Documento: Referencia del Lenguaje B4A
Nota: Tomar en cuenta las Reglas de Clase (solo utilizar como referencia).

El Lenguaje de B4A

BASIC

B4A es un dialecto de BASIC (Beginner's All-purpose Symbolic Instruction Code), una familia de lenguajes de programación de alto nivel diseñados para ser fáciles de usar. Creado en 1964, en una época en la que la programación todavía era técnicamente difícil, BASIC fue diseñado para ser fácil de usar y se generalizó con la introducción de los microordenadores.
Aparecieron muchos dialectos y los escritos por la nueva Microsoft fueron especialmente populares. Visual Basic de la compañía es muy utilizado para desarrollar programas para Windows.

B4A

En 2005, la empresa israelí Anywhere Software creó "Basic for PPC", un sistema para el desarrollo de aplicaciones para ordenadores Pocket PC. En 2010, apareció una versión que podía crear Apps para dispositivos Android y que se convirtió en Basic4Android en 2011, rebautizada como B4A en 2014.

Reglas del Léxico

Las reglas del Léxico determinan cómo debe escribirse el código. Las reglas básicas son:
- B4A no distingue entre mayúsculas y minúsculas. El editor cambiará automáticamente el caso de las palabras clave.
- A diferencia de algunos idiomas, no se requiere un punto y coma (;) al final de cada línea. Simplemente se terminan con un retorno de carro.

Separador de declaraciones

Se pueden escribir dos sentencias en una línea separándolas con dos puntos:

```
Dim intX As Int: If intY > 3 Then intX = 2 Else intX = 9
```

(Debería considerar que su código es más fácil de leer si se colocan en líneas separadas).

Comentarios

Para muchas aplicaciones, se gasta más tiempo en mantener y mejorar el código que el que se empleó incialmente en escribirlas, por lo que es esencial que sean fáciles de leer y entender. Para este propósito, los comentarios son importantes. Explican el propósito de las variables y subs. Se utiliza una única comilla para añadir un comentario en una línea. Por ejemplo:

```
'Enviar una solicitud POST con el archivo proporcionado como datos del
mensaje.
' Este método no funciona con archivos de recursos (assets).
Public Sub PostFile(Link As String, Dir As String, FileName As String)
 If Dir = File.DirAssets Then ' Dir no es válido
  Msgbox("No se pueden enviar archivos de la carpeta de recursos.",
"Error")
     Return
 Else
  '...
 End If
End Sub
```

Esto ilustra algunos principios importantes que facilitarán el mantenimiento de su código:

Nombres Representativos

Elija nombres con significado para variables y subs, con el objetivo que su función sea clara.

Comentarios como Documentación
Documente sus subs, agregando comentarios antes de ellas. Vea aquí para más información.

Dividir líneas muy largas
Las largas líneas de código son difíciles de leer:
```
Sub dblSecsToJ2000 (intYear As Int, intMonth As Int, intDay As Int,
intHour As Int, intMin As Int, intSec As Int, floLat As Float, floLong
As Float, bRound As Boolean) As Double
```
El carácter de subrayado se puede utilizar para dividir líneas largas. Por ejemplo:
```
Sub dblSecsToJ2000 ( _
 intYear As Int, intMonth As Int, intDay As Int, _
 intHour As Int, intMin As Int, intSec As Int, _
 floLat As Float, floLong As Float, bRound As Boolean _
) As Double
```
Tenga en cuenta que Smart Strings puede definir cadenas multilínea sin necesidad de carácter de subrayado.

Variables
Una **variable** es un nombre simbólico dado a una cierta cantidad o información para permitir que los datos sean fácilmente manipulados y modificados.

Constantes
Para definir una constante, utilice la palabra clave **Const**:
```
Dim Const dblDiameterEarth As Double = 12756.2
```
Una vez declarado, el valor de una constante no puede modificarse

Tipos
El **tipo** de una variable es el tipo de datos que puede asignarsele. El tipo de variables en B4A se deriva directamente del sistema de tipos de Java. Hay dos tipos de variables: primitivas y no primitivas.

Tipos Primitivos
Estos son los tipos fundamentales en B4A.
En la siguiente lista de tipos primitivos con sus rangos, "~" significa "aproximadamente igual a"

Boolean
Tipo..... Boolean
valor min..... FALSE
valor max..... TRUE
Note que FALSE se almacena como 0.

Byte
Tipo..... 8 bits (1 byte con signo)
valor min..... $-2^7 = -128$
valor max..... $2^7 - 1 = 127$

Short
Tipo..... integer 16 bits (2 bytes con signo)
valor min..... $-2^{15} = -32768$
valor max..... $2^{15} -1 = 32767$

Int
Tipo..... integer 32 bits (4 bytes con signo)
valor min..... $-2^{31} = -2147483648$
valor max..... $2^{31} -1 = 2147483647$

Long
Tipo..... long integer 64 bits (8 bytes con signo)
valor min..... $-2^{63} = -9,223,372,036,854,775,808$
valor max..... $2^{63} -1 = 9,223,372,036,854,775,807$

Float
Tipo..... floating point number 32 bits (4 bytes, ~7 digits)
valor negativo max..... $-(2-2^{-23}) * 2^{127} \sim -3.4028235*10^{38}$
valor negativo min..... $-2^{-149} \sim -1.4*10^{-45}$
valor positivo min..... $2^{-149} \sim 1.4*10^{-45}$
valor positivo max..... $(2-2^{-23}) * 2^{127} \sim 3.4028235*10^{38}$

Double
Tipo..... número de doble precisión de 64 bits (8 bytes, ~15 digits)
valor negativo max..... $-(2-2^{-52}) * 2^{1023} \sim -1.7976931348623157*10^{308}$
valor negativo min..... $-2^{-1074} \sim -2.2250738585072014*10^{-308}$
valor positivo min..... $2^{-1074} \sim 2.2250738585072014*10^{-308}$
valor positivo max..... $(2-2^{-52}) * 2^{1023} \sim 1.7976931348623157*10^{308}$

Char
Tipo..... caracteres, 2 bytes sin signo

String
Tipo..... array de caracteres

Literales Hexadecimales
B4A soporta la escritura de números enteros en notación hexadecimal, (a menudo acortada a "hex"). Para más detalles sobre el hex, ver aquí.
Debe poner como prefijo el número con 0x (el 0 es el número cero). Por lo tanto, puede escribir

```
Dim iSize As Int
iSize = 0x2C
Log (iSize) ' muestra 44
```

Tipos No-Primitivos
Todos los demás tipos, incluidas los arrays de tipos primitivos, se clasifican como tipos no-primitivos.

Tipos Básicos
En las siguientes secciones, damos detalles de los tipos no-primitivos incluidos en B4A; los llamados Tipos Básicos.

Pasar por Referencia los tipos No-Primitivos
Cuando se pasa una variable de tipo no-primitivo a un Sub, o cuando se asigna un no-primitivo a una variable diferente, se pasa una referencia a la variable[37]. Esto significa que los datos en sí no están duplicados. Para ejemplos, vea abajo Pasar por Referencia.

[37] Por referencia se entiende como un enlace a la propia variable (como un puntero a su posición en memoria)

Conversión de Tipos

En B4A, los tipos de variables se convierten automáticamente según se necesiten. Por ejemplo:

```
Dim str As String
Dim i As Int
i = 3
str = i ' conversión automatic de tipo
Log (str) ' muestra 3

' conversión de cadena a entero
str = "4"
i = str ' conversión automatic de tipo
Log (i) ' muestra 4
```

Error al convertir cadena a entero

```
str = "hello"
i = str
```

La última línea no se puede ejecutar y produce un error en tiempo de ejecución: NumberFormatException
Este problema se puede resolver usando código como:

```
If IsNumber(str) Then
 i = str
End If
```

Error convertiendo números en Coma Flotante a Cadenas de texto

Los números en coma flotante (Floats y Doubles) sólo se mantienen como valores aproximados en B4A, y por lo tanto convertirlos en cadenas puede causar errores:

```
Dim flt As Float = 1.23
Dim str As String = flt
Log ("str desde flt da " & str)
```

El log muestra

```
str desde flt da 1.2300000190734863
```

Errores similares pueden aparecer al convertir Double a Cadena.
La solución es utilizar Round2 or NumberFormat or NumberFormat2, por ejemplo:

```
Dim flt As Float = 1.23
str = Round2(flt, 4)
Log ("Round2 flt da " & str)
```

El log muestra

```
Round2 flt da 1.23
```

Rank

Puede que a veces vea un error en tiempo de compilación tal como:

```
Cannot cast type: {Type=Int,Rank=0} to: {Type=Int,Rank=1}
```

Rank=0 es una variable simple, Rank=1 se refiere a un array.

Creando sus propios Tipos

Puede crear un nuevo tipo utilizando la palabra clave Type. Vea aquí para más detalles.

Objectos (Objetos)

Un objeto es un concepto útil en programación ya que nos permite representar objetos del mundo real en nuestro código. Esto nos ayuda a diseñar mejores aplicaciones y más robustas. Los objetos pueden tener atributos (también llamados propiedades) y comportamientos (también llamados funciones o métodos). En B4A, estos se denominan colectivamente Miembros.
Además, un objeto puede responder a las acciones del usuario invocando Eventos, que describimos en otra sección.
Por ejemplo, un botón tiene atributos **Left** y **Top** (que determinan su posición en la pantalla), y tiene comportamientos que determinan cómo responde a los comandos.

```
Dim btn As Button
btn.Initialize("Menu")
btn.Left = 20dip
btn.BringToFront
```

Si no estamos seguros de qué tipo de variable vamos a tratar, podemos declarar una variable como Objeto. Un Objeto puede contener cualquier tipo de variable.

```
Dim objThing As Object
```

Más tarde podemos comprobar su tipo:

```
If objThing Is Bitmap Then
```

Si una variable que contiene un Objeto se asigna a una segunda variable, ambas se referirán al mismo objeto:

```
Dim btnTest As Button
Dim btnSameAsTest As Button
btnSameAsTest = btnTest
```

Ahora cualquier cosa que haga para **btnSameAsTest** también afecta a **btnTest**. Este es un ejemplo de pasar por referencia.
Una colección, como un **List** o un **Map**, opera con objetos y por lo tanto puede almacenar cualquier tipo de datos. No es necesario que todos sus elementos contengan el mismo tipo. Por otro lado, un array puede almacenar un solo tipo en todos sus elementos.

Inicialización de Objetos

Los Objetos deben inicializarse (es decir, asignarle un valor) antes de su utilización. De lo contrario, no se pueden utilizar. Considere por ejemplo un botón. Primero lo declaramos:

```
Sub Globals
 Dim btnAddRoute As Button
End Sub
```

Luego lo inicializamos y declaramos el nombre del evento que se utilizará para gestionar sus eventos:

```
Sub Activity_Create(FirstTime As Boolean)
 btnAddRoute.Initialize("GetPath")
End Sub
```

Luego creamos subs para gestionar cada evento que se requiera:

```
Sub GetPath_click
 ' hacer algo
End Sub
Sub GetPath_LongClick
' hacer algo
End Sub
```

El IDE proporciona el sistema de AutoCompletado, que es una forma fácil de crear estos subs y asegurarse de que tienen los argumentos correctos.

Declaración de Variables

Por "declarar una variable" se entiende "decir a B4A el nombre, tipo y (quizás) número de dimensiones de una variable".

Sentencia Dim

La forma de declarar una variable es usar la sentencia Dim. La palabra "Dim" viene de "dimension" porque, si se desea utilizar un array, hay que declararlo y especificar el número de dimensiones.
En B4A, no es esencial declarar una variable antes de usarla, pero es una práctica recomendable. Esta es una buena manera de reducir errores lógicos dentro de su código, ya que le dice al compilador que permita sólo asignar valores del tipo específico a esa variable. Si no declara una variable antes de usarla, B4A asume que es un tipo **`String`**.
Las variables se declaran con la palabra clave **`Dim`** seguida del nombre de la variable, la palabra clave **`As`** y el **`tipo`** de variable. Si se trata de un array, el nombre de la variable va seguido de paréntesis que encierran el número de dimensiones. Las variables también se pueden inicializar cuando se declaran.
Ejemplos:

```
Dim dblCapital As Double
Dim dblExpenses(10) As Double
Dim i = 0 As Int
Dim intData(3, 5, 10) As Int
```

Las variables del mismo tipo pueden declararse junto con sus nombres separados por comas y seguidos de la declaración de tipo. Se pueden inicializar al mismo tiempo:

```
Dim dblCapital, dblInterest, dblRate As Double
Dim i = 0, j = 2, k = 5 As Int
```

Se pueden declarar variables de diferentes tipos en la misma línea:

```
Dim txt As String, value As Double, flag As Boolean
```

Sin embargo, esto puede ser complicado de leer:

```
Dim txt = "test" As String, value = 1.05 As Double, flag = False As
Boolean
```

Éstas deberían distribuirse mejor en varias sentencias Dim. Por lo general, lo mejor es hacer que tu código sea lo más simple posible para que los humanos puedan leerlo y entenderlo, especialmente tú mismo, ¡cuando tengas que mantener tu propia aplicación!.

Visibilidad o Ámbito de las Variables

Además de la sentencia Dim, también puede declarar variables utilizando las palabras clave **`Public`** o **`Private`**, que se describen aquí.

No Opción Explicita

A veces los desarrolladores pueden perder tiempo buscando errores causados por nombres de variables mal escritos. A diferencia de Microsoft Visual Basic, no hay ninguna Opción Explícita en B4A. La Opción Explícita requería que antes de usar una variable, esta fuera declarada usando la sentencia, pero en B4A no tiene que declarar variables. Por lo tanto, las siguientes líneas compilarán sin problema con B4A:

```
Sub Activity_Create(bFirstTime As Boolean)
 intX = 16
 Log (intX)
End Sub
```

Avisos del editor sobre variables no declaradas

El editor IDE resalta una variable no declarada en rojo (como **intX** que se muestra arriba) como advertencia de que se está usando antes de que haya sido declarada, y también pone ese mensaje en el área de advertencia en la parte superior de la ventana Logs, pero sin embargo el código si compilará.
Nota: **intX** se declarará automáticamente como un **String**, lo que claramente ¡no es la intención del desarrollador!

Asignación de Valores

Para asignar un valor a una variable, escriba su nombre seguido del signo igual, seguido del valor, por ejemplo:

```
Capital = 1200
LastName = "SMITH"
```

Nota: los valores de las strings, como `LastName`, deben escribirse entre comillas dobles.

Verificación de Tipo

El principal beneficio de declarar una variable es que, si se intenta asignar un tipo incorrecto de datos a una variable (lo que indica un error lógico de su parte):

```
Dim yourAge As Int
yourAge = "Seventeen"
```

se producirá un error de ejecución y el programa se detendrá, mostrando el error en la ventana Log:

```
Error occurred on line: 35 (Main)
java.lang.NumberFormatException: Invalid double: "Seventeen"
```

Estos errores deben detectarse durante los test, ¡si su testeo es eficaz!

Uso de Variables no asignadas

Las variables, declaradas o no, no pueden utilizarse antes de que se les asigne un valor. Lo siguiente (que escribe mal el nombre de la variable) producirá un error cuando intente compilar el código:

```
myAge = 16
yourAge = myAg * 2
```

El error que se produce será:

```
Parsing code.    Error
Error parsing program.
Error description: Undeclared variable 'myag' is used before it was
assigned any value.
Occurred on line: 37
yourAge = myag * 2
```

Pasar por Valor

Los tipos primitivos siempre se pasan por valor a otros subs o cuando se asignan a otras variables. La alternativa, pasando una referencia a una variable primitiva, no se ha implementado. Esto significa que no se puede alterar el valor original desde dentro de una subrutina.
Ejemplo:

```
Sub S1
 Dim A As Int
 A = 12
 ' pasar una copia del valor de A a la rutina S2
 S2(A)
 Log(A)
 ' Muestra 12. Este valor de A no se modifica
End Sub

Sub S2(A As Int)
 ' Esta A es una copia local
 A = 45
 ' Sólo se modifica el valor de la copia local
End Sub
```

Pasar por Referencia

Los tipos no-primitivos, tales como arrays, objetos y tipos siempre se pasan a otros subs por referencia. Si se modifica la referencia, se modifica el objeto original. Por ejemplo:

```
Sub S1
 Dim A(3) As Int
 A(0) = 12
 ' pasar una referencia de A a la rutina S2
 S2(A)
 Log(A(0))
 ' Muestra 45
End Sub

Sub S2(B() As Int)
 ' Esta B es una referencia al original
 B(0) = 45
 ' Se modifica el valor original A(0)
End Sub
```

Lo mismo es cierto cuando un tipo no-primitivo como un array se asigna a otra variable. La segunda variable es una referencia a la primera. Ejemplo:

```
Dim A(3), B(3) As Int
A(0) = 12
B = A
' B es una referencia a A
Log(B(0)) ' Muestra 12
' Cambiar tanto A como B
A(0) = 45
Log(B(0)) ' Muestra 45
```

Lo mismo es cierto para cualquier variable de tipo no-primitivo, como un objeto:

```
Dim lbl1, lbl2 As Label
lbl1.Initialize("")
lbl2.Initialize("")
lbl1.TextSize = 20
Log (lbl1.TextSize) ' muestra 20
lbl2 = lbl1
' lbl2 es una referencia a lbl1
' si cambia lbl2 cambiará también lbl1
lbl2.TextSize = 40
Log (lbl1.TextSize) ' muestra 40
```

Nombres de Variables

Debe identificar sus variables dándoles nombres. Un nombre de variable debe comenzar con una letra y debe estar compuesto de los siguientes caracteres: A-Z, a-z, 0-9, y guión bajo "_". No se pueden utilizar espacios, paréntesis, etc.
Los nombres de variables no distinguen entre mayúsculas y minúsculas. Esto significa que "Var" y "var" se refieren a la misma variable. El editor cambia automáticamente el caso de un nombre de variable para que coincida con el caso que usó cuando lo declaró.
No puede utilizar palabras reservadas (palabras clave listadas en este capítulo) como nombres de variables. Sin embargo, puede utilizar tipos de objeto como Bitmap. Así:

```
Dim Int As Int            ' Esto es erróneo
Dim Bitmap As Bitmap      ' Esto es correcto, aunque no una práctica
                          ' recomendable
Dim Bitmap1 As Bitmap     ' Esto está mejor
Dim bmpMyPhoto As Bitmap  ' Esto es Perfecto
```

Nota: el uso de tipos de objeto como nombres de variables (Bitmap, por ejemplo) es ampliamente considerado como una mala práctica, ya que puede causar confusión; por ejemplo, el IDE aplicará erróneamente el color la variable . La mejor práctica es utilizar la notación húngara (punto siguiente).

Notación Húngara

Puede ayudar a recordar qué tipo de datos necesita una variable utilizando la llamada notación húngara. En Hungría (y en otras culturas), el apellido se cita antes del nombre. Por lo tanto, en las variables que usan esta convención, la primera parte del nombre de la variable le dice qué tipo de objeto está manejando. Por ejemplo, un entero podría llamarse intMyAge, una cadena podría llamarse strMyName y así sucesivamente. Algunos prefijos sugeridos:

```
Dim bMale As Boolean
Dim btnNext As Button
Dim byteMyData As Byte
Dim chkFavorite As CheckBox
Dim chrInitial As Char
Dim dblSunDistance As Double
Dim edtInterest As EditText
Dim fltWeight As Float
Dim intAge As Int
Dim lblCapital As Label
Dim lngDate As Long
Dim lstNames As List
Dim mapPeople As Map
Dim pnlBackground As Panel
Dim shrtAge As Short
Dim strName As String
Dim spnChoice As Spinner
Dim wbvMyPage As WebView
```

Arrays (Matrices) Arreglos

Un array es una colección de valores u objetos del mismo tipo. Estos elementos se mantienen dentro del array en un orden fijo y los elementos individuales pueden ser seleccionados especificando su posición usando un número de índice.

Dimensiones

Los arrays pueden tener múltiples dimensiones. Piense en un array como una matriz unidimensional de una fila de objetos (vector). Escoge uno de ellos contando a lo largo de la fila hasta que encuentres el que quieres. Consideraremos matriz al array multidimensional. Una matriz bidimensional es como un tablero de ajedrez con cada cuadrado conteniendo un objeto. Para seleccionar uno de ellos, debe especificar dos números, uno para la posición horizontal y otro para la vertical. Este plan puede extenderse a cualquier número de dimensiones, ¡aunque cada vez son más difíciles de imaginar!

Declarando un Array (Arreglo)

Una array unidimensional se declara como sigue:

```
Dim strLastName(50) As String
```

La declaración contiene la palabra clave **Dim** seguida del nombre de la variable **strLastName**, las dimensiones entre paréntesis **(50)**, la palabra clave As y, opcionalmente, el tipo de variable por ejemplo **String**. Este array puede contener un total de 50 cadenas, desde **strLastName(0)** hasta **strLastName(49)**.

Tipo por Defecto

Si se omite el tipo, por defecto será String.

```
Dim arr(3) ' este array puede almacenar 3 Strings
```

Otros Ejemplos

Un array de dos dimensiones (matriz) de tipo Doubles, con un total de 9 elementos.

```
Dim dblMatrix(3, 3) As Double
```

Un array de tres dimensiones (matriz) de tipo integer, con un total de 150 elementos.

```
Dim intData(3, 5, 10) As Int
```

No se puede cambiar las dimensiones

A diferencia de Visual Basic, no es posible cambiar el número de dimensiones de un array. Si necesita esto, sería mejor usar un tipo list o map.

Guardando y Recuperando Datos

Para almacenar datos en un array, debe especificar en qué posición desea almacenarlos. Esto se hace dando un número de índice, comenzando con 0 como primera posición.

```
Dim strLastName(2) As String
strLastName(0) = "Jones"
strLastName(1) = "Smith"
```

Puede leer datos de un array si conoce su posición dentro de dicho array. Por ejemplo, para elegir el primer elemento, usted diría:

```
Dim strPatient As String
strPatient = strLastName(0)
```

Ahora **strPatient** es "Jones"

En un array, el primer índice de cada dimension empieza en 0:

```
strLastName(0), dblMatrix(0,0), intData(0,0,0)
```

El ultimo índice de una dimension es el número de elementos (establecido en la declaración) menos 1.

```
Dim dblMatrix(3,3) As Double
dblMatrix(2,2) = 1.233
Dim intData(3,5,10) As Int
intData(2,4,9) = 36676
```

El siguiente ejemplo muestra cómo acceder a todos los elementos de un array tridimensional:

```
Dim intData(3,5,10) As Int
For i = 0 To 2
 For j = 0 To 4
  For k = 0 To 9
   intData(i,j,k) = i + j + k
  Next
 Next
Next
```

Las dimensiones se pueden especificar con variables

El ejemplo anterior demuestra que puede utilizar variables (i, j y k) para especificar el índice cuando almacena o recupera datos. También se pueden usar variables para especificar el número de elementos cuando se declara un array, como se muestra en la última línea de este código:

```
Dim intFriends As Int
' leer el número de amigos desde la entrada del usuario
intFriends = txtNumFriends.Text
' Declarar array para guardar los nombres de los amigos
Dim strLastName(intFriends) As String
```

Rellenar un array utilizando la palabra clave Array

Un array se puede declarar sin especificar su tamaño:

```
Dim strNames() As String
```

Y se puede rellenar después utilizando la palabra clave **Array**:

```
strNames = Array As String("Miller", "Smith", "Johnson", "Jordan")
```

De hecho, esto crea un nuevo array y luego establece strNames como referencia.

Tamaño de un Array

Puede determinar cuántos elementos hay en un array utilizando el método Length:

```
Dim strNames() As String = Array As String( _
 "Miller", "Smith", "Johnson", "Jordan")
Log(strNames.Length) ' muestra 4
```

Tenga en cuenta que Length sólo le dice cuántos elementos hay en la primera dimensión, así que

```
Dim x(3,4) As Int
Log(x.Length) ' muestra 3
```

Arrays de Objetos

Las vistas u otros objetos pueden ser almacenados en un Array. Se muestra un ejemplo en la sección Gestor de Eventos Compartidos. Si la palabra clave Array se usa sin un tipo, entonces se asume Objeto:

```
Dim person() As Object = Array ("Tom", 37, tomsPhoto)
```

Las Dimensiones del Array son Fijas

Una de las limitaciones de los arrays es que sus dimensiones son fijas. Una vez que se ha creado un array, el número de elementos que puede contener es fijo. No puede decidir más tarde hacerlo más grande a menos que lo reemplace por un nuevo array:

```
strNames = Array As String("Jones", "Windor")
' reemplazar los datos originales con algunas cadenas nuevas
strNames = Array As String("Miller", "Smith", "Johnson", "Jordan")
```

El array ha cambiado sus dimensiones pero los datos originales se pierden.
Esta limitación puede evitarse utilizando **Lists** o **Maps**, que le permiten añadir datos a las estructuras existentes:

Lists (Listas)

Las listas **List**s son similares a los arrays pero son dinámicas: puede añadir y eliminar elementos de una lista y su tamaño cambiará:

```
Dim lstNames As List
lstNames.Initialize
lstNames.Add("David")
lstNames.Add("Goliath")
```

Las listas se asemejan a los arrays en el sentido de que se accede a sus elementos utilizando un número de índice. Fíjese que el índice de una lista empieza también en cero:

```
Dim strName As String
strName = lstNames.Get(1)
Log(strName) ' Muestra Goliath
lstNames.RemoveAt(0) ' elimina David
```

Hay otros beneficios de usar listas. Por ejemplo, las listas pueden contener cualquier tipo de objeto. Una descripción detallada de todas las funciones se encuentra en la Sección List.

Maps (Mapas)

Un **Map** se asemeja a un **List**, pero accedes a sus miembros no sólo con un número de índice sino también con una clave. Una clave puede ser una cadena o un número. Al igual que un **List**, un **Map** puede almacenar cualquier tipo de objeto.

```
Dim mapPerson As Map
mapPerson.Initialize
Dim photo As Bitmap
...
mapPerson.Put("name", "smith")
mapPerson.Put("age", 23)
mapPerson.Put("photo", photo)
```
Más detalles en la Sección Map.

Type. Definir tipos de variables

La palabra clave **Type** se utiliza para crear sus propios tipos o estructuras. Puede utilizar estos tipos para crear estructuras simples que agrupen algunos valores. Sin embargo, también se puede utilizar para crear colecciones más complejas. Defina un tipo con la palabra clave **Type**:

```
Sub Process_Globals
 Type Person( _
   LastName As String, FirstName As String, Age As Int, _
   Address As String, City As String _
 )
End Sub
```
Podemos declarar variables individuales o arrays de este tipo:
```
Dim CurrentUser As Person
Dim User(10) As Person
```
Para acceder a un elemento en particular, utilizamos el nombre de la variable y su nombre del campo de datos, separados por un punto:
```
CurrentUser.FirstName = "Wyken"
CurrentUser.LastName = "Seagrave"
```
Si la variable es un array, entonces el nombre es seguido por el índice deseado entre paréntesis:
```
User(1).LastName = "Seagrave"
```
Es posible asignar un elemento de un tipo determinado a otra variable del mismo tipo:
```
CurrentUser = User(1)
```

Declaración de Tipos

Un Tipo no puede ser privado. Los tipos deben declararse en Process_Globals.
Una vez declarado, está disponible en todas partes (similar a los módulos Class).

Tipos Recursivos

Es posible utilizar el tipo actual como tipo para uno de los campos de la variable.
```
Sub Process_Globals
 Type Element (NextElement As Element, Val As Int)
 Dim Head As Element 'declarar una variable de este tipo
 Dim Last As Element
End Sub
```
La capacidad de declarar tales tipos recursivos es muy potente. El ejemplo anterior podría utilizarse para una lista enlazada, como se explica en este tutorial en línea

Inicialización de un Tipo Recursivo

Antes de poder acceder a cualquiera de los campos de tipo en un tipo recursivo, se debe inicializar llamando a su método Initialize:
```
Head.Initialize
```

Nota: si su tipo sólo incluye campos y cadenas numéricas, entonces no hay necesidad de llamar a *Initialize* (aunque no hay nada malo en llamarlo).

Casting

Casting significa cambiar el tipo de un objeto. B4A cambia de tipo automáticamente según sea necesario. También convierte números en cadenas y viceversa automáticamente. A veces es necesario convertir explícitamente un objeto a un tipo específico.
Por ejemplo, puede tener un Gestor de eventos que necesite leer datos del objeto que generó el evento. Puede obtener una referencia a ese objeto usando la palabra clave Sender, pero para usar las propiedades de ese objeto, debe convertirlo al tipo correcto. Esto se puede hacer asignando el objeto a una variable del tipo requerido.

```
Sub Btn_Click
 ' Crear un objeto del tipo correcto para que podamos acceder a sus
propiedades
 Dim btn As Button
 ' Copia el Objeto que generó este evento.
 '  Esto cambiará su tipo a Button
 btn = Sender
 ' Ahora podemos acceder a sus propiedades
 btn.Color = Colors.RGB(Rnd(0, 255), Rnd(0, 255), Rnd(0, 255))
End Sub
```

Visibilidad de las Variables

Discutimos esta cuestión aquí.

Expresiones y Operadores

Una expresión es una combinación de valores, constantes, variables, operadores y funciones que se combinan utilizando operadores para producir un valor, por ejemplo:

```
2 + intAge
strName.Length - 1
```

Expresiones matemáticas

Los operadores matemáticos ("+", "-", etc.) deben ejecutarse en un orden determinado. Esto se llama precedencia. **La precedencia 1 es la más alta**. El Nivel de Precedencia se abrevia PL en la siguiente tabla:

Operador	Ejemplo	PL	Operación
`Power`	`Power(x,y)`	1	Power of, x^y
`Mod`	`x Mod y`	2	Módulo
`*`	`x * y`	2	Multiplicación
`/`	`x / y`	2	División
`+`	`x + y`	3	Suma
`-`	`x - y`	3	Resta

Así, por ejemplo, en la expresión 4 + 5 * 3 + 2, primero se evalúa la multiplicación, para obtener 4 + 15 + 2, con lo que se obtiene 21.
`Power` significa "Potencia", multiplicar un número por sí mismo varias veces, así que Power (2,3) significa 2 * 2 * 2.
`Mod` (abreviatura de modulo) devuelve el resto después de una división. Así, 11 Mod 4 es el resto de 11 / 4, por lo que devuelve 3.

Operadores Relacionales

Los operadores relacionales comparan dos valores y deciden si son iguales, si uno es más grande que el otro, etc. Estos operadores devuelven **`True`** o **`False`**.

Operador	Ejemplo	Devuelve **`True`** si
=	x = y	Los dos valores son iguales
<>	x <> y	Los dos valores no son iguales
>	x > y	el valor de la expresión izquierda es mayor que el de la derecha
<	x < y	el valor de la expresión izquierda es menor que el de la derecha
>=	x >= y	el valor de la expresión izquierda es mayor o igual que el de la derecha
<=	x <= y	el valor de la expresión izquierda es menor o igual que el de la derecha

Operadores Lógicos

Los operadores lógicos o "booleanos" se utilizan para determinar si una expresión es verdadera o falsa. Se usan típicamente en sentencias condicionales como **`If-Then`**. Devuelven valores de **`True`** o **`False`**

Operador	Ejemplo	Devuelve **`True`** si
Or	X Or Y	si X o Y es **`True`**, o si ambos son **`True`**
And	X And Y	**`True`** solo si X e Y son **`True`**
Not ( )	Not(X)	**`True`** solo si X es **`False`**

Expresiones Regulares

Las Expresiones Regulares se presentan varias veces en B4A, y proporcionan un método muy poderoso (aunque no muy fácil de programar) de especificar un patrón (a veces muy complejo) para buscar dentro de una cadena. Por ejemplo:
- para buscar un carácter de tabulación se utiliza la expresión "\t".
- para que coincida con cualquier carácter, utilice un punto "."
- para que coincida con uno o más caracteres se utilizará ".*"

Así, por ejemplo:
"c.t" coincidiría "cat" y "cot" pero no con "cart"
"c.*t" coincidiría "cat", "cot" y "cart" pero no con "ct"
Hay muchas reglas como estas.
También hay varios tipos de expresiones regulares. B4A utiliza el tipo de Java.

En B4A, las expresiones regulares pueden ser procesadas usando el objeto Regex y su Objeto Matcher asociado.
Las expresiones regulares se utilizan en los delimitadores de la Librería de Funciones de Cadena.
Hay un tutorial de B4A sobre expresiones regulares aquí.

Ejemplo de Constructores de Regex en tipo Java

La siguiente es una lista incompleta:

x	El carácter x
\\	El carácter de barra invertida
.	Cualquier carácter (puede o no coincidir con los caracteres de final de línea)
X*	X, cero o más veces
X?	X, una o ninguna vez
X+	X, una o más veces
[abc]	a, b, o c (clase simple)
[^abc]	Cualquier carácter excepto a, b, o c (negación)
[a-z]	a to z inclusive (range)
\d	Un dígito: [0-9]
\D	Un carácter que no sea un dígito: [^0-9]
\s	Un espacio en blanco: [\t\n\x0B\f\r]
\S	Un carácter que no sea un espacio en blanco.: [^\s]
\w	Un carácter alfanumérico: [a-zA-Z_0-9]
\W	Un carácter no alfanumérico: [^\w]
^	Principio de una cadena
$	Final de una cadena
\b	posición de una palabra entre espacios en blanco, puntuación o el inicio/final de una cadena.
\B	posición entre dos caracteres alfanuméricos o dos no-alfanuméricos

Ejemplo simple de uso

```
' comprobar que la fecha tiene formato nn-nn-nnnn
If Regex.IsMatch("\d\d-\d\d-\d\d\d\d", "26-12-16") Then
 Log ("Valida")
Else
 Log ("Invalida")
End If
' log muestra Invalid
```

Cómo obtener más ayuda con las expresiones regulares

Para un resumen en línea vea aquí. Para una lista completa de las construcciones de regex en Java vea aquí.
Puedes probar tus expresiones regulares B4A en línea aquí.
Si necesitas usar expresiones regulares de forma habitual, le recomiendo que invierta algo de tiempo en RegexBuddy. No sólo proporciona una herramienta para crear y probar expresiones regulares, sino que tiene tutoriales útiles (aunque no fáciles de asimilar) para explicar las partes más abstractas de la compleja sintaxis.

Sentencias Condicionales

En B4A se dispone de varias sententcias condicionales.

If – Then

Una comprobación simple que ejecuta una sola sentencia cuando la condición es **True**,

```
If intA = 20 Then intA = intA - 3
```

If-Then-End If

Si es necesario ejecutar varias sentencias cuando la evaluación es verdadera, es común escribirlas en líneas separadas y terminarlas con una sentencia **`End If`** :

```
If intA = 20 Then
 intA = intA - 3
 intB = intB + 3
End If
```

If-Then-Else-End If

Si se require ejecutar alguna sentencia cuando evaluación es falsa, se utiliza **`Else:`**

```
If intA = 20 Then
 intA = intA - 3
Else
 intA = intA + 1
End If
```

Tenga en cuenta que esto podría escribirse como:

```
If intA = 20 Then intA = intA - 3 Else intA = intA + 1
```

Pero para que el código sea más legible, es recomendable utilizar varias líneas de programa.

Explicación detallada de como funciona If-Then-Else-End If

Considere este caso general:

```
If test1 Then
 ' código1
Else If test2 Then
 ' código2
' más test son posibles
Else
 ' códigoN
End If
```

- Al llegar a la línea con la palabra clave **`If`**, se evalúa "test1". El test puede ser cualquier tipo de sentencia condicional con dos posibilidades: **`True`** o **`False`**.
- Si el resultado de la evaluación test1 es **`True`**, se ejecuta el "código1" hasta la línea con la palabra clave **`Else If`**, y luego la ejecución continúa en la línea siguiente a la palabra clave **`End If`**.
- Si el resultado del test1 es falso, se evalúa "test2".
- Se repite lo mismo.
- Si todas las evaluaciones de test son falsas, entonces se ejecuta el "códigoN" después de la palabra clave **`Else`**.

Diferencias entre B4A y Visual Basic

1. B4A usa **`Else If`** mientras que VB usa: **`ElseIf`**
2. La siguiente línea se interpreta de manera diferente en B4A y VB:

```
If b = 0 Then a = 0: c = 1
```

En B4A es equivalente a:

```
If b = 0 Then
 a = 0
End If
c = 1
```

Pero en VB es:

```
If b = 0 Then
 a = 0
 c = 1
End If
```

Select – Case

La estructura **`Select - Case`** le permite comparar una expresión de evaluación con otras expresiones y ejecutar diferentes secciones de código de acuerdo a las coincidencias con la expresión que se evalúa. Esto es similar al comando **`switch`** en C, PHP y otros lenguajes.

```
Select TestExpression
 Case ExpressionList1
   ' código1
 Case ExpressionList2
   ' código2
 Case Else
   ' código3
End Select
```

"TestExpression" es cualquier expresión o valor.
"ExpressionList1" es una lista de cualquier expresión o valor, separados por comas.
La estructura **`Select - Case`** funciona como se muestra a continuación:

- "TestExpression" se evalúa.
- Si uno de los elementos de la "ExpressionList1" coincide con "TestExpression", se ejecuta el "código1" y el control pasa a la línea que sigue a la palabra clave **`End Select`**.
- De lo contrario, si uno de los elementos de la "ExpressionList2" coincide con "TestExpression", se ejecuta " código2" y el control pasa a la línea que sigue a la palabra clave **`End Select`**.
- De lo contrario, si ninguna expresión coincide con "TestExpression", se ejecuta " código3" y el control continúa en la línea que sigue a la palabra clave **`End Select`**.

Nota: el tipo de cada valor en cada ExpressionList tiene que ser el mismo que el tipo de TestExpression. De lo contrario, se producirá un error de compilación o un error en tiempo de ejecución. Algunos ejemplos:

```
Dim intA As Int
intA  = Rnd(1,100)
Select intA
 Case 1, 2, 99
 ' código
 Case 5
 ' código
 Case Else
 ' código
End Select
```

Nota: si accidentalmente usa la misma expresión en dos Sentencias de Case, se produce un error de compilación. Algunos ejemplos más:

```
Dim intA, intB  As Int
intA = Rnd(1,100)
intB = Rnd(1,100)
Select intA + intB
Case 2,3,4,5
 Log("small")
Case Else
 Log("big")
End Select
'----------
Dim strCode As String
Select strCode
 Case "walk"
 ' código
 Case "run"
 ' código
 Case Else
 ' código
End Select
'----------
Sub Activity_Touch (Action As Int, X As Float, Y As Float)
 Select Action
  Case Activity.ACTION_DOWN
   ' código
  Case Activity.ACTION_MOVE
   ' código
  Case Activity.ACTION_UP
   ' código
 End Select
End Sub
```

Diferenciass entre B4A y Visual Basic:

- B4A usa **`Select`**, donde VB usa **`Select Case`**.
- B4A sólo permite una lista, por ejemplo: **`Case 1,2,3`**, donde VB también permite un rango como por ejemplo: **`Case 1 To 3`**

Estructuras de Bucle (CICLO!)

Basic dispone de varias estructuras para bucles: (CICLOS)

For – Next

En un bucle **`For-Next`**, el mismo código se ejecutará un número de veces controlado por una variable llamada "iterador". Por ejemplo:

```
For i = 1 To 9 Step 2
 ' Su código
Next
```

En este caso i es el iterador. Así es como se ejecuta el código:

- El Iterador **`i`** se establece en el primer valor **`1`** y se ejecutará **`su código`**.
- Cuando la ejecución llegue a **`Next`**, la ejecución volverá a la sentencia **`For`** e **`i`** se incrementará con el valor de **`Step 2`** a 1+2 o 3.

- Si **i** es menor o igual que el valor superior 9, entonces **su código** se ejecutará de nuevo.
- Se repetirá hasta que **i** sea mayor que el valor superior.
- El control entonces pasa la línea después de **Next**.

Así **su código** en el ejemplo anterior se ejecutará exactamente cinco veces, cuando **i** = 1,3,5,7 y 9.
Si la variable iterador i no fue declarada previamente, será de tipo Int.
Nota: los límites de los bucles (en el caso anterior, 1 y 9) pueden ser expresiones que dependen de variables. En ese caso, sólo se calcularán una vez, antes de la primera iteración.

El Valor Step

Nota: si se omite el valor de **Step**, entonces se asume que es 1, sin importar cuál sea el valor inicial del iterador. Así que:

```
For i = 1 To 10
```

Es lo mismo que

```
For i = 1 To 10 Step 1
```

La variable **Step** puede ser negativa:

```
For i = 10 To 6 Step -1
```

Iteradores No-enteros

Tenga en cuenta que el iterador (**i** en los ejemplos anteriores) se supone que es un número entero, a menos que se declare previamente. Pero si se declara correctamente, se puede utilizar cualquier valor numérico como iterador:

```
Dim i As Float
For i = 1.1 To 1.4 Step 0.1
' Su código
Next
```

Exit

Es posible salir de un bucle **For–Next** con la palabra clave **Exit**. Cuando la ejecución del código encuentra la palabra clave **Exit**, continúa en la línea después de **Next**. Lo siguiente mostrará desde 1 hasta 4 en el log:

```
For i = 1 To 10
 If i = 5 Then Exit
 Log (i)
Next
```

Continue

Si desea detener la ejecución de la iteración actual pero continuar con la siguiente, utilice **Continue**:

```
For i = 1 To 10
 If i = 5 Then Continue
 Log (i)
Next
```

Mostrará en el log 1-4 y 6-10, pero no 5.

Diferencias entre B4A y Visual Basic

- B4A usa **Next**, mientras VB usa **Next i**
- B4A usa **Exit**, VB usa **Exit For**

For-Each

For-Each es una variante del bucle **For-Next**. Mientras que **For-Next** se limita a usar un número para controlar el bucle, **For-Each** puede usar arrays, listas, mapas o cualquier otro "IterableList" que pueda crear.

Ejemplo:
```
Dim strName() As String = Array As String("a", "b", "c")
For Each name As String In strName
 Log (name)
Next
```
Cada valor de strName se asigna, a su vez, al nombre de la variable, por lo que el resultado es:
```
a
b
c
```
Un ejemplo de iteración con un **Map**:
```
Dim balances As Map
balances.Initialize
balances.Put("Fred", 123.45)
balances.Put("Tom", 543.21)
Dim value As Float
For Each Person As String In balances.Keys
 value = balances.Get(Person)
 Log (Person & " tiene de saldo " & value)
Next
```
También puede obtener los valores en un map como una lista iterable:
```
For Each v As Int In map1.Values
  Log(v)
Next
```
Los objetos views de una actividad son una lista iterable:
```
For Each vw As View In Activity
 ' verificar su tipo
 If vw Is Button Then
 ' necesita un objeto con el tipo correcto para
 ' poder acceder a las propiedades
Dim btn As Button
 ' hacer copia de la vista original
 btn = vw
 Log (btn.Text)
 End If
Next
```
También puede iterar todas las vistas pertenecientes a un panel:
```
For Each vw As View In pnlMain.GetAllViewsRecursive
 vw.Color = Colors.RGB(Rnd(0,255), Rnd(0,255), Rnd(0,255))
Next
```
Do-While
Puede realizar un bucle mientras una determinada condición es **True**. Por ejemplo, esto disminuirá aleatoriamente un número que comienza con 10000 y registrará el resultado mientras sea mayor que 0:
```
Dim i As Int = 10000
Do While i > 0
 ' disminuye aleatoriamente i
 i = i - Rnd(20, 200)
 Log (i)
Loop
```

Do-While es útil si conoce la condición de inicio cuando comienza el bucle. Por ejemplo, cuando lee un archivo de texto. Lo siguiente lee un archivo de texto y lo usa como texto para una **Label**:

```
Dim lbl As Label
Dim strLines As String
Dim tr As TextReader
tr.Initialize(File.OpenInput(File.DirAssets, "test.txt"))
lbl.Initialize("")

strLines = tr.ReadLine
Do While strLines <> Null
 lbl.Text = lbl.Text & CRLF & strLines
 strLines = tr.ReadLine
Loop
tr.Close
Activity.AddView(lbl, 10dip, 10dip, 100dip, 100dip)
```

Do-While podría no ser ejecutado

Nota: en algunos lenguajes, como C, la sintaxis hace que un bucle do-while se ejecute siempre al menos una vez, porque la condición que controla el bucle no se prueba hasta **después** de ejecutar el código. Por ejemplo:

```
// Ejemplo de código en C
do {
 /* "¡Hola mundo!" se muestra al menos una vez
 Incluso aunque la condición sea falsa */
 printf( "¡Hola mundo!\n" );
} while ( x != 0 );
```

Por otra parte, en B4A, la condición se comprueba **antes** de ejecutar el bucle. Por ejemplo, el siguiente código B4A NO producirá salidas en el log:

```
Dim i As Int = 0
Do While i > 3
 Log (i)
 i = i - 1
Loop
```

Do-Until

A veces, no sabemos el valor inicial que queremos utilizar. Sólo sabemos cuándo queremos detener el bucle. En este caso, usamos el bucle **Do Until**:

```
i = Rnd(20, 200)
Do Until i <= 0
' disminuye aleatoriamente i
 i = i - Rnd(20, 200)
 Log (i)
Loop
```

Salir de un Bucle

Es posible salir de cualquiera de estas estructuras Do-Loop utilizando la palabra clave **Exit**.

```
Dim i As Int = 10000
Dim magicNumber As Int = 1234
Do While i > 0
' disminuye aleatoriamente i
 i = i - Rnd(20, 200)
 Log (i)
 If i = magicNumber Then
  Log ("Alcanzado el número mágico para terminar el bucle")
  Exit
 Else
  Log (i)
 End If
Loop
```

Diferencias entre B4A y Visual Basic

En Visual Basic, el tipo de bucle se especifica después de **`Exit`**, por ejemplo, **`Exit Loop`**
En B4A, solo se utiliza **`Exit`**.
Visual Basic también acepta los siguientes bucles:

```
Do ... Loop While test
Do ... Loop Until test
```

Que NO son soportados en B4A.

Subs

Una subrutina ("**`Sub`** ") es un fragmento de código. Tiene un nombre propio y una visibilidad definida (como se discutió anteriormente). En B4A, una subrutina se llama **`Sub`**, y es equivalente a procedimientos, funciones, métodos y subs en otros lenguajes de programación.
Puede ayudar a que su código sea más legible y más robusto utilizando Subs para encapsularlo en unidades lógicas y podrá probar cada **`Sub`** por separado del resto del código No es recomendable que los Subs sean demasiado largos, ya que tienden a ser menos legibles.

Declarando un Sub

Un Sub se declara de la siguiente manera:

```
Sub CalcInterest(Capital As Double, Rate As Double) As Double
 Return Capital * Rate / 100
End Sub
```

Comienza con la palabra clave **`Sub`**, seguido por el nombre del Sub **`CalcInterest`**, seguido por una lista de parámetros entre paréntesis (**`Capital As Double, Rate As Double`**), seguido por el tipo de valor a devolver **`Double`**. Esto es seguido por el código que el sub ejecuta. El sub termina con las palabras clave **`End Sub`**.
No hay límite en el número de subs que puede añadir a su programa, pero no se le permite tener dos subs con el mismo nombre en el mismo módulo.
Los subs se declaran siempre en el nivel superior del módulo. Es decir, NO SE PUEDEN anidar dos sub uno dentro del otro.

Nombre de un Sub

Para un Sub, puede usar cualquier nombre que sea legal para una variable. Es muy recomendable nombrar al Sub con un nombre significativo para que su código se autodocumente.

LLamar a un Sub

Cuando desee ejecutar un Sub en el mismo módulo, simplemente utilice el nombre del Sub.

```
Sub Activity_Resume
 doSomething
End Sub

Sub doSomething
 ' el código va aquí
End Sub
```

LLamar a un Sub desde otro Módulo

Como discutiremos más extensamente en la sección Visibilidad de Subrutinas, los subs públicos dentro de una clase o módulo de código pueden ser llamados directamente por llamadas en cualquier otro tipo de módulo como **`CodeModule.mySub`** o **`myClassInstance.mySub`**.

Los Subs declarados en los módulos de Actividad y Servicio, por otra parte, **no pueden** llamarse directamente como se muestra arriba, pero pueden llamarse por cualquier otro módulo usando las funciones CallSub o CallSubDelayed, siempre y cuando la actividad no esté en pausa o el servicio haya comenzado.

Parámetros

Los parámetros de entrada se pueden transmitidos al Sub. Esto le permite hacer que el sub haga diferentes cosas dependiendo de sus parámetros de entrada. La lista de parámetros se incluye entre paréntesis y sus tipos son obligatorios:

```
Sub CalcInterest(Capital As Double, Rate As Double) As Double
 Return Capital * Rate / 100
End Sub
```

Para invocar un sub que necesita parámetros, añada los parámetros a la llamada:

```
Interest = CalcInterest(1234.56, 3.2)
```

Si un Sub no necesita parámetros, entonces los paréntesis no son necesarios cuando se define o cuando llama al Sub:

```
i = getRate
...
Sub getRate
 Return 3
End Sub
```

Valor de retorno

Un sub puede devolver un valor. Este puede ser cualquier objeto. La devolución de un valor se realiza con la palabra clave **`Return`**. El tipo de valor de retorno se define después de la lista de parámetros. Así que lo siguiente devolverá un **`Double`**

```
Sub CalcInterest(Capital As Int, Rate As Int) As Double
```

Creando Tooltips para Subs

Puede crear un tooltip para documentar lo que hace un Sub. Consulte Comentarios como Documentación para obtener más información.

Subs Reanudables

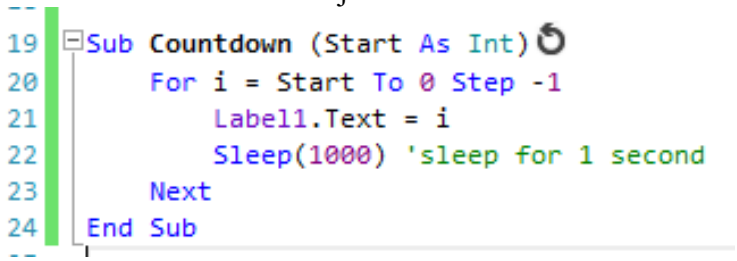
Se pueden pausar

Los subs reanudables simplifican drásticamente el manejo de tareas asíncronas.

Un Sub reanudable es aquel que puede ser pausado sin pausar el hilo de ejecución, y luego ser reanudado. El programa no espera a que el sub reanudable continúe. El resto de eventos se evocan y ejecutan como de costumbre.

Contiene Sleep o Wait For

Lo que hace que un Sub resumeable es que contiene una o más llamadas a **`Sleep`** o **`Wait For`**. El IDE muestra una flecha circular junto a una declaración del sub reanudable:



```
19  Sub Countdown (Start As Int)
20      For i = Start To 0 Step -1
21          Label1.Text = i
22          Sleep(1000) 'sleep for 1 second
23      Next
24  End Sub
```

El Sub reanudable está en pausa

Cada vez que se llama a **`Sleep`** o **`Wait For`**, el sub reanudable se detiene y el control se devuelve al que llama. El gestor de eventos interno se encarga de reanudar el sub pausado cuando se produce el evento. Si el evento nunca se produce, el sub nunca se reanudará. El programa seguirá funcionando correctamente.

Tipo ResumableSub

Debido a que el Sub reanudable está en pausa y el control ya ha sido devuelto al Sub de llamada, no puede utilizar simplemente **`Return`** para devolver un valor al Sub de llamada.

Por ejemplo el siguiente código:

```
Sub Button1_Click
 Sum(1, 2)
 Log("después de la suma")
End Sub

Sub Sum(a As Int, b As Int)
 Sleep(100) 'esto causará que el flujo de código regrese al padre
 Log(a + b)
End Sub
```

Dará el resultado:

```
después de la suma
3
```

La solución es utilizar un subconjunto con el tipo de **`ResumableSub`**

```
Sub Button1_Click
 Wait For(Sum(1, 2)) Complete (Result As Int)
 Log("resultado: " & Result)
 Log("después de la suma ")
End Sub
Sub Sum(a As Int, b As Int) As ResumableSub
 Sleep(100)
 Log(a + b)
 Return a + b
End Sub
```

Dará el resultado:

3
resultado: 3
después de la suma

Nota: puede declarar una variable como **ResumableSub**. Así que **Button1_Click** podría escribirse como:

```
Sub Button1_Click
 Dim rs As ResumableSub = Sum(1, 2)
 Wait For(rs) Complete (Result As Int)
 Log("resultado: " & Result)
 Log("después de la suma ")
End Sub
```

Nota: también es posible utilizar **CallSubDelayed2** para producir un resultado similar, pero lo anterior es una mejor solución.
Hay dos maneras de hacer que una llamada se reanude: mediante el uso de **Sleep** o **Wait For**.

Sleep

Puede pausar un sub durante un número específico de milisegundos utilizando la palabra clave **Sleep**:

```
  Sleep(1000)
```

Una llamada a **Sleep** o **Wait For** en un sub reanudable hace que el flujo de código vuelva al padre. Por lo tanto, no es posible que un sub reanudable simplemente devuelva un valor al final del sub.
Consulte el apartado **Tipo ResumableSub** anteriormente para obtener información sobre la forma en que un sub reanudable puede devolver un valor al sub que lo ha llamado.

Nota: usar **Wait For** para esperar un evento es mejor que llamar a **Sleep** en un bucle. Véase más adelante.

Sleep como Sustituto de DoEvents

Sleep(0) es un sustituto útil de la palabra clave ya obsoleta **DoEvents**. Sirve para lo mismo, permitir la actualización de la interfaz de usuario.

```
  Sub ShowProgress
      Dim iMax As Int = 100000
      Dim i As Int
      For i = 1 To iMax
            If i Mod 1000 = 0 Then
                  progBar.Progress = i * 100 / iMax
                  Sleep(0) ' actualizar pantalla
            End If
      Next
  End Sub
```

Múltiple Instancias de Sub Reanudables

Cada llamada a un sub reanudable crea una instancia diferente que no se ve afectada por otras llamadas.

```
  Sub btn_Action
     Dim b As Button = Sender
     For i = 10 To 0 Step - 1
       b.Text = i
       Sleep(100)
     Next
     b.Text = "Takeoff!"
  End Sub
```

Wait For

B4A es orientado a eventos (véase la siguiente sección). Las tareas asíncronas se ejecutan en segundo plano y generan un evento cuando la tarea se completa. Con la palabra clave **Wait For** puede manejar el evento dentro del sub actual:

```
Wait For activity_keypress (keycode As Int)
Msgbox ( keycode, "You clicked")
```

Una llamada a **Wait For** o **Sleep** en un sub reanudable hace que el flujo de código vuelva al padre. Por lo tanto, no es posible que un sub reanudable simplemente devuelva un valor al final del mismo. Consulte el apartado **Tipo ResumableSub** más arriba para obtener información sobre la forma en que un sub reanudable puede devolver un valor al sub que realiza la llamada.

Identificación del Evento (Event Signature)

En el primer ejemplo anterior, el parámetro **activity_keypress (keycode As Int)** es una firma de evento. La estructura de una firma de evento variará dependiendo de qué evento se espera.
Una firma de evento más compleja (utilizando el objeto FTP en la Librería adicional de Red) podría ser:

```
Wait For FTP_ListCompleted (ServerPath As String, Success As Boolean,
Folders() As FTPEntry, Files() As FTPEntry)
```

Sub está en Pausa

Si más tarde se llama a **Wait For** con el mismo evento, la nueva instancia del sub reemplazará a la anterior. Vea la siguiente sección para ver una forma de permitir que un sub reanudable maneje eventos de diferentes emisores.

Esperar por un evento específico

Es posible que necesitemos llamar varias veces a un sub que contenga **Wait For** para procesar los eventos generados por varios emisores. En este caso, el sub debe ser capaz de identificar qué objeto provocó el evento. Para ello debemos utilizar un parámetro opcional que especifique el objeto emisor:

```
Wait For (Sender) <Event Signature>
```

Nota: los paréntesis se requieren alrededor del emisor.
El siguiente ejemplo usa la librería OkHttpUtils2 para descargar dos imágenes en dos **ImageViews**:

```
Sub Activity_Create(FirstTime As Boolean)
  Activity.LoadLayout("Layout1")
  DownloadImage("https://www.b4x.com/images3/android.png", ImageView1)
  DownloadImage("https://www.b4x.com/images3/apple.png", ImageView2)
End Sub

Sub DownloadImage(Link As String, iv As ImageView)
     Dim job As HttpJob
     job.Initialize("", Me)
     job.Download(Link)
     Wait For (job) JobDone(job As HttpJob)
     If job.Success Then
          iv.Bitmap = job.GetBitmap
     End If
     job.Release
End Sub
```

Más Información

Más información sobre los subs reanudables en línea aquí.

Eventos

Los objetos B4A pueden reaccionar a los eventos. Estas pueden ser acciones del usuario o eventos generados por el sistema. El número y el tipo de eventos que un objeto puede provocar depende del tipo de objeto.

Eventos básicos de los Objetos

Muchos objetos básicos de B4A generan eventos. Ejemplos son **`Animation`**, **`AudioRecordApp`**, **`Camera`**, **`DayDream`**, **`GameView`**, **`GPS`**, **`HTTPClient`**, **`IME`**, **`MediaPlayerStream`**, **`Timer`**, etc. Consulte la documentación de cada uno de estos objetos para descubrir qué eventos pueden generar.

Respuesta a un Evento

Para responder a un evento, se debe escribir una subrutina con el nombre correcto. Debe escribir un Sub con el nombre del objeto que está generando el evento, seguido de un guión bajo seguido del nombre del evento. Por ejemplo:

```
Sub Timer1_Tick
```

`Timer1` es el nombre del objeto que está invocando el evento. Este nombre se decide al inicializar el objeto, por ejemplo

```
Timer1.Initialize("Timer1", 1000)
```

La parte **`Tick`** del nombre de la subrutina es el nombre del evento. Esto viene determinado por el propio objeto. Es necesario consultar la documentación del objeto para descubrir qué eventos puede provocar. Algunos objetos pueden producir múltiples eventos.
Debe unir estas dos partes del nombre junto con un guión bajo _, por ejemplo **`Timer1_Tick`**.
Nota: el IDE proporciona una forma sencilla de Autocompletar Subrutinas de Eventos.

Ejemplo

Para dar un ejemplo concreto, un **`Timer`** se ejecutará en segundo plano hasta que haya terminado su tarea, y luego generará un evento (en este caso **`Tick`**) al que su código necesita responder. Por ejemplo, Timer1_Tick como en el siguiente ejemplo:

```
Sub Process_Globals
 ' Declarar aquí para no obtener múltiples timers cuando la actividad
es recreada
 Dim Timer1 As Timer
End Sub

Sub Globals
End Sub

Sub Activity_Create(FirstTime As Boolean)
 ' hacer que el temporizador dure 1000 milisegundos
 Timer1.Initialize("Timer1", 1000)
 ' Iniciar el temporizador
 Timer1.Enabled = True
End Sub
```

```
Sub Timer1_Tick
 ' el temporizador ha terminado
 Log ("temporizador ha terminado ")
End Sub
```

Gestor de Eventos Compartido

Puede usar un solo Sub para manejar los eventos de muchos objetos. Por ejemplo, puede tener varios botones, todos los cuales realizan una función similar, por lo que sólo necesita un único gestor de eventos. Puede determinar qué objeto generó el evento utilizando la palabra clave **Sender**. Lo siguiente produce una columna de botones etiquetados Test 1 a Test 7, todos los cuales comparten el mismo sub gestor **Buttons_Click**:

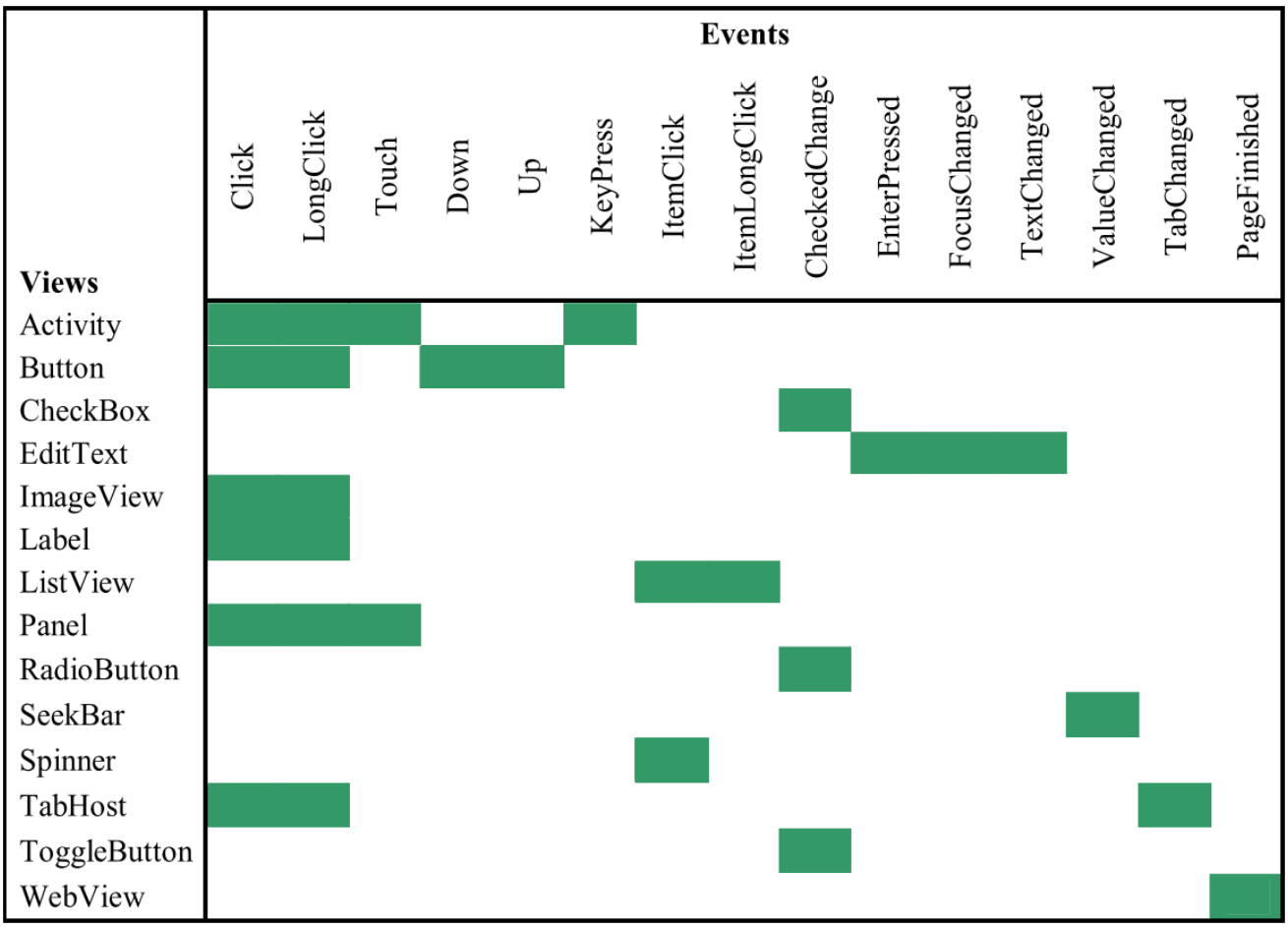
```
Sub Globals
 Dim b1, b2, b3, b4, b5, b6, b7 As Button
 Dim Buttons() As Button
End Sub

Sub Activity_Create(FirstTime As Boolean)
 ' índice para manejar los botones
 Dim i As Int
 Buttons = Array As Button(b1, b2, b3, b4, b5, b6, b7)

 For i = 0 To 6
  ' todos los botones comparten el gestor de evento Buttons_Click
  Buttons(i).Initialize("Buttons")
  ' usar el índice para posicionar correctamente los botones
  Activity.AddView(Buttons(i), 10dip, 10dip + i * 60dip, _
   150dip, 50dip)
  ' añadir etiqueta para poder identificar qué botón es
  Buttons(i).Tag = i + 1
  Buttons(i).Text = "Test " & (i + 1)
 Next
End Sub

Sub Buttons_Click
 ' gestor de eventos para todos los botones
 Dim btn As Button
 btn = Sender
 Activity.Title = "Botón " & btn.Tag & " pulsado"
End Sub
```

View Events

Muchos eventos son generados por Views los cuales son procesados por su código de la misma manera que los Eventos de Objetos Básicos. El Diseñador es capaz de generar los esqueletos de subs, tales como:

```
Sub btnTest_Click
 ' añade tu código aquí
End Sub
```

Aquí hay un resumen de los eventos para diferentes vistas:



Eventos de vista más comunes

Los eventos más comunes son los siguientes. Tenga en cuenta que los eventos admitidos varían según el tipo de vista:

Click

Evento que se produce cuando el usuario hace clic en la vista. Ejemplo:

```
Sub Button1_Click
  ' Su código
End Sub
```

LongClick

Evento que se produce cuando el usuario hace clic en la vista y la mantiene pulsada durante aproximadamente un segundo. Ejemplo:

```
Sub Button1_LongClick
' Su código
End Sub
```

Touch(Action As Int, X As Float, Y As Float)

Evento que se produce cuando el usuario toca la pantalla.
Se gestionan tres acciones diferentes:

- Activity.Action_DOWN: el usuario toca la pantalla.
- Activity.Action_MOVE: el usuario mueve el dedo sin levanter el contacto de la pantalla.
- Activity.Action_UP: el usuario deja de tocar la pantalla.

Se indican las coordenadas X e Y de la posición del dedo. Ejemplo:

```
Sub Activity_Touch (Action As Int, X As Float, Y As Float)
 Select Action
 Case Activity.ACTION_DOWN
  ' Su código para la acción DOWN
 Case Activity.ACTION_MOVE
  ' Su código para la acción MOVE
 Case Activity.ACTION_UP
  ' Su código para la acción UP
 End Select
End Sub
```

CheckChanged (Checked As Boolean)

Evento que se produce cuando el usuario pulsa sobre un **CheckBox** o un **RadioButton**.
Checked es **True** si la vista está marcada o **False** si no lo está.
Ejemplo:

```
Sub CheckBox1_CheckedChange(Checked As Boolean)
 If Checked = True Then
  ' Su código si está marcado CheckBox1
 Else
  ' Su código si CheckBox1 no está marcado
 End If
End Sub
```

KeyPress (KeyCode As Int) As Boolean

Este evento (que sólo pertenece al objeto **Activity**) se produce cuando el usuario pulsa una tecla física o virtual (excepto la tecla Home, que llama **Activity_Pause**).
KeyCode es el código de la tecla pulsada. Puede obtener una lista de estos en el IDE escribiendo **KeyCodes** y un punto, o en este libro aquí.
Su evento **KeyPress** debe devolver **True**, en cuyo caso el evento se consume y nunca llega al sistema operativo, o **False**, en cuyo caso el evento se pasa al sistema para acciones posteriores.
Ejemplo:

```
Sub Activity_KeyPress(KeyCode As Int) As Boolean
 ' Confirmar que el usuario desea salir si pulsa la tecla de retorno
 Dim Answ As Int
 Dim Txt As String

 ' Compruebe si KeyCode es BackKey
 If KeyCode = KeyCodes.KEYCODE_BACK Then
  ' Confirmar que el usuario desea salir
  Txt = "¿Realmente quieres salir del programa?"
  Answ = Msgbox2(Txt, "A T E N C I Ó N", "Si", "", "No", Null)
  If Answ = DialogResponse.POSITIVE Then
   ' El usuario desea salir
   Return False
  Else
   ' No salir
   Return True
  End If
 End If
End Sub
```

Control de Errores

Errores en Tiempo de Ejecución

Algunos errores son capturados por el compilador, pero otros errores más sutiles sólo son descubiertos cuando el código se ejecuta. Este "error en tiempo de ejecución" se produce en el siguiente ejemplo:

```
Dim str As String
Dim i As Int
str = "hello"
i = str
```

La última línea produce un error de ejecución porque Java (que es lo que usa Android) no puede convertir una cadena no numérica en un número.

Excepciones

Cuando se produce un error en tiempo de ejecución, se genera una Excepción en lenguaje Java. Puedes añadir código Try-Catch a tu aplicación para gestionar las Excepciones. Si no ha añadido este código, cuando se produce una Excepción el programa se detiene y se muestra un error en el dispositivo o emulador, como se describe a continuación.

Excepciones en tiempo de ejecución no capturadas

Si se produce un error en tiempo de ejecución fuera de un bloque Try-Catch, lo que vea el usuario dependerá de cómo se haya distribuido la aplicación.

Gestión de excepciones en tiempo de ejecución no capturadas por defecto

Si distribuyes a través de Google Play y su aplicación genera un error que no se captura internamente, se pedirá al usuario que envíe un informe de error. Esto sucede automáticamente. Si el usuario acepta enviar el informe, puede ver el resultado en Google Play Developer Console.
Si distribuye su aplicación directamente con un archivo apk y su aplicación genera un error que no se captura internamente, y el error ocurre dentro de una actividad, por defecto el usuario verá un informe de error que le preguntará si desea continuar, como se muestra a continuación.

Pero el usuario no sabe si la aplicación puede continuar correctamente después de la excepción no capturada. El diálogo también es inconsistente, ya que sólo aparece cuando el error ocurre dentro de una Actividad, pero no dentro de cualquier otro tipo de módulo.

Mejor Gestión de Excepciones en tiempo de ejecución no capturado

Por lo tanto, es preferible anular este comportamiento y manejar el error usted mismo, por ejemplo, enviando el error por correo electrónico.
Esto requiere que su aplicación contenga un Servicio Starter. Por defecto, el Servicio Starter contiene un Sub llamado **Application_Error**:

```
'Devuelve true para permitir que el manejador de excepciones por
defecto del sistema operativo gestione la excepción no capturada.
Sub Application_Error (Error As Exception, StackTrace As String) As
Boolean
    Return True
End Sub
```

Puede usar este Sub para, por ejemplo, capturar el log y enviarlo por correo electrónico o usar HttpUtils2 para enviar el StackTrace (volcado de la pila de llamadas) a su servidor y luego matar el proceso en el evento JobDone.
Aquí se muestra un ejemplo de envío de los logs por correo electrónico.
Observe lo siguiente
Si devuelve True desde **Sub Application_Error** entonces se llama al controlador de excepciones por defecto del sistema operativo. El resultado es que la aplicación se detendrá y el informe del fallo se enviará a Google Play (si el usuario lo permite). Probablemente, esta sea la mejor manera de gestionar la mayoría de los errores.
Si devuelve False, no se llamará al controlador de excepciones predeterminado y la aplicación seguirá ejecutándose.
Sub Application_Error sólo se llamará en modo Release. En el modo Debug el programa mostrará el mensaje de error en los logs y terminará.
Los errores que ocurran al iniciar la aplicación, antes de que el servicio Starter esté listo, no son capturados. El controlador de excepciones por defecto del SO manejará esos errores.
El servicio Starter debe estar en marcha para que este sub se lance. Estará funcionando a menos que lo detenga explícitamente.

Try-Catch

B4A proporciona un mecanismo para gestionar los errores de ejecución, denominado bloque **Try-Catch**.
Ejemplo:

```
Try
 ' bloque de sentencias
Catch
 Log(LastException.Message)
 ' gestionar el problema si es necesario
End Try
```

Ahora cuando la Excepción ocurre en el bloque **Try**, el control se mueve al bloque **Catch**. Su programa puede tomar medidas para gestionar el problema.

Cuando utilizar un Try-Catch

Try-Catch no debe usarse para protegerse de errores de programación. Debe asegurarse de que su código es lógica y sintácticamente correcto probándolo antes de su distribución.
Try-Catch sólo debe utilizarse cuando exista un problema que no pueda controlar. Por ejemplo, cuando analiza un archivo descargado, el propio archivo puede tener problemas. O cuando se intenta actualizar una base de datos utilizando una Transacción y hay un problema. Por ejemplo:

```
SQL.BeginTransaction
Try
 'bloque de sentencias
 For i = 1 To 10
 SQL.ExecNonQuery2("INSERT INTO demo VALUES (?,?)", Array As Object(i,
"Tom Brown"))
 Next
 SQL.TransactionSuccessful
Catch
 Log(LastException.Message) 'no se realizan los cambios
End Try
SQL.EndTransaction
```

Try-Catch se utiliza fundamentalmente durante el desarrollo.
Nota: si se detecta un error en medio de una subrutina muy larga, no se puede realizar una corrección y, a continuación, retroceder y reanudar la ejecución donde se produjo el error. Sólo se ejecuta el código en el bloque **Catch**.

Manipulación de cadenas

B4A permite manipulaciones de cadenas como otros lenguajes Basic, pero con algunas diferencias. Estas manipulaciones se pueden hacer directamente en la propia cadena.
Ejemplo:

```
strTxt = "123,234,45,23"
strTxt = strTxt.Replace(",", ";")
```

Resultado: `123;234;45;23`

Cadenas Mutables

La manipulación repetitiva de las cadenas puede ser muy lenta. Dado que son inmutables, se debe crear una nueva cadena cada vez que se desee modificarla. Si está haciendo una gran cantidad de manipulación de cadenas, debería considerar el uso de StringBuilder.

Las funciones de Cadenas

Aquí enumeramos las funciones de cadena. Para más detalles, ver más abajo.
CharAt(Index)
.... Devuelve el carácter en la posición dada por **Index**, donde el primer carácter es 0.
CompareTo(Other)
.... Compara lexicográficamente la cadena con la cadena **Other**
Contains(SearchFor)
.... Devuelve **True** si la cadena contiene la cadena dada en **SearchFor**.
EndsWith(Suffix)
.... Devuelve **True** si la cadena finaliza con la subcadena dada en **Suffix**.
EqualsIgnoreCase(Other)
.... Devuelve **True** si ambas cadenas son iguales ignorando mayúsculas y minúsculas. Ejemplo:

```
If firstString.EqualsIgnoreCase("Abc") Then
```

GetBytes(Charset)
.... Codifica la cadena **Charset** en un nuevo array de bytes.
IndexOf(SearchFor)
.... Devuelve el índice de la primera aparición de **SearchFor** en la cadena, o **-1** si no se encuentra.

IndexOf2(SearchFor, Index)
.... Devuelve el índice de la primera aparición de **`SearchFor`** en la cadena, o **`-1`** si no se encuentra. Comienza la búsqueda desde el **`Index`** dado.
LastIndexOf(SearchFor)
.... Devuelve el índice de la primera aparición de **`SearchFor`** en la cadena, o **`-1`** si no se encuentra. Comienza la búsqueda desde el final de la cadena.
Length
.... Devuelve el número de caracteres en la cadena.
Replace(Target, Replacement)
.... Devuelve una nueva cadena, como resultado de la sustitución de todas las ocurrencias de Target por Replacement.
StartsWith(Prefix)
.... Devuelve **`True`** si esta cadena comienza con el **`Prefix`** dado.
Substring(BeginIndex)
.... Devuelve una nueva cadena que es una subcadena de la cadena original. La nueva cadena incluirá el carácter en **`BeginIndex`** y se extenderá hasta el final de la cadena.
Substring2(BeginIndex,EndIndex)
.... Devuelve una nueva cadena que es una subcadena de la cadena original. La nueva cadena incluirá el carácter en **`BeginIndex`** y se extenderá al carácter anterior a **`EndIndex`**.
ToLowerCase
.... Devuelve una nueva cadena que es el resultado de convertir a minúsculas todos los caracteres.
ToUpperCase
.... Devuelve una nueva cadena que es el resultado de convertir a mayúsculas todos los caracteres.
Trim
.... Devuelve una copia de la cadena original sin los espacios en blanco iniciales y finales.

Formateado de números

Los números se pueden mostrar como cadenas con diferentes formatos. Hay dos palabras clave: **`NumberFormat`** y **`NumberFormat2`**.

NumberFormat (Number As Double, MinimumIntegers As Int, MaximumFractions As Int)
Pinche el enlace para ver el significado de los argumentos. Ejemplos:

```
NumberFormat(12345.6789, 0, 2)
' produce 12,345.68
NumberFormat(1, 3, 0)
' produce 001
NumberFormat(Value, 3, 0)
' se pueden usar variables.
NumberFormat(Value + 10, 3, 0)
' se pueden usar operaciones aritméticas.
NumberFormat((lblscore.Text + 10), 0, 0)
' paréntesis necesarios Si una variable Es una cadena.
```

NumberFormat2(Number As Double, MinimumIntegers As Int, MaximumFractions As Int, MinimumFractions As Int, GroupingUsed As Boolean)
Pinche el enlace para ver el significado de los argumentos. Ejemplo:

```
NumberFormat2(12345.67, 0, 3, 3, True)
' Esto producirá "12,345.670".
```

VB6 versus B4A

Hay algunas diferencias entre B4A y Visual Basic de Microsoft. El siguiente análisis de las diferencias entre B4A y Visual Basic 6 se ha extraído del trabajo de nfordbscndrd, miembro del foro de B4A. Destaca algunas de las diferencias entre los dos IDEs y sus lenguajes Puede ser útil para desarrolladores experimentados de VB6.

Controless vs. Vistas

Los objetos a los que B4A llama Vistas (Views : botones, texto de edición, etiquetas, etc.) se llaman Controles en Visual Basic.
En la ventana de código VB6, la lista desplegable de la parte superior izquierda contiene todos los controles que ha colocado en el formulario actual y la lista de la derecha contiene todos los eventos para cada control. El equivalente en B4A se puede encontrar haciendo clic en el menú Diseñador[Herramientas > Generar miembros]. Una vez que haya creado Subs en la ventana de codificación del programa, la ventana de Módulos listará cada uno de los Subs en el módulo actual, debajo de la lista de todos los módulos.
En B4A, se empieza escribiendo "Sub" seguido de un espacio. El IDE le pedirá más detalles. Esto se describe en la sección Autocompletado de Subroutines de Eventos.
En VB6, puede nno poner ". Text" después de los nombres de los controles (Propiedad por defecto) cuando le asigna una cadena, pero esto no está permitido en B4A.

Dim

VB6: Dim name(n) le dará n+1 elementos con índice 0 a n. Por ejemplo, **`Dim strName(12)`** le dará 13 cadenas.
B4A: Dim name(n) le dará n elementos con índice 0 a n-1. Esto puede ser confuso ya que, para **`Dim strName(12)`**, el último elemento es realmente **`strName(11)`**.

ReDim

VB6: ReDim name(n) no existe en B4A, donde simplemente usaría otro **`Dim name(n)`**. Tampoco existe VB6 "ReDim Preserve". Si necesita esto, sería mejor usar una lista o mapa.

Operaciones Boleanas

Supongamos que lo siguiente se ha declarado en cualquiera de los dos lenguajes:

```
Dim i Int
Dim b as Boolean
```

Not

VB6 no requiere paréntesis: "**`If Not b Then`**"
En B4A, se requieren paréntesis: **`If Not(b) Then`**

Uso de números enteros como booleanos

En VB6, un entero que sea igual a cero se considera igual que un Booleano FALSE; cualquier cosa que no sea cero es TRUE. Por ejemplo: " **`If i Then`**"
En B4A, NO SE PUEDE utilizar un valor booleano en una función matemática. En su lugar, debe probar el valor de una variable, por ejemplo:

```
If i > 0 Then
```

Global Const

B4A no tiene función Global Const.
En VB6 puedes poner

```
Global Const x=1
```

En B4A pones
Sub Globals

```
Dim x as Int = 1
```

Sin embargo, x no es una constante. Su valor se puede cambiar.

Estructuras de repetición

For…Next

VB6: For i...Next i
B4A: For i…Next

Loops, If-Then, Select Case

VB6: Loop…Until, Loop…While
Esta estructura no está permitida en B4A. Sin embargo, puede utilizar el formato alternativo:
B4A: Do While…Loop, Do Until… Loop
Ver Do-While y Do-Until para más detalles.

Exit

VB6: Exit Do/For
B4A: Exit

ElseIf/EndIf

VB6: ElseIf/EndIf
B4A: Else If/End If

Colores

En VB6, los colores tienen nombres como "vbRed". En B4A, se utiliza el objeto Colores, por ejemplo:
`Colors.Red`

Subroutinas

Declaración de Sub

VB6: Sub SubName()
B4A: Sub SubName() As Int/String/etc.

LLamando una Sub

VB6: SubName x, y
B4A: SubName(x, y)

Funciones

Las funciones no existen en B4A. En su lugar, cualquier **`Sub`** se puede usar como una Función añadiendo un tipo de variable.

VB6: Function FName() As Int
B4A: **`Sub FName() As Int`**
Si no se devuelve valor con **`Return`**, entonces se devuelve cero, **`False`** o "" (cadena vacía).

Exit Sub

Exit Sub no existe en B4A. Utilice Return en su lugar.
VB6: Exit Sub / Exit Function
B4A: Return / Return [value]

DoEvents

Mientras que **`DoEvents`** existe en B4A, llamar a DoEvents en un bucle consume muchos recursos y demasiada energía de la batería porque Android nunca volverá al "bucle inactivo" principal donde se ejecutan las medidas de ahorro de energía del hardware. Además, **`DoEvents`** no permite que el sistema procese correctamente todos los mensajes en espera. En resumen, **siempre que sea possible debe evitarse en dispositivos móviles utilizar bucles durante periodos largos de tiempo**.

Format

VB6: Format()
B4A: NumberFormat & NumberFormat2

InputBox

En VB6, InputBox() muestra un cuadro de diálogo y espera a que el usuario introduzca texto o haga clic en un botón, y luego devuelve una cadena que contiene el contenido del cuadro de texto.
B4A no tiene ningún cuadro de diálogo que permita al usuario introducir texto. En su lugar, se crea algo similar utilizando un EditText en un layout, o puede utilizar una de las siguientes opciones:

- La Librería de Diálogos creada por un usuario, que ofrece InputDialog para texto, TimeDialog para horas, DateDialog para fechas, ColorDialog y ColorPickerDialog para colores, NumberDialog para números, FileDialog para carpetas y nombres de archivos y CustomDialog.
- InputList para mostrar un diálogo modal con una lista de opciones y botones de selección y devolver un índice indicando cuál ha seleccionado el usuario.
- InputMultiList para mostrar una lista donde el usuario puede seleccionar varios elementos antes de volver.
- InputMap para mostrar un diálogo modal con una lista de elementos y casillas de verificación. El usuario puede seleccionar varios elementos.

Bucle

VB6: Loop … Until / While
B4A: Do While / Do Until … Loop

MsgBox

VB6: MsgBox "text" / i=MsgBox()
B4A: Tiene varias alternativas:
MsgBox("texto", "título")
MsgBox2(Message, Title, Positive, Cancel, Negative, Icon) as Int

ToastMessageShow(text, b)

Números aleatorios

Los números aleatorios generados por ordenadores no son realmente aleatorios. Son "pseudo-aleatorios" y se crean utilizando un algoritmo que parte de un número, la "semilla", para generar el siguiente.

Rnd

En VB6, **`Rnd()`** devuelve un número flotante < 1.
En B4A, **`Rnd(min, max)`** devuelve un entero >= min y < max.

RndSeed

Si # es un número, entonces en VB6, Rnd(-#) pone la "semilla" del generador de números aleatorios en #. Después de esta llamada, Rnd devolverá la misma secuencia de números cada vez.
En B4A, **`RndSeed(#)`** establece la semilla del generador de números aleatorios de la misma manera. # debe ser un número de tipo Long.

Randomize

Si # es un número, entonces en VB6, Randomize(#) usa # para inicializar el generador de números aleatorios de la función Rnd, usando # como nuevo valor semilla. Randomize() sin el número utiliza el valor devuelto por el temporizador del sistema como nuevo valor semilla. Si no se utiliza Randomize, la función Rnd (sin argumentos) siempre utiliza el mismo número como semilla la primera vez que se llama, y a partir de entonces utiliza el último número generado como valor de semilla.
En B4A, no hay equivalente a Randomize, porque la semilla de Rnd siempre es aleatoria automáticamente.

Round

VB6: Round(n) donde n es un número en coma flotante.
B4A: **`Round(n)`** o **`Round2(n, x)`** donde n es un **`Double`** y x=número de posiciones decimales.

Val()

VB6: i = Val(string)
B4A: If IsNumber(string) Then i = string Else i = 0
Un intento de usar i=string provoca una excepción de NumberFormatException si la cadena no es un número válido.

SetFocus

VB6: control.SetFocus
B4A: **`view.RequestFocus`**

Divide por Zero

VB6 lanza una excepción para la división por 0. B4A devuelve 2147483647 o Infinito, dependiendo de si el resultado es un entero o una cadena:

```
Dim i As Int
i = 12/0
Log (i) ' 2147483647

Dim str As String
str = 12/0
Log (str) ' Infinity
```

Shell

VB6: x = Shell("...")
B4A: Ver "Intent".
Esto no es un sustituto completo, pero permite códigos como los siguientes:

```
Dim Intent1 As Intent
Intent1.Initialize(Intent1.ACTION_MAIN, "")
Intent1.SetComponent("com.google.android.youtube/.HomeActivity")
StartActivity(Intent1)
```

Timer

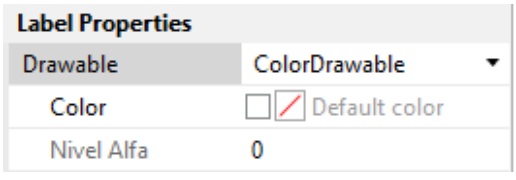
VB6: t = Timer
B4A: **`t = DateTime.Now`**, que devuelve el número de milisegundos desde 1-1-70

TabIndex

En VB6, TabIndex puede configurarse para controlar el orden en el que los controles se enfocan en un formulario cuando se pulsa Tab.
En un dispositivo Android, se maneja la secuencia de acuerdo a su posición. Sin embargo, en el Diseñador o en el código, puede establecer **`EditText.ForceDone`** a **`True`** en todos sus EditTexts: **`EditText1.ForceDoneButton = True`**. Esto obliga al teclado virtual a mostrar el botón *Listo*. A continuación, puede capturar el evento **`EditText_EnterPressed`** y establecer explícitamente el foco en la siguiente vista (con **`EditText.RequestFocus`**).

Ajuste de Transparencia de Label[38]

Puede controlar la transparencia de una label (etiqueta)e como se indica a continuación:
VB6: [Properties > Back Style]
B4A Designer: [Drawable > Nivel Alfa]



Constantes

Hay varias constantes predefinidas útiles en VB6, por ejemplo,
VB6: vbCr, vbCrLf
B4A: CRLF (El equivalente en Android del CRLF de Windows, aunque en realidad es el carácter de salto de línea Chr(10)).

[38] NT: Seguimos manteniendo el nombre del objeto en inglés para una mejor compresión.

String “Members”

VB6 el primer caracter es la posición 1.
B4A la función CharAt() utiliza la posición 0 como primer caracter.
Todas las líneas siguientes producen “a”:
VB6: Mid$("abcde", 1)
VB6: Mid$("abcde", 1, 1)
B4A: "abcde".CharAt(0)
B4A: "abcde".SubString2(0,1)
Las siguientes producen “abc”:
VB6: Mid$("abcde", 1, 3)
B4A: "abcde". SubString2(0, 3)

Left$ y Right$

Estos no existen en B4A. Puede recrearlos como se indica a continuación:
VB6: Left$("abcde", 3)
B4A: "abcde".SubString2(0, 3)

VB6: Right$("abcde", 2)
B4A: "abcde".SubString("abcde".Length - 2)

VB6: If Right$(text, n) = text2
B4A: If text.EndsWith(text2)...

VB6: If Left$(text, n) = text2
B4A: If text.StartsWith(text2)...

VB6: If Lcase$(text) = Lcase$(text2)
B4A: If text.EqualsIgnoreCase(text2)

Len

VB6: x = Len(text)
B4A: x = text.Length

Replace

VB6: text = Replace(text, str, str2)
B4A: text.Replace(str, str2)

Case

VB6: Lcase(text)
B4A: text.ToLowerCase
VB6: Ucase(text)
B4A: text.ToUpperCase

Trim

VB6: Trim(text)
B4A: text.Trim
No hay LTrim o RTrim en B4A

Instr

VB6: Instr(text, string)
B4A: text.IndexOf(string)

VB6: Instr(int, text, string)
B4A: text.IndexOf2(string, int)

VB6: If Lcase$(x) = Lcase$(y)
B4A: If x.EqualsIgnoreCase(y)

VB6: text = Left$(text, n) & s & Right$(Text, y)
B4A: text.Insert(n, s)

Intercepción de errores

VB6

```
Sub SomeSub
  On [Local] Error GoTo ErrorTrap
   ...algún código...
  On Error GoTo 0 [Fin opcional de captura de errores]
  ...código adicional opcional...
  Exit Sub [para evitar ejecutar el código de ErrorTrap]
ErrorTrap:
  ...Código opcional para correction de errores...
  Resume [opcional: "Resume Next" o "Resume [etiqueta de la línea]".
End Sub
```

B4A

```
Sub SomeSub
  Try
   ...algún código...
  Catch [Solo se ejecuta si se produce error]
   Log(LastException) [opcional]
   ...Código opcional para correction de errores...
  End Try
  ...código adicional opcional...
End Sub
```

Con B4A, si se detecta un error en medio de una subrutina grande, NO SE PUEDE hacer una corrección y reanudar dentro del código que se estaba ejecutando. Sólo se ejecuta el código en "Catch". Esto puede parecer que hace que **`Try-Catch-End Try`** se use principalmente durante el desarrollo.

"Immediate Window" vs. Pestaña "Logs"

Comentarios, valores de variables, etc., se pueden mostrar en *Ventana* Inmediato de VB6 utilizando "Debug.Print ...." en el código.
En B4A, mostrar valores de variables, etc. en la Pestaña de eventos Logs.
Tanto VB6 como B4A permiten un paso a paso a través del código mientras se está ejecutando y viendo los valores de las variables. VB6 también permite cambiar el valor de las variables, cambiar el código, saltar a

otras líneas desde la línea actual, etc. Debido a que B4A se ejecuta en un PC mientras que la aplicación se ejecuta en un dispositivo separado, B4A actualmente no puede duplicar todas estas características de depuración de VB6.