Capítulo 5

# Conceptos básicos sobre fundamentos de programación. Los bloques

# Contenido

## 1. Introducción

El manejo de *MIT App Inventor 2* no implica tener conocimientos de programación sobre ningún lenguaje concreto. Se aprenderá lo necesario durante el desarrollo de este manual y en este capítulo se proporcionarán algunas sencillas nociones sobre conceptos básicos de programación que facilitarán el aprendizaje y el uso de los diferentes bloques allanando el camino del desarrollo de aplicaciones *Android* con *MIT App Inventor 2.*

No solo se aprenderán nuevos conceptos sino que se conocerá la jerga necesaria para que los bloques que ofrece *MIT App Inventor* sean familiares desde un primer momento. Si bien es el inglés el idioma en el que están basados, se trata de un inglés básico y repetitivo que se llegará a dominar con un poco de práctica.

Entre los conceptos básicos que se van a tratar en este capítulo se conocerá el significado de variable, instrucción o bucle, entre otros, además de afinar sobre otros que ya se conocen, como evento o método.

## 2. Variables

Una variable representa un valor a través de un identificador, valor que podrá verse alterado mientras la aplicación se encuentre en ejecución. Para hacer uso de una variable, esta se debe crear. El proceso de creación de una variable se denomina *definir* o *declarar* una variable. Cuando una variable ya definida toma un valor se dice que a la variable se le ha *asignado* dicho valor. De otro modo, cuando a una variable se le asigna por primera vez un valor, se dice que la variable ha sido *inicializada*. Esto quiere decir que la variable posee un valor conocido o no nulo.

En cuanto al identificador de una variable, este debe ser único, no pueden existir dos variables con el mismo identificador. De hecho, si por error se intenta establecer un identificador ya utilizado con anterioridad a una variable, *MIT App Inventor 2* añadirá un número natural al final de esta para diferenciarlo.

### Ejemplo

Si se tiene una variable con identificador ‘suma’ y se crea otra con el mismo identificador, automáticamente esta se creará como ‘suma2’. Si ahora se creara otra con identificador ‘suma2’ se le asignaría el identificador ‘suma3’, y así sucesivamente.

Además, se utiliza un sistema de identificación *sensitivo a mayúsculas,* esto quiere decir que una letra minúscula y la misma letra mayúscula son caracteres diferentes.

### Ejemplo

El identificador totalsuma es diferente al identificador totalSuma.

Además de esto, se ha de saber que el identificador de una variable deberá siempre comenzar por un carácter letra o de otro modo, por los caracteres guion bajo ‘_’ o dólar ‘$’. En los sucesivos podrán repetirse estos mismos o ser caracteres numéricos.

En la siguiente imagen se observará un ejemplo en el que se definen una serie de variables que almacenan datos de un vehículo y que tienen por identificadores los nombres *cantidad*, *fabricacion*, *promocion*, *color* o *extras*. Cada una de ellas tiene asignado un valor de diferente naturaleza o *tipo*. Entre los ejemplos propuestos se aprecian valores de tipo *cadena de texto, numérico, booleanos, color* o *lista.*

### Sabía que...

Un tipo booleano es aquel que solo permite los valores lógicos verdadero o falso, en inglés, *true o false.*

Los bloques que correspondan a cada uno de los valores asignados de diferentes tipos a una variable poseerán un color diferente que los identifique.

En *MIT App Inventor 2* se pueden definir dos tipos diferentes de variables, como son **globales** y **locales.** Una variable global se establece con la propiedad *global* delante del identificador de la misma, significando que dicha variable será visible en cualquier ámbito de la aplicación. Por ejemplo, podrá utilizarse desde el editor de bloques tanto dentro como fuera de un procedimiento.

### Nota

Se conoce como *ámbito* de una variable al contexto que tiene dentro de una aplicación y establece las zonas de la misma en las que puede ser utilizada.

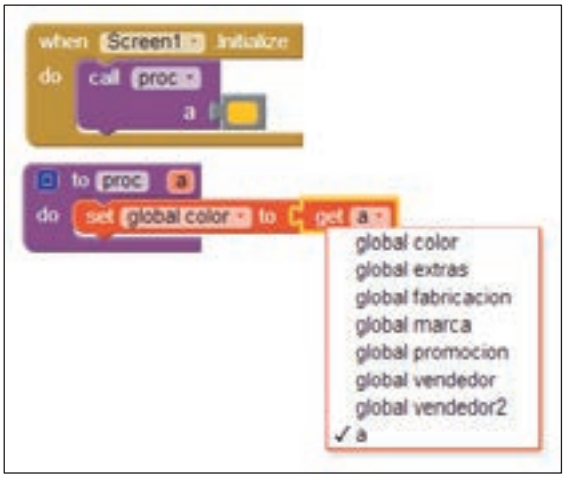

*Variables globales*

De otro modo, al definir una variable *local*, esta llevará la propiedad *local* delante de su identificador y solo será visible dentro de su bloque de definición, como se muestra a continuación. En el ejemplo, la variable local *nombre* solo será accesible desde dentro de su bloque de definición. Obsérvese que desde fuera del mismo no existe.


*Variables locales*

Por otro lado, desde un procedimiento se podrán definir parámetros de entrada, que no son más que variables locales a dicho procedimiento y no serán accesibles fuera de él. En el siguiente ejemplo se pasa como parámetro el color amarillo, cuyo identificador en el procedimiento es *a*. En este caso, obsérvese

cómo desde dentro del procedimiento sí que puede utilizar el parámetro *a*, con lo que aparece en la lista. Fuera de él no existiría ámbito de aplicación para la variable local.




*Variables locales*

## Actividades

1. Indague sobre los tipos de variables que existen en App Inventor

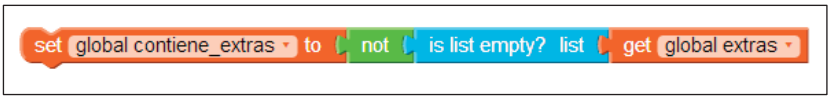
## 3. Instrucciones

Los diferentes bloques que componen la lógica de una aplicación en *MIT App Inventor 2* determinarán el funcionamiento de la misma. Para ello, el dispositivo *Android* deberá interpretar el código de la aplicación y las diferentes instrucciones que se localicen en este. En este caso, el código de *MIT App Inventor 2* se encuentra inmerso en una serie de bloques, cada cual con instrucciones acordes a la acción que se pretende realizar.

Una instrucción se puede definir como pieza clave dentro del funcionamiento de una aplicación. La clasificación que *MIT App Inventor 2* hace de los diferentes bloques es análoga en algunos casos a los tipos de instrucciones que existen, como es el caso, por ejemplo, de instrucciones *aritméticas,* cuya finalidad es realizar operaciones matemáticas; *lógicas,* que realizan operaciones con valores booleanos; de *texto,* que manipulan cadenas de caracteres; o de *control,* que se aprenderá en detalle en el siguiente apartado.

A continuación se verá un ejemplo de los tipos de instrucciones más comunes.

Por ejemplo, la siguiente instrucción establece el valor booleano verdadero sobre la variable *contiene_extras* si la lista extras no está vacía. En caso contrario, se asignará falso como valor. Se podrá comprobar que se pueden combinar bloques de diferente tipo para construir una instrucción.

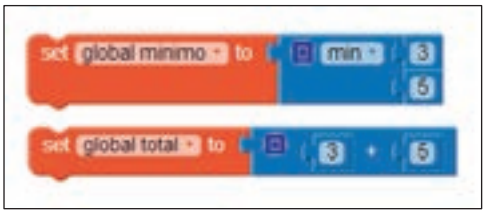



*Ejemplo de instrucción*

En el siguiente ejemplo se asigna a la variable *total* el número de caracteres de la cadena *frase,* incluyendo los espacios. El valor de *total* será 22.


*Ejemplo de instrucción*

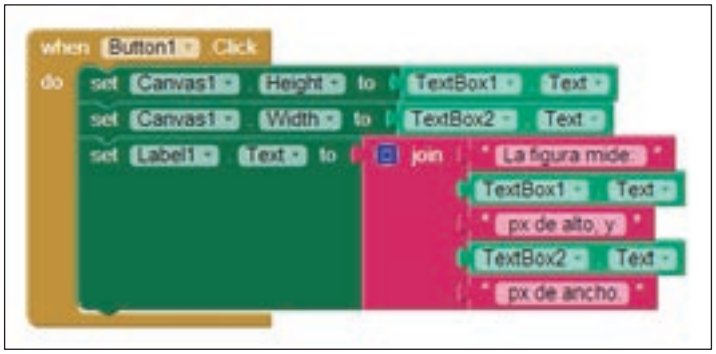
En el próximo caso, se realizan diferentes asignaciones. La primera asigna a la variable *minimo* el menor de los números que se indican. La segunda asigna la suma de dos números a la variable *total*.




*Ejemplo de instrucción*

El último de los ejemplos de instrucciones que se proponen incluye la interacción con componentes de la interfaz gráfica de la aplicación. En él se establece el tamaño de una figura mediante la inserción de sus valores alto y ancho a través de dos campos de entrada de texto que serán aplicados al pulsar un botón. Además se ofrecerá información del nuevo tamaño en una etiqueta. En la primera imagen se observará la interfaz gráfica inicial de la aplicación; en la segunda, el resultado; y en la tercera, las instrucciones que hacen posible su funcionamiento.





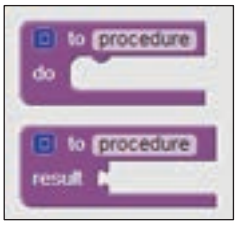

*Ejemplo de instrucción*

Las instrucciones vistas hasta ahora son solo un ejemplo. Se podrán definir y combinar las que se necesiten para implementar la lógica de su aplicación con el objetivo de que esta se adapte a los requisitos.

## 4. Procedimientos y funciones

Cuando es necesario utilizar la misma instrucción o conjunto de ellas en diferentes partes de nuestra aplicación, es conveniente agruparlas en bloques y acceder a ellas con el objetivo de no repetir dichas instrucciones en diferentes ocasiones. Para estos casos se podrá hacer uso de *procedimientos*, que no son más que diferentes instrucciones agrupadas y que, entre todas, realizan una tarea conjunta. El uso de procedimientos hará que el visor del panel de bloques esté más limpio y organizado.

El uso de un procedimiento se llevará a cabo en tres sencillos pasos. El primero de ellos será definir el procedimiento; el segundo, insertar las instrucciones necesarias en el mismo; y el tercero, realizar una llamada a dicho procedimiento. Desde la categoría *Procedures* del panel **Blocks** del editor de bloques se podrá añadir el bloque correspondiente al procedimiento. En la siguiente imagen se podrán comprobar los diferentes bloques relacionados con procedimientos que existen. A partir de ellos, cada vez que se cree un procedimiento, irán apareciendo los bloques correspondientes a las llamadas de los procedimientos creados. *MIT App Inventor 2* predefine por defecto el nombre de un procedimiento, que se podrá cambiar posteriormente solo con hacer clic sobre él y escribiendo uno nuevo. Hay que tener en cuenta que no podrán repetirse los nombres de procedimientos y deberán respetarse las mismas directrices que se expusieron para la definición de los identificadores de las variables. Si el nombre que se elija es incorrecto, *MIT App Inventor 2* lo hará saber especificando un nombre genérico y si se elige uno que ya existe se añadirá, al igual que ocurría para la declaración de variables, un número natural seguido del nombre con la finalidad de diferenciarlos.



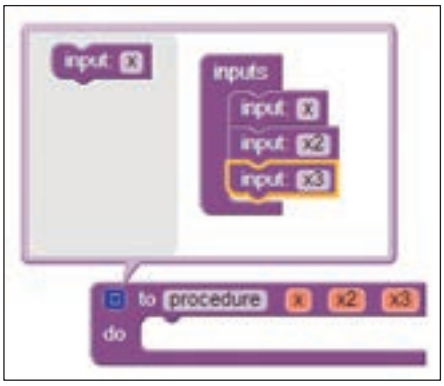

*Bloques de procedimientos*

Se habrá podido comprobar en la imagen anterior que existen dos bloques iniciales de procedimientos. Estos dos bloques se diferencian y se clasifican en función de si devuelven datos o no. Un procedimiento puede realizar simplemente una tarea o, además, devolver un valor. Un procedimiento que devuelve algún dato se denomina **función.**

Además, un procedimiento puede recibir datos con los que se podrá operar. Estos datos se conocen con el nombre de **argumentos** o **parámetros.** Un argumento se podrá definir una vez insertado el bloque del procedimiento, simplemente seleccionando la figura azul de la esquina superior izquierda del procedimiento y arrastrando un bloque de entrada o *input* desde la parte izquierda a la derecha. Obsérvese cómo cada uno de los argumentos que se añade al bloque *inputs* aparecerá en la zona superior del bloque del procedimiento. En el ejemplo siguiente se distinguirán tres argumentos con nombres x, x2 y x3.

## Actividades

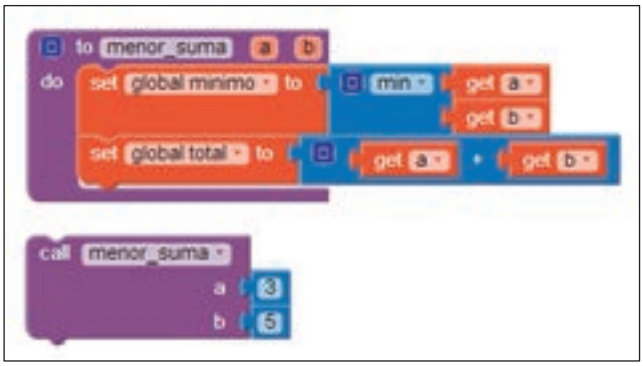
2. Busque información sobre los tipos que se permiten en los argumentos o parámetros de un procedimiento.




*Ejemplo de argumentos*

En relación a los datos, en un procedimiento existen datos de entrada y datos de salida. Un dato de entrada será el parámetro o parámetros recibidos con los que trabajará un procedimiento y, por otro lado, un dato de salida se corresponderá con el resultado que el procedimiento devuelva en caso de tratarse de una función.

Una vez definido el procedimiento, se necesitará de un bloque de llamada al mismo para poder ejecutar las instrucciones que se encierran en él. Por ejemplo, se define un procedimiento llamado *menor_suma*. El procedimiento, primero, asigna a la variable *minimo* el menor de los números que se han pasado como argumentos, y segundo, asigna la suma de ellos a la variable *total*. Para utilizar el procedimiento *menor_suma* se realiza una llamada al procedimiento y se le pasan como argumentos los números 3 y 5. Este bloque de llamada se podrá encontrar en la categoría *Procedures* del panel **Blocks** después de haber definido el procedimiento en cuestión.




*Ejemplo de uso de un procedimiento*

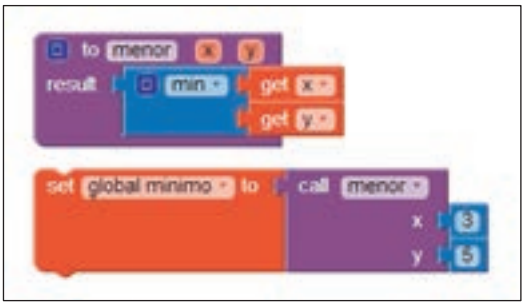
Extrapólese el procedimiento anterior a una función. Esta, en lugar de almacenar el menor de dos números pasados como argumentos o la suma de ellos a una variable desde dentro del procedimiento, lo que hará será devolver los valores calculados. Para ello, se definirá la función *menor* y posteriormente su llamada especificando además los valores pasados como argumentos. El valor devuelto en la llamada será asignado a la variable *minimo*.

## Recuerde

Una vez definido el procedimiento se necesitará de un bloque de llamada al mismo para poder ejecutar las instrucciones que se encierran en él.

## Actividades

3. ¿En qué casos puede serle útil una función en contra de un procedimiento?



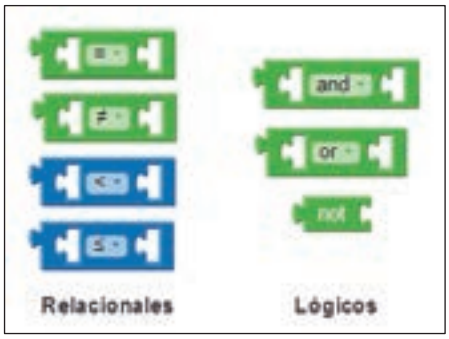

*Ejemplo de uso de una función*

## 5. Bloques de control

Hasta el momento se ha aprendido como, a partir de sencillos bloques, se pueden definir variables, instrucciones o procedimientos. Aun así, se necesitarán una serie de bloques adicionales que aumenten las posibilidades de diseño de la aplicación. Entre estos se encuentran un grupo de bloques de tipo condicional que permitirán definir expresiones lógicas, además de ciertas estructuras, que permitirán ejecutar repetidas veces un conjunto de instrucciones como son los bucles. Con la combinación de ambos grupos de bloques junto con los que ya se conocen, se podrá definir el funcionamiento de la aplicación.

### 5.1. Expresiones lógicas

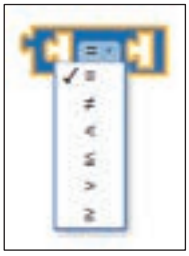
Un tipo de instrucciones que cabe destacar por su utilidad son las instrucciones lógicas. Este tipo de instrucción permitirá evaluar condiciones sobre expresiones lógicas. Para ello, se utilizarán bloques de operadores relacionales y lógicos. En la siguiente imagen se observan ejemplos diferentes de cada uno de ellos.




*Operadores relacionales y lógicos*

Para el primer grupo, se muestran los operadores relacionales *igual* y *distinto* que se aplicarán sobre valores booleanos determinando la igualdad o no igualdad entre estos valores, y los operadores relacionales *menor que* y *menor o igual que*, que se aplican sobre valores numéricos y evaluarán si un dato es menor que otro, o si es menor o incluso igual que otro. En el caso de los operadores *igual* y *distinto* también existen bloques para evaluar otro tipo de datos, como son numérico o cadena de caracteres. En el segundo de los grupos, se encontrarán los operadores lógicos, *y*, *o* y *no*, todos aplicables a tipos de datos booleanos.

Si se hace clic con el botón izquierdo del ratón encima del operador y se selecciona aparecerá un listado de operadores y se podrá cambiar el actual por el que se elija de la lista.



*Cambiar operador*

### Actividades

4. Exponga diferentes casos de uso de un operador lógico.

## 5.2. Instrucciones condicionales

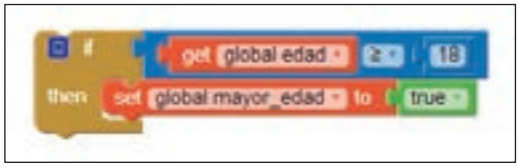
No todas las instrucciones que se añadan al editor de bloques deberán ejecutarse siempre y en todas las condiciones. Se puede necesitar ejecutarlas solo en determinados casos. Una instrucción condicional permitirá que una instrucción se ejecute solo si se cumple cierta condición definida.

Entre los bloques de instrucciones condicionales se destacan los siguientes:

**if-then**

Esta instrucción se puede resumir como *IF condición THEN acciones*. En ella se evaluará una condición definida en el apartado *if* y permitirá, o no, la ejecución de las instrucciones que se encuentren dentro del bloque en el apartado *then*. El significado de este bloque se podría traducir de la siguiente forma: "cuando se cumpla la condición especificada, ejecutar las acciones indicadas en el bloque; en otro caso no se hará nada".

Un ejemplo de uso de este bloque se muestra en la siguiente imagen. El bloque evaluará que la edad sea mayor o igual a 18 para asignar la categoría de mayoría de edad a verdadero de una persona. Si no se cumple la condición, no se establece la mayoría de edad.




*Bloque condicional if-else*

**if-then-else**

Como extensión de la instrucción condicional *if-then,* vista anteriormente, se dispone de *if-then-else*. En ella, además, se permite la ejecución de un grupo de instrucciones si no se cumple la condición especificada. Su modo de actuación se resume de la siguiente forma: *IF condición THEN acciones1 ELSE acciones2*. Su significado sería, "cuando se cumpla la condición especificada, ejecutar acciones1 indicadas en el primer bloque; en otro caso, ejecutar acciones2, indicadas en el segundo bloque".

Para este caso, obsérvese el ejemplo que se muestra en la siguiente imagen. El bloque evaluará que la edad sea mayor o igual a 18 para asignar la categoría de mayoría de edad a verdadero de una persona. Si no se cumple la condición, se establece la mayoría de edad a falso.


*Bloque condicional if-then-else*

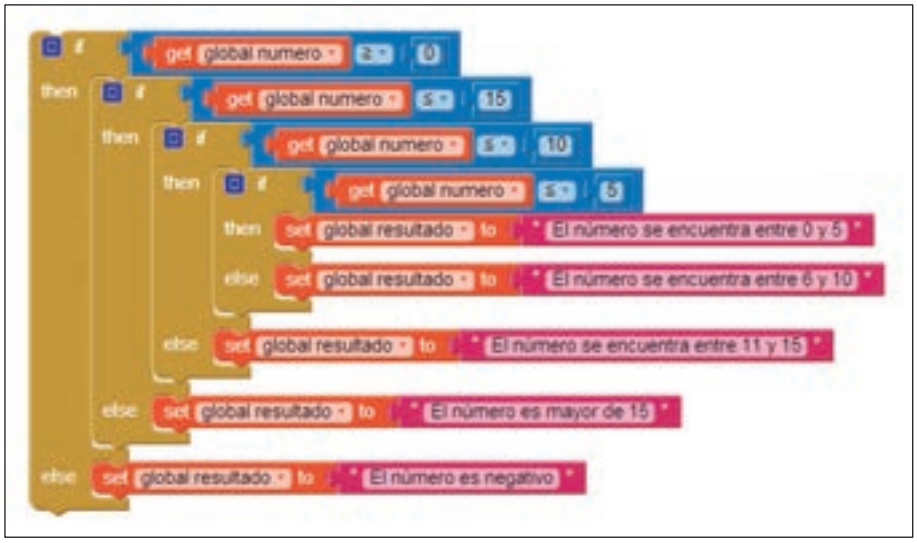
## Actividades

5. Exponga diferentes casos de uso de un bloque condicional if-then-else.

## 5.3. Anidación de instrucciones condicionales

Es habitual que en una aplicación se necesiten realizar diferentes comprobaciones como las vistas en los bloques de instrucciones condicionales y que,

además, unas comprobaciones dependan de otras. En el siguiente caso, se evalúa el valor de un número y se almacena en la variable *resultado* una descripción sobre el valor de dicho número. La casuística de este ejemplo anida una serie de instrucciones condicionales *if-then-else* hasta ubicar el valor del número en la que corresponda y así asignar el mensaje correcto a la variable *resultado*. La anidación permite incluir instrucciones de control dentro de otras instrucciones del mismo tipo.




*Bloque condicional if-then-else anidado*

De otro modo, se puede necesitar realizar diferentes instrucciones condicionales que sean excluyentes entre sí. Esto quiere decir que un valor depende de una única condición y de ninguna más. El siguiente ejemplo muestra otra forma de anidar bloques de control que mantiene el editor de bloques más limpio y legible.


*Bloque condicional if-then-else anidado*

## Nota

Siempre se debe utilizar el menor número de bloques posible que permita desarrollar la casuística necesaria para su aplicación. Una estructura clara en el editor de bloques la mantendrá más legible.

## Aplicación práctica

**Diseñe con los bloques vistos hasta el momento una aplicación con interfaz gráfica y lógica necesaria para que un procedimiento determine si dos números pasados como argumento son iguales. La aplicación mostrará 'Números iguales' si lo son y, en caso de no serlo, se especificará cuál es el mayor y cuál el menor.**

**Nota: Utilice componentes TextBox para introducir los números, un Button para evaluarlos y un Label para mostrar el resultado.**

**SOLUCIÓN (Posible solución)**

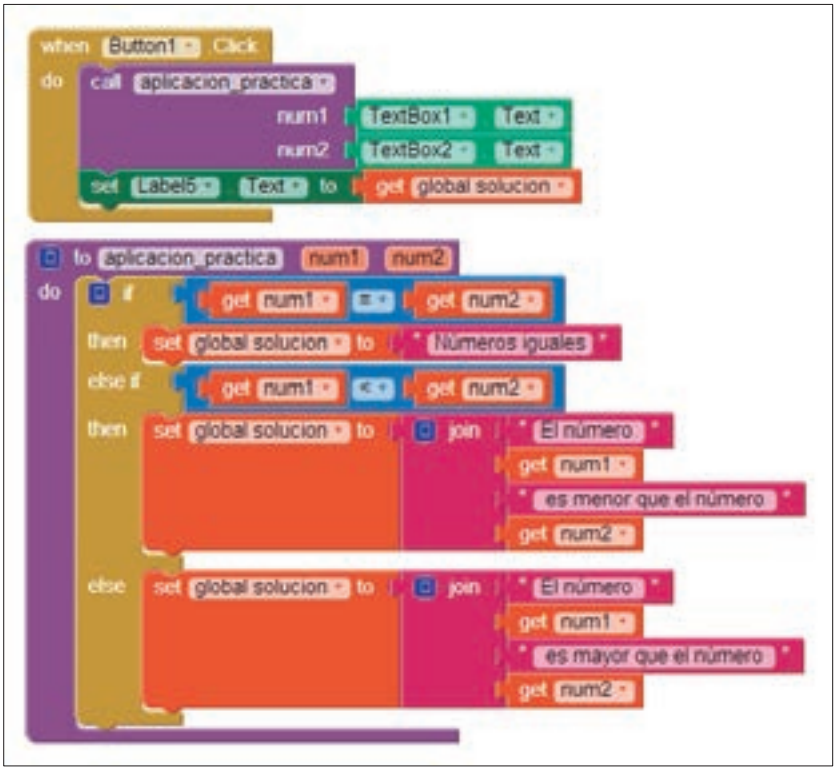
- Interfaz gráfica:


*Interfaz gráfica*

Continúa en página siguiente >>

>> Viene de página anterior

- Lógica de la aplicación:




*Lógica de la aplicación*

## 5.4. Bucles

Otro tipo de estructura que se utilizará en los desarrollos será la de **bucle.** Este tipo de estructura se caracteriza por permitir la ejecución repetida de la instrucción o conjunto de ellas que se encuentren en su interior. A cada ejecución de las instrucciones contenidas en el bucle se le denomina **iteración.** El número de veces que un bucle es ejecutado lo determina una condición o será fijado por el desarrollador de antemano. La utilidad que tiene un bucle con

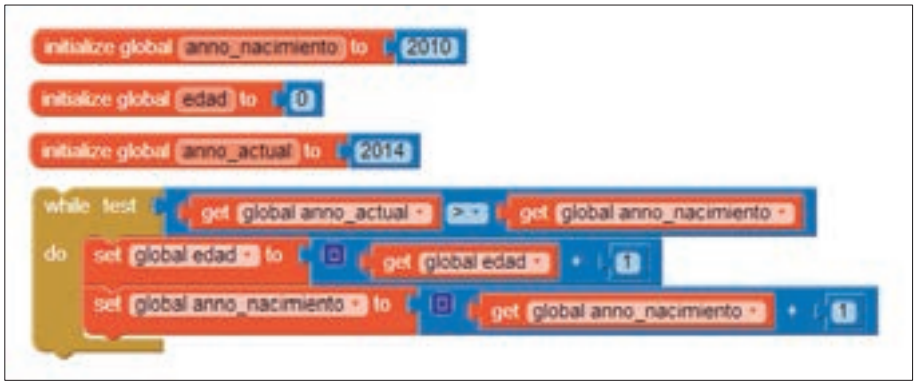
respecto al uso de otro tipo de bloques que realicen las mismas instrucciones es que facilita el desarrollo de aplicaciones reduciendo el número de bloques a emplear, sobre todo los que se deben repetir numerosas veces.

A continuación se conocerán los bloques de dos tipos de bucles que *MIT App Inventor 2* proporciona, que son *while* y *foreach*. Ambos se encontrarán en la sección ***Control*** del panel **Blocks** del **Editor de bloques.**

**while**

El bloque *while* permitirá la ejecución de las instrucciones que se encuentren en su interior hasta que su condición deje de ser verdadera. La sintaxis de este bucle es *WHILE TEST condición DO acciones* y su significado, "mientras la condición se cumpla, se ejecutarán las acciones".

En la siguiente imagen se podrá comprobar un ejemplo de este tipo de bucle. En él se observa cómo se compara el año actual con el año de nacimiento de una persona. Las acciones que se llevan a cabo por el bucle en cada iteración son sumar uno a la variable edad que inicialmente está inicializada a 0 y sumar uno al año de nacimiento. Con esto el bucle irá calculando los años que tiene esa persona hasta que el año actual deje de ser mayor que el de nacimiento, con lo que se saldrá del bucle cuando estos sean iguales. Se ha de tener en cuenta que es solo un ejemplo para mostrar el funcionamiento del bucle *while*. Si se quisieran calcular los años correctamente habría que tener en cuenta tanto el día y mes de nacimiento como el día y mes actual, además de los años.




*Bucle while*

En la siguiente tabla se irán viendo los valores que contiene cada variable en cada iteración que se ejecute. El resultado al finalizar las comprobaciones determina que la persona tiene 4 años de edad y los valores que no cumplen la condición del *while* se dan cuando el año actual se iguala al año de nacimiento, es decir 2014.

**Tabla de valores**

| Iteración | Edad | Año nacimiento | Año actual |
|---|---|---|---|
| Valores iniciales | 0 | 2010 | 2014 |
| 1 | 1 | 2011 | 2014 |
| 2 | 2 | 2012 | 2014 |
| 3 | 3 | 2013 | 2014 |
| 4 | 4 | 2014 | 2014 |

## Actividades

6. Exponga diferentes casos de uso de un bucle while.

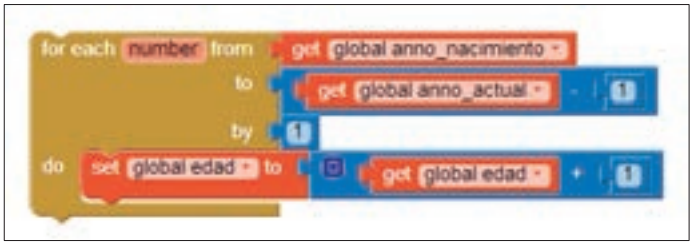
### for each

Otro bucle que *MIT App Inventor 2* incluye es el bucle *for each*, que se utiliza en caso de que sea posible saber o, por lo menos, intuir de antemano el número de veces que se ejecutará el bucle. Este número de veces será deducido partiendo de un valor inicial proporcionado por el parámetro *from* y ejecutándose el bucle tantas veces hasta llegar a un valor final facilitado por el parámetro *to*. Además, el valor de incremento entre ellos se podrá especificar en el parámetro *by*. Esto quiere decir que no tiene por qué ir incrementándose este valor de uno en uno, sino que se puede especificar cualquier otro valor

o instrucción que calcule algún valor. El *incremento* en un bucle también se conoce como *paso*.

La sintaxis de este bucle es *FOR EACH iteración FROM valor inicial TO valor final BY incremento DO acciones* cuyo significado se traduce en "para cada valor de iteración desde un valor inicial hasta un valor final con incrementos ejecutar las acciones".

En relación al ejemplo visto anteriormente para el bucle *while* se podría haber realizado una lógica similar mediante el bucle *for each*. Préstese atención en la siguiente imagen y se podrá comprobar que tanto las operaciones como el resultado son similares. Para cada iteración se sumará uno a la edad y se realizarán tantas iteraciones, partiendo del valor del año de nacimiento, hasta alcanzar el año actual menos uno con incrementos de uno. Al año actual se le resta uno para que no se cuente. Si se recuerda, en el ejemplo del bucle *while* la condición era que el año actual fuera mayor que el año de nacimiento y no mayor o igual. En este caso, el proceso de ir sumando uno al valor del año de nacimiento hasta llegar al año actual es equivalente al incremento realizado por el valor del paso determinado en el argumento *by*.




*Bucle for each*

La tabla de valores correspondiente al bucle *for each* es la siguiente, que como se podrá comprobar es similar a la tabla de valores obtenida con el bucle *while* anterior, por el hecho de que cada bucle se ejecuta cuatro veces.

| Tabla de valores | | | |
|---|---|---|---|
| **Iteración** | **Edad** | **From** | **To** |
| Valores iniciales | 0 | 2010 | 2013 |
| 1 | 1 | 2010 | 2013 |
| 2 | 2 | 2011 | 2013 |
| 3 | 3 | 2012 | 2013 |
| 4 | 4 | 2013 | 2013 |

## Nota

En función de la complejidad de su aplicación se podrá requerir la anidación entre bucles, es decir, incluir bucles dentro de bucles.

## Actividades

7. Exponga diferentes casos de uso de un bucle for each.

### Bucle infinito

Al hablar de bucles es necesario conocer el concepto de *bucle infinito* dado que será uno de los posibles culpables de que se pierda tiempo buscando errores en la lógica de la aplicación. Como bien se sabe, un bucle realiza repetidas ejecuciones de código hasta que se cumple una condición. Imagínese que esa condición no se cumpliera nunca. Esto provocaría que el flujo de ejecución de la aplicación se quede dentro del bucle indefinidamente y no avance. Si

esto ocurre, no se podría hacer uso de la aplicación correctamente, además de tener que abortarla de manera brusca si es que no se produce un cuelgue del sistema antes.

Con esto, se quiere exponer que el uso de un bucle en un momento dado puede ser tanto necesario como peligroso. Deben usarse repasando al detalle que la condición que se establece para salir del bucle es correcta.

Imaginar que en el ejemplo anterior sobre el bucle *while* se olvida sumar uno al año de nacimiento, con lo que en la condición siempre será menor que el año actual. Esto provocará un bucle infinito y el error bloqueará el terminal si está haciendo pruebas mediante *MIT AI Companion,* por ejemplo, o de otro modo el emulador. Si ocurre esto se debe reiniciar la conexión desde el menú **Connect** a través de la opción **Reset Connection.** Si aun así no fuera posible reiniciar la conexión se deberá abortar mediante la opción **Hard Reset** del mismo menú.


*Opción Hard Reset*

## Actividades

8. Exponga diferentes casos de uso de un bucle infinito,

## 6. Interacción con la aplicación. Eventos

Una aplicación correctamente diseñada y desarrollada con un propósito bien definido puede parecer a priori ser candidata a buena aplicación. Piénsese por un momento que, además de los factores descritos como positivos, se sume otro como que el usuario no tenga una interacción agradable con ella. Con esto, el usuario no encontrará una experiencia satisfactoria y quizás deje de usarla. Por ello, se debe establecer una correcta comunicación entre la aplicación y el usuario. Este objetivo se llevará a cabo mediante la utilización de los diferentes bloques que *MIT App Inventor 2* proporciona para definir eventos. En capítulos anteriores se introdujo el concepto de evento, quedando pendiente conocer los bloques mediante los que se pueden llevar a cabo. En ellos se profundizará en este apartado.

**Recuerde**

Un evento es un mecanismo usado en programación para notificar que ha ocurrido algo. Por ejemplo, se ha pulsado una tecla, se ha hecho clic con el ratón o se ha modificado la propiedad de algún componente.

Cada bloque correspondiente a un evento poseerá un nombre relacionado al evento que tiene lugar y estará vinculado al componente en el cual se produce. No todos los componentes estarán provistos de bloques que respondan a todos los eventos. Incluso existen componentes que, por su naturaleza, no responden a ningún tipo de evento. Piénsese, por ejemplo, en los componentes *Button* y *Slider*. Un botón estará provisto de una serie de bloques que respondan a eventos, como por ejemplo *Click* o *LongClick,* que un deslizador no poseerá. En caso contrario, un deslizador poseerá un evento que detecte el cambio de posición de su marcador de posición que en un botón carecería de sentido. Con esto se deduce que cada componente poseerá sus propios bloques de eventos relacionados al funcionamiento del mismo y la mayaría de ellos actúan en respuesta a una acción del usuario con la interfaz gráfica de la aplicación.

Junto a los eventos relacionados con cada componente podrán encontrarse bloques que respondan a eventos producidos en la propia pantalla de la aplicación. Además, existen componentes que no forman parte de la interfaz gráfica y que corresponden a elementos internos *hardware* de un dispositivo *Android* físico. Se podrán reconocer de un vistazo los bloques de eventos por su color amarillo mostaza y porque su descripción es del tipo *WHEN componente.evento DO acciones*. A continuación, se conocerán los bloques de eventos más relevantes y comunes que se podrán utilizar en *MIT App Inventor 2*. La clasificación se realiza en base a los componentes y a la categoría de estos en la paleta de componentes, aunque en primer lugar se conocerá los correspondientes al componente *Screen*.

## 6.1. Screen

### *BackPressed*

Evento que detecta que el botón *volver* ha sido pulsado y realiza las opciones definidas en dicho evento.

### *ErrorOccurred*

Detecta que se produce un error, lo indica mediante una notificación que muestra un código y una descripción del error. Este evento permite tratar cada error de una manera determinada con el fin de que su aplicación reaccione del mejor modo a un imprevisto surgido.

### *Initialize*

Evento que se dispara al iniciar la aplicación y es útil para determinar configuraciones y establecer valores iniciales.

### *OtherScreenClosed*

El evento detecta que una pantalla en cuestión ha vuelto a tomar el control tras cerrarse otra.

#### *ScreenOrientationChanged*

En este caso, se realizan las acciones indicadas tras detectarse un cambio de orientación en la pantalla del dispositivo.

## 6.2. User Interface

### Button

#### *Click*

Indica que el usuario ha hecho clic en el botón.

#### *GotFocus*

Indica que el dedo del usuario se ha situado sobre el componente *Button* y este tiene el foco, por lo que ahora es posible hacer clic en él.

#### *LongClick*

Indica que el usuario ha hecho clic largo sobre el botón.

#### *LostFocus*

Indica que el dedo ya no está situado sobre el componente *Button* y este ha perdido el foco, por lo que ahora no es posible hacer clic en él.

### CheckBox

#### *Changed*

Detecta que el *CheckBox* ha cambiado de estado.

***GotFocus***

Indica que el componente *CheckBox* tiene el foco, por lo que ahora es posible cambiar su estado.

***LostFocus***

Indica que el componente *CheckBox* ha perdido el foco, por lo que ahora no es posible cambiar su estado.

**ListPicker**

***AfterPicking***

Detecta que un elemento de la lista del *ListPicker* ha sido seleccionado y procede a ejecutar las acciones definidas en él.

***BeforePicking***

Establece los elementos a mostrar en el *ListPicker* antes de dar opción a que se haya elegido ninguna.

***GotFocus***

Indica que el componente *ListPicker* tiene el foco, por lo que ahora es posible seleccionarlo.

***LostFocus***

Indica que el componente *ListPicker* ha perdido el foco, por lo que ahora no es posible seleccionarlo.

**PasswordTextBox**

***GotFocus***

Indica que el cursor se sitúa sobre el componente *PasswordTextBox* y este tiene el foco, por lo que ahora es posible introducir texto en él.

***LostFocus***

Indica que el cursor ya no está situado sobre el componente *Password-TextBox* y este no tiene el foco, por lo que ahora no es posible introducir texto en él.

**Slider**

***PositionChanged***

Indica que la posición del deslizador ha cambiado.

**TextBox**

***GotFocus***

Indica que el cursor se sitúa dentro del componente *TextBox* y este tiene el foco, por lo que ahora es posible introducir texto en él.

***LostFocus***

Indica que el cursor ya no está situado en el componente *TextBox* y este no tiene el foco, por lo que ahora no es posible introducir texto en él.

**Clock**

***Timer***

Indica que el temporizador se ha acabado, pudiendo realizar alguna acción determinada.

**Notifier**

***AfterChoosing***

Detecta que una opción ha sido elegida, indica la selección que se ha realizado mediante la llamada a *ShowChooseDialog* y realiza las acciones definidas.

***AfterTextInput***

Detecta que una respuesta ha sido introducida mediante la llamada a *ShowTextInput* y realiza las acciones definidas.

## 6.3. Media

**CamCorder**

***AfterRecording***

Indica que un vídeo ha sido grabado con la cámara y proporciona su ruta de almacenamiento.

**Camera**

***AfterPicture***

Indica que una fotografía ha sido tomada con la cámara y proporciona su ruta de almacenamiento.

**ImagePicker**

***AfterPicking***

Detecta que una imagen de la galería ha sido seleccionada y procede a ejecutar las acciones definidas en él.

***BeforePicking***

Detecta que el componente *ImagePicker* ha sido pulsado y establece las acciones definidas antes de que se elija una imagen de la galería.

***GotFocus***

Indica que el componente *ListPicker* tiene el foco, por lo que ahora es posible seleccionarlo.

***LostFocus***

Indica que el componente *ListPicker* ha perdido el foco, por lo que ahora no es posible seleccionarlo.

**Player**

***Completed***

Detecta que el archivo multimedia ha llegado a su fin e indica las acciones a realizar.

**SoundRecorder**

***AfterSoundRecorder***

Facilita la localización del sonido una vez grabado.

***StartedRecording***

Inicia la grabación del nuevo sonido.

***StoppedRecording***

Indica que la grabación actual se ha detenido y puede comenzar la grabación de otro sonido.

**SpeechRecognizer**

***AfterGettingText***

Detecta que ha finalizado el reconocimiento de voz y realiza las acciones indicadas.

***BeforeGettingText***

Detecta que el componente de reconocimiento de voz ha sido invocado y realiza las acciones definidas antes de que comience el reconocimiento de voz.

**TextToSpeech**

***AfterSpeaking***

Detecta que se ha finalizado de sintetizar el texto y realiza las acciones indicadas.

***BeforeSpeaking***

Detecta que el componente de síntesis de voz ha sido invocado y realiza las acciones definidas antes de que comience la conversión de texto a voz.

**VideoPlayer**

***Completed***

Detecta que el vídeo ha llegado a su fin e indica las acciones a realizar.

## 6.4. Drawing and Animation

### Canvas

#### *Dragged*

Este evento detecta un movimiento de arrastre entre dos puntos realizado por el usuario sobre el componente lienzo.

#### *Flung*

Detecta que se realiza el gesto de lanzar un componente dentro del lienzo. Entre otros, se conocen, la posición inicial del movimiento o la velocidad del componente lanzado.

#### *TouchDown*

Detecta la posición en la que el usuario comienza a tocar el lienzo. Es decir, cuando sitúa su dedo en la pantalla del dispositivo dentro del lienzo.

#### *TouchUp*

Detecta la posición en la que el usuario deja de tocar el lienzo. Es decir, cuando levanta su dedo de la pantalla del dispositivo dentro del lienzo.

#### *Touched*

Detecta la posición en la que el usuario ha tocado el lienzo. Es decir, ha realizado un ligero toque con su dedo sobre la pantalla del dispositivo dentro del lienzo. Comprende dos acciones, tocar y dejar de tocar el lienzo en un breve espacio de tiempo.

## Ball – ImageSprite

### *CollidedWith*

Este evento detecta cuándo dos componentes Ball o ImageSprites colisionan entre sí.

### *Dragged*

Este evento detecta un movimiento de arrastre entre dos puntos de un componente *Ball* o *ImageSprite* realizado por el usuario sobre el componente lienzo.

### *Edge Reached*

Detecta que un componente *Ball o ImageSprite* llega a tocar el borde de la pantalla.

### *Flung*

Detecta que se realiza el gesto de lanzar un componente *Ball o ImageSprite* dentro del lienzo. Entre otros, se conocen, la posición inicial del movimiento o la velocidad del componente lanzado.

### *NoLongerCollidingWith*

Evento que indica que dos componentes *Ball* o *ImageSprites* no han chocado dentro de un lienzo.

### *TouchDown*

Detecta la posición en la que el usuario comienza a tocar un componente *Ball* o *ImageSprite* sobre el lienzo. Es decir, cuando sitúa su dedo en la pantalla del dispositivo dentro del lienzo sobre uno de estos componentes.

#### *TouchUp*

Detecta la posición en la que el usuario deja de tocar un componente *Ball* o *ImageSprite* sobre el lienzo. Es decir, cuando levanta su dedo de la pantalla del dispositivo dentro del lienzo sobre uno de estos componentes.

#### *Touched*

Detecta la posición en la que el usuario ha tocado un componente *Ball* o *ImageSprite* sobre el lienzo. Es decir, ha realizado un ligero toque con su dedo sobre la pantalla del dispositivo dentro del lienzo en uno de estos componentes. Comprende dos acciones, tocar y dejar de tocar uno de estos componentes sobre el lienzo en un breve espacio de tiempo.

## 6.5. Sensors

### AcelerometerSensor

#### *AccelerationChanged*

Detecta los cambios de aceleración en los valores de cualquiera de las tres dimensiones y realiza las funciones definidas para ello.

#### *Shaking*

Evento que lanza las acciones en él definidas cuando detecta que el dispositivo es agitado.

### BarcodeScanner

#### *AfterScan*

Evento que detecta la lectura por parte de un escáner de código de barras y devuelve el resultado leído.

**NearField**

***TagRead***

Realiza las acciones definidas en el evento tras ser detectada por el dispositivo una etiqueta y leído un mensaje.

***TagWritten***

El evento define el comportamiento de la aplicación después de que se haya escrito una etiqueta.

**OrientationSensor**

***OrientationChanged***

Detecta que la orientación del dispositivo ha cambiado y realiza las opciones definidas en tal caso. Dispone los valores de las variables *azimuth, pitch* y *roll.*

**LocationSensor**

***LocationChanged***

El evento define las acciones a realizar cuando detecta que la localización del dispositivo ha cambiado.

***StatusChanged***

Realiza las acciones definidas cuando cambia el proveedor de servicio.

## 6.6. Social

### ContactPicker

#### *AfterPicking*

Detecta que un elemento de la lista del *ContactPicker* ha sido seleccionado y procede a ejecutar las acciones definidas en él.

#### *BeforePicking*

Detecta que el componente *ContactPicker* ha sido pulsado y establece las acciones definidas antes de que se elija un contacto de la lista.

#### *GotFocus*

Indica que el componente *ContactPicker* tiene el foco, por lo que ahora es posible seleccionarlo.

#### *LostFocus*

Indica que el componente *ContactPicker* ha perdido el foco, por lo que ahora no es posible seleccionarlo.

### EmailPicker

#### *GotFocus*

Indica que el componente *EmailPicker* tiene el foco, por lo que ahora es posible introducir una dirección de email.

#### *LostFocus*

Indica que el componente *EmailPicker* ha perdido el foco, por lo que ahora no es posible introducir una dirección de email.

**PhoneNumberPicker**

***AfterPicking***

Detecta que un elemento de la lista del *PhoneNumberPicker* ha sido seleccionado y procede a ejecutar las acciones definidas en él.

***BeforePicking***

Detecta que el componente *PhoneNumberPicker* ha sido pulsado y establece las acciones definidas antes de que se elija un número de teléfono de la lista.

***GotFocus***

Indica que el componente *PhoneNumberPicker* tiene el foco, por lo que ahora es posible seleccionar un número de teléfono.

***LostFocus***

Indica que el componente *PhoneNumberPicker* ha perdido el foco, por lo que ahora no es posible seleccionar un número de teléfono.

**Texting**

***MessageReceived***

Evento que detecta cuándo un mensaje de texto es recibido por el dispositivo y permite definir acciones como respuesta. Obtiene tanto el número del remitente como el propio mensaje.

**Twitter**

***DirectMessagesReceived***

Evento que detecta la recepción de mensajes directos que han sido demandados a través del método *RequestDirectMessages*.

***FollowersReceived***

Evento que detecta la recepción de seguidores que han sido demandados a través del método *RequestFollowers*.

***FriendTimelineReceived***

Evento que detecta la recepción de la línea de tiempo que ha sido demandada a través del método *RequestFriendTimeline*. A su vez, cada elemento de la línea de tiempo es una lista en la que, en primer lugar, se encuentra el nombre de usuario, y en el segundo, el estado twitteado por dicho usuario.

***IsAuthorized***

Este evento detecta que la llamada al método *Authorize* ha sido correcta. De otro modo, cuando se llame al método *CheckAuthorized* y ya se posean credenciales válidas también será detectado por el evento.

***MentionsReceived***

El evento detecta que se han recuperado las menciones solicitadas a través del método *RequestMentions.*

***SearchSuccessful***

El evento detecta que ha finalizado una búsqueda iniciada a través del método *SearchTwitter.*

## 6.7. Storage

### File

#### *GotText*

Este evento se manifiesta cuando el contenido de un fichero ha sido leído. La forma habitual de hacerlo será después de llamar al método *ReadForm* y pasarle el nombre del fichero.

### FusionTablesControl

#### *GotResult*

El evento revela cuándo ha terminado con resultado un proceso de consulta a *Fusion Tables*.

### TinyWebDB

#### *GotValue*

El evento detecta que se ha producido con éxito una petición realizada sobre el servidor a través del método *GetValue*.

#### *ValueStored*

El evento detecta que se ha producido satisfactoriamente una petición realizada sobre el servidor a través del método *StoreValue*.

#### *WebServiceError*

Evento que detecta un error durante la comunicación con el servicio web.

## 6.8. Connectivity

**ActivityStarter**

***AfterActivity***

El evento ejecuta las acciones en él definidas tras detectar que se ha iniciado una actividad concreta y definida en la aplicación.

**BluetoothServer**

***ConnectionAccepted***

Detecta que el componente *BluetoothServer* ha aceptado una conexión y establece las acciones definidas en él.

**Web**

***GotFile***

Este evento se dispara cuando ha terminado una solicitud a través del componente *Web* obteniéndose un fichero como resultado.

***GotText***

Este evento se dispara cuando ha terminado una solicitud a través del componente *Web* obteniéndose texto como resultado.

### Actividades

9. Practique en MIT App Inventor 2 el uso de los eventos que más le hayan llamado la atención en un proyecto de aplicación real.

Como se ha podido comprobar, existen muchos eventos asociados a los componentes que determinarán los comportamientos de los mismos. En la siguiente imagen, y a modo de resumen, se puede observar la morfología de estos bloques, en concreto los correspondientes al componente *Screen* y que son similares a los ya vistos con anterioridad y, que de manera conjunta, compondrán la aplicación.


*Eventos del componente Screen*

## 7. Resumen

Una *variable* representa un valor a través de un *identificador*, valor que podrá verse alterado mientras la aplicación se encuentre en ejecución. Para hacer uso de una variable, esta se debe crear. Cuando a una variable se le asigna por primera vez un valor, se dice que la variable ha sido *inicializada*. El identificador de una variable debe ser único y siempre deberá comenzar por un carácter letra, guion bajo '_' o dólar '$'.

En *MIT App Inventor 2* se definen dos tipos diferentes de variables, como son *globales* y *locales*. Una variable *global* se establece con la propiedad *global*

delante del identificador de la misma, significando que dicha variable será visible en cualquier ámbito de la aplicación. Sin embargo, al definir una variable *local*, esta llevará la propiedad *local* delante de su identificador y solo será visible dentro de su bloque de definición.

*Un procedimiento* es un conjunto de instrucciones agrupadas y que, entre todas, realizan una tarea conjunta, pudiendo además devolver un valor. Un procedimiento que devuelve algún dato se denomina *función*. Desde un procedimiento se podrán definir *parámetros de entrada*, que no son más que variables locales al procedimiento y no serán accesibles fuera de él.

La clasificación que *MIT App Inventor 2* hace de los diferentes bloques es análoga en algunos casos a los tipos de instrucciones que existen, como es el caso, por ejemplo, de instrucciones *aritméticas,* cuya finalidad es realizar operaciones matemáticas; *lógicas,* que realizan operaciones con valores booleanos; de *texto,* que manipulan cadenas de caracteres; o de *control*.

Un bloque *relacional* determina si dos valores son o no iguales, además de determinar si un valor es mayor o menor que otro. De otro modo, un bloque *lógico* realiza operaciones con valores lógicos entre las que se encuentran *y*, *o* y *no*.

Los bloques *condicionales* permitirán que una instrucción se ejecute solo si se cumple cierta condición definida. Se podrán realizar *anidaciones* entre diferentes bloques condicionales.

Un *bucle* es un tipo de estructura que permite la ejecución repetida de la instrucción o conjunto de ellas que se encuentren en su interior. A cada ejecución de las instrucciones contenidas en el bucle se le denomina *iteración* y el número de veces que un bucle es ejecutado lo determina una condición o será fijado por el desarrollador de antemano.

Un *evento* es un mecanismo usado en programación para notificar que ha ocurrido algo. Por ejemplo, se ha pulsado una tecla, se ha hecho clic con el ratón o se ha modificado la propiedad de algún componente.