //We are safe from perceived threats
End If
```

Observa que Xojo saltará al final de la instrucción If tan pronto como se cumpla una de las condiciones. En otras palabras, si el anterior código encuentra un robot, no se comprobará la bio del usuario por referencias a piratas o zombies. Si necesitas comprobar cada una de ellas de forma separada, sólo has de separar la instrucción If para cada caso.

La instrucción Elseif añade un buen grado de flexibilidad a la lógica de Xojo. Pero una vez que hayas añadido más de unos cuantos Elseif, tu código empezará a tornarse difícil de manejar. Afortunadamente, hay un mejor modo de hacerlo.

3.3 Select Case

La instrucción Select Case de Xojo te proporciona como desarrollador un modo más claro de comprobar una variable o expresión frente a múltiples valores. Te permite indicar cada condición que desees comprobar, utilizando la instrucción Case, así como el código que debería de ejecutarse cuando se cumpla la condición. Xojo almacena el mes actual como un número entre 1 y 12; de modo que el siguiente código te indicará el nombre del mes actual:

```
Var today As New Date
Select Case today.Month
Case 1
    MessageBox("Es Enero")
Case 2
    MessageBox("Es Febrero")
Case 3
    MessageBox("Es Marzo")
Case 4
    MessageBox("Es Abril")
Case 5
    MessageBox("Es Mayo")
Case 6
    MessageBox("Es Junio")
Case 7
    MessageBox("Es Julio")
Case 8
    MessageBox("Es Agosto")
Case 9
    MessageBox("Es Septiembre")
```

```
Case 10
    MessageBox("Es Octubre")
Case 11
    MessageBox("Es Noviembre")
Case 12
    MessageBox("Es Diciembre")
Else
    MessageBox("No se puede determinar.")
End Select
```

También puedes indicar la acción por defecto utilizando Else, tal y como se ha visto en el anterior ejemplo. Observa que en una aplicación bien escrita, cualquier acción indicada bajo Else nunca debería de ocurrir, pero tu código ha de estar preparado para manejarlo en cualquier caso.

Cada instrucción Case también puede tener un rango de valores. Puedes indicarlo con una lista separada por comas o utilizando la palabra clave To. Este es un ejemplo que usa ambas formas:

```
Var today As New Date
Select Case today.Month
Case 1, 2, 3
    MessageBox("It's Q1")
Case 4, 5, 6
    MessageBox("It's Q2")
Case 7 To 9
    MessageBox("It's Q3")
Case 10 To 12
    MessageBox("It's Q4")
Else
    MessageBox("Unable to determine")
```



```
current quarter")  
End Select
```

Otros lenguajes de programación comunes utilizan “switch” en vez de “select case” para este tipo de comparación. Más allá de este mínimo cambio, la funcionalidad es idéntica.

3.4 For...Next

Cuando necesites realizar una misma operación sobre una serie de variables, el bucle For...Next es un modo fácil de hacerlo. En su forma más simple, un bucle For...Next simplemente recorre una lista y ejecuta el código indicado. Cada bucle For...Next necesita una variable que usará como contador.

Crea un nuevo proyecto desktop en Xojo. En la vista de Diseño de Window1, haz doble clic en la ventana para abrir su Editor de Código. Ubica el evento Opening de Window1 e introduce este código:

```
Var counter As Integer
For counter = 1 To 10
    MessageBox(counter.ToString)
Next
```

A continuación, ejecuta tu proyecto. Deberías ver 10 mensajes, desde el número 1 hasta el número 10. Tras cerrar los 10 mensajes, sal de la aplicación.

No estás limitado a utilizar sólo el contador. Cambia tu código por el siguiente:

```
Var counter As Integer
For counter = 1 To 10
    MessageBox(Str(counter * 2))
Next
```

Ejecuta de nuevo tu proyecto para ver la diferencia.

Tu contador también puede decrementarse en vez de incrementarse:

```
Var counter As Integer
For counter = 10 DownTo 1
    MessageBox(counter.ToString)
Next
```

El contador también puede saltar números, usando la palabra clave Step:

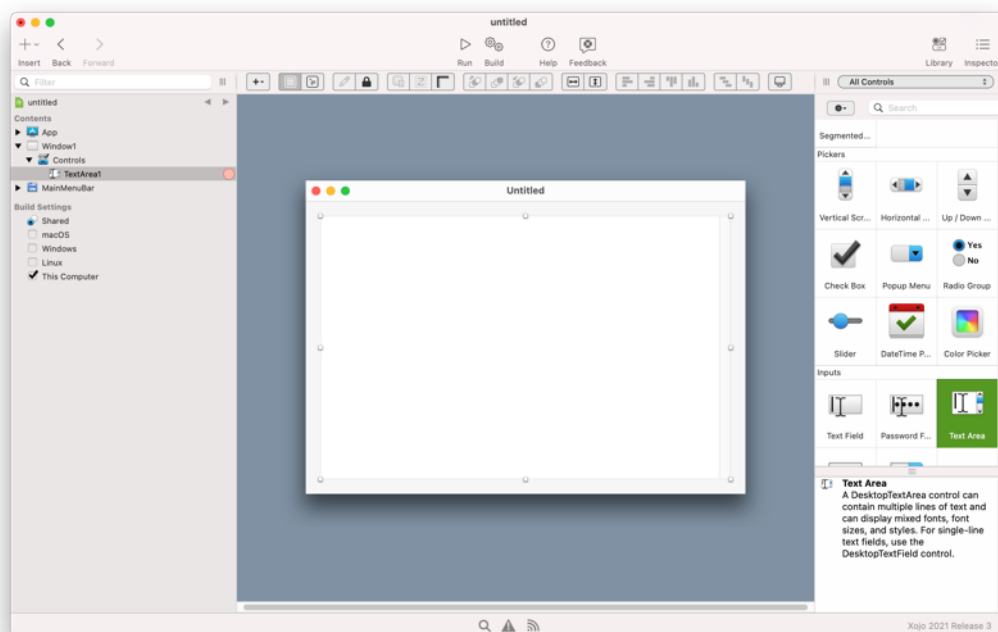
```
Var counter As Integer
For counter = 1 To 10 Step 2
    MessageBox(counter.ToString)
Next
```

Cada vez que utilices un bucle For...Next tu código hará algo más interesante que simplemente mostrar un mensaje. Hay ocasiones en las que tendrás un grupo de objetos similares. Y probablemente los guardes en un array. Aprenderás sobre los array con detalle en el Capítulo Cinco; por ahora, piensa en un array como en una lista numerada de objetos o variables similares. Esta sección y las siguientes cubrirán como recorrer grupos de objetos, pero en vez de introducirnos en los detalles relativos al funcionamiento de los array, verás un modo fácil de

acceder a un array de tu ordenador: tus fuentes. Xojo incorpora una función llamada FontCount que devuelve el número de fuentes instaladas. Utilizarás dicha función para reunir un grupo de objetos, y que en este caso será una lista de fuentes.

- 1) **Crea un nuevo proyecto desktop en Xojo.**
- 2) **Abre la Vista de Diseño para Window1 y añade un control TextArea a la ventana.**

Cambia su tamaño para que use la mayor parte de la superficie de la ventana:



- 3) **Una vez que tengas el TextArea en su sitio, haz doble clic sobre ella para acceder a su Editor de Código y ubica su evento Opening.**

En dicho evento vas a necesitar dos variables:

```
Var counter As Integer
Var myFontList As String
```

- 4) **Define un bucle For...Next.**

```
For counter = 0 To System.FontCount - 1
    myFontList = myFontList &
        System.FontAt(counter) & EndOfLine
Next
```

Este bucle irá añadiendo nombres de fuentes a MyFontList. Cuando se complete la lista, la mostrará en la TextArea.

No te preocupes sobre cómo funciona System.FontAt(Counter). Esto quedará más claro en el Capítulo Cinco. Por ahora, sólo necesitas saber que te proporciona el nombre de la siguiente fuente. El nombre de la fuente está seguido por un salto de línea.

- 5) **Tras el bucle, muestra la lista en la TextArea:**

```
Me.Text = myFontList
```

Todo junto, el código del evento Open del TextArea debería verse así:

```
Var counter As Integer
Var myFontList As String

For counter = 0 To System.FontCount - 1
    myFontList = myFontList &
        System.FontAt(counter) & EndOfLine
Next

Me.Text = myFontList
```

6) Ejecuta tu proyecto.

Tras unos segundos verás un listado con las fuentes de tu ordenador mostrada en el TextArea.

7) Sal de tu aplicación.

Veamos ahora como acelerar tu código.

Tu bucle empieza con esta línea:

```
For counter = 0 To FontCount - 1
```

Recuerda que FontCount (que es parte del módulo System - por tanto, System.FontCount) es una función incluida en Xojo. Una cosa que debes recordar es que cada vez que se ejecuta este bucle también se ejecuta dicha función. Si tienes un ordenador rápido o pocas fuentes puede que no adviertas el impacto en el rendimiento, pero hay un modo de optimizar este código. Añade una nueva variable:

```
Var myFontCount As Integer
```

Utiliza MyFontCount para cachear el resultado de la función FontCount:

```
myFontCount = System.FontCount - 1
```

Puede que te preguntes por qué restas uno de FontCount. Esto se debe a que las fuentes del ordenador se almacenan en un array, y los array en Xojo utilizan el cero como primer valor del índice, tal y como ocurre con otros lenguajes de programación. Esta explicación será más clara en el Capítulo Cinco.

8) Cambia la primera línea de tu bucle:

```
For Counter = 0 To myFontCount
```

Ahora tu bucle se referirá al valor cacheado en vez de ejecutar la función FontCount cada vez que. Tu nuevo código será como este:

```
Var counter As Integer
Var myFontList As String
Var myFontCount As Integer

myFontCount = System.FontCount - 1

For counter = 0 To myFontCount
    myFontList = myFontList &
        + System.FontAt(counter) + EndOfLine
Next
Me.Text = myFontList
```

9) Ejecuta tu proyecto y deberías ver el mismo resultado.

Puede que no adviertas la diferencia de rendimiento en este ejemplo, pero cuando tus proyectos aumenten en complejidad, este tipo de optimización será más evidente.

10) Sal de la aplicación.

3.5 Do...Loop

Otro tipo de bucle disponible en Xojo es Do...Loop. Un Do...Loop es útil cuando necesitas hacer una comprobación para una determinada condición cada vez que se ejecuta el bucle.

Do...Loop tiene dos formas, una donde no se garantiza que el bucle se ejecute una vez como mínimo, y otra en la que se garantiza.

Este es un ejemplo de Do...Loop (como en el resto de los ejercicios de este libro, siéntete libre de crear un nuevo proyecto desktop en Xojo e introducir este código en el evento Open de Window1 para probarlo):

```
Var x As Integer
x = 1
Do Until x > 100
    x = x * 2
    MessageBox(x.ToString)
Loop
```

Si ejecutas este proyecto verás una serie de mensajes, cada uno mostrando un valor que duplica el anterior: 2, 4, 8, 16, 32, 64, 128. Una vez que alcanza 128, el bucle se detiene porque has indicado que finalice una vez que x sea mayor que 100.

Probemos ahora una versión ligeramente distinta para este bucle. Define x a 101:

```
Var x As Integer
x = 101
Do Until x > 100
    x = x * 2
    MessageBox(x.ToString)
Loop
```

Ejecuta el proyecto de nuevo y no aparecerá ningún mensaje. Esto se debe a que la condición ($x > 100$) ya se ha cumplido, de modo que se sale del bucle. Sin embargo, si cambias de nuevo el bucle deberías de ver un resultado distinto:

```
Var x As Integer
x = 101
Do
    x = x * 2
    MessageBox(x.ToString)
Loop Until x > 100
```

En esta ocasión aparecerá un mensaje conteniendo el número 202. Cuando la palabra clave Until se encuentra al final del bucle, este se ejecutará una vez como mínimo. Si la palabra clave Until está al comienzo del bucle, es posible que no se ejecute en el caso de que ya se cumpla la condición.

3.6 While...Wend

Un tercer tipo de bucle es While...Wend (Wend es una abreviatura de While End). Este es similar a Do...Loop. He aquí un ejemplo:

```
Var x As Integer
While x < 100
    x = x + 1
Wend
MessageBox(x.ToString)
```

Cuando se ejecute, este código mostrará un mensaje conteniendo el número 100. El bucle While...Wend es particularmente útil cuando se trabaja con bases de datos, tal y como verás en el Capítulo Doce.

3.7 Exit y Continue

Hay casos en los que tu bucle está buscando un valor y puede detener la búsqueda una vez que se encuentre. Para ello podemos usar la instrucción Exit. Para utilizar una versión modificada del anterior ejemplo de fuentes, asume que quieres salir del bucle For...Next una vez que hayas encontrado la fuente Courier New. Añade estas tres líneas de código, una simple instrucción If...Then:

```
If System.FontAt(counter) = "Courier New" Then
    Exit
End If
```

Añadirás este código dentro del bucle For...Next, de modo que el código completo será como este:

```
Var counter As Integer
Var myFontList As String
Var myFontCount As Integer

myFontCount = System.FontCount - 1

For counter = 0 To myFontCount
    If System.FontAt(counter) = "Courier New" Then
        Exit
    End If
    myFontList = myFontList &
        + System.FontAt(counter) + EndOfLine
Next
```

```
Me.Text = myFontList
```

Si ejecutas este proyecto, el TextArea utilizado para mostrar el listado de fuentes sólo listará ahora las fuentes hasta (y no incluida) Courier New. Si quieres que se incluya Courier New, deberías mover la instrucción If...Then tras la línea que crea la lista de fuentes:

```
myFontList = myFontList &
System.FontAt(counter) + EndOfLine
If System.FontAt(counter) = "Courier New" Then
    Exit
End If
```

El uso de la instrucción Exit para salir de un bucle es otra buena forma de optimizar tu código, de modo que tu usuario pase menos tiempo esperando.

Otra forma de modificar el comportamiento de tus bucles es con la instrucción Continue. Supongamos que tienes un bucle como este (este pseudo código no es para ejecutarlo):

```
Var x As Integer
For x = 1 To 100
    DoMyFunction(x)
    DoAnotherFunction(x)
    DoAThirdFunction(x)
Next
```

Asumamos que DoMyFunction, DoAnotherFunction y DoAThirdFunction son funciones que harán algo interesante con tu entero x. Pero supongamos que sólo quieres ejecutar la primera función para el número 72. Este código debería de hacerlo:

```
Var x As Integer
For x = 1 To 100
    DoMyFunction(x)
    If x = 72 Then
        Continue
    End If
    DoAnotherFunction(x)
    DoAThirdFunction(x)
Next
```

La instrucción Continue indica a Xojo que detenga dicha iteración del bucle y vuelva al principio. Este es un modo de ejecutar sólo la primera función para los números pares, mientras que los impares ejecutarán las tres funciones:

```
Var x As Integer
For x = 1 To 100
    DoMyFunction(x)
    If x Mod 2 = 0 Then
        Continue
    End If
    DoAnotherFunction(x)
    DoAThirdFunction(x)
Next
```

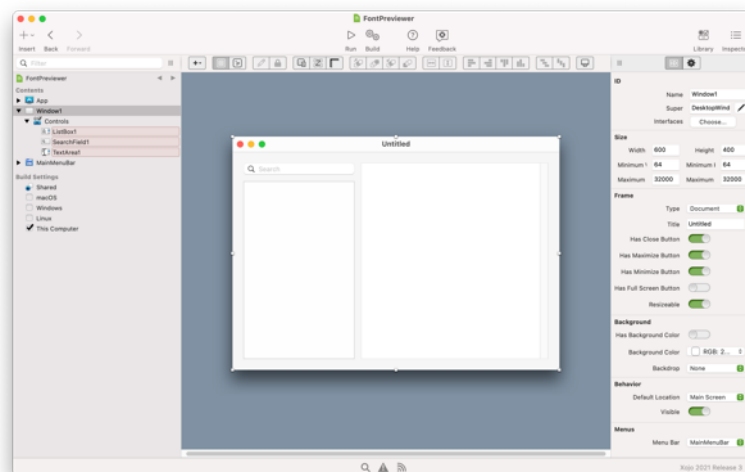
3.8 Bucles en la práctica

Para el proyecto de este capítulo crearemos el visualizador de fuentes que vimos al inicio del capítulo.

- 1) **Crea un nuevo proyecto desktop en Xojo y guárdalo como “FontPreviewer”.**
- 2) **Define el título de Window1 como “Font Previewer”.**
- 3) **Window1 tendrá tres controles: un SearchField, un ListBox, y un TextArea. Utiliza los siguientes nombres:**

Control	Nombre
SearchField	Search
TextArea	PreviewField
ListBox	FontListBox

Diseña tu interfaz de usuario usando tu propia creatividad, puedes utilizar esta imagen como guía:



- 4) **Define la propiedad Text de PreviewField a: “How razorback-jumping frogs can level six piqued gymnasts!”**

Dado que tu aplicación previsualizará fuentes, el usuario final necesitará algún texto que pueda ver. Esta es una frase suficientemente corta que muestra todas las letras del alfabeto inglés. Además, define su propiedad TextSize a 36. La verás que en la pestaña Advanced del Inspector (el icono de rueda dentada).

Tu interfaz está completa ahora. Este proyecto tendrá mucho más código en comparación con el del anterior capítulo, pero también hará mucho más. El ListBox, con el nombre FontListBox, mostrará un listado con todas las fuentes instaladas en el ordenador. Al hacer clic sobre el nombre de cada una de ellas se actualizará el PreviewField y cambiará la fuente por la seleccionada. El pequeño SearchField sobre el FontListBox, que se ha nombrado como Search, permitirá al usuario que busque fuentes en el ordenador. El usuario podrá introducir parte o todo el nombre de una fuente, momento en el que se actualizará el FontListBox para mostrar sólo las fuentes coincidentes.

Con esto en mente, lo primero que has de hacer es crear tu listado de fuentes. Trabajarás con más controles y eventos que los aprendidos hasta ahora, pero no te preocupes por los detalles técnicos en este punto. Estos problemas se cubrirán en profundidad en el Capítulo Seis. Para crear tu listado de fuentes, haz doble clic en FontListBox para acceder a la ventana Add Event Handler. Selecciona el evento Opening y haz clic en OK.

- 5) **Añade el siguiente código en el evento Opening de FontListBox:**

```
Var counter, myFontCount As Integer  
myFontCount = System.FontCount
```

```
For counter = 0 To myFontCount - 1
```



```
Me.AddRow(System.FontAt(counter))  
Next
```

Todo o parte de este código debería de resultarte familiar. Estás usando la función FontCount para determinar la cantidad de fuentes en el ordenador y para recorrerlas con un bucle For...Next. En cada iteración del bucle añades una nueva fila al FontListBox. Observa que en cualquiera de los eventos de un control, el uso de “Me” se refiere al control. Por tanto:

```
Me.AddRow(System.FontAt(counter))
```

es equivalente a:

```
FontListBox.AddRow(System.FontAt(counter))
```

Usando Me en vez del nombre del control es un atajo, aunque puede resultar muy útil en el caso de que posteriormente cambies el nombre del control.

6) **Añade el manejador de evento SelectionChanged a FontListBox (el botón “+” en la barra de herramientas). Añade este código:**

```
If Me.SelectedRowIndex <> -1 Then  
    PreviewField.FontName = Me.SelectedRowValue  
End If
```

El evento SelectionChanged tiene lugar cuando el usuario selecciona o deselecciona una fila en el ListBox. Dado que el usuario puede estar quitando la selección, has de comprobar si está seleccionada una fila o no.

Nuevamente, no te preocupes mucho ahora sobre cómo funciona el código. Quedará más claro en los siguientes capítulos.

7) **Ejecuta el proyecto y prueba a seleccionar diferentes fuentes.**

Cada selección debería cambiar la fuente mostrada en PreviewField. El SearchField aún no funciona, pero lo hará luego.

8) **Sal de la aplicación.**

9) **Haz doble clic en el SearchField y añade el manejador de evento TextChanged con este código:**

```
Var searchString, theFontName As String  
Var counter, myFontCount As Integer
```

Este evento tiene lugar cada vez que se modifica el texto en el SearchField, ya sea escribiendo, borrando o pegando texto. El código de este evento será similar al del evento Opening de FontListBox, pero tendrás que hacer algunas comprobaciones adicionales. Este código empieza declarando las variables.

Tienes searchString, theFontName así como dos enteros: Counter y MyFontCount, que deberían de resultar familiares de los anteriores ejemplos:

10) **Antes de continuar, guarda FontCount:**

```
myFontCount = System.FontCount-1
```

Esta optimización es incluso más importante aquí, dado que este código se ejecutará cada vez que se escriba una letra o se borre en el SearchField. Dado que se ejecutará con frecuencia, ha de ser lo más rápido posible.

11) Añade este código:

```
If Me.Text <> "" Then
    //We'll do more stuff here
End If
```

Esto comprueba si SearchField contiene texto. Si es así, recorrerás las fuentes y mostrarás las coincidentes. Si no, mostrarás el listado completo de fuentes.

Este código puede tomar dos caminos. En primer lugar, si el SearchField no está vacío, recorrerás todas las fuentes y mostrarás sólo aquellas cuyo nombre contenga lo almacenado en SearchString. El siguiente código se encarga de esto:

```
searchString = Me.Text
For counter = 0 To myFontCount
    theFontName = System.FontAt(counter)
    If theFontName.IndexOf(searchString) >= 0 Then
        FontListBox.AddRow(theFontName)
    End If
Next
```

Por otra parte, si SearchField está vacío deberías asumir que el usuario final quiere ver todas las fuentes instaladas en el ordenador. Este es el código para ello (observa que es similar al modo en el que se rellena originalmente el FontListBox):

```
For counter = 0 To myFontCount - 1
```

```
FontListBox.AddRow(System.FontAt(counter))
Next
```

12) Añade una línea para cada rama de la instrucción If:

```
FontListBox.RemoveAllRows
```

Verás este paso con más detalle en el Capítulo Seis.

Cuando esté finalizado, el evento TextChanged del SearchField debería ser:

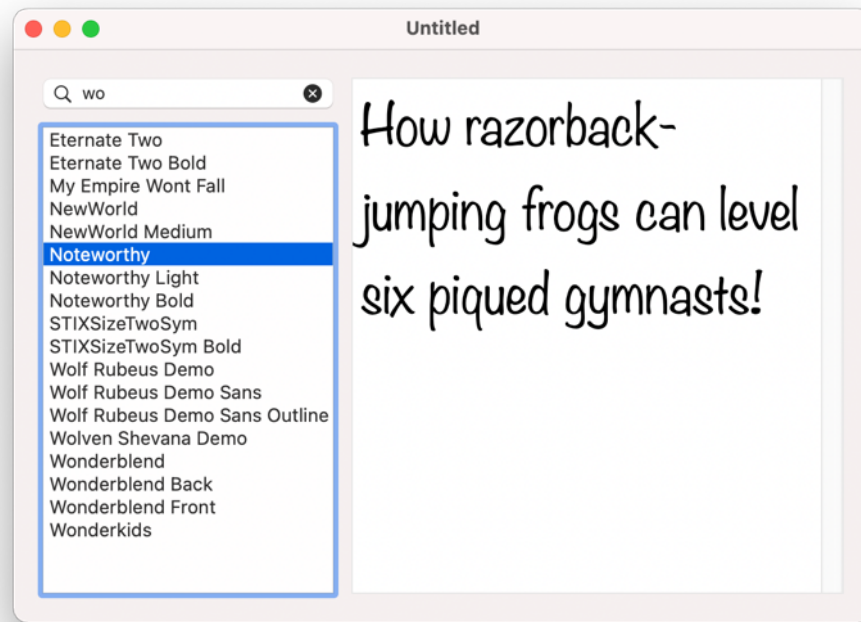
```
Var searchString, theFontName As String
Var counter, myFontCount As Integer

myFontCount = System.FontCount-1

If Me.Text <> "" Then
    searchString = Me.Value
    FontListBox.RemoveAllRows
    For counter = 0 To myFontCount
        theFontName = System.FontAt(counter)
        If theFontName.IndexOf(searchString) >= 0 Then
            FontListBox.AddRow(theFontName)
        End If
    Next
Else
    FontListBox.RemoveAllRows
    For counter = 0 To myFontCount
        FontListBox.AddRow(System.FontAt(counter))
    Next
End If
```

13) Ejecuta tu proyecto.

Deberías ver una aplicación similar a esta:



Al elegir cualquier fuente del listado provocará que el PreviewField se actualice con dicha fuente. Al teclear texto en SearchField debería causar que FontListBox muestre sólo las fuentes coincidentes. Como extra, puedes cambiar el texto usado en la previsualización por el que quieras.

14) Sal de la aplicación.

En este capítulo has aprendido varias formas de reaccionar a diferentes condiciones en tu código. Dado que es prácticamente imposible predecir cada condición en tu aplicación, este aprendizaje es muy útil en las situaciones de programación del día a día.

Hacer las Cosas

CONTENIDOS

1. **Introducción al Capítulo**
2. **Métodos Sencillos**
3. **Parámetros**
4. **Valores por Defecto**
5. **Comentarios**
6. **Funciones y Valores Devueltos**



4.1 Introducción al Capítulo

Hasta ahora has introducido código en eventos concretos para asegurarte de que ocurren ciertas cosas en determinados momentos. Este es un modo común de programar, pero puede derivar en código duplicado. Por ejemplo, en tu Font Previewer tenías unas cuantas líneas de código cuya función era la de rellenar el FontListBox con todas las fuentes del ordenador. Esta es parte del código en el evento Opening de FontListBox:

```
For counter = 0 To myFontCount - 1
    Me.AddRow(System.FontAt(counter))
Next
```

Tenías código muy similar en el evento TextChanged del SearchField:

```
For counter = 0 To myFontCount - 1
    FontListBox.AddRow(System.FontAt(counter))
Next
```

El código no es idéntico, pero casi. De hecho, si sustituyeses Me.AddRow por FontListBox.AddRow en tu instrucción If, el código sería totalmente idéntico y funcionaría igual.

Esto viola una de las reglas de la programación: *No te repitas*.



Para ilustrarlo, veamos un ejemplo del mundo real. Imagina que te pido que hagas espaguetis. Si no sabes como, puedo indicártelo (en términos generales):

- 1) **Hierve agua**
- 2) **Cocina los espagueti**
- 3) **Calienta salsa de espagueti**
- 4) **Escurre los espagueti**
- 5) **Añade la salsa**
- 6) **Adereza con queso mozzarella (opcional)**

Si te preguntase nuevamente la próxima semana por los espagueti, ya sabrías como hacerlos... ya sea de memoria o

porque hubieses escrito los pasos. En otras palabras, tendrías un *método* para preparar los espagueti.

En programación te encuentras frecuentemente con métodos para realizar ciertas tareas, especialmente aquellas tareas que puedan realizarse múltiples veces. Si defines un método para cocinar espaguetis, esto te ahorraría tiempo posteriormente; en vez de tener que volver a enumerar los seis pasos sólo tendrías que escribir:

```
CookSpaghetti
```

En este capítulo aprenderás sobre los métodos y las funciones, y realizarás algunos cambios a tu proyecto Font Previewer para limpiar tu código con el uso de métodos y funciones.

4.2 Métodos sencillos

Como se ha indicado anteriormente, un método es simplemente un conjunto de pasos para realizar una tarea. Abre el proyecto Font Previewer en Xojo y abre Window1. En el menú Insert, selecciona Method. Xojo te mostrará cuatro campos en el Inspector para que los rellenes.

El primero es Method Name. Nombra tu método FillFontListBox. Por ahora, deja en blanco Parameters y Return Type, mientras que Scope (ámbito) debería ser Public (Público). Introduce el siguiente código en el Editor de Código, en el centro de la ventana del espacio de trabajo.

```
Var counter, myFontCount As Integer
myFontCount = System.FontCount-1

FontListBox.RemoveAllRows
For counter = 0 To MyFontCount
    FontListBox.AddRow(System.FontAt(counter))
Next
```

Este código debería de resultarte familiar; es el código utilizado para listar las fuentes en FontListBox. Ahora, ve al evento Opening de FontListBox. Borra todo el código y escribe eso:

```
FillFontListBox
```

Observa cómo el autocompletado de Xojo ya conoce el método FillFontListBox y te ofrecerá completarlo.

Ejecuta el proyecto. Su comportamiento debería ser idéntico, dado que cuando el ordenador llega a la línea de código que dice “FillFontListBox”, irá a tu método y ejecutará cada línea.

Sal de la aplicación y dirígete al evento TextChanged del SearchField. También puedes abreviar el código utilizando el método. Cambiar tu instrucción If para que sea así:

```
If Me.Text <> "" Then
    searchString = Me.Text
    FontListBox.RemoveAllRows
    For counter = 0 To myFontCount - 1
        theFontName = System.FontAt(counter)
        If theFontName.IndexOf(searchString)>= 0 Then
            FontListBox.AddRow(theFontName)
        End If
    Next
Else
    FillFontListBox
End If
```

Ejecuta de nuevo el proyecto y, de nuevo, debería ofrecer el mismo comportamiento. Sal de la aplicación.

Si observas el código en el evento TextChanged de SearchField aún verás algunas similitudes aquí. El código para rellenar FontListBox con las fuentes coincidentes no es idéntico al otro código, pero ciertamente muy similar.

4.3 Parámetros

Ya hemos hablado sobre los espagueti. Algunas personas, pero no todas, prefieren que los espagueti tengan albóndigas. Tu método teórico CookSpaghetti puede manejar los espagueti, pero ¿cómo le indicas las albóndigas y sólo en ciertas ocasiones?



Los métodos pueden tomar parámetros. Un parámetro es un dato que das a un método; el método puede hacer algo directamente sobre dicho dato o bien usarlo para determinar cómo ha de funcionar. Tu método CookSpaghetti puede tomar como parámetro un booleano llamado AddMeatballs. Si

AddMeatballs es cierto, CookSpaghetti debería mezclar carne, especias y pan, y añadir luego el resultado a la comida.

Para ver un ejemplo más concreto, vuelve al proyecto Font Previewer.

Tal y como se ha mencionado, en el evento TextChanged del SearchField tienes código que no se duplica en otro código, pero casi. Si no tienes una searchString, el código hace esto:

```
For counter = 0 To myFontCount
    FontListBox.AddRow(System.FontAt(counter))
Next
```

Si has de hallar una coincidencia con searchString, haces esto:

```
FontListBox.RemoveAllRows
For counter = 0 To myFontCount
    theFontName = System.FontAt(counter)
    If theFontName.IndexOf(searchString) >= 0 Then
        FontListBox.AddRow(theFontName)
    End If
Next
```

Tan sólo difieren dos líneas de código: la instrucción If. Vas a añadir este código al método FillFontListBox proporcionándole un parámetro.

1) **Dirígete al método FillFontListBox e introduce esto en el campo Parameters:**

```
searchString As String
```

Cada parámetro necesita tener un nombre y tipo de dato, tal y como en las variables. De hecho puedes pensar en un parámetro como una variable que puede utilizarse en el método. Si es necesario, cambia el triángulo de ampliación próximo a la línea que dice “Sub FillFontListBox” en la parte superior.

2) **Ejecuta tu proyecto.**

Esta vez no funcionará, porque tienes un error de programación. Has indicado al ordenador que FillFontListBox tiene que recibir una cadena cuando se ejecute, pero no le has dado ninguna cadena.

Proporcionar el parámetro al método cuando lo llamas se denomina “pasar” el parámetro. Has de pasar una cadena a FillFontListBox en dos lugares: en el evento Open de FontListBox y en el evento TextChanged de SearchField.

3) **En el evento Open de FontListBox, puedes pasar una cadena vacía. Para pasar un parámetro, inclúyelo inmediatamente tras el nombre del método, entre paréntesis:**

```
FillFontListBox( " " )
```

Puedes hacer esto porque no quieres encontrar el nombre de ninguna fuente. En el evento TextChanged de SearchField, las cosas son un poco más complicadas, dado que vas a mover la mayor parte de la lógica al método FillFontListBox. TextChanged de SearchField se verá así ahora:

```
FillFontListBox(Me.Text)
```

Ya no necesitas comprobar si hay una cadena vacía o modificar el comportamiento de tu código o declarar variables, dado que todo ocurre ahora en el método FillFontListBox, donde tienes que ampliarlo ahora:

```
Var theFontName As String
Var counter, myFontCount As Integer
myFontCount = System.FontCount-1

FontListBox.RemoveAllRows

If searchString <> "" Then
    For counter = 0 To MyFontCount
        theFontName = System.FontAt(counter)
        If theFontName.IndexOf(searchString) >= 0 Then
            FontListBox.AddRow(theFontName)
        End If
    Next
Else
    For counter = 0 To MyFontCount
        FontListBox.AddRow(System.FontAt(counter))
    Next
End If
```

Toda la lógica que se encontraba anteriormente en el evento TextChanged de SearchField está contenida ahora en el método FillFontListBox. Al haber cambiado así el proyecto también has eliminado todo el código duplicado y el código prácticamente duplicado.



que ejecutas un método has de asegurarte de que los parámetros se pasan en el orden correcto.

Volviendo a nuestra cena con espaguetis, es bien sabido que a muchas personas les gusta el pan de ajo con sus espaguetis, tanto si incluye albóndigas como si no. El método imaginario `CookSpaghetti` no acepta múltiples parámetros, pero no es problema. Puedes añadirle un `addMeatballs As Boolean` e `includeGarlicBread As Boolean`. Para añadir múltiples parámetros en la declaración de un método en Xojo, sólo has de introducirlos en el campo `Parameters` y separarlos con comas:

```
addMeatballs As Boolean, ↵  
    includeGarlicBread As Boolean
```

El orden de los parámetros es crítico. Imagina que fueses a un restaurante y pidieses espaguetis sin albóndigas pero con pan de ajo; y tu plato llegase con albóndigas y sin pan de ajo. Cada vez

4.4 Valores por Defecto

Amplíemos tu restaurante Italiano virtual e imaginemos un nuevo método llamado CookLasagna. Será similar al método CookSpaguetti, pero las albóndigas no serán una opción - sólo el pan de ajo. De modo que el campo Parameters será como este:

```
includeGarlicBread As Boolean
```

Supongamos ahora que el chef ha insistido en que comensal debería de recibir pan de ajo salvo que indique lo contrario. Podrías cambiar el campo Parameters por esto:

```
includeGarlicBread As Boolean = True
```

Esta línea de código no sólo describe el parámetro, sino que le proporciona un valor por defecto. Por supuesto, esto no está limitado a valores booleanos:

```
myLastName As String = "Smith"
```

Definir un valor por defecto para un parámetro te permite omitir dicho parámetro cuando llamas al método. Vuelve a tu proyecto Font Previewer. Recuerda que en el evento Opening de FontListBox llamabas al método FillFontListBox pasando una cadena vacía:

```
FillFontListBox("")
```

Dirígete al método FillFontListBox y cambia el campo Parameters por esto:

```
searchString As String = ""
```

Es decir comillas dobles, o cadena vacía. Esto indica al método que si no indicas el valor para searchString, este asume que no hay ninguna. Ahora puedes cambiar el evento Opening de FontListBox por:

```
FillFontListBox
```

Ejecuta el proyecto. Nuevamente, su funcionalidad es idéntica a la anterior pero has simplificado y limpiado más tu código.

4.5 Comentarios

Habrás observado que en algunos de los ejemplos usados hay líneas de código que empiezan con dos barras seguidas por una anotación. Estos son comentarios. Los comentarios suponen un modo de simplificar el código. En general, tu código debería documentarse por sí mismo; es decir, los nombres de tus métodos y variables deberían dejar claro su función para cualquiera que lea el código. Con el código que hemos visto hasta ahora, esto es relativamente fácil de lograr; pero a medida que tu código crece en complejidad, sin embargo, los comentarios pueden ser una herramienta realmente valiosa.

Un comentario es una línea de código especial que sirve simplemente como referencia para el desarrollador. Puede que necesites documentar cómo funciona un método o variable, o puede que necesites incorporar algún tipo de anotación para el futuro.

Puede añadirse un comentario de dos formas: la doble barra y la comilla simple. Estas son intercambiables y pueden añadirse en cualquier parte del código. Puedes introducir una línea completa de comentarios:

```
' This method will unleash killer bees. Do not use.
```

O puedes introducir un comentario como parte de la línea de código:

```
MyLastName = NameField.Value // NOTE: use 2 fields
```

Observa que todo lo introducido tras la marca de comentario se considera parte del comentario.

Hay ocasiones en las que resulta útil convertir líneas de código en comentarios de forma temporal. Esto es especialmente cierto cuando intentas encontrar un error. Puedes hacerlo manualmente, escribiendo la doble barra o la comilla simple delante de cada línea. Xojo incluye también una característica que comentará/descomentará varias líneas de código a la vez. Selecciona el código que quieras comentar y elige a continuación Comment en el menú Edit. Si el código ya está comentado con comillas simples, este comando lo descomentará.

4.6 Funciones y Valores Devueltos

Algunos métodos pueden devolverte valores. Este resultado se denomina valor retornado. Puede ser cualquier tipo de dato: un booleano que te indique si un método ha tenido éxito, un resultado numérico o un cálculo matemático, o bien la concatenación de texto. Un método que devuelve un valor se conoce como función.

Vuelve a tu proyecto Font Previewer.

1) **Añade un nuevo método llamado “GetSelectedFont” a Window1.**

Puedes añadir un nuevo método seleccionando Method en el menú Insert. No recibirá parámetros, pero su campo Return Type debería ser String, dado que este método te devolverá texto.

Este método comprobará si se ha seleccionado una fuente en FontListBox y, de ser así, nos devolverá su nombre.

```
Var currentFont As String
currentFont = FontListBox.SelectedRowValue
Return currentFont
```

Ahora es necesario modificar la interfaz de Font Previewer.

- 2) **Reduce la longitud de FontListBox y añade un PushButton.**
- 3) **Escribe “FontButton” como nombre del PushButton.**

Nuevamente, no te preocupes mucho sobre las posiciones exactas de tus controles; usa tu creatividad. Usa “Which Font?” para su propiedad Caption.

4) **Añade este código al evento Pressed de FontButton:**

```
Var fontName As String
fontName = GetSelectedFont
MessageBox("You have selected " + fontName)
```

- 5) **Ejecuta tu proyecto y selecciona una fuente.**
- 6) **Haz clic en FontButton y debería de mostrarse un mensaje.**
- 7) **Sal de la aplicación.**

Ahora es momento de simplificar el código en el evento Pressed de FontButton. Has accedido al valor devuelto por el método GetSelectedFont usando esta línea de código:

```
fontName = GetSelectedFont
```

Aunque está perfectamente bien, es un paso innecesario. Tan pronto como se ejecute tu método, tendrás el valor devuelto y puedes usarlo en tu código. Casi podrías tratar el método como si fuese una variable. Borra todo el código del evento Pressed en FontButton y escribe esto:

```
MessageBox("You have selected " + GetSelectedFont)
```

Ejecuta de nuevo el proyecto y deberías obtener el mismo comportamiento pero, una vez más, con un código más sencillo.

Crear una Lista

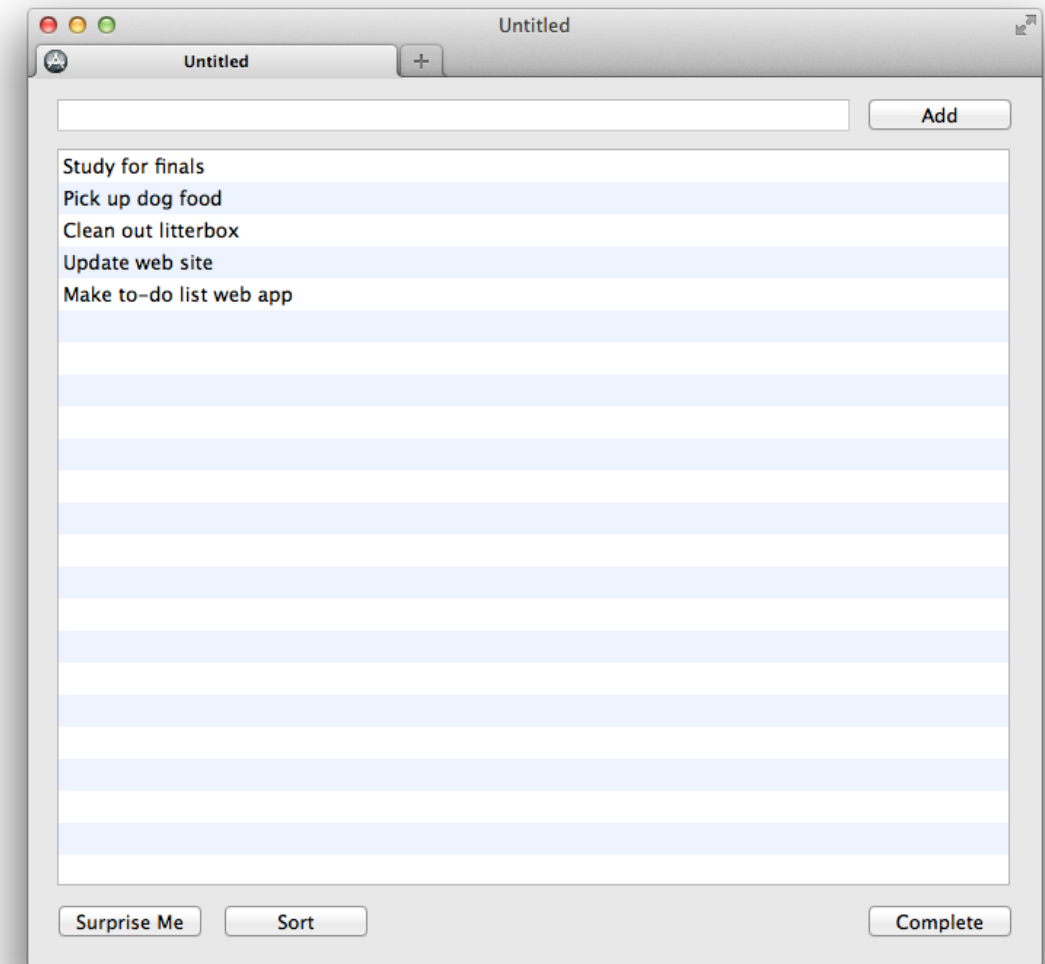
CONTENIDOS

1. **Introducción al Capítulo**
2. **Añadir a la Lista**
3. **Limpiado**
4. **Gestionar el Orden**
5. **Convertir texto en una Lista**
6. **Usar pares Clave/Valor para guardar datos**
7. **Arrays en la práctica**

5.1 Introducción al Capítulo

Un problema que te sueles encontrar con frecuencia en programación es el de mantener una lista de elementos. Estos elementos pueden ser texto, números, colores, fechas o cualquier tipo de datos que puedas imaginar. En Xojo, y en la mayoría del resto de los lenguajes de programación, esto se realiza utilizando un array. Un array es simplemente una lista numerada de elementos.

En este capítulo usarás arrays para incrementar tu conocimiento sobre los capítulos anteriores a la hora de crear un gestor de tareas. Una gran diferencia es que esta app será una app web. Esto es lo que crearás (esta copia de la app está funcionando en Firefox en un Mac, pero funcionará en cualquier navegador moderno en cualquier sistema operativo):



5.2 Añadir a la Lista

Crear un array es muy similar a crear cualquier otro tipo de variable. Sólo hay una cosa que debes añadir cuando declares la variable. Recuerda que puedes crear una cadena utilizando esta sintaxis (y recuerda del Capítulo 2 que una String es simplemente un fragmento de texto):

```
Var myString As String
```

Si quisieras un array de strings, sólo tendrías que añadir unos paréntesis:

```
Var myStringList() As String
```

myStringList es ahora una lista de cadenas. En este ejemplo, tu lista no contiene entradas. Puedes añadirle entradas, pero si quisieras un array con una cantidad concreta de entradas, podrías indicar dicho número entre los paréntesis:

```
Var myStringList(10) As String
```

En muchas ocasiones no sabrás cuantas filas hay en un array. Puedes preguntar a Xojo cuantas filas hay en un array utilizando la función Count del array:

```
myArrayCount = myStringList.Count
```

Como se mencionó anteriormente, myStringList no es una cadena; es una lista de cadenas. Dado que no es una cadena, no puedes utilizarlo del mismo modo que has estado utilizando las string hasta ahora. Por ejemplo, puedes mostrar un mensaje al usuario usando:

```
MessageBox(myString)
```



Pero no puedes mostrar un array al usuario del mismo modo. Sin embargo puedes mostrar una de las cadenas de la lista. Si quisieras mostrar el primer elemento en tu lista de cadenas, podrías utilizar el siguiente código:

```
MessageBox(myStringList(0))
```

Observa que la primera entrada en el array es el item con el índice cero. Hay motivos históricos por lo que esto es así, pero conviene tenerlo en cuenta, dado que uno de los errores más comunes a la hora de utilizar los array es comenzar con el número uno en vez de con el número cero. Esto también significa que el último elemento en el array (el que tiene el índice más alto) es uno menos que el valor devuelto por Count. Si un array tiene diez elementos, el índice más alto de una entrada es nueve. Afortunadamente, los array también tienen la función LastRowIndex que devuelve este valor.

De modo que puedes tratar myStringList(0) en el mismo modo que tratarías cualquier otra cadena. Si quieres asignar un valor a dicha posición, puedes hacerlo así:

```
myStringList(0) = "Just your average string here"
```

Y puedes hacer lo mismo con myStringList(1), myStringList(2), etc. Cada entrada en la lista es simplemente una variable normal. En este caso un array de cadenas, cada entrada es simplemente una cadena. Si tuvieses un array de enteros, cada elemento sería un entero normal. Para declarar un array de enteros, utilizas una sintaxis similar a la anterior:

```
Var myNumberList() As Integer
```

De nuevo, esto nos proporcionará una lista vacía de enteros. Si quieres especificar la cantidad de entradas en la lista, puedes hacerlo como antes:

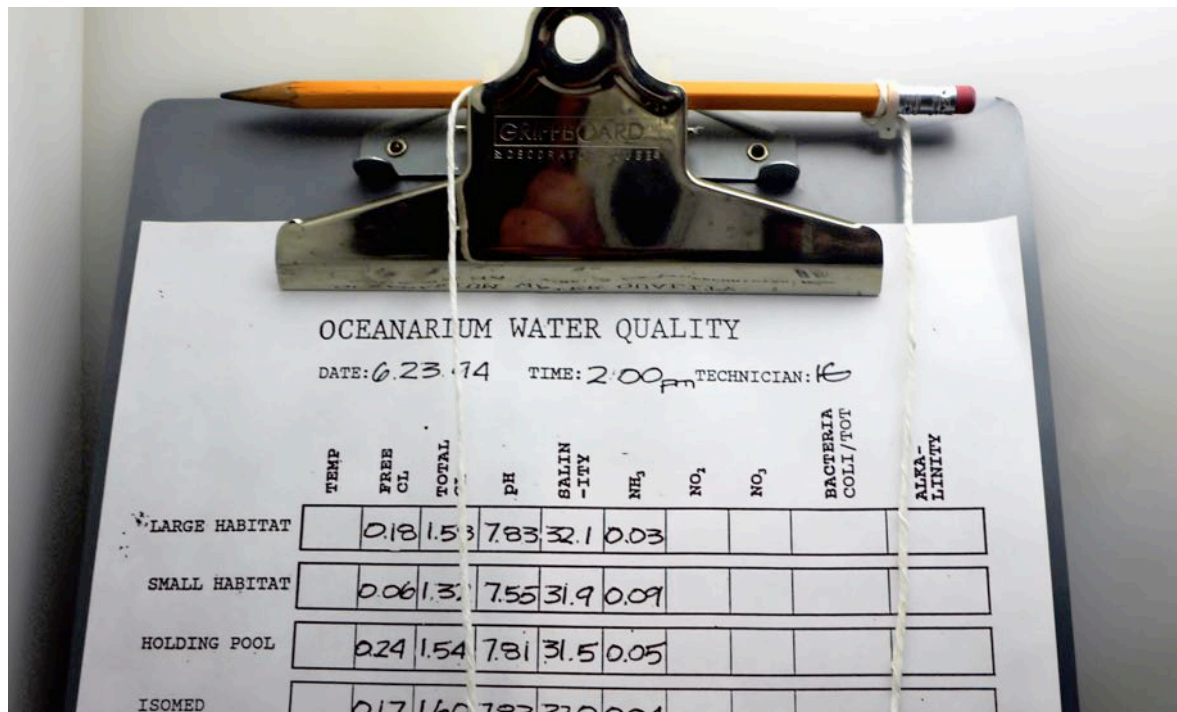
```
Var myNumberList(5) As Integer
```

Esto nos dará una lista con seis entradas. Como puedes ver, creas y tratas los arrays de la misma forma del tipo de dato que contienen. De igual modo, tratas cada entrada como una variable normal de su tipo de dato.

Crear un array es una cosa, pero un array no es de mucho valor hasta que contiene entradas. Para añadir una entrada a un array (o para añadir una entrada a una lista), puedes utilizar los métodos AddRow o AddRowAt.

Cuando añades una entrada a un array esta se añade al final del array, de modo que se incrementa en uno el número de entradas en el array (así como LastRowIndex):

```
Var myStringList(10) As String  
// myStringList.LastRowIndex es igual a 10  
myStringList.AddRow("Hey there")  
// myStringList.LastRowIndex es igual a 11
```



Pero no siempre querrás añadir una entrada al final del array. Utiliza el método `AddRowAt` si necesitas añadir una entrada en una posición concreta. Mientras que el método `AddRow` sólo requiere de un parámetro (el valor que quieres añadir al array), el método `AddRowAt` necesita dos: en primer lugar, un entero que especifica la posición en el array, y luego el valor propiamente dicho. La ubicación de una entrada en el array se conoce como su índice. Cuando se utiliza el método `AddRowAt` se incrementa en uno la cantidad de elementos en el array, así como el índice de las entradas sucesivas, Se ilustra mejor con este código de ejemplo:

```
Var stooges() As String
// stooges.LastRowIndex es igual a -1
stooges.AddRow("Larry")
```

```
stooges.AddRow("Curly")
stooges.AddRow("Moe")
// stooges.LastRowIndex is ahora igual a 2
```

el array `stooges` es ahora como este:

Índice	Valor
0	Larry
1	Curly
2	Moe

Si ejecutas la siguiente línea de código:

```
stooges.AddRowAt(1, "Shemp")
```

`stooges.LastRowIndex` es igual a tres, y el array será como sigue:

Índice	Valor
0	Larry
1	Shemp
2	Curly
3	Moe

5.3 Limpiado

Como has visto en el ejemplo anterior, el método `AddRowAt` “desplazará” las otras entradas en el array. Otro método que desplaza las entradas en un array es `RemoveRowAt`. Este método, como implica su nombre, eliminará una entrada del array, reducirá el número de entradas en el array en uno, y también reducirá el índice de las entradas restantes. Continuando con el ejemplo anterior:

```
stooges.RemoveRowAt(2)
```

El valor de `stooges.LastRowIndex` es ahora igual a dos, y el array será el siguiente:

Índice	Valor
0	Larry
1	Shemp
2	Moe

Eliminar una entrada cada vez es útil, pero en muchas ocasiones necesitarás borrar todas las entradas. Si bien podrías utilizar un `For...Loop` para ello, esto no sería eficiente y podría dar lugar a errores. Por fortuna, los array en Xojo incluyen el método `RemoveAllRows`:

```
stooges.RemoveAllRows
```

El array `stooges` estaría ahora vacío y sin entradas. Su `LastRowIndex` es ahora menos uno.

La mayoría de otros lenguajes de programación se refieren a las entradas en un array como elementos. Xojo utiliza “row” porque muchas personas piensan en un array como en una lista o columna de una hoja de cálculo.

5.4 Gestionar el Orden

El orden de las entradas importa en el manejo de un array. El ordenador recordará qué elemento hay en cada posición y preservará dicha información. Por ello, cada vez que trabajes con el mismo array podrás saber que el dato que almacenaste en la décima posición, por ejemplo, permanecerá en la décima posición hasta que indiques lo contrario.

Pero te encontrarás en situaciones en las que quieras reordenar las entradas de tu array. Hay métodos de “fuerza bruta” para ello, pero Xojo incluye dos funciones de array que son muy útiles para reordenar los array.

La forma más evidente de reordenar las entradas en un array es ordenándolas. Para ello, utiliza el método Sort:

```
Var contacts() As String
contacts.AddRow("Zeke")
contacts.AddRow("Abe")
contacts.AddRow("Mary")
contacts.Sort
```

El array contacts se verá ahora así:

Índice	Valor
0	Abe
1	Mary
2	Zeke

Un array de cadenas se ordenará alfabéticamente en orden ascendente. Si tu array contiene datos numéricos entonces se ordenará numéricamente. Cuando se ordena el array, el índice de cada entrada (o posición en el array) se ajustará en consecuencia.



Otra forma de modificar el orden de los elementos en un array es haciéndolo aleatoriamente. Imagina que estás creando un juego

que incluya una baraja de 52 cartas. Como en la mayoría de los juegos de cartas, querrás barajarlas en algún momento.

Asumamos que en tu juego de cartas imaginario tienes un array de cadenas llamado `cards`, que contiene 52 elementos (de dos a diez, más la sota, la reina y el rey, así como el as para cada uno de los cuatro palos). Para barajar las cartas:

```
cards.Shuffle
```

Las cartas estarán ahora en un orden aleatorio. Eso es todo.

`Shuffle` funciona en cualquier tipo de dato y es completamente aleatorio. Si haces un `Shuffle` dos veces en el mismo array, obtendrás resultados distintos cada vez (existe una probabilidad estadística insignificante de que el ordenador produzca los mismos resultados dos veces seguidas).

`Shuffle` es útil más allá de los juegos de cartas. Supongamos que estuvieses creando una interfaz para un pasatiempos. Podrías utilizar `Shuffle` para reordenar aleatoriamente las preguntas, así como el orden de las posibles respuestas para cada pregunta.

5.5 Convertir Texto en una Lista

Hasta ahora has estado añadiendo datos a tus arrays usando el método `AddRow`, y si bien es perfectamente correcto hacerlo así, habrá ocasiones en las que necesites crear un array basado en una cadena existente. Imagina que estuvieses escribiendo una aplicación que permitiese al usuario “etiquetar” contenido mediante la introducción de palabras clave separadas por comas. Podrías usar un `TextField` llamado `TagField` para que el usuario introduzca datos, y podrías acabar con una cadena que fuese por ejemplo.

Movies, Comedy, 90s

Imagina ahora que necesitas convertir esta cadena en un array. Podrías utilizar la función `IndexOf` que aprendiste en el Capítulo Tres para encontrar todas las comas y parsear los datos tu mismo. O podrías usar la función `ToArray` incluida en `string`. `ToArray` tomará la cadena y la separará en un array basándose para ello en el delimitador proporcionado. Volviendo al ejemplo de etiquetado:

```
Var tags() As String
tags = TagField.Value.ToArray(",")
```

El array `Tags` contendría una fila por cada fragmento de la cadena que esté separado por una coma. Si el usuario ha introducido “Movies, Comedy, 90s”, como se ha indicado anteriormente, el array `Tags` tendría tres entradas:

```
// tags(0) = "Movies"
// tags(1) = "Comedy"
// tags(2) = "90s"
```

La función `String.FromArray` funciona a la inversa. `FromArray` toma un array de cadenas y un delimitador, y forma una cadena a partir de ellos:

```
Var tags() As String
Var combinedTags As String
tags.AddRow("Movies")
tags.AddRow("Comedy")
tags.AddRow("90s")
combinedTags = tags.FromArray(",")
// combinedTags es ahora "Movies,Comedy,90s"
```

Advierte en este ejemplo que no hay espacios en `combinedTags`. Si quieres tener una coma y un espacio entre cada elemento, deberías utilizar eso como tu delimitador. El delimitador, tanto en `ToArray` como en `FromArray`, puede ser cualquier string que quieras usar.

5.6 Usando Pares de Claves/Valores para Guardar Datos

Como se ha visto, los array son útiles para almacenar listas de elementos similares. Además, mantienen el orden de los elementos hasta que lo modifiques. Pero el único modo por el que puedes hallar un valor es mediante el uso del índice. Algunos lenguajes tienen una característica denominada arrays asociativos que, esencialmente, te permiten utilizar cualquier tipo de dato como un “índice” por el cual puedas referirte posteriormente a un valor. De modo que si un array convencional se ve así:

Índice	Valor
0	Abe
1	Mary
2	Zeke

Cada entrada en un array asociativo contendría una clave en vez de un índice:

Clave	Valor
0	Abe
1	Mary

Xojo no tiene arrays asociativos, pero tiene un tipo de dato denominado Diccionario que es similar a un array asociativo en muchas cosas. Puede utilizarse para guardar listas de datos, pero no se mantiene el orden de los datos. De igual modo, en vez de utilizar un índice numérico asociado a la posición de cada valor en la lista, el Diccionario utiliza una relación Clave/Valor, donde la clave (Key) y el valor (Value) pueden ser cualquier tipo de dato. El Diccionario es técnicamente una clase, de modo que ha de instanciarse antes de que pueda usarse (de forma similar a como se hace con el tipo de dato DateTime que aprendiste en el Capítulo Dos).

Para crear un Diccionario (Dictionary), utiliza la palabra clave Var tal y como en el resto de los tipos de datos:

```
Var settings As New Dictionary
```

Observa que este ejemplo utiliza el método abreviado para instanciar tu variable, incluyendo el operador New en la misma línea que la palabra clave Var.



Para añadir datos a un array utilizabas los métodos `AddRow` y `AddRowAt`. El diccionario no incluye dichos métodos. En vez de ello, defines un valor y proporcionas una clave:

```
Var settings As New Dictionary
settings.Value("Name") = "Anakin Skywalker"
settings.Value("Title") = "Sith Lord"
settings.Value("Dark Side") = True
settings.Value("Age") = 39
settings.Value("Trainer") = "Kenobi"
```

Si echas un vistazo al código, observarás que has utilizado valores de cadena, enteros y booleanos como Valores en tu Diccionario. Esto es perfectamente válido; el diccionario puede utilizar cualquier tipo de dato como valor (Value) o como clave

(Key), a diferencia de lo que ocurre con los array donde cada elemento ha de ser del mismo tipo de dato declarado.

Aunque puede que no resulte evidente a partir del ejemplo, también has proporcionado claves. Observa esta línea de código del ejemplo:

```
settings.Value("Dark Side") = True
```

La clave en esta línea es “Dark Side” y el Valor es un booleano, en concreto True en este ejemplo. De modo que esencialmente estás almacenando datos en gran medida como en los array, pero asignando manualmente una clave a ellos en vez de utilizar un índice numérico.

¿Por qué es útil? Supongamos que quieres recuperar información del Diccionario, como el Valor almacenado con la clave “Title”.

Siguiendo con el código:

```
Var theTitle As String
theTitle = settings.Value("Title")
// theTitle = "Sith Lord"
MessageBox(theTitle)
```

Dado que ahora puedes asignar cualquier cosa para la Clave, un error común es un `KeyNotFoundException`, el cual tiene lugar cuando intentas recuperar un valor para una clave que no existe. Para evitar este error, nombra tus claves con cuidado y de forma lógica para que puedas recordarlas.

Si te encuentras en una situación en la que sabes cuál debería ser la clave, pero no estás seguro de si existe o no, puedes utilizar la función Lookup. La función Lookup toma dos parámetros: el primero es la clave que crees que es, y la segunda es el valor por defecto que se usa en el caso de que no se encuentre la clave. En tu anterior diccionario, no has definido un valor para la clave “Lightsaber Color”, pero puedes intentar acceder de todas formas:

```
MessageBox(settings.Lookup(
    "Lightsaber Color", "Red"))
```

Dado que la clave “Lightsaber Color” no existe, el mensaje mostrará en este caso “Red.” Si usas una clave existente:

```
MessageBox(settings.Lookup("Title", "Pilot"))
```

... entonces verás el valor del diccionario como esperas.

Para comprobar la existencia de una clave concreta, utiliza la función HasKey. HasKey toma la clave que estás buscando como parámetro y devuelve un booleano: True si la clave existe y False si no:

```
If settings.HasKey("Mentor") Then
    MessageBox("Mentor: " + settings.Value("Mentor"))
Else
    MessageBox("No mentor was found")
```

End If



Para eliminar una entrada del Diccionario, utiliza el método Remove pasando la Clave como parámetro:

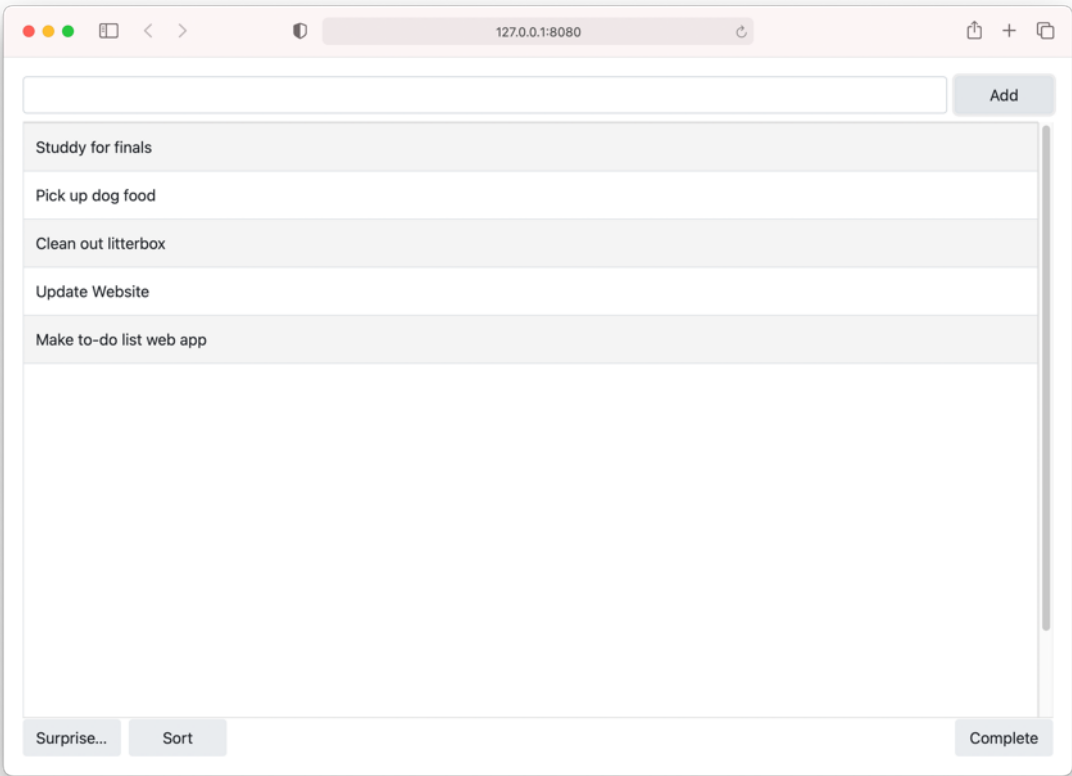
```
settings.Remove("Dark Side")
```

Para eliminar todas las entradas del Diccionario a la vez, utiliza el método RemoveAll:

```
settings.RemoveAll
```

5.7 Manos a la obra con los Array

Como se ha mencionado al inicio del capítulo, tu próximo proyecto de ejemplo es un gestor de tareas web. La aplicación web final será como esta:



- 1) Si aún no lo has hecho, ejecuta Xojo y crea una nueva aplicación Web. Guárdala como “ToDoList”.
- Para empezar, crearás la interfaz, añadiendo posteriormente el código. La siguiente tabla enumera los tipos y nombres de los controles que

necesitarás. No te preocupes mucho sobre la posición exacta de los controles; utiliza la captura de pantalla como referencia pero siéntete libre de emplear tu creatividad.

Control	Nombre	Etiqueta (Caption)
WebTextField	ToDoField	N/A
WebButton	AddButton	Add
WebListBox	ToDoListBox	N/A
WebButton	ShuffleButton	Surprise Me
WebButton	SortButton	Sort
WebButton	CompleteButton	Complete

El usuario introducirá tareas (To Dos) en el ToDoField, luego hará clic en AddButton para añadirlas a TodoListBox. Los datos de TodoListBox se guardarán en un array.

Dado que TodoListBox no necesita una cabecera de columna puedes desactivarla conmutando a OFF la propiedad HasHeading en el Inspector.

2) Bloquea los controles como se indica a continuación.

Si has cambiado el tamaño de las aplicaciones de este libro durante su funcionamiento hasta ahora, probablemente habrás observado que los controles no se han comportado según lo esperado; en vez de ampliarse para completar el espacio disponible en la ventana, estos habrán permanecido en su posición y con el mismo tamaño siempre respecto al margen izquierdo y superior de la ventana. Esto puede solucionarse cambiando el anclaje del control en el Inspector. Puedes bloquear o anclar cualquier margen de un control. Cuando dicho margen está bloqueado, este mantendrá la distancia sobre ese mismo margen de la ventana. Por lo tanto, un control con el margen superior e izquierdo bloqueados

permanecerán en la misma posición, mientras que un control con los márgenes superior y derecho bloqueados siempre estarán en la esquina superior derecha de la ventana. Tu app de tareas debería rellenar la ventana del navegador, de modo que bloquee los controles como se indica en esta tabla:

Control	Bloqueo
ToDoField	Izquierda, Arriba, Derecha
AddButton	Arriba, Derecha
ToDoListBox	Izquierda, Arriba, Derecha, Abajo
ShuffleButton	Izquierda, Abajo
SortButton	Izquierda, Abajo
CompleteButton	Derecha, Abajo

Para bloquear un control sólo has de hacer clic sobre el candado en la sección “Locking” del Inspector. Un candado cerrado indica que ese margen del control está bloqueado, mientras que un candado abierto indica lo contrario, naturalmente.

3) **Añade una propiedad a WebPage1. Su tipo debería ser String y su nombre debería ser “ToDoList()”.**

En el último capítulo aprendiste algo sobre el ámbito (scope), o cuando puede accederse a una variable. Dado que cada control de la ventana necesitará acceder a tu array de tareas, no puedes declararlo simplemente en un método. Necesitas que sea una propiedad de la página web. Añadir una propiedad a la página web (o a una ventana en una app Desktop) significa que es accesible para cualquier código en dicha página web, tanto si es parte de un método creado o en el evento de un control.

4) **Añade un método llamado “UpdateToDoListBox” a WebPage1.**

Cada vez que se hace un cambio en un array necesitas actualizar los datos mostrados en ToDoListBox. Dado que tendrás que hacerlo desde varias ubicaciones en tu código, crearás un método para ello. Selecciona Method en el menú Insert. Nombra el método como UpdateToDoListBox.

5) **Añade este código al método UpdateToDoListBox:**

```
ToDoListBox.RemoveAllRows
For Each row As String in ToDoList
    ToDoListBox.AddRow(row)
Next
```

UpdateToDoListBox se encargará de asegurarse de que lo mostrado en ToDoListBox se corresponda con lo que hay en ToDoList. Borrará cualquier dato existente de ToDoListBox, añadiendo entonces una fila por cada entrada en el array ToDoList. En este método estamos usando un tipo diferente para el bucle For llamado bucle For Each. Defines una variable (row en este caso) y el bucle For Each carga cada valor del array ToDoList en la variable row. Como puedes ver, simplifica en gran medida recorrer un array. No necesita una variable de contador o utilizar un índice.

6) **Añade este código al evento Pressed de AddButton:**

```
ToDoList.AddRow(ToDoField.Text)
UpdateToDoListBox
ToDoField.Text = ""
```

Este código proporciona un modo de que el usuario obtenga datos en el array. Los datos provienen de ToDoField, y ToDoButton se encargará de añadirlo al array. Entonces ejecutará el método UpdateToDoListBox. Por

último, borrará los contenidos de ToDoField de modo que el usuario no introduzca la misma tarea múltiples veces por accidente.

Como se indicaba, la primera línea añade el texto del usuario al array. La segunda línea actualiza la pantalla; y la tercera línea borra cualquier texto existente en ToDoField. Si ejecutas ahora tu proyecto, deberías de poder añadir entradas en la lista.

7) **Añade este código en el evento Pressed de CompleteButton:**

```
If ToDoListBox.SelectedRowIndex <> -1 Then  
    ToDoList.Remove(ToDoListBox.SelectedRowIndex)  
    UpdateToDoListBox  
End If
```

Cualquier gestor de tareas debería de permitir que marques elementos como completados. Este será el trabajo del botón CompleteButton. Su tarea será más compleja de lo que pueda parecer. En primer lugar, necesitará determinar si una elemento está seleccionado. Si lo está, tendrá que determinar su posición en el array y borrarlo. Por último actualizará la pantalla. Puedes recordar de un capítulo anterior que puedes ver la fila seleccionada de un ListBox con la propiedad SelectedRowIndex. Si el valor de dicha propiedad es menos uno, entonces no hay nada seleccionado. De lo contrario SelectedRowIndex será el número de la fila que está seleccionada (al igual que en los array, los ListBox se basan en índice cero, de modo que la primera fila tiene el índice cero).

8) **Añade este código al evento Pressed de SortButton:**

```
ToDoList.Sort  
UpdateToDoListBox
```

9) **Añade este código al evento Pressed de ShuffleButton:**

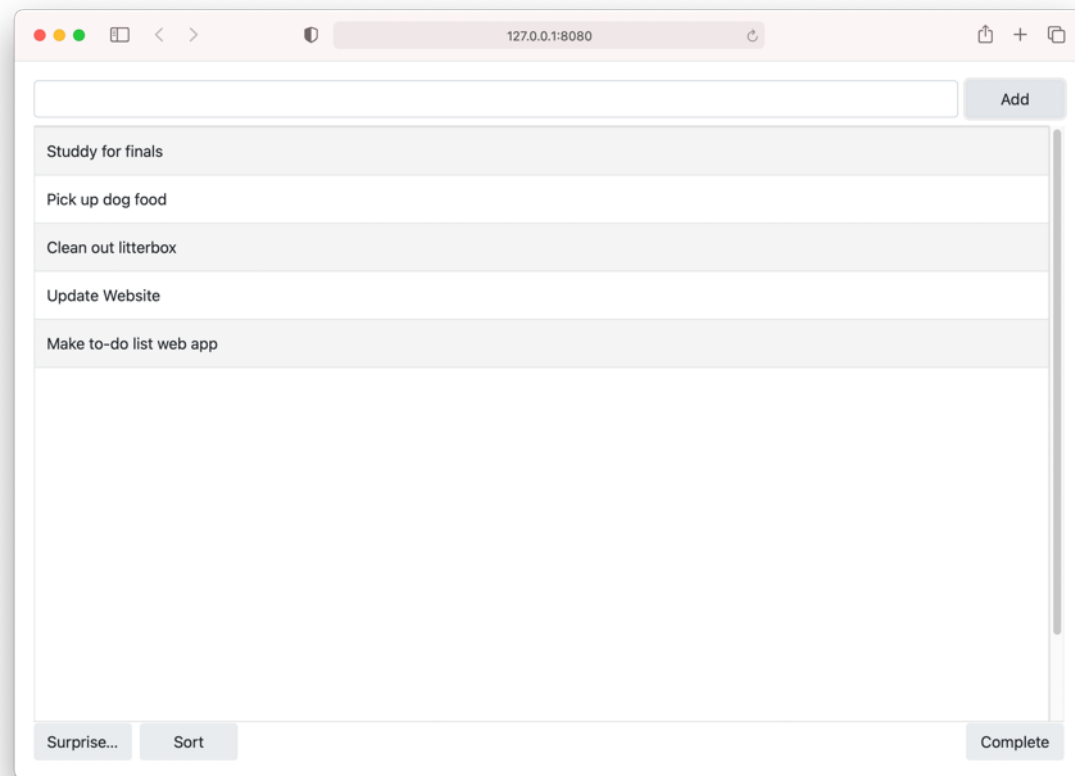
```
ToDoList.Shuffle  
UpdateToDoListBox
```

Por último, querrás permitir que tus usuarios ordenen y muevan aleatoriamente sus tareas. Los anteriores dos pasos proporcionan dicha funcionalidad.

En realidad, ordenar aleatoriamente tus tareas probablemente no tenga un mucho valor real, pero es simplemente un ejemplo que se puede aplicar a otras apps y conceptos.

10) **Guarda y ejecuta tu proyecto.**

Tu app se abrirá en el navegador por defecto de tu ordenador. Ahora deberías de poder añadir y eliminar ítems, así como ordenarlos aleatoriamente o bien alfabéticamente.



11) Sal de la aplicación.

¿Puedo Tomar su Pedido?

CONTENIDOS

1. **Introducción al Capítulo**
2. **Introducción a los Eventos**
3. **Ventanas**
4. **Entrada**
5. **Botones**
6. **Selectores**
7. **Controles en la Práctica**



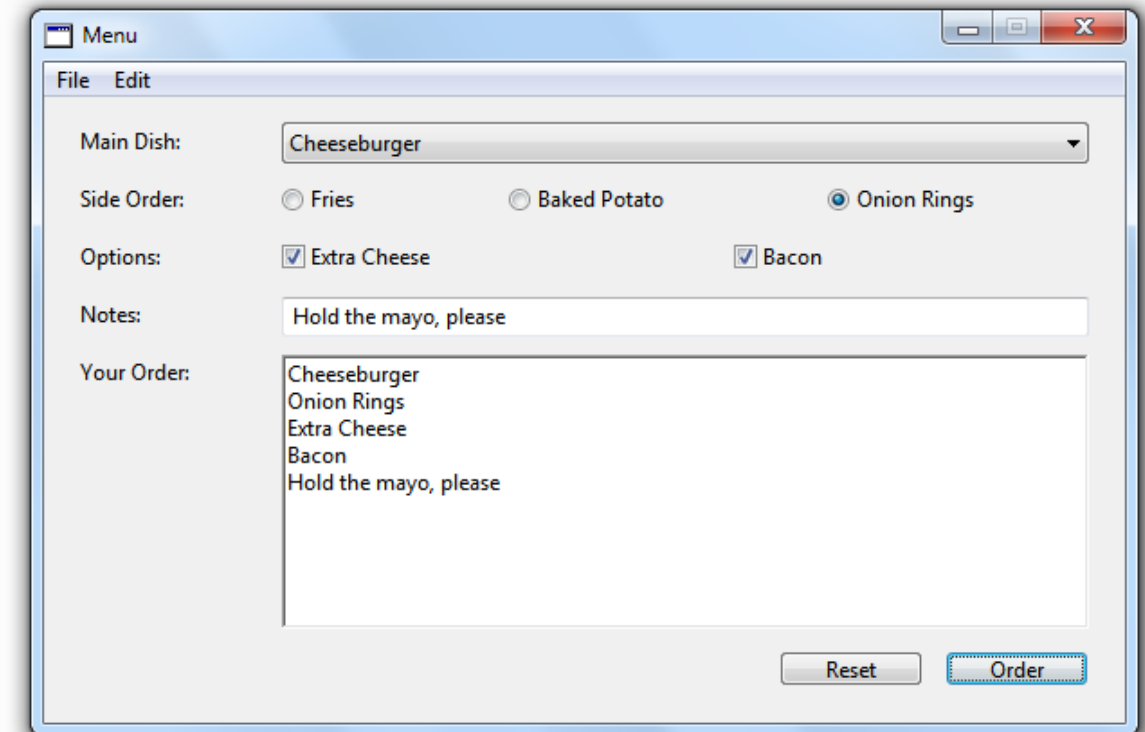
6.1 Introducción al Capítulo

En este capítulo y el siguiente aprenderás sobre dos aspectos críticos de Xojo: eventos y controles. Los eventos y los controles te permiten responder a las acciones de los usuarios de forma significativa.

Los eventos son eso: cosas que ocurren. Por ejemplo, puede dispararse un evento porque el usuario haya hecho clic en un botón o tecleado en un campo. O puede ser una acción iniciada por el ordenador. En este capítulo aprenderás algunos de los eventos usados con más frecuencia.

Un control, como se ha visto en anteriores capítulos, es un elemento de interfaz de usuario como por ejemplo un botón, un campo de texto o un menú desplegable. Aprenderás sobre diferentes controles en este capítulo y el siguiente, pero dejaremos algunos sin tratar para una posterior revisión (como por ejemplo los controles que son específicos a una plataforma, como en Windows o macOS).

Tu proyecto de ejemplo en este capítulo será un menú de comida electrónico.



6.2 Introducción a los Eventos

Como se ha mencionado anteriormente, un evento es algo que ocurre. Algunas veces se dispara un evento cuando el usuario hace algo, como hacer clic en un botón o seleccionar un elemento de menú. Otras veces, se dispara un evento por el ordenador o por la app, como cuando se ejecuta una aplicación o se abre una ventana.

Al igual que ocurre con los métodos y funciones, los eventos nos proporcionan una oportunidad de determinar cómo ha de comportarse la app. Tal y como ya has introducido código en tus métodos y funciones, también puedes escribir código en un evento. La diferencia es que has de indicar al ordenador cuando ha de ejecutar un método o función; mientras que con los eventos indicas al ordenador que se ejecute un determinado código cuando ocurra algo.

Por ejemplo, muchos controles tienen un evento llamado `Opening`. Este evento se dispara cuando se crea un control en una ventana (o en términos técnicos, se instancia). Puede que tengas un menú desplegable que necesite un determinado conjunto de ítems como opciones. Puedes definir dichos elementos en el evento `Opening` del menú desplegable.

Tu menú desplegable también tiene un evento `SelectionChanged` que se dispara cuando el usuario hace una selección. Puedes utilizar el evento `SelectionChanged` para responder a la selección del usuario.

Los eventos no están limitados a los controles. Tus ventanas también tienen eventos. ¡Incluso tu aplicación tiene eventos!

6.3 Ventanas



Se trata del elemento de interfaz de usuario en tu pantalla que contiene otros controles.

- 1) **Ejecuta Xojo y crea una nueva aplicación desktop. Guárdala como “Events”.**
- 2) **Añade un ListBox a Window1.**

Siéntete libre de usar cualquier tamaño, pero asegúrate de que sea lo suficientemente ancha como para mostrar varias palabras y lo suficientemente alta como para mostrar varias líneas a la vez.

- 3) **Añade este código al evento Opening de Window1:**

```
Listbox1.AddRow("Opening")
```

Una vez que tu ListBox esté en posición, haz clic sobre el fondo de la ventana y dirígete al menú Insert para seleccionar Event Handler (es importante asegurarse de que está seleccionada la ventana, dado que Xojo añadirá el manejador de evento sobre el control seleccionado). Aparecerá un listado con eventos. Selecciona el evento Opening y pulsa el botón OK. Te llevará al evento Opening de Window1.

- 4) **Repite el Paso 3 para los eventos Activate, Deactivate, Moved y MouseDown en Window1.**

No añadas la palabra “Opening” al ListBox, utiliza cada nombre de evento.

- 5) **Ejecuta tu proyecto.**
- 6) **Hac clic fuera de tu aplicación y vuelve luego a ella. Arrastra la ventana. Haz clic en la ventana, fuera del ListBox.**

Deberías de ver los eventos añadidos al ListBox.

- 7) **Sal de la aplicación.**

Esto debería de proporcionarte una idea sobre como tienen lugar los eventos. Cuando mueves la ventana, cambias entre aplicaciones o haces clic en la ventana, se disparan los eventos implementados. En realidad, los eventos siempre se disparan... depende de ti que quieras responder a ellos.

Otro evento útil es el evento Closing. Como está implícito en su nombre, este se dispara cuando la ventana se va a cerrar. Es un buen evento para guardar cosas como la posición de la ventana de modo que puedas presentarla en el mismo lugar la próxima vez que se abra.

Las ventanas tienen otros eventos que no se cubren en este libro. Te animo a que los explores por ti mismo. El método `ListBox` es una forma excelente de aprender qué dispara los diferentes eventos.

Xojo no es el único lenguaje que utiliza eventos; otros lenguajes también los usan, como por ejemplo Swift, JavaScript, Java y C++.

Además de los eventos, las ventanas también tienen diferentes propiedades que pueden ser definidas. Muchas de estas propiedades pueden definirse usando el Inspector. Dos propiedades importantes de la ventana que se suelen confundir con frecuencia son `Name` y `Title`. El nombre de la ventana es como te refieres a la ventana en tu código. El nombre por defecto de una ventana es `Window1`, pero puedes cambiarlo por cualquier otro que quieras; siempre es buena idea utilizar nombres significativos, uno que refleje su propósito, como `EditingWindow` o `PreferencesWindow`. El Título de la ventana, por otra parte, es el texto que aparecerá en la parte superior de la ventana cuando se ejecute la aplicación. También se puede definir el título de una ventana desde código en el caso de que necesites cambiarlo mientras que se está ejecutando la aplicación.



Una ventana también tiene varias propiedades relacionadas con su tamaño. Las más evidentes son su altura y ancho, expresadas en puntos. Es importante recordar cuando diseñes tu aplicación que las pantallas tienen diferentes tamaños. Si tienes una pantalla muy grande, recuerda que necesitas diseñar para una pantalla más pequeña. Por ejemplo, tu pantalla puede tener una resolución de 1920 x 1080 (común en las pantallas de 24 pulgadas), pero las pantallas de muchos usuarios pueden tener una resolución inferior a esta. En general, es buena idea asumir que las ventanas de tu aplicación necesitan caber en una pantalla con una resolución de 1024 por 768.

Eso no significa que los usuarios con pantallas más grandes no puedan cambiar el tamaño de la ventana para aprovechar mejor la resolución de sus monitores. Aquí es cuando entran en juego

dos propiedades más de la ventana: `MaxWidth` y `MaxHeight`. Estos números indican el máximo tamaño permitido para la ventana. El valor por defecto para ambas propiedades es de 32.000 puntos, lo que es suficientemente grande como para que no suponga una limitación. Si necesitas definir un tamaño máximo más bajo, utiliza estas propiedades para ello.

Por otro lado también puedes definir un tamaño mínimo para tu ventana usando para ello las propiedades `MinimumWidth` y `MinimumHeight`. Estas propiedades determinan cuan pequeña puede ser la ventana. Por ejemplo, es probable que tengas ciertos elementos de interfaz de usuario que requieran una determinada cantidad de espacio. Usando `MinimumWidth` y `MinimumHeight` garantizarás que el usuario no pueda ajustar el tamaño por debajo de dichos valores.

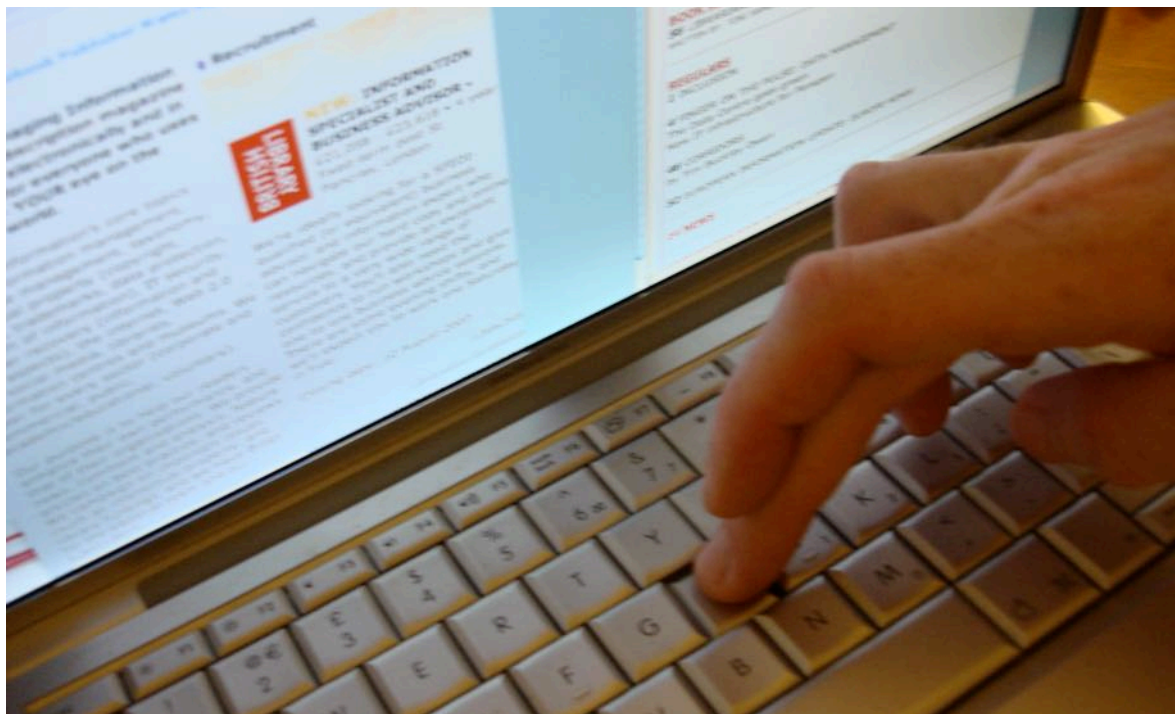
Los anchos y alturas máximo y mínimo garantizan que el tamaño de tu ventana se pueda ajustar según lo esperado. Para que el usuario pueda cambiar su tamaño puedes definir la propiedad `Resizable`. Si `Resizable` es `True`, entonces el usuario podría reducir o ampliar la ventana utilizando las capacidades nativas del sistema operativo. Si es `False`, el usuario no podrá hacerlo (aunque podrás seguir ajustando el ancho y altura mediante código cuando sea necesario).

En la mayoría de los sistemas operativos modernos, las ventanas tienen tres botones en las ventanas propiamente dichas: cerrar, minimizar y maximizar (a menudo llamado zoom). Puedes decidir

si quieres que estos botones estén activos o no ajustando las propiedades `HasCloseButton`, `HasMinimizeButton` y `HasMaximizeButton`. En función del sistema operativo sobre el que se esté ejecutando tu app, puede que los botones sigan apareciendo pero desactivados.

6.4 Entrada

En anteriores capítulos utilizaste un TextField para obtener información en tu aplicación. El TextField es uno de los controles de entrada que puedes usar en tus proyectos, siendo otros el TextArea, PasswordField y ComboBox. En esta sección aprenderás sobre los eventos y propiedades para cada uno de estos controles.



Pero en primer lugar, todos estos controles tienen algunas propiedades y eventos en común. Por ejemplo, los cuatro controles de entrada tienen propiedades para definir su tamaño y posición: Left, Top, Width y Height. Estos se miden en puntos.

Observa que las propiedades Left y Top son relativas a la ventana que contiene el control y no la pantalla propiamente dicha.

De igual modo, estos cuatro controles tienen una propiedad Name. Al igual que la propiedad Name de la ventana, este es el nombre por el que te referirás al control en tu código. Nuevamente, es buena práctica proporcionar nombres significativos a tus controles y relacionados con su propósito, como por ejemplo FirstNameField o UserNameField.

Estos controles también tienen algunas propiedades comunes relacionadas con sus contenidos. Una de estas propiedades es Text. La propiedad Text guarda el contenido del control, como por ejemplo el texto en un TextField. Por tanto, si tienes un TextField llamado FirstNameField, puedes acceder a sus contenidos usando este código:

```
firstName = FirstNameField.Text
```

En algunos casos tu usuario tendrá seleccionado parte del texto. El TextField, TextArea y PasswordField tienen una propiedad llamada SelectedText, y que te proporcionará sólo el texto seleccionado. Si quieres coger el texto seleccionado de un TextField llamado FirstNameField, puedes utilizar el siguiente código:

```
currentSelection = FirstNameField.SelectedText
```

Otras veces puedes necesitar información sobre el texto seleccionado, como por ejemplo su longitud o posición, sin necesidad de saber exactamente cuál es el texto. Los cuatro controles de entrada tienen dos propiedades llamadas `SelectionStart` y `SelectionLength`. `SelectionStart` te proporciona la posición del primer carácter seleccionado y `SelectionLength` te dará la longitud del texto seleccionado.

El `TextField` propiamente dicho es una simple caja de texto que permite la introducción de texto por parte del usuario. No soporta diferentes estilos como negrita o cursiva o bien el uso de diferentes fuentes. Sin embargo puedes indicar que todo el texto del `TextField` aparezca en negrita, cursiva o bien con una fuente específica. Con independencia del estilo, fuente y tamaño de texto que elijas, este se aplicará a todo el `TextField`.



Esto difiere del control `TextArea`, y que incluye la propiedad `Styled`. Si la propiedad `Styled` está definida como `True`, entonces el `TextArea` puede soportar múltiples fuentes, tamaños de texto y estilos. Estos estilos pueden ser el resultado de texto pegado desde cualquier fuente externa o bien definido mediante código. Unos párrafos atrás te presentamos las propiedades `SelectedText`, `SelectionStart` y `SelectionLength`. El `TextArea`, dado que soporta texto con estilo, se basa en estas con algunas propiedades de selección relacionadas con el estilo: `SelectionBold`, `SelectionItalic`, `SelectionFont` y `SelectionFontSize`, entre otras. Supongamos que tu aplicación tenía un `TextArea` llamado `BiographyField`, donde un usuario tenía que introducir información sobre sí mismo, y querías permitir que el usuario pudiese aplicar negrita a ciertas palabras. Puedes añadir un `PushButton` a la ventana con este código en el evento `Action`:

```
BiographyField.SelectionBold = True
```

Definiendo `SelectionBold` de nuevo a `False` desactiva el estilo negrita. La mayoría de los botones de estilo conmutan un estilo entre encendido y apagado, algo que puedes hacer con este código:

```
BiographyField.SelectionBold = ¬  
Not BiographyField.SelectionBold
```

Este código conmutará el estilo de negrita sin necesidad de conocer cuál es su estado actual. La palabra clave `Not` del ejemplo anterior indica negación, de modo que puedes conmutar la propiedad `Bold` entre activada o desactivada sin necesidad de saber cuál es su actual estado.

La propiedad `SelectionFontName` toma una cadena, el nombre de una fuente, en vez de un booleano. La propiedad `SelectionFontSize` toma un entero que define el tamaño del texto en puntos.

El `PasswordField` es muy similar al `TextField` con una excepción notable: el usuario no puede ver lo que está escribiendo. El `PasswordField` enmascara los caracteres introducidos sustituyéndolos por boliches para preservar la privacidad del password. Los contenidos continúan siendo accesibles para tu aplicación; sólo están ocultos para el usuario. En realidad, el `PasswordField` es un `TextField` con la propiedad `Password` definida a `True`. Se proporciona como un control independiente para tu comodidad.

El `ComboBox` permite que el usuario elija entre varias opciones predeterminadas o introduzca una de su propia elección. Estas opciones predeterminadas se definen utilizando el método `AddRow` del `ComboBox`. Imagina por ejemplo que tienes un `CombBox` para que el usuario seleccione su grado en el colegio. Podrías utilizar el evento `Opening` del `ComboBox` para crear unas cuantas opciones comunes:

```
Me.AddRow( " 6 " )  
Me.AddRow( " 7 " )  
Me.AddRow( " 8 " )  
Me.AddRow( " 9 " )  
Me.AddRow( " 10 " )  
Me.AddRow( " 11 " )  
Me.AddRow( " 12 " )
```

El `ComboBox` permite que el usuario seleccione una de las opciones presentadas o introduzca la suya propia como si fuese un `TextField`.

El método `AddRow` usado en el último ejemplo de código te permite añadir opciones al `ComboBox`. Verás métodos similares en otros controles al final de este capítulo. `AddRow` toma una cadena y añade una nueva fila al final de la lista de opciones en un `ComboBox`. Si necesitas añadir una nueva fila en una determinada posición, podrías usar el método `AddRowAt`. `AddRowAt` toma un entero para saber la posición en la que deseas insertar la nueva fila en la lista, seguido por una `String` con el texto que quieres añadir (como en los arrays, las filas en un `ComboBox` están basadas en un índice cuyo índice mínimo es cero).

Si necesitas borrar todas las filas de un `ComboBox`, también tiene un método llamado `RemoveAllRows` y que hace exactamente lo que indica su nombre.

Si bien no se trata exactamente de un control de entrada, es un buen momento para aprender sobre Label, un control cuya función es principalmente decorativa. Su propósito es el de servir como etiqueta para otros controles, como TextFields, TextAreas y otros. Por ello, su propiedad Caption es muy importante. Ya has utilizado Labels en el proyecto de ejemplo del Capítulo Dos. En una aplicación bien diseñada, muchos de los controles tendrán su correspondiente Label.

6.5 Botones

En esta sección aprenderás sobre diferentes tipos de botones. Puedes preguntarte por qué necesitarías más de un tipo de botón, pero cada uno sirve un propósito ligeramente diferente. Elegir el elemento de interfaz de usuario adecuado para la tarea es una parte fundamental en el diseño de interfaces de usuario.

Xojo ofrece varios tipos de botones, pero sólo trataremos tres de ellos en esta sección: `PushButton`, `BevelButton` y `SegmentedButton`. El aspecto común más importante entre estos botones es el evento `Pressed`. Dicho llanamente, el evento `Pressed` se dispara cada vez que se pulsa el botón.



Tal y como ocurre con los controles de entrada vistos anteriormente, estos botones también tienen propiedades relacionadas con sus tamaños y posición: `Left`, `Top`, `Width` y `Height`. Encontrarás que esto es algo común a todos los controles. De nuevo, cada uno de estos controles tienen una propiedad `Name`, siendo este el que utilizas en tu código para referirte al control. Como con el resto de los controles, es mejor proporcionar a tus botones nombres significativos relacionados con su propósito, como por ejemplo `CancelButton`, `SendEmailButton`, etc.

El `PushButton` es el más simple de estos controles y es uno de los más comunes en los elementos de interfaz de usuario. Además de su propiedad `Name`, también tiene `Caption`, siendo el texto mostrado en el botón. También puedes cambiar la fuente, tamaño y estilo del texto en la etiqueta (`Caption`) del control, pero salvo que tengas una razón para hacerlo, siempre es mejor dejar dichas propiedades definidas a sus valores por defecto. De ese modo, el botón tendrá el aspecto y comportamiento esperado en cada uno de los sistemas operativos.

Aunque la mayoría de las interacciones con los `PushButton` se realizan con el ratón, también tiene dos propiedades relacionadas con el teclado: `Default` y `Cancel`. Si se define `Default` a `True`, entonces el `PushButton` responderá a la tecla de retorno. Si `Cancel` está definido a `True`, entonces responderá a la pulsación

de la tecla de escape (así como Comando-Punto en macOS o Control-Punto en Windows y Linux).

El BevelButton también tiene una propiedad Caption, como en el PushButton. De hecho, puede usarse casi como un PushButton. Pero hay algunas diferencias. En primer lugar, el BevelButton no tiene propiedades Default y Cancel, de modo que no puede responder al teclado del mismo modo que hace el PushButton. En segundo lugar, si bien puedes hacer que se comporte como un PushButton, no tendrá el mismo aspecto que un PushButton, de modo que tienes que ser cuidadoso acerca de dónde lo usas.

De las múltiples propiedades que tiene el BevelButton y que no tiene un PushButton, una especialmente interesante es la propiedad Value. Value es un booleano, y cuando está definido a True, el BevelButton tendrá un aspecto oscurecido (“pulsado”). Esto hace que el BevelButton sea útil como conmutador.

- 1) **En tu proyecto Events, añade un BevelButton a Window1.**
- 2) **En el Inspector, define la propiedad ButtonStyle del BevelButton a ToggleButton.**
- 3) **Escribe una etiqueta (Caption) para el BevelButton.**
- 4) **Ejecuta el proyecto.**
- 5) **Haz clic en BevelButton. Observa como cambia su aspecto.**
- 6) **Sal de la aplicación**

En tu código, puedes determinar si un BevelButton está en su estado “pulsado” o no accediendo a la propiedad Value. Si Value es True, entonces el BevelButton está pulsado.

Un uso común para un BevelButton que conmuta es el de definir estilos para el texto.

- 7) **En tu proyecto Events, añade un TextArea a Window1.**

Nombra el TextArea como StyleDemoField. Asegúrate de que su propiedad AllowStyledText esté definida a True.

- 8) **Cambia el Caption del BevelButton a “Bold” y define su Name a “BoldButton”.**
- 9) **Añade otro BevelButton con el Caption “Italic” y el Name “ItalicButton”.**
- 10) **En el evento SelectionChanged de StyleDemoField, añade el siguiente código:**

```
BoldButton.Value = Me.SelectionBold
```

```
ItalicButton.Value = Me.SelectionItalic
```

11) En el evento Pressed de BoldButton, añade este código:

```
StyleDemoField.SelectionBold = Me.Value
```

12) En el evento Pressed de ItalicButton, añade este código:

```
StyleDemoField.SelectionItalic = Me.Value
```

13) Ejecuta el proyecto.

14) Escribe algo de texto en el TextArea.

Deberías de poder utilizar los botones Bold e Italic para modificar el estilo del texto. Además, cuando haces clic sobre el texto (o seleccionas texto con estilo aplicado) los botones Bold e Italic deberían cambiar para reflejar el estado del texto.

15) Sal de la aplicación.

Como puedes ver en este ejemplo, con el uso de eventos y propiedades en tus controles puedes empezar a crear algunas interacciones complejas. Basándote en este ejemplo no te llevaría mucho más trabajo el añadir más estilos así como selección de fuentes y tamaño de texto.

El SegmentedButton también tiene un evento Pressed, pero tiene una diferencia significativa en comparación con PushButton y BevelButton. El evento Pressed de SegmentedButton te proporciona un parámetro: SegmentIndex como Integer. Esto se

debe a que SegmentedButton se utiliza para realizar selecciones, y no una acción simple como la de PushButton. Más concretamente, SegmentedButton está dirigido a situaciones donde el usuario debe elegir entre dos o más opciones autoexcluyentes. Sólo puede tomarse una y no más de una. El parámetro segmentIndex proporcionado en el evento Pressed indica la opción tomada por el usuario; es decir, el segmento sobre el cual ha hecho clic el usuario.

Un SegmentedButton está formado por un array de Segmentos. Puedes obtener el segmento sobre el que ha pulsado el usuario pasando el parámetro segmentIndex al método SegmentAt de SegmentedButton. Entonces puedes determinar exactamente en qué segmento se ha pulsado utilizando la propiedad Title del Segmento. Si suena confuso probablemente lo veas con más claridad mediante un ejemplo:

16) En tu proyecto Events, añade un SegmentedButton a tu ventana.

Siéntete libre de borrar los controles de los ejemplos más antiguos si tu ventana comienza a estar demasiado poblada.

Por omisión, tu SegmentedButton tendrá dos elementos: One y Two. Puedes editarlos o añadir más utilizando el Inspector, pero por ahora seguiremos con los proporcionados por omisión.

- 17) En el evento Pressed del SegmentedButton, introduce el siguiente código:**

```
Var button As Segment  
button = Me.SegmentAt(segmentIndex)  
MessageBox(button.Title)
```

Examinemos el código antes de ejecutar el proyecto. La primera línea crea una variable llamada button con un tipo de dato Segment. Recuerda que SegmentedButton contiene un array de Segments, de modo que utilizarás esta variable para guardar el ítem seleccionado por el usuario. Esto es lo que ocurre en la segunda línea: utilizas el parámetro segmentIndex proporcionado por el evento Pressed para recuperar el Segmento apropiado del array Segments (segmentIndex indica qué segmento ha sido seleccionado por el usuario). Por último, presenta el texto del ítem seleccionado por el usuario mostrando en un mensaje su propiedad Title (título).

- 18) Ejecuta el proyecto.**
- 19) Selecciona diferentes segmentos.**

Verás un mensaje con el nombre del ítem con cada cambio.

- 20) Sal de la aplicación.**

6.6 Selectores

Otra categoría de controles es la conocida como selectores (pickers, en inglés). Se utilizan estos controles para permitir al usuario que defina opciones y realice selecciones. En esta sección aprenderás sobre CheckBox, RadioButtons, Sliders, UpDownArrows y PopupMenu. Al igual que con el resto de los controles tratados en este capítulo, estos controles tienen el conjunto habitual de propiedades relativas a su tamaño y posición: Left, Top, Width y Height.

El CheckBox se suele utilizar cuando el usuario necesita conmutar entre activada y desactivada una opción o ajuste. Seguramente te hayas encontrado con estos antes en aplicaciones o incluso formularios Web. El CheckBox tiene una propiedad Name, que es como te refieres al CheckBox desde el código, y una propiedad Caption, que es el texto presentado al usuario.

La propiedad más importante del CheckBox es su propiedad Value. Cuando el CheckBox está marcado, su propiedad Value es True, y cuando no es así tiene el valor False.

El CheckBox también tiene un evento Pressed, disparado cuando el usuario lo marca o desmarca.

El RadioButton es similar al CheckBox, pero no puede estar desmarcado. Por lo menos, no por el usuario. Esto se debe a que

el RadioButton está diseñado para su uso en grupos. Cada grupo de RadioButtons contiene un set de opciones autoexcluyentes. Al seleccionar una se deselectan el resto.

1) Crea un nuevo proyecto Xojo y guárdalo como “Options”.

2) En Window1, añade tres RadioButtons.

Para este ejercicio no importa como los llames o lo que muestren sus etiquetas.

3) Ejecuta tu proyecto.

4) Observa que cuando seleccionas un RadioButton, se deselectan los otros.

5) Sal de tu aplicación.



Supón que quieres dos grupos de opciones. Imaginemos que estuvieses creando una app para tomar pedidos de sandwiches, y cada persona pudiese elegir un tipo de carne y queso. Los RadioButton serían un buen modo de diseñar esta interfaz.

Vuelve a tu proyecto Options.

- 6) **Escribe las siguientes etiquetas para tus tres RadioButtons: “Ham”, “Turkey”, y “Salami”.**
- 7) **Añade tres RadioButtons más con estas etiquetas: “Swiss”, “Cheddar”, y “Provolone”.**
- 8) **Ejecuta tu proyecto.**

¿Ves el problema? Al elegir la carne para el sandwich se deseleccionan todas las opciones de queso, y al elegir un queso se deseleccionan todas las opciones de carne. Terminarías con algunos clientes frustrados.
- 9) **Sal de la aplicación.**

¿Por qué se llaman “botones de radio” en cualquier caso? En las primeras radios de los coches había cinco o seis botones, cada uno de los cuales se podía asignar a una emisora (recuerda, esto fue antes de la era de iTunes, Sirius y Spotify). Dado que sólo podías escuchar una emisora a la vez, cuando pulsabas un botón el botón previamente seleccionado se deseleccionaba. Los botones de radio de los ordenadores funcionan de forma similar, de modo que sólo puedes tener una opción seleccionada.

Así que, ¿cómo determinas qué RadioButtons están relacionados con otros y cuáles no forman parte de un grupo? Esto se logra consultando el padre del control.

- 10) **Borra todos los RadioButton de Window1**

- 11) **Añade un Canvas a Window1.**

Veremos el Canvas con mayor detalle posteriormente en este capítulo. Asegúrate de que el canvas tenga la mitad del ancho de la ventana y prácticamente la misma altura.

- 12) **Añade un RadioButton a Window1, pero suéltalo dentro del Canvas.**

Observa que el borde del Canvas se muestra en rojo. Esto indica que el Canvas es ahora el padre del RadioButton.

- 13) **Añade dos RadioButton más dentro del Canvas.**

- 14) **Añade un RadioButton fuera del Canvas.**

- 15) **Ejecuta tu proyecto.**

- 16) **Observa que los tres RadioButton dentro del Canvas actúan como un grupo, mientras que los RadioButton fuera del Canvas no se ven afectados por sus selecciones.**

- 17) **Sal de tu aplicación.**

El Slider es un modo genial de permitir a tus usuarios la selección de un valor numérico dentro de un rango determinado; por ejemplo, si necesitas que tus usuarios seleccionen un número entre 1 y 100. Para establecer estos límites, utiliza las

propiedades `MinimumValue` y `MaximumValue` del `Slider`. En el Inspector puedes ver también otra propiedad entre ellas, denominada `Value`. Este es el valor numérico mostrado por defecto por el `Slider` hasta que el usuario lo cambie o hasta que el código lo cambie.

- 1) **Crea un nuevo proyecto Xojo y guárdalo como “Slider”.**
- 2) **Añade un `TextField` a `Window1`.**
- 3) **Añade un `Slider` a `Window1`.**

En el Inspector, define el Valor Mínimo del slider a 1, su Valor Máximo a 100, y su Valor a 50. Ajusta la propiedad `AllowLiveScrolling` del `Slider` a `True`.

- 4) **Añade el siguiente código al evento `ValueChanged` del `Slider`:**

```
TextField1.Text = Me.Value.ToString
```

- 5) **Ejecuta el proyecto.**
- 6) **Cambia el valor del slider y observa que dicho valor se actualiza en el `TextField`.**
- 7) **Sal de la aplicación.**

Además de como respuesta a la acción del usuario, puedes definir el `Value` de un `Slider` en código, utilizando la sintaxis:

```
MySlider.Value = 50
```

Otro control utilizado para solicitar un valor numérico al usuario es `UpDownArrows`. Estas son las pequeñas flechas hacia arriba y hacia abajo. No tienen propiedades `Minimum`, `Maximum` o `Value`; de modo que si necesitas estas propiedades tendrás que gestionarlas tu mismo o bien usar un `Slider`.

El control `UpDownArrows` tienen dos eventos importantes: `Up` y `Down`. El evento `Up` se dispara cuando el usuario hace clic en la flecha que apunta hacia arriba, y el evento `Down` se dispara cuando el usuario hace clic en la flecha que apunta hacia abajo.

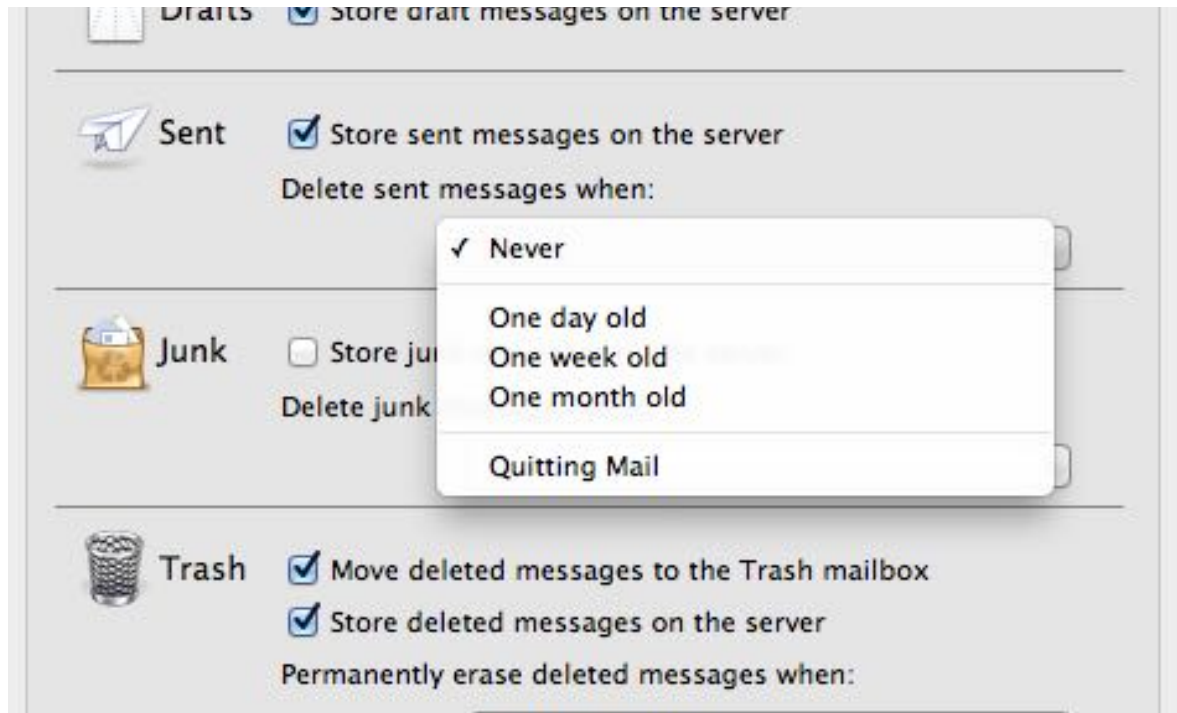
- 1) **Crea un nuevo proyecto Xojo y guárdalo como “Arrows”.**
- 2) **Añade un `TextField` a `Window1`**
- 3) **Añade un `UpDownArrows` a `Window1`.**
- 4) **Añade este código en el evento `Down` de tu `UpDownArrows`:**

```
Var i As Integer  
i = TextField1.Text.ToInteger  
i = i - 1  
TextField1.Text = i.ToString
```

- 5) **Añade este código en el evento `Up` de tu `UpDownArrows`:**

```
Var i As Integer  
i = TextField1.Text.ToInteger  
i = i + 1  
TextField1.Text = i.ToString
```

- 6) Ejecuta el proyecto.
- 7) Usa el control **UpDownArrows** para cambiar el valor mostrado en el **TextField**.
- 8) Sal de la aplicación.



El último control que trataremos en la categoría “selectores” es el **PopupMenu**. Este es el más complicado del grupo, pero también es un control potente. Al igual que en el resto de los controles, tiene una propiedad **Name** que puedes definir en el **Inspector**; este es el nombre que utilizarás para referirte al control desde el código.

Accederás a la mayoría del resto de las propiedades usadas con frecuencia desde código. La propiedad que utilizarás con más

frecuencia es **SelectedRowValue**. Esta propiedad devuelve el valor de la fila (o elemento) seleccionado en el **PopupMenu**.

Pero antes de que tengas un **SelectedRowValue** con el que trabajar, tendrás que añadir algunos ítems a tu **PopupMenu**. Esto se hace usando el método **AddRow**.

- 1) Crea un nuevo proyecto **Xojo** y guárdalo como “**Popup**”.
- 2) Añade un **PopupMenu** a **Window1**.
- 3) Añade el siguiente código en el evento **Opening** de **PopupMenu**:

```
Me.AddRow( "PopupMenu" )
Me.AddRow( "TextArea" )
Me.AddRow( "UpDownArrows" )
Me.AddRow( "Window" )
```

- 4) Añade este código en el evento **SelectionChanged** de **PopupMenu**:

```
MessageBox( "You have chosen " +  
            + Me.SelectedRowValue + " .")
```

El evento **SelectionChanged** se dispara cuando el usuario hace una selección en el **PopupMenu**.

- 5) **Ejecuta el proyecto.**
- 6) **Elige un item del menú. Deberías ver un mensaje mostrando el texto del item seleccionado.**
- 7) **Sal de la aplicación.**

Como puedes ver en el anterior ejemplo, el método `AddRow` toma una string como parámetro. Esta string es el texto a mostrar para ese elemento de menú. `AddRow` añadirá siempre un elemento al final de la lista. Si necesitas añadir un elemento de menú en una posición determinada, tienes que usar el método `AddRowAt`. `AddRowAt` toma dos parámetros: el primero es un entero indicando su posición en la lista (nuevamente, está basado en índice cero), y luego la string que aparecerá para dicho item de menú.

El `PopupMenu` también tiene el método `RemoveRowAt`, tomando un entero que indica el número del item que quieres eliminar. Al igual que ocurre con el `ComboBox`, el `PopupMenu` también tiene el método `RemoveAllRows`.

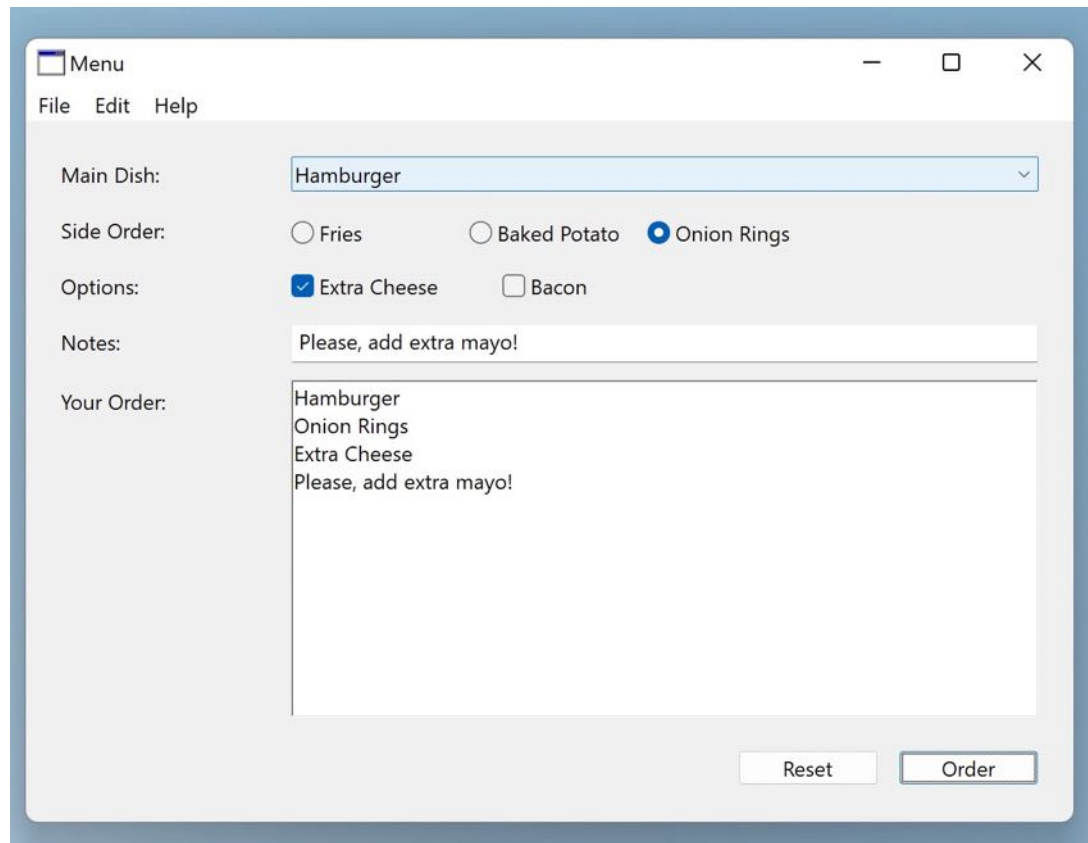
Cuando el usuario hace una selección, se ajusta `SelectedRowIndex` en el `PopupMenu`. `SelectedRowIndex` es un entero que indica el número del item seleccionado. Para acceder al texto asociado con dicho elemento, utiliza la propiedad `SelectedRowValue` del `PopupMenu` tal y como hemos hecho antes:

```
myChoice = MyPopupMenu.SelectedRowValue
```

6.7 Controles en la Práctica

Para el proyecto de ejemplo de este capítulo vamos a crear un menú de comidas electrónico para un restaurante ficticio, el cual tomará pedidos mediante el ordenador en vez de usar camareros y camareras. El usuario podrá seleccionar un plato principal, elegir un complemento, seleccionar opciones y dejar anotaciones para el cocinero.

Tu aplicación será similar a esta:



- 1) Crea un nuevo proyecto en Xojo y guárdalo como “Menu”.
- 2) Cambia el título de Window1 a algo con significado, como por ejemplo “Menu” o “Order Here”.
- 3) Añade una Label a Window1 y cambia su propiedad Value a “Main Dish:”.
- 4) Añade un PopupMenu a continuación de la Label. Cambia su nombre a “MainDishMenu”.
- 5) Añade el siguiente código en el evento Opening de MainDishMenu:

```
Me.AddRow( "Hamburger" )  
Me.AddRow( "Cheeseburger" )  
Me.AddRow( "Pizza" )
```

Si quieres ser creativo con las opciones, ¡siéntete libre de hacerlo!

- 6) Añade otra Label a Window1 y cambia su propiedad Value a “Side Order:”.
- 7) Añade un RadioGroup a Window1, llamado “SideOrderRadio”. Edita su Initial Value a “Fries”, “Baked Potato”, y “Onion Rings”. Conmuta Horizontal a activado.
- 8) Añade otra Label a Window1 y cambia su propiedad Value a “Options:”.
- 9) Añade dos CheckBox a Window1, llamadas “CheeseCheckBox” y “BaconCheckBox”. Define sus Caption a “Extra Cheese” y “Bacon”.

- 10) **Añade otra Label a Window1 y cambia su propiedad Value a “Notes:”.**
- 11) **Añade un TextField a continuación de dicha Label y cambia su Name a “NotesField”.**
- 12) **Añade otra Label a Window1 y cambia su propiedad Value a “Your Order:”.**
- 13) **Añade un TextArea a continuación de dicha Label y cambia su Name a “OrderArea”.**
- 14) **Añade dos Buttons cerca de la parte inferior de Window1. Cambia sus Name a “ResetButton” y “OrderButton”, y cambia sus Caption a “Reset” y “Order”.**
- 15) **Ajusta la propiedad Enabled de OrderButton a OFF.**

Esto evitará que los usuarios hagan sus pedidos sin haber elegido un plato principal.

- 16) **Añade este código en el evento SelectionChanged de MainDishMenu:**

```
If Me.SelectedRowIndex = -1 Then
    OrderButton.Enabled = False
Else
    OrderButton.Enabled = True
End If
```

- 17) **Introduce este código en el evento Pressed de ResetButton para reiniciar todos los controles de la ventana::**

```
OrderArea.Text = ""
```

```
MainDishMenu.SelectedRowIndex = -1
SideOrderRadio.SelectedRowIndex = -1
CheeseCheckBox.Value = False
BaconCheckBox.Value = False
NotesField.Text = ""
```

- 18) **Añade un método a Window1 llamado “CompileOrder”. Su código es el siguiente:**

```
Var theOrder As String
If MainDishMenu.SelectedRowIndex <> -1 Then
    OrderArea.Value = ""
    theOrder = MainDishMenu.↵
        SelectedRowValue + EndOfLine
    theOrder = theOrder + sideOrderRadio. ↵
        ItemAt(sideOrderRadio.SelectedIndex). ↵
        Caption
    theOrder = theOrder + EndOfLine

    If CheeseCheckBox.Value Then
        theOrder = theOrder ↵
            + CheeseCheckBox.Caption
        theOrder = theOrder + EndOfLine
    End If
    If BaconCheckBox.Value Then
        theOrder = theOrder ↵
            + BaconCheckBox.Caption
        theOrder = theOrder + EndOfLine
    End If
    theOrder = theOrder + NotesField.Value
    OrderArea.Text = theOrder
End If
```

19) Añade este código en el evento Pressed de OrderButton:

```
CompileOrder
```

20) Ejecuta tu proyecto.

21) Realiza un pedido y elige tus opciones.

22) Sal de la aplicación.

Evidentemente, este proyecto de ejemplo no realizará pedidos reales. Su objetivo es el de mostrarte una combinación de controles funcionando juntos en una interfaz. Para tener más práctica, prueba a refinar la interfaz de usuario de este proyecto utilizando para ello GroupBox y SegmentedButton.

Navegando

CONTENIDOS

1. **Introducción al Capítulo**
2. **ListBox**
3. **Decoración**
4. **Organizadores**
5. **Indicadores**

7.1 Introducción al Capítulo

Este capítulo es esencialmente una prolongación del Capítulo 6. Aprenderás sobre algunos controles adicionales disponibles, y crearás un navegador web mínimo pero funcional.



7.2 ListBox

Uno de los controles principales que aun no se ha cubierto es el ListBox. El ListBox es un control extremadamente útil con varios eventos y propiedades. Esta sección proporciona una introducción a dicho control.

Puedes pensar en el ListBox como en una tabla. De hecho, en algunos lenguajes es conocido como tabla o rejilla. Contiene filas y columnas y la intersección de una fila y una columna se denomina celda, tal y como ocurre con una hoja de cálculo. Los contenidos de un ListBox pueden desplazarse, tanto vertical como horizontalmente.

Al igual que en otros controles de este capítulo, el ListBox ofrece las habituales propiedades para ajustar el tamaño y posición. Un simple vistazo al Inspector revela, sin embargo, muchas más propiedades. Dos propiedades importantes del ListBox son ColumnCount y HasHeader. ColumnCount es un entero que indica cuántas columnas tiene el ListBox. Las columnas propiamente dichas están basadas en una lista de índice cero; de modo que un ListBox con tres columnas tendrá una columna 0, columna 1, y columna 2. HasHeader es un booleano que determina si el ListBox ha de mostrar una fila de cabecera. Una fila de cabecera es diferente visualmente sobre el resto, además de mantener su posición en la parte superior del ListBox mientras que se desplazan el resto de filas.



Puede cambiarse el texto contenido en las celdas de la cabecera mediante el Inspector definiendo la propiedad InitialValue. Introduce el texto de las cabeceras separado por tabuladores y se aplicarán al ListBox.

El ListBox también tiene algunos eventos y métodos que pueden resultar familiares de otros controles. Por ejemplo, has visto que PopupMenu y ComboBox tienen un método llamado AddRow. El ListBox también tiene dicho método. Al igual que en el resto de los controles con este evento, AddRow toma una string como parámetro. Dicha string se añadirá al final del ListBox, en la primera columna.

Esto puede llevarte a que te preguntes sobre cómo puedes asignar valores al resto de celdas para dicha fila. Este es un ejemplo en el evento Opening de un ListBox con tres columnas:

```
Me.AddRow( "Dylan" )
Me.CellValueAt( Me.LastAddedRowIndex, 1 ) = "Bob"
Me.CellValueAt( Me.LastAddedRowIndex, 2 ) = "Musician"
```

Dicho código añadirá una fila con tres celdas. Añade unas cuantas filas más y podría verse de forma similar a esto:

Dylan	Bob	Musician
Bezos	Jeff	Businessman
Bolt	Usain	Athlete

Las celdas de un ListBox pueden almacenarse en un array bidimensional llamado Cell (recuerda, dado que es un array, está basado en cero). Por lo tanto, para actualizar los contenidos de la celda en Fila 1, Columna 2 (“Businessman”), podrías usar este código:

```
Me.CellValueAt(1, 2) = "Entrepreneur"
```

También puedes observar que el anterior código utiliza una propiedad llamada LastAddedRowIndex. Dicha propiedad es el número (basado en cero) del índice correspondiente a la última fila añadida al ListBox. Por lo tanto, usando LastAddedRowIndex

inmediatamente después de AddRow siempre te permitirá añadir datos a las celdas más allá de la primera columna.

Además del método AddRow, el ListBox también soporta el método AddRowAt. Al igual que ocurre con el PopupMenu y con ComboBox, AddRowAt toma dos parámetros: un entero indicando su posición en la lista, y la cadena a añadir al ListBox. LastAddedRowIndex funciona con AddRowAt del mismo modo que funciona con AddRow, de modo que se puede usar con seguridad independientemente de como añadas datos al ListBox.

Cuando el usuario selecciona una fila en el ListBox, se dispara el evento SelectionChanged. Este no indica la fila seleccionada; pero, como has visto en anteriores capítulos, puedes acceder a esta información usando la propiedad SelectedRowIndex del ListBox. Esta propiedad está basada en índice cero, de modo que la primera fila tiene el índice cero. Si no hay una fila seleccionada, entonces SelectedRowIndex será -1.

El ListBox también tiene el evento DoublePressed. En muchas aplicaciones que utilizan un ListBox un clic sencillo (que dispara el evento SelectionChanged) selecciona una fila, mientras que un clic doble abrirá una nueva ventana para editar o ver sus contenidos (como cuando se hace un doble clic en un mensaje de email). En este evento puedes utilizar SelectedRowIndex para determinar sobre qué fila se ha hecho doble clic. Esta ha sido sólo una introducción breve al ListBox. Aprenderás más sobre él en los próximos capítulos.

7.3 Decoración

Un vistazo rápido a la sección Decor del listado de controles en Xojo revelará unos cuantos controles en esta categoría. En esta sección aprenderás sobre GroupBox y el Canvas.

Si recuerdas el ejemplo RadioButton del Capítulo Seis, necesitas un modo de agrupar los RadioButton. Dicho ejemplo usaba un Canvas, pero otra forma de hacerlo es usando el GroupBox. Como su nombre implica, se usa el GroupBox para agrupar controles. Esto puede ser por motivos lógicos, como en el agrupamiento de RadioButton, o por motivos estéticos, como simplemente proporcionar un mejor aspecto visual a tu aplicación. Verás que a menudo se usan los GroupBox en las preferencias de las aplicaciones y ajustes de las ventanas, de modo que se dispongan un conjunto de controles en categorías.

Para organizar los RadioButton en un GroupBox, añade el GroupBox a la ventana en primer lugar, y arrastra luego los RadioButton sobre él. El borde del GroupBox se verá de color rojo, indicando que se ha convertido en el control “padre” de los RadioButton. Esta jerarquía va más allá de eso; el padre del GroupBox es la ventana propiamente dicha.

Desde el punto de vista de la ejecución de código, el GroupBox no hace realmente mucho. Desde el punto de vista del diseño, es útil para contener otros controles, como en el caso indicado de

los RadioButton. Es capaz de contener cualquier otro control. Para el diseño de tu aplicación, también es importante definir la propiedad Caption del GroupBox.



El Canvas es uno de los controles más potente ofrecido por Xojo. En esta sección sólo rascarás la superficie de lo que puede hacer. Aprenderás sobre algunas de sus capacidades en el Capítulo Diez. De hecho, en esta sección sólo verás uno de los eventos en el Canvas: el evento Paint.

El evento Paint te proporciona un parámetro: g como Graphics. Aprenderás mucho más sobre la clase Graphics en el Capítulo Diez, pero esto te proporcionará una breve introducción.

1) Crea un nuevo proyecto Xojo y guárdalo como “Canvas”.

2) Añade un Canvas a Window1.

3) Añade el siguiente código en el evento Paint de Canvas1:

```
g.DrawText("Hello!", 20, 20)
```

4) Ejecuta el proyecto.

Cuando aparezca Window1, este mostrará “Hello!”.

5) Sal de la aplicación.

6) Cambia el código en el evento Paint por este:

```
Var d As DateTime = DateTime.Now  
g.DrawText(d.ToString, 20, 20)
```

7) Ejecuta el proyecto.

En esta ocasión, aparecerán la fecha y hora en la ventana.

8) Sal de la aplicación.

9) Cambia el código en el evento Paint por este:

```
g.DrawRectangle(20,20,20,20)  
g.FillRectangle(40,40,40,40)  
g.DrawOval(60,60,60,60)
```

10) Ejecuta el proyecto.

Deverías ver varias figuras dibujadas en la ventana.

11) Sal de la aplicación.

Por ahora, puede que te preguntes qué está ocurriendo. El objeto Graphics proporcionado por el evento Paint del Canvas te permite hacer prácticamente cualquier cosa; desde mostrar texto a dibujar figuras e imágenes. Puede que esto no signifique mucho por ahora, pero cuando tratemos los gráficos en este libro, así como la impresión y las subclases, verás que puedes crear incluso tus propios controles mediante el Canvas.

7.4 Organizadores

Esta sección cubrirá el TabPanel y el PagePanel. Estos controles, al igual que el GroupBox, son útiles para agrupar otros controles en unidades lógicas; pero también ofrecen la ventaja de ocultar y mostrar estos grupos de controles.



Probablemente te hayas topado ya con el TabPanel en varias aplicaciones, especialmente en sus ventanas de preferencias y opciones. El TabPanel, como su nombre implica, se utiliza para mostrar un grupo determinado de controles al tiempo que oculta otros. Los controles ocultos permanecen activos y accesibles desde código, pero son invisibles para el usuario.

- 1) Crea un nuevo proyecto Xojo y guárdalo como “Tabs”.
- 2) Arrastra un TabPanel en Window1.
- 3) En el Inspector, haz clic en el botón Edit próximo a Panels bajo Appearance.
- 4) Cambia Tab0 a “Global Settings”.
- 5) Cambia Tab1 a “User Settings”.
- 6) Vuelve al editor de la ventana, con la primera pestaña del TabPanel seleccionada, añade algunos controles de tu elección.
- 7) Cambia a la segunda pestalla del TabPanel y añade un grupo de controles diferente.
- 8) Ejecuta el proyecto.
- 9) Cambia entre las pestañas y observa como se conmuta la visibilidad de unos controles y otros.
- 10) Sal de la aplicación.

Si necesitas saber qué pestaña se ha seleccionado, el TabPanel proporciona el evento PanelChanged. Puedes saber qué pestaña está seleccionada accediendo a la propiedad SelectedPanelIndex del TabPanel.

El PagePanel es similar al TabPanel, salvo que no se proporciona una vía de navegación. Mientras que el TabPanel proporciona pestañas a lo largo de su parte superior, para permitir que el usuario pueda cambiar con facilidad, en el caso de los PagePanel

has de proporcionar tu propio sistema de navegación. Esto hace que el PagePanel sea muy útil como un tipo de interfaz “asistente” que guíe al usuario a través de varios pasos en un orden determinado.

La navegación se realiza definiendo la propiedad `SelectedPanelIndex` del `PagePanel`. Dicha propiedad es un entero que indica la página activa (la lista de páginas está basada en índice cero). Las páginas se configuran cuando diseñas la ventana, del mismo modo en el que definías las pestañas de un `TabPanel`.

Hay varias formas de diseñar la navegación de tu `PagePanel`. El más sencillo es tener un `PushButton` o `BevelButton` fuera del `PagePanel`, con un código similar a este en su evento `Pressed`:

```
PagePanel1.SelectedPanelIndex =  $\neg$ 
PagePanel1.SelectedPanelIndex + 1
```

También se puede proporcionar un botón adicional para volver a la página anterior, utilizando por ejemplo este código:

```
PagePanel1.SelectedPanelIndex =  $\neg$ 
PagePanel1.SelectedPanelIndex - 1
```

Otra forma de proporcionar navegación es añadiendo un botón dedicado en cada página del `PagePanel`, llevándote a la siguiente

página en su evento `Pressed`. Esto puede tornarse difícil de gestionar a medida que la aplicación aumente en complejidad.

Un tercer modo consiste en dirigir al usuario a una página concreta en función de sus acciones. Por ejemplo, imagina que estás creando una interfaz de “asistente” para guiar al usuario en la configuración de una cuenta en una red social. Puede que ofrezcas al usuario la opción entre crear un nuevo nombre de usuario y contraseña o bien utilizar un sistema de autenticación externo como Google o Facebook. Si el usuario crea una nueva cuenta, puedes definir `PagePanel` para que le lleve a la página de acceso. Si elige una autenticación externa, el `PagePanel` puede mostrarle una página en la que introduzca sus credenciales para dicho servicio.

7.5 Indicadores

Uno de los aspectos más importantes en una interfaz de usuario bien diseñada es su reacción. Sin embargo, en algunos casos el código simplemente requiere de algún tiempo para su ejecución. En estos casos es mejor proporcionar a tu usuario algún tipo de indicación de modo que sepa lo que está ocurriendo. A menudo, si una aplicación no proporciona este tipo de retroalimentación durante más de unos pocos segundos o parece que no esté haciendo nada, el usuario asumirá que la aplicación se ha colgado. Si tu aplicación está procesando un conjunto de datos de gran tamaño, puede resultar muy útil mostrar un `ProgressBar` o un `ProgressWheel`. Esto indica al usuario que la aplicación continúa funcionando.



Debería de usarse la `ProgressBar` cuando puedas cuantificar la cantidad de tiempo que le llevará a tu aplicación, o bien la cantidad de elementos que han de procesarse. La `ProgressBar` tiene una propiedad `MaximumValue` que debería definirse a dicho número. En otras palabras, si tu aplicación ha de procesar un array con 200 items, deberías definir el `MaximumValue` de la `ProgressBar` a 200. Por otro lado, la propiedad `Value` de `ProgressBar` debería reflejar el avance del proceso.

- 1) **Crea un nuevo proyecto Xojo y guárdalo como “Progress”.**
- 2) **Arrastra una ProgressBar en Window1.**
- 3) **En el Inspector, asegúrate de que MaximumValue está configurado a 100 para la ProgressBar, y de que su Value está definido a cero.**
- 4) **Arrastra un Timer a Window1.**

Observa que el Timer se sitúa bajo el Editor de la Ventana (en una área denominada “Shelf” o bandeja). Esto se debe a que no será parte de la interfaz de usuario de la aplicación. Se tratarán los Timer en profundidad en un capítulo posterior. Por ahora, sólo has de saber que un Timer realizará una tarea dada una vez que haya transcurrido el intervalo de tiempo indicado.

- 5) **Haz doble clic en el Timer e introduce este código en su evento Action:**

```
ProgressBar1.Value = ProgressBar1.Value + 1
```

- 6) **Ejecuta el proyecto.**

La ProgressBar debería de comenzar a rellenarse de izquierda a derecha, completando el relleno cada segundo.

- 7) **Sal de la aplicación.**

Cuando se utiliza en combinación con los Timer y los Thread (ambos tratados en un capítulo posterior), la ProgressBar puede ser realmente valiosa a la hora de proporcionar información al usuario sobre el estado actual de la aplicación.

Sin embargo, en muchas ocasiones no sabrás cuántos elementos han de procesarse o cuanto llevará completar una tarea. Por ejemplo puede que estes esperando a recibir datos de un servidor externo, y la velocidad dependerá de muchos factores lejos de tu control. En estos casos, dado que no se indica cuánto tiempo resta para una tarea determinada, el ProgressWheel es un modo adecuado de indicar al usuario que la aplicación aún está funcionando.

En vez de contar con un MaximumValue y un Value, se define el ProgressWheel como visible o invisible según necesitemos.

- 1) **Abre tu proyecto Progress en Xojo.**
- 2) **Borra la ProgressBar y el Timer en Window1.**
- 3) **Añade un ProgressWheel a Window1.**
- 4) **En el Inspector, define la propiedad Visible del ProgressWheel a OFF.**
- 5) **Añade un PushButton a Window1.**
- 6) **Añade este código en el evento Pressed del PushButton:**

```
ProgressWheel1.Visible = Not ProgressWheel1.Visible
```

- 7) **Ejecuta el proyecto.**
- 8) **Haz clic en el PushButton para conmutar la visibilidad del ProgressWheel.**
- 9) **Sal de la aplicación.**

Como en la ProgressBar, el ProgressWheel es más valioso cuando se utiliza en combinación con los Timer y los Thread.

Házlo tu Mismo

CONTENIDOS

1. **Introducción al Capítulo**
2. **Programación Orientada a Objetos**
3. **Clases y Objetos**
4. **Variantes**
5. **Métodos y Funciones en Clases**
6. **Módulos**

8.1 Introducción al Capítulo

Hasta ahora hemos introducido los tipos de datos, variables, eventos, controles, arrays, métodos, funciones, bucles y más. De hecho, en este punto, tienes suficientes herramientas para crear una aplicación sofisticada; siempre y cuando utilices los tipos de datos proporcionados por Xojo. Pero encontrarás casos, y serán muchos, en los que necesites definir tus propios tipos de datos con propiedades y métodos únicos. Para ello, utilizas una clase. Una clase es algo que representa un objeto de la vida real o una idea abstracta. Por ejemplo, puedes tener una clase que represente un coche (un objeto de la vida real) o una transacción bancaria (una idea abstracta).

En este capítulo aprenderás como crear y usar tus propias clases, y también aprenderás sobre los módulos, que te ofrecen una forma de proporcionar datos y métodos globales a tu aplicación.

8.2 Programación Orientada a Objetos

Hay dos formas básicas de desarrollar software: Programación Procedural y Programación Orientada a Objetos.

La Programación Procedural está ejemplificada en el lenguaje BASIC. El ordenador tenía que ejecutar cada línea, en orden, y luego detenía la ejecución. Puede que esta aproximación realice la tarea, pero no deja mucho espacio al error y puede ser muy complicado añadir nuevas características y corregir fallos.

La Programación Orientada a Objetos, por otra parte, toma una aproximación distinta. En esencia, creas objetos y les indicas cómo han de comportarse: responder a diferentes eventos, mostrar datos, etc. Estos objetos pueden interactuar entre sí prácticamente en cualquier modo que puedas imaginar.

Xojo es un lenguaje Orientado a Objetos, como Swift, C#, Objective-C o Java. Si bien no necesitas usar todos los principios de la Programación Orientada a Objetos, puede ayudarte a crear código más flexible y fácil de mantener.

Una referencia completa a la Programación Orientada a Objetos está más allá del enfoque de este libro, pero este capítulo y algunos de los siguientes se aprovecharán de que Xojo sea orientado a objetos.

8.3 Clases y Objetos

Como se ha indicado, una clase es algo que representa un objeto de la vida real (algo que puedes tocar) o una idea abstracta (un concepto o idea “intangible”). Para ilustrar una clase que representa un objeto de la vida real, crearás una clase llamada Student.

Antes de que empieces a escribir código, detente a pensar sobre los atributos de un estudiante. Algunas cosas evidentes que vienen a la cabeza son su nombre, apellido, fecha de nacimiento, y grado. A continuación, piensa sobre los tipos de datos que necesitarás para almacenar dicha información. Por ejemplo, el nombre y el apellido son strings. La fecha de nacimiento es un DateTime. Para mayor simplicidad, hagamos que el Grado sea un entero (asumamos que es la escuela secundario o intermedia, sin clase de párvulos).

Los atributos de un objeto de la vida real se pueden expresar como propiedades de una clase. Tu clase estudiante (student) podría tener las siguientes propiedades:

Propiedad	Tipo de Dato
First Name	String
Last Name	String
Middle Name	String
Birthdate	DateTime

Puedes continuar añadiendo propiedades, como el color del pelo, color de los ojos, altura, peso, etc. Por ahora, esta lista de propiedades es suficiente.

Pero los nombres de las propiedades están sujetas a las mismas reglas de los nombres de la variables (no se pueden usar espacios o signos de puntuación aparte del guión bajo); de modo que una lista actualizada de las propiedades debería ser como esta:

Propiedad	Tipo de Dato
FirstName	String
LastName	String
MiddleName	String
Birthdate	DateTime
GradeLevel	Integer

Un ejemplo de una clase que represente una idea abstracta podría ser un curso. Un curso no es un objeto físico, y no puedes tocarlo, pero es ciertamente un concepto que impacta en la vida real de un estudiante. Pensemos ahora en los atributos de un curso. Un curso tiene un título, un instructor, una clase o ubicación, y un tema como por ejemplo matemáticas, sociedad, ciencia, etc. Estos pueden almacenarse como strings. De modo que tu clase curso (course) podría tener estas propiedades:

Propiedad	Tipo de Dato
Title	String
Instructor	String
Room	String
Subject	String

Nuevamente, podrías continuar añadiendo más propiedades pero estas servirán por ahora.

Algunos de los tipos de datos que has aprendido en anteriores capítulos son ejemplos de clases proporcionados por Xojo. Como dichos tipos de datos, las clases han de instanciarse, o crearse, usando la palabra clave New; tal que así:

```
Var s As Student
s = New Student
```

Hablar de clases en abstracto está bien, pero no serán útiles para tus aplicaciones hasta que las añadas a tus proyectos.

1) Crea un nuevo proyecto Xojo Desktop y guárdalo como “StudentInformation”.

2) Dirígete al menú Insert y selecciona Class.

Aparecerá el editor de clase. Aquí es donde introducirás el nombre de tu clase. Llama a esta clase “Student”. Los otros campos (Super e Interfaces) pueden dejarse en blanco por ahora (aprenderás más sobre estos temas en el Capítulo Trece).

3) Con tu clase Student seleccionada en el Navegador, dirígete al menú Insertar y selecciona Property.

Tu propiedad necesita un nombre (Name) y tipo (Type), además de que puedes definir un valor por defecto y el ámbito (scope) de la propiedad. Nombra tu propiedad FirstName y define Type a String. El Scope debería ser Public (público). Puedes dejar Default vacío por ahora.

4) Repite este proceso y añade otras propiedades: LastName como String, MiddleName como String, BirthDate como DateTime, y GradeLevel como Integer.

5) Vuelve al menú Insert y selecciona Class para añadir la siguiente clase. Nombra esta clase “Course”.

6) Añade estas propiedades a tu clase Course: Title como String, Instructor como String, Room como String, y Subject como String.

7) En el Navegador, selecciona Window1 y selecciona Property en el menú Insert.

Esta nueva propiedad te permitirá añadir algunos cursos (Course) a tu aplicación para que tengas algunos datos con los que trabajar. El nombre de la propiedad debería ser MyCourses() y su tipo debería de ser Course. Esto creará un array de Courses que puedes usar. Observa que Courses se autocompleta en el campo Type.

8) Crea un nuevo método llamado “GenerateCourses” en Window1 “GenerateCourses”.

9) Añade este código a GenerateCourses:

```
Var c As Course
```



```

c = New Course
c.Instructor = "Mr. Smith"
c.Room = "101"
c.Subject = "Science"
c.Title = "Biology"
MyCourses.AddRow(c)
c = New Course
c.Instructor = "Mrs. Jones"
c.Room = "202"
c.Subject = "Mathematics"
c.Title = "Geometry"
MyCourses.AddRow(c)
c = New Course
c.Instructor = "Ms. Jackson"
c.Room = "301"
c.Subject = "World Language"
c.Title = "Spanish III"
MyCourses.AddRow(c)

```

Este código creará algunos Cursos de ejemplo. Normalmente esto se haría desde una base de datos o una fuente de datos externa pero, por ahora, puedes crearlos en código.

Siéntete libre de añadir más cursos a la lista si lo deseas. Observa que cada vez que utilizas la línea “c = New Course”, se descarta el anterior valor y se crea uno nuevo. Observa también que las propiedades añadidas están disponibles utilizando el autocompletado de Xojo mediante la notación por punto.

10) **Añade un ListBox a Window1 y nómbralo “CourseBox”.**

Se utilizará el ListBox para mostrar un listado con los cursos disponibles.

11) **Crea otro método llamado “ListCourses” en Window1.**

12) **Añade este código a ListCourses:**

```

CourseBox.RemoveAllRows
For Each c as Course in MyCourses
    CourseBox.AddRow(c.Title)
Next

```

Este método recorrerá tu array de cursos (usando el bucle For...Each) y añadirá cada título al LisBox.

13) **En el evento Opening de Window1, tendrás que ejecutar estos dos métodos:**

```

GenerateCourses
ListCourses

```

14) **Ejecuta tu proyecto.**

Deberías ver los cursos en el ListBox.

15) **Sal de la aplicación.**

Puede resultar tentador en este punto pensar que el ListBox contiene ahora la información sobre tus cursos. En un sentido muy limitado, esto es así, pero sólo contiene el título de cada curso. Con simplemente el título, resultaría muy difícil buscar otra información sobre el curso salvo que recorrieses todo el array para hallar una coincidencia, o bien cambiar los datos de curso de un array a un diccionario. Incluso en ese caso tendrías que asegurarte de no tener nombres de cursos duplicados.

El motivo por el que no tienes toda la información de tus cursos en el ListBox es porque sólo has utilizado una de sus propiedades. Para tener acceso a toda la información del curso, necesitas utilizar un objeto.

Si piensas en una clase como en un plano, puedes pensar en un objeto como en la casa real.

Una clase es una descripción de algo, y el objeto es la cosa en sí misma. Cuando utilizas la palabra clave New, estás creando un nuevo objeto en tu código. Por tanto, cuando en el código anterior creabas algunos cursos de ejemplo, cada uno de ellos era un objeto. En este ejemplo sólo tomabas el título de cada objeto Course, pero es posible almacenar todo el objeto en el ListBox. Cambia el método ListCourses por esto:

```
CourseBox.RemoveAllRows
For Each c as Course in MyCourses
    CourseBox.AddRow(c.Title)
    CourseBox.CellTagAt(CourseBox.¬
        LastAddedRowIndex,0) = c
Next
```



8.4 Variantes

La pregunta evidente que puede derivarse del cambio en el código es: ¿qué es CellTag? Como has visto anteriormente, cada ListBox tiene un array bidimensional de celdas. También tiene un array bidimensional de CellTags. Puedes pensar en una CellTag como en un “compartimento secreto” en el que puedes almacenar datos.

Mientras que sólo puedes almacenar strings en una Cell, un CellTag almacena un tipo de dato denominado Variant (Variante). Una Variant es un tipo de dato flexible que puede almacenar una string, un entero, un double, un DateTime, un Diccionario, o cualquier otro tipo de dato, incluso tus propios objetos Course. Por lo tanto, el código anterior asigna cada objeto Course a la CellTag en el ListBox. Teniendo en cuenta el modo en el que está estructurado el código, cada Cell del ListBox contendrá una CellTag con datos relacionados del curso, almacenado como variante.

Esto lanza otra pregunta: ¿cómo puedes obtener y asignar datos de una variante? Asignar datos a una variante es algo directo, equivalente al modo de hacerlo con cualquier otro tipo de dato.

- 1) **Crea un nuevo proyecto Xojo y guárdalo como “Variants”.**
- 2) **Escribe este código en el evento Opening de Window1:**

```
Var v As Variant  
v = "Hello!"  
MessageBox(v)
```

- 3) **Ejecuta tu proyecto.**

Deberías ver un mensaje diciendo “Hello!”. Como ves, en el caso de datos string, el uso de una variante es como usar cualquier otro tipo de dato sencillo.

- 4) **Sal de la aplicación.**
- 5) **Cambia el código del evento Opening en Window1 por este:**

```
Var v As Variant  
v = 123  
MessageBox(v)
```

En esta ocasión, en vez de almacenar una string en la variante, has almacenado un entero. Recuerda del Capítulo Dos que no puedes presentar un entero en un mensaje al usuario sin convertirlo primero mediante la función ToString. Cuando usas una Variante, sin embargo, se convierte el dato automáticamente por tí.

Esto puede resultar extremadamente conveniente, pero también peligroso. Por ejemplo:

```

Var first, second, third As Variant
first = 123
second = "456"
third = first + second
MessageBox(third)

```

¿Qué debería mostrarse en el mensaje? Podrías decir que el mensaje debería de mostrar “123456” (si se combinan los valores de las variantes como strings), pero también podría ser “579” (si se suman los valores como enteros). Cambia el código del evento Open en Window1 por este código y compruébalo por ti mismo. ¿Te ha sorprendido el resultado?

Esto ilustra que si bien las variantes son ciertamente potentes y útiles, deben utilizarse con cuidado y sólo cuando sea necesario. Almacenar un objeto en el CellTag de un Listbox es un ejemplo perfecto de cuando sí es necesario hacerlo. Aun así, aun es preciso tener cuidado.

Ese cuidado se puede poner en práctica utilizando las propiedades de Variant para forzar a tu código para que trate el dato como un tipo determinado.

6) Cambia el código del evento Opening en Window1 por este:

```

Var first, second, third As Variant
first = 123
second = "456"
third = first.IntegerValue + second.IntegerValue
MessageBox(third)

```

La propiedad IntegerValue de Variant fuerza a que el ordenador trate el dato como numérico en vez de hacerlo como string. Variant también tiene una propiedad StringValue, así como BooleanValue, DataValue y otras.

Antes de que puedas usar un objeto guardado en una variante, debes indicarle al ordenador de qué tipo de objeto se trata. Volviendo al ejemplo StudentInformation, debes indicar al ordenador que la variante almacenada en el CellTag es un objeto Course. Antes de hacerlo, has de asegurarte de que el objeto es, de hecho, un Course.

7) En el proyecto StudentInformation, añade este código al evento DoublePressed de CourseBox:

```

Var c As Course
If Me.SelectedRowIndex <> -1 Then
    If Me.CellTagAt(Me.SelectedRowIndex, 0) IsA Course Then
        c = Course(Me.CellTagAt(Me.SelectedRowIndex, 0))
        MessageBox(c.Instructor)
    End If
End If

```

Este código puede parecer confuso a simple vista, pero es más fácil cuando se descompone en partes. En primer lugar creas una variable, C, que utilizarás para acceder al objeto Course. En segundo lugar, verificas que el usuario ha seleccionado una fila válida en el ListBox comprobando su SelectedRowIndex.

En la siguiente línea se usa la función IsA para comprobar si la variante (el CellTag en la Columna 0 de la Row/Fila seleccionada en el ListBox) es de hecho un objeto Course. IsA devuelve un valor booleano: True si la variante es realmente de ese tipo de dato y False si no es así.

A continuación, el código hace un proceso conocido como casting, lo que significa que le está diciendo al ordenador que el CellTag en cuestión debería tratarse como un objeto Course, almacenado en la variable que has llamado C.

Tras dicha línea, C es un objeto Course que puedes utilizar como tal. En el ejemplo anterior se muestra el Instructor del curso al usuario mediante un mensaje.

8) Ejecuta el proyecto.

Pruébalo. Cuando haces doble clic en un Curso en el ListBox, deberías ver un mensaje mostrando el nombre del Instructor.

9) Sal de la aplicación.

8.5 Métodos y Funciones en Clases

En el Capítulo Cuatro aprendiste sobre métodos y funciones. En dicho capítulo creaste dichos métodos y funciones como parte de la ventana. También puedes crear métodos y funciones como parte de tus propias clases.

- 1) **Vuelve al proyecto StudentInformation.**
- 2) **Con tu clase Course seleccionada en el Navegador, selecciona la opción Method en el menú Insert. Llama al método "DisplayCourseInfo".**
- 3) **Añade este código en DisplayCourseInfo:**

```
Var info As String
info = "Course Title: " + Self.Title + " ("
info = info + Self.Subject + ")" + EndOfLine
info = info + "Instructor: " &
    + Self.Instructor + EndOfLine
info = info + "Location: " + Me.Room
MessageBox(info)
```

Este método simplemente reunirá alguna información sobre el curso y la mostrará al usuario en un mensaje.

- 4) **En el evento DoublePressed de CourseBox, cambia esta línea:**

```
MessageBox(c.Instructor)
```

a esta:

```
c.DisplayCourseInfo
```

- 5) **Ejecuta el proyecto.**
- 6) **Haz doble clic en el nombre del curso en el ListBox para que muestre información sobre dicho curso en un mensaje.**
- 7) **Sal de la aplicación.**

Los métodos y las funciones de una clase no siempre tienen por qué mostrar información en un mensaje. Pueden ser tan sencillos o complejos como precisen que sean. También pueden tomar parámetros, tal y como en cualquier otro método o función. Los parámetros recibidos pueden ser de cualquier tipo de dato, tanto si es de entre los incluidos en Xojo o bien un tipo de dato personalizado, como por ejemplo la clase Student creada anteriormente.

- 8) **Añade una propiedad a tu clase Course: EnrolledStudents() As Student.**

Este es un array con los estudiantes actualmente apuntados al Curso.

- 9) **Añade un método a tu clase Course llamado "EnrollStudent".**

Este método recibe un parámetro: s As Student. El trabajo de este método es el de añadir el estudiante proporcionado al array EnrolledStudents.

10) Añade esta línea de código a EnrollStudent:

```
Me.EnrolledStudents.AddRow(s)
```

11) Añade ahora un método a tu clase Student llamado “Constructor”.

Hay dos nombres especiales para los métodos y funciones: Constructor y Destructor. Si una clase tiene un método llamado Constructor, dicho método se ejecutará tan pronto como se instancie la clase. En este caso, tan pronto como crees un objeto Student con el operador New, se ejecutará el método Constructor. El Destructor es similar, pero se ejecuta cuando se destruye el objeto en vez de cuando es creado. El constructor puede recibir parámetros.

12) Proporciona a tu Constructor dos parámetros: “fName As String” y “lName as String”.

Recuerda que los parámetros han de estar separados entre sí mediante comas. Estos dos parámetros se utilizarán para definir las propiedades FirstName y LastName del Student.

13) Añade este código al Constructor:

```
Self.FirstName = fName  
Self.LastName = lName
```

Ahora tienes una forma de crear objetos Student y un modo de añadir dichos estudiantes a un curso. Hay dos piezas críticas que aún faltan: en primer lugar un modo de ver los estudiantes apuntados a un curso; y en segundo lugar un mecanismo para que el usuario pueda añadir estudiantes a un curso (el método existe, pero no está disponible para el usuario).

14) Añade otro ListBox a Window1 y llámalo “StudentBox”.

15) Añade dos TextField llamados “FirstNameField” y “LastNameField”.

16) Añade un Button llamado “EnrollButton”.

El usuario introducirá el nombre y apellido en los TextField, luego usará el PushButton para añadir dicho estudiante al curso seleccionado en el otro ListBox.

17) Define la propiedad Enabled de EnrollButton a OFF en el Inspector.

18) Añade este código al evento SelectionChanged de CourseBox.

```
EnrollButton.Enabled = (Me.SelectedRowIndex <> -1)
```

Sólo debería de poder hacerse clic en EnrollButton cuando está seleccionado un curso. En un proyecto de ejemplo anterior utilizaste una sentencia de varias líneas para activar y desactivar un PushButton en función de lo que estaba seleccionado. Esta es una forma más simple y corta de hacerlo.

19) Añade este código justo tras la línea recién introducida:

```
Var theCourse As Course  
StudentBox.RemoveAllRows  
If Me.CellTagAt(Me.SelectedRowIndex, 0) =  
    IsA Course Then  
    theCourse = Course(Me.CellTagAt(  
        (Me.SelectedRowIndex, 0))  
    For Each s As Student In
```

```

        theCourse.EnrolledStudents
        StudentBox.AddRow ↵
            (s.LastName + ", " + s.FirstName)
    Next
End If

```

Este código mostrará los estudiantes de un curso. Crea una variable que contiene el objeto Course, obtiene el objeto Course de la CellTag y recorre los estudiantes del curso listándolos en StudentBox.

20) Añade este código al evento Pressed de EnrollButton:

```

Var theStudent As Student
Var theCourse As Course
Var newRow As String
theStudent = New Student(FirstNameField.Value, ↵
    LastNameField.Value)
If CourseBox.CellTagAt(CourseBox.↵
    SelectedRowIndex, 0) IsA Course Then
    theCourse = Course(CourseBox.CellTagAt( ↵
        CourseBox.SelectedRowIndex, 0))
    theCourse.EnrollStudent(theStudent)
    newRow = theStudent.LastName + ", "
    newRow = newRow + theStudent.FirstName
    StudentBox.AddRow(newRow)
End If

```

Este código agrega al Student a un Course. También añade el nombre del estudiante a StudentBox. Presta especial atención a la cuarta línea, en la que se saca provecho del Constructor a la hora de crear un nuevo Student con el nombre y su apellido.

21) Ejecuta el proyecto.

22) **Selecciona un curso y añade algunos estudiantes. Selecciona diferentes cursos y observa que el listado de estudiantes se actualiza cada vez que seleccionas un curso.**

23) **Sal de la aplicación.**

24) **Añade un nuevo método llamado “FullName” a tu clase Student. Este no recibirá parámetros y devolverá una string. Este es su código:**

```
Return Self.LastName + ", " + Self.FirstName
```

En el anterior código había momentos en los que era necesario obtener el apellido del estudiante, seguido por una coma y seguido por el nombre del estudiante. Dado que se utilizaba este código más de una vez, es un buen candidato para convertirlo en una función.

25) **Vuelve al proyecto StudentInformation y utiliza este método para abreviar tu código en el evento SelectionChanged del CourseBox y en el evento Pressed de EnrollButton.**

8.6 Módulos

Justo ahora, tu proyecto tiene un array con los cursos disponibles. Este array está almacenado en Window1. No hay nada malo en esta aproximación, pero habrá momentos en los que esto te limitará lo que puedas hacer. Por ejemplo, si la interfaz de tu aplicación requiere más ventanas, estas nuevas ventanas no tendrán acceso al listado de cursos.

Estas son ocasiones en las que es apropiado guardar la información en una variable global. Una variable global es una variable que siempre está disponible para todos los objetos, ventanas y métodos.

Las variables globales se suelen guardar en Módulos (Modules, en inglés). Puede resultar tentador el pensar que un Módulo es similar a una clase, pero hay algunas diferencias importantes. Mientras que un módulo puede contener propiedades, métodos y funciones tal y como es el caso en una clase; nunca es necesario instanciar un módulo con el operador New. En otras palabras, un módulo siempre existe en tu aplicación.

- 1) **En tu proyecto StudentInformation, añade un módulo (Module) seleccionando Module en el menú Insert mientras que la clase App está seleccionada en el panel Contents. Nombra este módulo como “Globals”.**
- 2) **Añade una propiedad al módulo: MyCourses() As Course.**

- 3) **En Window1, borra la propiedad MyCourses().**

- 4) **Ejecuta el proyecto.**

Funcionará de forma idéntica. La diferencia es que el listado de cursos está disponible para toda la aplicación y no sólo para Window1.

- 5) **Sal de la aplicación.**

- 6) **Añade una nueva ventana al proyecto seleccionando Window en el menú Insert. Deja el nombre de Window2.**

- 7) **Añade un ListBox a Window2 llamado “OtherCourseBox”. Añade este código a su evento Opening:**

```
OtherCourseBox.RemoveAllRows
For Each c As Course In MyCourses
    OtherCourseBox.AddRow(c.Title)
    OtherCourseBox.CellTagAt( 0,
        OtherCourseBox.LastAddedRowIndex, 0) = c
Next
```

- 8) **Añade un PushButton a Window1 y añade esto en Pressed:**

```
Window2.Show
```

- 9) **Ejecuta el proyecto**

- 10) **Haz clic en el PushButton para abrir Window2.**

Debería verse el mismo listado de cursos en ambas ventanas. Esto es porque ambas toman los datos del mismo array en tu módulo Globals.

- 11) **Sal de la aplicación.**

Entrada y Salida

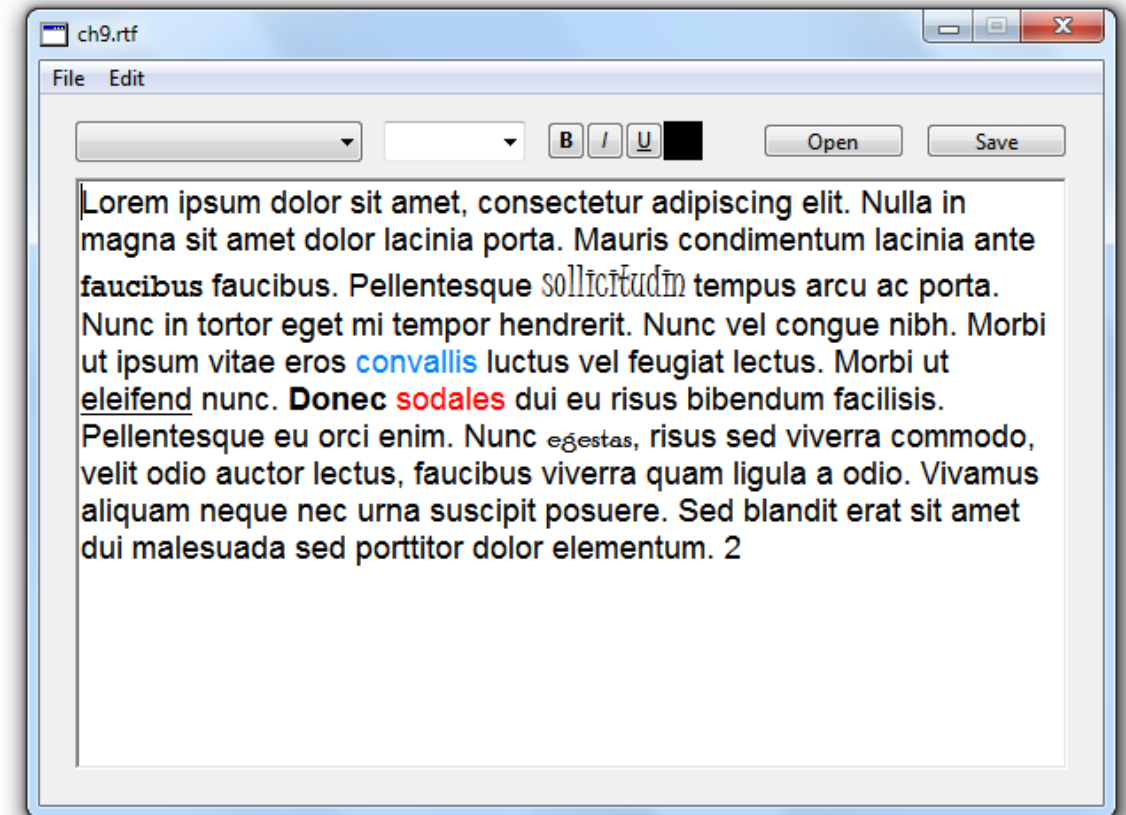
CONTENIDOS

1. **Introducción al Capítulo**
2. **Tipos de Archivos**
3. **Trabajar con Archivos**
4. **Diálogos Abrir/Guardar**
5. **Abrir Archivos**
6. **Crear y Guardar Archivos**
7. **Trabajando con Archivos**

9.1 Introducción al Capítulo

Las apps creadas hasta ahora en el libro tienen un fallo común fatal. No es tu fallo, simplemente se trata de algo que no has aprendido como hacer todavía. Todos los proyectos que has creado se “resetean a sí mismos” cada vez que los ejecutas. No almacenan ningún dato y no recuerdan nada sobre la última vez que los ejecutaste. Incluso el pequeño editor de textos con estilo creado unos capítulos atrás no puede guardar o abrir archivos.

En este capítulo crearás un editor de texto con estilos que abre, edita, y guarda archivos. Tu app también podrá cambiar la fuente, tamaño, estilo y color del texto. Tu app será parecida a esta:



Si te estás preguntando qué es todo este “lorem ipsum” sin sentido, es lo que se conoce como Latin simulado. No tienen ningún significado en particular, pero proporciona una buena distribución de las letras, prácticamente idéntica a la del Español; de modo que resulta útil para sopesar como queda un diseño sin la necesidad de tener que incluir texto “real”, que pudiese distraer al visor sobre el diseño propiamente dicho. Se utiliza con frecuencia en el prototipado de interfaces.

9.2 Tipos de Archivos

Antes de que empieces a escribir código necesitas una buena comprensión de lo que es un archivo. De forma resumida, un archivo es una pieza única de datos en tu ordenador. Puede contener datos de una imagen, un artículo, una canción o una película; incluso de una colección de ajustes. Cuando descargas una canción de iTunes o Amazon, se trata de un archivo. Cuando escribes un informe en un Word y lo guardas, se trata de un archivo.



Prácticamente todos los archivos tienen un tipo. El tipo de un archivo indica qué tipo de datos contiene. Por ejemplo, un archivo de texto contiene texto sin estilos. Un archivo JPEG

contiene una imagen. Un archivo MP3 contiene audio. Los tipos de archivo (File, en inglés) son muy específicos: en vez de un tipo de archivo de “imagen” estos son JPEG, PNG, GIF, BMP u otros. Cada uno de estos puede contener una imagen, incluso la misma imagen, pero tu ordenador necesita saber como abrir y leer cada uno de ellos de forma diferente. Por lo tanto, un tipo de archivo no sólo indica al ordenador qué tipo de dato encontrará en el archivo, sino también como leerlo.

Puedes ayudar a que tu aplicación busque un tipo de archivos determinado utilizando el File Type Group (Grupo de Tipos de Archivo). Para añadir un File Type Group a tu proyecto, selecciona File Type Group en el menú Insert. Da un nombre significativo a tu File Type Group, como por ejemplo PictureTypes (o cualquier otro que describa los archivos especificados). Haz clic en el botón de la izquierda de la barra de comandos en el Editor de Grupo de Tipos de Archivo para acceder a un listado con los tipos de archivo comunes. Puedes elegir algo como “image/png” para crear un File Type que utilice archivos PNG. Cuando lo selecciones, se añadirá a tu File Type Group con todas sus propiedades agregadas por ti.

Verás como utilizar el File Type Group para filtrar determinados tipos de archivos en la sección 9.4.

9.3 Trabajar con Archivos

En Xojo, un archivo está representado por una clase llamada `FolderItem`. Dicho nombre puede resultar confuso, y muchas personas asumen inicialmente que un `FolderItem` puede representar únicamente una carpeta; pero un `FolderItem` puede representar cualquier elemento que esté contenido en una carpeta, tanto si es un archivo como si es otra carpeta.

Un `FolderItem`, como puedes imaginar, tiene varias propiedades y métodos. Algunas de las propiedades más usadas son `Name` (una cadena), su `CreationDateTime` (un `DateTime` que indica la fecha y hora de creación del archivo), su `ModificationDateTime` (un `DateTime` que indica la última vez que se ha modificado el archivo), `IsWriteable` (un booleano que indica si puedes hacer cambios en el archivo), y su `Length` (un entero que indica el tamaño del archivo en disco, en bytes). La mayoría de estas propiedades son autoexplicativas, de modo que te animo a que experimentes luego con dichas propiedades.

El `FolderItem` también tiene algunas propiedades cuyo uso y significado puede que no sea aparente. Una puede ser `IsFolder`. `IsFolder` es un booleano que te indica si el `FolderItem` es una carpeta (también llamado directorio). Puede resultar muy útil saberlo: no querrás intentar abrir una carpeta pensando que se trata de una imagen.

Otra propiedad que puede resultar curiosa es la propiedad `Exists`. Esta propiedad, de forma bastante apropiada, te indica si un `FolderItem` existe o no en tu disco. Esto puede resultar raro. Después de todo, ¿cómo podrías tener un `FolderItem` que apunte a un archivo que no existe? La respuesta corta es que se trata de algo bastante común. De hecho es el único modo de crear un nuevo archivo o carpeta que aun no exista. Esto será más claro un poco más adelante en el capítulo.



Si el `FolderItem` con el que estás trabajando es una carpeta y no un archivo, entonces la propiedad `Count` te indicará cuantos elementos contiene. Este número puede incluir otras carpetas, pero no los elementos contenidos en dichas subcarpetas.

Relacionado con Count, el FolderItem tiene una función llamada Child. Child recibe el nombre de un archivo o carpeta y devuelve un FolderItem que representa un archivo o carpeta. De modo alternativo, ChildAt es una función que recibe un entero indicando el índice del archivo o carpeta y devuelve un FolderItem. Si te resulta menos confuso, puedes pensar en Item como en un array de FolderItems en vez de un método.

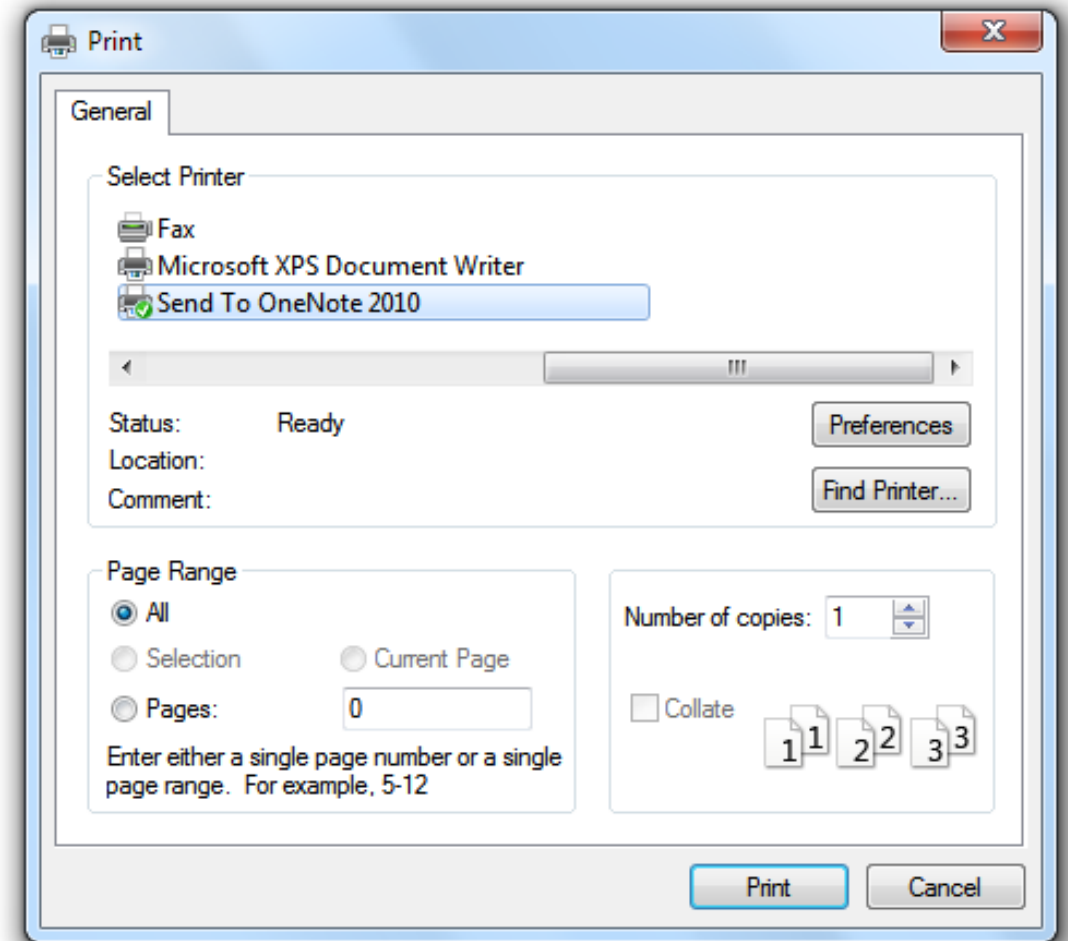
Advierte también la propiedad Parent. El Parent es el FolderItem que contiene tu FolderItem. De modo que si tienes un archivo llamado MyXoyoProject y está almacenado en una carpeta llamada My Projects, entonces la propiedad Parent de MyXoyoProject te dará un FolderItem que representa a la carpeta My Projects.

9.4 Diálogos de Abrir/Guardar

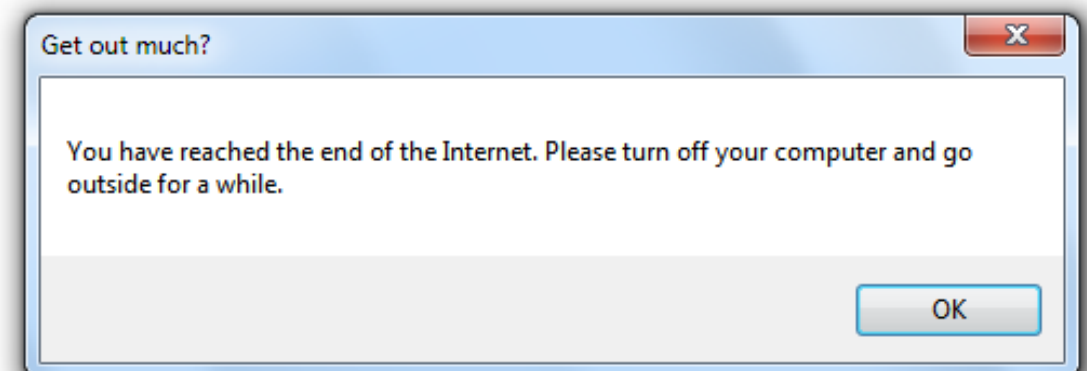
Toda esta charla sobre FolderItems está genial, pero probablemente te estés preguntando sobre cómo puedes indicarle a Xojo con qué archivos quieres trabajar. Esto se gestiona mejor mediante la clase FolderItemDialog.

Pero antes de que vemos los diálogos, puede que esté bien explicar qué es un diálogo. En términos sencillos, un diálogo es una ventana mínima que puede recuperar información de un usuario o proporcionar información al usuario.

Probablemente ya habrás visto un diálogo que recupera información del usuario. Un ejemplo clásico es el diálogo de impresión en el que se pide que indiques la impresora y posiblemente debas de seleccionar otros ajustes:



Un diálogo que proporciona información al usuario puede que se parezca más a este:



Hay tres versiones especiales de la clase FolderItemDialog que puedes usar: OpenFileDialog, SaveFileDialog y SelectFolderDialog. Cuando quieras preguntar al usuario que abra un archivo, utiliza OpenFileDialog. Cuando quieras que el usuario guarde un archivo, usa SaveFileDialog. Por último, cuando quieras que el usuario elija una carpeta, utiliza SelectFolderDialog.

(También hay formas en las que puedes abrir y guardar archivos concretos sin la ayuda del usuario. Esto se tratará en al final del capítulo.)

Comencemos con SelectFolderDialog.

- 1) **Crea un nuevo proyecto Xojo y guárdalo como “Dialogs”.**
- 2) **Añade un Botón y un ListBox a Window1.**
- 3) **Nombra el ListBox “FileBox” y define “Select Folder” como Caption del Botón.**
- 4) **Añade este código al evento Action del Botón:**

```
Var myFolder As FolderItem

Var d As SelectFolderDialog
d = New SelectFolderDialog
myFolder = d.ShowModal
If myFolder <> Nil Then
    FileBox.AddRow("Name: " + myFolder.Name)
    FileBox.AddRow("Size: " + ↵
        myFolder.Length.ToString + " bytes")
    FileBox.AddRow("Items: " + ↵
```

```
        myFolder.Count.ToString)
    FileBox.AddRow("Parent: " + ↵
        + myFolder.Parent.Name)
End If
```

Lo primero que hace este código es crear dos variables: una para SelectFolderDialog y otra para el FolderItem. Dado que SelectFolderDialog es una clase, necesita ser instanciada con el operador New.

ShowModal es una función disponible en todas las clases de diálogo para archivos. Muestra una ventana modal al usuario (una ventana modal es una que bloquea el resto de la aplicación hasta que el usuario la cierre, ya sea porque haya seleccionado un item o cancelado la operación; lo cual contrasta con un diálogo estándar donde aun se permite a parte o a toda la aplicación). La función devolverá Nil si el usuario a pulsado el botón cancel, siendo esto lo que comprobarás para saber si myFolder es Nil antes de continuar, o bien devolverá un FolderItem representando la carpeta seleccionada por el usuario.

Una vez que el código verifica que el FolderItem es válido, este añade unas cuantas filas a FileBox mostrando algunas propiedades de la carpeta seleccionada. Observa la sintaxis utilizada para mostrar el Nombre del Parent correspondiente al FolderItem: myFolder.Parent.Name. Puesto que Parent también es un FolderItem, puedes acceder a sus propiedades como harías con el FolderItem seleccionado inicialmente.

- 5) **Ejecuta el proyecto.**
- 6) **Haz clic en el PushButton, selecciona una carpeta, y examina las propiedades en FileBox:**

El nombre y su tamaño deberían de ser evidentes.

El Parent se ha tratado unas líneas más arriba.

Observa el número de elementos que contiene el FolderItem. Verifica ahora por ti mismo cuántos elementos tiene. El número listado en FileBox es uno más que la cantidad que puedes ver en la carpeta. Esto es así porque el número cero es el FolderItem propiamente dicho. Esto puede provocar algo de confusión, de modo que conviene recordarlo y ser consciente de ello cuando escribas código que trate con carpetas.

7) Sal de la aplicación.

Para abrir un archivo, utiliza la clase OpenFileDialog. La añadirás al proyecto Dialogs a continuación.

8) Añade otro Botón a Window1. Define “Open File” como su Caption. Añade este código a su evento Pressed:

```
Var myFile As FolderItem
Var d As OpenFileDialog
d = New OpenFileDialog
myFile = d.ShowDialog
If myFile <> Nil Then
    FileBox.AddRow("Name: " + myFile.Name)
    FileBox.AddRow("Size: " + myFile.Length.ToString + " bytes")
    FileBox.AddRow("Parent: " + myFile.Parent.Name)
    Var theDate As String
    theDate = myFile.CreationDateTime.ToString
    FileBox.AddRow("Created on: " + theDate)
    theDate = myFile.ModificationDateTime.ToString
    FileBox.AddRow("Last modified on: " + theDate)
End If
```

9) Ejecuta el proyecto.

10) Haz clic en el Botón, selecciona un archivo, y examina las propiedades en FileBox.

En esta ocasión has añadido su CreationDateTime y ModificationDateTime. Dado que ambos son objetos DateTime, puedes tratarlos como cualquier otro DateTime, usando por tanto ToString y el estilo de formato Short. En esta ocasión también has eliminado la propiedad Count del FolderItem, dado que no se trata de una carpeta.

11) Sal de la aplicación.

Cuando se abren archivos es probable que no quieras que el usuario elija tipos de archivos inapropiados. Por ejemplo, si estás creando un editor de imágenes probablemente no querrás que el usuario abra un documento de Word o un archivo XML. Aquí es donde File Type Groups entra en juego, usando la propiedad Filter para el FolderItemDialog.

1) Crea un nuevo proyecto Xojo y guárdalo como “FileTypes”.

2) Crea un File Type Group llamado “ImageFiles” (selecciona File Type Group en el menú Insert).

3) Añade estos FileTypes al set: image/jpeg, image/png

4) Añade un Botón a Window1. Añade este código a su evento Pressed:

```
Var f As FolderItem
Var d As OpenFileDialog
d = New OpenFileDialog
d.Filter = ImageFiles.All
```



```
f = d.ShowDialog()  
If f <> Nil Then  
    MessageBox(f.Name)  
End If
```

5) Ejecuta el proyecto.

6) Haz clic en el Botón y navega por el ordenador en busca de archivos para abrir.

Tu aplicación debería de permitirte elegir sólo archivos PNG y JPG.

7) Sal de la aplicación.

También puedes usar el File Type Group para permitir que el usuario elija un formato de archivo a la hora de guardar un archivo.

8) Añade otro Botón con este código en su evento Pressed:

```
Var f As FolderItem  
Var d As SaveFileDialog  
d = New SaveFileDialog  
d.Filter = ImageFiles.All  
MessageBox(f.name)
```

9) Sal de la aplicación.

9.5 Abrir Archivos

Una cosa es mirar datos sobre los archivos, pero probablemente sea más productivo mostrar y posiblemente editar los contenidos de dichos archivos. Es posible leer los datos en la aplicación de formas muy distintas, en función del tipo de archivo. Un archivo MP3 será muy distinto de un archivo de texto plano el cual, a su vez, se leerá de forma muy distinta al archivo de una base de datos (sobre lo cual aprenderás en el Capítulo Doce).

Prácticamente toda aplicación en la que puedas pensar tendrá que leer datos, escribir datos, o ambas cosas.

Un stream o flujo es un modo común de leer o escribir datos. De hecho, muchos otros lenguajes, como Java, utilizan streams para ello.

Una forma común de leer los datos de un archivo es el `BinaryStream`. Puede que estés familiarizado con los servicios de música en streaming, como Spotify, Apple Music o Pandora. Estos servicios envían parte de la canción a tu ordenador, esperan un poco, y luego envían más datos de la canción en vez de enviarlo todo a la vez. El `BinaryStream`, y de hecho todos los streams de archivo, funcionan de una forma muy similar. Una vez que se ha seleccionado un archivo, se utiliza `BinaryStream` para extraer datos de dicho archivo en pequeños fragmentos, cada uno de los cuales es procesado a medida que se lee en la

aplicación. Cuando se han leído todos los datos se presenta al usuario.

1) **Crea un nuevo proyecto Xojo Desktop y guárdalo como “Streams”.**

2) **Añade un Botón y una `TextArea` a `Window1`.**

No te preocupes mucho sobre sus tamaños y posiciones, pero asegúrate de que el `TextArea` sea tan grande como puedas en la ventana.

3) **Añade este código al evento `Pressed` del Botón:**

```
Var myFile As FolderItem
Var d As OpenFileDialog
Var b As BinaryStream
d = New OpenFileDialog
myFile = d.ShowModal
If myFile <> Nil Then
    b = BinaryStream.Open(myFile)
    TextArea1.Text = b.Read(b.Length)
    b.Close
End If
```

Este código permite que el usuario seleccione un archivo y luego muestre sus contenidos en el `TextArea` usando un `BinaryStream`. La sintaxis para ello es un poco distinta de lo que estás acostumbrado.

Las primeras líneas de código deberían de resultarte familiares: crear unas cuantas variables, instanciar un nuevo `OpenFileDialog`, y seleccionar un archivo. Pero una vez que se ha seleccionado el archivo, verás la siguiente línea:

```
b = BinaryStream.Open(myFile)
```

BinaryStream.Open es un método compartido. Por lo general, los métodos de una clase pueden ejecutarse una vez que se ha creado una instancia de dicha clase con el operador New; pero un método compartido nos permite ejecutar dicho método en cualquier momento. Esto se explica con más detalle en el Capítulo Trece. Por ahora, advierte simplemente la diferencia en la sintaxis.

Una vez que se ha instanciado el BinaryStream, este lee los datos de myFile y los muestra en el TextArea. El método Read del BinaryStream proporciona una porción de los contenidos del archivo a tu aplicación. El tamaño de dicha porción depende de ti. El método Read toma un entero que indica cuántos bytes ha de leer. En este caso has dicho que se lea todo de una vez al indicar el Length (longitud o tamaño) del BinaryStream (la propiedad Length del BinaryStream es como la propiedad Length del FolderItem).

4) Ejecuta el proyecto.

5) Haz clic en el PushButton y abre cualquier archivo.

Observa lo que aparece en el TextArea. Si has elegido un archivo de texto plano, podrás leer ahora sus contenidos. Si has elegido cualquier otra cosa, como un MP3 o una imagen, entonces no podrás sacar ningún sentido de los datos mostrados.

6) Sal de la aplicación.

Este proyecto ilustra algo muy importante sobre la mayoría de los archivos de tu ordenador: no son legibles por humanos. Un archivo legible por humanos es uno que tiene sentido para ti simplemente viendo sus contenidos sin tratar. Hay algunas excepciones, como puedan ser los archivos XML, HTML o bien los archivos de texto plano; pero en la mayoría de los casos, los

archivos de un ordenador sólo pueden ser leídos por una app diseñada para leer ese tipo de archivo.

Un formato de archivo personalizado es algo así como un mapa o clave que indica al ordenador qué tipo de datos están almacenados y en qué parte del archivo.

Afortunadamente, Xojo incluye algunas funciones que facilitan la apertura de los tipos de archivos más comunes.

Por ejemplo, así puedes abrir un archivo de imagen.

1) Crea un nuevo proyecto Xojo Desktop y guárdalo como “OpenPic”.

2) Añade un Botón y un Canvas a Window1.

El Canvas debería cubrir la mayor superficie posible de la ventana. El botón pedirá al usuario que seleccione una imagen, siendo mostrada posteriormente en el Canvas.

3) Añade este código al evento Pressed del Botón:

```
Var myPic As Picture
Var myFile As FolderItem
Var d As OpenFileDialog
d = New OpenFileDialog
myFile = d.ShowModal
If myFile <> Nil Then
    myPic = Picture.Open(myFile)
    If myPic <> Nil Then
        Canvas1.Backdrop = myPic
    End If
```

End If

La clase Picture, como has visto en el anterior BinaryStream, tiene un método compartido llamado Open que toma un FolderItem como parámetro. Nuevamente, dado que es un método compartido puedes utilizarlo en cualquier momento, sin necesidad de instanciar un objeto primero.

El código define a continuación la propiedad Backdrop del Canvas. El Backdrop es un Picture como fondo de un Canvas. El siguiente capítulo tratará los gráficos, imágenes y el Canvas con mayor profundidad.

Sin embargo, antes de definir la propiedad Backdrop, debes comprobarlo en primer lugar para asegurarte de que el Picture no sea Nil, tal y como haces con el FolderItem.

4) Ejecuta el proyecto.

5) Haz clic en el Botón y selecciona una imagen de tu ordenador.

Si es una imagen válida, el Canvas debería de mostrarla.

6) Sal de la aplicación.

En muchos casos, los archivos que necesitas abrir sólo contienen texto. Esto son fáciles de abrir y de leer utilizando la clase TextInputStream. TextInputStream te permite leer todo el texto de un archivo a la vez (usando la función ReadAll), leerlo línea a línea (usando la función ReadLine), o bien leer una cantidad determinada de caracteres cada vez (usando la función Read). Todas estas funciones devuelven un string. Depende de ti saber

qué hacer con dicha cadena, ya sea procesarla para mostrarla en un TextField, almacenarla en un Diccionario o bien realizar alguna otra tarea.

1) Crea un nuevo proyecto Xojo Desktop y guárdalo como “TextRead”.

2) Añade un Botón y una TextArea a Window1.

El TextArea debería completar la mayor cantidad de espacio posible de la ventana.

3) Añade este código en el evento Pressed del Botón:

```
Var myFile As FolderItem
Var d As OpenFileDialog
Var t As TextInputStream
d = New OpenFileDialog
myFile = d.ShowDialog
If myFile <> Nil Then
    t = TextInputStream.Open(myFile)
    TextArea1.Text = t.ReadAll
    t.Close
End If
```

Este ejemplo utiliza la función ReadAll del TextInputStream para leer todos los contenidos del archivo seleccionado.

4) Ejecuta el proyecto.

5) Haz clic en el Botón y selecciona un archivo de texto.

Los contenidos del archivo deberían de mostrarse en el TextArea.

6) Sal de la aplicación.

7) Cambia esta línea en el evento Pressed del Botón:

```
TextArea1.Text = t.ReadAll
```

por esta:

```
TextArea1.Text = t.ReadLine
```

8) Ejecuta el proyecto.

9) Haz clic en el PushButton y selecciona un archivo.

En esta ocasión sólo se mostrará la primera línea del archivo. Puede que te estés preguntando por qué querrías hacer algo como esto. ¿Qué utilidad tiene leer sólo la primera línea de un archivo? Puede utilizarse para una buena cantidad de cosas. Un ejemplo sería echar un vistazo rápido a los contenidos de un archivo; o puede que quieras mostrar los contenidos línea a línea en un ListBox.

10) Sal de la aplicación.

11) Borra el TextArea de Window1 y añade un ListBox.

12) En el evento Pressed del PushButton, cambia esta línea:

```
TextArea1.Text = t.ReadLine
```

Por esta:

```
ListBox1.AddRow(t.ReadLine)
```

13) Ejecuta el proyecto.

14) Haz clic en el PushButton y selecciona un archivo de texto.

¿Ves el problema? La app sólo ha cargado la primera línea del archivo en el ListBox porque le has pedido que lo haga sólo una vez.

15) Sal de la aplicación.

Para leer más de una línea necesitarías usar la función ReadLine más de una vez. Podrías repetir la línea, como por ejemplo:

```
ListBox1.AddRow(t.ReadLine)  
ListBox1.AddRow(t.ReadLine)  
ListBox1.AddRow(t.ReadLine)
```

Cada vez que ejecutas ReadLine, la app coge la siguiente línea. Pero esta aproximación tiene dos problemas. En primer lugar, no es código eficiente en absoluto. Las tareas repetitivas como esta podrían hacerse en un bucle. Esto nos lleva al segundo problema. ¿Cómo saber cuántas líneas tiene un archivo?

De forma resumida, continúas leyendo datos hasta que te quedes sin datos que leer. Esto ocurre cuando alcanzas el final del archivo o EndOfLine. EndOfLine es un booleano: cuando es False aun quedan datos por leer; cuando es True, ya no queda más datos. Por tanto, la solución a este problema es usar el bucle While...Wend en combinación con la propiedad EndOfLine del TextInputStream.

16) Cambia el código del evento Action en el Botón por este:

```
Var myFile As FolderItem
Var d As OpenFileDialog
Var t As TextInputStream
d = New OpenFileDialog
myFile = d.ShowDialog
If myFile <> Nil Then
    t = TextInputStream.Open(myFile)
    While Not t.EndOfFile
        Listbox1.AddRow(t.ReadLine)
    Wend
    t.Close
End If
```

Recuerda de los capítulos anteriores que el operador Not indica a tu app que lo que quieres usar es lo contrario del valor booleano al que te estás refiriendo. En este caso, el bucle continuará siempre que EndOfFile no sea True.

17) Ejecuta el proyecto.

18) Haz clic en el Botón y selecciona un archivo de texto.

Cada línea del archivo de texto debería mostrarse en una línea diferente del ListBox.

19) Sal de la aplicación.

9.6 Crear y Guardar Archivos

Leer archivos está bien, pero probablemente querrás crearlos también. En esta sección aprenderás a usar `TextOutputStream` que, como su nombre implica, es esencialmente lo opuesto de `TextInputStream`.

Recuerda que `TextInputStream` tiene funciones para leer todo el texto, leer sólo una línea de texto y leer una determinada cantidad de caracteres. `TextOutputStream` tiene unos métodos similares para escribir texto, aunque sólo dos de ellos: `Write` y `WriteLine`.

El método `WriteLine` toma una string como parámetro. Escribe dicha cadena sobre el archivo seleccionado y añade un carácter `EndOfLine` (que puede ser un salto de línea, retorno de línea o una combinación de estos, en función del tipo de ordenador que estés usando). Si prefieres utilizar un delimitador diferente puedes definirlo en la propiedad `Delimiter` del `TextOutputStream`.

La función `Write` también toma un parámetro, también un string. Escribe dicha string sobre el archivo seleccionado pero no añade delimitadores o caracteres de `EndOfLine`.

Este es un ejemplo que pone en práctica el método `Write`.

1) **Crea un nuevo proyecto Xojo Desktop y guárdalo como “TextOut”.**

2) **Añade un Botón y una `TextArea` a `Window1`.**

Cambia el tamaño del `TextArea` para que ocupe la mayor área posible de `Window1`.

3) **Añade el siguiente código en el evento `Pressed` del Botón:**

```
Var myFile As FolderItem
Var d As SaveFileDialog
Var t As TextOutputStream
d = New SaveFileDialog
myFile = d.ShowModal
If myFile <> Nil Then
    t = TextOutputStream.Create(myFile)
    t.Write(TextArea1.Value)
    t.Close
End If
```

Las primeras líneas de código deberían de resultarte familiares, aunque esta es la primera vez que veas el diálogo `SaveAsDialog`. Este diálogo es similar a `OpenFileDialog`, pero te permite *crear* archivos en vez de *abrir* archivos. Por tanto, aunque `myFile` no es `Nil`, el archivo que respresenta aun no existe en tu ordenador. El archivo se escribe a disco cuando se llama a la función `Create` del `TextOutputStream`. `Create` es un método compartido (`Shared Method`), tal y como has visto en anteriores ejemplos.

Una vez que se ha creado el archivo ya tienes un `TextOutputStream` con el que trabajar; entonces puedes añadir datos al archivo utilizando el método `Write`. En este ejemplo estás añadiendo todo el texto del `TextArea` de una sola vez al archivo.

El método Close del `TextOutputStream` indica a tu app que has terminado de escribir datos en el archivo, de modo que es seguro utilizarlo en otras aplicaciones.

4) Ejecuta el proyecto.

5) Añade algo de texto al `TextArea` y haz clic en el Botón.

Se te pedirá que guardes el archivo. Dale un nombre y guárdalo.

6) Sal de la aplicación.

7) Encuentra un nuevo archivo y ábrelo en otro editor de textos.

Algunos editores de textos pueden ser `NotePad` en Windows y `TextEdit` en macOS. Cuando abras el archivo verás los mismos contenidos introducidos en tu aplicación. ¡Enhorabuena! Ahora puedes intercambiar datos con el resto del mundo.

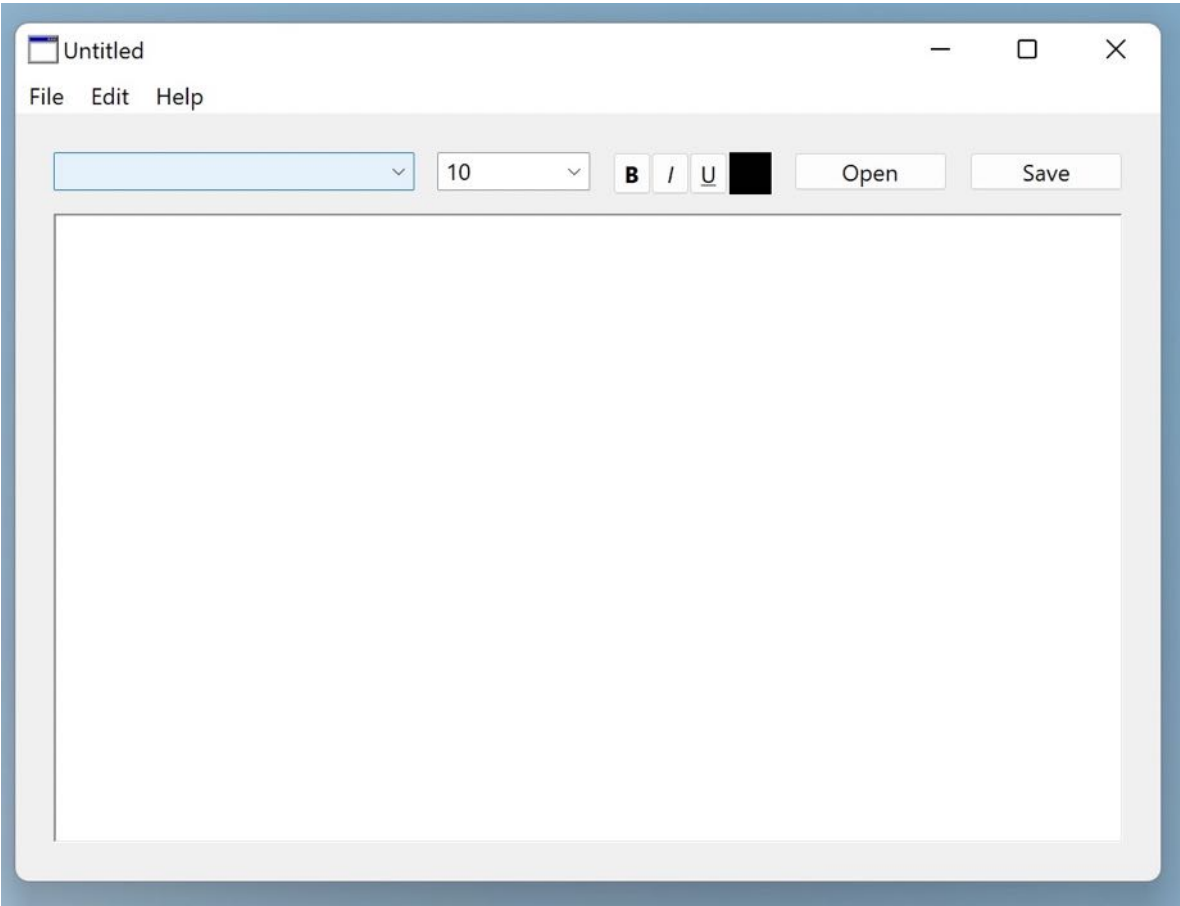
9.7 Trabajando con Archivos

El proyecto de ejemplo de este capítulo es un editor de textos con estilos. Aplicarás toda la información que has aprendido sobre abrir y guardar archivos, así como el uso de varios controles revisados en anteriores capítulos.

El texto con estilos es simplemente texto que permite el uso de diferentes fuentes, tamaños y estilos como negrita, subrayado y cursiva. Este proyecto usará un formato llamado RTF, o Formato de Texto Enriquecido (Rich Text Format, en inglés), para almacenar la información de estilo. RTF es un modo estándar y multiplataforma de almacenar texto con estilo, y los datos RTF pueden almacenarse como texto normal, de modo que puedes utilizar `TextInputStream` y `TextOutputStream` para abrir y guardar no sólo tus archivos, sino cualquier otro archivo RTF que puedas tener en tu ordenador.

1) **Crea un nuevo proyecto Desktop y guárdalo como “StyledTextEditor”.**

Este es un ejemplo del aspecto que tendrá la interfaz, siéntete libre de ser creativo con la tuya:



2) **Estos son los controles que necesitarás en Window1 y los nombres que utilizará el código de ejemplo:**

Nombre	Etiqueta	Control
FontMenu	N/A	PopupMenu
FontSizeField	N/A	ComboBox
BoldButton	B	BevelButton
ItalicButton	I	BevelButton
UnderlineButton	U	BevelButton
ColorButton	N/A	Rectangle
OpenButton	Open	Button

3) Con Window1 seleccionado, dirígete al Botón Insert y selecciona Property para añadir una nueva propiedad en Window1: TheFile As FolderItem.

Dado que el editor de texto necesitará recordar el archivo con el que está trabajando cuando llegue el momento de guardarlo, ha de ser una propiedad de la ventana y no algo que se crea y se destruye con cada evento Pressed del botón. Este es un ejemplo de ámbito (scope). Como se mencionó anteriormente, el ámbito indica hasta qué punto estará disponible la variable y quién más puede acceder a ella. Si creases MyFile en el evento Pressed de OpenButton, entonces la variable estaría “fuera de ámbito” una vez que se completase el evento; luego, cuando fuese el momento de guardar el archivo, SaveButton no sabría qué archivo usar sin preguntar de nuevo al usuario, lo cual podría resultar molesto para el usuario por no mencionar que sería susceptible de errores (los usuarios cometen fallos y tu código debería protegerles de ello tanto como sea posible). Al hacer que MyFile sea una propiedad de la ventana, puedes garantizar que no quedará fuera de ámbito.

4) Define el ButtonStyle de cada BevelButton a ToggleButtons.

Para añadir un toque visual a tu editor de textos, utiliza el Inspector para definir el estilo de cada BevelButton de modo que se corresponda a su propósito. Utiliza el icono de rueda dentada en la pestaña del Inspector para acceder a la propiedad Font y activar la propiedad Bold en BoldButton, la propiedad Italic en ItalicButton, y la propiedad Underline en UnderlineButton. Esto dará a tus usuarios una indicación visual clara del propósito de cada botón. Probablemente te hayas encontrado con alguna interfaz similar antes en un editor de textos o tratamiento de textos.

5) Añade este código en el evento Pressed de BoldButton:

```
EditingField.SelectionBold = Me.Value
```

SelectionBold es un booleano que indica si el texto seleccionado en el TextArea tiene el estilo negrita o no. Al ajustar SelectionBold en la propiedad Value del BevelButton (conmutando su estado), puedes asegurarte de que se aplica negrita al texto seleccionado cuando se pulsa el botón.

6) Añade este código al evento Pressed de ItalicButton:

```
EditingField.SelectionItalic = Me.Value
```

7) y este código al evento Pressed de UnderlineButton:

```
EditingField.SelectionUnderline = Me.Value
```

8) Añade este código en el eventoMouseDown del Rectangle:

```
Return True
```

En el caso de ColorButton se utiliza un Rectangle (Rectángulo). Este control no se ha tratado en el capítulo sobre los controles y eventos porque no hace mucho. Por ejemplo, no dispone de evento Pressed. Sin embargo, puedes utilizarlo como un botón. Define su propiedad FillColor a negro en el Inspector.

La línea “Return True” es, admitámoslo, algo extraño a primera vista. Para asegurarte de que el Rectángulo responda a un clic del ratón has de implementar sus eventos MouseDown y MouseUp. MouseUp es donde tiene lugar el clic, porque así es como funcionan los botones: la acción tiene lugar cuando se suelta el botón. Sin embargo, el evento MouseUp del rectángulo no se dispara salvo que se devuelva True en el evento MouseDown.

9) Añade este código en el evento MouseUp del Rectangle:

```
Var c As Color
If color.SelectedFromDialog=
(c, "Choose a Text Color") Then
    EditingField.SelectionTextColor = c
    Me.FillColor = c
End If
```

Este código utilizará la función Color.SelectFromDialog para pedir al usuario que seleccione un color. Color.SelectFromDialog devuelve True en el caso de que el usuario haya seleccionado un color y False si no lo ha hecho. Si el usuario elige un color, este código define el color del texto seleccionado con el color seleccionado por el usuario. También define la propiedad FillColor del Rectángulo propiamente dicho para que se corresponda con el del texto. Si el usuario cancela, y SelectColor devuelve False, entonces puedes ignorarlo.

10) Añade este código al evento Opening de FontMenu:

```
Var i, myFontCount As Integer
myFontCount = System.FontCount - 1
For i = 0 To myFontCount
    Me.AddRow(System.FontAt(i))
Next
```

El código de FontMenu debería de resultarte familiar, dado que trabajaste con un código parecido en los primeros capítulos. Este código simplemente itera sobre cada una de las fuentes instaladas añadiendo su nombre al PopupMenu.

11) Añade este código al evento SelectionChanged de FontMenu:

```
EditingField.SelectionFontName = Me.SelectedRowValue
```

Este código define la fuente del texto seleccionado a la elegida por el usuario, tras asegurarse de que el usuario a seleccionado una fila que no esté vacía.

12) Añade este código al evento TextChanged de FontSizeField:

```
EditingField.SelectionFontSize = Me.Text.ToInteger
```

Esta es una línea de código sencilla que coje el valor de un ComboBox, lo convierte en valor numérico y define el texto seleccionado del TextArea a dicho tamaño.

FontSizeFields es un ComboBox. Esto significa que el usuario puede utilizar cualquiera de las opciones pre-establecidas al tiempo de que también puede introducir cualquier otro tamaño de texto que desee. Para añadir valores a FontSizeField, haz clic en el icono de Lápiz que aparece cuando sitúas el apuntador del ratón sobre el control. Añade algunos valores en el diálogo que aparece y que desees que se muestren en tu menú. Puedes utilizar aquí cualquier valor que desees, si bien algunos tamaños de fuente habituales son 10, 11, 12, 14, 18, 24, 36 y 48. Para asignar otros valores, el usuario siempre puede introducir el tamaño que desee.

El ComboBox no tiene un evento Pressed. Cuando el usuario hace una selección o cambia un valor se dispara el evento TextChanged.

13) Ejecuta el proyecto.

Aun no puedes abrir o guardar documentos, pero puedes probar el editor. Introduce algo de texto en el TextArea y juega con los estilos, fuentes, tamaños y colores. Selecciona ahora algo de texto cuyo estilo ya hayas cambiado. ¿Ves el problema? El TextArea está respondiendo a los cambios de estilo, pero los botones de estilo, el menú de fuente y otros controles no están reflejando el estilo del texto seleccionado.

14) Sal de la aplicación.

15) Añade este código al evento SelectionChanged del TextArea:

```
Var i, fontListCount As Integer
ColorButton.FillColor = Me.SelectionTextColor
BoldButton.Value = Me.SelectionBold
ItalicButton.Value = Me.SelectionItalic
UnderlineButton.Value = Me.SelectionUnderline
FontSizeField.Text = Me.SelectionFontSize.ToString
fontListCount = FontMenu.LastRowIndex
For i = 0 To fontListCount
    If Me.SelectionFontName = "
        FontMenu.RowValueAt(i) Then
            FontMenu.SelectedRowIndex = i
            Exit
    End If
Next
```

El evento SelectionChanged se dispara cuando el usuario cambie su selección, tanto si es al hacer clic en una palabra, usando las teclas de cursor, al navegar por el texto o al seleccionar el texto. El menú fuente, el campo de tamaño de fuente, los botones de estilo y el botón de color debería cambiar ahora para corresponderse con lo que esté seleccionado.

La mayoría de este código simplemente funciona de forma inversa al modo en el que lo hacen los botones y menús, asegurándose de que estos elementos se correspondan con lo seleccionado. Observa que cambiar el texto mostrado en FontMenu es más complejo que simplemente ajustar la propiedad RowValueAt. Esta propiedad no puede definirse directamente, de modo que se precisa recorrer cada elemento en el menú y comprobar la correspondencia. Cuando encuentres el correspondiente, seleccionas el SelectedRowValue del PopupMenu y sales del bucle.

16) Ejecuta el proyecto de nuevo.

17) Introduce texto en el TextArea y juega con los estilos, fuentes, tamaños y colores.

En esta ocasión, los menús y los botones deberían cambiar cuando selecciones un texto; pero aún no puedes abrir o guardar documentos.

Observa que en macOS no todas las fuentes soportan negrita o cursiva. Una fuente diferente (como Arial) puede que no utilice todos los estilos.

18) Sal de la aplicación.

19) Añade este código al evento Pressed del evento OpenButton:

```
Var myFile As FolderItem
Var d As OpenFileDialog
Var textData As TextInputStream
Var rtf As StyledText
d = New OpenFileDialog
d.Title = "Select A File"
myFile = d.ShowDialog
If myFile <> Nil Then
    textData = TextInputStream.Open(myFile)
    EditingField.StyledText.RTFData =
```

```

        = textData.ReadAll
    textData.Close
    TheFile = myFile
    Self.Title = TheFile.Name
End If

```

Este código solicitará al usuario que seleccione un archivo y utiliza `TextInputStream` para obtener sus contenidos. Dado que los datos serán RTF, necesitarás leerlos en la propiedad `RTFData` de la propiedad `StyledText` del `TextArea` en vez de hacerlo directamente sobre la propiedad `Text`. Esto suena más confuso de lo que es realmente en la práctica.

Más allá de la incorporación para usar `RTFData`, este código debería ser muy similar al que ya has aprendido en este capítulo. Una cosa agradable desde el punto de vista de la interfaz es que también define el Título (Title) de la ventana con el Nombre del archivo editado. También ajusta la propiedad `TheFile` de la ventana al archivo seleccionado.

20) Añade este código al evento Pressed de SaveButton:

```

Var myFile As FolderItem
Var d As SaveFileDialog
Var textData As TextOutputStream
If TheFile = Nil Then
    d = New SaveFileDialog
    d.Title = "Save Your File"
    myFile = d.ShowModal
Else
    myFile = TheFile
End If
If myFile <> Nil Then
    textData = TextOutputStream.Create(myFile)
    textData.Write(EditingField.StyledText.RTFData)
    Self.Title = MyFile.Name

```

```
End If
```

Aquí, en el evento `Pressed` de `SaveButton`, las cosas no son tan sencillas. Si el usuario ya ha abierto un archivo guardarás los datos en el archivo utilizando la propiedad `TheFile` de la ventana. Sin embargo, en el caso de que el usuario haya empezado desde cero, tendrás que preguntar al usuario si quiere guardar el archivo y cómo quiere llamarlo. Una vez que esté hecha esa parte, es cuestión de escribir los datos en `RTFData` en el archivo.

Observa que cuando el archivo está guardado se actualiza el título de la ventana en el caso de que se haya creado un nuevo archivo.

21) Ejecuta el proyecto.

22) Experimenta con abrir y guardar archivos y con editar el texto y aplicarle estilo. Si tienes acceso a otros archivos RTF, ábrelos también.

23) Sal de la aplicación.

En este capítulo has aprendido los fundamentos de leer y escribir datos. También tendrás un conocimiento sólido sobre cómo leer y escribir archivos, especialmente archivos de texto. Aún queda mucho por leer, especialmente cuando se trata de imágenes y bases de datos, aspectos que aprenderás en los próximos capítulos.

Capítulo 10

Dibújalo (e imprímelo!)

CONTENIDOS

1. **Introducción al Capítulo**
2. **Trabajar con Imágenes**
3. **Dibujar desde Código**
4. **Imprimir**
5. **Imprimir en la Práctica**

10.1 Introducción al Capítulo

Ahora que has aprendido cómo guardar los datos del usuario mediante la creación de archivos, el próximo paso es permitir que los usuarios impriman su información. La impresión en Xojo se realiza mediante la clase Graphics.

Además, a medida que las interfaces de usuario se tornan más sofisticadas también es más importante ofrecer gráficos de gran calidad en tu app. Algunas veces este es el tipo de imágenes que importas en tu proyecto, y otras es el tipo de imágenes que generas en tu código.

En el Capítulo 2 has aprendido sobre los tipos de datos soportados por Xojo, incluyendo strings, tipos de datos numéricos, fechas y colores. Cuando se trata de información de imágenes existen principalmente dos tipos de datos. Uno de ellos es Picture. Como probablemente hayas imaginado por su nombre, la clase Picture representa una imagen. Puede ser una imagen cargada de archivo o bien que hayas dibujado mediante código.

El otro tipo de dato que trataremos es la clase Graphics. Puede que resulte confuso tener una clase Picture y una clase Graphics, pero los motivos de ello se verán con más claridad a medida que

leas este capítulo. En realidad, las clases Graphics y Picture trabajan juntas para permitirte trabajar con datos de imágenes.

Como proyecto de ejemplo de este capítulo, harás un cambio al menú de comida online creado en un capítulo anterior, proporcionando la capacidad de imprimir el pedido del usuario.

Una nota sobre Pantallas HiDPI (Retina)

Si tienes una pantalla HiDPI (Retina), puedes activar la propiedad HiDPI para hacer que tanto el texto como las imágenes se vean con mayor nitidez. Para asegurarte de que está activado, dirígete a Shared Build Settings y activa la propiedad Supports Retina / HiDPI a su posición ON.

10.2 Trabajar con Imágenes

Como se ha indicado anteriormente, la clase `Picture` contiene una imagen. En vez de hablar de ello de forma abstracta, veámoslo mediante un ejemplo.

- 1) **Crea un nuevo proyecto Desktop en Xojo y guárdalo como “Pictures”.**
- 2) **Añade un Botón a Window1. Añade este código a su evento Pressed:**

```
Var f As FolderItem
Var d As OpenFileDialog
Var myPic As Picture
d = New OpenFileDialog
f = d.ShowModal()
If f <> Nil Then
    myPic = Picture.Open(f)
End If
```

Parte de este código debería de resultarte familiar. El código crea unas cuantas variables y pide al usuario que seleccione un archivo. Una cosa distinta es que una de estas variables es del tipo de dato `Picture`.

La variable `Picture`, `myPic`, es asignada cuando se utiliza `Picture.Open`. `Picture.Open` es un método compartido de la clase `Picture`, tal y como los métodos compartidos que has visto en el último capítulo. Este toma un

`FolderItem` como parámetro y devuelve un objeto `Picture`, el cual se asigna a `myPic` (asumiendo que `FolderItem` es válido y no `Nil`).

- 3) **Ejecuta el proyecto.**
- 4) **Haz clic en el PushButton y selecciona un archivo de imagen (cualquier formato de imagen común servirá, como JPEG, GIF, PNG o TIFF).**

Observa lo que ocurre una vez que hayas abierto el archivo: nada. No ha ocurrido nada porque no le has indicado a tu código que haga nada todavía con el objeto de imagen.

- 5) **Sal de la aplicación.**
- 6) **Añade un ImageViewer a Window1.**

El `ImageViewer` se utiliza para mostrar una imagen. Ajusta el tamaño del `ImageViewer` para que utilice la mayor parte de `Window1`.

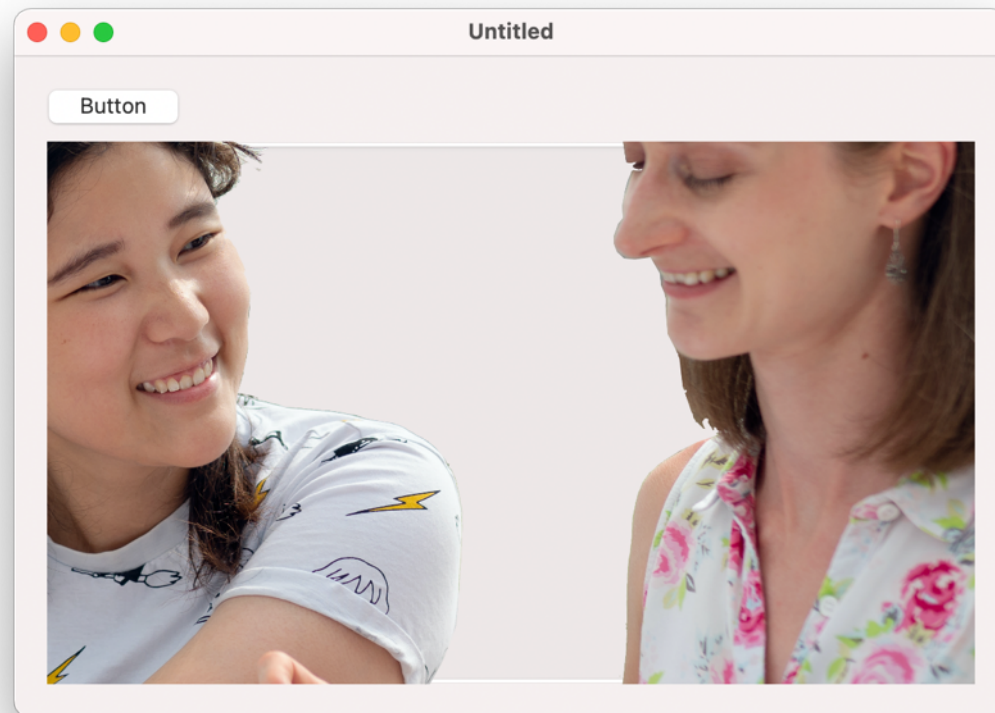
- 7) **Añade este código justo antes de la línea “End If” en el evento Pressed del PushButton:**

```
ImageViewer1.Image = myPic
```

- 8) **Ejecuta el proyecto.**

Nuevamente, haz clic en el `PushButton` y selecciona un archivo de imagen.

- 9) Esta vez deberías ver la imagen seleccionada mostrada en el ImageViewer:



Evidentemente tu imagen será distinta.

- 10) **Sal de la aplicación.**

10.3 Dibujar desde Código

Ahora que sabes las bases de abrir y mostrar una imagen es el momento de aprender sobre la clase Graphics. Si bien la clase Picture es muy útil para almacenar datos de imagen, la clase Graphics hace la mayoría del “trabajo pesado” cuando se trata de mostrar y manipular datos de imagen.

- 1) **Crea un nuevo proyecto Desktop y llámalo Pictures-Canvas.**
- 2) **Añade un Botón y añade un Canvas. Cambia el tamaño del Canvas para que utilice la mayor parte de la ventana.**
- 3) **Añade una propiedad a la Ventana: MyPic As Picture.**
- 4) **Añade este código al evento Pressed del botón:**

```
Var f As FolderItem
Var d As OpenFileDialog
d = New OpenFileDialog
f = d.ShowDialog()
If f <> Nil Then
    MyPic = Picture.Open(f)
    Canvas1.Refresh
End If
```

Este es prácticamente idéntico al código introducido anteriormente, sólo han cambiado unas pocas líneas. Ahora la imagen seleccionada se asigna

a la propiedad y se indica al Canvas que se actualice a sí mismo (llamando a Refresh).

- 5) **Añade este código en el manejador de evento Paint del Canvas:**

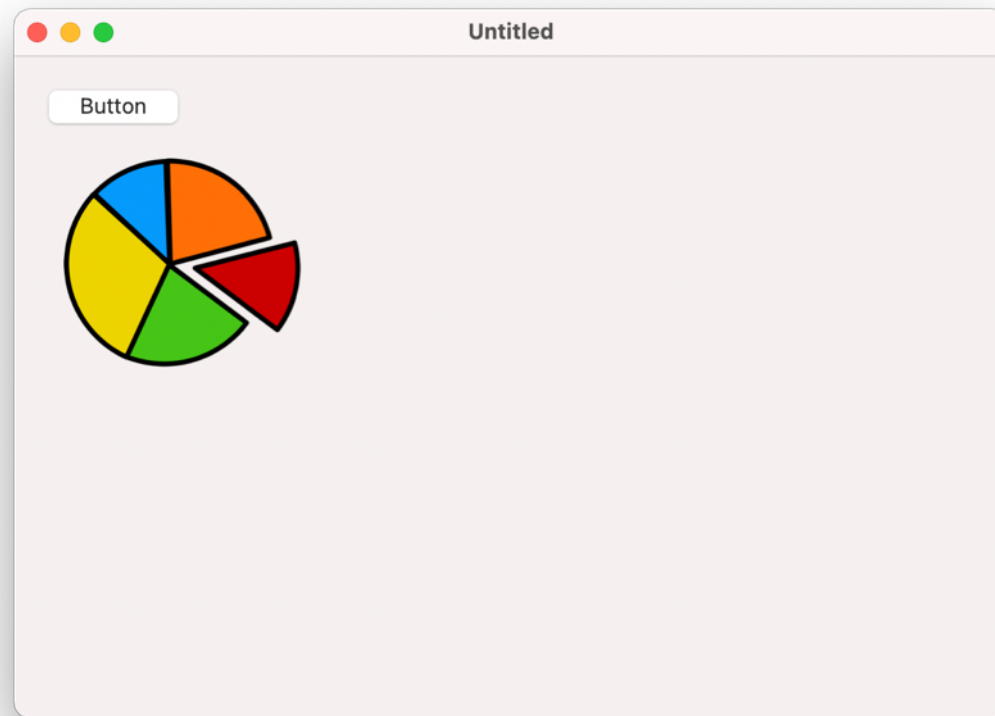
```
If MyPic <> Nil Then
    g.DrawPicture(MyPic, 10, 10)
End If
```

Este código dibuja la imagen en el Canvas (siempre que esté disponible una imagen).

La clase Graphics (proporcionada por el parámetro g del evento Paint) tiene varios métodos para manipular los datos de imagen. En este ejemplo, se utiliza el método DrawPicture para dibujar el objeto de imagen seleccionado en el Canvas (tanto si está en la propiedad Graphics del Canvas como si se muestra en pantalla). Aprenderás mucho más sobre el método DrawPicture en esta sección pero, por ahora, deberías saber que los tres parámetros dado para dibujar la imagen son las coordenadas X e Y para indicar dónde ha de dibujarse.

- 6) **Ejecuta el proyecto.**
- 7) **Haz clic en el botón y selecciona un archivo de imagen.**

En esta ocasión verás la imagen seleccionada mostrada por el Canvas; tal y como se aprecia en la siguiente captura de pantalla:



Observa que la imagen mostrada en el Canvas no tiene borde, tal y como ocurre en el ImageViewer. Esto se debe a que el Canvas no proporciona un marco automáticamente. Observa también que mientras que el ImageViewer centraba la imagen por ti, este no es el caso del Canvas. En este punto puede que te preguntes por qué utilizar un Canvas en vez de un ImageViewer. En resumen, el Canvas te proporciona un mayor control sobre el modo de mostrar tu imagen.

8) Sal de la aplicación.

El anterior ejemplo utilizaba el método DrawPicture con tres parámetros: la imagen a dibujar, seguido de las coordenadas X e Y. Las coordenadas son relativas a la esquina superior izquierda del Canvas, de modo que al dibujar la imagen en 0,0 esta lo hará

sobre la esquina superior izquierda del Canvas, mientras que si se dibuja la imagen en 72,36 resultará en la imagen dibujada aproximadamente a una pulgada del margen izquierdo del Canvas y una pulgada y media desde la parte superior. Toma algunos minutos para experimentar con diferentes valores para las coordenadas X e Y. Observa como cambia la posición de la imagen.

Pero no te limites a pensar que sólo puedes proporcionar números predeterminados como coordenadas. Después de todo, sólo son enteros, de modo que puedes dar a DrawPicture cualquier valor que quieras calcular. Por ejemplo, supongamos que quieres centrar la imagen en el Canvas. Si sabes el tamaño de la imagen y el tamaño del Canvas, puede resultar relativamente sencillo hacer el cálculo para centrar la imagen. El reto llega cuando no sabes alguno de los tamaños por adelantado. Por fortuna hay un modo simple de centrar la imagen. La coordenada X debería ser la mitad del ancho del Canvas. La coordenada Y debería ser igual, salvo que usando la altura en vez del ancho. Tanto la clase Canvas como Picture te proporcionan esta información mediante sus propiedades Height y Width.

9) Vuelve a tu proyecto Pictures-Canvas en Xojo.

10) Cambia el evento Paint para calcular la posición central de la imagen. El código será como este:

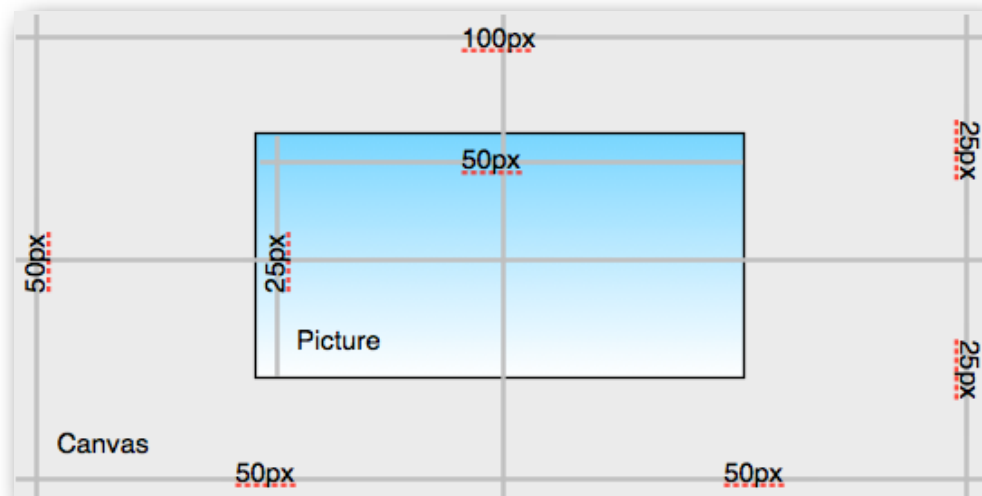
```
If MyPic <> Nil Then
```

```

Var x, y As Integer
x = g.Width / 2 - MyPic.Width / 2
y = g.Height / 2 - MyPic.Height / 2
g.DrawPicture(MyPic, x, y)
End If

```

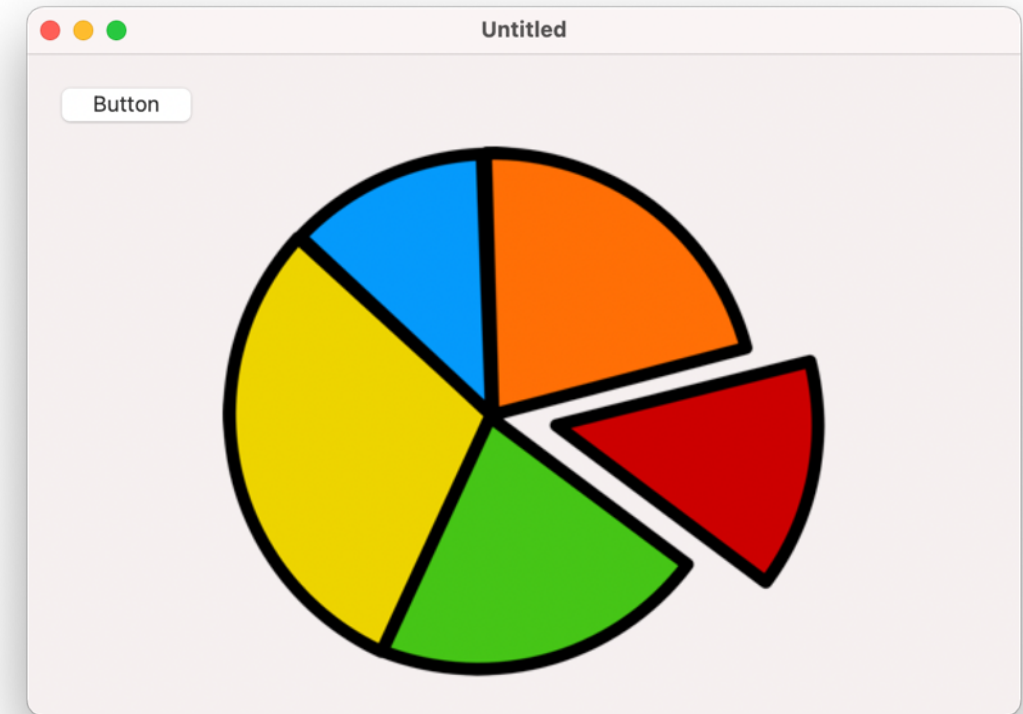
El anterior código calcula X e Y. X es la mitad del ancho del Canvas menos la mitad del ancho del Picture. Adicionalmente, Y es la mitad de la altura del Canvas menos la mitad de la altura del Picture. Si dicha fórmula te resulta confusa, el siguiente diagrama puede resultarte de ayuda:



11) Eecuta el proyecto.

12) Haz clic en el Botón y selecciona un archivo de imagen.

Esta vez deberías de ver la imagen seleccionada centrada en el Canvas:



Recuerda que la imagen está centrada en el Canvas, no respecto a la ventana.

13) Sal de la aplicación.

Ahora que has aprendido más sobre cómo posicionar tus imágenes usando el método DrawPicture, puede que te preguntes sobre si es posible recortar o escalar tus imágenes. Y es posible.

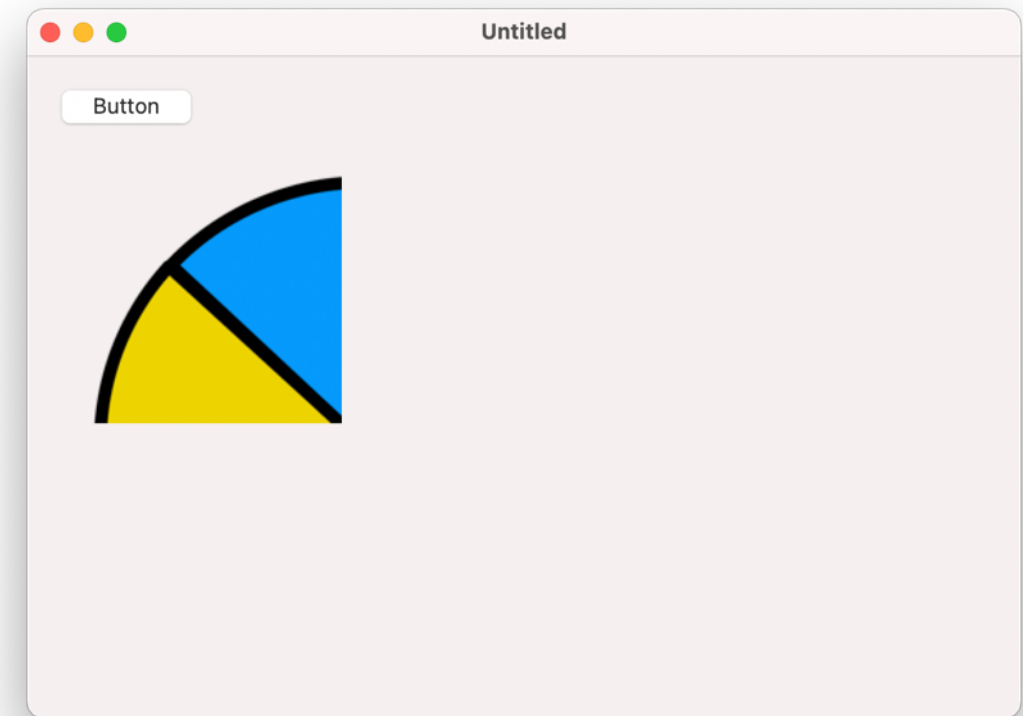
DrawPicture requiere de tres parámetros que ya has visto, pero puede tomar más si necesitas un mayor control sobre cómo ha

de mostrarse la imagen. De hecho, DrawPicture puede tomar hasta nueve parámetros.

En tu proyecto Picture-Canvas, dirígete al evento Paint y cambia la línea de código que contiene DrawPicture por esta:

```
g.DrawPicture(MyPic, 20, 20, 150, 150)
```

Estos dos parámetros adicionales son DestWidth y DestHeight, y determinan el ancho y altura de la imagen a mostrar. Cuando ejecutes el proyecto verás que la imagen seleccionada no va más allá de los 150 puntos de ancho y 150 puntos de alto (si la imagen es inferior a 150 puntos en cualquier dimensión, no advertirás ninguna diferencia). En resumen, DestWidth y DestHeight te permiten recortar la imagen:

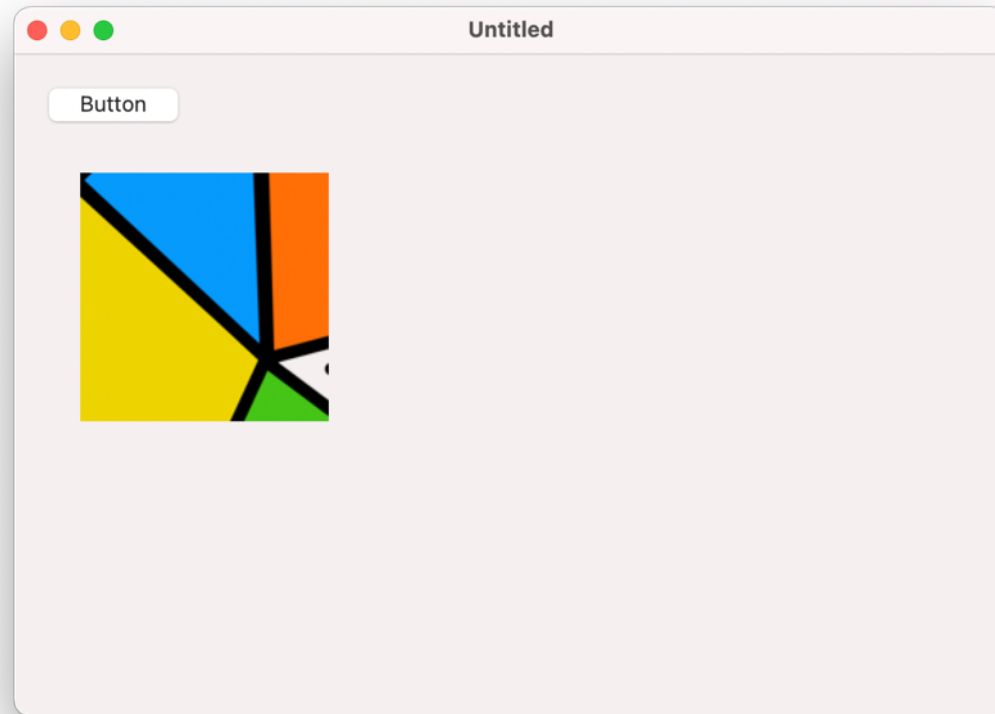


Añade ahora dos parámetros más, de modo que la línea de código sea esta:

```
g.DrawPicture(MyPic, 20, 20, 150, 150, 50, 50)
```

Estos dos parámetros son SourceX y SourceY, e indican a DrawPicture desde donde ha de comenzar a dibujar la imagen fuente. Por omisión ambos son cero, de modo que DrawPicture comienza en la esquina superior izquierda de la imagen.

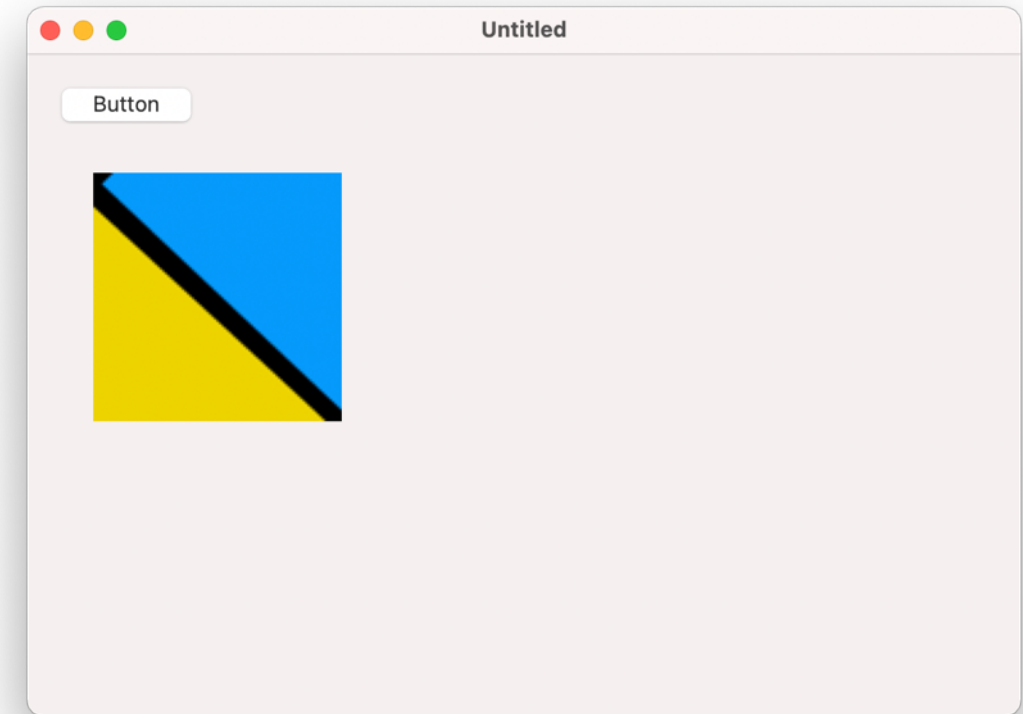
Con SourceX y SourceY, puede que veas la imagen más como esto:



Hasta ahora llevamos siete parámetros. Como se ha indicado anteriormente, DrawPicture puede tomar hasta nueve. Los dos últimos son SourceWidth y SourceHeight. Estos permiten escalar la imagen. Cambia la línea DrawPicture una vez más:

```
g.DrawPicture~  
  (myPic, 20, 20, 150, 150, 50, 50, 100, 100)
```

En esta ocasión, la imagen puede verse así:



Observa que la imagen está ampliada porque se ha escalado.

Estos parámetros adicionales de DrawPicture serán prácticamente siempre valores calculados en vez de números escritos directamente en el código, tal y como se ha demostrado. Cuando se utilizan combinados te proporcionan un gran control a la hora de mostrar una imagen.

Está muy bien si tienes una imagen que puedas usar a mano en tu ordenador; pero probablemente querrás dibujar tu propio control en algún momento. La clase Graphics tiene varios métodos que te permiten crear imágenes.

- 1) Crea un nuevo proyecto Xojo Desktop y guárdalo como “Drawing”.
- 2) Añade un Canvas a Window1. Cambia su tamaño para que cubra la mayor parte de la ventana.
- 3) Añade este código en el evento Paint del Canvas:

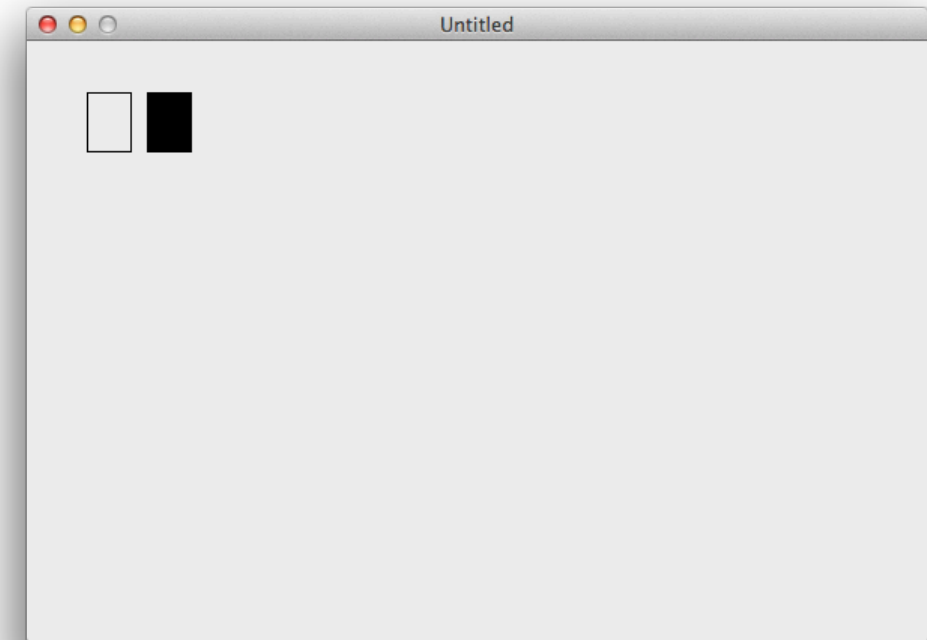
```
g.DrawRectangle(20, 20, 30, 40)
g.FillRectangle(60, 20, 30, 40)
```

La variable “g” es la propiedad Graphics de Canvas1. El evento Paint proporciona dicha propiedad para ti:

	Paint(g As Graphics, areas() As REALbasic.Rect)
—	<code>g.DrawRectangle(20, 20, 30, 40)</code>
—	<code>g.FillRectangle(60, 20, 30, 40)</code>

4) Ejecuta el proyecto.

El evento Paint de Canvas1 se dispara directamente, de modo que no necesitas hacer clic en ningún botón. Deberías ver una ventana como esta:



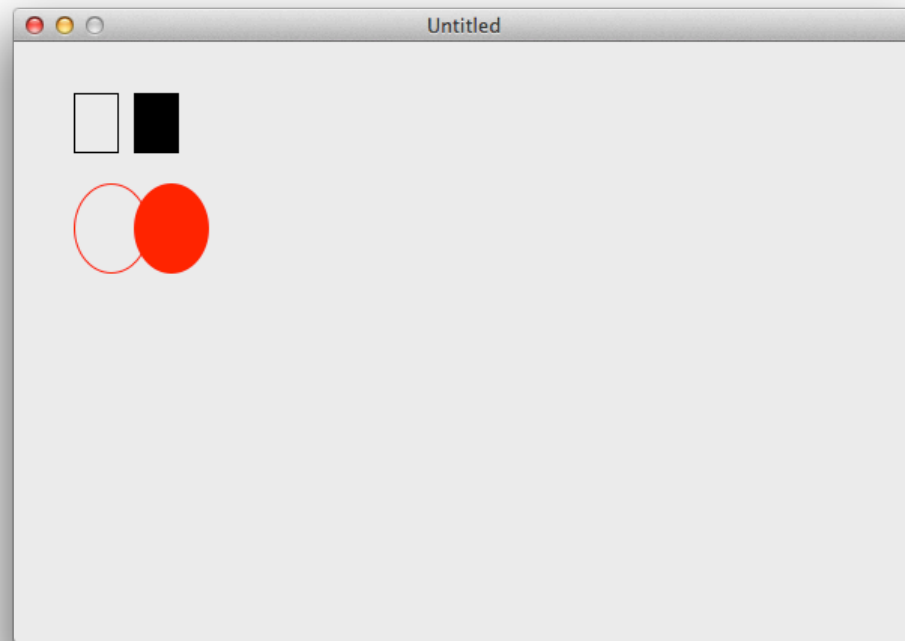
Observa que un rectángulo está vacío y el otro está relleno. Esta es la diferencia entre el método DrawRectangle, que sólo dibuja el contorno del rectángulo, y el método FillRectangle que colorea el rectángulo. Ambos toman los mismos cuatro parámetros: X, Y, Width y Height. Por tanto, para dibujar un cuadrado, deberías asegurarte de que tanto el ancho como la altura tienen el mismo valor.

- 5) Sal de la aplicación.
- 6) Añade estas líneas de código al final del evento Paint de Canvas1:

```
g.DrawingColor = RGB(255, 0, 0)
g.DrawOval(20, 80, 50, 60)
g.FillOval(60, 80, 50, 60)
```

7) Ejecuta de nuevo tu proyecto

Deberías ver una ventana como esta:



Observa que los óvalos son rojos. Esto se debe a que has definido la propiedad `DrawingColor` de la clase `Graphics` antes de dibujarlos. `DrawOval` y `FillOval` son como `DrawRectangle` y `FillRectangle`, tomando cuatro parámetros: `X`, `Y`, `Width` y `Height`.

8) Sal de la aplicación.

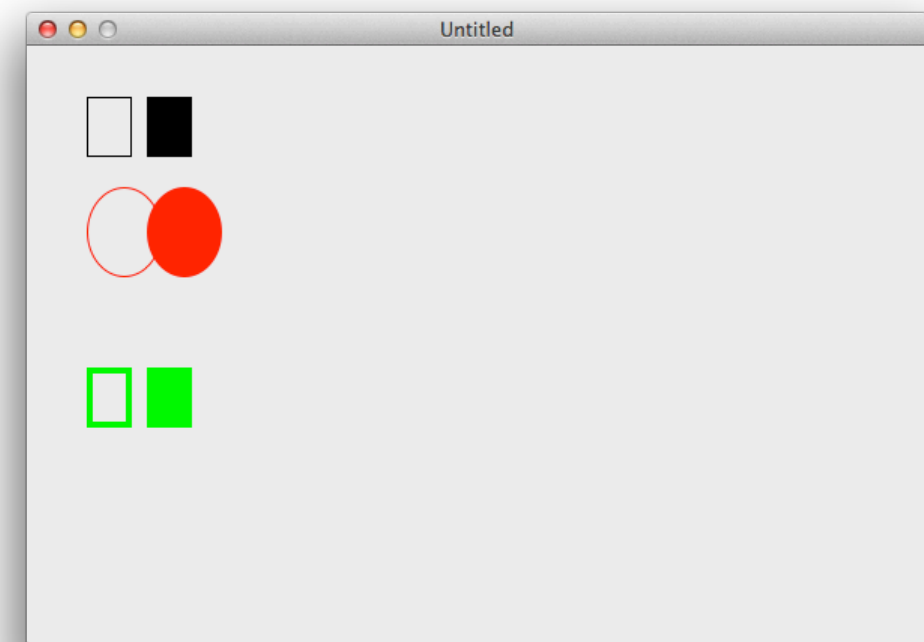
9) Añade estas líneas de código al final del evento `Paint` de `Canvas1`:

```
g.DrawingColor = RGB(0, 255, 0)
g.PenSize = 4
g.DrawRectangle(20, 200, 30, 40)
```

```
g.FillRectangle(60, 200, 30, 40)
```

Esta vez has definido el `DrawingColor` como verde y también has hecho que el “lápiz” sea más grueso ajustando la propiedad `PenSize` a 4.

10) Ejecuta el proyecto y verás el resultado:



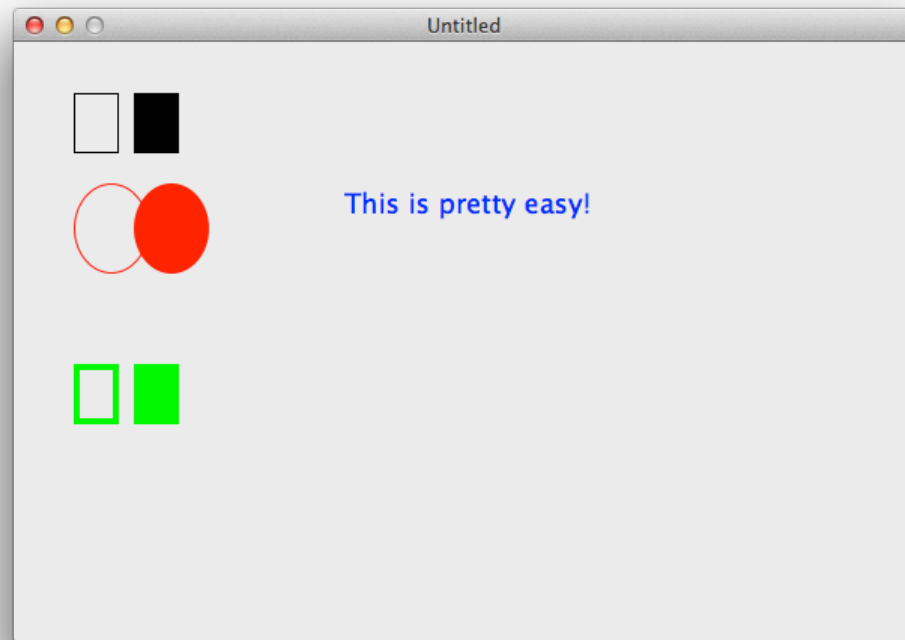
Observa que el contorno del rectángulo verde es más grueso.

11) Sal de la aplicación.

12) Añade estas líneas de código a final del evento `Paint` de `Canvas1`:

```
g.DrawingColor = RGB(0, 0, 255)
g.FontSize = 18
g.DrawText("This is pretty easy!", 200, 100)
```

13) Ejecuta la aplicación y deberías ver una ventana como esta:



El método DrawText dibuja el texto pasado, tal y como implica su nombre.

14) Sal de la aplicación.

La clase Graphics también tiene varias propiedades relacionadas con el dibujo de Strings. Por ejemplo, puedes definir la propiedad bold a True antes de llamar a DrawText para dibujar texto en negrita (pero no olvides volver a ajustarla a False cuando ya no necesites que el texto esté en negrita). Lo mismo ocurre con Italic (cursiva) y Underline (subrayado). Además, puedes definir el FontName (nombre de la fuente) y FontSize (tamaño de texto) y, como has visto anteriormente, DrawingColor para controlar el color del texto.

Ahora que has creado una obra maestra, querrás guardarla. Puede que resulte raro, pero no hay forma de guardar los contenidos de un objeto Graphics. Sólo puede guardarse a disco el objeto Picture. Sin embargo, es fácil de hacer dibujando en un Picture en vez de hacerlo directamente en el Canvas.

- 1) **Vuelve a tu proyecto Drawing en Xojo y usa Save As para crear un nuevo proyecto llamado Drawing-Save.**
- 2) **Añade una propiedad a la Ventana: MyPic As Picture.**
- 3) **Cambia el código en el evento Paint del Canvas por este:**

```
MyPic = New Picture(g.Width, g.Height)
MyPic.Graphics.DrawRectangle(20, 20, 30, 40)
MyPic.Graphics.FillRectangle(60, 20, 30, 40)
MyPic.Graphics.DrawingColor = RGB(255, 0, 0)
MyPic.Graphics.DrawOval(20, 80, 50, 60)
MyPic.Graphics.FillOval(60, 80, 50, 60)
MyPic.Graphics.DrawingColor = RGB(0, 255, 0)
MyPic.Graphics.PenSize = 4
MyPic.Graphics.DrawRectangle(20, 200, 30, 40)
MyPic.Graphics.FillRectangle(60, 200, 30, 40)
MyPic.Graphics.DrawingColor = RGB(0, 0, 255)
MyPic.Graphics.FontSize = 18
MyPic.Graphics.DrawText~
    ("This is pretty easy!", 200, 100)
g.DrawPicture(MyPic, 0, 0)
```

- 4) **Ejecuta el proyecto.**

Deberías ver una ventana muy similar a la anterior.

5) Sal de la aplicación.

6) Mejora la calidad para pantallas HiDPI (Retina).

El anterior código funciona bien, pero si tienes una pantalla HiDPI (Retina), y has activado Supports Retina / HiDPI, entonces advertirás que el dibujo tiene un aspecto borroso. Esto se debe a que el objeto Picture (p) que has creado no tiene la escala correcta para pantallas HiDPI. Para obtener un objeto Picture con la escala correcta, llama al método `BitmapForCaching` de `Window`. Cambia la línea “New Picture” por esta:

```
MyPic = Self.BitmapForCaching(g.Width, g.Height)
```

7) Ejecuta de nuevo tu proyecto y observa que tanto el texto como los gráficos se ven ahora más nítidos.

8) Para guardar la imagen, añade un Botón con este código en su evento Action:

```
Var f As FolderItem
Var d As SaveFileDialog
d = New SaveFileDialog
f = d.ShowDialog()
If f <> Nil Then
    MyPic.Save(f, Picture.Formats.JPEG)
End If
```

Ahora que tienes tu dibujo en un objeto Picture (la propiedad `MyPic`), puede guardarse como archivo. Esto debería resultar familiar. El código pide al usuario que cree un nuevo archivo, usando a continuación el método `Save` de `Picture` para escribir la imagen a un archivo. `Save` requiere de dos parámetros: el `FolderItem` en el cual guardar el `Picture` y el formato de archivo. En este ejemplo verás que el `Picture` es `JPEG`.

9) Ejecuta tu proyecto y haz clic en el botón para guardar el dibujado.

Te preguntará que guardes tu archivo. Guárdalo como `masterpiece.jpg`.

10) Sal de la aplicación.

Abre `masterpiece.jpg` en cualquier editor de imágenes y deberías ver el dibujo creado mediante el uso de los diversos métodos de la clase `Graphics`.

10.4 Impresión

Ahora que sabes cómo gestionar Pictures y Graphics, tienes casi todo lo que necesitas para imprimir. La impresión también se realiza con un objeto Graphics, pero no es parte del Canvas o de Picture. Este objeto Graphics es el devuelto por la función ShowPrinterDialog de la clase PrinterSetup. ShowPrinterDialog pide al usuario que confirme lo que quiere imprimir, y entonces le da un objeto Graphics. Cualquier cosa dibujada en dicho objeto Graphics será lo enviado a la impresora.



- 1) **Crea un nuevo proyecto Xojo desktop y guárdalo como “Printing”.**
- 2) **Añade un Botón a Window1. Añade este código a su evento Action:**

```
Var g As Graphics
Var ps as New PrinterSetup
g = ps.ShowPrinterDialog()
If g <> Nil Then
    g.DrawString
        ("This is my first print job!", 100, 100)
End If
```

Como puedes ver, ShowPrinterDialog te devuelve un objeto Graphics con el que puedes trabajar. Has de asegurarte de que no sea Nil antes de intentar dibujar sobre el objeto Graphics. Si el usuario pulsa el botón Cancel en el diálogo de impresión, entonces “g” será Nil, y no podrás dibujar o imprimir.

Una vez que hayas verificado que “g” es un objeto Graphics válido, puedes usar cualquiera de los métodos y propiedades de la clase Graphics. La única diferencia es que cualquier cosa que dibujes se imprimirá en vez de ser mostrada en pantalla.

- 3) **Ejecuta el proyecto.**
- 4) **Haz clic en el Botón.**

La aplicación debería de imprimir tu mensaje.

5) **Sal de la aplicación.**

6) **Cambia el código en el evento Action del Botón por este:**

```
Var g As Graphics
Var ps as New PrinterSetup
g = ps.ShowPrinterDialog()
If g <> Nil Then
    g.DrawString(
        ("This is my first print job!",100,100)
    g.NextPage
    g.DrawString("This is my second page!",100,100)
    g.NextPage
    g.DrawString("This is my third page!",100,100)
End If
```

7) **Ejecuta el proyecto.**

8) **Haz clic de nuevo en el Botón.**

Deberías de ver un documento de tres páginas. Ahora sabes como imprimir un documento de varias páginas.

9) **Sal de la aplicación.**

Una regla de impresión general es: nunca asumas. No puedes predecir lo que hará el usuario, de modo que tu código ha de estar preparado para gestionar varias situaciones distintas, como diferentes tamaños de página, diferentes márgenes e incluso diferentes orientaciones de página. Es por esto por lo que resulta mejor no emplear valores fijos en código como posiciones y el tamaño de los objetos a dibujar; sino que es mejor utilizar valores relativos y escalar proporcionalmente el dibujo.

Puede que estés pensando que imprimir de esta forma una buena cantidad de texto con estilo puede resultar algo muy tedioso, y así es. Es por esto por lo que existe `StyledTextPrinter`.

- 1) **Abre el proyecto `StyledTextEditor` creado en el capítulo anterior.**
- 2) **Añade un Botón a la ventana. Define su etiqueta (Caption) a “Print” y escribe el siguiente código en su evento Action:**

```
Var g As Graphics
Var ps As New PrinterSetup
Var stp As StyledTextPrinter
g = ps.ShowPrinterDialog
If g <> Nil Then
    stp = EditingField.StyledTextPrinter(g, g.Width)
    stp.DrawBlock(0, 0, g.Height)
End If
```

El `TextArea` tiene una función llamada `StyledTextPrinter` que devuelve una instancia de la clase `StyledTextPrinter`. La función toma dos parámetros: un objeto `Graphics` (que puede ser el proporcionado por `ShowPrinterDialog`) y el ancho deseado del área de impresión. En este ejemplo, el área de impresión será todo el ancho de la página, pero puedes reducir dicho valor para imprimir en una columna estrecha o incluso en múltiples columnas.

La clase `StyledTextPrinter` tiene un método llamado `DrawBlock` que dibuja su `StyledText` en el objeto `Graphics` especificado anteriormente. `DrawBlock` toma tres parámetros: la coordenada X, la coordenada Y y la altura del bloque a imprimir. Nuevamente, en este ejemplo se utilizará toda la página.

Nota: `StyledTextPrinter` no está soportado en Windows.

- 3) **Ejecuta el proyecto.**
- 4) **Añade texto con estilo en EditingField o bien abre un documento RTF.**
- 5) **Una vez que tengas algo de texto con estilo con el cual trabajar, haz clic en el botón Print.**

Deberías de ver la impresión de tu texto con estilo.

- 6) **Sal de la aplicación.**

El objeto Graphics utilizado por StyledTextPrinter es como cualquier otro objeto Graphics, lo que significa que además de tu texto con estilo también puedes dibujar otras figuras y objetos.

10.5 Impresión en la práctica

Abre el proyecto Menu con el que trabajaste en el Capítulo 6. Para el proyecto de ejemplo de este capítulo, permitirás que el usuario pueda imprimir su pedido, además de mostrarlo en pantalla.

- 1) **Añade un nuevo método a Window1 llamado "PrintOrder". No recibirá parámetros.**
- 2) **En el evento Pressed de OrderButton, añade esta línea:**

```
PrintOrder
```

- 3) **Añade este código al método PrintOrder:**

```
Var g As Graphics
Var yOffset As Integer
Var ps as New PrinterSetup
g = ps.ShowPrinterDialog()
If g <> Nil Then
    If MainDishMenu.SelectedRowIndex <> -1 Then
        g.Bold = True
        g.DrawText("Main Dish:",20,20)
        g.Bold = False
        g.DrawText(MainDishMenu.Text,100,20)
        g.Bold = True
        g.DrawText("Side Order:",20,40)
        g.Bold = False
```

```
If FriesRadio.Value Then
    g.DrawText(FriesRadio.Caption,100,40)
End If
If PotatoRadio.Value Then
    g.DrawText(PotatoRadio.Caption,100,40)
End If
If OnionRingRadio.Value Then
    g.DrawText(OnionRingRadio.Caption,100,40)
End If
yOffset = 60
If CheeseCheckBox.Value Then
    g.Bold = True
    g.DrawText("Extra:",20,yOffset)
    g.Bold = False
    g.DrawText(CheeseCheckBox.Caption, 100,yOffset)
    yOffset = yOffset + 20
End If
If BaconCheckBox.Value Then
    g.Bold = True
    g.DrawText("Extra:",20,yOffset)
    g.Bold = False
    g.DrawText(BaconCheckBox.Caption, 100,yOffset)
    yOffset = yOffset + 20
End If
g.Bold = True
g.DrawText("Notes:",20,yOffset)
g.Bold = False
g.DrawText(NotesField.Value,100,yOffset, 100,
(g.Width-40))
End If
End If
```


Se trata ciertamente de mucho código, más del que hayas visto hasta ahora en este libro. Pero basado en todo lo que has aprendido hasta ahora, no hay nada nuevo.

Una aproximación para imprimir el pedido podría ser dibujar cada elemento del pedido mediante `DrawText`, línea a línea en la página incrementando la coordenada Y. Esta es la técnica básica utilizada por este código, pero recuerda que tu código ha de ser inteligente. Si el usuario no elige un complemento o una opción, has de asegurarte de que tu aplicación no imprima una línea en blanco en las posiciones en las que no exista una opción. Del mismo modo, el usuario puede elegir cero, una o dos opciones, por lo tanto has de asegurarte de que habrá suficiente espacio para ellas.

Por ello, este código mantiene un registro de la posición vertical de la página utilizando la variable `yOffset`. Si ha de imprimirse una línea, entonces se incrementa `yOffset`, y si es necesario saltarse una línea, `yOffset` no se incrementará.

Este código es muy similar al del método `CompileOrder`. La única diferencia es que en vez de añadir los elementos al `TextArea`, los dibuja en el objeto `Graphics`. También conmuta `Bold` a `True` para las etiquetas y a `False` para los elementos propiamente dichos.

4) Ejecuta el proyecto.

5) Selecciona tu pedido y haz clic en `OrderButton`.

Se te pedirá que imprimas el pedido.

6) Sal de la aplicación.

En este capítulo has aprendido conceptos valiosos para tratar con imágenes y gráficos, así como para imprimir los datos. Ahora

tienes otra opción que ofrecer a tus usuarios cuando utilicen tus soluciones.

Conexiones

CONTENIDOS

1. Introducción al Capítulo
2. Fundamentos de redes
3. Haciendo Conexiones
4. Conexiones Web
5. Enviar email
6. Anotaciones sobre Protocolos

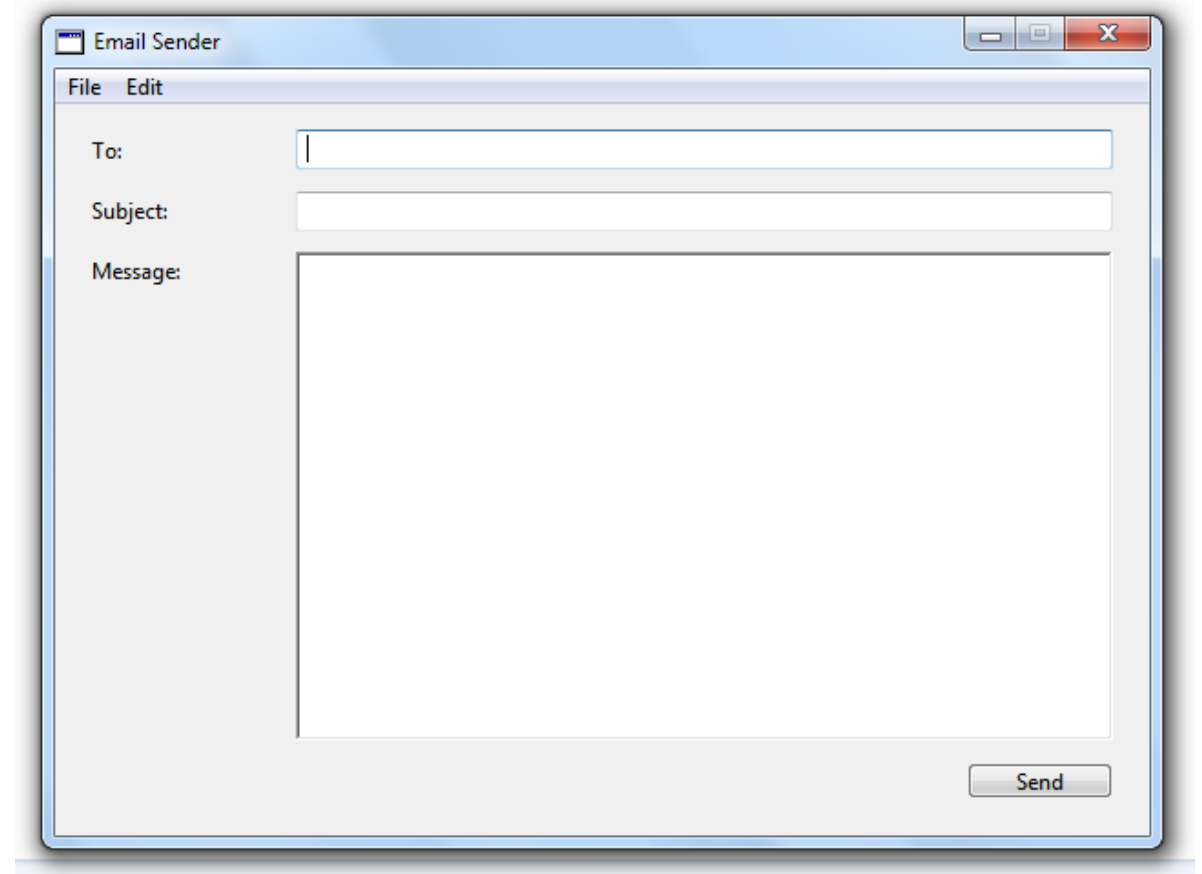


11.1 Introducción del Capítulo

Gran parte de lo que hacemos con los ordenadores y dispositivos móviles en la actualidad tiene que ver con las redes y con Internet en particular.

Redes es la palabra utilizada para describir como dos o más ordenadores hablan entre sí. Esta puede ser una conversación “uno a uno” entre dos ordenadores transfiriendo un archivo. Puede ser una decena de amigos jugando a un juego multiplataforma sobre Internet. O puede ser miles de personas usando Facebook al mismo tiempo para chatear y compartir fotografías y actualizaciones de estado. En el fondo, cada uno de estos escenarios involucra el uso de redes.

En este capítulo aprenderás cómo añadir algunas capacidades de red a tus proyectos. También crearás un proyecto de ejemplo llamado Email Sender y que, como habrás adivinado por el nombre, envía mensajes de email. Tu proyecto se verá como este:



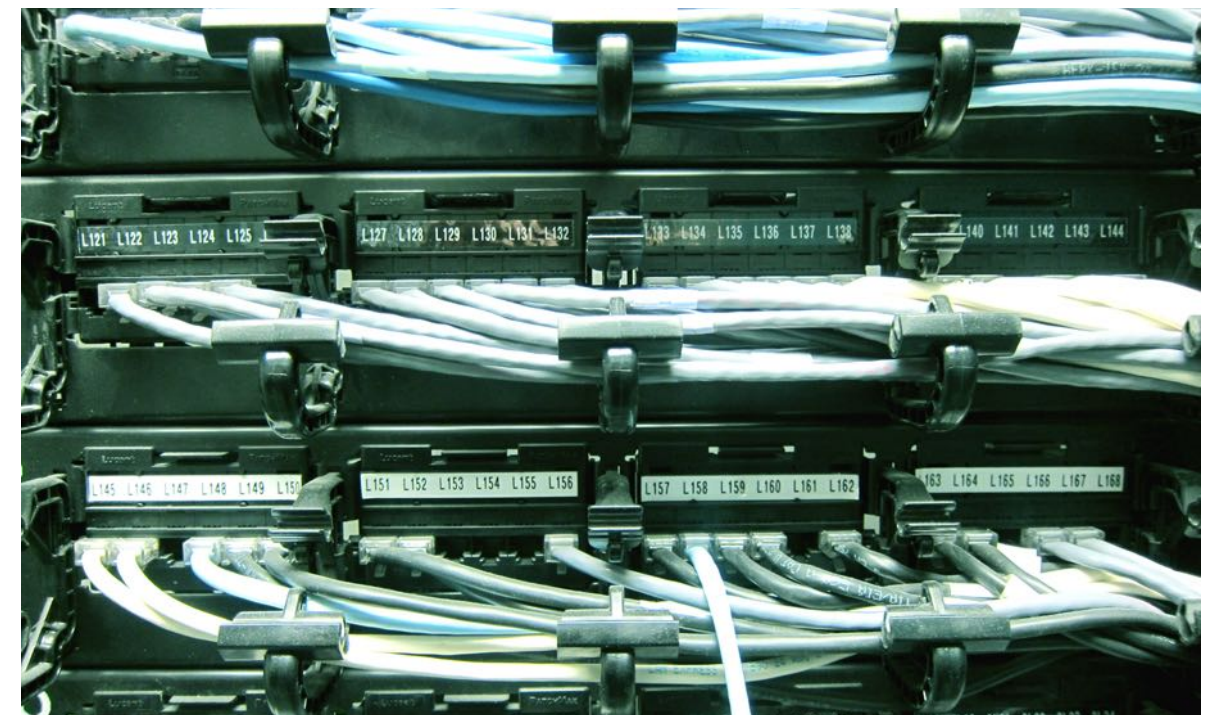
11.2 Fundamentos de redes: Protocolos, Puertos, y Direcciones

Esta sección puede resultar, admitámoslo, un poco dura. Pero hay ciertos términos que necesitas conocer y comprender antes de que puedas añadir capacidades de conexión a redes en tus aplicaciones.

Imagina que vas a otro país donde se habla un idioma y se tienen unas costumbres diferentes. Imagina ahora que no hablas dicho lenguaje y que tampoco conoces sus costumbres. Si fueses a una entrevista de trabajo en dichas circunstancias muy probablemente no sería un buen asunto. No podrías comunicarte con tu entrevistador y también podrías ofenderle si intentas estrecharle la mano.

La conexión a redes es similar. Cada vez que un ordenador o dispositivo se comunica con otro ordenador o dispositivo en una red, deben de ocurrir ciertas cosas. En primer lugar, ambos dispositivos deben de acordar el modo en el que se van a comunicar. Esto es lo que se llama protocolo. Un protocolo es un conjunto de instrucciones que dictan cómo han de tener lugar las comunicaciones. Por ejemplo, cuando utilizas un navegador web estás usando un protocolo llamado HyperText Transfer Protocol

(también conocido como HTTP; por ello que veas a menudo las direcciones web comenzando con “http”). La documentación para un protocolo es extremadamente técnica y difícil de leer para no ingenieros. Esto no significa que no puedas leer e implementar un protocolo tu mismo; sólo es un aviso de que puede ser una tarea compleja y que requiere de mucho trabajo. Por fortuna, Xojo incluye de serie soporte para una buena cantidad de protocolos frecuentes.



Imagina ahora que estás esperando fuera de la casa de un amigo, tratando de hablar con él a través de la ventana. Para que esto ocurra, ambos tenéis que estar en la misma ventana. Si estuvieses gritándole por la ventana del salón mientras que tu amigo está en las escaleras... no podríais comunicaros. Pero si ambos estáis en la misma ventana, entonces la conversación

estaría limitada sólo a vuestra imaginación. De forma similar, un puerto es “donde” los dos dispositivos tratan de comunicarse. El puerto es un número, no un puerto físico real en tu ordenador (como el puerto de vídeo o USB). Cada ordenador conectado a una red tiene disponibles miles de números de puerto. Por lo general, los primeros 1024 puertos están reservados para su uso por parte del sistema operativo, si bien están disponibles muchos miles más. La mayoría de los protocolos definen el número de puerto sobre el cual están diseñados para operar. Por ejemplo, HyperText Transfer Protocol casi siempre opera en el puerto 80. Los protocolos pueden operar en otros puertos cuando es necesario, pero se espera que cada protocolo tenga un puerto. Si vas a usar un puerto no estándar, sólo has de asegurarte de que ambos dispositivos estén informados del cambio.

Por último, cada dispositivo involucrado en la comunicación tiene una dirección. Por lo general esta dirección puede tomar dos formas: la dirección IP y el nombre DNS. Probablemente hayas visto una dirección IP con anterioridad (IP significa Internet Protocol). Está compuesto de cuatro números bajo la versión IPv4, cada uno de ellos entre cero y 255. Por ejemplo, es posible que la IP de tu router WiFi sea similar a 192.168.1.1.

Dado que resulta difícil recordar estos números, es por lo que se utilizan con frecuencia los nombres DNS. Un nombre DNS es una serie de palabras (o sílabas) asociadas a una dirección IP. Por ejemplo, seguramente te resulte más sencillo recordar

“google.com” como la dirección de un sitio web; y seguramente que harías menos búsquedas si tuvieses que recordar y escribir “72.14.204.102” cada vez que necesitas usar Google. DNS significa Domain Name Service, y existen servidores DNS “maestros” en Internet que mantienen un registro sobre qué nombres se corresponden con cada dirección IP.

En resumen, puedes encontrar que tu app necesita hablar con google.com (o 72.14.204.102) en el puerto 80 usando el protocolo HyperText Transfer Protocol.

La versión IP usada por la mayoría de los actuales dispositivos es la versión 4, pero actualmente muchos están cambiando sus redes a IPv6. IPv4 proporciona un total de 4.294.967.296 direcciones posibles. Puede que parezcan muchas, pero cada dispositivo en Internet ha de tener un número único: cada ordenador, cada teléfono inteligente, tablet, etc. Hay unas cuantas formas creativas de “duplicar” las direcciones IP disponibles, pero lo cierto es que cuando leas esto ya nos habremos quedado sin números disponibles. IPv6 soluciona este problema al proporcionar un total de 2^{128} posibles direcciones únicas. Esto es, dos elevado a 128... lo cual se conoce en el campo de las matemáticas como un RON o Ridiculously Large Number. Formateado como entero, dicho número sería 340.282.366.920.938.000.000.000.000.000.000.000.000.000.000. Esto es suficiente para que cada persona del planeta tenga un total de 51.557.934.381.960.373.252.026.455,671 para sí mismo.

11.3 Haciendo Conexiones

Si te sientes apabullado, no te preocupes. Xojo tiene una clase llamada SocketCore que se encarga de una buena cantidad de los detalles por tu. No usas SocketCore directamente, sino alguna de sus subclases (una subclase es una derivación de otra clase. Aprenderás más sobre las subclases en el Capítulo 13). Estas subclases de SocketCore son referidas generalmente como Sockets.

En anteriores capítulos aprendiste sobre el uso de diferentes tipos de controles para crear la interfaz de usuario de tu aplicación. El Socket también es un control, de hecho es un ejemplo de control no visual. En otras palabras, aunque lo arrastres sobre tu ventana, el usuario final no lo verá en tu aplicación. Te proporciona eventos a los que puedes responder, pero no ofrece elementos de interfaz de usuario.

- 1) **Crea un nuevo proyecto Xojo Desktop y guárdalo como "Sockets".**
- 2) **Añade un Botón a Window1. Añade también un TCPSocket.**

Observa que el TCPSocket se añade bajo el editor de la ventana (en la "bandeja"). Esto se debe a que no forma parte del diseño de tu interfaz de usuario.

- 3) **Añade este código al evento Action del Botón:**

```
TCPSocket1.Address = "www.google.com"  
TCPSocket1.Port = 80  
TCPSocket1.Connect
```

El código le indica al Socket qué dirección ("www.google.com") y qué puerto (80) ha de usar. Posteriormente le indica al Socket que se conecte a dicha dirección. Observa que también puedes definir la dirección y puerto del Socket en el Inspector.

- 4) **Añade un manejador de evento al Socket para que gestione el evento Connected. Añade este código:**

```
MessageBox("You are connected!")
```

Este evento se dispara cuando se ha conectado el socket al dispositivo en la dirección indicada y utilizando el puerto especificado.

- 5) **Ejecuta tu proyecto y haz clic en el Botón.**

Tras un momento deberías de ver un mensaje indicándote que estás conectado.

- 6) **Sal de la aplicación.**

Muy bien, pero hemos de admitir que este proyecto está lejos de impresionar a nadie. Después de todo, el último fin de las comunicaciones es enviar y recibir información entre dispositivos. Para enviar datos con un Socket utilizas el método Write.

- 1) **Añade una TextArea a Window1. Cambia su tamaño para que cubra la mayor parte de la ventana.**
- 2) **Cambia el código en el evento Connected del TCPSocket por este:**

```
TextArea1.AddText("You are connected!" +  
    + EndOfLine)  
Me.Write("GET")
```

- 3) **Añade este código en el evento SendProgress del TCPSocket:**

```
Var s As String  
s = "Sent " + BytesSent.ToString  
s = s + " bytes so far..."  
s = s + EndOfLine  
TextArea1.AddText(s)
```

- 4) **Ejecuta el proyecto y haz clic en el PushButton.**

El TextArea debería de mostrar ahora que tu socket se ha conectado correctamente y luego debería revelar que has enviado tres bytes de datos en la red.

- 5) **Sal de la aplicación.**

Nuevamente, esto sigue sin ser impresionante; pero ahora sabes cómo enviar datos a través del Socket utilizando para ello el método Write. Podría ser más impresionante si, por ejemplo, pudiésemos recibir datos. Aquí es donde las cosas se tornan decididamente más difíciles. Al mismo tiempo, aquí también es

donde entran en juego los protocolos. Como se indicó anteriormente, un protocolo es un conjunto de reglas para que dos dispositivos se comuniquen entre sí a través de una red; y, sin un protocolo, dichas “conversaciones” son cuanto menos difíciles si no imposibles.

Cada TCPSocket con el que has estado trabajando ha estado funcionando en una especie de campana de vacío. Sin un protocolo establecido, este no sabe cómo ha de hablar con el otro dispositivo o bien cómo responderle. Más importante, no sabes cómo indicarle que lo haga... salvo que decidas usar un protocolo.

11.4 Conexiones Web

Como se indicaba anteriormente, Xojo incorpora soporte para algunos protocolos frecuentes de red, entre los cuales se encuentra el HyperText Transfer Protocol o HTTP. El control `URLConnection` ya conoce y comprende el protocolo HTTP, liberándote así para que te ocupes de otras áreas de tu app, en vez de tener que lidiar con las cuestiones intrínsecas de las comunicaciones de red.

1) **Crea un nuevo proyecto Xojo desktop y guárdalo como “HTTPStuff”.**

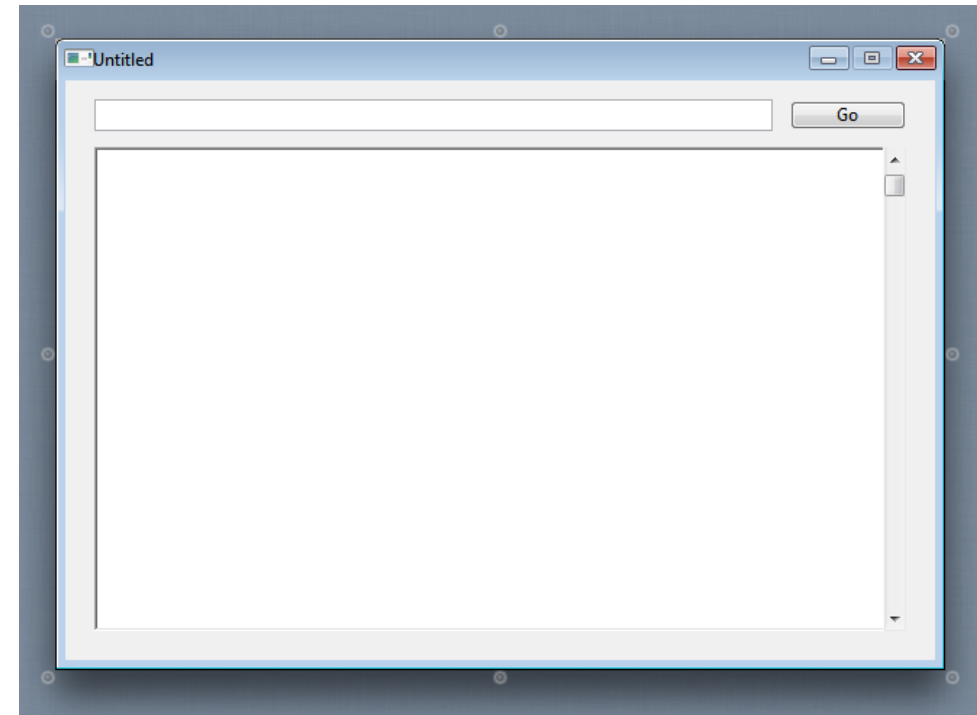
2) **Arrastra un control `URLConnection` sobre `Window1`.**

Como se ha indicado anteriormente, observa que `URLConnection` se sitúa en la bandeja del editor de ventana. Esto se debe a que no forma parte de la interfaz de usuario de tu aplicación.

3) **Añade un `TextField`, un Botón, y una `TextArea` a `Window1`.**

4) **Cambia la etiqueta del `PushButton` a “Go” y modifica el tamaño del `TextArea` de modo que ocupe la mayor parte de `Window1`.**

Sobre la disposición general de los elementos, tu interfaz debería parecerse a esta; pero siéntete libre de usar tu propia creatividad:



5) **Añade este código al evento `Pressed` del Botón:**

```
URLConnection1.Send("GET", TextField1.Text)
```

Este código indica a `URLConnection` que *obtenga* (Get) los datos de la página web a la que apunte el URL del `TextField1`. El método Get ofrecerá dos resultados: o bien `URLConnection` ha podido recuperar con éxito los contenidos de la página, o bien se ha producido un error.

6) **Introduce este código en el evento `Error` de `URLConnection1`:**

```
TextArea1.Text = "Error: " + e.Message
```

En el caso de que se produzca un error, se disparará el evento `Error` del `URLConnection`. El evento `Error` te proporciona un `RuntimeException` (en el

parámetro e) y el motivo de dicho error se encuentra en la propiedad Message.

Por supuesto, la respuesta preferible es la recuperación correcta con los contenidos de la página. Esto ocurre en el evento PageReceived. El evento ContentReceived proporciona cuatro variables: URL, una cadena que indica el URL de la página; HTTPStatus, un entero que indica el estado de la transmisión; Headers, una instancia de la clase InternetHeader que proporciona detalles adicionales sobre la página; y Content, una cadena que es el contenido real de la página en cuestión.

7) Introduce este código en el evento ContentReceived de URLConnection:

```
TextArea1.Text = Content
```

Dicho código simplificará el llenado de TextArea1 con los contenidos de la página.

8) Ejecuta el proyecto.

9) Introduce el URL de un sitio web (como por ejemplo “<https://www.google.com>” o “<https://www.xojo.com>”) en el TextField y pulsa el PushButton. Observa que en macOS, debes de usar HTTPS.

Deberían de mostrarse los contenidos de la página como texto HTML en el TextArea. Si no es así, deberías de ver el mensaje de error. Para ver un ejemplo, introduce “microsoft.apple.google” en el TextField y haz clic en el PushButton. Probablemente veas el mensaje “Unsupported URL” en el TextArea. Esto significa que URLConnection no entiende el protocolo porque el URL no empieza con HTTPS://.

10) Sal de la aplicación.

11.5 Enviar email



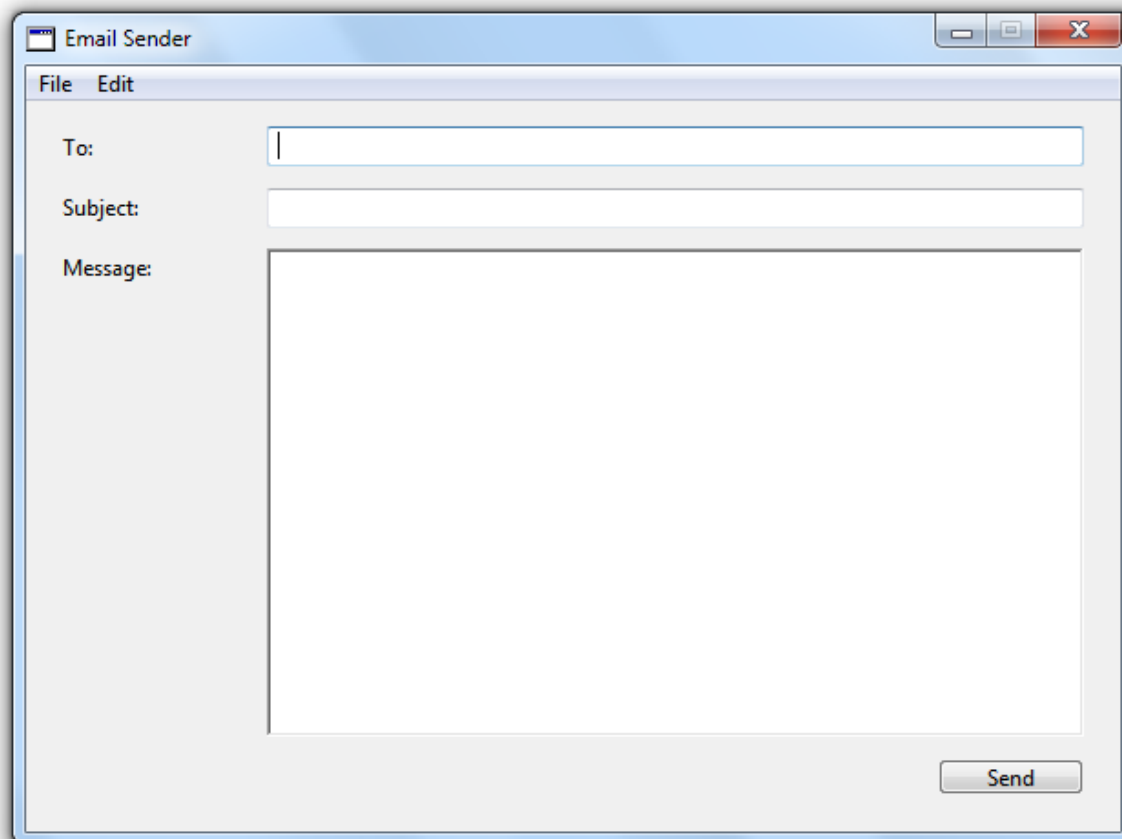
En esta sección crearás el proyecto “Email Sender” y también aprenderás sobre el SMTPSocket de Xojo. SMTP significa Simple Mail Transfer Protocol, y es el protocolo usado con más frecuencia para el envío de mensajes de correo electrónico por Internet. Para esta sección necesitarás una cuenta de email de Gmail, Yahoo, o iCloud.

Observa que SMTP se utiliza sólo para el envío de mensajes. La recepción de mensajes se realiza mediante uno de estos protocolos: POP (Post Office Protocol) e IMAP (Internet Message Access Protocol). Dado a que la recepción de mensajes es algo decididamente más complicado que enviarlos, este proyecto se

centrará en enviarlos. A primera vista puede resultar un tanto simplón la creación de una app que envía emails sin recibirlos, pero lo cierto es que se trata de algo bastante frecuente. Muchas aplicaciones incluyen soporte para enviar reportes de fallos al desarrollador; algo que se realiza mediante el envío de un email. Además, muchas aplicaciones tienen una opción “Share” que a menudo incluye el envío de enlaces por email a tus amigos.

- 1) **Crea un nuevo proyecto Xojo desktop y guárdalo como “MailSender”.**
- 2) **Añade un TCPSocket a Window1 y define su nombre como “MailSocket”.**
- 3) **En el Inspector, define la propiedad Super a SMTPSecureSocket.**
- 4) **Añade dos TextFields, llamados “ToField” y “SubjectField” a Window1. Añade las correspondientes Label con cada uno.**
- 5) **Añade un TextArea llamado “MessageArea”, también con su Label asociada.**
- 6) **Añade un Botón con “Send” como su Caption.**

Tu interfaz puede similar a esta, pero siempre eres libre de usar tu propia creatividad:



7) Crea un nuevo método llamado “SendTheMessage”.

```
Var m As EmailMessage
m = New EmailMessage
m.AddRecipient(ToField.Value)
m.Subject = SubjectField.Value
m.BodyPlainText = MessageArea.Value
m.FromAddress = "example@DoNotUseThisExample.com"
MailSocket.Messages.AddRow(m)
MailSocket.SendMail
```

Este método reunirá la información de tu interfaz de usuario y la usará en un objeto EmailMessage, que de hecho será gestionado por un MailSocket

para su envío. La mayoría de las propiedades de la clase EmailMessage son autoexplicativas gracias a sus nombres. Observa que la propiedad FromAddress debería ser configurada con tu propia dirección de email.

Un método interesante de la clase EmailMessage es AddRecipient. Este puede usarse múltiples veces para un mismo mensaje, de modo que se añadan varias personas al campo “to” del email.

El SMTPSecureSocket (así como el SMTPSocket) tiene un array de EmailMessages denominado Messages. Puedes añadir tus propios mensajes a la cola agregando un EmailMessage a dicho array, tal y como se ha hecho en el anterior código. Una vez que se han añadido todos los mensajes necesarios, se llama al método SendMail de SMTPSecureSocket. El método SendMail envía todos los EmailMessages del array Messages. Con cada uno de los mensajes enviados, el socket dispara el evento MessageSent. Este evento te proporciona una variable llamada Email, la cual es una instancia de la clase EmailMessage que representa el último mensaje enviado. Una vez que se han enviado todos los mensajes, el Socket dispara el evento MailSent. En este punto, el array Messages está vacío y el SMTPSecureSocket está listo para usarlo de nuevo.

8) Añade este código al evento MailSent del MailSocket:

```
MessageBox("All mail has been sent!")
```

9) Añade este código al evento ServerError del MailSocket:

```
MessageBox(ErrorMessage)
```

Usando el evento MailSent te permitirá mantener informado a tu usuario sobre el estado de la app, en vez de que se tenga que preguntar si ha

ocurrido algo. El evento `ServerError` permitirá que tu usuario sepa si algo ha ido mal.

Ahora llega la parte “peculiar”. Como se ha mencionado anteriormente, este proyecto requiere que tengas una cuenta de email con Gmail, iCloud o Yahoo. Sin embargo, cada uno de estos sistemas de correo utiliza un método ligeramente distinto para conectar y autenticar; por lo que el siguiente código en el evento `Action` del `PushButton` puede ser distinto para cada caso.

10) Si estás usando Gmail, procede con el Paso 11. Para iCloud, dirígete al Paso 12. Para Yahoo! Mail, ve al Paso 13.

11) Gmail - Introduce este código en el evento `Pressed` del `PushButton`:

```
MailSocket.Address = "smtp.gmail.com"
MailSocket.Username =
    = "YourGMAILUsername@GMAIL.com"
MailSocket.Password = "YOUR GMAIL PASSWORD"
MailSocket.Port = 465
MailSocket.ConnectionType = SMTPSecureSocket.TLSv1
MailSocket.SMTPConnectionType =
    SMTPSecureSocket.SMTPConnectionTypes.SSLTLS
MailSocket.Secure = True
MailSocket.Connect
SendTheMessage
```

Observa que tu nombre de usuario debe ser tu dirección de email completa, incluyendo el símbolo @. Para conectar con Gmail, has de activar la opción “allow less secure apps”, tal y como se describe el documento de soporte de Google:

<https://support.google.com/accounts/answer/6010255>

Salta al paso 14.

12) iCloud - Introduce este código en el evento `Pressed` del `PushButton`:

```
MailSocket.Address = "smtp.mail.me.com"
MailSocket.Username =
    = "YourICLOUDUsername@icloud.com"
MailSocket.Password = "YOUR ICLOUD PASSWORD"
MailSocket.Port = 587
MailSocket.ConnectionType = SMTPSecureSocket.TLSv1
MailSocket.Secure = True
MailSocket.Connect
SendTheMessage
```

Observa que el nombre de usuario debe ser tu dirección de email completa, incluyendo el símbolo @. Salta al Paso 14.

13) Yahoo! Mail - Introduce este código en el evento `Pressed` del `PushButton`:

```
MailSocket.Address = "smtp.mail.yahoo.com"
MailSocket.Username = "YourYAHOOUsername@yahoo.com"
MailSocket.Password = "YOUR YAHOO PASSWORD"
MailSocket.Port = 465
MailSocket.Secure = True
MailSocket.Connect
SendTheMessage
```

Observa que tu nombre de usuario ha de ser sólo parte de tu dirección de email, la situada antes del símbolo @.

14) Ejecuta el proyecto.

15) Introduce una dirección de email, un asunto y un mensaje.

16) Haz clic en el Botón para enviar tu mensaje.

¿Ves un mensaje de éxito o de error?

17) Sal de la aplicación.

En todos los listados de código anteriores, tu nombre de usuario y contraseña han de estar entre comillas dobles, puesto que son strings. Del mismo modo, observa que la inclusión de nombres de usuario y contraseñas en tu código fuente no es una buena práctica. No importa cual sea el propósito de este proyecto, pero una ampliación interesante del mismo sería añadir campos adicionales a tu interfaz para la introducción del nombre de usuario y contraseña, de modo que no se guarden como parte de tu código fuente.

Si los anteriores ejemplos de Gmail, iCloud y Yahoo no están funcionando en tu caso, entonces deberías de comprobar los ajustes SMTP en los servicios para asegurarte de que están utilizando el puerto recomendado, la seguridad y los servidores de este tipo de servicios suelen cambiar de cuando en cuando.

11.6 Algunas anotaciones sobre los Protocolos

Los protocolos se encuentran generalmente en dos categorías: protocolos existentes y bien establecidos, y protocolos que creas tu mismo. Un protocolo existente podría ser como HyperText Transfer Protocol (HTTP), utilizado para enviar datos a un navegador web, o los protocolos utilizados en el correo electrónico: Post Office Protocol (POP) y Simple Mail Transfer Protocol (SMTP). Estos protocolos son aceptados generalmente por los grupos de la industria tecnológica.

Tu puedes crear tus propios protocolos, si bien esto llevar apareado un gran trabajo.

Xojo te ofrece algo para que la creación de tus propios protocolos sea una tarea algo más sencilla, siempre y cuando los únicos dispositivos que lo utilicen sean otros ordenadores que ejecuten aplicaciones creadas con Xojo.

La clase EasyTCPSocket te permite crear tus propios protocolos sólo para Xojo. Si quieres aprender más sobre EasyTCPSocket, por favor, consulta los proyectos de ejemplo que acompañan a Xojo.

Filas y Columnas

CONTENIDOS

1. Información del Capítulo
2. Introducción a las Bases de Datos
3. Introducción a las Consultas
4. Crear y Conectar con Bases de Datos Locales
5. Acceder a la Base de Datos Xojo
6. Más sobre Bases de Datos

12.1 Información del Capítulo

Muchas personas encuentran intimidante el trabajo con bases de datos, pero las bases de datos están por todas partes. Puede que no te des cuenta de ello, pero usas bases de datos prácticamente a diario. Cada vez que buscas en Google, te desplazas por entre los contactos de tu teléfono o usas Twitter, estás accediendo a una base de datos.

En términos sencillos, una base de datos es una colección organizada de datos. Una característica clave de una base de datos es la capacidad de hacerle preguntas sobre los datos... y recibir respuestas. Por supuesto, debes aprender cómo realizar las preguntas correctas de la forma correcta.

En este capítulo aprenderás un poco sobre teoría de bases de datos, aprenderás algunos fundamentos de SQL (Structured Query Language), el lenguaje utilizado para hablar con las bases de datos, y crearás una aplicación sencilla de contactos: Address Book.

12.2 Introducción a las Bases de datos

Por ahora, tienes que comprender cuatro conceptos clave sobre las bases de datos: tablas, columnas, filas y consultas.

Una tabla de una base de datos es parecido a una clase en Xojo (o cualquier otro lenguaje de programación). Al igual que en una clase, esta representa un objeto del mundo real (como una persona) o bien un concepto abstracto (como una reserva de hotel).

Si recuerdas el Capítulo 8, recordarás que las clases tienen propiedades, y que cada propiedad tiene ciertos tipos de datos. La tabla de una base de datos es muy similar, salvo que estas propiedades se denominan Columnas en vez de propiedades. Cada columna de una tabla representa algún atributo del objeto o concepto que representa. Por ejemplo, una base de datos llamada “Personas” puede tener columnas para cosas como el nombre y el apellido, el número de teléfono o el título para dicha persona. Al igual que con las propiedades de una clase, las columnas en la tabla de una base de datos también han de tener un tipo de dato asociado. Algunos tipos de datos comunes son los listados a continuación junto con su equivalente en Xojo.

Tipo de dato en Base de Datos	Equivalente Xojo
VARCHAR	String
INTEGER	Integer
DATE	Date
DOUBLE	Double
BOOLEAN	Boolean

Afortunadamente, puedes ver que la mayoría de los tipos de datos tienen una correspondencia prácticamente exacta. Sólo los tipos de dato textuales tienen nombres distintos (aunque algunas bases de datos soportan muchos otros tipos de datos, no has de preocuparte por ellos hasta que tengas un conocimiento avanzado en el trabajo con bases de datos).

Con estos tipos de datos en mente, puedes tener una tabla de base de dato similar a la siguiente:

Nombre de Columna	Tipo de Dato
id	INTEGER
last_name	VARCHAR
first_name	VARCHAR
nickname	VARCHAR
birthdate	DATE

Observa que los nombre de columna en el anterior ejemplo utilizan el guión bajo. Al igual que ocurre con las variables en Xojo y otros lenguajes de programación, los nombres de las columnas en una base de datos no pueden contener espacios. Y si bien la mayoría de las bases de datos pueden utilizar mayúsculas y minúsculas para los nombres de las columnas, es una práctica común en el desarrollo de bases de datos usar el guión bajo para separar las palabras en el nombre de una columna. Probablemente veas nombres de columna de caja mixta en bases de datos como SQLite.

También recordarás que una clase es realmente un plano, y no realmente el concepto que representa. En tu código trabajas con instancias de la clase, muchas veces llamados objetos, creados mediante el operador New. Una base de datos es muy similar. La tabla es el plano, y cada instancia de lo que representa se denomina fila (o row, en inglés). Una fila de nuestra tabla personas podría ser como esta:

id	last_name	first_name	nickname	birth_date
1	Hewson	Paul	Bono	1960-05-10

Es posible que también te preguntes qué pasa con la fecha de nacimiento. En las bases de datos, las fechas se almacenan de forma universal con lo que se denomina formato ISO: los cuatro dígitos del año, seguido por los dos dígitos del mes y seguido por los dos dígitos del día. Por lo tanto, la fecha en la tabla anterior

sería el 10 de Mayo de 1960. El formato ISO fue adoptado como estándar por la International Standards Organization. Otros formatos de fecha pueden resultar ambiguos. Por ejemplo, 12/10/2005 es el 10 de Diciembre de 2005 para los Americanos, pero el 12 de Octubre en muchos otros países.

Estos tres conceptos: tablas, columnas y filas, son las partes más críticas de entender para una base de datos; y en realidad no son complicadas.

12.3 Introducción a las Consultas de Bases de Datos

Pero, como se ha mencionado anteriormente, hay cuatro conceptos que debes saber por ahora... no sólo tres. El cuarto es la consulta. Una consulta es un modo de comunicarse con la base de datos. Esto se realiza mediante algo denominado Structured Query Language o SQL. Prácticamente todos los sistemas de bases de datos tienen una versión ligeramente personalizada de SQL, pero los fundamentos son los mismos.

SQL no es el único nombre relacionado con las bases de datos. Hay otras alternativas de SQL como son Cassandra, MongoDB, CouchDB y Memcached. Si bien algunas de estas bases de datos son muy rápidas y están diseñadas para funcionar a una escala masiva, no es muy frecuente encontrarlas en aplicaciones desktop, móviles o en sitios web más pequeños que Facebook y Google.

Todo buen lenguaje de programación tiene verbos, y SQL no es una excepción. SQL tiene cuatro verbos principales que has de conocer: SELECT, INSERT, UPDATE, y DELETE. Si te estás preguntando sobre el por qué se han escrito en mayúsculas, es porque se trata de una convención habitual la de escribir en

mayúsculas los comandos SQL. Las minúsculas también funcionan, pero cuando comienzas a mezclar consultas SQL con tu código Xojo, las mayúsculas hacen que destaquen más.

Estos cuatro verbos describen sus propias funciones. Se usa SELECT para obtener datos de la base de datos. Se usa INSERT para añadir datos a una base de datos. Se usa UPDATE para modificar datos ya existentes en la base de datos. Y se usa DELETE para eliminar datos de la base de datos. Estos cuatro verbos funcionan con columnas y filas. Hay otros verbos para trabajar con bases de datos: se usa CREATE para añadir una tabla, ALTER para modificar una tabla, y DROP para borrar una tabla.

Para ver como se combinan, este es el código SQL para crear tu propia tabla “people”:

```
CREATE TABLE people (  
  id INTEGER NOT NULL UNIQUE,  
  last_name VARCHAR NOT NULL,  
  first_name VARCHAR NOT NULL,  
  nickname VARCHAR,  
  birthdate DATE,  
  PRIMARY KEY(id));
```

La parte CREATE TABLE es autoexplicativa: crea una tabla en la base de datos. La palabra “people” es el nombre de la tabla a crear. Dentro de los paréntesis se encuentra una lista con el nombre de las columnas y sus respectivos tipos de datos, junto

con alguna información adicional para cada columna. Por ejemplo, puedes ver que la columna “last_name” será VARCHAR (o texto) pero, ¿qué significa el NOT NULL que le sigue?

Este es un ejemplo de un límite. NOT NULL significa que la columna no puede contener datos nulos, incluso si se trata de una cadena vacía. En las bases de datos, NULL es equivalente a la nada, de modo que una cadena vacía es NOT NULL. NULL significa que el valor no existe de ninguna forma posible. Para aclararlo, si alguien te hiciese una pregunta y tu no tuvieses una respuesta, eso sería similar a NULL o “nada”. Por otra parte, si tienes la respuesta pero optases por no darla, eso sería más parecido a una cadena vacía (NOT NULL).

Otra limitación listada anteriormente está en la columna “id”. UNIQUE. Esto significa que cada fila de la tabla debe tener un valor que no esté presente en otra fila. En otras palabras, ha de ser único. El uso de “id” como columna es muy frecuente en el diseño de bases de datos, puesto que proporciona un identificador único para cada fila, lo que facilita enormemente las operaciones de selección, actualización y borrado sobre una fila en concreto posteriormente.



La última línea, “PRIMARY KEY(id)” es similar a las restricciones NOT NULL y UNIQUE en la columna “id”. PRIMARY KEY indica a la base de datos que debe asegurarse de que “id” exista, sea único y que esté garantizado el que siempre se va a referir a su fila. Esto es también una restricción, pero un límite de tabla en vez de tratarse de una restricción de columna.

Todas estas restricciones pueden que te lleve a preguntarte por qué preocuparte por ellas. Después de todo, puedes escribir tu código de forma que se asegure de que un cierto valor siempre exista o que nunca entre en conflicto con otro valor. Pero la belleza de las restricciones reside en que será la base de datos quien trabaje por ti. Con una única configuración, puedes confiar ahora en que la base de datos lanzará un error cuando algo vaya

mal, en vez de confiar en que tu mismo recuerdes posteriormente cada detalle.

Ahora que se ha creado tu (imaginaria) tabla “people”, necesitas añadir algunos datos en ella. Esto es lo que se hace mediante el comando INSERT. La estructura básica de INSERT tiene el siguiente aspecto:

```
INSERT INTO [table_name]
(column1, column2, column3, ... columnN)
VALUES ('value1', 'value2', 'value3', ... 'valueN')
```

Si bien se han utilizado varias líneas para este ejemplo, un comando SQL puede expresarse en una única línea, siempre y cuando dejes un espacio entre cada elemento (como por ejemplo, tras el nombre de la tabla).

Evidentemente, deberías sustituir [table_name] por el nombre real de la tabla sobre la cual vayas a insertar datos; al igual que con las columnas listadas. Los valores listados en el segundo grupo de paréntesis se corresponden con una columna en el primer grupo de paréntesis; de modo que en el ejemplo anterior se corresponderían de la siguiente forma:

Columna	Valor
column1	value1
column2	value2
column3	value3

Puedes indicar cualquier cantidad de columnas en tu INSERT, siempre y cuando exista un valor asociado a cada una de ellas.

Puede que hayas observado que nuestros valores en INSERT están encerrados en comillas simples. En SQL, cada valor de cadena (o VARCHAR, en términos de bases de datos) debe estar contenido entre comillas simples. La misma regla se aplica a los valores de fecha y a lo booleanos. Los datos numéricos no precisan de comillas simples. Por tanto, el anterior ejemplo podría ser como siguiente utilizando datos reales:

```
INSERT INTO people
(id, last_name, first_name, nickname, birthdate)
VALUES (1, 'Hewson', 'Paul', 'Bono', '1960-05-10')
```

Esto lanza una pregunta interesante: ¿Qué pasa si tus datos de texto ya tienen comillas simples? Si recuerdas lo aprendido acerca de las variables de cadena en los anteriores capítulos, puedes “escapar” una comilla doble en Xojo mediante el uso de dobles comillas consecutivas. Lo mismo se aplica para las comillas simples en SQL (observa las dos comillas simples en el último nombre):

```
INSERT INTO people
(id, last_name, first_name)
VALUES (2, 'O''Henry', 'Thomas')
```

Tras haber ejecutado estos dos comandos INSERT, nuestra tabla “people” será como esta:

id	last_name	first_name	nickname	birth_date
1	Hewson	Paul	Bono	1960-05-10
2	O’Henry	Thomas	NULL	NULL

Observa los dos valores NULL en la segunda fila; esto se debe a que los valores no se han especificado como parte del INSERT.

Ahora que tienes datos en la tabla, puedes usar el comando SELECT. El comando SELECT lista qué columnas quieres de la tabla especificada:

```
SELECT last_name, first_name FROM people
```

Dicha consulta devolvería este conjunto de datos:

last_name	first_name
Hewson	Paul
O’Henry	Thomas

Observa que la base de datos sólo devuelve las columnas indicadas y en el orden indicado. El orden de las filas, sin embargo, no se ha especificado, salvo que uses la cláusula ORDER BY:

```
SELECT last_name, first_name FROM people
```

```
ORDER BY last_name, first_name
```

En este ejemplo concreto, el conjunto de datos se ve igual, pero se habrá garantizado el orden de las filas.

Verás con frecuencia consultas SQL que comienzan con un SELECT *, como en “SELECT * FROM PEOPLE”. El * indica a la base de datos que quieres todas las columnas de la tabla. Esto resulta útil para depurar, pero no es recomendable para código de producción por dos motivos. El primero, no podrás estar seguro del orden de las columnas devueltas por la base de datos. En segundo lugar, podrías estar recuperando más datos de los necesarios, ralentizando tu aplicación y también usando más espacio en disco o recursos de red.

También puedes buscar filas concretas utilizando la cláusula WHERE:

```
SELECT last_name, first_name FROM people  
WHERE last_name = 'Hewson'
```

Esto debería devolver sólo la primera fila, dado que es la única fila donde la columna last_name es igual a ‘Hewson’.

También se puede usar la cláusula WHERE con rangos de datos:

```
SELECT last_name, first_name, birthdate FROM people  
WHERE birthdate > '1900-01-01'  
AND birthdate < '2012-01-01'
```

También puedes hacer una búsqueda de una cadena parcial, usando el símbolo % (a veces llamado comodín dado que encuentra cualquier texto) y el operador LIKE:

```
SELECT last_name, first_name FROM people
WHERE last_name LIKE '%nr%'
```

Esto debería devolver el registro Thomas O'Henry's.

El símbolo comodín, o %, encuentra el texto de cualquier longitud (incluyendo cero caracteres); de modo que una búsqueda de 'B%' encontraría registros que terminasen con una b minúscula. Una búsqueda de registros con coincidencias '%b%' encontraría cualquier registro que contenga una b en cualquiera de sus posiciones.

Otra cosa a tener en cuenta cuando se hacen SELECTs es que prácticamente cualquier base de datos SQL es sensible a la caja (mayúsculas/minúsculas) del texto; de modo que trata 'Hewson' de forma distinta a 'hewson' o 'HEWSON'. Cada una es una cadena diferente por lo que respecta a la base de datos. Para forzar una búsqueda no sensible a mayúsculas/minúsculas, la mayoría de las bases de datos soportan la función LOWER, de modo que se fuerza a que todo esté en minúsculas:

```
SELECT last_name, first_name FROM people
WHERE LOWER(last_name) LIKE ('%hewson%')
```

Esto debería devolver el registro de Bono, puesto que la función LOWER ha convertido todo a minúsculas antes de comparar las cadenas.

Por último, también puedes buscar valores NULL mediante el operador IS:

```
SELECT last_name, first_name FROM people
WHERE birthdate IS NULL
```

O también:

```
SELECT last_name, first_name FROM people
WHERE birthdate IS NOT NULL
```

La actualización de los datos actuales es un proceso similar utilizando el comando UPDATE. Por favor, advierte que el 99,9% de las veces que utilizas el comando UPDATE también querrás usar la cláusula WHERE.

```
UPDATE people SET birthdate = '1980-10-01'
WHERE id = 2
```

También puedes actualizar múltiples columnas:

```
UPDATE people SET nickname = 'Tommy',
birthdate = '1980-10-01'
WHERE id = 2
```

Si eliminas la cláusula WHERE y ejecutas un comando como este, entonces estarías cambiando la fecha de nacimiento en cada fila de la tabla:

```
UPDATE people SET birthdate = '1980-10-01'
```

Un comando UPDATE o DELETE, que carezca de la cláusula WHERE, actualizaría o borraría cada fila de la tabla, algo que probablemente no es lo que quieres hacer la mayoría de las ocasiones.

Merece la pena repetirlo, con énfasis: *un comando UPDATE o DELETE sin cláusula actualizará o borrará todas las filas de la tabla, algo que seguramente no sea lo que quieres hacer en la mayoría de los casos.*

La cláusula WHERE en un comando UPDATE funciona del mismo modo al que lo hace con el comando SELECT; de modo que puedes usar LIKE, IS NULL y otros ejemplos como se ha visto.

Por último, el comando DELETE es prácticamente idéntico al comando UPDATE con una excepción: no necesitas indicar las columnas, dado que borra toda la fila:

```
DELETE FROM people WHERE id = 2
```

Esto borraría el registro de Thomas O'Henry. Para borrar todas las filas:

```
DELETE FROM people
```

Recuerda, omitir la cláusula WHERE borrará todo en la tabla. Si esto es lo que quieres hacer, entonces es muy útil, pero utiliza dicho comando con cautela.

12.4 Crear y Conectar a una Base de Datos Local

Ahora que has visto algunos fundamentos SQL, es el momento de aprender a crear un archivo de base de datos en Xojo. Hay dos tipos de conexiones de bases de datos: local y remota. Este libro te ayudará a aprender como crear y trabajar con un archivo de bases de datos local; lo que es ciertamente un archivo en tu ordenador que contendrá todos los datos.



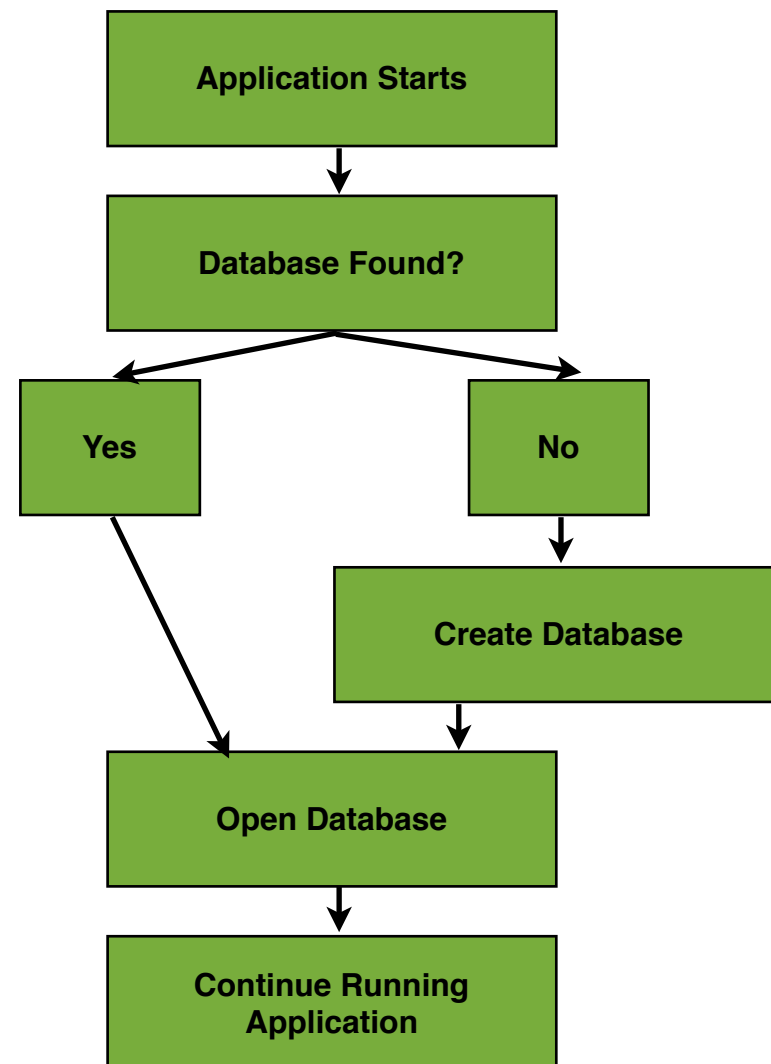
Una base de datos remota, o servidor de bases de datos, es un animal distinto. Aunque interactúas con ella del mismo modo, una base de datos remota está funcionando en un servidor de bases de datos, generalmente en otro ordenador. Por lo general,

muchas personas pueden acceder al servidor de bases de datos de forma simultánea. Algunos servidores de bases de datos comunes que puedas encontrar son Microsoft SQL Server, Oracle y las bases de datos de código abierto como MySQL y PostgreSQL. Estos tipos de bases de datos pueden requerir de hardware y software especializado, y necesitan por lo general un administrador de bases de datos para mantenerlas, puesto que pueden llegar a ser bastante complejas.



Pero una base de datos local sólo necesita un archivo. En las próximas secciones aprenderás a crear una base de datos local, como conectarte a ella y como extraer datos de ella. En el proceso, crearás el proyecto de ejemplo de este capítulo: una Agenda de email simple.

Crea un nuevo proyecto Xojo y guárdalo como “AddressBook”. El siguiente diagrama de flujo representa el funcionamiento de la aplicación:



1) Añade dos nuevas propiedades a Window1: MyDatabase como SQLiteDatabase y MyDBFile como FolderItem.

MyDatabase será la variable que mantendrá una referencia a la base de datos que estés usando y MyDBFile representa el archivo de base de datos en tu ordenador.

2) Crea un nuevo método en Window1 llamado “CreateSchema”.

```
Var sql As String
sql = "CREATE TABLE addressbook ( "
sql = sql + "name varchar, "
sql = sql + "email varchar"
sql = sql + " )"
Try
    MyDatabase.ExecuteSQL(sql)
Catch error As DatabaseException
    MessageBox(error.Message)
End Try
```

Este método tendrá como trabajo la creación de la tabla de tu base de datos. Para ello, crearás un String con los comandos SQL. Para crear la tabla (como has visto anteriormente) has de pasar la cadena sql al método de la base de datos llamado ExecuteSQL, el cual, como su propio nombre implica, ejecuta el comando SQL en la base de datos.

El bloque Try...Catch indica a tu código lo que ha de hacer en el caso de que ExecuteSQL falle y provoque una excepción (error). Aprenderás más sobre esto en el capítulo 14.

La creación de un esquema en código no es infrecuente, pero muchas personas prefieren usar una herramienta visual. Hay disponibles muchas aplicaciones para la creación de esquemas, incluyendo MySQL Workbench, SQL Developer de Oracle y SQL Server Management Studio de Microsoft. La mayoría de estas herramientas son específicas para un servidor de bases de datos concreto, pero también existen otras que son agnósticas con respecto a la base de datos utilizada.

3) Crea otro método en Window1 llamado "CreateDatabase" con un tipo de dato devuelto como Booleano.

```
MyDBFile = SpecialFolder.ApplicationData.Child("Ch12")
If MyDBFile <> Nil Then
    MyDatabase = New SQLiteDatabase
    MyDatabase.DatabaseFile = MyDBFile
    Try
        MyDatabase.CreateDatabase
        MyDatabase.Connect
        CreateSchema
        Return True
    Catch error As DatabaseException
        Return False
    End Try
End If
```

Este método devolverá True si crea la base de datos con éxito, y Falso si no es así. Este método funciona asignando un FolderItem a la propiedad DatabaseFile de la base de datos. La creación de la base de datos, la

	Mac	Windows	Linux
Application Data	/Users/ UserName/ Library/ Application Support	\Users\User Name\AppData Local\Roamin g\	/home/ UserName/

conexión a ella y la creación del esquema se hace en una instrucción Try...Catch. Esto está diseñado para intentar ejecutar los comandos, y atrapar el error en el caso de que cualquiera de ellos falle. En este caso, devuelve False cuando se produce un fallo. La creación de la base de datos se realiza mediante un método llamado CreateDatabase. A continuación intenta conectarse a la nueva base de datos usando el método Connect. Por último, llama al método CreateSchema que has escrito anteriormente para crear la tabla addressbook.

Este código utiliza el módulo SpecialFolder para obtener la ubicación de un archivo en el sistema. En este caso SpecialFolder.ApplicationData se refiere a una de las carpetas que puedes ver en la tabla superior, dependiendo del OS que estés usando.

4) Añade un método llamado "OpenDatabase" con un tipo devuelto de Booleano.

```
MyDBFile = SpecialFolder.ApplicationData.Child("Ch12")
If MyDBFile.Exists Then
    MyDatabase = New SQLiteDatabase
    MyDatabase.DatabaseFile = MyDBFile
    Try
        MyDatabase.Connect
        Return True
    Catch error As DatabaseException
        MessageBox("Error connecting to database.")
    End Try
```

```
Else  
    Return CreateDatabase  
End If
```

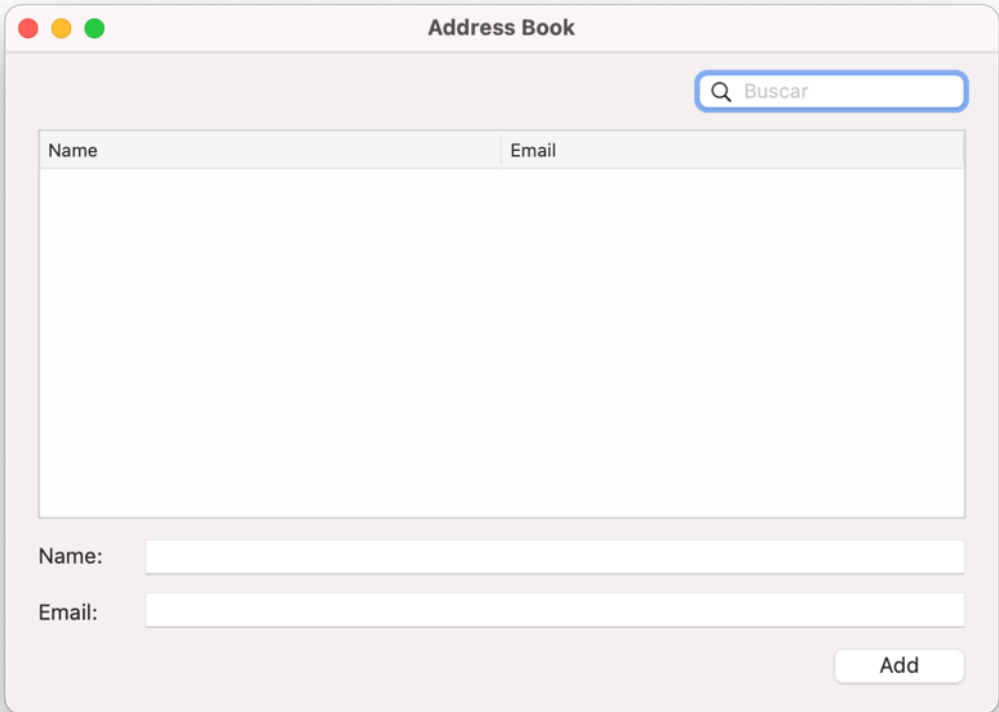
Este método, que devuelve un Booleano, intentará abrir el archivo de base de datos. Si tiene éxito, la app continuará con normalidad. Si no, ejecutará el método CreateDatabase y devolverá un resultado booleano.

5) Para empezar este proceso, añade este código al evento Opening de Window1:

```
If Not OpenDatabase Then  
    MessageBox("Unable to open or create database.")  
    Quit  
Else  
    Populate //You'll create this method ↵  
            in the next section  
End If
```

12.5 Acceder a la Base de Datos Xojo

Ahora que tu archivo de base de datos está configurada y tu esquema definido, tienes que aprender acerca de cómo leer información de la base de datos e insertar información en la base de datos usando Xojo. El código introducido en el evento Opening de Window1, a final del último capítulo, se refiere al método que aun no has creado, llamado Populate. El trabajo de dicho método será el de hablar con la base de datos y recuperar cualquier dato que encuentre, incorporando luego dichos datos al ListBox de Window1. También necesitarás otros controles, y mientras que estés añadiendo el ListBox, será un buen momento para diseñar el resto del proyecto de ejemplo. Usa tu propia creatividad para diseñar la interfaz de usuario. Podría tener el aspecto mostrado a continuación:



Esta es una lista de los controles que necesitarás, así como de los nombres que deberías asignarles:

Control	Nombre
ListBox	AddressesBox
SearchField	Search
TextField	NameField
TextField	EmailField
PushButton	AddButton

De igual modo, añade una Label para NameField y EmailField.

Estás prácticamente listo para el método `Populate`. Sin embargo necesitas una cosa antes, es una pequeña función llamada `SQLify`. Si recuerdas del anterior capítulo, SQL utiliza una comilla simple como delimitador de texto, lo que significa que siempre que tengas una en tus datos, las cosas pueden ir terriblemente mal. Esto es por lo que necesitas “escapar” cada comilla simple duplicándola. Puedes hacerlo manualmente cada vez que hables con la base de datos utilizando la función `ReplaceAll` de Xojo, pero terminarías repitiendo el mismo código una y otra vez, lo que viola una de las reglas de la programación: No te repitas. En vez de ello, crea un método en `Window1` llamado “`SQLify`”. Tomará un parámetro, `Source As String`, y devolverá una `String`. Su código es muy simple, utilizando la función `ReplaceAll` de Xojo para encontrar cada comilla simple y sustituirla por dos comillas simples:

```
Var result As String
result = ReplaceAll(Source,"'","''")
Return result
```

De modo que si por ejemplo ejecutas este código:

```
MessageBox(SQLify~
(("If it ain't broke, don't fix it!"))
```

Verás un mensaje con este texto:

```
If it ain''t broke, don''t fix it!
```

Observa que no estás viendo las marcas de cita normales incluidas en el texto: estas son dos comillas simples una tras la otra. Puede resultar difícil distinguirlas con determinadas fuentes.

Ahora que `SQLify` está escrita, ya estás listo para el método `Populate`. El método `Populate` se basa en tu conocimiento ya adquirido del `ListBox` (en especial los métodos `RemoveAllRows` y `AddRow`), y también introducirá la clase `RowSet`. Un `RowSet` es un conjunto de datos devueltos por una consulta contra la base de datos (utilizando para ello la función `SelectSQL` de la clase `Database`). Una `RowSet` puede contener múltiples filas (rows). Se trabaja con cada fila de una en una. Puedes navegar por las filas usando el método `MoveToNextRow` de `RowSet`. La propiedad `AfterLastRow` del `RowSet` devolverá `True` cuando tu código se mueva más allá de la última fila. Cada fila del `RowSet` puede contener múltiples columnas, que puedes recuperar por su nombre utilizando el método `Column` de `RowSet`. En dicho método utilizas el mismo nombre de columna indicado en el esquema de la base de datos.

Una vez se ha recuperado el `RowSet`, el método `Populate` creará una nueva fila en el `ListBox` para cada fila del `RowSet`, rellenando sus datos a medida que se hace. Pero antes de añadir nuevas filas, se eliminarán todas las filas existentes. Si te saltases este paso, tu `ListBox` podría terminar mostrando datos duplicados y continuaría mostrando duplicados cada vez que se rellenase.

El método Populate también crea una consulta SQL. Si no se ha introducido nada en el SearchField, la consulta será directa, seleccionando simplemente todos los datos de la tabla “addressesbook”. Sin embargo, si el usuario ha introducido algo en el SearchField, entonces Populate tendrá esto en cuenta y devolverá sólo los registros coincidentes (SearchField operará tanto en las columnas name y email).

Con dicha introducción en mente, este es el código para el método Populate:

```
Var sql As String
Var rs As RowSet
sql = "SELECT name, email "
sql = sql + "FROM addressbook "
If Search.Text <> "" Then
    sql = sql + "WHERE LOWER(name) LIKE LOWER('%" &
        SQLify(Search.Text) + "%') "
    sql = sql + "OR LOWER(email) LIKE LOWER('%" &
        SQLify(Search.Text) + "%') "
End If
sql = sql + "ORDER BY name"
Try
    rs = MyDatabase.SelectSQL(sql)
    AddressesBook.RemoveAllRows
    While Not rs.AfterLastRow
        AddressesBook.AddRow(rs.Column("name").StringValue)
        AddressesBook.CellTextAt
            (AddressesBook.LastAddedRowIndex, 1) =
            rs.Column("email").StringValue
        rs.MoveNextRow
    Wend
```

```
Catch error As DatabaseException
    MessageBox(error.Message)
End Try
```

Si no lo has hecho recientemente, este es buen momento para guardar tu proyecto.

Ahora que puedes recuperar datos de la base de datos, necesitas añadir algunos datos de forma que puedas recuperarlos. Esto se llevará a cabo usando el evento Pressed de AddButton. El código escrito aquí creará la instrucción SQL para insertar los datos de los controles NameField y EmailField en la base de datos, utilizando para ello el comando ExecuteSQL de la clase Database. Si te estás preguntando sobre la diferencia entre SelectSQL y ExecuteSQL, es principalmente sobre lo que quieres obtener de la base de datos. Si quieres recuperar datos y necesitas un RowSet, usa SelectSQL. Si estás insertando, actualizando o borrando datos, usa ExecuteSQL; puesto que en estos casos no necesitas un RowSet de vuelta.

El código también mostrará cualquier error que pueda ocurrir. Si no hay errores, entonces hará un commit sobre la base de datos, borrará cualquier dato de los campos y ejecutará el método Populate.

Este es el código para el evento Pressed de AddButton:

```
Var sql As String
sql = "INSERT INTO addressbook ( "
```

```

sql = sql + "name, email"
sql = sql + ") VALUES ("
sql = sql + "'" + SQLify(NameField.Text) + "', "
sql = sql + "'" + SQLify(EmailField.Text) + "'"
sql = sql + ")"
Try
    MyDatabase.ExecuteSQL(sql)
    NameField.Text = ""

    EmailField.Text = ""
    NameField.SetFocus
    Populate
Catch error As DatabaseException
    MessageBox(error.Message)
End Try

```

Si no te gusta poner todas estas consultas SQL juntas, también puedes utilizar la API de la Database en Xojo para insertar el registro sin usar SQL. Este código es una alternativa al código anterior (¡no ejecutes los dos!).

```

Var row As DatabaseRow
row = New DatabaseRow
row.Column("name") = NameField.Text
row.Column("email") = EmailField.Text
Try
    MyDatabase.AddRow("addressbook", row)
Catch error As DatabaseException
    MessageBox(error.Message)
End Try

```

Por último, necesitas asegurarte de que el método Populate se ejecute cada vez que el usuario introduzca texto en el SearchField. Esto se logra añadiendo esta línea de código en el evento TextChanged del SearchField:

Populate

Dado que el método Populate tienen en cuenta el texto introducido en el SearchField no es necesario hacer nada más aquí.

Ejecuta el proyecto y prueba a añadir algunos datos a tu base de datos. Tras haber añadido algunas filas, prueba también la función de búsqueda. Sal de la aplicación.

12.6 Más sobre Bases de Datos

En este capítulo has aprendido bastante sobre bases de datos, y también como interactuar con ellas mediante Xojo. Aunque te has encontrado con una buena cantidad de información sólo has empezado a rascar la superficie de las bases de datos. Dado que las bases de datos forman una parte integral de las aplicaciones modernas, te animo a que aprendas más sobre teoría de bases de datos y SQL.

Todo en Familia: Subclases

CONTENIDOS

1. **Introducción al Capítulo**
2. **Clases y Subclases**
3. **Subclases en la Práctica**



13.1 Introducción al Capítulo

¿Sabías que tu código es como una familia? Puede que suene raro, pero es cierto. Al igual que has heredado ciertas características físicas o rasgos de personalidad de tus padres, partes de tu código puede heredar propiedades y comportamientos de sus padres.

Una gran diferencia es que en el desarrollo de software, uno no se refiere a hijos y padres; sino de subclases y superclases.

En el Capítulo 8 has aprendido sobre las clases. Tal y como leiste, una clase es algo que representa un objeto de la vida real (algo que puedes tocar o apuntar) o bien una idea abstracta (un concepto o idea “intangible”). Por supuesto, en el mundo real, fuera de tu código, las cosas no siempre son así de simples. Algunos tipos diferentes de objetos pueden contar con atributos que se solapan.

En este capítulo aprenderás cómo diferentes clases pueden relacionarse con otras, y crearás un control personalizado que puedes reutilizar en otros proyectos.

13.2 Clases y Subclases

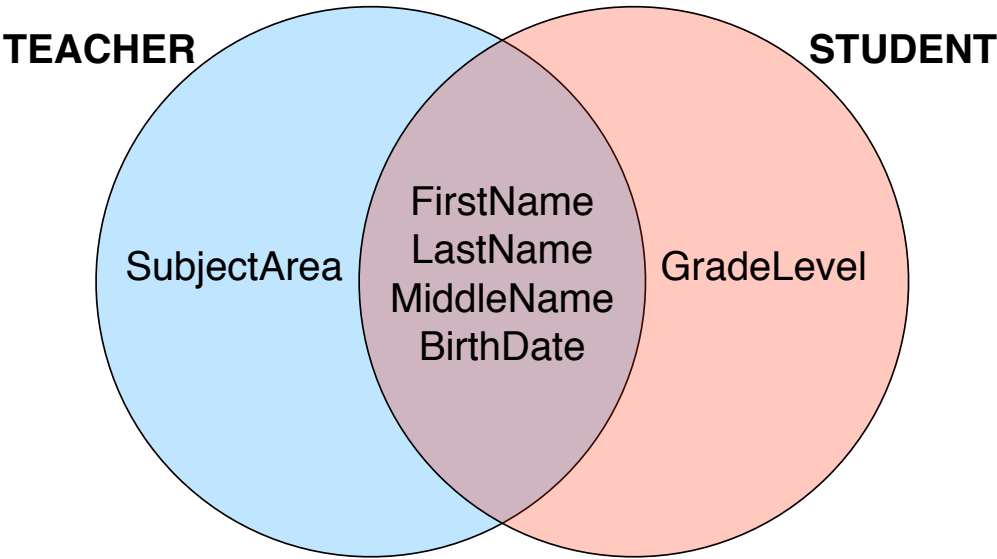
Retomemos la clase Student que creaste en el Capítulo 8. Tienes las siguientes propiedades:

Propiedad	Tipo de Dato
FirstName	String
LastName	String
MiddleName	String
Birthdate	DateTime
GradeLevel	String

Estas propiedades son ciertamente relevantes para un estudiante, pero algunas se podrían aplicar a otras cosas, incluso cosas que puedan estar representadas en tu aplicación, como profesores. De hecho, una clase Teacher podría tener estas propiedades:

Propiedad	Tipo de Dato
FirstName	String
LastName	String
MiddleName	String
Birthdate	DateTime
SubjectArea	String

Esta es otra forma de verlo:



Como puedes ver, algunas de las propiedades son las mismas. De hecho, en este ejemplo, la mayoría lo son.

Esto se debe a que tanto un profesor y un estudiante son ambos *personas*, y la mayoría de la gente tiene determinados atributos en común con otros. Por tanto, esta situación está pidiendo una clase llamada Person (Persona). La clase Person tendrá propiedades comunes, tal y como hemos visto:

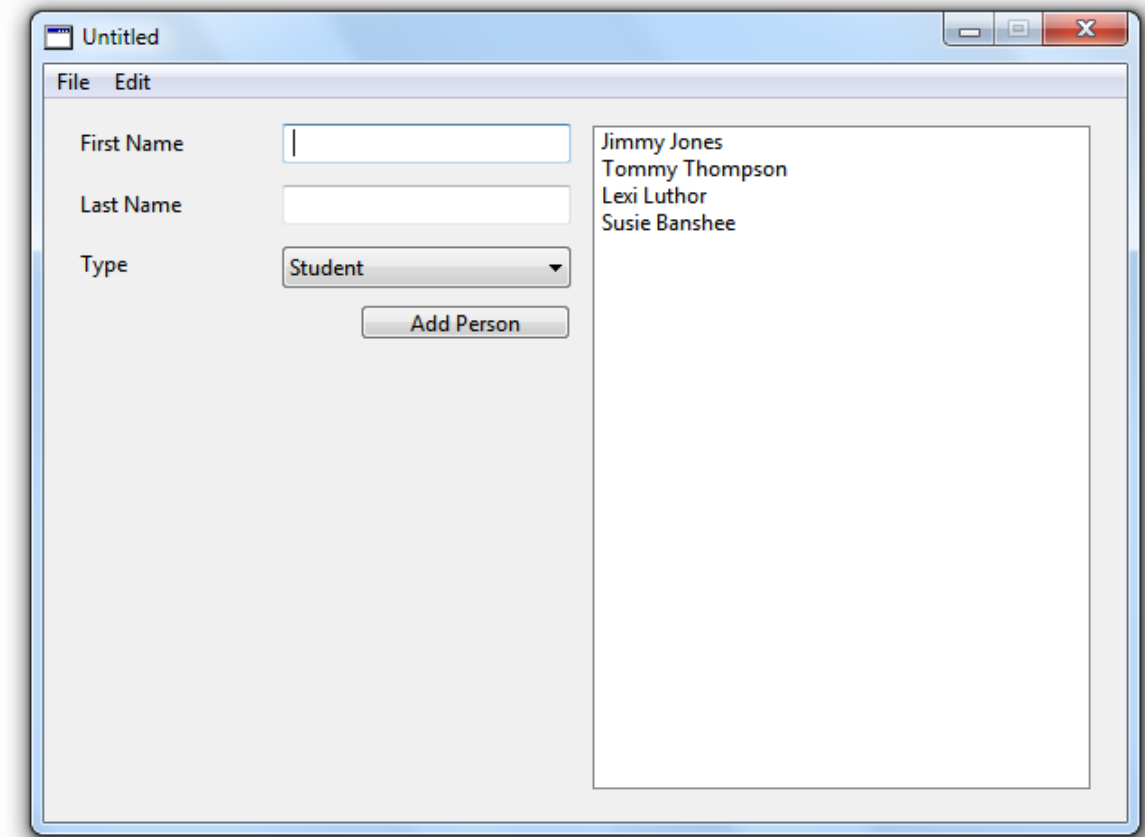
Propiedad	Tipo de Dato
FirstName	String
LastName	String
MiddleName	String
Birthdate	DateTime

Pero aun necesitas una forma de almacenar información y comportamientos para estudiantes y profesores. Aquí es donde

entran en juego las subclases. Una subclase es una clase que deriva propiedades y comportamientos de otra clase, llamada superclase. Esto es similar a la relación padre/hijo descrita anteriormente.

No necesitas hacer nada especial para crear una superclase. Cada clase que creas (y, de hecho, muchas de las incluidas ya en Xojo) pueden ser potencialmente una superclase. Una clase se convierte en superclase a medida que creas una subclase a partir de ella.

Crea un nuevo proyecto Xojo desktop y guárdalo como Subclases. Crearás una pequeña aplicación que te permitirá añadir estudiantes y profesores a un listado común, al tiempo que retienen información sobre ellos. Esta es una previsualización del a interfaz (aunque la tuya puede ser distinta):



- 1) Crea una nueva clase desde el menú **Insert > Class**. Nombre la clase como **"Person"**.
- 2) Añade cuatro propiedades públicas a la clase **Person**: **FirstName** como **String**, **LastName** como **String**, **MiddleName** como **String**, y **BirthDate** como **DateTime**.
- 3) Añade un nuevo método a la clase **Person** llamado **"AnnounceName"**.

```
MessageBox("Hello, my name is " + FirstName +  
          + " " + LastName + "!")
```

4) **Añade otro nuevo método a la clase Person llamado “SetName”.**

```
FirstName = fName  
LastName = lName
```

Este método toma dos parámetros. **fName As String** y **lName As String**. Este te permite definir las propiedades FirstName y LastName con una línea de código.

5) **Crea otra clase nueva, llamada “Student”.**

Cuando crees la clase, define su propiedad Super a Person. Puede que necesites hacerlo escribiendo “Person” en el campo Super o haciendo clic en el botón “edit” y desplazándote por el listado resultante. Student es ahora una subclase de Person. Dado que Student es una *subclase* de Person, heredará automáticamente las propiedades y métodos de Person. Por tanto, Student tiene propiedades para FirstName, LastName, MiddleName y BirthDate; así como los métodos AnnounceName y SetName. Observa que, aunque tiene dichos métodos, estos no se muestran en la clase Student.

6) **Añade una nueva propiedad a la clase Student: GradeLevel como String.**

7) **Repite el anterior proceso para crear otra subclase de Person, llamada “Teacher”.**

8) **Añade una propiedad a Teacher: SubjectArea As String.**

9) **Vuelve a la clase Person y añade otro método: “AnnounceType”.**

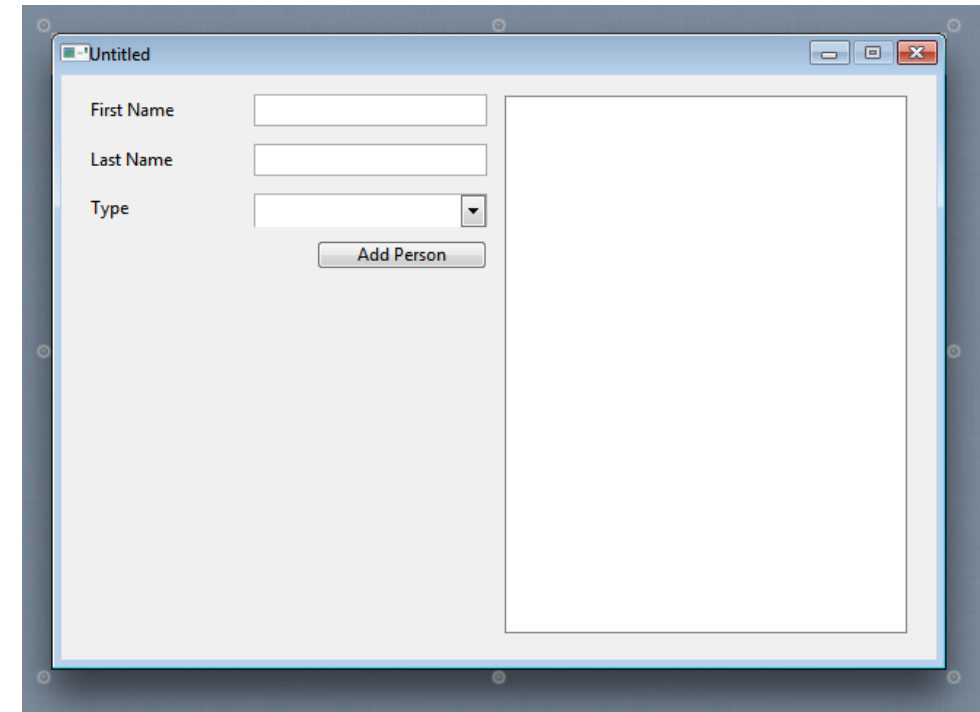
```
If Self IsA Student Then
```

```
    MessageBox("I'm a student!")  
Else  
    MessageBox("I'm a teacher!")  
End If
```

La tarea de este método es determinar si el objeto activo Person es un Student o un Teacher, mostrando el resultado en un mensaje. Para determinar esta información, utilizarás IsA, un operador proporcionado por Xojo que te permite saber si un objeto es también otro tipo de objeto. En este ejemplo, lo usarás para ver si el objeto activo Person es también un Student o un Teacher.

10) **Diseña la interfaz de tu aplicación.**

Nuevamente, puede que tenga un aspecto como este, pero siéntente libre de seguir tu propia creatividad:



11) Añade estos controles a la ventana.

Control	Nombre
TextField (con/Label)	FirstNameField
TextField (con/Label)	LastNameField
PopupMenu (con/Label)	TypeMenu
PushButton	AddButton
ListBox	PeopleList

12) Para poblar TypeMenu con una lista de opciones, utiliza el siguiente código en su evento Opening:

```
Me.AddRow( "Student" )
Me.AddRow( "Teacher" )
Me.SelectedIndex = 0
```

13) Añade este código en el evento Pressed de AddButton:

```
Var t As Teacher
Var s As Student
Var newName As String
newName = FirstNameField.Text &
    " " & LastNameField.Text
If TypeMenu.SelectedIndex = "Student" Then
    s = New Student
    s.SetName(FirstNameField.Text, &
        LastNameField.Text)
    PeopleList.AddRow(newName)
    PeopleList.CellTagAt(PeopleList.
        LastAddedRowIndex, 0) = s
Else
    t = New Teacher
```

```
t.SetName(FirstNameField.Text, &
    LastNameField.Text)
PeopleList.AddRow(newName)
PeopleList.CellTagAt(PeopleList.
    LastAddedRowIndex, 0) = t
End If
FirstNameField.Text = ""
LastNameField.Text = ""
FirstNameField.SetFocus
```

Este código necesitará crear un nuevo objeto Person (ya sea creando un objeto Student o Teacher) y luego añadirá el nombre de esa Persona al ListBox. También pondrá el objeto recién creado en una de las CellTag del ListBox. Antes de crear el objeto, necesitarás saber si vas a instanciar un Teacher o un Student, de modo que tendrás que comprobar lo que dice TypeMenu. Por último, como un poco de limpieza, han de borrarse FirstNameField y LastNameField y volver a poner el foco en FirstNameField.

14) Añade este código en el evento DoublePressed de PeopleList:

```
Var p As Person
If Me.SelectedIndex <> -1 Then
    p = Me.CellTagAt(Me.SelectedIndex, 0)
    p.AnnounceName
    p.AnnounceType
End If
```

Este código ha activado PeopleList para que muestre información sobre el objeto que contiene. Dado que has almacenado un objeto Teacher o Student en una CellTag, puedes recuperar dicho objeto y acceder a sus propiedades y métodos. Recuerda que CellTag es una variante (Variant), de

modo que puede contener cualquier objeto sin tener que mostrarlo en la interfaz de usuario.

15) Ejecuta el proyecto.

16) Añade algunos estudiantes y profesores al listado.

Tus estudiantes y profesores deberían de aparecer en el ListBox. Haz doble clic en unas cuantas entradas y observa si reportan su tipo correctamente.

17) Sal de la aplicación.

Si bien el anterior ejemplo es un tanto limitado, este ilustra alguna información valiosa sobre las subclases. En primer lugar, observa como las subclases Student y Teacher tienen todas las propiedades y métodos de sus superclases. Esto se denomina herencia. En segundo lugar, y gracias a la herencia, las subclases pueden ahorrarte una buena cantidad de tiempo: en vez de tener que añadir métodos idénticos y/o propiedades a varias clases, los abstraes en una superclase y creas subclases a medida que las necesites (recuerda uno de los puntos cardinales de la programación: ¡No te repitas!)

13.3 Manos a la obra con las Subclases

Hablando de no repetirse, las subclases son también un modo muy útil de crear controles personalizados o controles que proporcionan un comportamiento específico.

Por ejemplo, considera un control Label. Tal y como es resulta adecuado para indicar el propósito de un control asociado, como pueda ser un TextField o un PopupMenu; pero no hace nada realmente interesante, por lo menos no por defecto. Supongamos que quieres que tu usuario haga clic en una etiqueta para ver más información sobre algo, o bien para visitar una URL que puedes indicar mediante código. Sería relativamente simple implementar este comportamiento.

A simples rasgos, necesitarías añadir un Label a la Ventana. Necesitarías implementar su evento MouseDown; recuerda que devolviendo True en MouseDown causa el disparo de MouseUp. MouseUp es donde podrías utilizar el método ShowURL para abrir el URL en cuestión, el cual habrás proporcionado en tu código. Si lo deseas, también podrías implementar los eventos MouseEnter y MouseExit (disparados cuando el ratón se mueve sobre el control y cuando lo abandona, respectivamente) para cambiar el color del texto en la etiqueta y también para cambiar el cursor por el cursor estándar del “dedo apuntador” usado por

algunos navegadores web para indicar enlaces (lo que sería un buen indicador visual para hacer saber a tus usuarios de que pueden hacer clic sobre el elemento).

Los pasos descritos deberían funcionar bien, ¿pero qué ocurre si necesitas la misma funcionalidad del Label en otra ventana? Tendrías que repetir de nuevo todos los pasos, cambiando posiblemente sólo el URL a visitar.

Es una gran duplicación de esfuerzos, lo que significa una gran cantidad de tiempo malgastado y la posibilidad de incrementar los fallos. Esta es una situación perfecta para las subclases.

Crea un nuevo proyecto desktop de Xojo y guárdalo como “CustomControl”.

1) **Arrastra la Label desde la Librería hacia el Navegador en la parte izquierda de la pantalla.**

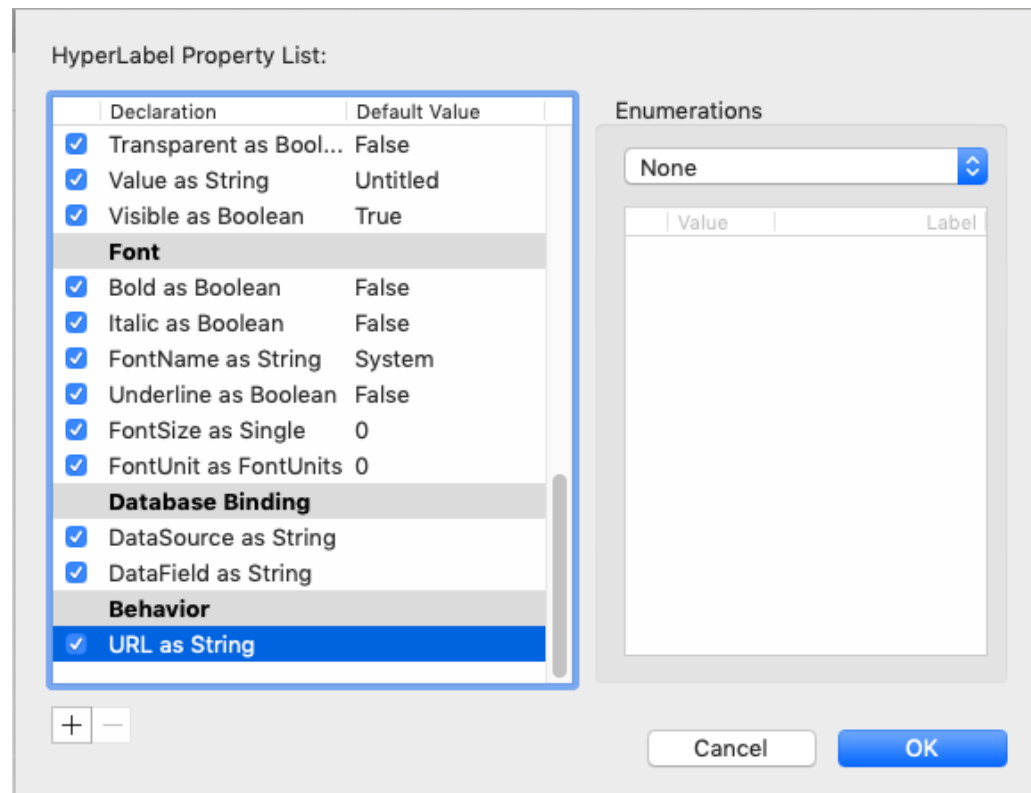
Se creará automáticamente una subclase con el nombre CustomLabel. Renómbralo como “HyperLabel”, dado que la usarás para hiperenlaces.

2) **Añade una nueva propiedad a HyperLabel: URL como String.**

Tu HyperLabel necesitará una URL, y sería ideal si pudiese definir esta en el Inspector en vez de tener que hacerlo mediante código para cada instancia.

3) Haz clic derecho en HyperLabel en el Navegador y selecciona Inspector Behavior en el menú desplegable.

Verás un listado con las propiedades de HyperLabels, incluyendo tanto las propiedades que has creado como las heredadas de su superclase:



Asegúrate de activar la casilla de verificación asociada con URL (aparecerá al final); lo que causará que la propiedad URL aparezca en el Inspector. Si quieres, también puedes desactivar las propiedades que no necesites ver en el Inspector. Haz clic en OK cuando hayas finalizado.

4) Añade este código en el evento Opening de HyperLabel:

```
Self.Underline = True
Self.TextColor = RGB(0, 0, 255)
```

Esto hace que el texto aparezca subrayado y de color azul de modo que HyperLabel parecerá más como el enlace de una página web.

5) Añade este código en el eventoMouseDown:

```
Return True
```

6) Y añade este código en el evento MouseUp:

```
ShowURL (URL)
```

Has de implementar sus eventos MouseDown y MouseUp para hacer que HyperLabel responda a los clic, tal y como se ha indicado anteriormente.

7) Añade este código al evento MouseEnter:

```
Self.TextColor = RGB(0, 0, 150)
MouseCursor = System.Cursors.FingerPointer
```

Para asegurarte de que HyperLabel se comporte más como el hiperenlace de una página web, cambia su color ligeramente cuando el usuario lo apunte, y cambia el cursor del ratón al cursor “del dedo apuntando” que ves generalmente en las páginas web.

8) Añade este código al evento MouseExit:

```
Self.TextColor = RGB(0, 0, 255)
MouseCursor = System.Cursors.StandardPointer
```

Este código “resetea” su aspecto cuando el usuario deja de apuntarlo.

- 9) **Arrastra HyperLabel en Window1. Sitúalo donde quieras.**
- 10) **En el Inspector, define su texto a “Xoyo” y su propiedad URL a “<https://www.xoyo.com>”.**
- 11) **Ejecuta el proyecto.**
- 12) **Sitúa el cursor sobre el HyperLabel y observa como cambia el cursor y el color del texto. Haz clic sobre él.**
- 13) **Sal de la aplicación.**

Como se ha mencionado anteriormente, ahora que has creado un control personalizado mediante una subclase puedes reutilizar este control sin necesidad de añadir código adicional en cualquier ventana de tu proyecto, o incluso en varias ubicaciones de la misma ventana. Además, puedes añadir este control a otros proyectos. Haz clic derecho sobre HyperLabel en el Navegador y selecciona Export HyperLabel para guardarlo sólo dicho control como un archivo que pueda importarse en tus otros proyectos.

Es una práctica habitual entre los desarrolladores la de crear tus propias “librerías” de controles personalizados que puedan reutilizarse en cualquier proyecto cuando lo necesites. Tu librería acaba de arrancar con HyperLabel.

Encerar y Pulir

CONTENIDOS

1. Información del Capítulo
2. Guías de Interfaz de Usuario
3. Crear Interfaces Ágiles
4. Timers
5. Gestionar Ventanas
6. Gestión de Errores



14.1 Información del Capítulo

Hasta ahora has aprendido mucho sobre cómo funcionan las aplicaciones: variables, tipos de datos, lógica, control de flujo, métodos, funciones, controles, clases, módulos, archivos, bases de datos y gráficos. Estas son cuestiones esenciales a conocer cuando se diseña una aplicación.

Pero otro tema importante, uno que muchos desarrolladores ignoran, es la experiencia de usuario. Algunas personas oyen esto e intentan pensar en algunas formas de añadir algún mágico factor “guau” a sus aplicaciones con el objetivo de capturar la atención de los usuarios, y lograr que su aplicación destaque sobre el resto.

En realidad, el factor “guau” no es algo tan importante. Lo realmente importante es proporcionar a tus usuarios una experiencia consistente e intuitiva.

Otro componente principal en una buena experiencia de usuario es su agilidad de respuesta. Tu aplicación nunca debería hacer sentir al usuario que está congelada. Esto está relacionado con mantener informado al usuario: el usuario nunca debería preguntarse qué está ocurriendo (o cuánto tiempo llevará).

Por último, tu aplicación debería tratar los errores de forma grácil (ya sean los generados por las acciones del usuario o el propio código).

En este capítulo aprenderás habilidades que podrás aplicar en estas tres áreas.

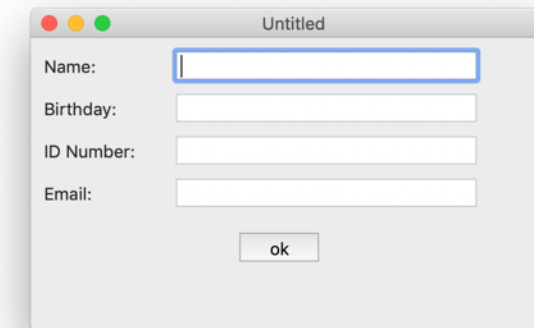
14.2 Guías de Interfaz de Usuario

Tanto si desarrollas para macOS, Windows, Linux, iOS, Android, la web o cualquier otro sistema operativo o plataforma común, hay ciertas guías para el aspecto y comportamiento de tu aplicación. Estas guías están creadas y mantenidas generalmente por las empresas que producen los sistemas operativos en cuestión. Por ejemplo, Google mantiene Las Android User Interface Guidelines, mientras que Microsoft se ocupa de las Windows User Experience Interaction Guidelines.

Estos documentos se actualizan por lo general cada vez que se publica una nueva versión del sistema operativo o plataforma. De modo que cada vez que Apple publica una nueva versión de macOS, este actualiza las Apple macOS Human Interface Guidelines para reflejar los cambios en el diseño de la interfaz de usuario. Con cada nueva revisión del sistema operativo hay cambios importantes (como cuando Apple introdujo la interfaz Aqua o Microsoft introdujo el aspecto Universal en Windows); pero incluso también se reflejan algunos pequeños cambios con el paso del tiempo.

Estas guías pueden ser bastante específicas, incluso hasta el punto de describir a cuantos píxeles debería de estar un control respecto a la parte superior, o cuántos píxeles debería de haber

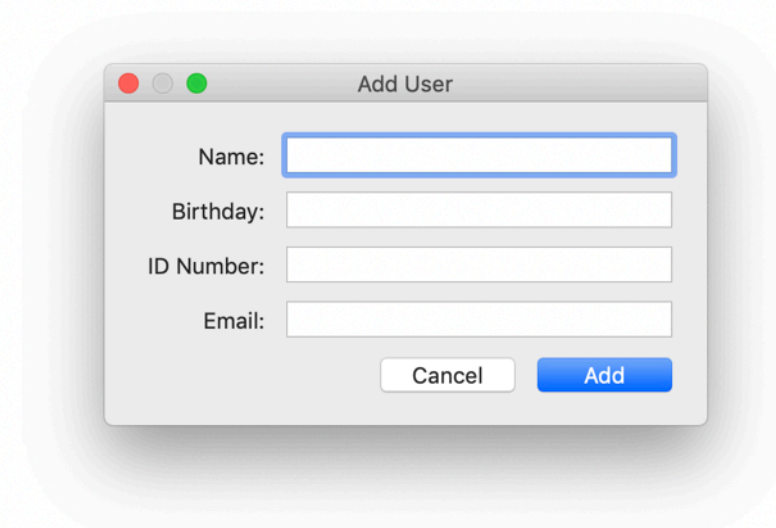
entre PushButton. Deberías seguir estas guías hasta el máximo posible de tu capacidad cuando se aplican a los propósitos de tu aplicación.



He aquí un ejemplo de una interfaz diseñada sin consultar ninguna de las Guías de Interfaz de Usuario:

Observa el espacio inconsistente, la escritura incorrecta de “OK”, e incluso el uso de botones no estándar.

Esta es la misma interfaz, corregida para seguir las Guías de Interfaz de Usuario:



Como puedes ver, mientras que ambas interfaces pueden realizar la misma tarea, una será más agradable de usar (y muy probablemente también más consistente y estable).

Xojo te ayuda a seguir estas guías recomendándote la posición correcta de tus controles (observa las líneas de color azul que aparecen cuando arrastras un control sobre el margen de una ventana, por ejemplo).

Sin embargo, no te preocupes de ir contra estas guías, o “romper las reglas” cuando sirva mejor a tus usuarios. Recuerda que las guías son simplemente... guías, y no normas estrictas y rígidas que deban de obedecerse. En general, no obstante, salvo que tengas un buen motivo contra ellas, sigue lo indicado por las reglas. La ventaja de ello es que tus usuarios estarán familiarizados al instante sobre el funcionamiento de determinados aspectos en tu aplicación, simplemente con exponer interfaces similares a las de otras aplicaciones.

Aquí puedes encontrar las guías de interfaz para varios sistemas operativos y plataformas:

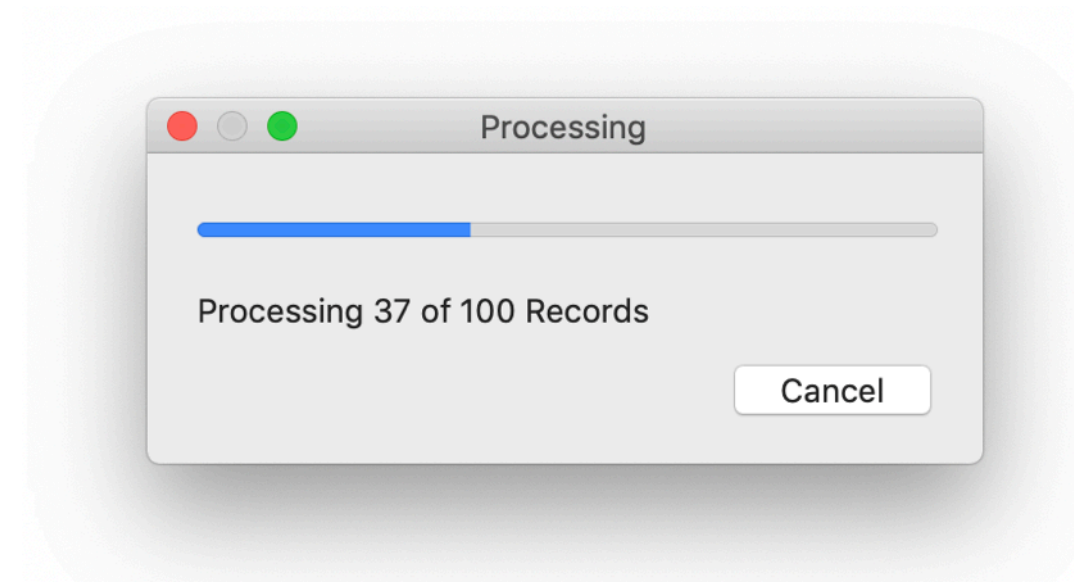
http://en.wikipedia.org/wiki/Human_interface_guidelines

Si bien puede resultar un material difícil de digerir, merece la pena mantener una copia de ellas en alguna parte a la que puedas acceder.

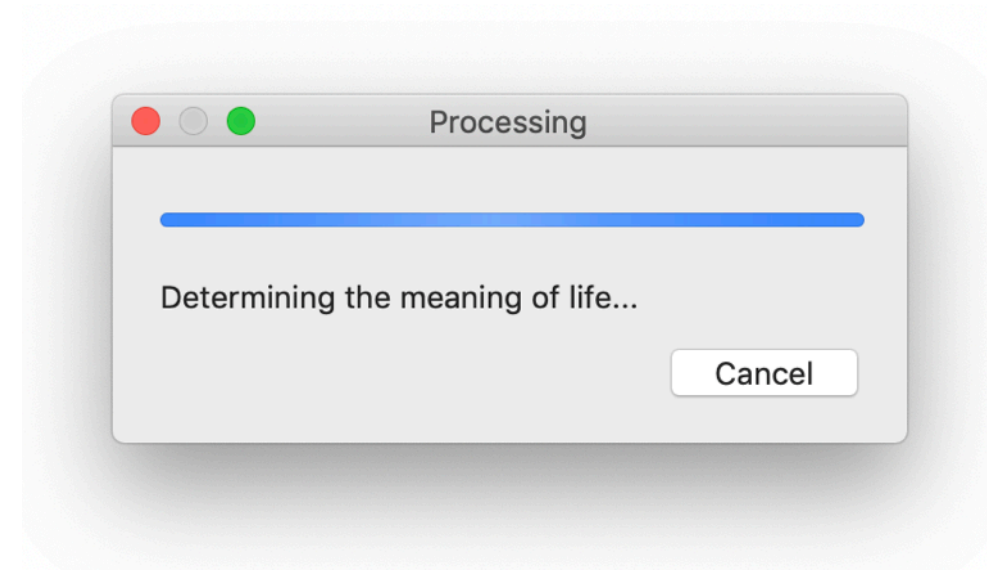
14.3 Crear una Interfaz ágil

¿Has utilizado alguna vez una aplicación en la que te has preguntado qué estaría haciendo? Puede que estuvieses intentando descargar un archivo, imprimir un documento o procesar una imagen. A menudo, una aplicación te proporcionará alguna indicación tanto de su progreso y del tiempo restante. Las aplicaciones que no proporcionan ninguna información dan a menudo la impresión de que están colgadas o congeladas; como usuario, esto puede resultar bastante frustrante, pues debes decidir en qué momento deberías de dejar de esperar y forzar la salida de la aplicación.

Piensa de nuevo en los controles que aprendiste en los capítulos anteriores. Dos controles excelentes para proporcionar información al usuario son el `ProgressWheel` y `ProgressBar`. Si lo recuerdas, el `ProgressBar` es más adecuado cuando puedes cuantificar lo que está haciendo la aplicación. En otras palabras, si estás procesando un número conocido de registros o puedes calcular la duración de un proceso, entonces el `ProgressBar` es genial. Esto se debe a que proporciona al usuario una indicación no sólo de que algo está ocurriendo sino también cuánto ha transcurrido y cuánto queda para completarse.



También se puede usar una `ProgressBar` para definir un estado indeterminado, lo cual envía al usuario un mensaje distinto. Esencialmente dice, “No sé cuánto tiempo llevará, pero por favor sé paciente porque aun estoy en ello.” Este modo está más orientado a situaciones en las que resulta difícil o imposible decidir cuánto tiempo llevará un proceso, como cuando se utiliza un `Socket` para conectar con un servidor remoto o se inicia la conexión con una base de datos.



El siguiente proyecto ilustra cómo usar una ProgressBar para mantener informado al usuario. El proyecto recorrerá 10.000.000 de números y multiplicará cada uno por un número aleatorio. Este es un ejemplo tontorrón, pero es una buena aproximación del tipo de proceso intensivo que han de llevar a cabo algunas aplicaciones. Crearás este proyecto con y sin una ProgressBar, de modo que puedas ver la diferencia.

Crea un nuevo proyecto Xojo desktop y guárdalo como “Progress”.

1) Añade un Botón a Window1.

2) Añade el siguiente código al evento Pressed del Botón:

```
Var i, j As Integer
Var r As Random
For i = 0 To 10000000
    r = New Random
    j = i * r.InRange(1, 1000)
Next
MessageBox("Done Processing!")
```

Puede que esta sea la primera vez que te encuentras con la clase Random, la clase Random, como su propio nombre implica, se utiliza para generar números aleatorios. En este ejemplo estarás usando la función InRange, la cual toma dos números como parámetros y devuelve un número entre ambos (en este caso, entre 1 y 1.000)

3) Ejecuta el proyecto.

4) Haz clic en el PushButton y espera por el mensaje.

Llevará unos cuantos segundos y, entre tanto, tu aplicación podrá parecer como no utilizable o sin respuesta. Cuando se complete el proceso, sal de la aplicación.

5) Añade una ProgressBar a Window1. Sitúala para que sea lo más ancha posible. Añade también una nueva propiedad a Window1: Progress As Integer.

6) Añade un Thread y un Timer a Window1.

Recuerda que el Thread y el Timer se sitúan en la Bandeja porque no son controles visuales. Aprenderás más sobre los Timer en la próxima sección.

7) Cambia el código en el evento Pressed del PushButton para que sea el siguiente:

```
ProgressBar1.MaximumValue = 10000000
ProgressBar1.Value = 0
Timer1.Period = 500
Timer1.RunMode = Timer.RunModes.Multiple
Thread1.Start
```

Este código definirá el MaximumValue y el valor actual del ProgressBar, además de definir el timer para que se ejecute cada medio segundo e indique al Thread que empieza a funcionar.

8) Añade este código en el evento Run del Thread:

```
Var i, j As Integer
Var r As Random
```

```

For i = 0 To 10000000
    r = New Random
    j = i * r.InRange(1, 1000)
    Progress = i
Next

```

Esta es una versión ligeramente modificada del código que se encontraba anteriormente en el evento Pressed del PushButton. Para cada número procesado por la app, se incrementará en uno la propiedad Progress de la ventana. Dado que está dentro de un Thread, la interfaz de usuario permanecerá sin respuesta durante la operación.

9) **Añade este código en el evento Action del Timer:**

```

ProgressBar1.Value = Progress
If ProgressBar1.Value >= ProgressBar1.MaxValue Then
    MessageBox("Done processing!")
    Me.RunMode = Timer.RunModes.Off
End If

```

Este ajustará el valor del ProgressBar al valor de la propiedad Progress definida en el Thread que se está ejecutando. El Thread no puede ajustar directamente el valor del ProgressBar porque (debido a ciertas restricciones del sistema operativo) los Thread no pueden modificar (o acceder) a la interfaz de usuario. El código muestra un mensaje y detiene el Timer cuando se llega al máximo.

10) **Ejecuta el proyecto.**

11) **De nuevo, haz clic en el Botón y espera a ver el mensaje.**

Aún le llevará unos segundos pero, en esta ocasión, el ProgressBar te permitirá saber que algo está pasando y que la aplicación no está congelada o colgada.

12) **Sal de la aplicación.**

14.4 Lograr que las cosas pasen según lo previsto

Habr  ocasiones en las que necesitar s un m todo o funci n que se ejecute a intervalos regulares o en un tiempo determinado, como por ejemplo cada diez segundos. En situaciones como esta, utiliza el control Timer.



Para ver el Timer en acci n vamos a crear una aplicaci n que mantiene el reloj funcionando para que el usuario est  informado de la hora actual.

Crea un nuevo proyecto desktop y gu rdalo como "Timers".

- 1) **A ade una Label a Window1. Define su nombre como "ClockLabel".**

Si ntete libre de situarlo donde quieras, pero aseg rate de que tenga su propiedad Width definida a 100 como m nimo.

- 2) **A ade un nuevo m todo a Window1 llamado "UpdateClock".**

```
Var today As DateTime = DateTime.Now  
ClockLabel.Value = today.ToString(DateTime.  
FormatStyles.Long)
```

Este m todo no tiene par metros. Su tarea es determinar la hora actual (usando un objeto DateTime) y mostrarlo en una ClockLabel.

- 3) **A ade un Bot n a Window1. Ejecuta el m todo UpdateClock en su evento Action:**

```
UpdateClock
```

- 4) **Ejecuta el proyecto.**
- 5) **Haz clic en el PushButton para actualizar la hora.**
- 6) **Sal de la aplicaci n.**

Hasta ahora es una experiencia de usuario bastante mala. Si el usuario necesita ver la hora actual, tiene que hacer clic en el PushButton. Es momento de mejorar este proyecto.

1) **Añade un Timer a Window1.**

Observa que el Timer se sitúa bajo la ventana y el resto de controles (en la Bandeja). Esto se debe a que el Timer no es un control visual; no tiene una interfaz visual que mostrar.

2) **En el Inspector, asegúrate de que el RunMode del Timer está definido a Multiple y que su Period está definido a 1000.**

El RunMode puede ser Off, Single o Multiple. Cuando el RunMode está definido a Off, el Timer está esencialmente domitando y no hará nada. Cuando RunMode está definido a Single, el evento Action del Timer se disparará una vez y eso será todo. Cuando RunMode esté configurado a Multiple, el evento Action del Timer se disparará repetidamente, en función del valor de la propiedad Period. El Period se define en milisegundos o milésimas de segundo, de modo que nuestro valor de 1000 causará que el evento Action del Timer se dispare aproximadamente cada segundo.

3) **Añade este código en el evento Action del Timer:**

```
UpdateClock
```

4) **Ejecuta el proyecto.**

Observa que ahora el reloj se actualiza cada segundo, tanto si se usa el Botón como si no.

5) **Sal de la aplicación.**

14.5 Gestionar Ventanas

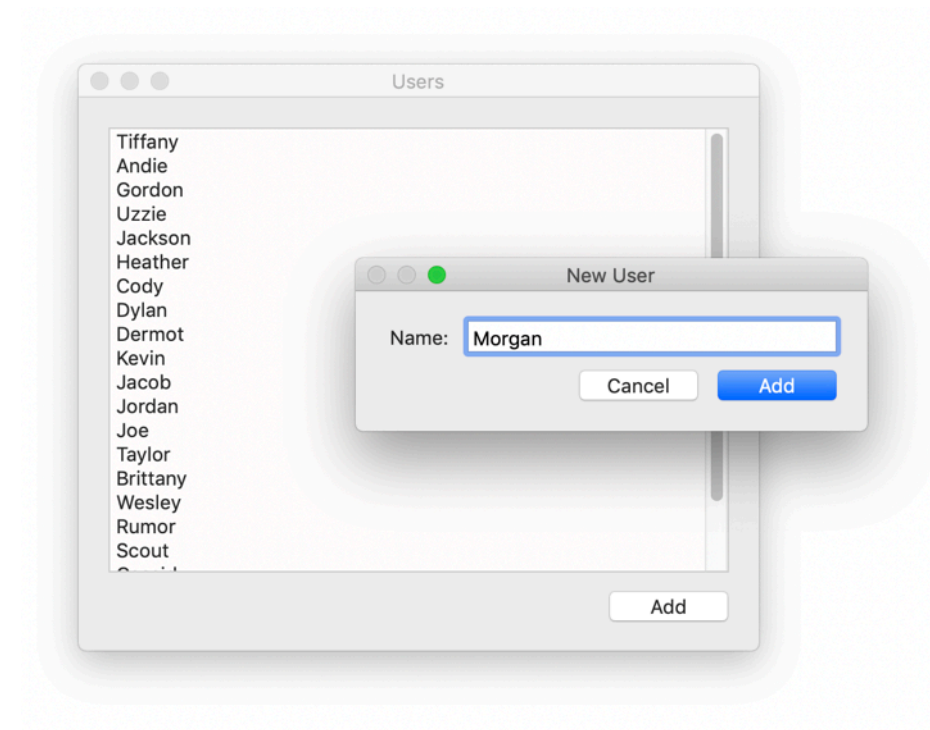
Hasta ahora, todos los proyectos y aplicaciones creados tienen una cosa en común: son aplicaciones con una única ventana. Esto por si mismo no es algo malo, especialmente teniendo en cuenta que las aplicaciones modernas tienden cada vez más a la filosofía de ventana única (esto es especialmente cierto en el caso de los teléfonos inteligentes y las interfaces de las tabletas). Sin embargo, habrá casos en los que necesites soportar varias ventanas en una aplicación. Por suerte, cada ventana que añadas a tu proyecto es en realidad una clase. Esto hace que su gestión resulte muy sencilla, dado que ya sabes cómo trabajar con clases.

Las aplicaciones con múltiples ventanas son bastante comunes. Por ejemplo, cada vez que usas una aplicación con una paleta de herramientas, estás usando múltiples ventanas. Muchos clientes de mail también tienen múltiples ventanas: una para el listado de los mensajes y otra para componer un nuevo mensaje. Además, algunas aplicaciones te permitirá abrir dos ventanas con los mismos datos.

A medida que trabajes con varias aplicaciones, mira a ver si puedes averiguar cuántas ventanas ha definido cada una.

Crea un nuevo proyecto desktop y guárdalo como “MultiWindows”. Este proyecto te permitirá utilizar una ventana y

añadir nombre a una lista, mostrada en otra ventana. Tendrá un aspecto similar a este:



- 1) **Renombra Window1 a “ListWindow”.**
- 2) **Añade otra Ventana a tu proyecto seleccionando Window en el menú Insert. Nómbrala como “DetailWindow”.**
- 3) **Añade un ListBox y un Botón a ListWindow.**
- 4) **Define el Caption del Botón como “Add”.**
- 5) **Añade una nueva propiedad a ListWindow: Names(-1) As String.**

Este es el array que almacenará la lista de nombres.

6) Añade un método llamado “PopulateNames” (sin parámetros) a ListWindow.

```
Names.Sort  
ListBox1.RemoveAllRows  
For Each name As String in Names  
    Listbox1.AddRow(name)  
Next
```

El trabajo de este método es ordenar los métodos en el array y añadir luego cada uno al ListBox usando un bucle For. Dado que se ejecutará múltiples veces, será necesario eliminar previamente todas las filas del ListBox.

7) Añade un método público a ListWindow llamado “AddName”. Este método recibe un parámetro: name As String.

```
Names.AddRow(Name)  
PopulateNames
```

La función de este método es la de añadir un nuevo valor al array Names y ejecutar luego el método PopulateNames.

8) En el evento Open de ListWindow, ejecuta el método PopulateNames:

```
PopulateNames
```

9) Añade este código al evento Action del Botón:

```
Var w As New DetailWindow
```

Con este código, el Botón será el responsable de crear una nueva instancia de DetailWindow y también de mostrarla.

10) En DetailWindow, añade dos Botones (nombrados como “OKButton” y “CancelButton”), un TextField (nombrado “NameField”), y una Label (con un Value de “Name:”).

11) Define la propiedad Default de OKButton a ON y la propiedad Cancel de CancelButton a ON.

Esto causará que el evento Action de OKButton se dispare cuando se pulse la tecla Retorno, y que el evento Action de CancelButton se dispare cuando se pulse la tecla de escape (en ambos casos, los botones también responderán a los clic de ratón, por supuesto).

12) Añade este código al evento Action de CancelButton:

```
Self.Close
```

Cuando se pulsa CancelButton, debería cerrarse la ventana que lo contiene. Este es el código que lo hace posible.

13) Añade este código al evento Action de OKButton:

```
ListWindow.AddName(NameField.Value)  
Self.Close
```

OKButton tiene que hacer más cosas. Necesita tomar el texto en NameField y añadirlo al array Names de ListWindow, cerrando luego la ventana que lo contiene. Dado que necesita referirse a ListWindow podrías asumir que necesitas crear una variable para ello, tal y como has hecho con DetailWindo. Si bien podría funcionar, esto también crearía otra instancia de ListWindow cuando se usase la palabra clave New. En Xojo,

las ventanas proporcionan algo denominado instanciación implícita, siendo una forma elegante de decir que una ventana ya está abierta y la aplicación sólo necesita una copia de dicha ventana; de modo que puedes hacer referencia a ella por su nombre en cualquier momento.

14) Ejecuta el proyecto.

15) Haz clic en el botón Add del ListWindow para acceder a DetailWindow.

Aquí puedes introducir nombres, que se ordenarán y añadirán al ListBox de ListWindow..

16) Sal de la aplicación.

Te animo a que experimentes con diferentes tipos de Ventanas cambiando la propiedad Type bajo Frame en el Inspector.

Observa como este proyecto de ejemplo se comporta de forma distinta cuando se usan diferentes tipos de ventanas.

14.6 Qué hacer cuando las cosas van mal

Otro aspecto importante de la experiencia de usuario es la gestión de errores. El primer paso en la gestión de errores es aceptar que tu aplicación tendrá errores, ya sean provocados por suposiciones incorrectas o por código erróneo. Una vez has aceptado que los errores ocurrirán, tienes que aprender cómo defender tu aplicación y a tus usuarios frente a estos errores.

Muchos de los errores en Xojo son llamados realmente excepciones. Una excepción es simplemente eso: significa que ha ocurrido algo excepcional, algo que no debería de haber ocurrido. Una excepción no gestionada en una aplicación de Xojo casi siempre provocará que la aplicación finalice su ejecución.

Una de las excepciones más comunes en las aplicaciones Xojo es `NilObjectException`. Una `NilObjectException` tiene lugar cuando tu código intenta acceder a un objeto que no existe. Tomemos como ejemplo el siguiente código:

```
Var d As DateTime
MessageBox(d.ToString(DateTime.FormatStyles.Short))
```

Si ejecutas dicho código, verás un `NilObjectException` porque `d`, el objeto `DateTime`, no se ha instanciado. La variable existe, pero

apunta a un objeto que aun no existe. Esto se puede solucionar fácilmente:

```
Var d As DateTime = DateTime.Now
MessageBox(d.ToString(DateTime.FormatStyles.Short))
```

Otras `NilObjectExceptions` pueden resultar más problemáticas de resolver. Imagina que has definido una clase llamada `Student` y que esta clase obtiene información de una base de datos. De modo que en función del número ID de un estudiante, puedes consultar a la base de datos y crear un nuevo objeto `Student` con las propiedades asignadas en consecuencia, como su nombre, apellido y fecha de nacimiento.

Tu código puede ser como el siguiente (asumiendo que `GetStudent` es una función que devuelve un objeto `Student`):

```
Var s As Student
s = GetStudent( 12345 )
MessageBox(s.BirthDate.↵
    ToString(DateTime.FormatStyles.Short))
```

Esto debería de funcionar bien siempre y cuando tengas un número ID válido. ¿Qué podría causar un número ID inválido? Hay varias posibilidades. En primer lugar, un usuario podría introducir un valor no válido a la hora de consultar un registro. En segundo lugar, es posible que desde el momento en el que has empezado a buscar por un registro determinado hasta el

momento en el que se muestre, alguien haya borrado el registro en la base de datos. También pueden darse otras posibilidades.

Con esto en mente, tu código debería verse más como este:

```
Var s As Student
s = GetStudent( 12345 )
If s <> Nil Then
    MessageBox(s.BirthDate.ToString)
End If
```

Esa instrucción If, para comprobar si “s” es Nil, puede protegerte a ti y a tus usuarios de un disgusto. Incluso podría mejorarse para que muestre un mensaje de error si, de hecho, es Nil:

```
Var s As Student
s = GetStudent( 12345 )
If s <> Nil Then
    MessageBox(s.BirthDate.ToString)
Else
    MessageBox("The student could not be found.")
End If
```

También podrías incluir un mensaje para contactar con soporte técnico u otra persona apropiada.

En el anterior ejemplo, sin embargo, sólo se protege si “s” es Nil. ¿Qué pasa si “s” es un Student válido pero no se ha definido la fecha de nacimiento por alguna razón? En este caso, al usar s.BirthDate.ToString también resultará en una NilObjectException.

Este es un ejemplo de algo que debería de haberse gestionado en la clase Student, quizá proporcionando una fecha por omisión o bien tratando aquí con el NilObjectException.

Otra excepción común es OutOfBoundsException. Esta se produce cuando el código ha intentado acceder a un elemento de la lista (array, ListBox, etc.) con un índice superior al de la lista. Por ejemplo, puede ocurrir un OutOfBoundsException si intentas acceder a la décima fila de un ListBox que sólo tiene cinco filas.

Pueden prevenirse la mayoría de las excepciones OutOfBoundsExceptions teniendo cuidado en el código. Por ejemplo, cuando se itera un array considera utilizar un bucle For...Each en vez de usar un bucle con contador.

Por supuesto, hay algunas excepciones en las que no puedes hacer gran cosa. Un ejemplo es el OutOfMemoryException, que tiene lugar cuando el ordenador ya no puede asignar la cantidad de memoria requerida por el proceso actual. En tal caso, lo mejor que puedes hacer es mostrar un mensaje:

```
Try
    Var s As Student
    s = GetStudent( 12345 )
    MessageBox s.BirthDate.ToString
Catch err As NilObjectException
    MessageBox("The student could not be found.")
End Try
```

Todo lo que se encuentra en la parte superior, tras el Try, se intenta. Si ocurre una `NilObjectException`, se ejecuta el código de la parte inferior.

Aprender a programar de forma defensiva es el mejor modo de protegerte a ti y a tus usuarios frente a errores inesperados.

El Libro de Reglas

CONTENIDO

1. Información del Capítulo
2. Las cosas más importantes
3. No te Repitas
4. Los Datos son Sagrados
5. El Principio del Mínimo Asombro
6. Siempre es fallo tuyo
7. Planificar el Futuro
8. Primero, que funcione
9. Documentación y Ayuda



15.1 Información del Capítulo

Si has completado los otros capítulos del libro, entonces tienes un buen punto de partida en los fundamentos de la programación, especialmente si usas Xojo.

Sin embargo, tanto si eliges continuar con Xojo o probar algo distinto como Swift, PHP, Objective-C, JavaScript, Ruby, Java, C/C++, C# o cualquier otro lenguaje, aun tendrás mucho que aprender sobre las reglas y principios de la programación.

Mientras que algunas de estas “reglas” son específicas al desarrollo de software, muchas otras también están directamente relacionadas con la buena gestión de proyectos y habilidades de comunicación.

Este capítulo será distinto del resto. En vez de proporcionarte instrucciones para un proyecto de ejemplo, este capítulo propondrá algunas preguntas para su discusión.

15.2 Las cosas más importantes

Muchos desarrolladores pueden verse atrapados en lo que se llama La Trampa del Programador. Tras pasar meses o años aprendiendo a escribir buen código, y tras varios meses más o años escribiendo aplicaciones, muchos desarrolladores caen en la trampa de pensar que su trabajo es escribir código, pero no lo es.

Incluso si tu título es Programador, Desarrollador de Software o Arquitecto de Software Jefe, tu principal trabajo no es escribir código. Este es un aspecto simple de tu trabajo. Tienes dos trabajos principales. El primero es solucionar problemas y el segundo es añadir valor. Por supuesto, tendrás que escribir código con frecuencia para ayudar a solucionar problemas y añadir valor. En este campo, es lo que se espera.

La parte más difícil de resolver problemas es que raramente se traduce en una solución funcional. Con más frecuencia de la que puedas pensar, la parte más difícil es definir el problema en primer lugar. Con frecuencia, los usuarios o clientes que te piden una app también tendrán dificultades para definir el problema. Pero eso es por lo que han acudido a ti.

El mejor modo de definir un problema es haciendo preguntas. Cuando obtienes las respuestas, continúa haciendo preguntas. Entonces repite las preguntas. Sigue así hasta que comprendas a fondo lo que el cliente o usuario quiere en realidad. Entonces, ponlo por escrito y vuelve sobre ello con los clientes. Antes de que escribas una línea de código, asegúrate de que comprendes el problema que estás intentando solucionar.



Tan importante como las preguntas que hagas son las personas con las que hables. Siempre es buena idea hablar con quien firma el cheque, por así decirlo, pero también es crítico entrar “en las trincheras” y hablar con quienes estarán usando tu aplicación, probablemente a diario.

Pero ten en cuenta que tus usuarios finales no siempre podrán especificar lo que quieren. Después de todo, tal y como señalón Henry Ford, si él hubiese preguntado a las personas qué es lo que querían, estos habrían pedido un caballo más rápido. Antes de que Apple lanzase el Macintosh y Microsoft publicase Windows, las personas querían equipos más rápidos basados en DOS.

A menudo, el problema que ha de solucionarse no es el especificado por el cliente o el usuario. En tales casos, tendrás que profundizar. Un ejemplo clásico es el relacionado con un hombre que dijo que necesitaba un martillo. Profundizando en su historia, resultón que si bien necesitaba un martillo, este no era el problema real. En realidad necesitaba clavar un clavo en la pared. Pero incluso esa no era su necesidad real. En realidad necesitaba un clavo en la pared para colgar un cuadro. Y quería colgar el cuadro porque quería un entorno agradable. Cuando se le preguntó, el hombre dijo que necesitaba un martillo, pero lo que realmente quería era arte y belleza en su entorno. Del mismo modo, necesitas hacer preguntas y profundizar.

Esto nos lleva al segundo trabajo principal del desarrollador: añadir valor. Tu trabajo y tu código debería de añadir valor a la experiencia de usuario. Esto puede tomar varias formas, como pueda ser proporcionar arte, tal y como se indicó anteriormente. Puede que se trate de tomar un documento, procesarlo y publicarlo en línea para que sea más rápido y eficiente.

Independientemente de la tarea, tu trabajo es lograr que la vida de tus usuarios sea mejor, incluso con pequeñas cosas.

PARA SU DISCUSIÓN:

- 1) Cuando se intenta determinar el problema a resolver, ¿qué tipo de preguntas deberías de hacer al usuario o cliente?**
- 2) ¿Cómo gestionas a alguien que se aproxima a ti con una solución detallada y ya trazada sin especificar cuál es el problema real?**
- 3) ¿De qué otras formas puedes añadir valor?**

15.3 No te Repitas

Como se mencionaba anteriormente en este libro, una de las reglas cardinales de la programación es “No te Repitas”, algunas veces indicado como el principio DRY (Don’t Repeat Yourself, en inglés). El principio DRY significa que si tienes el mismo código en más de un lugar en tu aplicación, entonces lo estás haciendo mal.

Cuando tienes código duplicado, estás invitando a los problemas y a los fallos en tu aplicación. De forma inevitable, las cosas cambian a lo largo del camino... y necesitarás hacer cambios en tu código. ¿Te acordarás de cambiarlo en todos los sitios relevantes? No importa cuan inteligente o diligente seas, la historia sugiere que, eventualmente, olvidarás hacer un cambio importante y tu código quedará desincronizado.

La solución de no repetirte es la de abstraer todo el código que sea posible en métodos y funciones. Como regla general a seguir, si una porción de código tiene más de dos o tres líneas y estás usando dicha porción de código en más de dos sitios (esta se conoce como la Regla de Tres), entonces te harías un favor moviendo dicho código a un método o función a la que puedas llamar cuando sea necesario.

Recuerda también que si tu código no es DRY, entonces es WET (WET significa, por sus siglas en inglés, Write Everything Twice; es decir, escríbelo todo dos veces).

PARA SU DISCUSIÓN:

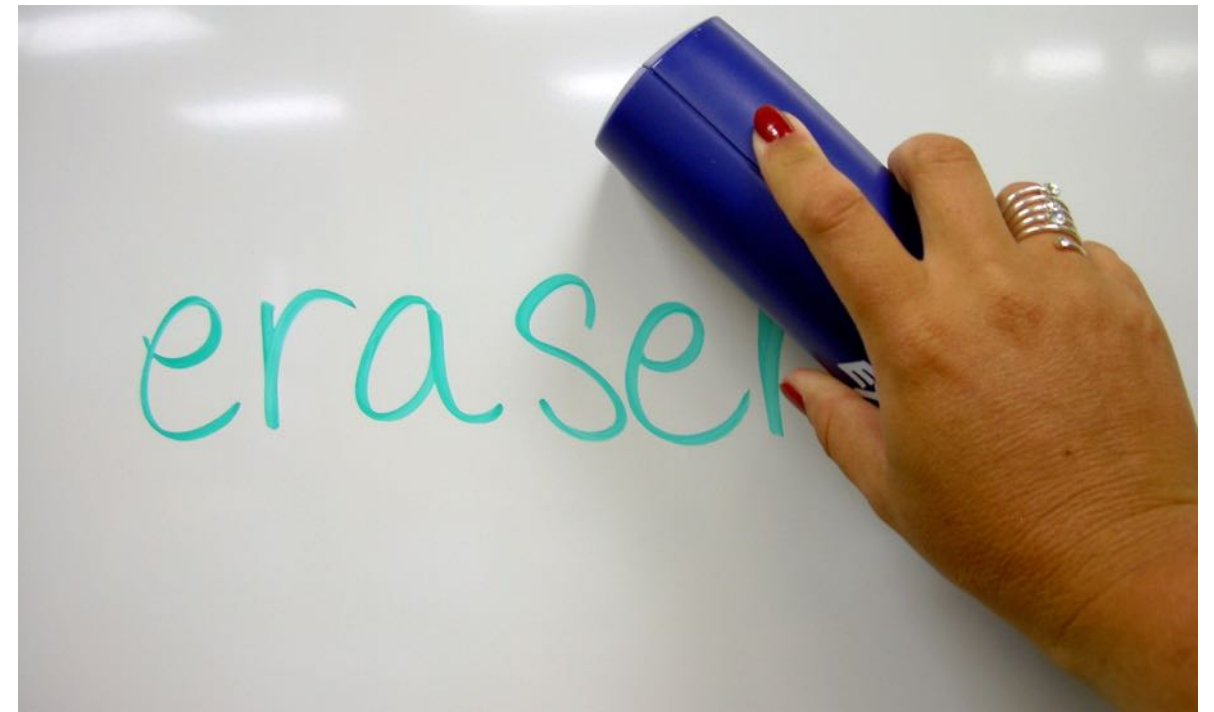
- 1) ¿Por qué es el principio DRY más importante que otros principios de programación?
- 2) ¿Cuáles son las desventajas del enfoque WET?

15.4 Los Datos del Usuario son Sagrados

Probablemente uses un servicio de email como Gmail o Yahoo Mail. Imagina conectarte a tu cuenta un día para descubrir que todos tus mensajes archivados ya no están. ¿Cómo reaccionarías?

O imagina que te has conectado a Facebook sólo para encontrarte con que se han borrado todas tus fotos.

Probablemente te sentirías enfadado (como poco) en ambos casos. Tu usuarios también lo estarían.



Otra regla de la programación es esta: Los Datos de los Usuarios son Sagrados. No borres nunca datos del usuario sin estas tres cosas: una muy buena razón, el permiso de tu usuario, y la posibilidad de que tu usuario pueda cancelarlo.

Es importante observar que esto sólo se aplica a los datos del usuario. Si tu código genera algún archivo temporal que ya no almacene información importante, entonces es seguro borrarlo. De hecho, tal y como ocurre con el resto de la vida, siempre es bueno hacer limpieza. Pero los datos creados o guardados han de mantenerse seguros.

Como se ha indicado anteriormente, incluso antes de borrar datos del usuario, necesitas una muy buena razón para hacerlo. El motivo más obvio es que el usuario haya decidido borrar algo.

Otra razón podría ser que los datos hubiesen expirado. También pueden existir otras. Un ejemplo de una mala razón para borrar datos del usuario es un error de programación. Si tu aplicación borra datos sin un buen motivo, ¡entonces puedes estar seguro de que muy pocas personas la usarán!

Siempre necesitarás permiso del usuario. Si el usuario ha iniciado el proceso de borrado, entonces es posible que puedas borrarlo. Si tu aplicación ha iniciado el proceso, entonces has de asegurarte de que, ya sea mediante un diálogo u otro mecanismo, tu usuario te ha garantizado el permiso para continuar.

Por último, necesitas proporcionar un modo de que el usuario pueda cancelar el borrado. Generalmente esto tiene la forma de un diálogo del tipo “¿Estás seguro?”. Por supuesto, esto puede resultar molesto como usuario, pero este inconveniente ofrece un buen equilibrio desde el punto de vista de la seguridad para los datos.

Como punto adicional, podrías proporcionar un método para que el usuario pueda deshacer el borrado de datos. Esto no es un requisito, aunque sí es muy común. Si no se puede deshacer el borrado, siempre es buena idea advertir de este hecho cuando se confirme el borrado.

PARA SU DISCUSIÓN:

1) ¿Cuáles son otros motivos válidos para el borrado de datos?

2) ¿Cómo crees que podrías implementar una función de Deshacer?

15.5 El Principio del Menor Asombro

Cuando haces clic en un PushButton etiquetado “Imprimir”, ¿qué esperas que ocurra? Si eres como la mayoría, probablemente esperes que algo salga por la impresora (por ejemplo, algo relacionado con lo que muestra la pantalla). ¿Qué ocurriría si hicieses clic en un botón Imprimir y la aplicación saliese? ¿O qué ocurriría si hicieses clic en una caja de verificación y la aplicación enviase un email?

Los usuarios esperan que ciertos elementos de la interfaz hagan determinadas cosas (consulta el anterior capítulo para ver más ejemplos). Este es parte del Principio del Menor Asombro: los elementos de interfaz de usuario deberían hacer lo que normalmente hacen. Esto significa que un botón de radio no debería de cerrar una ventana, y que un PushButton no debería desplegar un menú emergente.

Ahora bien, esto no significa que tu aplicación no deba de sorprender a tus usuarios. De hecho, algunas veces las sorpresas son excelentes. Pero una sorpresa debería dejar siempre al usuario encantado. Siempre, siempre, persigue encantar a tus usuarios; hay pocas formas mejores de que tus usuarios continúen utilizando tu software. Pero nunca intentes sorprenderlos de forma negativa.

Esto está relacionado con la regla conocida como KISS, que significa (por sus siglas en inglés), y si me perdonas la expresión, Keep It simple, Stupid! (¡Mantenlo Sencillo, Estúpido!).



Muchos desarrolladores caen en la trampa de hacer que muchos aspectos de sus aplicaciones sean configurables, lo que a menudo deriva en una interfaz de usuario poco manejable.

Un ejemplo excelente de una interfaz de usuario sencilla y mínima que arrasa a su competencia es, por ejemplo, la página de inicio de Google. Su sencilla caja de búsqueda redefinió la búsqueda en Internet cuando las páginas web de sus competidores se estaban tornando cada vez más complejas y resultaban más difíciles en su navegación. La interfaz de usuario sencilla y directa de Google, proporcionó a los usuarios más confianza en el

producto, y ahora Google es prácticamente el propietario de las búsquedas en Internet.

La regla general es diseñar tu aplicación para satisfacer al 80% de la gente. Mientras que este puede parecer un objetivo demasiado bajo, recuerda que no siempre podrás complacer a todos.

Recuerda, sin embargo, que sencillo para el usuario no se traduce necesariamente en sencillo de desarrollar. De hecho, y con bastante frecuencia, cuanto más simple sea la interfaz más complejo será el código que hay detrás.

PARA SU DISCUSIÓN:

- 1) ¿De qué forma son las interfaces sencillas superiores a las interfaces complejas?**
- 2) ¿Cuándo es apropiada una interfaz compleja?**
- 3) ¿Cuáles son algunas formas de que tu aplicación pueda encantar a tus usuarios en vez de sorprenderlos?**

15.6 Siempre es Fallo Tuyo

La aplicación se cuelga.

Los datos se corrompen.

Los archivos se pierden.

¿Qué tiene todo esto en común? Si se trata de tu aplicación, entonces son fallos tuyos.



Puede que suene duro, pero la realidad es que necesitas predecir todo lo que pueda ir mal, y defender a tus usuarios frente a ello.

Por lo general esto involucra defender a tus usuarios de ellos mismos, porque muchas veces el usuario es su peor enemigo.

Diseña tu aplicación para que minimice el daño que puede hacer un usuario. Parte de ello nos lleva de nuevo a la sección “Los Datos son Sagrados” de unas pocas páginas atrás. Pero has de ser incluso más proactivo que eso.

Por ejemplo, si tu aplicación tiene un `TextField` en el que se supone que el usuario sólo puede introducir números, no le permitas que pueda introducir ninguna letra (para ver cómo, consulta la función `Asc` y el evento `KeyDown`).

Si se supone que el usuario ha de introducir una fecha, no esperes simplemente a que el usuario la introduzca en el formato correcto. Algunos usarán barras mientras que otros utilizarán guiones. Algunos utilizarán el formato norteamericano y otros usarán el formato español. Algunos usarán el estándar empresarial y otros utilizarán el formato académico. Algunos simplemente podrán el mes. Es posible usar el control de selección de fechas (`DateTimePicker`) o proporcionar algunos menús desplegables para guiar al usuario en la introducción de la fecha.

El punto clave es este: si permites que tu usuario introduzca datos erróneos, lo que ocurra a partir de dicho punto es fallo tuyo. Permitir que el usuario pueda hacer algo envía el mensaje de que está bien hacerlo, de modo que introducir datos

arbitrarios en un campo numérico sin ningún tipo de advertencia, les indica que te encargarás de parsear correctamente el dato. Si necesitas los datos en un modo determinado, haz que resulte fácil para el usuario introducirlo de esa forma y que resulte extremadamente difícil (si no imposible) que pueda introducirlos de la forma incorrecta.

Te sorprendería la de cosas que los usuarios intentan hacer con tus aplicaciones. Algunas de estas serán grandes ideas que puedes implementar, pero muchas otras serán cosas que, en el mejo de los casos, conseguirán que te rasques la cabeza y te sorprendas.

En el último capítulo viste como atrapar errores y excepciones. Aquí es donde los errores y excepciones entran en juego en el mundo real. Puede que suene cínico, pero realmente necesitas estar preparado para que tus usuarios sean destructivos haciendo cosas aleatorias. Y si tu código lo permite, entonces es culpa tuya.

Relacionado con esto, si tu aplicación no soluciona el problema que se supone que soluciona, entonces esto significa que no hiciste las preguntas adecuadas o suficientes en una etapa inicial.

PARA SU DISCUSIÓN:

- 1) Más allá de las entradas numéricas y de fecha, ¿qué otros formatos podrían requerir de una especial consideración?**

- 2) ¿De qué modo puedes asegurarte de que tu aplicación esté solucionando los problemas correctos de la forma correcta?**

15.7 Planifica para el Futuro

La mayoría del software que utilizas no está en su primera versión. Apple ha estado trabajando trabajando en sus sistemas operativos incluso antes de 1984, y han estado trabajando en iOS con anterioridad a 2007. Microsoft tiene versiones de DOS y Windows también desde hace décadas. Word, Photoshop, Firefox, Google, Chrome y Xojo se han ido ampliando, mejorando y actualizando a lo largo de los años.

Tu código también tendrá que actualizarse. Puede que tu aplicación no se use durante décadas como alguno de estos ejemplos, pero las probabilidades de que estén completas y perfectas en la versión 1.0 son próximas a cero.



Por ello, necesitas planificar el futuro. El mejor modo de hacerlo es escribiendo código que pueda leerse. Esto significa que deberías ser lógico y consistente a la hora de nombrar los métodos, funciones y variables. También deberías de regirte por el principio DRY. Y deberías de comentar tu código de forma extensa. Algún día, a lo largo del tiempo, alguien tendrá que actualizar tu código y no podrá adivinar cómo funciona. Y hay muchas probabilidades de que dicha persona seas tu. Recuerda que lo que ahora está fresco en tu memoria, probablemente no lo esté dentro de un año.

Así que hazle un favor a tu yo del futuro y escribe código legible ahora. Y si no eres tu quien ha de mantenerlo, entonces harás muy feliz a algún otro desarrollador.

Otra forma de que tu código resista el paso del tiempo consiste en no hacer demasiadas suposiciones. Escribe el código para que sea flexible. Por ejemplo, puede que estés escribiendo una aplicación que por ahora sólo tenga que gestionar tres archivos, pero puede que algún día sean más, y tu aplicación debería de estar preparada para ello con cambios mínimos... o incluso sin tener que hacer cambios.

PARA SU DISCUSIÓN:

- 1) ¿Cuáles son otras formas prácticas de que tu código resista el paso del tiempo?**
- 2) Imagina que necesitas crear una pequeña aplicación que se comunique con una red social basada en web. ¿Cuáles son algunas características futuras que podrían añadirse con el paso del tiempo, y cómo podrías preparar tu código ahora para ello?**

15.8 Primero, haz que funcione

Otra trampa en la que caen muchos desarrolladores es la trampa de la optimización. Cuando esto ocurre, un desarrollador estará retrasando el despliegue de su aplicación mientras que sigue trabajando en pequeñas mejoras de velocidad (a veces no tangibles). Pero como dijo Steve Jobs, “Los verdaderos artistas, entregan”. En otras palabras, una aplicación que no usa nadie salvo tu mismo, no cuenta demasiado.



Cuando se trata de optimizar, toma una lección de Donald Knuth. Probablemente no hayas oído nunca de Donald Knuth, pero le

debes estar agradecido simplemente porque estás usando un ordenador. Nacido en 1938, Knuth es un pionero en Ciencia de Computación e Ingeniería de Software. Una de sus citas más conocidas sobre el desarrollo de software está directamente relacionada con la optimización: “Debemos de olvidarnos sobre pequeñas eficiencias; digamos que el 97% del tiempo: la optimización prematura es la raíz de todo el mal.”

Quizá Knuth estuviese exagerando un poco las cosas para mostrar su apreciación; pero es algo a considerar: no te dejes atrapar por lograr que tu aplicación sea más rápida, hasta el punto de que nunca llegues a desplegarla.

Knuth no estaba sólo acerca de esta opinión. Su contemporáneo Michael Anthony Jackson (no confundir con otro Michael Jackson, el último Rey del Pop), dijo lo siguiente: “La primera regla de la optimización de programas: no lo hagas. La segunda regla de la optimización de programas (¡sólo para expertos!): ¡no lo hagas todavía!”

El riesgo de la “optimización prematura” es que puedes verte tan atrapado en la velocidad de tu aplicación que probablemente termine comprometiendo tu diseño, además de que probablemente tomes algunas decisiones a corto plazo durante el proceso.

Esto está relacionado con otra trampa denominada la trampa de las características, y que ocurre cuando el desarrollador intenta

repetidamente añadir “una característica más” a una aplicación antes de publicarla. Esto ocurre con mucha frecuencia, y resulta bastante fácil caer en ella. Pero puede evitarse redactando dos importantes documentos: las especificaciones y la hoja de ruta.

Las especificaciones detallan exactamente las características requeridas para lanzar la aplicación. La hoja de ruta, por otra parte, establece cuando, por ejemplo ahora, se añadirán nuevas características. Por ejemplo, la especificación para un nuevo cliente de email debería de incluir elementos como enviar un mensaje, recibir un mensaje, gestión de contactos e incluso corrector ortográfico. Estas son necesidades. La hoja de ruta es donde puedes encontrar características como integración con Facebook, un stream de Twitter en tiempo real, y mapas interactivos de Google. Características chulas, seguro, pero no necesarias para la primera versión de tu aplicación.

PARA SU DISCUSIÓN:

- 1) ¿Qué otros ítems podrías encontrar en las especificaciones para un cliente de email básico? ¿Y en la hoja de ruta?**
- 2) ¿Qué piensas sobre la afirmación de que “Los artistas reales, entregan”?**

15.9 Documentación y Ayuda

Una de las piezas más vitales de cualquier aplicación no tiene nada que ver con el código: la documentación. Es una garantía virtual de que cuando despliegues la compilación final de una aplicación los usuarios querrán documentación. Tanto si se trata de forma impresa, como de una serie de hojas sueltas, screencasts o un sistema de ayuda en línea, la documentación dará a tus usuarios un sentido de seguridad y confianza en tu ausencia.

Algunas organizaciones desarrollan primero su documentación. En este caso, la documentación actúa esencialmente como las especificaciones o guías utilizada tanto por los desarrolladores como por los diseñadores a medida que avanzan en el desarrollo de la aplicación.

Puede que suene como una contradicción, pero al mismo tiempo también deberías de procurar desarrollar aplicaciones que sean tan sencillas de usar que pocos usuarios necesiten documentación.

PARA SU DISCUSIÓN:

- 1) ¿Cuáles son algunas de las formas sencillas y prácticas en las que podrías incluir documentación con una aplicación Xojo?

- 2) ¿Cuándo fue la última vez que utilizaste la documentación de una app y por qué?

Epílogo



CONTENIDOS

1. **Gracias**
2. **Acerca de los Autores**

Gracias

Gracias por tomarte el tiempo para leer *Introducción a la Programación con Xojo*. Tanto si has recorrido este libro sólo o como parte de una clase, espero que te haya proporcionado alguno de los fundamentos sobre la programación. Puede que no te propongas ejercer como alguien dedicado exclusivamente al desarrollo, pero siempre viene bien tener habilidades como programador a medida que va aumentando cada vez más un mundo altamente tecnológico.

Si tienes preguntas, comentarios o sugerencias sobre cualquier aspecto de este libro, por favor siéntete libre de enviar un email a docs@xojo.com.

Sobre los Autores

BRAD RHINE

Brad es un autodenominado geek de los ordenadores y ha estado trabajando como Desarrollador de Ordenadores, Desarrollador Web, Escritor Técnico, Administrador de Bases de Datos, Asistente del Director de Tecnología y, brevemente, como vendedor de árboles de Navidad.

También es un antiguo columnista de XDev Magazine y ha realizado presentaciones en las Xojo Developer Conference sobre diferentes temas.

Brad ha pasado gran parte de su carrera profesional trabajando en el sistema escolar público.

Cuando no está escribiendo código o escribiendo sobre código, puedes encontrar a Brad tocando su guitarra, pasando el tiempo con su familia, o corriendo.

Vive en la Pennsylvania rural con su mujer, sus dos hijos, su perro y u dos gatos inadaptados.

PAUL LEFEBVRE

Paul es un Ingeniero Xojo que también ha contribuido en gran medida a la documentación y ejemplos, entre otras cosas. Ha estado trabajando con ordenadores desde la primera vez que usó un Atari 400 en 1983.

Copyright y Licencia

Este trabajo está protegido por copyright © 2012-2020 by Xojo, Inc.

Este trabajo está licenciado bajo una licencia [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License](#).

Esto significa que eres libre de compartir el material de este libro siempre que no lo hagas para obtener cualquier tipo de beneficio y se mantenga la atribución a Xojo, Inc.

Esta es la 4ª Edición de *Introducción a la Programación con Xojo*.