LSE003-2015: Diseño de Sistemas Embebidos

Dr. Casimiro Gómez González
Facultad de Electrónica, UPAEP
correo: casimiro.gomez@upaep.mx
Tel: 222 229 9428

Primavera 2015

Prólogo

El presente material ha sido elaborado en el laboratorio de sistemas embebidos UPAEP, y se ha desarrollado con la experiencia de estudiantes y profesores que han colaborado en dicho laboratorio. Hay material propio de clases y otro material generado a través de proyectos de vinculación y consultaría. Cualquier comentario o corrección favor de enviarlo por correo al autor.

El autor
Casimiro Gómez González
Doctor en Ingeniería Mecatrónica
correo: casimiro.gomez@upaep.mx

Índice general

Capítulo 1

Conceptos Básicos de Sistemas Embebidos

Los últimos diez años mas o menos, el mundo de la computación se ha movido desde las máquinas de escritorio grandes y estáticas a los dispositivos embebidos pequeños y móviles. Sistemas de software corriendo en redes de móviles, y los dispositivos embebidos deben tener propiedades que no siempre requieren los sistemas tradicionales:

- Rendimiento cerca del óptimo
- Robustez
- Distribución
- Dinamismo
- Movilidad

Una de las diferencias en la ingeniería de software para sistemas embebidos es el conocimiento adicional que el ingeniero tiene de potencia eléctrica y electrónica; interfaces físicas de electrónica analógica y digital con la computadora; y diseño de software para sistemas embebidos y procesamiento digital de señales.

Cerca del 95 % de los sistemas de software son actualmente embebidos. Considera los dispositivos que tienes en casa de uso diario:

- Teléfono Celular, iPOD, microondas

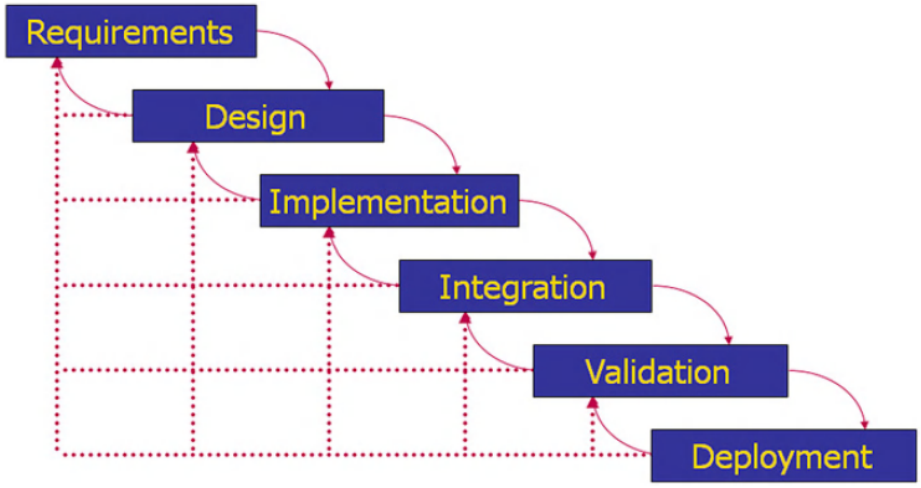

Figura 1.1: Modelo de desarrollo de software *Waterfall*

- receptor satelital de TV, receptor de TV de cable
- Unidad de control de automovil
- Reproductor de DVD

El desarrollo de software embebido usa los mismos modelos de desarrollo de software que las otras técnicas convencionales, incluyendo el modelo *Waterfall*, el modelo en *espiral* y el modelo *Agile*.

La fases principales para el desarrollo de sistemas embebidos se pueden describir:

1. **Definición del problema**: En esta fase se determina exactamente que quiere el cliente y el usuario. Esto incluye el desarrollo de un contrato con el cliente, dependiendo que tipo de producto esta siendo desarrollado. El objetivo de esta fase es especificar que producto de software se hará. Las dificultades incluyen: la solicitud del cliente por el producto incorrecto, el cliente no sabe sobre computación/software, lo cual limita la efectividad de esta fase, las especificaciones por lo regular son ambiguas, inconsistentes e incompletas.

2. **Arquitectura/diseño**: La arquitectura se refiere a la selección de los elementos de arquitectónicos, sus interacciones, y las restricciones de

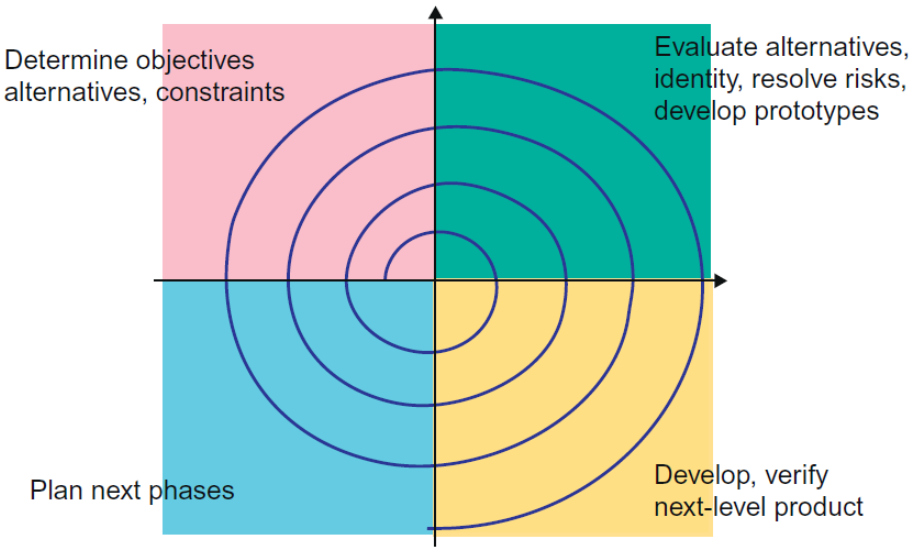

Figura 1.2: Modelo de desarrollo de software *Espiral*

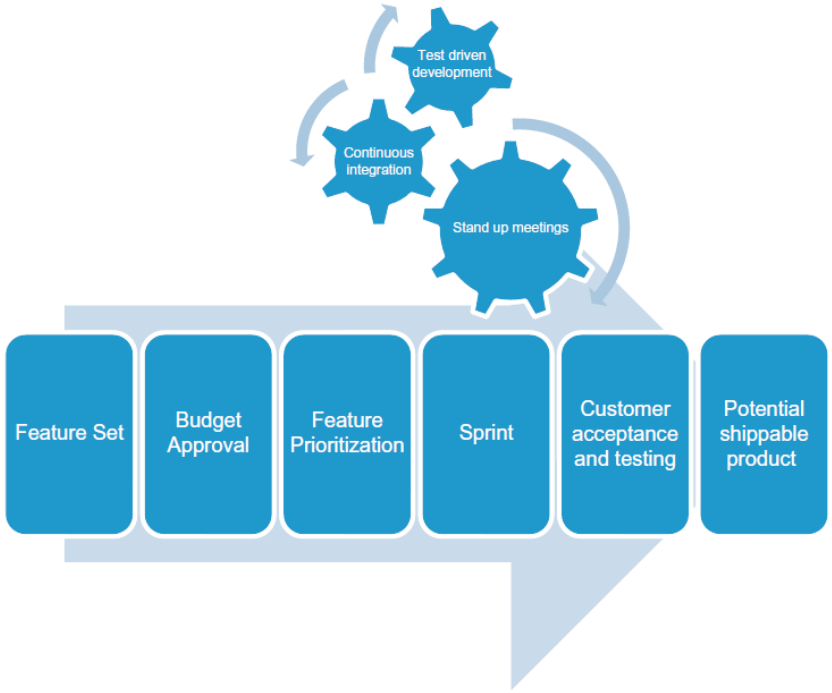

Figura 1.3: Modelo de desarrollo de software *Agile*

aquellos elementos y sus interacciones necesarios para proporcionar un ambiente de trabajo con el cual satisfacer los requerimientos y servir como base para el diseño. El diseño se refiere a la modularización y detalle de interfaz entre los elementos de diseño, sus algoritmos y procedimientos, y los tipos de datos necesarios para soportar la arquitectura y satisfacer los requerimientos. Durante la fase de arquitectura y diseño, el sistema es descompuesto en módulos de software con interfaces. Durante el diseño, el equipo de software desarrolla las especificaciones de los módulos (algoritmos y los tipos de datos), manteniendo un récord de las decisiones de diseño y la trazabilidad, y especifica como los productos de software harán estas tareas. La dificultad principal durante esta fase incluye la falta de comunicación entre los diseñadores de módulos y terminan desarrollando un diseño que puede ser inconsistente, incompleto y ambiguo.

3. **Implementación**: Durante esta fase el equipo de desarrollo implementa los módulos y componentes y verifica que cumplan las especificaciones. Los módulos se combinan de acuerdo con el diseño. Las implementaciones especifican como los productos del software hacen su trabajo. Algunas de las dificultades principales incluyen los errores de interacción entre los módulos y los errores de integración que pueden influenciar la calidad y la productividad.

4. **Verificación y Validación(V&V)**: Hay varias formas de V&V. Una forma es el "análisis". El análisis puede ser en la forma de verificación estática, científica y fromal y en revisiones informales. Las pruebas es una forma dinámica de V&V. Esta froma de pruebas viene en la froma de una caja blanca (se tiene acceso al código) y de caja negra (no hay acceso al código). Las pruebas pueden ser estructurales o de comportamiento.Hay pruebas estandares para realizar dependiendo del producto a desarrollar.

Mas y mas desarrollo de software para sistemas embebidos se esta moviendo al desarrollo basado en componentes.Este tipo de desarrollo es aplicado generalmente para componentes de tamaño razonable y reutilización a través del sistema, que es una tendencia creciente en sistemas embebidos. Los desarrolladores aseguran que estos componentes son adaptables a contextos variantes y extienden la idea mas alla del código a otros artefactos desarrollados tambien. Esta ténica cambia la ecuación desde "Integración, entonces

Bajar al Hardware" a "Bajar al Hardware, entonces Implementar".

Capítulo 2

Configurando las herramientas

Aún cuando muchas de las técnicas de programación para las tarjetas embebidas se realizan en Python, el hecho de que es un lenguaje interpretado provoca que muchas de las aplicaciones desarrolladas en este lenguaje sean mucho mas lentas que las desarrolladas en lenguajes compilados como C. Este problema es principalmente crítico para proyectos con aplicaciones gráficas o procesamiento gráfico. Una solución es utilizar un lenguaje que tenga la rapidez de C y la facilidad de programación de Python, la propuesta en el lenguaje Vala, con un enfoque moderno de programación orientada a objetos.

2.1. Instalando GTK+

Lo primero que debemos instalar es el software GTK+, para ello se ejecuta desde la terminal:

```
1 sudo apt-get install libgtk-3-dev
```

Si te marca un error al instalar esta librería entonces realizar los siguientes pasos:

```
1 sudo apt-get install aptitude
2 sudo aptitude install libgtk-3-div
```

2.2. Compartiendo Internet

En el directorio /boot editar el archivo cmdline.txt y al final de la línea poner: ip=ip de la rasp:: ip de la computadora : máscara de subred

:rpi:eth0:off

En mi caso ha sido ip=192.168.137.2::192.168.137.1:255.255.255.0:rpi:eth0:off

2.3. Instalando Anjuta como IDE de programación en Vala

Debido a que los proyectos que se realizarán serán en dos plataformas de programación, Vala y Python, el IDE que trabaja con ambos de manera muy eficiente se llama Anjuta. Y es este IDE el que se instalará.

2.3.1. Instalación de Vala

Para instalar vala en Ubuntu se debe ejecutar el siguiente comando:

```
1 sudo apt-get install valac
```

2.3.2. Pasos para instalar Anjuta en Ubuntu

Para instalar Anjuta para el lenguaje Vala y Python 3 debe seguirse los siguientes pasos:

Primer paso

Desde una terminal ejecutar:

```
1 sudo apt-get install anjuta anjuta-extras
2 sudo apt-get install python3 python3-pkg-resources
```

Segundo Paso

Bajar la última versión de python3. Y desde https://pypi.python.org/pypi/rope_py3k bajar la ultima versión de Rope (en esta caso rope_py3k-0.9.4-1.tar.gz). Una vez bajado el archivo tenemos que abrir una terminal en la carpeta donde se encuentra y allí ejecutar lo siguiente:

```
1 tar zxvf rope_py3k-0.9.4-1.tar.gz
2 cd rope_py3k-0.9.4-1/
3 sudo python3 setup.py install --prefix=/usr --install-lib=/
    usr/lib/python3/dist-packages
```

```
4 sudo ln /usr/lib/python3/dist-packages/rope_py3k-0.9.4_1-py3
    .4.egg-info /usr/lib/python3/dist-packages/rope-0.9.4_1-
    py3.4.egg-info
```

Tercer Paso

Ahora abrir una terminal y ejecutar python3. En la linea de comandos de Python3 checar si las siguientes instrucciones no marcan ningún error. De marcar algún error revisar los pasos anteriores.

```
1 import rope
2 import pkg_resources
3 pkg_resources.get_distribution('rope').version
```

Cuarto Paso

Ahora hay que corregir el archivo script de Anjuta para poder ejecutar Python3

```
1 sudo nano /usr/lib/anjuta/anjuta-python-autocomplete.py
```

agregando al principio de este archivo la siguiente linea

```
1 from __future__ import print_function
```

Entonces modifica todos los comandos **print blahblah** a la sintaxis de python3: **print(blahblah)**, los comandos **print** estan en las lineas 124, 143 y 144. Ahora inicia Anjuta y seleccionando **Edit->Preferences->Python->Environment**: cambiar el compilador a: **/usr/bin/python3**.

Errores comunes

Una vez instalado Anjuta es posible que al compilar el primer proyecto, marque un error de las librerías **libtool**, por lo regular estas librerías se instalan junto con el Anjuta. Sin embargo, la dependencia no es con respecto a las librerías sino con respecto a los binarios de dichas librerias, por lo tanto para instalarlo es necesario ejecutar:

```
1 sudo apt-get install libtool-bin
```

Una vez instalados los binarios de la librería, con el comando anterior, el problema se resuelve.

2.4. ¿Que es Vala?

Vala es un nuevo lenguaje de programación que permite usar técnicas de programación modernas para desarrollar aplicaciones que se ejecutan usando las bibliotecas de GNOME, aunque también es posible ejecutarlo en otros sistemas operativos y entornos gráficos, debido a sus bajas dependencias (GLib y GObject). Esta plataforma de desarrollo provee de un entorno completo de programación; con funcionalidades como el sistema de tipos dinámicos y el gestor de memoria asistida. Antes de la existencia de Vala, la única forma de programar para la plataforma era con la API nativa de C, o utilizando un lenguaje de programación de alto nivel que usan máquinas virtuales, como Python o C#, que necesitan un wrapper para usar esa biblioteca.

Vala es un lenguaje totalmente diferente de otros lenguajes y otras técnicas, ya que el compilador genera código en lenguaje C, que puede ser compilado para ser ejecutado sin ninguna biblioteca extra ni wrapper intermedio. Esto tiene una serie de consecuencias, pero entre las mas importantes se encuentran:

1. Los programas escritos en Vala debería tener un desempeño similar al mismo programa directamente escrito en lenguaje C. Siendo mucho más fácil y rápido de escribir y mantener.

2. Una aplicación escrita en Vala no puede hacer nada que un programa equivalente escrito en lenguaje C no pueda hacer. Sin embargo Vala introduce una serie de funcionalidades que no están disponibles en C, las cuales se mapean en construcciones escritas en C, siendo estas últimas difíciles y complejas de escribir directamente.

Vala es, por lo tanto, un lenguaje de programación moderno con todas las funcionalidades que se puede esperar de una plataforma actual (Python, .NET, etc).

Vala comparte bastante sintaxis con C#, pero algunas otras no se corresponden con construcciones de este lenguaje, por tanto no se entrará en comparación a menos que sea necesario, para evitar que este libro esté enfocado a programadores de C#.

Es importante darse cuenta que Vala realmente se ejecuta como un programa escrito en C compilado, y puede interactuar con bibliotecas escritas en C. Por tanto un conocimiento de C puede hacer comprender el funcionamiento de Vala.

2.5. Primer Programa de Vala

El primer programa en Vala va a ser el conocido como "Hola Mundo"que es el programa más simple (sin contar con el programa vacío) que se puede escribir en un lenguaje de programación.

El listado de código sería algo como lo que sigue:

```
1 class Demo.HelloWorld : GLib.Object {
2         public static int main(string[] args) {
3                 stdout.printf("Hola, mundo!\n");
4                 return 0;
5         }
6 }
```

La primera línea que se ve es:

```
1 class Demo.HelloWorld : GLib.Object {
```

Esta línea de código representa la definición de una nueva clase llamada HelloWorld. Se puede observar que antes del nombre de la clase se encuentra la palabra "Demo" seguida de un punto. Bien, esto indica el espacio de nombres en el que se mueve el programa. Asimismo, se puede observar que después del nombre de la nueva clase le siguen dos puntos y la cadena "Glib.Object". Esta cadena va en el mismo formato que la anterior, es decir, "espacio de nombres"."nombre de la clase" e indica una clase de la que hereda la clase anterior. La mayoría de las nuevas clases heredarán de la clase "GLib.Object" que puede ser considerada como la clase básica en el lenguaje Vala, ya que, una parte importante de las funcionalidades del lenguaje están supeditadas a que la clase sobre las que se aplican hereden de esta clase o de una clase descendiente de ésta. Como se ha podido ver la sintaxis de definición de una clase es bastante similar a otros lenguajes de programación como C++, Java o C#.

Siguiendo con el mismo listado de código tenemos que la siguiente línea de código es la que sigue a continuación:

```
1 public static int main(string[] args) {
```

Esta línea de código define un método de la clase HelloWorld llamado "main". Este método va precedido por la palabra reservada "public" que indica que se trata de un método público (es decir, puede ser llamado externamente a la clase a la que pertenece). La palabra reservada "static" nos indica que se trata de un método estático (está asociado a la clase y no a un objeto en concreto, es decir, puede ser llamado sin instanciar ningún objeto

de esa clase). Además se indica que el tipo de datos que devuelve este método es de tipo entero, lo que se hace mediante la palabra reservada "int". Por último, entre los paréntesis se indican los parámetros (los datos que utilizará internamente el método para trabajar). El parámetro es una lista de cadenas que contendrá los parámetros con los que fue llamado el programa desde la línea de comandos. El hecho de que el método definido se llame "main" no es casual, ya que, Vala considera a este método concreto como el punto de entrada del programa, es decir, que una vez compilado un programa escrito en Vala éste será el primer método en ser ejecutado. Este método es el que contiene la correspondientes inicializaciones que requiere el programa que estemos desarrollando, por ejemplo la inicialización de la interfaz gráfica de usuario.

```
1 stdout.printf("Hola, mundo!\n");
```

Esta línea de código imprime por la salida estándar (normalmente será la pantalla del ordenador) el mensaje encerrado entre comillas. Es una llamada al método "printf" del objeto "stdout". En el lenguaje de programación Vala existen una serie de objetos predefinidos a los cuales tenemos acceso por que se encuentran definidos dentro del espacio de nombres GLib que está definido por defecto en cualquier programa Vala. El objeto "stdout" nos permite el acceso a la salida estándar de la máquina y entre otros métodos contiene el método "printf" que muestra un mensaje formateado por dicha salida.

La última línea importante del código inicial es la siguiente:

```
1 return 0;
```

Esta línea hace que el programa finalice y devuelva un valor de 0 al sistema. Este valor normalmente se usa para indicar si un programa ha finalizado de forma correcta o no. Si estuviera definido en otro método cualquiera (que no fuera el punto de entrada del programa) devolvería el valor y se almacenaría en una variable. Por ejemplo:

```
1 int valor = objeto.metodo();
```

Las últimas líneas del listado únicamente cierran la definición de la clase y el método definido como en cualquier otro lenguaje como por ejemplo C++ o Java.

2.6. Compilación del programa en Vala

El siguiente paso es compilar dicho programa. Vala al ser un lenguaje de programación compilado y por lo tanto necesita de un compilador. El lenguaje de programación Vala dispone de un compilador llamado "valac" y que es compilador al mismo tiempo del lenguaje de programación Genie.

Asumiendo que tenemos el compilador de Vala instalado en nuestra máquina, para compilar el primer programa "Hola mundo" suponiendo que está escrito en un fichero llamado "hola.vala" sería:

```
1 valac hola.vala
```

Esto generaría un fichero ejecutable llamada "hola" (dependiendo del sistema operativo en el que nos encontremos el fichero binario resultante tendrá o no una extensión como por ejemplo la ".exe" en Windows). Para ejecutar dicho programa habrá que escribir en la línea de comandos algo como:

```
1 ./hola
```

Y nos mostrará la salida:

```
1 Hola, mundo!
```

Los archivos de código fuente de vala tienen, normalmente, la extensión ".vala". El lenguaje de programación Vala no fuerza a que los proyectos tengan una determinada estructura, en cuanto a los paquetes o los nombres de los archivos que contiene una clase, como hacen otros lenguajes como Java. En lugar de eso la estructura se define dentro de los archivos de código mediante texto, definiendo la localización y estructura lógica mediante elementos como los espacios de nombres. Cuando se quiere compilar código Vala, se le pasa al compilador una lista de los archivos de código fuente necesarios, y el compilador determina como ensamblarlos todos juntos.

La ventaja de todo esto es que se pueden definir tantas clases o funciones como se desee dentro de un único archivo de código, incluso combinando distintos espacios de nombres todos dentro del mismo archivo (aunque no se demasiado recomendable por cuestiones de legibilidad y estructura). Vala por lo tanto no exige de manera inherente una determinada estructura para los proyectos que se desarrollan usando esta plataforma. Sin embargo, si existen ciertas convenciones a la hora de estructurar un proyecto desarrollado en Vala. Un buen ejemplo sería como se estructura el propio proyecto del compilador de Vala.

Todos los archivos de código que pertenezcan al mismo proyecto son suministrados al compilador "valac" mediante la línea de comandos, junto con los correspondientes parámetros de compilación. Esto funciona de forma similar a como se haría con el código fuente compilado en Java. Por ejemplo:

```
1 $ valac compiler.vala --pkg libvala
```

Esa línea de código produciría un archivo binario llamado çompiler"que sería enlazado con el paquete "libvala"(cuando nos referimos a paquete lo estamos haciendo a un concepto similar a biblioteca de funciones o la mal traducida librería). De hecho, así es como se genera el compilador de Vala en realidad.

Si se quiere que el archivo ejecutable producido tenga un nombre específico (distinto al que el compilador le da) o si se le pasan varios archivos al compilador, es posible especificar el nombre del fichero ejecutable mediante la opción "-o":

```
1 $ valac source1.vala source2.vala -o myprogram
2 $ ./myprogram
```

Como se ha comentado anteriormente el compilador de vala es capaz de generar código en lenguaje C en lugar de un archivo ejecutable. Así para realizar esto existe la opción "-C" que indicará al compilador que genere todo el código C necesario para crear el ejecutable del proyecto. El compilador creará un archivo con extensión ".c" por cada archivo con extensión ".vala" que le pasemos. Si abrimos el contenido de esos archivos generados se puede ver el código C equivalente al programa que hemos desarrollado en Vala. Se puede observar como se crean las mismas clases que en Vala pero mediante la biblioteca GObject y cómo se registran de forma dinámica en el sistema. Este es un ejemplo del poder que tiene la plataforma de desarrollo de GNOME. Sin embargo, todo esto es a título informativo, ya que, por suerte no es necesario generar todos estos ficheros, ni entender como funcionan internamente para poder programar en Vala. Si en lugar de generar el fichero de código C sólo necesitamos generar la cabecera (por ejemplo si estamos generando una biblioteca de funciones será necesario para poder usarla en C o en otro lenguaje) existe la opción "-H" con la que conseguiremos ese objetivo. A continuación un ejemplo que genera los dos ficheros (un fichero de código C y su correspondiente cabecera ".h"):

```
1 $ valac hello.vala -C -H hello.h
```

Capítulo 3

Configurando FreeBSD

3.1. Cambiar idioma al español

En la carpeta local de los usuarios (esta carpeta se encuentra en /usr/home/TU_USUARIO).

```
ee .login_conf

me:
:charset=ISO-8859-15:
:lang=es_ES.ISO8859-15:
:tc=default:
```

Ahora se configura el archivo .profile

```
ee .profile

LANG=es_ES.ISO8859-15; export LANG
MM_CHARSET=ISO-8859-15; export MM_CHARSET
```

Se configura el archivo de inicio de xwindows

```
ee .xinitrc

LANG=es_ES.ISO8859-15; export LANG
setenv LANG es_ES.ISO8859-15
```

3.2. Como instalar Xwindows en freebsd para ARM

Para la instalación de Xwindows en la tarjeta embebida con FreeBSD es necesario instalar los controladores primero:

```
# pkg install xf86-video-scfb xf86-video-fbdev
# pkg install xorg
```

Crear el archivo xorg.conf ejecutando el comando:

```
ee /etc/X11/xorg.conf
```

y capturar el siguiente código:

```
Section "Device"
Identifier "Generic␣FB"
Driver "scfb"
EndSection

Section "Screen"
Identifier "Screen"
Device "Generic␣FB"
Monitor "Monitor"
SubSection "Display"
Depth 16 #24 #32
EndSubsection
EndSection
```

Una vez configurado XWINDOWS se ejecuta con el comando:

```
#startx
```

Para instalar el ambiente de escritorio MATE se ejecuta el siguiente comando:

```
pkg install  mate-desktop mate
```

para ejecutar MATE:

```
xinit mate-session
```

Capítulo 4

Sintaxis de Vala

La sintaxis de Vala es una mezcla, basada principalmente en la de C#. Como resultado, la mayor parte de esta debería ser familiar para los programadores que conozcan cualquier lenguaje similar a C. Teniendo esto en cuenta, se hará una descripción breve de la misma.

El alcance de declaración se define mediante corchetes. Un objeto o referencia sólo es válido entre y . Estos delimitadores se usan también para definir clases, métodos, bloques de código, etc, de manera que tengan automáticamente su propio alcance. Vala no es estricto acerca de dónde se declaran las variables.

Un identificador se define por su tipo y nombre, por ejemplo, int c describe un entero llamado c. En el caso de los tipos por valor, esto también crea un objeto del tipo dado. Para tipos referenciados esto simplemente define una nueva referencia que no apunta a nada inicialmente.

Para los nombres de identificadores, se aplican las mismas reglas que para los identificadores de C: el primer carácter debe estar entre [a-z], [A-Z] o subrayados (_), los siguientes pueden estar entre estos y [0-9]. No se permiten otros caracteres Unicode. Es posible usar palabras reservadas como identificadores, añadiéndoles el carácter @ como prefijo. Este carácter no forma parte del nombre. Por ejemplo, puede llamar a un método foreach escribiendo @foreach, siendo esta una palabra reservada de Vala.

Los tipos por referencia se instancian empleando el operador new y el nombre de un método de construcción, que generalmente es el propio nombre del tipo, por ejemplo, **Object o = new Object()** crea un **Object** nuevo y convierte a o en una referencia a este.

4.1. Comentarios

Vala permite varias maneras de escribir comentarios en el código:

```
1 // Los comentarios continuan hasta el final de la linea
2
3 /* Los comentarios estan entre los delimitadores*/
4
5 /**
6 * Comentarios de Documentos
7 */
```

Estos comentarios se manejan de la misma manera que en la mayoría de lenguajes, y requieren por ello poca explicación. Los comentarios para documentación no son específicos de Vala, pero una herramienta de generación de documentación como Valadoc los reconoce.

4.2. Tipos de datos

En general, hay dos tipos de dato en Vala: **tipos referenciados** y **tipos por valor**. Estos nombres describen cómo se usan en el sistema las instancias de los tipos; un tipo por valor se copia cada vez que se asigna a un identificador nuevo, un tipo referenciado no se copia, en lugar de esto el nuevo identificador es una nueva referencia al mismo objeto.

Una constante se define poniendo **const** delante del tipo. El convenio para constantes es **TODO_EN_MAYUSCULAS**.

4.3. Tipos por valor

Vala soporta un conjunto de tipos simples, como la mayoría de lenguajes.

- Byte: **char, uchar**; sus nombres son char por motivos históricos.
- Carácter: **unichar**; un carácter Unicode de 32 bits.
- Entero: **int, uint**.
- Entero «largo»: **long,ulong**.
- Entero «corto»: **short, ushort**.

- Entero de tamaño garantizado: **int8, int16, int32, int64**, así como sus versiones sin signo **uint8, uint16, uint32, uint64**. Los números indican las longitudes en bits.
- Número real en coma flotante: **float, double**.
- Booleano, bool: sus valores posibles son **true (verdadero)** y **false (falso)**.
- Compuesto: **struct**.
- Enumerado: **enum**; representado por valores enteros, no por clases, como las enumerados de Java.

A continuación se muestran algunos ejemplos.

```
/* Tipos Atomicos */
unichar c = 'u';
float percentile = 0.75f;
const double MU_BOHR = 927.400915E-26;
bool the_box_has_crashed = false;

/* Definiendo una estructura */
struct Vector {
        public double x;
        public double y;
        public double z;
}

/* Definiendo un enum */
enum WindowType {
        TOPLEVEL,
        POPUP
}
```

La mayoría de estos tipos pueden tener diferentes tamaños en distintas plataformas, exceptuando los tipos de tamaño garantizado. El operador **sizeof** devuelve el tamaño que ocupa, en bytes, una variable de un tipo dado:

```
ulong nbytes = sizeof(int32); // nbytes will be 4 (= 32 bits)
```

Se pueden determinar los valores máximo y mínimo de un tipo numérico con **.MIN** y **.MAX**, por ejemplo, **int.MIN** e **int.MAX**.

4.4. Cadena de Caracteres

El tipo de dato para cadenas de caracteres es **string**. Las cadenas de Vala están codificadas en **UTF-8** y son inmutables.

```
1 string text = "A␣string␣literal";
```

Vala ofrece una característica llamada cadenas literales («verbatim strings»). Son cadenas de caracteres en las cuales las secuencias de escape (como puede ser \n) no se interpretan, los saltos de línea se conservan y los signos de puntuación no tienen que enmascararse. Estas cadenas se delimitan con tres comillas dobles. Las posibles sangrías tras un salto de línea también forman parte de la cadena.

```
1 string verbatim = """This␣is␣a␣so-called␣"verbatim string".
2 Verbatim␣strings␣don't␣process␣escape␣sequences,␣such␣as␣\n,␣
     \t,␣\\,␣etc.
3 They␣may␣contain␣quotes␣and␣may␣span␣multiple␣lines.""";
```

Las cadenas precedidas por @ son consideradas patrones. En ellas se evalúan las variables y expresiones precedidas por $:

```
1 int a = 6, b = 7;
2 string s = @"$a␣*␣$b␣=␣$(a␣*␣b)";   // => "6 * 7 = 42"
```

Los operadores de igualdad == y != comparan el contenido de dos cadenas de caracteres, a diferencia del comportamiento de Java, donde se compara la igualdad entre referencias.

Es posible dividir una cadena con [inicio:fin]. Los valores negativos representan posiciones relativas al final de la cadena:

```
1 string greeting = "hello,␣world";
2 string s1 = greeting[7:12];          // => "world"
3 string s2 = greeting[-4:-2];         // => "or"
```

Obsérvese que los índices en Vala empiezan en 0 como en la mayoría de lenguajes de programación. Desde Vala 0.11 se puede acceder a un solo byte mediante [índice]:

```
1 uint8 b = greeting[7];               // => 0x77
```

Sin embargo, no se puede asignar un byte nuevo a esta posición ya que, como se ha comentado anteriormente, las cadenas de caracteres en Vala son inmutables.

Muchos de los tipos básicos tienen métodos intuitivos para tomar el valor de una cadena, así como para convertir su valor a una cadena, por ejemplo:

```
bool b = bool.parse("false");            // => false
int i = int.parse("-52");                // => -52
double d = double.parse("6.67428E-11"); // => 6.67428E-11
string s1 = true.to_string();            // => "true"
string s2 = 21.to_string();              // => "21"
```

Dos métodos útiles para escribir y leer cadenas en y desde la línea de comandos (y para las primeras pruebas con Vala) son, respectivamente, **stdout.printf()** y **stdin.read_line()**:

```
stdout.printf("Hello, world\n");
stdout.printf("%d %g %s\n", 42, 3.1415, "Vala");
string input = stdin.read_line();
int number = int.parse(stdin.read_line());
```

Ya se ha presentado **stdout.printf()** en el ejemplo HolaMundo. Este método acepta un número arbitrario de argumentos de distintos tipos, mientras que el primer argumento es una cadena de formato, que sigue las mismas reglas que las cadenas de formato de C. Si se requiere mostrar un mensaje de error, se puede emplear ***stderr.printf()*** en lugar de ***stdout.printf()***.

Adicionalmente, la operación in puede utilizarse para determinar si una cadena contiene a otra, por ejemplo:

```
if ("ere" in "Able was I ere I saw Elba.") ...
```

4.5. Matrices

Las matrices se declaran dando un nombre de tipo seguido de **[]** y se crean usando el operador **new**, por ejemplo, **int[]a = new int[10]** para crear una matriz de enteros. La longitud de estas matrices se puede obtener accediendo al miembro variable **length**, por ejemplo, **int count = a.length**. Se debe tener en cuenta que si se escribe **Object[] a = new Object[10]** no se creará ningún objeto, sólo la matriz para almacenarlos.

```
int[] a = new int[10];
int[] b = { 2, 4, 6, 8 };
```

Una matriz se puede dividir con [inicio:fin]:

```
int[] c = b[1:3];      // => { 4, 6 }
```

El resultado de dividir una matriz es una referencia a los datos requeridos, no una copia de estos. Sin embargo, si se asigna este resultado a una variable owned (como se ha hecho antes) sí que se copian los datos. Si se quiere evitar

la copia, se puede optar por asignar el resultado a una matriz unowned o se puede pasar directamente a un argumento (los argumentos son, de manera predeterminada, unowned).

```
unowned int[] c = b[1:3];      // => { 4, 6 }
```

Las matrices multidimensionales se definen con [,] o [„], etc.

```
int[,] c = new int[3,4];
int[,] d = {{2, 4, 6, 8},
            {3, 5, 7, 9},
            {1, 3, 5, 7}};
d[2,3] = 42;
```

Este tipo de matriz se representa como un bloque de memoria contigua. Las matrices multidimensionales compuestas ([][], también conocidas como "matrices apiladas" o "matrices de matrices"), en las que cada fila puede tener diferente longitud, no están soportadas aún.

Para obtener la longitud de cada dimensión de una matriz multidimensional, el miembro length pasa a ser una matriz que almacena la longitud o cada dimensión respectivamente.

```
int[,] arr = new int[4,5];
int r = arr.length[0];
int c = arr.length[1];
```

Tenga en cuenta que se no se pueden obtener una matriz de una dimensión a partir de una matriz multidimensional, ni siquiera dividir una matriz multidimensional:

```
int[,] arr = {{1,2},
              {3,4}};
int[] b = arr[0];   // won't work
int[] c = arr[0,];   // won't work
int[] d = arr[:,0];   // won't work
int[] e = arr[0:1,0];   // won't work
int[,] f = arr[0:1,0:1];   // won't work
```

Se pueden añadir elementos a una matriz dinámicamente haciendo uso del operador +=. Sin embargo, sólo funciona para las matrices definidas localmente o privadas. La matriz se reubica automáticamente si es necesario. Internamente esta reubicación se realiza con tamaños crecientes en potencias de 2 por razones de eficiencia en tiempo de ejecución. No obstante, .length contiene el número actual de elementos, no el tamaño interno.

```
int[] e = {};
```

```
2 e += 12;
3 e += 5;
4 e += 37;
```

El tamaño de una matriz se puede variar llamando a resize() sobre esta. Esta operación mantiene el contenido original (tanto de éste como quepa en el nuevo tamaño).

```
1 int[] a = new int[5];
2 a.resize(12);
```

Si se ponen corchetes después del identificador junto a una indicación de tamaño, se creará una matriz de tamaño fijo. Las matrices de tamaño fijo se almacenan en la pila (si se usan como variables locales) o se almacenan in-line (si se usan como campos) y no se pueden reubicar más tarde.

```
1 int f[10];      // no 'new ...'
```

Vala no comprueba los límites en tiempo de ejecución en el acceso a matrices. Si se requiere mayor seguridad, se debería usar una estructura de datos más sofisticada, como el ArrayList.

4.6. Tipos referenciados

Los tipos referenciados son todos los tipos declarados como una clase, independientemente de si descienden o no del Object de GLib. Vala se asegurará de que cuando se pasa un objeto por referencia el sistema mantenga la cuenta del número de referencias realizadas actualmente con el propósito de gestionar la memoria de forma automática. El valor de una referencia que no apunta a nada es null.

```
1 /* defining a class */
2 class Track : GLib.Object {              /* subclassing 'GLib.
    Object' */
3         public double mass;                    /* a public field
              */
4         public double name { get; set; }     /* a public
              property */
5         private bool terminated = false;     /* a private
              field */
6         public void terminate() {            /* a public
              method */
7                 terminated = true;
8         }
```

```
9 }
```

4.7. Promoción de tipos estáticos

En Vala es posible promocionar una variable de un tipo a otro. Para promocionar una variable de un tipo estático se escribe el tipo deseado entre paréntesis. La promoción estática no impone ningún tipo de comprobación de seguridad de los tipos en tiempo de ejecución. Funciona para todos los tipos disponibles en Vala. Por ejemplo:

```
1 int i = 10;
2 float j = (float) i;
```

Vala soporta otro mecanismo de promoción llamado *promoción dinámica* que realiza comprobación de tipos en tiempo de ejecución y se describe en la sección sobre programación orientada a objetos.

4.8. Inferencia de Tipos

Vala cuenta con un mecanismo llamado *inferencia de tipos*, con la que una variable local se puede definir usando ***var*** en vez de indicar un tipo, mientras que no haya ambigüedad en el tipo que se ha querido indicar. El tipo se infiere de la asignación del lado derecho del igual. Esto ayuda a reducir redundancia innecesaria en el código sin sacrificar el tipado estático:

```
1 var p = new Person();      // same as: Person p = new Person()
    ;
2 var s = "hello";           // same as: string s = "hello";
3 var l = new List<int>();   // same as: List<int> l = new List<
    int>();
4 var i = 10;                // same as: int i = 10;
```

Este mecanismo sólo funciona para variables locales. La inferencia de tipos es especialmente útil para tipos con argumentos genéricos. Compare:

```
1 MyFoo<string, MyBar<string, int>> foo = new MyFoo<string,
    MyBar<string, int>>();
```

frente a:

```
1 var foo = new MyFoo<string, MyBar<string, int>>();
```

4.9. Definir un tipo nuevo a partir de otro

Definir un tipo nuevo significa derivarlo del que se necesite. Por ejemplo:

```
1 /* Define a new type from a container like GLib.List with
     elements type GLib.Value */
2 public class ValueList : GLib.List<GLib.Value> {
3         [CCode (has_construct_function = false)]
4         protected ValueList ();
5         public static GLib.Type get_type ();
6 }
```

4.10. Operadores

=

Asignación. El operando de la izquierda debe ser un identificador, y el de la derecha un valor o referencia, según el caso.

+, -, /, *, %

Aritmética básica, aplicada a los operandos izquierdo y derecho. El operador + también puede concatenar cadenas de caracteres.

+=, -=, /=, *=, %=

Operaciones aritméticas entre los operandos izquierdo y derecho, donde el operando de la izquierda debe ser un identificador al que se asigna el resultado.

++, –

Operaciones de incremento y decremento con asignación implícita. Estos operandos sólo requieren un argumento, que debe ser un identificador de un tipo de dato simple. El valor se cambia y se asigna de nuevo al identificador. Estos operadores deben situarse como prefijo o como sufijo; en el primer caso el valor que se evalúe será el recién calculado, mientras que en el segundo caso se evaluará el valor original.

|, ^ , &, ~ , |=, &=, ^=

Operaciones a nivel de bit: disyunción (o ó "or"), disyunción exclusiva ("xor"), conjunción (y o "and") y negación. Los operadores del segundo grupo incluyen asignación y son análogos a las versiones aritméticas. Todos estos operadores se pueden aplicar a cualquier tipo de dato simple. El hecho de que no exista un operador de asignación asociado a ~ se debe a que este es un operador unario. La operación equivalente es simplemente a = ~a.

«, »

Operaciones de desplazamiento de bits. Desplazan el operando de la izquierda un número de bits de acuerdo al operando de la derecha.

<<=

,

>>=

Operaciones de desplazamiento de bits. Desplazan el operando de la izquierda un número de bits de acuerdo al operando de la derecha. El operando de la izquierda debe ser un identificador, al que se asigna el resultado.

==

Comprobación de igualdad. Evalúa a un valor bool dependiendo de si los operandos izquierdo y derecho son iguales. En caso de los tipos por valor esto significa que los valores son iguales, en el caso de los tipos por referencia implica que los objetos sean la misma instancia. Una excepción a esta regla es el tipo string, que se evalúa según la igualdad del valor.

<, >, >=, <=, !=

Comprobaciones de desigualdad. Se evalua a un valor bool dependiendo de si los operandos izquierdo y derecho son distintos de la manera descrita en cada caso. Son validos para tipos por valor, y para el tipo string. En este último caso estos operadores compara orden lexicográfico.

!, &&, ||

Operaciones lógicas: negación (no), conjunción (y) y disyunción (o). Estos operadores se pueden aplicar a valores booleanos, requiriendo el primero un solo valor y los otros, dos valores.

? :

Operador condicional ternario. Evalúa una condición y devuelve el valor de la sub-expresión a la izquierda o a la derecha basándose en la veracidad de la condición: condición ? valor si es cierta : valor si es falsa.

??

Operador de comprobación de nulidad y asignación: a ?? b es equivalente a a != null ? a : b. Este operador es útil, por ejemplo, para asignar un valor predeterminado en caso de que una referencia sea nula:

```
1 stdout.printf("Hello, %s!\n", name ?? "unknown person");
```

Comprueba si el operando de la derecha contiene al operando de la izquierda. este operador funciona en matrices, cadenas de caracteres, colecciones de datos o cualquier otro tipo que tenga un método contains() apropiado. Para cadenas puede realizar búsqueda de subcadenas.

Los operadores no se pueden sobrecargar en Vala. Hay operadores adicionales, válidos en el contexto de las declaraciones lambda y otras tareas específicas; estos se explicaran en el contexto en el que sean aplicables.

in

Comprueba si el operando de la derecha contiene al operando de la izquierda. este operador funciona en matrices, cadenas de caracteres, colecciones de datos o cualquier otro tipo que tenga un método contains() apropiado. Para cadenas puede realizar búsqueda de subcadenas.

Los operadores no se pueden sobrecargar en Vala. Hay operadores adicionales, válidos en el contexto de las declaraciones lambda y otras tareas específicas; estos se explicaran en el contexto en el que sean aplicables.

4.11. Estructuras de Control

A continuación se muestran varios ejemplos de código de diferentes estructuras de control y el efecto de dicho código:

```
1 while (a > b) { a--; }
```

Decrementa a repetidamente, comprobando antes de cada iteración que a sea mayor que b.

```
1 do { a--; } while (a > b);
```

Decrementa a repetidamente, comprobando después de cada iteración que a sea mayor que b.

```
1 for (int a = 0; a < 10; a++) { stdout.printf("%d\n", a); }
```

Inicializa a a 0, después se imprime a repetidamente hasta que su valor deje de ser mayor de 10, el valor de a se incrementa tras cada iteración.

```
1 foreach (int a in int_array) { stdout.printf("%d\n", a); }
```

Imprime en la salida cada entero de una matriz, o cualquier otra colección iterable. El significado de «iterable» se describirá más adelante.

Los cuatro tipos anteriores de bucle se pueden controlar con las palabras reservadas break y continue. Una instrucción break provocará que el bucle termine inmediatamente, mientras que continue saltará a la parte de la comprobación de la iteración (terminando así la iteración actual, pero sin salir del bucle).

```
1 if (a > 0) { stdout.printf("a␣is␣greater␣than␣0\n"); }
2 else if (a < 0) { stdout.printf("a␣is␣less␣than␣0\n"); }
3 else { stdout.printf("a␣is␣equal␣to␣0\n"); }
```

Ejecuta partes de código concretas dependiendo de un conjunto de condiciones. La primera condición que cumplir decide qué código se ejecuta, si a es mayor que 0 no se realizará la comprobación de si es menor que 0. Se permite cualquier cantidad de bloques else if, y uno o ningún bloque else.

```
1 switch (a) {
2         case 1:
3         stdout.printf("one\n");
4         break;
5         case 2:
6         case 3:
7         stdout.printf("two␣or␣three\n");
8         break;
```

```
9           default:
10          stdout.printf("unknown\n");
11          break;
12  }
```

Una sentencia switch ejecuta exactamente una o ninguna sección de código basándose en el valor que se le pasa. En Vala no hay paso a través de los distintos casos (bloques etiquetados con la palabra reservada case), excepto en los casos vacíos. Para asegurar esto, cada case no vacío debe terminar con una de las palabras reservadas break, return o throw. Se pueden utilizar las sentencias switch con cadenas de caracteres.

Como nota para los programadores de C: las condiciones deben evaluarse siempre un valor booleano. Es decir, que si uno quiere comprobar si una variable es nula o 0, debe hacerlo explícitamente: if (object != null) { } or if (number != 0) { }.

4.12. Elementos del lenguaje

4.12.1. Métodos

En Vala, las funciones se llaman métodos, independientemente de si están definidas dentro de una clase o no. Desde este momento se empleará el término método.

```
1  int method_name(int arg1, Object arg2) {
2          return 1;
3          }
```

El anterior fragmento de código define un método con el nombre method_name, que toma dos argumentos, un entero y un Object (el primero pasado por valor y el segundo por referencia, como se ha descrito anteriormente). El método devuelve un entero, que en este caso es 1.

Todos los métodos Vala son funciones C, y por ello toman un numero arbitrario de argumentos y devuelven un valor (o ninguno si el método se declara como void). Se puede aproximar un mecanismo para devolver más valores, poniendo datos en lugares conocidos por el código que llama al método. Los detalles de este mecanismo se explicarán en la sección "Dirección de los parámetros", en la parte avanzada de este tutorial.

El convenio para los nombres de métodos en Vala es todo_en_minúscula con subrayados como separadores entre palabras. Este proceder puede parecer

poco familiar a programadores de Java o C#, acostumbrados a los nombres de métodos en CamelCase o mixedCamelCase. Pero con este estilo se consigue consistencia con otras bibliotecas de Vala y C/GObject.

No es posible tener varios métodos con el mismo nombre pero distinto prototipo dentro del mismo ámbito de definición (mecanismo conocido como "sobrecarga de métodos"):

```
1 void draw(string text) { }
2 void draw(Shape shape) { }  // not possible
```

Esto se debe al hecho de que se pretende que las librerías producidas con Vala puedan ser utilizadas también por programadores de C. En su lugar, en Vala se haría lo siguiente:

```
1 void draw_text(string text) { }
2 void draw_shape(Shape shape) { }
```

Eligiendo nombres ligeramente distintos puede evitarse la colisión entre los nombres. En lenguajes que soportan sobrecarga de métodos, esta suele emplearse para ofrecer métodos de conveniencia con menos parámetros que enlazan con un método más general:

```
1 void f(int x, string s, double z) { }
2 void f(int x, string s) { f(x, s, 0.5); }  // not possible
3 void f(int x) { f(x, "hello"); }            // not possible
```

En este caso puede usarse la característica predeterminada de los argumentos de Vala con el fin de conseguir un comportamiento parecido con un solo método. Pueden definirse valores por defecto para los últimos parámetros de un método, para evitar pasarlos explícitamente al llamar al método:

```
1 void f(int x, string s = "hello", double z = 0.5) { }
```

Algunas llamadas posibles a este método podrían ser:

```
1 f(2);
2 f(2, "hi");
3 f(2, "hi", 0.75);
```

Incluso se puede definir métodos con una lista de argumentos de longitud variable (varargs) como stdout.printf, aunque no es del todo recomendable. Más adelante se hablará de esto.

Vala realiza una comprobación de nulidad básica en los parámetros y los valores devueltos de los métodos. Si se permite que un parámetro o un valor de retorno sea null, el símbolo del tipo debe ir seguido con un modificador

?. Esta información adicional ayuda al compilador de Vala a realizar comprobaciones estáticas y añadir verificaciones en tiempo de ejecución en las precondiciones de los métodos, lo que puede ayudar a evitar errores relacionados como eliminar una referencia a un puntero a null.

```
1 string? method_name(string? text, Foo? foo, Bar bar) {
2         // ...
3 }
```

En este ejemplo text, foo y el valor de retorno pueden ser null, no obstante bar no debe ser null.

4.13. Delegados

```
1 delegate void DelegateType(int a);
```

Los delegados («delegate») representan métodos, permitiendo que se puedan pasar fragmentos de código entre objetos. En el ejemplo de arriba se define un tipo llamado DelegateType que representa métodos que toman un int y no devuelven nada. Cualquier método que concuerde con este prototipo puede asignarse a una variable de este tipo o pasarse como un argumento de este tipo a un método.

```
1  delegate void DelegateType(int a);
2
3  void f1(int a) {
4          stdout.printf("%d\n", a);
5  }
6
7  void f2(DelegateType d, int a) {
8          d(a);       // Calling a delegate
9  }
10
11 void main() {
12         f2(f1, 5);  // Passing a method as delegate argument
               to another method
13 }
```

En este código se ejecuta el método f2, pasándole una referencia al método f1 y el número 5. f2 ejecutará entonces el método f1, pasándole a este el número.

Los delegados también pueden crearse localmente. Un método miembro de una clase puede asignarse a un delegado, por ejemplo:

```
class Foo {

        public void f1(int a) {
                stdout.printf("a␣=␣%d\n", a);
        }

        delegate void DelegateType(int a);

        public static int main(string[] args) {
                Foo foo = new Foo();
                DelegateType d1 = foo.f1;
                d1(10);
                return 0;
        }
}
```

4.14. Métodos anónimos o clausuras

```
(a) => { stdout.printf("%d\n", a); }
```

Un método anónimo, también conocido como expresión lambda, función literal o clausura, puede definirse en Vala con el operador =>. La lista de parámetros se sitúa al lado izquierdo del operador, mientras que el cuerpo del método va en el lado derecho.

Un método anónimo por sí mismo, como el del ejemplo de arriba, no tiene mucho sentido. Estos métodos sólo son útiles si se asignan directamente a una variable de un tipo delegado o si se pasan como argumento a otro método.

Tenga en cuenta que ni los tipos de los parámetros ni los del valor de retorno se indican explícitamente. En lugar de esto, los tipos se infieren del prototipo del delegado con el que se utilizan.

A continuación se muestra un ejemplo de asignación de un método anónimo a una variable delegada:

```
delegate void PrintIntFunc(int a);

void main() {
        PrintIntFunc p1 = (a) => { stdout.printf("%d\n", a);
           };
                p1(10);

                // Curly braces are optional if the body
                   contains only one statement:
                PrintIntFunc p2 = (a) => stdout.printf("%d\n"
                   , a);
```

```
9                   p2(20):
10          }
```

Y seguidamente el paso de un método anónimo a otro método:

```
1  delegate int Comparator(int a, int b);
2
3  void my_sorting_algorithm(int[] data, Comparator compare) {
4          // ... 'compare' is called somewhere in here ...
5  }
6
7  void main() {
8          int[] data = { 3, 9, 2, 7, 5 };
9          // An anonymous method is passed as the second
               argument:
10         my_sorting_algorithm(data, (a, b) => {
11                 if (a < b) return -1;
12                 if (a > b) return 1;
13                 return 0;
14         });
15 }
```

Los métodos anónimos son verdaderas clausuras. Esto significa que se puede acceder a las variables locales del método externo desde el interior de la expresión lambda:

```
1  delegate int IntOperation(int i);
2
3  IntOperation curried_add(int a) {
4          return (b) => a + b;  // 'a' is an outer variable
5  }
6
7  void main() {
8          stdout.printf("2␣+␣4␣=␣%d\n", curried_add(2)(4));
9  }
```

En este ejemplo, curried_add (consulte Currificación) devuelve un método recién creado que conserva el valor de a. El método devuelto se llama directamente más adelante con 4 como argumento, dando como resultado la suma de los dos números.

4.15. Espacios de Nombres

```
1  namespace NameSpaceName {
2          // ...
3  }
```

Todo lo que se encuentra entre las llaves en el código anterior está dentro del espacio de nombres **NameSpaceName** y se debe hacer referencia a ello como tal. Cualquier fragmento de código fuera de este espacio de nombres, deberá usar nombres completos para acceder a todo lo que se encuentra dentro, o bien estar situado en un archivo con la declaración **using** adecuada para importar este espacio de nombres:

```
1 using NameSpaceName;
2
3 // ...
```

Por ejemplo si el espacio de nombres Gtk se importa con **using Gtk;** se puede escribir simplemente Window en lugar de **Gtk.Window**. Un nombre completo sólo será necesario en caso de ambigüedad, por ejemplo entre **GLib.Object** y **Gtk.Object**.

El espacio de nombres **GLib** se importa de manera predeterminada. Imagine una línea **using GLib;** al principio de cada archivo Vala.

Todo lo que no se encuentre dentro de un espacio de nombres separado, quedará en el espacio de nombres anónimo global. Si se debe hacer referencia al espacio global explícitamente debido a una ambigüedad, puede hacerse usando el prefijo **global::**.

Los espacios de nombres puede estar anidados, bien anidando sus declaraciones o bien dando un nombre del tipo **NameSpace1.NameSpace2**.

Muchas otras definiciones pueden declararse dentro de un espacio de nombres siguiendo el mismo convenio, por ejemplo **class NameSpace1.Test ...** . Tenga en cuenta que cuando se hace esto el nombre final de la definición será aquel en el cual esté anidada la declaración más el espacio de nombre declarado en la definición.

4.16. Struct

```
1 struct StructName {
2         public int a;
3 }
```

El código presentado arriba define un tipo de **struct** (o estructura), es decir, un tipo de valor compuesto. Una estructura Vala puede contener métodos con ciertas limitaciones así como miembros privados, dado lo cual se requiere el modificador de acceso **public**.

```
1 struct Color {
2         public double red;
```

```
3         public double green;
4         public double blue;
5 }
```

De la siguiente manera se puede inicializar una estructura:

```
1 // without type inference
2 Color c1 = Color();
3 Color c2 = { 0.5, 0.5, 1.0 };
4 Color c3 = Color() {
5         red = 0.5,
6         green = 0.5,
7         blue = 1.0
8 };
9
10 // with type inference
11 var c4 = Color();
12 var c5 = Color() {
13         red = 0.5,
14         green = 0.5,
15         blue = 1.0
16 };
```

Las estructuras se almacenan en la pila («stack») y se copian al asignarlas.

4.17. Clases

```
1 class ClassName : SuperClassName, InterfaceName {
2 }
```

Este ejemplo define una clase, es decir un tipo por referencia. En contraposición a las estructuras, las instancias de las clases se almacenan en el montón («heap»). Hay mucha más sintaxis relacionada con las clases, que se presentará en profundidad en la sección sobre programación orientada a objetos.

4.18. Interfaces

```
1 interface InterfaceName : SuperInterfaceName {
2 }
```

El ejemplo de arriba define una interfaz, esto es, un tipo no instanciable. Para crear una instancia de una interfaz, primero deben implementarse sus métodos abstractos en una clase no abstracta. Las interfaces Vala son mucho más potentes que las interfaces de Java o C#. De hecho se pueden usar

como «mixins». Los detalles de las interfaces se describen en la sección sobre programación orientada a objetos.

4.19. Atributos del código

Los atributos del código indican al compilador de Vala detalles sobre cómo se supone que debe funcionar el código en la plataforma objetivo. Su sintaxis es **[AttributeName]** o **[AttributeName(param1 = value1, param2 = value2, ...)]**.

Se utilizan principalmente para enlaces entre lenguajes («bindings») en los achivos vapi, siendo **[CCode(...)]** el más empleado en estos casos. Otro ejemplo es el atributo **[DBus(...)]**, usado para exportar interfaces remotas vía D-Bus.

Capítulo 5

Programación Básica con Vala

Un programa simple “Hola Mundo":

```
1 void main(){
2         print("Hola␣Mundo!\n");
3 }
```

Para compilar el programa y ejecutarlo:

```
1 valac Hola.vala
2 ./Hola
```

Si el archivo binario debe tener un nombre diferente entonces:

```
1 valac Hola.vala -o Saludo
2 ./Saludo
```

5.1. Leyendo información del Usuario

```
1 void main () {
2         stdout.printf ("Introduce␣tu␣Nombre:␣");
3         string name = stdin.read_line ();
4         if (name != null) {
5                 stdout.printf ("Hola,␣%s!\n", name);
6         }
7 }
```

Vala proporciona los objetos ***stdin*** (entrada estándar), ***stdout***(Salida Estándar) y ***stderr*** (error estándar) para los tres flujos estándares. El método

printf toma la cadena formateada y un numero variable de argumentos como parámetros.

5.2. Matemáticas

Las funciones matemáticas están dentro de *namespace* Math.

```
1  void main () {
2          stdout.printf ("Introduce␣Radio␣del␣Circulo:␣");
3          double radius = double.parse (stdin.read_line ());
4          stdout.printf ("Circunferencia:␣%g\n", 2 * Math.PI *
               radius);
5          stdout.printf ("sin(pi/2)␣=␣%g\n", Math.sin (Math.PI
               / 2));
6
7          // Random numbers
8
9          stdout.printf ("Resultados␣de␣la␣loteria␣de␣Hoy:");
10         for (int i = 0; i < 6; i++) {
11                 stdout.printf ("␣%d", Random.int_range (1,
                       49));
12         }
13         stdout.printf ("\n");
14
15         stdout.printf ("Numeros␣aleatorios␣entre␣0␣y␣1:␣%g\n"
               , Random.next_double ());
16 }
```

La compilación de este programa necesita de las librerías matemáticas, para ello es necesario indicarle al compilador que enlace las librerías:

```
1 valac Mate.vala  -X -lm
2 ./Mate
```

5.3. Argumentos de linea de comando y códigos de Exit

```
1 int main (string[] args) {
2
3 // Imprimir el numero de argumentos
4 stdout.printf ("%d␣argumentos␣de␣la␣linea␣de␣comando:\n",
      args.length);
5
```

```
6  // Imprimir todos los argumentos
7  foreach (string arg in args) {
8          stdout.printf ("%s\n", arg);
9  }
10
11 // Exit code (0: success, 1: failure)
12 return 0;
13 }
```

La primera linea de argumentos (**args[0]**) siempre es el nombre del mismo programa.

5.4. Leyendo y escribiendo Archivos de Texto

Este es un programa muy básico para escribir archivos

```
1  void main () {
2          try {
3                  string filename = "data.txt";
4
5                  // Writing
6                  string content = "Hola, Mundo";
7                  FileUtils.set_contents (filename, content);
8
9                  // Reading
10                 string read;
11                 FileUtils.get_contents (filename, out read);
12
13                 stdout.printf("El contenido del archivo '%s'
                       es:\n%s\n", filename, read);
14         } catch (FileError e) {
15         stderr.printf ("%s\n", e.message);
16 }
17 }
```

5.5. Programación Orientada a Objetos

A pesar de que Vala no forza al desarrollador a trabajar con objetos, algunas de sus características no están disponibles de otra forma. De esta manera, es deseable programar con un estilo orientado a objetos la mayor parte del tiempo. Como con la mayoría de lenguajes actuales, para definir tipos de objeto propios debe definirse una clase.

La definición de una clase indica qué datos contiene un objeto de ese tipo, a qué otros tipos de objeto puede hacer referencia, y qué métodos pueden ejecutarse sobre este. La definición puede incluir el nombre de otra clase de la cual la presente será una subclase. Una instancia de una clase también puede ser una instancia de cualquiera de sus superclases, ya que hereda de esta todos sus métodos y datos, aunque puede no tener acceso a toda esta información. Una clase también puede implementar un número ilimitado de interfaces, que son conjuntos de definiciones de métodos que deben implementarse por la clase (una instancia de una clase es también una instancia de cada una de las interfaces implementadas por su clase o sus superclases).

Las clases en Vala también tienen miembros «static» (estáticos). Este modificador permite, tanto a los datos como a los métodos, definirse como pertenecientes a la clase como un todo, en lugar de a las instancias específicas de esta. Se puede acceder a este tipo de miembros incluso sin poseer una instancia de la clase.

5.6. Conceptos Básicos

Una clase simple podría definirse como sigue:

```
1  public class TestClass : GLib.Object {
2
3          /* Atributos */
4          public int first_data = 0;
5          private int second_data;
6
7          /* Constructor */
8          public TestClass() {
9                  this.second_data = 5;
10         }
11
12         /* Metodos */
13         public int method_1() {
14                 stdout.printf("Dato␣Privado:␣%d", this.
                       second_data);
15                 return this.second_data;
16         }
17 }
```

El código anterior define un nuevo tipo (registrado automáticamente por el sistema de tipos de la biblioteca gobject) que contiene tres miembros. Dos

de los miembros son datos, los enteros definidos en la parte superior, y un método llamado method_1, que devuelve un entero. La declaración de la clase indica que esta clase es una subclase de GLib.Object, y por tanto sus instancias son también Objects, y también contienen los miembros de ese tipo. El hecho de que esta clase descienda de Object también significa que se puede emplear algunas de las propiedades especiales de Vala para acceder a algunas de las características del tipo Object.

Esta clase se ha descrito como **public** (de manera predeterminada, las clases son **internal**). Lo que esto implica es que se puede hacer referencia a ésta directamente mediante código que se encuentre fuera de este archivo (si usted es un programador C que ha trabajado con Glib/Gobject, reconocerá esto como un equivalente a definir las interfaces de una clase en un archivo de cabecera separado que el código externo pueda incluir).

Los miembros también se describen como **public** o **private**. El miembro **first_data** es **public**, por tanto es visible directamente para cualquier usuario de la clase, y puede modificarse sin que la instancia que lo contiene tenga constancia de ello. El segundo dato es **private**, así que sólo se puede hacer referencia a él desde el código perteneciente a esta clase. Vala soporta cuatro tipos distintos de modificadores de acceso:

- **public**.- Sin restricciones de acceso
- **private**.- El acceso se limita al interior de la definición de la clase o estructura. Es el predeterminado si no se ha especificado ningún modificador de acceso
- **protected**.- El acceso está limitado a la definición de la clase y a cualquier clase que herede de la misma
- **internal**.- El acceso se limita a clases definidas dentro del mismo **package**

El constructor inicializa las nuevas instancias de una clase. Tiene el mismo nombre de la clase, puede o no requerir argumentos y se define sin tipo de retorno.

La última parte de esta clase es la definición de un método. Este método se llama **method_1**, y devuelve un entero. Dado que este método no es estático, se puede ejecutar únicamente sobre una instancia de esta clase, y por tanto puede acceder a miembros de dicha instancia. Puede hacer esto último

por medio de la referencia **this**, que siempre apunta a la instancia sobre la que el método se llama. Mientras no haya ambigüedad el identificador **this** puede omitirse si así se desea.

Esta nueva clase se puede utilizar de la siguiente manera:

```
1 TestClass t = new TestClass();
2 t.first_data = 5;
3 t.method_1();
```

El programa completo queda representado de la siguiente forma:

```
1  public class TestClass : GLib.Object {
2
3          /* Atributos */
4          public int first_data = 0;
5          private int second_data;
6
7          /* Constructor */
8          public TestClass() {
9                  this.second_data = 5;
10         }
11
12         /* Metodos */
13         public int method_1() {
14                 stdout.printf("Dato␣Privado:␣%d", this.
                       second_data);
15                 return this.second_data;
16         }
17 }
18 void main() {
19 TestClass t = new TestClass();
20 t.first_data = 5;
21 t.method_1();
22 }
```

5.7. Construcción

Vala soporta dos métodos de construcción ligeramente distintos: el estilo Java/C# al que se atenderá de momento, y el estilo **GObject** que se describirá en la última sección de este capítulo.

Vala no soporta la sobrecarga de constructores por los mismos motivos que no se permite la sobrecarga de métodos; esto significa que una clase no puede tener varios constructores con el mismo nombre. Sin embargo, no es

un problema ya que Vala soporta constructores con nombre. Si se quieren ofrecer varios constructores se les debe añadir distintas extensiones:

```
1  public class Button : Object {
2
3          public Button() {
4          }
5
6          public Button.with_label(string label) {
7          }
8
9          public Button.from_stock(string stock_id) {
10         }
11 }
```

La instanciación es análoga:

```
1 new Button();
2 new Button.with_label("Click␣me");
3 new Button.from_stock(Gtk.STOCK_OK);
```

Se pueden enlazar constructores por medio de **this()** o **this.extension_del_nombre()**:

```
1  public class Point : Object {
2          public double x;
3          public double y;
4
5          public Point(double x, double y) {
6                  this.x = x;
7                  this.y = y;
8          }
9
10         public Point.rectangular(double x, double y) {
11                 this(x, y);
12         }
13
14         public Point.polar(double radius, double angle) {
15                 this.rectangular(radius * Math.cos(angle),
                      radius * Math.sin(angle));
16         }
17 }
18
19 void main() {
20         var p1 = new Point.rectangular(5.7, 1.2);
21         var p2 = new Point.polar(5.7, 1.2);
22 }
```

5.8. Destrucción

A pesar de que Vala gestiona la memoria automáticamente, puede ser que se necesite añadir destructores propios si se elige gestionar manualmente la memoria con punteros o si deben liberarse otros recursos. La sintaxis es la misma que en C# y C++:

```
1 class Demo : Object {
2         ~Demo() {
3                 stdout.printf("in destructor");
4         }
5 }
```

Dado que la gestión de memoria de Vala se basa en conteo de referencias en lugar de recolección de basura los destructores son deterministas y pueden emplearse para implementar el patrón RAII para la gestión de recursos (cerrar flujos, conexiones a bases de datos, etc.).

5.9. Señales

Las señales son un sistema proporcionado por la clase Object de GLib, y accesible fácilmente gracias a Vala desde todos los descendientes de Object. Una señal se puede ver como un evento para los programadores de C# o, para los programadores de Java, como una implementación alternativa de los **«event listeners»**. En resumen, una señal es simplemente una forma de ejecutar un número arbitrario de métodos idénticos desde el exterior (es decir, con el mismo prototipo) aproximadamente en el mismo instante. Los mecanismos reales de ejecución son internos a GObject, y no son importantes para los programas Vala.

Una señal se define como un miembro de una clase, y se parece a un método sin cuerpo. Los manejadores de señales pueden añadirse a la señal usando el método **connect()**. Con el fin de profundizar en el asunto, el siguiente ejemplo también presenta **las expresiones lambda**, una forma muy práctica de escribir manejadores de señales en Vala:

```
1 public class Test : GLib.Object {
2
3         public signal void sig_1(int a);
4
5         public static int main(string[] args) {
6                 Test t1 = new Test();
```

```
7
8                   t1.sig_1.connect((t, a) => {
9                           stdout.printf("%d\n", a);
10                  });
11
12                  t1.sig_1(5);
13
14                  return 0;
15          }
16  }
```

En el código anterior se crea una nueva clase llamada "Test", usando la sintaxis ya familiar. El primer miembro de la clase es una señal llamada "sig_1" que, según su definición, pasa un entero. En el método **main** de este programa, primero se crea una instancia de Test (un requisito, dado que las señales siempre pertenecen a instancias de las clases). A continuación, se conecta a la señal "sig_1" de la instancia creada un manejador, definido en línea como una expresión lambda. La definición indica que el método recibirá dos argumentos, llamados "t" y "a", pero no da sus tipos. Se puede hacer esto por que Vala ya conoce la definición de la señal, y por tanto sabe qué tipos se requieren.

El motivo por el que se pasan dos parámetros al manejador es por que cuando una señal se emite, el objeto sobre el cual se emite se pasa como primer argumento al manejador. El segundo argumento es el que pasa la señal.

Finalmente se emite la señal. Esto se hace llamándola como si se tratase de un método de la clase, y GObject se encarga de enviar el mensaje a todos los métodos manejadores conectados a la señal. Entender los mecanismos subyacentes mediante los que se hace esto no es un necesario para usar las señales desde Vala.

Actualmente el modificador de acceso **public** es la única opción posible (todas las señales se pueden conectar y emitir desde cualquier código). Las señales se pueden anotar con una combinación de **«flags»**:

```
1 [Signal (action=true, detailed=true, run=true, no_recurse=
     true, no_hooks=true)]
2 public signal void sig_1 ();
```

Las señales son una forma conveniente para que los objetos se informen entre sí acerca de eventos. Esta técnica es especialmente útil para la programación GUI. Por ejemplo, un botón puede informar a otros objetos a través de la señal *clicked* que el usuario a pulsado sobre el botón.

Se definen las señales en una clase y las aplicaciones interesadas registran sus funciones o métodos a estas señales de una instancia. La instancia puede emitir las señal en la forma de la llamada a un método cada función de llamada (tambien nombrado **Manejador**) conectado a las señal será llamado. Como se muestra en el siguiente código

```
1  class Foo : Object {
2          public signal void some_event ();    // definition of
               the signal
3
4          public void method () {
5                  some_event ();                    // emitting
                      the signal (callbacks get invoked)
6          }
7  }
8
9  void callback_a () {
10         stdout.printf ("Callback␣A\n");
11 }
12
13 void callback_b () {
14         stdout.printf ("Callback␣B\n");
15 }
16
17 void main () {
18         var foo = new Foo ();
19         foo.some_event.connect (callback_a);       //
               connecting the callback functions
20         foo.some_event.connect (callback_b);
21         foo.method ();
22 }
```

Para compilar el programa anterior se tiene que ejecutar:

```
1 $ valac signals.vala
2 $ ./signals
```

Es importante puntualizar que una clase con señales siempre tiene que heredar de la clase Objeto o de alguna sub-clase. Las funciones de llamada (callbacks) se conectan a las señales llamando *.connect()* en la señal.

Una señal puede tener múltiples parámetros. Las firmas de las funciones de llamada tienen que coincidir con una de las señales, excepto que se pueden dejar parámetros en la firma tantos como se desee o proporcionar una fuente de señal adicional como primer parámetro de la función de llamada. Por ejemplo

```
public class Foo : Object {
public signal void some_event (int x, int y, double z);
// ...
}
```

Las siguientes firmas de funciones de llamada coinciden con la señal

```
void on_some_event ()
void on_some_event (int x)
void on_some_event (int x, int y)
void on_some_event (int x, int y, double z)
void on_some_event (Foo source, int x, int y, double z)
```

Los nombres de los parámetros y de las funciones de llamada pueden escogerse libremente. El objeto fuente puede ayudar a distinguir en el caso de que se conecte una función de llamada a diferentes instancias del mismo tipo.

Con la siguiente sintaxis se pueden conectar señales a funciones anónimas (lambdas):

```
foo.some_event.connect ((source, x, y, z) => {
        stdout.printf ("%d␣%d␣%g\n", x, y, z);
});
```

Se puede notar que no hay declaración de tipos en la lista de parámetros de una función anónima. El compilador automáticamente infiere los tipos de datos desde la definición de las señales.

5.9.1. Desconectando Señales

Se pueden desconectar señales de las funciones de llamada de dos maneras. La primera es simplemente llamando a

```
myobject.mysignal.disconnect(callback)
```

La forma más avanzada es almacenar el valor de retorno del método *connect()* en algún lugar. Esto es un tipo **ulong** conteniendo el id del manejador de señales. Pasar este id del manejador de señal a *myobjet.desconnect()* - es importante notar que se esta invocando *disconnect()* en un objeto y no en la señal.

```
foo.some_event.connect (on_some_event);
foo.some_event.disconnect (on_some_event);

ulong handler_id = foo.some_event.connect (() => { /* ... */
   });
```

```
5 foo.disconnect (handler_id);
```

5.9.2. Manejadores de señales por default y connect_after()

Una señal declarada virtual puede tener una implementación de un manejador de señal por default:

```
1 class Demo : Object {
2         public virtual signal void sig () {
3                 stdout.printf ("default␣handler\n");
4         }
5 }
```

Los manejadores de señal van conectados antes al manejador de señales por default. Si se desea conectar una señal después, el manejador de default usa *connect_after()*.

```
1 void main () {
2         var demo = new Demo ();
3         demo.sig.connect (() => stdout.printf ("before\n"));
4         demo.sig.connect_after (() => stdout.printf ("after\n
              "));
5         demo.sig (); // emit signal
6 }
```

la salida es

```
1 before
2 default handler
3 after
```

El manejador por default lo pueden sobreescribir por una subclase

```
1 class Sub : Demo {
2         public override void sig () {
3                 stdout.printf ("overridden␣default␣handler\n"
                      );
4         }
5 }
```

5.9.3. Acerca de *user_data*

Vala usa los argumentos de *user_data* implicitamente en el código C generado, o para el contexto de las clausuras o por la referencia a la instancia (this). Asi que se puede acceder o por los datos de salida dentro de la clausula o por una variable de instancia del manejador de clase.

```
1  class Foo {
2          public signal void sig ();
3  }
4
5  class Bar {
6          private int data = 42;
7
8          public void handler () {
9                  stdout.printf ("Data␣via␣instance:␣%d\n",
                       this.data);
10         }
11 }
12
13 void main () {
14         var foo = new Foo ();
15
16         int data = 42;
17         foo.sig.connect (() => {         // 'user_data' in C
               code = variables from outer context
18                 stdout.printf ("Data␣via␣closure:␣%d\n", data
                       );
19         });
20
21         var bar = new Bar ();
22         foo.sig.connect (bar.handler);  // 'user_data' in C
               code = 'bar'
23
24         // Emit signal
25         foo.sig ();
26 }
```

5.10. Propiedades

Una buena práctica en la programación orientada a objetos es ocultar los detalles de la implementación de las clases creadas (principio de ocultación de información (enlace en inglés)), esto da la posibilidad de modificar su interior sin romper la API publica. Una forma de llevarlo a la práctica es hacer los campos privados y proporcionar métodos de acceso para obtener y modificar sus valores («getters» y «setters»).

Según la programación en Java esto se podría hacer así:

```
1  class Person : Object {
2          private int age = 32;
```

```
3
4           public int get_age() {
5                   return this.age;
6           }
7
8           public void set_age(int age) {
9                   this.age = age;
10          }
11  }
```

Esto funcionaría, pero Vala puede hacerlo mejor. El problema es que trabajar con estos métodos puede resultar incómodo. Por ejemplo si se quisiera incrementar la edad de una persona un año:

```
1  var alice = new Person();
2  alice.set_age(alice.get_age() + 1);
```

Aquí es donde las propiedades entran en juego:

```
1  class Person : Object {
2           private int _age = 32;  // underscore prefix to avoid
                name clash with property
3
4           /* Property */
5           public int age {
6                   get { return _age; }
7                   set { _age = value; }
8           }
9  }
```

Esta sintaxis debería ser familiar para los programadores de C#. Una propiedad tiene bloques **get** y **set** para obtener y modificar su valor. **value** es una palabra reservada que representa el nuevo valor que se asignará a la propiedad.

Así se puede acceder a la propiedad como si se tratase de un campo público. Pero por detrás se estará ejecutando el código de los bloques **get** y **set**.

```
1  var alice = new Person();
2  alice.age = alice.age + 1;  // or even shorter:
3  alice.age++;
```

Si sólo se realiza la implementación estándar, cómo se ha visto antes, se puede escribir la propiedad incluso más brevemente:

```
1  class Person : Object {
```

```
2         /* Property with standard getter and setter and
             default value */
3         public int age { get; set; default = 32; }
4 }
```

Con las propiedades se puede modificar el trabajo interno de las clases sin modificar la API pública. Por ejemplo:

```
1 static int current_year = 2525;
2
3 class Person : Object {
4         private int year_of_birth = 2493;
5
6         public int age {
7                 get { return current_year - year_of_birth; }
8                 set { year_of_birth = current_year - value; }
9         }
10 }
```

Esta vez la edad se calcula al vuelo a partir del año de nacimiento (**year_of_birth**). Como se puede observar, se puede realizar algo más que un simple acceso a una variable o una asignación dentro de los bloques get y set. Se podrían hacer incluso accesos a bases de datos, registros de actividad, actualizaciones de caché, etc.

Si se quiere hacer que una propiedad sea de sólo lectura para los usuarios de una clase, se debe declarar el **setter** como privado:

```
1 public int age { get; private set; default = 32; }
```

O, alternativamente, se puede quitar el bloque set:

```
1 class Person : Object {
2         private int _age = 32;
3
4         public int age {
5                 get { return _age; }
6         }
7 }
```

Las propiedades pueden tener, además de su nombre, una descripción breve (llamada **nick**) y una descripción (llamada **blurb**). Estas pueden anotarse con un atributo especial:

```
1 [Description(nick = "age␣in␣years", blurb = "This␣is␣the␣
     person's␣age␣in␣years")]
2 public int age { get; set; default = 32; }
```

Las propiedades y sus descripciones adicionales pueden consultarse en tiempo de ejecución. Algunos programas, como Glade, el diseñador de interfaces gráficas de usuario, pueden hacer uso de esta información. En este sentido, Glade puede presentar descripciones legibles para los widgets GTK+.

Cada instancia de una clase derivada de GLib.Object tiene una señal llamada **notify**. Esta señal se emite cada vez que una propiedad de dicho objeto cambia. Se puede, por tanto, conectar esta señal si se necesita obtener notificaciones de cambios en general:

```
1 obj.notify.connect((s, p) => {
2         stdout.printf("Property␣'%s'␣has␣changed!\n", p.name)
              ;
3 });
```

s es la instancia a la que pertenece la señal (obj en este ejemplo), p es la información de la propiedad, del tipo ParamSpec, para la propiedad modificada. Si sólo interesa obtener notificaciones de las modificaciones de una propiedad, se puede emplear esta sintaxis:

```
1 alice.notify["age"].connect((s, p) => {
2         stdout.printf("age␣has␣changed\n");
3 });
```

Se observa que en este caso se debe usar la representación como cadena de caracteres del nombre propiedad, donde los subrayados se sustituyen por guiones: nombre_de_propiedad pasará a ser nombre-de-propiedad en dicha representación, que sigue el convenio de nombres de GObject.

Las notificaciones de modificación se pueden desactivar con una etiqueta del atributo CCode situada inmediatamente antes de la declaración de la propiedad:

```
1 public class MyObject : Object {
2         [CCode(notify = false)]
3         // notify signal is NOT emitted upon changes in the
              property
4         public int without_notification { get; set; }
5         // notify signal is emitted upon changes in the
              property
6         public int with_notification { get; set; }
7 }
```

Existe otro tipo de propiedades llamadas propiedades de construcción que se describen más adelante, en la sección sobre la construcción al estilo **GObject**.

En caso de que la propiedad sea del tipo **struct**, para que el valor de la propiedad pueda cogerse con **Object.get()**, se debe declarar la variable como en el siguiente ejemplo:

```
1  struct Color
2  {
3          public uint32 argb;
4
5          public Color() { argb = 0x12345678; }
6  }
7
8  class Shape: GLib.Object
9  {
10         public Color c { get; set; default = Color(); }
11 }
12
13 int main()
14 {
15         Color? c = null;
16         Shape s = new Shape();
17         s.get("c", out c);
18 }
```

De esta manera, c es una referencia en lugar de una instancia de Color en la pila. Lo que se pasa en s.get() es Color ** en lugar de Color *.

5.11. Herencia

En Vala, una clase puede derivarse de una o de ninguna clase. En la práctica suele ser una pero, a pesar de ello, no hay herencia implícita como la hay en otros lenguajes, como Java.

Cuando se define una clase que hereda de otra, se crea una relación entre las clases donde las instancias de la subclase son también instancias de la superclase. Esto significa que las operaciones sobre instancias de la superclase son también aplicables a las instancias de la subclase. Así, en cualquier lugar en que se requiera una instancia de la superclase, se puede proporcionar una instancia de la subclase.

Al escribir la definición de una clase, es posible tener control preciso sobre quién puede acceder a los métodos y los datos del objeto. El siguiente ejemplo muestra algunas de estas opciones:

```
1  class SuperClass : GLib.Object {
```

```
2
3           private int data;
4
5           public SuperClass(int data) {
6                   this.data = data;
7           }
8
9           protected void protected_method() {
10          }
11
12          public static void public_static_method() {
13          }
14  }
15
16  class SubClass : SuperClass {
17
18          public SubClass() {
19                  base(10);
20          }
21  }
```

data es miembro de las instancias de **SuperClass**. Habrá un miembro de este tipo en cada instancia de **SuperClass** y, al estar declarado como privado, sólo es accesible desde el código que forma parte de **SuperClass**.

protected_method es una método de las instancias de **SuperClass**. Este método sólo se puede ejecutar sobre las instancias de **SuperClass** o alguna de sus subclases, y únicamente desde el código que pertenece a **SuperClass** o alguna de sus subclases (esta última regla es resultado del modificador **protected**).

public_static_method tiene dos modificadores. El modificador static significa que este método puede llamarse sin poseer una instancia de **SuperClase** o de alguna de sus subclases. Como resultado, este método no tiene acceso a la referencia **this** cuando se ejecuta. El modificador **public** indica que el método se puede llamar desde cualquier fragmento de código, independientemente de su relación con **SuperClass** o sus subclases.

Dadas estas definiciones, una instancia de **SubClass** contendrá los tres miembros de **SuperClass**, pero sólo podrá acceder a los miembros no privados. El código externo sólo podrá acceder al método público.

Con la referencia base, el constructor de una subclase puede enlazar con el constructor de su clase base.

5.12. Clases abstractas

Existe otro modificador para los métodos, llamado **abstract**. Este modificador permite describir un método que no se implementa realmente en la clase. En lugar de ello, el método debe implementarse en las subclases antes de poderse llamar. Esto permite que se definan operaciones a las que se pueda acceder desde todas las instancias de un tipo, asegurando a la vez que cada tipo específico proporcione su propia versión de la funcionalidad.

Una clase que contenga un método abstracto debe declararse a su vez como **abstract**. El resultado de esto es prevenir cualquier instanciación de este tipo.

```
1  public abstract class Animal : Object {
2
3          public void eat() {
4                  stdout.printf("*chomp␣chomp*\n");
5          }
6
7          public abstract void say_hello();
8  }
9
10 public class Tiger : Animal {
11
12         public override void say_hello() {
13                 stdout.printf("*roar*\n");
14         }
15 }
16
17 public class Duck : Animal {
18
19         public override void say_hello() {
20                 stdout.printf("*quack*\n");
21         }
22 }
```

La implementación de un método abstracto debe marcarse con override. Las propiedades también pueden ser abstractas.

5.13. Interfaces / Mixins

Una clase en Vala puede implementar un número arbitrario de interfaces. Cada interfaz es un tipo, como una clase, pero no se puede instanciar. Imple-

mentando una o más interfaces, una clase puede declarar que sus instancias son también instancias de la interfaz, y por tanto podrán usarse en cualquier situación en la que se espere una instancia de dicha interfaz.

El proceso para implementar una interfaz es el mismo que para heredar de clases abstractas con métodos, para que la clase se pueda usar, debe proporcionar implementaciones para todos los métodos descritos pero no implementados.

A continuación se presenta una definición simple de una interfaz:

```
1 public interface ITest : GLib.Object {
2         public abstract int data_1 { get; set; }
3         public abstract void method_1();
4 }
```

Este código describe una interfaz "ITest" que requiere GLib.Object como padre y contiene dos miembros. "data_1" es una propiedad, como las descritas anteriormente, excepto que aquí se ha declarado como abstract. Vala no implementará esta propiedad, sino que requerirá que las clases que implementen la interfaz tengan una propiedad llamada "data_1" que tenga los métodos de acceso get y set (es obligatorio que se defina como abstracta, ya que las interfaces no pueden tener datos como miembros propios). El segundo miembro "method_1" es un método. Aquí se declara que este método debe ser implementado por las clases que implementen "ITest".

La implementación completa más sencilla de esta interfaz es:

```
1 public class Test1 : GLib.Object, ITest {
2         public int data_1 { get; set; }
3         public void method_1() {
4         }
5 }
```

Y se puede usar de la siguiente forma:

```
1 var t = new Test1();
2 t.method_1();
3
4 ITest i = t;
5 i.method_1();
```

Las interfaces en Vala no pueden heredar de otras interfaces, pero pueden declararlas como prerrequisitos, esta técnica es más o menos equivalente. Por ejemplo, se podría querer que cada clase que implemente una interfaz "List" debe implementar también una interfaz "Collection". La sintaxis para hacerlo

es exactamente la misma que para describir la implementación de una interfaz en las clases:

```
1 public interface List : Collection {
2 }
```

Esta definición de "List" no se puede implementar en una clase sin que se implemente también "Collection", y por tanto Vala fuerza un estilo de declaración para las clases que quieran implementar "List", en el que todas las interfaces deben describirse:

```
1 public class ListClass : GLib.Object, Collection, List {
2 }
```

Las interfaces Vala también pueden tener una clase como prerrequisito. Si se da el nombre de una clase en la lista de prerrequisitos, la interfaz sólo se podrá implementar en clases que deriven de la clase listada. Esto se suele usar para asegurar que una instancia de una interfaz es también una subclase de GLib.Object, y por tanto que la interfaz se pueda usar, por ejemplo, como el tipo de una propiedad.

El hecho de que las interfaces no puedan heredar de otras interfaces es sobretodo una distinción técnica (en la práctica, Vala funciona como otros lenguajes en este aspecto, pero con la característica adicional de las clases prerrequisito).

Existe otra diferencia importante entre las interfaces Vala y las de Java/C#: las de Vala pueden tener métodos no abstractos. Realmente, Vala permite implementaciones de métodos en las interfaces, de aquí que se requiera que los métodos abstractos sean declarados como abstract. Debido a este hecho, las interfaces de Vala pueden funcionar como «mixins». Es una forma restringida de herencia múltiple.

5.14. Polimorfismo

El término polimorfismo describe la manera en la que un mismo objeto puede usarse como si fuera de más de un tipo distinto. Varias de las técnicas que ya se han descrito aquí sugieren cómo es posible hacer esto en Vala: una instancia de una clase puede usarse como una instancia de una superclase, o de cualquier interfaz que implemente, sin necesidad de conocer cual es realmente su tipo.

Una extension lógica de esta capacidad es permitir a un subtipo comportarse de forma distinta a su tipo padre cuando se referencia de la misma

manera. Dado que este concepto es difícil de explicar, se presenta un ejemplo de lo que pasaría si no se pretende emplear esta propiedad directamente:

```
1  class SuperClass : GLib.Object {
2          public void method_1() {
3                  stdout.printf("SuperClass.method_1()\n");
4          }
5  }
6
7  class SubClass : SuperClass {
8          public void method_1() {
9                  stdout.printf("SubClass.method_1()\n");
10         }
11 }
```

Estas dos clases implementan un método llamado "method_1", y por tanto "SubClass" tiene dos métodos llamados "method_1", ya que hereda uno de "SuperClass". Cada uno de estos métodos puede llamarse de la siguiente manera:

```
1 SubClass o1 = new SubClass();
2 o1.method_1();
3 SuperClass o2 = o1;
4 o2.method_1();
```

En el código anterior se llama realmente a dos métodos distintos. En la segunda línea se toma a "o1" por un objeto del tipo "SubClass" y se llama su versión del método. En la cuarta línea se interpreta que "o2" es un objeto del tipo "SuperClass" y se llama a la versión del método correspondiente a dicha clase.

El problema que expone este ejemplo, es que cualquier código que referencia a "SuperClass" llamará a métodos descritos en esta clase, incluso si el tipo real del objeto es "SubClass". La técnica empleada para cambiar este comportamiento es el uso de métodos virtuales. Usándola, el ejemplo anterior se reescribiría como:

```
1  class SuperClass : GLib.Object {
2          public virtual void method_1() {
3                  stdout.printf("SuperClass.method_1()\n");
4          }
5  }
6
7  class SubClass : SuperClass {
8          public override void method_1() {
9                  stdout.printf("SubClass.method_1()\n");
```

```
10          }
11  }
```

Cuando este código se usa de la misma manera que antes, el "method_1" de "SubClass" se llamará dos veces. Esto es así porque se le dice al sistema que "method_1" es un método virtual, es decir que si se sobreescribe en una subclase, la nueva versión será la que se ejecute sobre las instancias de esa subclase, sin importar lo que se sepa en el momento de la llamada.

Probablemente, esta distinción sea familiar a los programadores de algunos lenguajes, tales como C++, pero es el comportamiento contrario al de lenguajes del estilo de Java, en los que se deben seguir ciertos pasos para evitar que un método sea virtual.

Puede que en este punto el lector haya reconocido que cuando un método se declara como abstract también debe ser virtual. De no ser así no sería posible ejecutar este método dada una instancia que aparentemente es del tipo en el que se ha declarado. Cuando se implementa un método abstracto en una subclase, se debe elegir si se declara la implementación como override (sobreescritura), aclarando así la naturaleza virtual del método y permitiendo que los subtipos hagan mismo.

También es posible implementar los métodos de una interfaz de manera que las subclases puedan cambiar su implementación. El proceso en este caso consiste en declarar la implementación inicial del método como virtual, y entonces las subclases pueden sobreescribirlo.

Cuando se escribe una clase, es habitual querer usar funcionalidad heredada de una clase padre. Esto se complica cuando el nombre del método se usa más de una vez en el árbol de herencias de la clase que se está escribiendo. Para estos casos Vala proporciona la palabra reservada base. El caso más común de su uso se da cuando se ha sobreescrito un método para obtener alguna funcionalidad adicional, pero se sigue necesitando llamar al método de la clase padre. El siguiente ejemplo muestra este caso:

```
1 public override void method_name() {
2         base.method_name();
3         extra_task();
4 }
```

Finalmente, Vala también permite que las propiedades sean virtuales:

```
1 class SuperClass : GLib.Object {
2         public virtual string prop_1 {
3                 get {
4                         return "SuperClass.prop_1";
```

```
5                   }
6           }
7   }
8
9   class SubClass : SuperClass {
10          public override string prop_1 {
11                  get {
12                          return "SubClass.prop_1";
13                  }
14          }
```

5.15. Ocultación de métodos

Usando el modificador new se puede ocultar un método heredado con un nuevo método con el mismo nombre. El nuevo método puede tener un prototipo distinto. La ocultación de métodos no debe confundirse con la sobreescritura, ya que la ocultación de métodos no tiene un comportamiento polimórfico.

```
1   class Foo : Object {
2           public void my_method() { }
3   }
4
5   class Bar : Foo {
6           public new void my_method() { }
7   }
```

Se puede seguir llamando al método original mediante la promoción a la clase base o interfaz de la que se ha heredado.

```
1   void main() {
2           var bar = new Bar();
3           bar.my_method();
4           (bar as Foo).my_method();
5   }
```

5.16. Información de tipos en tiempo de ejecución

Dado que las clases Vala se registran en tiempo de ejecución y cada instancia lleva consigo la información de su tipo, se puede comprobar el tipo de

un objeto empleando el operador is:

```
1 bool b = object is SomeTypeName;
```

También se puede obtener la información del tipo de instancias de **Object** con el método **get_type()**:

```
1 Type type = object.get_type();
2 stdout.printf("\%s␣n", type.name());
```

Con el operador **typeof()** se puede obtener información de un tipo directamente. A partir de esta información de tipo, se pueden crear nuevas instancias escribiendo **Object.new()**:

```
1 Type type = typeof(Foo);
2 Foo foo = (Foo) Object.new(type);
```

El constructor al que se llamará con esta acción es el bloque **construct** , que se describirá en la sección sobre construcción con el estilo de **GObject**.

5.17. Promoción de tipos dinámicos

Para la promoción dinámica de una variable a **TipoObjetivo**, se hace uso de la expresión as **TipoObjetivo** tras el nombre de la variable que se quiere promocionar. Vala realizará una comprobación para asegurar que la promoción es razonable (si la promoción no es posible, se devolverá **null**). Sin embargo, esto requiere que tanto el tipo fuente como el tipo objetivo sean tipos referenciados.

Por ejemplo,

```
1 Button b = widget as Button;
```

Si por algún motivo la clase de la instancia widget no es Button o una de sus subclases o no implementa la interfaz Button, b será null. Esta forma de promoción es equivalente a:

```
1 Button b = (widget is Button) ? (Button) widget : null;
```

5.18. Genericidad

Vala incluye un sistema de genericidad en tiempo de ejecución por medio del cual una instancia concreta de una clase se puede restringir a un tipo concreto o un conjunto de tipos, escogidos en el momento de su construcción.

Esta restricción se usa generalmente para requerir que los datos almacenados en el objeto deban ser de un tipo concreto, por ejemplo, para implementar una lista de objetos de cierto tipo. En este ejemplo, Vala se asegurará de que solo objetos del tipo requerido se puedan añadir a la lista, y que al recuperarlos todos los objetos se promocionen a dicho tipo.

En Vala, la genericidad se maneja durante la ejecución. Cuando se define una clase que se puede restringir a un tipo, sigue existiendo sólo una clase, con cada instancia personalizada individualmente. En constraste con C++, donde se crea una nueva clase por cada restricción de tipo existente (el sistema de Vala, es similar al empleado por Java). Esto tiene varias consecuencias, la más importante es que los miembros estáticos se comparten por el tipo como un todo, sean cuales sean las restricciones para cada instancia. Otra consecuencia es que dada una clase y una subclase, un tipo genérico refinado por la subclase se puede usar como un tipo genérico refinado por la clase.

El siguiente fragmento de código muestra como se usa el sistema de genericidad para definir una clase envoltorio mínima:

```
1  public class Wrapper<G> : GLib.Object {
2          private G data;
3
4          public void set_data(G data) {
5                  this.data = data;
6          }
7
8          public G get_data() {
9                  return this.data;
10         }
11 }
```

Esta clase Wrapper debe restringirse con un tipo para que se pueda instanciar (en este caso el tipo se identifica como G, y por tanto las instancias de esta clase contendrán un objeto del tipo G, y tendrán métodos para modificar ese objeto). (La razón de este ejemplo en concreto es proporcionar una explicación razonada de que actualmente una clase generica no puede hacer uso de las propiedades del tipo al que está restringida y por ello, en lugar de acceder directamente, esta clase tiene métodos set y get simples.)

Para instanciar esta clase, se debe escoger un tipo, por ejemplo el tipo string proporcionado por el lenguaje (en Vala no hay restricción sobre los tipos usados para la genericidad). A continuación se muestra un breve ejemplo del uso de la clase anteriormente definida:

```
1  var wrapper = new Wrapper<string>();
```

```
wrapper.set_data("test");
var data = wrapper.get_data();
```

Como se puede observar, cuando los datos se recuperan del envoltorio, se asignan a un identificador sin tipo explícito. Esto es posible porque Vala sabe que tipo de objetos se almacenan en cada instancia de Wrapper.

El hecho de que Vala no cree varias clases a partir de la definición generica, significa que se puede escribir lo siguiente:

```
class TestClass : GLib.Object {
}

void accept_object_wrapper(Wrapper<Glib.Object> w) {
}

...
var test_wrapper = new Wrapper<TestClass>();
accept_object_wrapper(test_wrapper);
...
```

Dado que todas las instancias de "TestClass"son también Object, el método accept_object_wrapper aceptará sin problemas el objeto que se le pasa, y lo tratará como el objeto al que envuelve como si fuera una instancia de GLib.Object.

5.19. Construcción al estilo de GObject

Como se ha dicho en el punto anterior, Vala soporta un esquema de construcción alternativo que es ligeramente diferente al descrito anteriormente, pero parecido a la manera en que funciona la construcción en GObject. El método preferible depende de si el usuario viene del lado de GObject o del lado de Java o C#. El esquema de construcción estilo GObject introduce algunos elementos sintácticos nuevos: propiedades establecidas en la construcción (construct properties), la llamada especial Object(...) y un bloque construct. Echemos un vistazo a cómo funciona esto:

```
public class Person : Object {

        /* Construction properties */
        public string name { get; construct; }
        public int age { get; construct set; }

        public Person(string name) {
```

```
8                   Object(name: name);
9           }
10
11          public Person.with_age(string name, int years) {
12                  Object(name: name, age: years);
13          }
14
15          construct {
16                  // do anything else
17                  stdout.printf("Welcome %s\n", this.name);
18          }
19  }
```

Con el esquema de construcción estilo GObject, cada método de construcción solo contiene una llamada Object(...) para asignar valores a las propiedades establecidas en la construcción. La llamada Object(...) toma un numero variable de argumentos de la forma propiedad: valor. Estas propiedades se deben declarar como propiedades construct. Se ajustarán a los valores indicados y luego se llamará a todos los bloques construct en la jerarquía de objetos, desde GLib.Object hasta el de la clase creada.

Se garantiza que el bloque construct se llama cuando se crea una instancia de esta clase, incluso si se crea como un subtipo. Sin tener ningún parámetro, ni un valor de retorno. Dentro de este bloque se puede llamar a otros métodos y establecer las variables miembro cuando sea necesario.

Las propiedades de construcción se definen como get y set, y por lo tanto se puede ejecutar código arbitrario en su asignación. Si se necesita que la inicialización esté basada en una única propiedad de construcción, es posible escribir un bloque construct personalizado para esta propiedad, que se ejecutará inmediatamente en su asignación, y antes de cualquier código de construcción.

Si una propiedad de construcción se declara sin set se denomina propiedad construct only, lo que significa que sólo se puede asignar en la construcción, pero no después. En el ejemplo anterior name es una propiedad de este tipo.

A continuación se muestra un resumen de los diferentes tipos de propiedades junto a la nomenclatura que se suele encontrar en la documentación de librerías basadas en GObject:

```
1 public int a { get; private set; }     // Read
2 public int b { private get; set; }     // Write
3 public int c { get; set; }             // Read / Write
4 public int d { get; set construct; }   // Read / Write /
    Construct
```

```
5 public int e { get; construct; }      // Read / Write-
    Construct-Only
```

En algunos casos es posible que también se quiera realizar algún tipo de acción, no cuando las instancias de una clase se crean, sino cuando el sistema GObject crea la propia en si. En la terminología de GObject estamos hablando de un fragmento de código que se ejecuta dentro de la función class_init para la clase en cuestión. En Java esto se conoce como bloques inicializadores estáticos. En Vala es así:

```
1 /* This snippet of code is run when the class
2 * is registered with the type system */
3 static construct {
4         ...
5 }
```

5.20. Primeras clases

```
1  /* class derivada de GObject */
2  public class SimpleBasica : Object {
3
4          /* instancia del Metodo Publico */
5          public void run () {
6                  stdout.printf ("Hola␣Mundo\n");
7          }
8  }
9
10 /* Punto de entrada del Metodo */
11 int main (string[] args) {
12         // Instanciar esta clase, asianando la instancia a
13         // una variable
14         var sample = new SimpleBasica ();
15         // llamar al metodo run
16         sample.run ();
17         // regresar del metodo principal
18         return 0;
19 }
```

El punto de entrada puede tambien estar dentro de la clase, como se muestra a continuación:

```
1 public class SimpleBasica : Object {
2
3         public void run () {
```

```
4                   stdout.printf ("Hola␣Mundo\n");
5           }
6
7           static int main (string[] args) {
8                   var sample = new SimpleBasica ();
9                   sample.run ();
10                  return 0;
11          }
12  }
```

En este caso el método **main** debe ser declarado ***static***

Capítulo 6

Manejo de Archivos con Vala

GIO es similar al ambiente de trabajo de Java IO. Las Referencias a archivos y directorios son representados como objetos ***File*** los cuales puedes crear desde el nombre de un archivo o un URI. El GIO proporciona dos clases base para entrada y salida: *InputStream* y *OutputStream*. Abrir un archivo para leer su contenido resultará en un *FileInputStream* el cual es un subclase de *InputStream*.

Los *Streams* son formateados de acuerdo con un patrón de decoración para proporcionales una funcionalidad adicional. Por ejemplo, es posible leer el contenido de un archivo linea por linea decorando sus *FileinputStream* con un *DataInputStream*. O puedes aplicar un *FileInputStream* para filtrados de propósitos particulares.

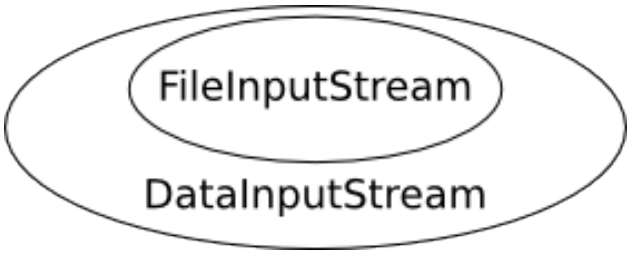

Figura 6.1: Input Stream

6.1. Leyendo archivos de texto linea por linea

El siguiente ejemplo lee el contenido de un archivo de texto linea por linea y lo imprime en la pantalla:

```
1  int main () {
2          // A reference to our file
3          var file = File.new_for_path ("data.txt");
4
5          if (!file.query_exists ()) {
6                  stderr.printf ("File␣'%s'␣doesn't␣exist.\n",
                       file.get_path ());
7                  return 1;
8          }
9
10         try {
11                 // Open file for reading and wrap returned
                       FileInputStream into a
12                 // DataInputStream, so we can read line by
                       line
13                 var dis = new DataInputStream (file.read ());
14                 string line;
15                 // Read lines until end of file (null) is
                       reached
16                 while ((line = dis.read_line (null)) != null)
                       {
17                     stdout.printf ("%s\n", line);
18                 }
19         } catch (Error e) {
20                 error ("%s", e.message);
21         }
22
23         return 0;
24 }
```

Para compilar y ejecutar este programa se ejecuta en una terminal:

```
1 $ valac --pkg gio-2.0 gio-sample.vala
2 $ ./gio-sample
```

Es importante puntualizar que no se tiene que cerrar el stream explícitamente. Estos son cerrados automáticamente después de ser ejecutados.

6.2. Objetos *File*

Un objeto ***File*** es la representación de una ruta a un recurso. Por ejemplo, este puede ser un archivo, o también un directorio. Es importante tener en cuenta que esto no implica nada acerca de la existencia del recurso representado porque el objeto ***File*** propio no hace operaciones de Entrada/Salida, sólo almacena la ruta. Sin embargo, se puede comprobar la existencia del objeto ***File*** con el método **query_exists ()** :

```
1 if (file.query_exists ()) {
2         // File or directory exists
3 }
```

Para saber si el recurso es un directorio o no se puede buscar el *FileType*:

```
1 if (file.query_file_type (0) == FileType.DIRECTORY) {
2         // It's a directory
3 }
```

Se puede crear un Objeto File desde una ruta de archivo de sistema o una URI:

```
1 var data_file = File.new_for_path ("data.txt");
2 var message_of_the_day = File.new_for_path ("/etc/motd");
3 var home_dir = File.new_for_path (Environment.get_home_dir ()
    );
4 var web_page = File.new_for_uri ("http://live.gnome.org/Vala"
    );
```

Estos son métodos estáticos de fábrica, no constructores, ***File*** es una interface. Se puede crear una instancia de clases implementando ***File***.

GIO trabaja transparentemente a través de varios protocolos. No importa si el acceso es hecho directamente en el sistema local de archivos HTTP, FTP, SFTP, SMB o DAV. Esto es implementado por una capa de sistema de archivo llamado GVFS el cual es diseñado a ser extendible como respaldo de otros protocolos.

```
1 var home_dir = File.new_for_path (Environment.get_home_dir ()
    );   // ~
2 var bar_file = home_dir.get_child ("foo").get_child ("bar.txt
    ");  // ~/foo/bar.txt
3 var foo_dir = bar_file.get_parent ();
```

6.3. Algunas Operaciones simples de Archivos

Creando, renombrando, copiando y borrando archivos. Todas las operaciones en esta muestra son no-sincrónicas.

```
1  int main () {
2          try {
3
4                  // Reference a local file name
5                  var file = File.new_for_path ("samplefile.txt
                       ");
6
7                  {
8                          // Create a new file with this name
9                          var file_stream = file.create (
                               FileCreateFlags.NONE);
10
11                         // Test for the existence of file
12                         if (file.query_exists ()) {
13                                 stdout.printf ("File␣
                                       successfully␣created.\n");
14                         }
15
16                         // Write text data to file
17                         var data_stream = new
                               DataOutputStream (file_stream);
18                         data_stream.put_string ("Hello,␣world
                               ");
19                 } // Streams closed at this point
20
21                 // Determine the size of file as well as
                       other attributes
22                 var file_info = file.query_info ("*",
                       FileQueryInfoFlags.NONE);
23                 stdout.printf ("File␣size:␣%lld␣bytes\n",
                       file_info.get_size ());
24                 stdout.printf ("Content␣type:␣%s\n",
                       file_info.get_content_type ());
25
26                 // Make a copy of file
27                 var destination = File.new_for_path ("
                       samplefile.bak");
28                 file.copy (destination, FileCopyFlags.NONE);
29
30                 // Delete copy
```

```
31                  destination.delete ();
32
33                  // Rename file
34                  var renamed = file.set_display_name ("
                        samplefile.data");
35
36                  // Move file to trash
37                  renamed.trash ();
38
39                  stdout.printf ("Everything␣cleaned␣up.\n");
40
41          } catch (Error e) {
42          stderr.printf ("Error:␣%s\n", e.message);
43          return 1;
44  }
45
46  return 0;
47  }
```

Compilando y ejecutando

```
1  $ valac --pkg gio-2.0 gio-file-operations.vala
2  $ ./gio-file-operations
```

6.4. Escribiendo Datos

Este ejemplo crea un archivo de salida, abre un flujo (stream) a ese archivo y escribe algún texto. Para posibles cadenas largas escribe un lazo, el cual checa el número de bytes que han sido escritos, es usado.

```
1  int main () {
2          try {
3                  // an output file in the current working
                       directory
4                  var file = File.new_for_path ("out.txt");
5
6                  // delete if file already exists
7                  if (file.query_exists ()) {
8                          file.delete ();
9                  }
10
11                 // creating a file and a DataOutputStream to
                       the file
12                 /*
```

```
13                  Use BufferedOutputStream to increase write
                        speed:
14                  var dos = new DataOutputStream (new
                        BufferedOutputStream.sized (file.create (
                        FileCreateFlags.REPLACE_DESTINATION),
                        65536));
15                  */
16                  var dos = new DataOutputStream (file.create (
                        FileCreateFlags.REPLACE_DESTINATION));
17
18                  // writing a short string to the stream
19                  dos.put_string ("this␣is␣the␣first␣line\n");
20
21                  string text = "this␣is␣the␣second␣line\n";
22                  // For long string writes, a loop should be
                        used, because sometimes not all data can
                        be written in one run
23                  // 'written' is used to check how much of the
                         string has already been written
24                  uint8[] data = text.data;
25                  long written = 0;
26                  while (written < data.length) {
27                          // sum of the bytes of 'text' that
                                already have been written to the
                                stream
28                          written += dos.write (data[written:
                                data.length]);
29                  }
30          } catch (Error e) {
31          stderr.printf ("%s\n", e.message);
32          return 1;
33  }
34
35  return 0;
36  }
```

Compilar y ejecutar

```
1  $ valac --pkg gio-2.0 gio-write-data.vala
2  $ ./gio-write-data
```

6.5. Leyendo datos binarios

Esta muestra lee información del archivo de encabezado de imagen BMP así como de la imagen actual de datos.

```
1  int main () {
2          try {
3
4                  // Reference a BMP image file
5                  var file = File.new_for_uri ("http://wvnvaxa.
                       wvnet.edu/vmswww/images/test8.bmp");
6                  //      var file = File.new_for_path ("sample
                       .bmp");
7
8                  // Open file for reading
9                  var file_stream = file.read ();
10                 var data_stream = new DataInputStream (
                       file_stream);
11                 data_stream.set_byte_order (
                       DataStreamByteOrder.LITTLE_ENDIAN);
12
13                 // Read the signature
14                 uint16 signature = data_stream.read_uint16 ()
                       ;
15                 if (signature != 0x4d42) {   // this hex code
                        means "BM"
16                     stderr.printf ("Error:␣%s␣is␣not␣a␣
                           valid␣BMP␣file\n", file.
                           get_basename ());
17                     return 1;
18                 }
19
20                 data_stream.skip (8);        // skip
                       uninteresting data fields
21                 uint32 image_data_offset = data_stream.
                       read_uint32 ();
22
23                 data_stream.skip (4);
24                 uint32 width = data_stream.read_uint32 ();
25                 uint32 height = data_stream.read_uint32 ();
26
27                 data_stream.skip (8);
28                 uint32 image_data_size = data_stream.
                       read_uint32 ();
29
```

```
30                  // Seek and read the image data chunk
31                  uint8[] buffer = new uint8[image_data_size];
32                  file_stream.seek (image_data_offset, SeekType
                        .CUR);
33                  data_stream.read (buffer);
34
35                  // Show information
36                  stdout.printf ("Width: %ld px\n", width);
37                  stdout.printf ("Height: %ld px\n", height);
38                  stdout.printf ("Image data size: %ld bytes\n"
                        , image_data_size);
39
40          } catch (Error e) {
41          stderr.printf ("Error: %s\n", e.message);
42          return 1;
43  }
44
45  return 0;
46  }
```

Compilar y ejecutar

```
1  $ valac --pkg gio-2.0 gio-binary-sample.vala
2  $ ./gio-binary-sample
```

6.6. Listando el contenido del directorio

```
1  int main (string[] args) {
2          try {
3                  var directory = File.new_for_path (".");
4
5                  if (args.length > 1) {
6                          directory = File.
                               new_for_commandline_arg (args[1]);
7                  }
8
9                  var enumerator = directory.enumerate_children
                       (FileAttribute.STANDARD_NAME, 0);
10
11                 FileInfo file_info;
12                 while ((file_info = enumerator.next_file ())
                       != null) {
13                         stdout.printf ("%s\n", file_info.
                               get_name ());
```

```
14                  }
15
16          } catch (Error e) {
17          stderr.printf ("Error: %s\n", e.message);
18          return 1;
19  }
20
21  return 0;
22  }
```

Compilar y ejecutar

```
1  $ valac --pkg gio-2.0 gio-ls.vala
2  $ ./gio-ls
```

6.7. Listado de Archivos Asíncronos

```
1  using Gtk;
2
3  /**
4  * Loads the list of files in user's home directory and
       displays them
5  * in a GTK+ list view.
6  */
7  class ASyncGIOSample : Window {
8
9          private ListStore model;
10
11         public ASyncGIOSample () {
12
13                 // Set up the window
14                 set_default_size (300, 200);
15                 this.destroy.connect (Gtk.main_quit);
16
17                 // Set up the list widget and its model
18                 this.model = new ListStore (1, typeof (string
                       ));
19                 var list = new TreeView.with_model (this.
                       model);
20                 list.insert_column_with_attributes (-1, "
                       Filename",
21                 new CellRendererText (), "text", 0);
22
23                 // Put list widget into a scrollable area and
                        add it to the window
24                 var scroll = new ScrolledWindow (null, null);
```

```
25                 scroll.set_policy (PolicyType.NEVER,
                       PolicyType.AUTOMATIC);
26                 scroll.add (list);
27                 add (scroll);
28
29                 // start file listing process
30                 list_directory.begin ();
31         }
32
33         private async void list_directory () {
34                 stdout.printf ("Start␣scanning␣home␣directory
                       \n");
35                 var dir = File.new_for_path (Environment.
                       get_home_dir ());
36                 try {
37                         // asynchronous call, to get
                               directory entries
38                         var e = yield dir.
                               enumerate_children_async (
                               FileAttribute.STANDARD_NAME,
39                         0, Priority.DEFAULT);
40                         while (true) {
41                                 // asynchronous call, to get
                                       entries so far
42                                 var files = yield e.
                                       next_files_async (10,
                                       Priority.DEFAULT);
43                                 if (files == null) {
44                                         break;
45                                 }
46                                 // append the files found so
                                       far to the list
47                                 foreach (var info in files) {
48                                         TreeIter iter;
49                                         this.model.append (
                                               out iter);
50                                         this.model.set (iter,
                                               0, info.get_name
                                               ());
51                                 }
52                         }
53                 } catch (Error err) {
54                 stderr.printf ("Error:␣list_files␣failed:␣%s\
                       n", err.message);
55         }
```

```
}

static int main (string[] args) {
        Gtk.init (ref args);

        var demo = new ASyncGIOSample ();
        demo.show_all ();

        Gtk.main ();
        return 0;
}
}
```

Compilar y ejecutar

```
$ valac --pkg gtk+-3.0 gio-async.vala
$ ./gio-async
```

6.8. Flujo de lectura asíncrona

```
MainLoop main_loop;

async void read_something_async (File file) {
        var text = new StringBuilder ();
        print ("Start...\n");

        try {
                var dis = new DataInputStream (file.read ());
                string line = null;
                while ((line = yield dis.read_line_async (
                   Priority.DEFAULT)) != null) {
                        text.append (line);
                        text.append_c ('\n');
                }
                print (text.str);
        } catch (Error e) {
        error (e.message);
}
main_loop.quit ();
}

void main (string[] args) {

        var file = File.new_for_uri ("http://www.gnome.org");

        if (args.length > 1) {
```

```
26                  file = File.new_for_commandline_arg (args[1])
                        ;
27          }
28
29          main_loop = new MainLoop ();
30          read_something_async (file);
31          main_loop.run ();
32  }
```

Compilar y ejecutar

```
1  $ valac --pkg gio-2.0 gio-async-reading.vala
2  $ ./gio-async-reading
```

Capítulo 7

Programación con GTK+

En términos de ingeniería del software, un widget es un componente software visible y personalizable. Visible porque está pensado para ser usado en los interfaces gráficos de los programas, y personalizable porque el programador puede cambiar muchas de sus propiedades. De esta forma se logra una gran reutilización del software, un objetivo prioritario en ingeniería del software. Los widgtes se combinan para construir los interfaces gráficos de usuario. El programador los adapta según sus necesidades sin tener que escribir más código que el necesario para definir los nuevos valores de las propiedades de los widgets.

La librería GTK+ sigue el modelo de programación orientado a objetos. La jerarquía de objetos comienza en GObject de la librería Glib del que hereda GtkObject. Todos los widgets heredan de la clase de objetos GtkWidget, que a su vez hereda directamente de GtkObject. La clase GtkWidget contiene las propiedades comunes a todos los widgets; cada widget particular le añade sus propias propiedades.

Los widgets se definen mediante punteros a una estructura GtkWidget. En GTK+, los widgets presentan una relación padre/hijo entre sí. Las aplicaciones suelen tener un widget **ventana** de nivel superior que no tiene padre, pero aparte de él, todos los widgets que se usen en una aplicación deberán tener un widget padre. Al widget padre se le denomina contenedor. El proceso de creación de un widget consta de dos pasos: el de creación propiamente dicho y el de visualización. La función de creación de un widget tiene un nombre que sigue el esquema gtk_nombre_new donde **nombre** debe sustituirse por el nombre del widget que se desea crear. La función gtk_widget_show hará visible el widget creado.

La función de creación de un widget gtk_nombre_new devuelve un puntero a un objeto de tipo GtkWidget y no un puntero a un widget del tipo creado. Por ejemplo, la función gtk_button_new devuelve un puntero a un objeto de GtkWidget y no una referencia a un botón. Esta referencia puede convertirse a una referencia a un objeto GtkButton mediante la macro GTK_BUTTON, si se desea utilizar en lugares donde se requieran objetos botones. Aunque sería posible pasar en esos casos la referencia genérica, el compilador se quejará si se hace así posibilitando un control de tipos de objetos. Todo widget tiene una macro de conversión de una referencia genérica a una referencia al tipo propio. Eso sí, la macro únicamente funcionará correctamente si el widget referenciado fue creado con la función de creación del tipo apropiado, lo que incluye el propio tipo del widget o un descendiente.

El interfaz gráfico de una aplicación se construye combinando diferentes widgets (ventanas, cuadros combinados, cuadros de texto, botones, ...) y se establecen diversas retrollamadas (callbacks) para estos widgets, de esta forma se obtiene el procesamiento requerido por el programa a medida que se producen ciertas señales que a su vez provocan las retrollamadas. Las señales se producen por diversos sucesos como oprimir el boton de un ratón que se encuentra sobre un widget botón, pasar el cursor por encima de un widget u oprimir una tecla.

GTK+ utiliza GDK para visualizar los widgets. GDK es una interfaz de programación (API) de aplicaciones que se situa por encima de la API gráfica nativa como Xlib o Win32. De esta forma portando GDK pueden utilizarse las aplicaciones construidas con GTK+ en otras plataformas.

GTK+ o The GIMP Toolkit es un conjunto de bibliotecas multiplataforma para desarrollar interfaces gráficas de usuario (GUI), principalmente para los entornos gráficos GNOME, XFCE y ROX aunque también se puede usar en el escritorio de Windows, Mac OS y otros.

Inicialmente fueron creadas para desarrollar el programa de edición de imagen GIMP, sin embargo actualmente se usan bastante por muchos otros programas en los sistemas GNU/Linux. Junto a Qt es una de las bibliotecas más populares para Wayland y X Window System.

Licenciado bajo los términos de LGPL, GTK+ permite la creación de tanto software libre como software propietario. GTK+ es parte del proyecto GNU.

GTK+ se basa en varias bibliotecas desarrolladas por el equipo de GTK+ y de GNOME:

- **GLib** Biblioteca de bajo nivel estructura básica de GTK+ y GNOME. Proporciona manejo de estructura de datos para C, portabilidad, interfaces para funcionalidades de tiempo de ejecución como ciclos, hilos, carga dinámica o un sistema de objetos.
- **GTK** Biblioteca que actúa como intermediario entre gráficos de bajo nivel y gráficos de alto nivel.
- **ATK** Biblioteca para crear interfaces con características de una gran accesibilidad muy importante para personas discapacitadas o minusválidos. Pueden usarse utilerías como lupas de aumento, lectores de pantalla, o entradas de datos alternativas al clásico teclado o ratón.
- **Pango** Biblioteca para el diseño y renderizado de texto, hace hincapié especialmente en la internacionalización. Es el núcleo para manejar las fuentes y el texto de GTK+2.
- **Cairo** Biblioteca de renderizado avanzado de controles de aplicación.

7.1. Ejemplo Básico

En el siguiente ejemplo se muestra el programa gráfico que despliega un boton. Y al pulsarlo cambia la etiqueta de un botón.

```
1  using Gtk;
2
3  int main (string[] args) {
4      Gtk.init (ref args);
5
6      var window = new Window ();
7      window.title = "Primer␣Programa␣GTK+";
8      window.border_width = 10;
9      window.window_position = WindowPosition.CENTER;
10     window.set_default_size (350, 70);
11     window.destroy.connect (Gtk.main_quit);
12
13     var button = new Button.with_label ("Pulsame!");
14     button.clicked.connect (() => {
15         button.label = "Gracias";
16     });
17
18     window.add (button);
```

```
19      window.show_all ();
20
21      Gtk.main ();
22      return 0;
23  }
```

para compilar el programa se debe ejecutar:

```
1 $ valac --pkg gtk+-3.0 gtk-hello.vala
2 $ ./gtk-hello
```

7.2. Sincronización de Widgets

En la siguiente aplicación se sincronizan dos widgets:

```
1  using Gtk;
2
3  public class SyncSample : Window {
4
5      private SpinButton spin_box;
6      private Scale slider;
7
8      public SyncSample () {
9          this.title = "Introduce␣tu␣edad";
10         this.window_position = WindowPosition.CENTER;
11         this.destroy.connect (Gtk.main_quit);
12         set_default_size (300, 20);
13
14         spin_box = new SpinButton.with_range (0, 130, 1);
15         slider = new Scale.with_range (Orientation.HORIZONTAL
               , 0, 130, 1);
16         spin_box.adjustment.value_changed.connect (() => {
17             slider.adjustment.value = spin_box.adjustment.
                   value;
18         });
19         slider.adjustment.value_changed.connect (() => {
20             spin_box.adjustment.value = slider.adjustment.
                   value;
21         });
22         spin_box.adjustment.value = 35;
23
24         var hbox = new Box (Orientation.HORIZONTAL, 5);
25         hbox.homogeneous = true;
26         hbox.add (spin_box);
27         hbox.add (slider);
```

```
28          add (hbox);
29      }
30
31      public static int main (string[] args) {
32          Gtk.init (ref args);
33
34          var window = new SyncSample ();
35          window.show_all ();
36
37          Gtk.main ();
38          return 0;
39      }
40  }
```

```
1  $ valac --pkg gtk+-3.0 gtk-sync-sample.vala
2  $ ./gtk-sync-sample
```

7.3. Temporizador

El siguiente programa dispara una función de contador cada segundo

```
1  public class Sample : Object
2  {
3          public static int conta;
4     private static bool task()
5     {
6        stdout.printf("Ya␣%g␣\n", conta);
7        stdout.flush();
8           conta++;
9        if (conta<=10) {
10                  return true; // false para terminar el
                        temporizador
11                  }
12          else {
13                 return false;
14          }
15    }
16
17    public static int main(string[] args)
18    {
19            conta=0;
20       Timeout.add_seconds(1, task);
21       new MainLoop().run();
22       return 0;
```

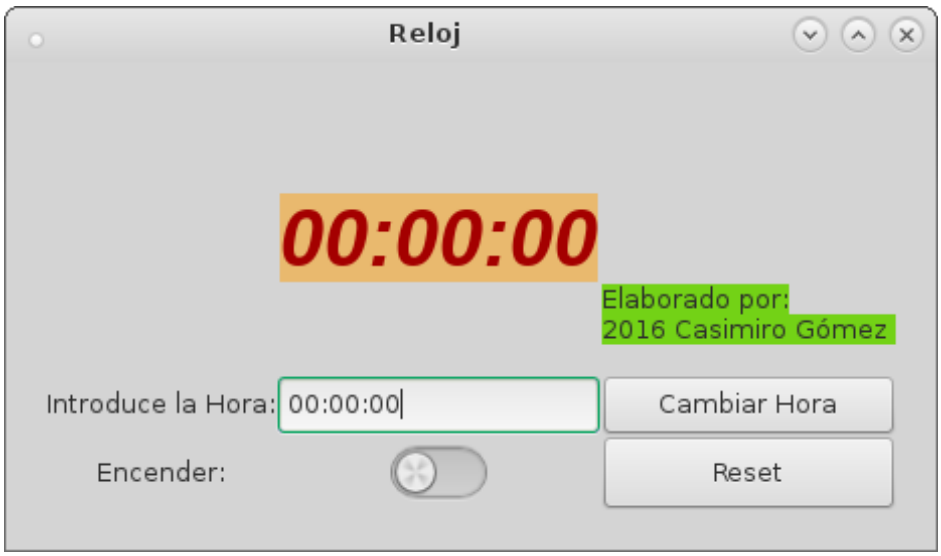

Figura 7.1: Pantalla del reloj diseñada en Glade

```
23     }
24 }
```

El diseño del temporizador es muy importante para muchos proyectos cuya lectura es periódica ya sea esta lectura de información del usuario o de sensores, la clave del funcionamiento esta en la instrucción **timeout**, la cual configura, en este ejemplo, la función **task** para que se realice periódicamente en intervalos de un segundo.

Para compilar el programa se ejecuta el siguiente código:

```
1 $ valac temporizador.vala
2 $ ./temporizador
```

una aplicación interesante del temporizador es el diseño de un reloj.

7.3.1. Proyecto de Ejemplo: Diseñando un reloj

Para aplicar los conocimientos del temporizador se plantea el diseño de un reloj, para ello en el programa de glade se diseña la interface

En la figura 7.2 se muestran los widgets usados en el diseño de la pantalla del reloj usando el programa Glade. El diseño en Glade debe seguir cierto orden:

- 1.- Primero debe colocarse un contenedor, en este caso el contenedor usado es **Window** (ventana en español), a este contenedor se le deja el nombre de **window1** que aparece por defecto, en la figura 7.2 se observa el nombre como **GtkWindow**.

- 2.- Colocar la rejilla, en este caso se coloca una rejilla de $3X3$ a la cual se le deja el nombre por defecto **grid1**, en la figura 7.2 aparece como **GtkGrid**.

- 3.- Se colocan tres etiquetas (o Label en inglés) las cuales se nombran **reloj**, **IntroHora** y **Encender**. La etiqueta de reloj desplegará la hora con el formato HH:MM:SS y se actualizará cada segundo, las otras dos son solo etiquetas que desplegarán mensajes para entrada de datos y boton interruptor respectivamente.

- 4.- Se colocan dos botones, el botón de **bReset** y el boton de **bCambiar**. El botón de *bReset* reiniciará el reloj al valor **00:00:00** y el boton de *bCambiar* cambiara el valor del reloj al valor que el usuario introduzca en la entrada de datos **GtkEntry** llamada **ValorHora**.

- 5.- Se coloca una entrada de datos **GtkEntry** llamada **ValorHora**, en donde el usuario introducirá la corrección de hora del reloj y posteriormente la enviará cuando pulse el botón **bCambiar**.

Una vez terminado el diseño de la pantalla en Glade, se configuran las funciones que controlarán los botones, como se puede observa en las figuras 7.3, 7.4, y 7.5. En este caso se necesitan tres funciones manejadoras, dos para los botones **bCambiar** y **bReset**, y una para el boton interruptor **Interru**. Las funciones manejadoras para el boton **bCambiar**, **bReset** y **Interru** se definen en la propiedades de sus respectivos botones en las pestaña de señales se especifica el nombre de la función manipuladora. Como se puede observar cuando se da click sobre el manipulador y se escribe una **o** el mismo glade propone un nombre a la función, es recomendable utilizar el nombre de la función propuesta y adicionar el *namespace* y la *clase* al nombre de la función manipuladora, estos nombres se escriben en minúscula aún cuando el nombre del namespace y de la clase sean en mayúsculas. En el caso del botón *Interru*, el nombre propuesto fue **on_Interru_state_set** y al adicionar el *namespace* y la *clase* el nombre queda **beagle_control_on_Interru_state_set**

Para el botón bReset el código es el siguiente:

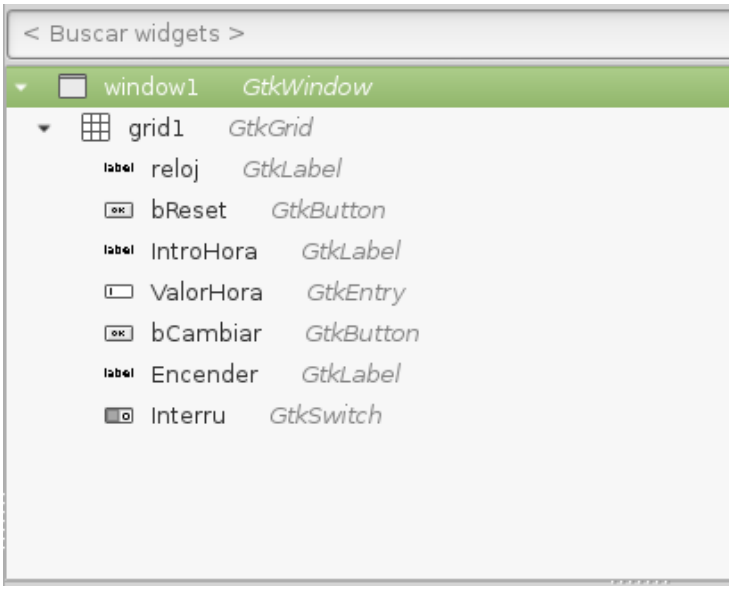

Figura 7.2: Widget usados en el diseño de la Pantalla del reloj en Glade

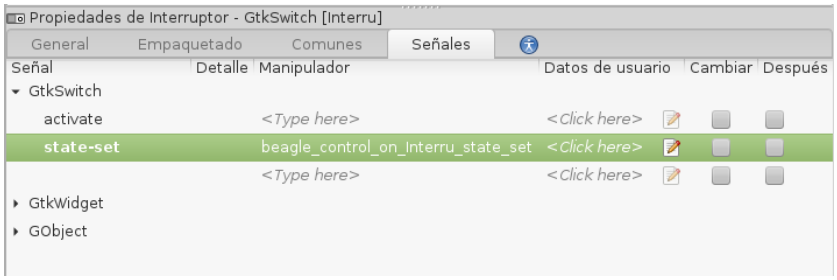

Figura 7.3: Manipulador del botón Interruptor

Figura 7.4: Manipulador del botón bCambiar

Figura 7.5: Manipulador del botón bReset

```
1         [CCode ( instance_pos = -1) ]
2         public void  on_bReset_clicked ( Button source ) {
3                 Useg=-1;
4                 Dseg=0;
5                 Umin=0;
6                 Dmin=0;
7                 Uhora=0;
8                 Dhora=0;
9         }
```

Como se muestra se inicializan las variables del reloj, cada variable esta dividida en segundo, minutos y horas y cada una respectivamente tiene sus unidades y sus decenas, de tal forma que la variable de unidades de segundos se llama Useg y cuenta del 0 al 9 y la variable de decenas de segundos cuenta del 0 al 5 y se llama Dseg. En el caso de los minutos es la misma técnica. Para las horas solo hay que tomar en cuenta que el límite son las 23 horas, si llegar a las 24 horas. Estas variables deben ser modificadas en cualquier método de la clase y actualizarse por lo cual se deben definir como varibales globales, como se muestra en el siguiente código

```
1  using Gtk ;
2
3  namespace Beagle {
4         public class Control:Object {
5
6         public string texto = "";
7         public static Entry entrada;
8         public static Label reloj;
9         public static bool bandera;
10
11     public static int Useg;
12     public static int Dseg;
13     public static int Umin;
```

```
14     public static int Dmin;
15     public static int Uhora;
16     public static int Dhora;
17     public static string hora;
```

La variable hora también es una variable global que es una cadena, esta cadena es resultada de la conversión y concatenación de las variables de cuenta Useg, Dseg, Umin, Dmin, Uhora y Dhora. Y es la que se imprime. Las variables entrada y reloj asimismo son variables globales que tomaran la entrada de datos del glade y permitirán la impresión de valores del reloj (en este caso de la varible hora) y la captura de hora que introduzca el usuario permitiendo su acceso en cualquier parte de la clase.

En el siguiente código se muestra la función manipuladora del botón bCambiar

```
1         [CCode ( instance_pos = -1) ]
2         public void  on_bCambiar_clicked ( Button source ) {
3                 texto= entrada.get_text();
4                  Dhora=int.parse(texto[0:1]);
5                  Uhora=int.parse(texto[1:2]);
6                  Dmin=int.parse(texto[3:4]);
7                  Umin=int.parse(texto[4:5]);
8                  Dseg=int.parse(texto[6:7]);
9                  Useg=int.parse(texto[7:8]);
10                reloj.set_text(texto);
11                stdout.printf("Hola␣Mundo␣%s␣oj␣\n", texto );
12        }
```

7.4. Programa Usando Hilos

Un programa escrito en Vala puede tener más de un hilo en ejecución, permitiendole hacer más de una cosa al mismo tiempo. Fuera del ámbito de Vala los hilos comparten un mismo procesador o no, dependiendo del entorno de ejecución.

Un hilo en Vala no se define en el tiempo de compilación, en lugar de eso se define una porción del código Vala para que se ejecute como un nuevo hilo. Esto se realiza mediante el método estático de la clase Thread de la biblioteca GLib, como puede verse en siguiente ejemplo:

```
1 public class MyThread : Object {
2         public int x_times { get; private set; }
3
```

```
4         public MyThread (int times) {
5                 this.x_times = times;
6         }
7
8         public int run () {
9                 for (int i = 0; i < this.x_times; i++) {
10                        stdout.printf ("ping! %d/%d\n", i +
                             1, this.x_times);
11                        Thread.usleep (10000);
12                }
13
14                // return & exit have the same effect
15                return 43;
16        }
17}
18
19public static int main (string[] args) {
20        // Check whether threads are supported:
21        if (Thread.supported () == false) {
22                stderr.printf ("Threads are not supported!\n"
                     );
23                return -1;
24        }
25
26        try {
27                // Start a thread:
28                MyThread my_thread = new MyThread (10);
29                Thread<int> thread = new Thread<int>.try ("My
                     fst. thread", my_thread.run);
30
31                // Wait until thread finishes:
32                int result = thread.join ();
33                // Output: 'Thread stopped! Return value: 42'
34                stdout.printf ("Thread stopped! Return value:
                     %d\n", result);
35        } catch (Error e) {
36                stdout.printf ("Error: %s\n", e.message);
37        }
38
39        return 0;
40}
```

7.5. Dos hilos

```
1  class MyThread {
2
3      private string name;
4      private int count = 0;
5
6      public MyThread (string name) {
7          this.name = name;
8      }
9
10     public void* thread_func () {
11         while (true) {
12             stdout.printf ("%s: %i\n", this.name, this.count)
                   ;
13             this.count++;
14             Thread.usleep (Random.int_range (0, 200000));
15         }
16     }
17 }
18
19 int main () {
20     if (!Thread.supported ()) {
21         stderr.printf ("Cannot run without thread support.\n"
               );
22         return 1;
23     }
24
25     var thread_a_data = new MyThread ("A");
26     var thread_b_data = new MyThread ("B");
27
28     try {
29         // Start two threads
30         /* With error handling */
31         Thread<void*> thread_a = new Thread<void*>.try ("
               thread_a", thread_a_data.thread_func);
32         /* Without error handling (is not using the try/catch
               ) */
33         Thread<void*> thread_b = new Thread<void*> ("thread_b
               ", thread_b_data.thread_func);
34
35         // Wait for threads to finish (this will never happen
                in our case, but anyway)
36         thread_a.join ();
37         thread_b.join ();
38
39     } catch (Error e) {
```

```
40          stderr.printf ("%s\n", e.message);
41          return 1;
42      }
43
44
45      return 0;
46  }
```

Capítulo 8

Controlando linux desde Vala

La manera de la que vamos a acceder a los GPIO (en este caso al 17) es como si fuesen directorios. Podemos utilizar comandos como **ls**, **cat** o **echo**, entre otros, para conocer la estructura y contenidos de los directorios.

Ahora mismo no tenemos ningún pin accesible. Ni de entrada ni de salida. Tenemos que crearlo nosotros mismos. Queremos tener acceso al GPIO 17, así que introducimos el siguiente comando:

```
1 echo 17 > /sys/class/gpio/export
```

Tras esto, el sistema ha creado un archivo con una estructura GPIO que corresponde al número 17. A continuación, tenemos que informar a la Raspberry Pi de si el pin va a ser de salida o de entrada. Como lo que queremos es encender un LED, el GPIO 17 será de salida. Introducimos el siguiente comando:

```
1 echo out > /sys/class/gpio/gpio17/direction
```

Con esto, el sistema ya sabe que el pin será de salida. Ahora tendremos que darle valores. Existen dos posibles: **0** y **1**.

8.0.1. Para encender el LED

```
1 echo 1 > /sys/class/gpio/gpio17/value
```

8.0.2. Para apagar el LED:

```
1 echo 0 > /sys/class/gpio/gpio17/value
```

Una vez hayamos acabado de encender y apagar el LED, tendremos que eliminar la entrada GPIO creada, es decir, el GPIO 17. Para ello introduciremos el siguiente comando:

```
1 echo 17 > /sys/class/gpio/unexport
```

Bibliografía