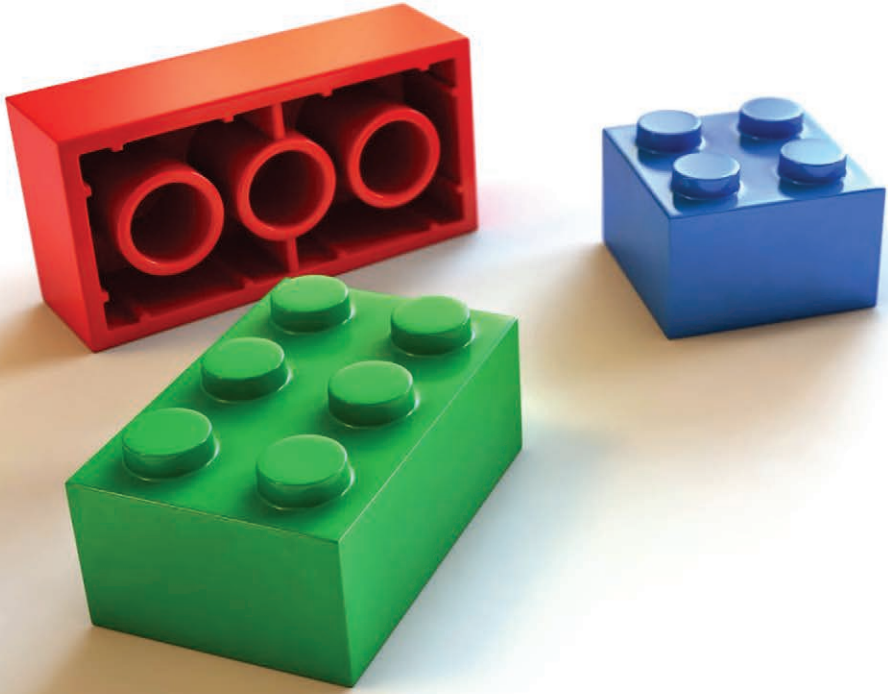
MANUAL

INTRODUCCIÓN A LA PROGRAMACIÓN

PRINCIPIOS BÁSICOS



ESCRITO POR: **LIC. JORGE ESPINAL**
Adobe Certified Instructor

SDQ TRAINING CENTER

MANUAL INTRODUCCIÓN A LA PROGRAMACIÓN

Diseño y portada: Joel Combes

Título original: © Manual introducción a la programación

1ª edición: Marzo 2015

SDQ Training Center

Calle Lea de Castro 256, Gazcue, Santo Domingo,

República Dominicana

info@sdq.com.do

www.sdq.com.do

LIC. JORGE ESPINAL

MANUAL
INTRODUCCIÓN A LA PROGRAMACIÓN

PRINCIPIOS BÁSICOS

SDQ TRAINING CENTER

1ª edición: Marzo 2015

CONTENIDO

INTRODUCCIÓN

Bienvenido al manual de programación básica. Este manual ha sido diseñado para aquellas personas que no tienen conocimiento alguno de programación y están interesadas en ingresar a este mundo, pero se les hace difícil debido a que las explicaciones que encuentran son muy complejas ya que la mayoría asume que se tiene conocimiento previo. En esta entrega compartiremos de una forma divertida y casi totalmente gráfica los conocimientos que son necesarios para abrir las puertas a las infinitas posibilidades de este universo de la programación.

A medida que avancemos veremos y comprenderemos como pensamos los humanos y como comunicar estas ideas a la computadora y hacer que esta nos entienda. Para ello utilizaremos tres formas distintas de escribir nuestras ideas, logrando así un mejor entendimiento, las expresaremos de manera gráfica utilizando imágenes para representar los enunciados, con bloques de Blockly que es una grandiosa herramienta que tienen a disposición las personas que están interesadas en aprender programación puesto que es un sistema desarrollado por Google para enseñar a las personas a programar de una forma divertida y amena. No esta de más decir que para obtener más provecho de este manual lo utilices simultáneamente con dicha herramienta, que podemos acceder gratuitamente a ella visitando la web: https://developers.google.com/blockly/ y también para trabajar con los ejercicios esta la web: https://blockly-games.appspot.com/.

La tercera forma de expresión es en modo de lenguaje de programación, que aunque no utilizaremos un lenguaje en especifico, mostraré las generalidades de estos y sus sintaxis con ejemplos, notas al margen, comentarios y ejercicios prácticos al final de cada tópico para que te pruebes a ti mismo como programador.

Espero que disfrutes y aprendas mucho en este pequeño viaje que comprende el manual y que sea un buen inicio al mundo de posibilidades infinitas de la programación.

1 Antes de iniciar a programar

En este capítulo

- Conocimiento
- Métodos del Conocimiento
- Lógica
- Algoritmo

CONOCIMIENTO

Antes de empezar a hablar de los fundamentos básicos de programación debemos de saber que es el conocimiento, esto hará que comprendamos mucho mejor la forma en la que pensamos y como ordenar nuestros pensamientos para poder externarlos correctamente.

Podríamos decir que el conocimiento es la facultad del ser humano para comprender por medio de la razón la naturaleza, cualidades y relación es de las cosas.

¿Sabías que?

En algunos escritos a estos conocimientos se les dice: empírico, sintético y analítico.

Nosotros los humanos tenemos básicamente dos tipos de conocimientos que son: el que nos proporcionan otras personas y el que experimentamos.

Adquirimos estos conocimientos a través de distintos métodos dependiendo de la capacidad intelectual de cada ser humano. Lo que nos lleva a hablar un poco de los métodos del conocimiento.

MÉTODOS DEL CONOCIMIENTO

El termino métodos del conocimientos se refiere nada más y nada menos que a las distintas formas que tenemos los seres humanos de aprender. De las cuales vamos a hablar un poco de las más comunes. Que son: la deductiva y inductiva.

Método deductivo

En palabras simples podríamos decir que este método lo utilizamos cuando sacamos nuestras propias conclusiones sobre alguna cosa, teniendo como parámetros **premisas diversas**.

Ejemplo:

Premisa 1:
Dios es amor,
Premisa 2:
El amor es ciego,
Premisa 3:
Mi vecino es ciego,
Conclusión:
Mi vecino es Dios.

Ejemplo:

Premisa 1:
Todas las frutas cítricas contienen vitamina C,
Premisa 2:
La piña es una fruta cítrica,
Conclusión:
La piña contiene vitamina C

Método inductivo

Este método lo utilizamos cuando sacamos nuestras propias conclusiones partiendo de **premisas similares.**

Ejemplo:

Premisa 1:
Hilda y Pedro tienen tres hijos: María, José y Juan
Premisa 2:
María es rubia,
Premisa 3:
José es rubio,
Premisa 4:
Juan es rubio,
Conclusión:
Por lo tanto todos los hijos de Hilda y Pedro son rubios.

Ejemplo:

Premisa 1:
El perro es mamífero y cuadrúpedo,
Premisa 2:
El gato es mamífero y cuadrúpedo,
Conclusión:
Por lo tanto todos los mamíferos son cuadrúpedos.

Método Deductivo

Considera que la conclusión se halla implícita dentro las premisas. Esto quiere decir que las conclusiones son una consecuencia necesaria de las premisas: cuando las premisas resultan verdaderas y el razonamiento deductivo tiene validez, no hay forma de que la conclusión no sea verdadera.

Método Inductivo

Obtiene conclusiones generales a partir de premisas particulares. Se trata del método científico más usual, en el que pueden distinguirse cuatro pasos esenciales:

1. Observación de los hechos para su registro.
2. Clasificación y el estudio de estos hechos.
3. Derivación inductiva que parte de los hechos.
4. Conclusión.

LÓGICA

La lógica en pocas palabras es la manera particular de pensar, de ver, de razonar o de actuar que se considera coherente, racional o de sentido común.

Entonces cuando adquirimos conocimientos sobre alguna cosa podemos utilizar la lógica para planear una secuencia de eventos que nos permite probar la validez de alguna proposición.

En este caso debemos comprender lo que es una proposición y lo que es un dato en si.

Debemos conocer el concepto de PROPOSICIÓN: que es un enunciado que tiene valor de verdad. Partiendo de esto debemos conocer que el valor de verdad es la capacidad de un enunciado de poder ser verdad o mentira.

Lógica

Método o razonamiento en el que las ideas o la sucesión de los hechos se manifiestan o se desarrollan de forma coherente y sin que haya contradicciones entre ellas.

Datos y proposiciones

Es importante diferenciar entre un Dato y una Proposición, ya que cuando decimos una palabra, ejemplo:

Tres
Camisa
Calcio
Perro

La Lógica Busca la Verdad

Siempre que utilizamos la lógica es para resolver problemas que tienen que ver con determinadas verdades.

Solo estamos diciendo un dato; pero cuando hablamos de proposiciones estamos enunciando una oración la cual tiene un dato y una acción que siempre dará un resultado verdadero o falso, ejemplo:

“El Rojo es un color.”
“El sol es caliente.”
“La camisa es negra.”

Es preciso conocer la Lógica Predicativa que se utiliza cuando comparamos si algo es mayor, menor, igual o diferente, y también el concepto de Lógica Proposicional que es la que conocemos básicamente como la Lógica de la Tabla de Verdad.

Para convertir datos en proposiciones debemos comparar datos : (si es mayor, menor, igual o diferente), cuando lo comparamos ya se convertirá en algo que es verdad o es mentira.

Un dato puede ser cualquier palabra o cantidad, el cual viéndolo de un modo aislado no sirve para nada. Sin embargo, cuando comenzamos a comparar valores los cuales no siempre son datos de verdades utilizamos la lógica Predicativa. En otro orden si comparamos proposiciones con conectores (y, o) o las negamos (no) estamos utilizando la lógica proposicional.

Los datos que tienen cantidades se pueden convertir en proposiciones.

Ejemplo1: A = 3, B=5

	DATOS			
ENUNCIADO	**A**		**B**	
A mayor que B es Verdad	3	>	5	V
A menor que B es Mentira	3	<	5	V
A igual a B es Mentira	3	=	5	V
A diferente de B es Verdad	3	!=	5	F

Ejemplo2: A=Mis amigos me aman, B=Mi familia me ama.

	DATOS			
ENUNCIADO	**A**		**B**	
Si mis amigos me aman y mi familia me ama soy feliz.	V	&&	V	V
Si mis amigos me aman y mi familia NO me ama no soy feliz.	V	&&	F	F
Si mis amigos NO me aman y mi familia me ama no soy feliz.	F	&&	V	F
Si mis amigos NO me aman y mi familia NO me ama no soy feliz.	F	&&	F	F

En el ejemplo 2 estamos extrayendo de la vida diaria proposiciones

	DATOS			
ENUNCIADO	**A**		**B**	
Cuando mis amigos me aman o mi familia me ama soy feliz.	V	||	V	V
Cuando mis amigos me aman o mi familia no me ama soy feliz.	V	||	F	V
Cuando mis amigos no me aman o mi familia me ama soy feliz.	F	||	V	V
Cuando mis amigos no me aman o mi familia no me ama no soy feliz.	F	||	F	F

Los datos por si mismo no tienen valor de verdad.

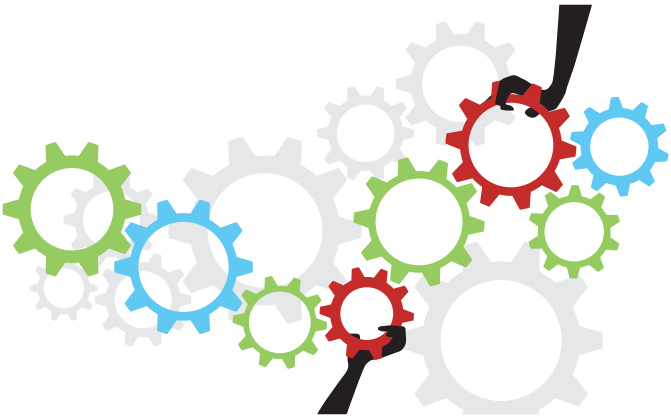
En algunos lenguajes de programación se utiliza dos signos && para representar la "Y", también dos barras verticales || para representar "O" y el símbolo de admiración ! para negar.

Operadores de Datos

- < Menor que
- > Mayor que
- = Igual
- != Diferente de

ALGORITMO

Un algoritmo es un procedimiento esquemático que comprende un conjunto de pasos secuenciales ordenados, para realizar una actividad específica. El algoritmo tiene las siguientes características:



Preciso

Cada instrucción tiene que ser clara y determinada a una acción.

Definido

Porque debe obtenerse los resultados determinados con las instrucciones de entrada.

Finito

Porque su diseño debe tener un numero limitado en cuanto a los pasos.

Ordenado

Porque tienen una secuencia de pasos.

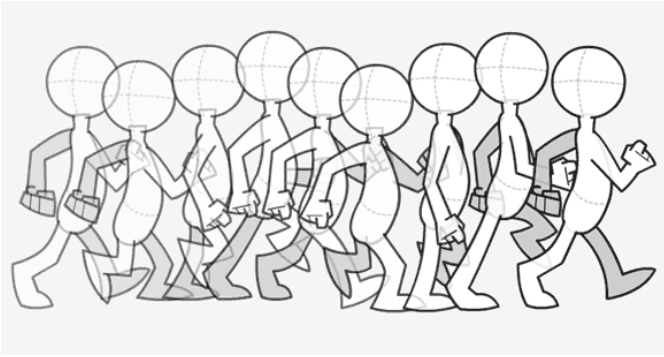
- Definir con claridad un problema
- Desarrollar un proceso de solución del problema

Todas las actividades que hacemos nosotros los seres humanos por simple que parezcan necesitan una especie de algoritmo, el cual escribimos y procesamos inconscientemente en nuestro cerebro.

Todos somos programamos desde que nacemos lo que sucede es que no somos conscientes de ello. Un bebe recién nacido cuando tiene hambre llora y cuando su madre se acerca a el, inmediatamente empieza a buscar sus pechos y una vez que lo encuentra, coloca su boca en posición y comienza a succionar.

Cuando vamos a hacer una acción como caminar por ejemplo. Las personas simplemente decidimos caminar y el cuerpo aparentemente hace todo solo. Pero no es así, el cerebro escribe

el algoritmo de caminar que como todo algoritmo está compuesto por una secuencia de pasos precisos, ordenados y finitos. Este lo



procesa y lo ejecuta todas las veces que sea necesario hasta que nuestros pies nos llevan al lugar de destino.

Entonces para programar en cualquier lenguaje de computadora sólo necesitamos saber dos cosas:

Que queremos que haga nuestro programa:

Esta es la parte más importante de la programación y de hecho a esto es lo que se le llama programar. Debemos tener bien claro nuestro objetivo y saber exactamente que es lo que queremos que haga nuestro programa al final y también debemos saber exactamente la secuencia lógica de pasos que el programa debe hacer para obtener ese resultado que se requiere.

Como se lo digo a la computadora:

Aquí es cuando elegimos en que lenguaje de computadora vamos a escribir nuestro programa y en que plataforma lo usaremos una vez esté programado. Aunque existen diversos lenguajes de programación todos tienen similitudes entre si, la sintaxis en la mayoría de ellos son iguales y los que no, son muy parecidas. Por esto cuando aprendemos un lenguaje se nos hace mucho más fácil aprender otros.

A continuación vamos a conocer las herramientas que necesitamos para tener una comunicación efectiva con la computadora y podamos transmitir con lujo de detalles lo que queremos cuando estamos programando. Aunque lo veremos de una manera simple y sencilla todos los tópicos que contiene este manual aplican para casi todos los lenguajes de programación que existen hoy día.

2 *Variables*

En este capítulo

- Introducción a las variables
- Tipos de datos

INTRODUCCIÓN A LAS VARIABLES

Una variable está formada por un espacio en el sistema de almacenaje que tiene un nombre simbólico el cual está asociado a dicho espacio.

Este espacio puede almacenar datos de diferentes tipos. Debido a que las variables contienen o apuntan a valores de tipos determinados, las operaciones sobre las mismas y el dominio de sus propios valores están determinadas por el tipo de datos en cuestión.

TIPO DE DATOS

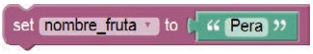
Cadenas de textos

(String)

Es una secuencia ordenada de longitud arbitraria (aunque finita) de elementos que pertenecen a un cierto lenguaje formal o alfabeto análogas a una fórmula o a una oración. En general, una cadena de caracteres es una sucesión de caracteres (letras, números u otros signos o símbolos).

Expresión gráfica:

Expresión de blockly:



Expresión en código de programación:

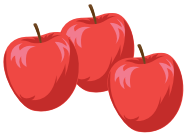
```
1   nombre_fruta = "Manzana";
2   nombre_fruta = "Pera";
```

Números:

Enteros (int):

Números que no pueden ser fraccionados.

Expresión gráfica:



Expresión de blockly:

Expresión en código de programación:

```
1   cantidad_frutas = 3;
```

En la mayoría de los lenguajes de programación es de carácter obligatorio terminar cada sentencia con un símbolo de punto y coma. [;]

Blockly es una aplicación desarrollada por Google que se esta utilizando para que las personas nuevas interesadas en programar puedan aprender de manera sencilla.

Para mas información puedes visitar:

https://developers.google.com/blockly

Decimales (float):

Números que pueden ser fraccionados.

Expresión gráfica:

Expresión de blockly:

Expresión en código de programación:

```
precio_fruta = 49.95;
```

En los distintos lenguajes de programación los números se colocan sin comillas, de lo contrario se contemplará como texto y no podrá ser operado como número.

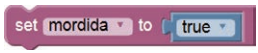
Lógico o booleano

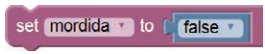
(Boolean)

También llamado booleano es en computación aquel que puede representar valores de lógica binaria, esto es 2 valores, valores que normalmente representan falso o verdadero.

Expresión gráfica:

Expresión de blockly:





Expresión en código de programación:

```
mordida = true;
mordida = false;
```

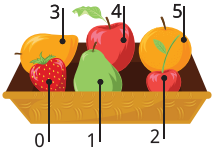
Listas o arreglos

(Array):

Son conjuntos de datos ordenados de forma lógica.

Expresión gráfica:

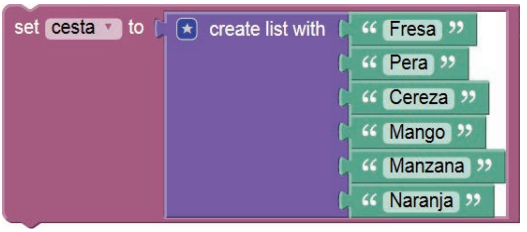
Cesta =



Cesta [0] = “Fresa”
Cesta [1] = “Pera”
Cesta [2] = “Cereza”
Cesta [3] = “Mango”
Cesta [4] = “Manzana”
Cesta [5] = “Naranja”

Cesta = [“Fresa”, “Pera”, “Cereza”, “Mango”, “Manzana”, “Naranja”]
[0] [1] [2] [3] [4] [5]

Expresión de blockly:



Expresión en código de programación:

```
cesta = [“Fresa”,
         “Pera”,
         “Cereza”,
         “Mango”,
         “Manzana”,
         “Naranja”];
```

Otra forma de expresar un arreglo en programación:

```
cesta[0] = “Fresa”;
cesta[1] = “Pera”;
cesta[2] = “Cereza”;
cesta[3] = “Mango”;
cesta[4] = “Manzana”;
cesta[5] = “Naranja”;
```

¿Sabías que?

En los arreglos puedes almacenar cualquier tipo de dato de los que hemos visto, es decir texto (string), números enteros (int), decimales (float), booleano, etc. De hecho un elemento de un arreglo puede ser otro arreglo o un objeto.

Aspecto técnico

Los arreglos son tratados como objetos en algunos lenguajes de programación y pueden ser creados con otros métodos, como por ejemplo:

- new Array();
- array();

Objetos

(Object):

Entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos). En palabras simples, que posee características y acciones.

Expresión gráfica:

fruta = { **Nombre:** "Manzana"
Color: "Rojo"
Precio: 45.95
Oferta: si
Hojas: 1 }

En los objetos, los atributos (cosas que tiene) son llamados propiedades y sus acciones (cosas que hace) son llamadas métodos.

Fruta.Nombre = "Manzana"	"Manzana"	**Texto** (String)
Fruta.Color = "Rojo"		**Texto** (String)
Fruta.Precio = 45.95	$ 45.95	**# Decimal** (Float)
Fruta.Oferta = true	oferta	**Si o no** (Boolean)
Fruta.Hojas = 1		**# Entero** (Int)

Expresión en código de programación:

```
1   fruta = {Nombre: "Fresa",
2            Color: "Rojo",
3            Precio: 45.95,
4            Oferta: true,
5            Hojas: 1};
```

Otra forma de expresar un objeto en programación:

```
1   fruta = new Object();
2   fruta.nombre = "Fresa";
3   fruta.color = "Rojo";
4   fruta.precio = 45.95;
5   fruta.oferta = true;
6   fruta.hojas = 1;
```

Autoevaluación

1. ¿Qué tipo de dato es la variable `fruta`?

```
1   canasta = ["Uva", "Pera", "Manzana"];
2   fruta = "Fresa";
3   oferta = true;
4   precio = 49.95;
```

A) Entero **B) Booleana** **C) Texto**
D) Flotante **E) Array**

2. ¿Qué valor tiene `canasta[1]`?

```
1   canasta = ["Uva", "Pera", "Manzana"];
2   fruta = "Fresa";
3   oferta = true;
4   precio = 49.95;
```

A) "Fresa" **B) true** **C) 49.95**
D) "Pera" **E) oferta**

3. ¿Qué tipo de dato es la variable `oferta`?

```
1   canasta = ["Uva", "Pera", "Manzana"];
2   fruta = "Fresa";
3   oferta = true;
4   precio = 49.95;
```

A) Entero **B) Booleana** **C) Texto**
D) Flotante **E) Array**

4. ¿Cómo obtenemos solo el dato del precio?

```
1   fruta =  {Nombre: "Fresa",
2             Color: "Rojo",
3             Precio: 45.95,
4             Oferta: true,
5             Hojas: 1};
```

A) Precio **B) fruta.Precio** **C) precio(fruta)**
D) precio.fruta **E) fruta(precio)**

5. ¿Qué tipo de dato es la variable `precio`?

```
1   canasta = ["Uva", "Pera", "Manzana"];
2   fruta = "Fresa";
3   oferta = true;
4   precio = 49.95;
```

A) Entero **B) Booleana** **C) Texto**
D) Flotante **E) Array**

Respuestas:
1-C, 2-D, 3-B, 4-B, 5-D

3 Operadores

En este capítulo

- Operadores aritméticos
- Operadores de comparación
- Operadores lógicos

OPERADORES ARITMÉTICOS

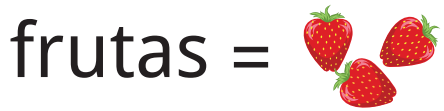
Los operadores son símbolos que indican la realización de alguna operación. En programación los operadores pueden ser de diferentes tipos:

Matemáticos o aritméticos: Es un símbolo matemático que indica que debe ser llevada a cabo una operación especificada. Los operadores aritméticos son:

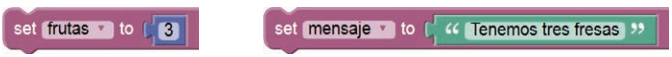
Asignación [=]:

Este operador asigna a lo que está a la izquierda el valor de la derecha.

Expresión gráfica:



Expresión de blockly:



Expresión en código de programación:

```
1   frutas = 3;
2   mensaje = “Tenemos tres fresas”;
```

Suma [+] :

Es utilizado para sumar el valor que esta en la izquierda del símbolo con el valor que esta a la derecha del símbolo.

Expresión gráfica:

Expresión de blockly:

Expresión en código de programación:

```
actuales = 2;
nuevas = 3;
total = actuales + nuevas;
// total = 5
```

Resta [-]:

Es utilizado para restar al valor que esta a la izquierda del símbolo el valor que esta a la derecha del símbolo.

Expresión gráfica:

Expresión de blockly:

Expresión en código de programación:

```
actuales = 3;
nuevas = 2;
total = actuales - nuevas;
// total = 1
```

*Multiplicación [*]:*

Es utilizado para multiplicar el valor que esta en la izquierda del símbolo por el valor que esta a la derecha del símbolo.

Expresión gráfica:

Expresión de blockly:

Expresión en código de programación:

```
1   cerezas = 3;
2   total = cerezas * 2;
3   // total = 6
```

División [/]:

Es utilizado para dividir el valor que esta a la izquierda entre el valor que esta a la derecha del símbolo.

Expresión gráfica:

Expresión de blockly:

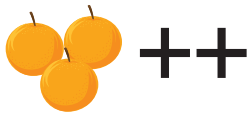
Expresión en código de programación:

```
1   mangos = 4;
2   total = mangos / 2;
3   // total = 2
```

Aumento [++]:

Es utilizado para aumentar uno al valor que esta a la derecha o a la izquierda del símbolo.

Expresión gráfica:



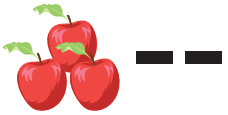
Expresión en código de programación:

```
1   naranjas = 3;
2   naranjas++;
3   // naranjas = 4
```

Disminución [--]:

Es utilizado para disminuir en uno el valor que esta a la derecha o a la izquierda del símbolo.

Expresión gráfica:



Expresión en código de programación:

```
1   manzanas = 3;
2   manzanas--;
3   // manzanas = 2
```

Módulo [%]:

Este operador divide el valor que está a la izquierda entre el valor que esta a la derecha pero devuelve el residuo de la división.

Expresión gráfica:

Expresión en código de programación:

```
1   peras = 5;
2   residuo = peras % 2;
3   // residuo = 1
```

Concatenación [+] [.]:

Este operador es utilizado para unir o enlazar datos. Dependiendo del lenguaje que se utilice el símbolo de concatenación puede variar pero usualmente suele ser un signo de más [+] y en otros un punto [.].

Expresión gráfica:

Expresión de blockly:

Expresión en código de programación:

```
1   fruta = "Fresa";
2   mensaje = fruta + ", fruta de temporada";
3   // "Fresa, fruta de temporada"
```

OPERADORES DE COMPARACIÓN

Este grupo de operadores se utilizan para comparar los valores que están a la izquierda con los valores que estén a la derecha. Estas operaciones sólo devuelven valores de verdad, es decir verdadero o falso. Los operadores de comparación son:

Mayor que [>]:

Compara que el valor de la izquierda sea mayor al valor de la derecha.

Expresión gráfica:

Expresión de blockly:

Expresión en código de programación:

```
nar_izq = 3;
nar_der = 2;
(nar_izq > nar_der)
// El resultado es verdadero (true)
fre_izq = 4;
fre_der = 5;
(fre_izq > fre_der)
// El resultado es falso (false)
```

En el código de programación las comparaciones se colocan en el lugar donde va la condición dentro de una estructura.

Véase el tema de las condiciones y/o bucles.

Menor que [<]:

Compara que el valor de la izquierda sea menor al valor de la derecha.

Expresión gráfica:

Expresión de blockly:

Expresión en código de programación:

```
fre_izq = 4;
fre_der = 5;
(fre_izq < fre_der)
// El resultado es verdadero (true)
```

```
5    nar_izq = 3;
6    nar_der = 2;
7    (nar_izq < nar_der)
8    // El resultado es falso (false)
```

Menor o igual [<=]:

Compara que el valor de la izquierda sea menor o igual al valor de la derecha.

Expresión gráfica:

Expresión de blockly:

Expresión en código de programación:

```
1    per_izq = 3;
2    per_der = 3;
3    (per_izq <= per_der)
4    // El resultado es verdadero (true)
5    man_izq = 4;
6    man_der = 3;
7    (man_izq <= man_der)
8    // El resultado es falso (false)
```

Mayor o igual [>=]:

Compara que el valor de la izquierda sea mayor o igual al valor de la derecha.

Expresión gráfica:

Expresión de blockly:

Expresión en código de programación:

```
1    nar_izq = 3;
2    nar_der = 5;
3    (nar_izq >= nar_der)
4    // El resultado es falso (false)
```

```
man_izq = 5;
man_der = 3;
(man_izq >= man_der)
// El resultado es verdadero (true)
```

Igual [==]:

Compara que el valor de la izquierda sea igual al valor de la derecha.

Expresión gráfica:

Expresión de blockly:

Expresión en código de programación:

```
per_izq = 3;
per_der = 2;
(per_izq == per_der)
// El resultado es falso (false)
man_izq = 3;
man_der = 3;
(man_izq == man_der)
// El resultado es verdadero (true)
```

Igual en valor y en tipo [===]:

Compara que el valor y el tipo de dato de la izquierda sea igual al valor y al tipo de dato de la derecha.

Expresión gráfica:

Expresión de blockly:

Expresión en código de programación:

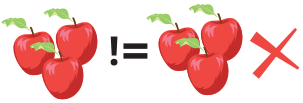
```
man_izq = 3;
man_der = 3;
(man_izq === man_der)
// El resultado es verdadero (true)
```

```
5    per_izq = 3;
6    nar_der = "3";
7    (per_izq === nar_der)
8    // El resultado es falso (false)
```

Diferente [!=]:

Compara que el valor de la izquierda sea diferente al valor de la derecha.

Expresión gráfica:



Expresión de blockly:

Expresión en código de programación:

```
1    per_izq = 3;
2    per_der = 2;
3    (per_izq != per_der)
4    // El resultado es verdadero (true)
5    man_izq = 3;
6    man_der = 3;
7    (man_izq != man_der)
8    // El resultado es falso (false)
```

Ejemplo como parte de un código en programación:

```
1    per_izq = 3;
2    per_der = 2;
3    suma = 0;
4    if(per_izq != per_der){
5      suma = per_izq + per_der;
6    } else {
7      suma++;
8    }
```

/*

En este caso como la variable **per_izq** es igual 3 y la variable **per_der** es 2, entonces la comparación en la linea 4 da como resultado verdadero porque 3 y 2 son diferentes, haciendo que la operación en la linea 5 sea ejecutada así la variable **suma** es igual a 5.

*/

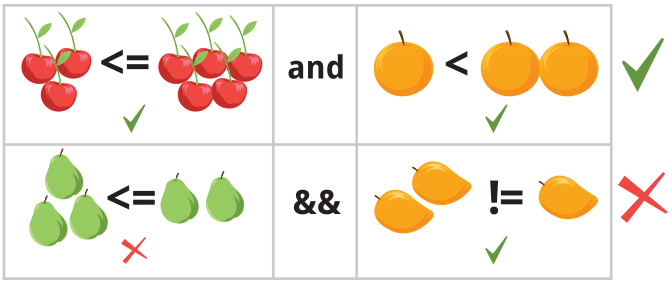
OPERADORES LÓGICOS

Este grupo de operadores trabajan como conjunciones entre los valores de verdad. Dependiendo del lenguaje los operadores lógicos pueden ser representado de formas distintas (con símbolos diferentes). Los operadores lógicos son:

Y [and] [&&]:

Este operador puede ser representado con la palabra “and” o con dos “&&” dependiendo del lenguaje en cual estemos trabajando. Dicho operador une dos o más sentencias con valor de verdad en una sola siendo esta nueva sentencia verdadera si todas las sentencias que la componen son verdaderas, y falsa si una o varias sentencias de las que la componen es falsa.

Expresión gráfica:



Expresión de blockly:

Expresión en código de programación:

```
1    a = 3;
2    b = 5;
3    c = 1;
4    d = 2;
5    (a <= b and c < d)
```

/*

El enunciado completo es verdadero porque ambas partes son verdaderas.

*/

```
6    (a <= d && d != c)
```

/*

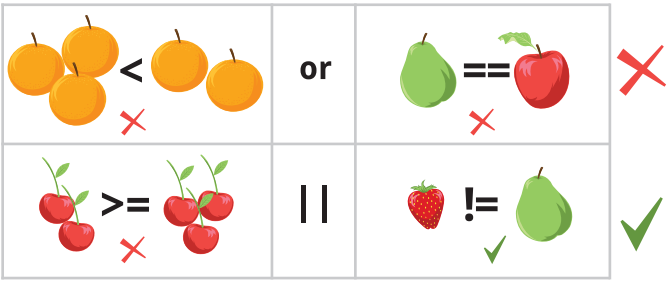
El enunciado completo es falso porque la primera parte es falsa y está separado por el operador **and**.

*/

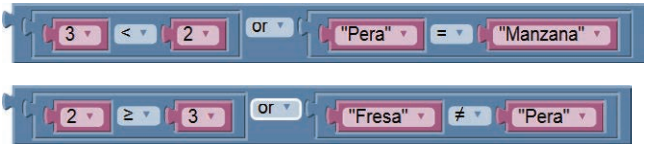
O [or] [||]:

El operador puede ser representado con la palabra “or” o con los símbolos “||” (dos barras verticales) dependiendo del lenguaje en cual estemos trabajando. Este operador une dos o más sentencias en una sola siendo esta nueva sentencia verdadera si por lo menos una de las sentencias que la componen es verdadera, y sólo es falsa si todas las sentencias que la componen son falsas.

Expresión gráfica:



Expresión de blockly:



Expresión en código de programación:

```
1    a = 3;
2    b = 2;
3    frutas = [“Pera”, “Manzana”, “Fresa”];
4    (a < b or frutas[0] == frutas[1])
```

/*

El enunciado completo es falso porque ambas partes son falsas.

*/

```
5    (b >= a || frutas[2] != frutas[0])
```

/*

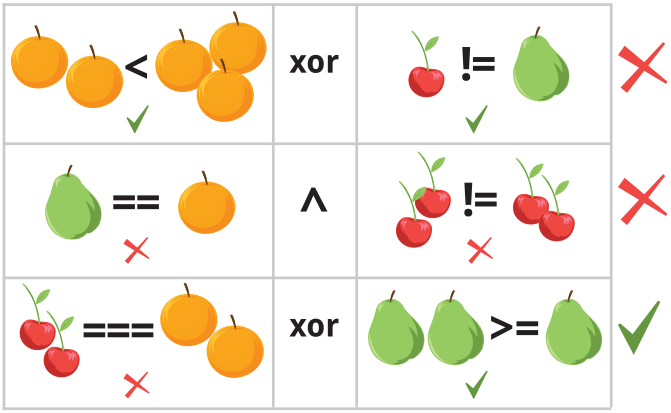
El enunciado completo es verdadero porque la primera parte es falsa y la segunda es verdadera y está separado por el operador **or**.

*/

Ninguno [xor] [^]:

Representado con la palabra “xor” o con el símbolo “^” dependiendo del lenguaje que estemos empleando. Este operador une dos o varias sentencias con valores de verdad siendo la nueva sentencia verdadera si todas las sentencias no tienen el mismo valor y falsa si todas las sentencias tienen el mismo valor de verdad.

Expresión gráfica:



Expresión en código de programación:

```
1   a = 3;
2   b = 2;
3   c = 1;
4   frutas = [“Naranja”, “Cereza”, “Pera”];
5   (b < a xor frutas[1] != frutas[2])
```

/*
El enunciado completo es falso porque ambas partes son verdaderas.
*/

```
6   (frutas[2] == frutas[0] ^ b != b)
```

/*
El enunciado completo es falso porque ambas partes son falsas.
*/

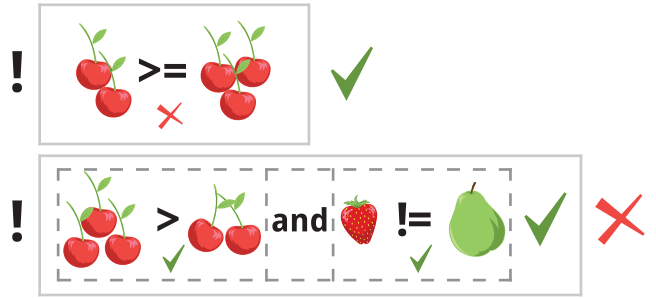
```
7   (frutas[1] === frutas[0] xor b >= c)
```

/*
El enunciado completo es verdadero porque las partes son diferentes, la primera es falsa y la segunda verdadera.
*/

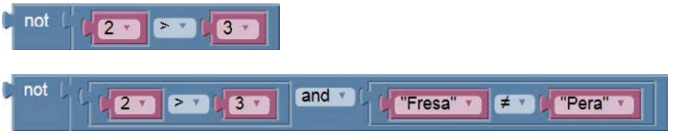
No (negación) [!]:

Este operador es representado con el símbolo "!" (admiración) y este niega el valor de la sentencia que este a su derecha.

Expresión gráfica:



Expresión de blockly:



Expresión en código de programación:

```
1   a = 3;
2   b = 2;
3   frutas = ["Fresa", "Pera"];
4   (!(b > a))
```

/*
El enunciado es verdadero porque **b** es menor que **a** pero se esta negando.
*/

```
5   (!(a > b && frutas[0] != frutas[1]))
```

/*
El enunciado completo es falso porque el resultado dentro del paréntesis es verdadero pero se esta negando el valor verdadero.
*/

Autoevaluación

1. En el código siguiente, conociendo que la variable total tiene valor cero (0) después de la operación ¿Cuál es el operador que falta?

```
1   fresas = 4;
2   total = fresas (operador) 2;
3   // total es 0
```

A) (+) Suma **B) (-) Resta** **C) (*) Multiplicación**
D) (%) Modulo **E) (--) Disminución**

2. ¿Cuál es el valor final de la variable tomates?

```
1   tomates = 4;
2   tomates = tomates - 2;
```

A) 4 **B) 2** **C) 0**
D) 6 **E) 3**

3. ¿Qué obtenemos al sumar estas variables?

```
1   dato1 = "Mangos";
2   dato2 = " Verdes";
3   total = dato1 + dato2;
```

A) "MangosVerdes" **B) 5** **C) 13**
D) "Mangos Verdes" **E) Mangos7**

4. ¿Cuál es el valor final de la variable bananas?

```
1   bananas = 0;
2   bananas = bananas + 2;
3   bananas++;
4   bananas = bananas * 2;
5   bananas--;
```

A) 0 **B) 2** **C) 4**
D) 6 **E) 5**

5. ¿Cuál sería la forma correcta para obtener el 18% de la variable precio?

```
1   precio = 120;
2   porcientoA = precio * 0.18;
3   porcientoB = precio * 18 / 100;
4   porcientoC = precio / 100 * 18;
```

A) porcientoA **B) porcientoB** **C) porcientoC**
D) Todas son correctas **E) Ninguna**

6. ¿Qué comprueba el operador utilizado en el siguiente código?

```
1    (manzanas != peras)
```

A) Igualdad **B) Mayor que** **C) Diferencia**
D) Mayor o igual que **E) Menor que**

7. ¿Cuál de estos operadores no es un operador lógico?

A) ! **B) and** **C) or**
D) xor **E) !=**

8. Dado el siguiente código ¿Qué resultado obtendríamos si colocamos un signo de admiración delante del paréntesis?

```
1    a = 2;
2    b = 3;
3    c = 4;
4    (a >= b && c > b || a < b && c != a)
```

A) Verdadero **B) Falso**

9. ¿Cuál es el resultado de siguiente expresión de blockly?

A) Falso **B) Verdadero**

10. ¿Cuál de estos operadores comprueba que los datos sean iguales y también del mismo tipo?

A) != **B) ==** **C) =**
D) ++ **E) ===**

Respuestas:
1-D, 2-B, 3-D, 4-E, 5-D, 6-C, 7-E, 8-B, 9-A, 10-E

4 Condicionales

En este capítulo

- If else
- Switch case

IF ELSE

If else:

If significa SI (condicional) en español. Su funcionamiento es simple. Se evalúa una condición, si es verdadera ejecuta un código, si es falsa, ejecuta otro código (o continúa con la ejecución del programa).

Expresión gráfica:

Expresión de blockly:

Expresión en código de programación:

```
1   if(hay_fresas){
2     jugo_de_fresas();
3   }
```

/* Si la variable **hay_fresas** es verdadera entonces se ejecuta el código **jugo_de_fresas**. Si la variable **hay_fresas** es falsa entonces no ejecuta el código **jugo_de_fresas**. */

Expresión gráfica:

Si (¿hay 🍓?) { **Hacer** un jugo de 🍓 }

Sino { **Servir** un 🥛 }

Expresión de blockly:

Expresión en código de programación:

```
1   if(hay_fresas){
2     jugo_de_fresas();
3   } else {
4     servir_vaso_de_agua();
5   }
```

/*

Si la variable **`hay_fresas`** es verdadera entonces se ejecuta el código **`jugo_de_fresas`**. Si la variable **`hay_fresas`** es falsa entonces se ejecuta el código **`servir_vaso_de_agua`**.

*/

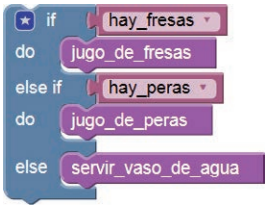
Expresión gráfica:

Si (¿hay 🍓 ?) { **Hacer** un jugo de 🍓 }

Sino si (¿hay 🍐 ?) { **Hacer** un jugo de 🍐 }

Sino { **Servir** un 🥛 }

Expresión de blockly:



¿Sabías que?

Entre if y else se puede colocar la cantidad de else if que consideremos, siempre y cuando la vayamos a utilizar.

Expresión en código de programación:

```
1   if(hay_fresas){
2     jugo_de_fresas();
3   } else if(hay_peras){
4     jugo_de_peras();
5   } else {
6     servir_vaso_de_agua();
7   }
```

/*

Si la variable **hay_fresas** es verdadera entonces se ejecuta el código **jugo_de_fresas**. Si la variable **hay_peras** es verdadera entonces se ejecuta el código **jugo_de_peras** Si la variable **hay_fresas** es falsa y la variable **hay_peras** también entonces se ejecuta el código **servir_vaso_de_agua**.

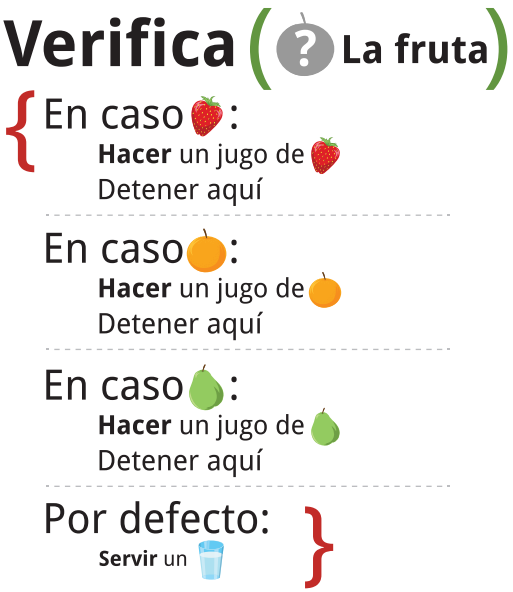
*/

SWITCH CASE

Switch case:

Es una estructura de control empleada en programación, se comprobar distintos valores para una variable, estos posibles valores que puede tener la variable son llamados casos. En la estructura podemos hacer que el programa haga algo diferente para cada uno de los casos y también podemos colocar un caso como predeterminado.

Expresión gráfica:



Expresión en código de programación:

```
1   switch(fruta){
2     case "Fresa":
3        jugo_de_fresa();
4     break;
5     case "Naranja":
6     case "Mandarina":
7        jugo_de_naranja();
8     break;
9     case "Pera":
10       jugo_de_pera();
11    default;
12       servir_vaso_de_agua();
13  }
```

Si queremos ejecutar un mismo código para diferentes casos, podemos colocar los distintos casos uno después del otro y luego el código que queremos ejecutar.

Tal como podemos ver en la linea 5 y 6.

Autoevaluación

1. ¿Qué obtenemos como resultado?.

```
1   canasta = ["Uva", "Pera", "Manzana"];
2   if(canasta[2] == "Manzana"){
3     imprimir("Hay manzana");
4   } else {
5     imprimir("No hay manzana");
6   }
```

A) Nada **B) Mensaje: No hay manzana** **C) true**
D) False **E) Mensaje: Hay manzana**

2. En caso de que la variable `numero` tenga valor 5 ¿Cuál de las funciones se ejecuta?

```
1   canasta = ["Uva", "Pera", "Manzana"];
2   switch(numero){
3     case 1:
4       jugo_pera();
5       break;
6     case 2:
7       jugo_manzana();
8       break;
9     default:
10      jugo_uva();
11  }
```

A) Ninguna **B) jugo_pera** **C) canasta**
D) jugo_uva **E) jugo_manzana**

3. ¿Cuantos `else if` podemos colocar entre `if` y `else`?

A) Solo 1 **B) Solo 2** **C) Los que necesitemos**
D) Ninguno **E) Solo 3**

4. Siendo la variable `activo` de valor booleano ¿Qué hace el código que se muestra a continuación?

```
1   if(activo){
2     activo = !activo;
3   }
```

A) Nada
B) Siempre que activo sea verdadera la cambia a falsa
C) Siempre que activo sea falsa la cambia a verdadera
D) La incrementa de uno en uno
E) B, C y D son correctas

5. Para que se ejecute la función guardar ¿Qué valor debe tener la variable documento?

```
switch(documento){
  case 1:
    copiar();
    break;
  case 2:
  case 5:
    guardar();
    break;
  case 3:
  case 4:
    borrar();
  default:
    reemplazar();
}
```

A) 1 **B) 2 o 3** **C) 2 o 5**
D) 3 o 4 **E) 0**

Respuestas:
1-E, 2-D, 3-C, 4-B, 5-C

5 Bucles o repeticiones

En este capítulo

- While
- Do
- For
- Foreach

WHILE

Un bucle o ciclo, en programación, es una sentencia que se realiza repetidas veces a un trozo aislado de código, hasta que la condición asignada a dicho bucle deje de cumplirse. Según el lenguaje de programación que estemos utilizando, los ciclos pueden ser de distintos tipos.

While:

While significa mientras en español. Mientras se da una condición la parte aislada del código es repetida tantas veces sea necesario

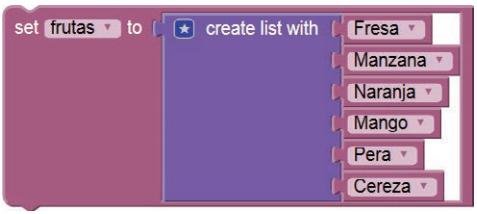
Los bucles o repeticiones no deben hacerse que se repitan infinitamente puesto que agotaría los recursos del sistema, repitiendo una acción que nunca termina.

Expresión gráfica:

Mientras ( ¿hay frutas?)

{ **Hacer** un jugo de una fruta de la canasta
Sacar una fruta de la canasta cada vez }

Expresión de blockly:



Expresión en código de programación:

```
1   frutas = ["Fresa", "Manzana", "Naranja",
2             "Mango", "Pera", "Cereza"];
3   i = 0;
4   while(frutas[i]){
5     hacer_jugo_de_frutas(frutas[i]);
6     retirar_fruta_de_lista(fruta[i]);
7     i++;
8   }
```

/*

Declaramos el array **frutas** y la variable **i**, asignando el valor **0**, que es el primer elemento del array. Luego se ejecuta el bucle **while**.

Si tratamos de efectuar este código mentalmente podriamos decir que para la primera vez, dado que la variable **i** es igual a **0**, el elemento del array **frutas[i]** en el inicio será **frutas[0]** que para el caso es **"fresa"**, entonces se ejecuta la siguiente parte del programa **hacer_jugo_de_frutas("fresa")**, **retirar_fruta_de_lista("fresa")** luego se incrementa la variable **i** con el operador **++**.

Para la segunda vez que se ejecuta el **while** la variable **i** es igual a **1**, siendo el elemento del array **frutas[1]** que es **"manzana"**, vuelve a ejecutarse **hacer_jugo_de_frutas("Manzana")**, **retirar_fruta_de_lista("Manzana")** y a incrementarse la variable **i** nuevamente teniendo el valor **2** para la tercera vez que se lea el bucle el elemento **frutas[i]** sería **frutas[2]** que es **"Naranja"** y asi sigue aumentando para las siguientes hasta llegar al último elemento de la lista.

*/

DO

Do:

Do significa hacer en español. Este ciclo es muy parecido al ciclo while con la diferencia de que actúa primero y luego verifica la condición, si la condición se mantiene se continúa repitiendo hasta que la condición cambie.

Expresión gráfica:

Ejecuta

{ **Hacer** un jugo de una fruta de la canasta
Sacar una fruta de la canasta cada vez }

Mientras ( ¿hay frutas?)

Expresión en código de programación:

```
1   frutas = ["Fresa", "Manzana", "Naranja",
2             "Mango", "Pera", "Cereza"];
3   i = 0;
4   do {
5     hacer_jugo_de_frutas(frutas[i]);
6     retirar_fruta_de_lista(fruta[i]);
7     i++;
8   } while(frutas[i])
```

/*

Esta forma es muy similar que la anterior, con la diferencia de que como la ejecución del código está primero que la condición, el **do** se efectúa siempre la primera vez sin importar la condición que este dentro de **while**.

*/

FOR

For:

For significa para en español. Este ciclo repite un código aislado para una variable que mantenga un rango numérico. Cuando la variable sale de el rango numérico definido la repetición se detiene.

Expresión gráfica:

Para (N Que inicia en cero (0)
N Siendo menor o igual a cinco (5)
N Aumenta uno (++))

{ **Hacer** un jugo de **Frutas** [N] }

Expresión de blockly:

Expresión en código de programación:

```
1   frutas = ["Fresa", "Manzana", "Naranja",
2             "Mango", "Pera", "Cereza"];
3
4   for(n = 0; n <= 5; n++) {
5     hacer_jugo_de_frutas(frutas[n]);
6   }
```

/*

El `for` es como contar ya sea aumentando o disminuyendo el valor de la variable que en este caso es `n` e inicia en `0`.

Haciendo que la primera vez que se ejecute la función `hacer_jugo_de_frutas(frutas[n])` sea con `frutas[0]` que para nuestro arreglo es `"Fresa"` e irá aumentando y lo hará con todas las frutas hasta que `n` sea menor o igual a `5`.

*/

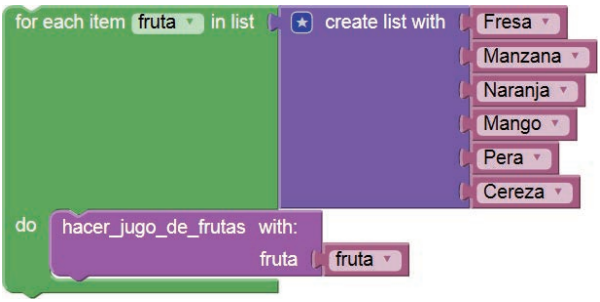
FOREACH

Foreach:

For each significa por cada. Este ciclo repite un código aislado por cada elemento de un array, también dependiendo del lenguaje que estemos utilizando podemos repetir un código por cada propiedad de un objeto.

Expresión gráfica:

Expresión de blockly:



Expresión en código de programación:

```
1   frutas = ["Fresa", "Manzana", "Naranja",
2             "Mango", "Pera", "Cereza"];
3
4   foreach(frutas as fruta) {
5     hacer_jugo_de_frutas(fruta);
6   }
```

/*

En el **foreach** la variable **frutas** en plural representa todos los elementos del arreglo y la variable **fruta** en singular representará cada una de las frutas dentro del array.

La función **hacer_jugo_de_frutas(fruta)** será ejecutada por cada ítem que este dentro del arreglo.

*/

Expresión en código de programación (javascript) :

```
1   frutas = [“Fresa“, ”Manzana”, ”Naranja”,
2             ”Mango”, ”Pera”, ”Cereza”];
3
4   for(fruta in frutas) {
5     hacer_jugo_de_frutas(fruta);
6   }
```

/*

En este caso especifico del lenguaje javascript podemos observar que en vez de **foreach** tenemos **for**. Pero dentro de los paréntesis tenemos una expresión un poco diferente de lo usual. Vemos **fruta in frutas**, que significa: por cada elemento en el arreglo.

*/

El **foreach** en algunos lenguajes como en el caso de javascript se puede lograr con **for** y cambiando el orden de las variables y **as** por **in**.

Autoevaluación

1. ¿Cuál de estos no es un bucle?.

A) while **B) do** **C) for**
D) switch **E) foreach**

2. ¿Qué valor tendrá la variable `i` después de la ejecución?

```
1   i = 1;
2   while(i <= 5){
3     i++;
4   }
```

A) 0 **B) 4** **C) 5**
D) -5 **E) 6**

3. ¿Cuál es el valor de la variable `conteo` al final de este código?

```
1   conteo = 0;
2   do {
3     conteo++;
4   } while (conteo < 0)
```

A) 1 **B) 0** **C) -1**
D) Ninguna **E) 2**

4. En el siguiente bucle ¿Cuál es el valor de la variable `n` en la linea 4?

```
1   for(n = 0; n > -5; n--){
2     imprimir(n);
3   }
4   imprimir(n);
```

A) -5 **B) -4** **C) 5**
D) 4 **E) 0**

5. ¿Que hace el siguiente código?

```
1   canasta = ["Uva", "Pera", "Manzana"];
2   foreach(canasta as fruta){
3     imprimir(fruta);
4   }
```

A) Muestra los números del 0 al 2
B) Muestra los "Uva", "Pera", "Manzana"
C) Nada
D) Muestra el nombre "Canasta"
E) Muestra el nombre "fruta"

6. En el siguiente bucle ¿Cuál es el valor final de la variable **`resultado`**?

```
1    resultado = 3;
2    for(n = resultado; n > 0; n--){
3      resultado = resultado * n;
4    }
```

A) 3 **B) -3** **C) 18**
D) 9 **E) 0**

7. ¿Cuál de estos bucles ejecuta primero la acción y después verifica la condición?

A) while **B) do** **C) for**
D) switch **E) foreach**

Respuestas:
1-D, 2-E, 3-A, 4-A, 5-B, 6-C, 7-B

6 Funciones

En este capítulo

- Introducción a las funciones

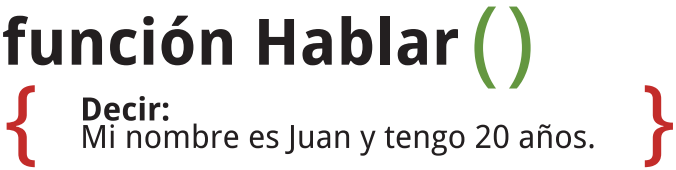
INTRODUCCIÓN A LAS FUNCIONES

En programación, una función es un grupo de instrucciones con un objetivo en particular y que se ejecuta al ser llamada desde otra función o procedimiento. Una función puede llamarse múltiples veces e incluso llamarse a sí misma en algunos lenguajes de programación (función recurrente).

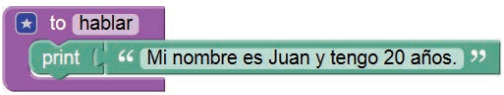
Las funciones pueden recibir datos desde afuera al ser llamadas a través de los parámetros y deben entregar un resultado.

Declaración:

Expresión gráfica:



Expresión de blockly:



Expresión en código de programación:

```
1   function hablar() {
2     imprimir("Mi nombre es Juan y tengo 20
      años");
3   }
```

/*

En la mayoría de los lenguajes de programación cuando se va a declarar una función, se coloca la palabra `function` y luego el nombre del que tendrá.

En caso de que la función no requiera parámetros, no se coloca nada dentro de los paréntesis.

*/

Llamada:

Expresión gráfica:

Hablar( )

Expresión de blockly:

Expresión en código de programación:

```
1   hablar();
```

/*

Al llamar la función solo colocamos el nombre y si no requiere ningún parámetro no colocamos nada dentro de los paréntesis.

*/

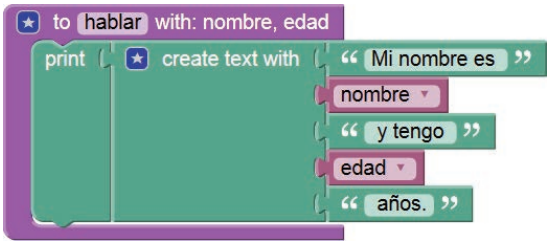
Parámetros:

Expresión gráfica:

Hablar(Nombre , Edad)

{ **Decir:**
Mi nombre es **Nombre** y tengo **Edad** años. }

Expresión de blockly:



Expresión en código de programación:

```
1   function hablar(nombre, edad){
2     imprimir("Mi nombre es " + nombre + " y
      tengo " + edad + "años.");
3   }
```

Las restricciones de los nombres de las funciones son igual a la de los nombres de las variables.

No deben tener espacios, guiones ni caracteres especiales.

/*

Los parámetros de una función son variables que se crean en el momento de la declaración de la misma y que pueden ser utilizadas dentro de estas.

Dichas variables pueden ser sustituidas por valores en el momento en que se llama la función.

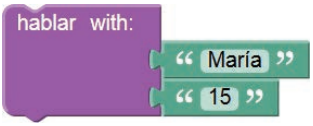
*/

Llamada con parámetros:

Expresión gráfica:

Hablar ("María", 15)

Expresión de blockly:



Expresión en código de programación:

```
1    hablar("María", 15);
```

/*

En este caso estamos llamando la función `hablar` y estamos sustituyendo `nombre` por `"María"` y `edad` por `15`.

El resultado que obtendríamos es el siguiente:

```
"Mi nombre es María y tengo 5 años."
```

*/

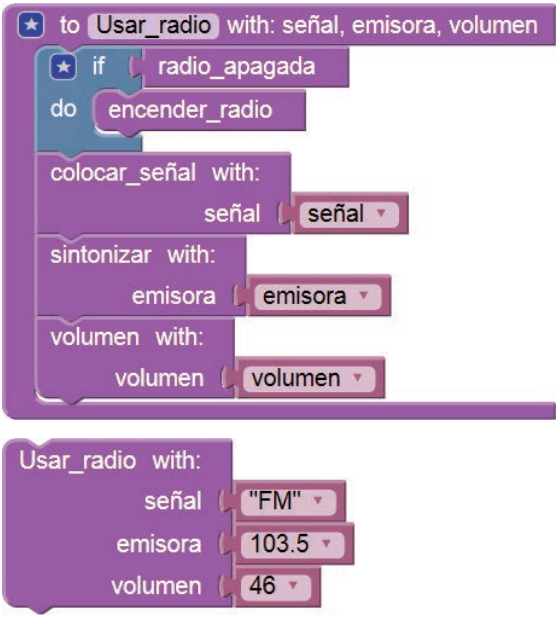
Expresión gráfica:

Usar_radio (señal, emisora, volumen)

```
{  si (¿radio apagada?)
   {  Encender radio  }
      colocar_señal ( señal )
      sintonizar ( emisora )
      volumen ( volumen )  }
```

Usar_radio ("FM", 103.5, 46)

Expresión de blockly:



Expresión en código de programación:

```
1   function usar_radio(señal, emisora, vol){
2     if(radio_apagada()){
3        encender_radio();
4     }
5     colocar_señal(señal);
6     sintonizar(emisora);
7     volumen(vol);
8   }
9
10  usar_radio("FM", 103.5, 46);
```

/*

Desde la primera línea hasta la octava declaramos la función **usar_radio**, la cual tiene tres parámetros que son: **señal**, **emisora** y **vol**.

Dentro de la función se pregunta si la radio esta apagada y si lo esta entonces se enciende, luego se coloca la señal, después la emisora y por ultimo el volumen.

Luego hay un llamado de la función **usar_radio** con señal en **"FM"**, emisora en **103.5** y volumen a **46** en la línea 10.

*/

Retorno:

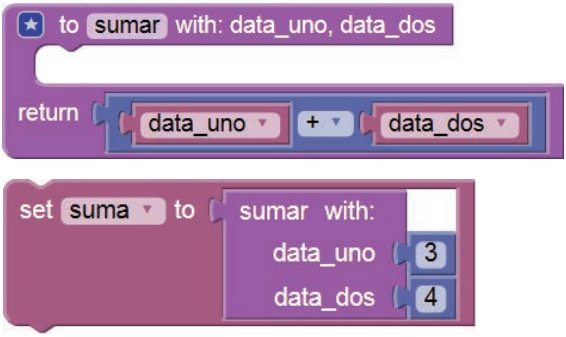
Cuando una función tiene como retorno algún dato, el llamado mismo de dicha función se convierte en el valor que esta retorna.

Expresión gráfica:

Sumar (data_uno, data_dos)

{ **Retornar:** data_uno + data_dos }

Expresión de blockly:



Expresión en código de programación:

```
1   function sumar(data_uno, data_dos){
2     return data_uno + data_dos;
3   }
4   suma = sumar(3, 4);
```

/*

La función **sumar**, pide dos parámetros: **data_uno** y **data_dos** luego dentro de esta se retorna la suma de dichos parámetros.

En la línea 4 se crea la variable **suma**, a la cual se asigna el llamado de la función **sumar** pasándose los valores **3** y **4**, esto hace que la parte del código **sumar(3, 4)** automáticamente sea igual a **7**, haciendo que la variable **suma** sea igual a **7**.

*/

Autoevaluación

1. ¿Cuál es la forma correcta para declarar la función hablar?.

A) hablar-function() **B) hablar()** **C) fn hablar()**
D) function hablar()

2. En caso de ser requerido ¿Qué colocamos dentro de los paréntesis en el momento de la declaración de una función?

A) Código de la acción **B) Parámetros** **C) El retorno**
D) Los operadores **E) Las condiciones**

3. En el código siguiente ¿Cuál es el valor de la variable `dato`?

```
1   function calcular(numero){
2     calculo = numero;
3     for(n = numero; n > 0; n--){
4      calculo = calculo * n;
5     }
6     return calculo;
7   }
8   dato = calcular(3);
```

A) 9 **B) 18** **C) 0**
D) 3 **E) -3**

4. ¿Cuál es la palabra que debemos utilizar si queremos que una función retorne un valor?

A) turn **B) save** **C) return**
D) function **E) back**

5. ¿Cuál de estas funciones tiene un nombre incorrecto?

A) cambiar() **B) encender-radio()** **C) salvar()**
D) set_vol() **E) obtener_emisora()**

Respuestas:
1-D, 2-B, 3-B, 4-C, 5-B

7 Clases y objetos

En este capítulo

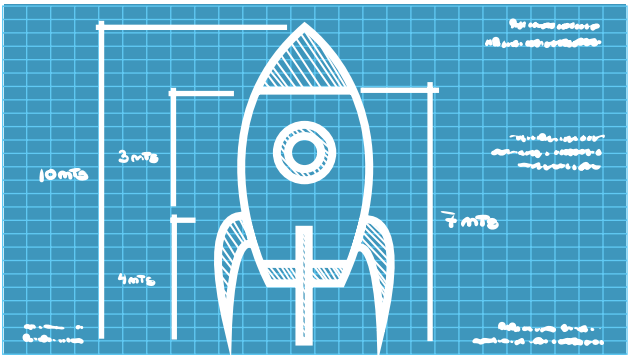
- Introducción a las clases
- Introducción a los objetos

INTRODUCCIÓN A LAS CLASES

Una clase es una plantilla para la creación de objetos de datos según un modelo predefinido. Las clases se utilizan para representar entidades o conceptos, como los sustantivos en el lenguaje. Cada clase es un modelo que define un conjunto de variables (el estado) y métodos apropiados para operar con dichos datos (el comportamiento). Cada objeto creado a partir de la clase se denomina instancia de la clase.

Crear clase

Expresión gráfica:



Expresión en código de programación:

```
1   class cohete {
2   }
```

Crear variables (Propiedades):

Las propiedades son un tipo especial de métodos. Debido a que suele ser común que las variables miembro sean privadas para controlar el acceso y mantener la coherencia, surge la necesidad de permitir consultar o modificar su valor mediante pares de métodos: GetVariable y SetVariable.

Propiedades:

Denominados, por lo general, datos miembros: esto es, los datos que se refieren al estado del objeto.

Que representan los datos asociados al objeto, o, lo que es lo mismo, sus propiedades o características. Los atributos y sus valores en un momento dado, determinan el estado de un objeto.

Expresión en código de programación:

```
class cohete {
  alto = 7;
  ancho = 3;
  color = "azul";
}
```

Crear Funciones (Métodos):

Los métodos implementan la funcionalidad asociada al objeto. Los métodos son el equivalente a las funciones en programación estructurada. Se diferencian de ellos en que es posible acceder a las variables de la clase de forma implícita. Cuando se desea realizar una acción sobre un objeto, se dice que se le manda un mensaje invocando a un método que realizará la acción.

Métodos:

Denominados, por lo general, funciones miembros: son funciones que pueden aplicarse a objetos.

Que acceden a los atributos de una manera predefinida e implementan el comportamiento del objeto.

Expresión en código de programación:

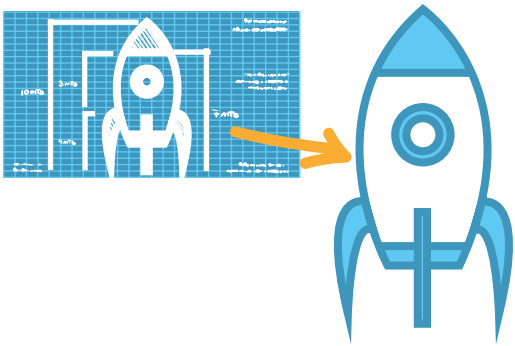
```
class cohete {
  alto = 7;
  ancho = 3;
  color = "azul";

  function despegar(){
  }

  function cambiarColor(color){
    this.color = color;
  }

  function aterrizar(){
  }
}
```

INTRODUCCION A LOS OBJETOS

Un objeto es una unidad dentro de un programa de computadora que consta de un estado y de un comportamiento, que a su vez constan respectivamente de datos almacenados y de tareas realizables durante el tiempo de ejecución. Un objeto puede ser creado instanciando una clase.

Crear instancia

Expresión gráfica:

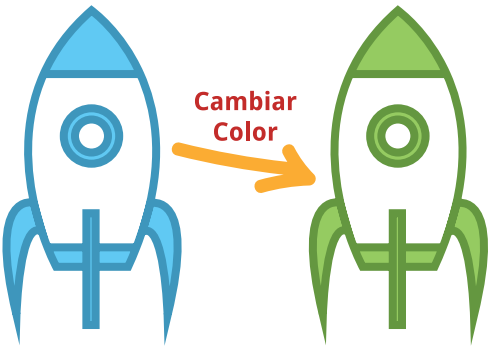


Expresión en código de programación:

```
1    nave = new cohete();
```

Cambiar propiedades

Expresión gráfica:



Expresión en código de programación:

```
1    nave = new cohete();
2    nave.color = "green";
```

Aplicar Funciones

Expresión gráfica:

Expresión en código de programación:

```
1   nave = new cohete();
2   nave.color = "green";
3   nave.despegar();
```

Autoevaluación

1. ¿Cuál es el termino técnico que se utiliza para llamar a un objeto que fue creado a partir de una clase?.

A) Instancia **B) Método** **C) Propiedad**
D) Copia **E) Dato**

2. ¿Cuál es la forma correcta para acceder a una propiedad de un objeto?

A) obj-prop **B) obj.prop** **C) obj,prop**
D) prop.obj **E) prop,obj**

3. ¿Cuál palabra debemos utilizar para crear una instancia de una clase?

A) object **B) instance** **C) new**
D) method **E) data**

4. ¿Cuál palabra debemos utilizar para crear una clase?

A) new **B) object** **C) method**
D) class **E) data**

5. ¿Cómo accedemos a un método de la clase a través de una instancia?

A) metodo.instancia **B) instancia,metodo()**
C) metodo(){ objecto } **D) objecto(){ metodo }**
E) objeto.metodo()

Respuestas:
1-A, 2-B, 3-C, 4-D, 5-E

8 Blockly

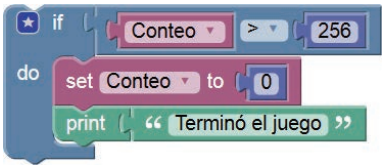
En este capítulo

- Introducción a Blockly
- Ejercicios en Blockly

INTRODUCCIÓN A BLOCKLY

Blockly es un editor visual que permite a los usuarios escribir programas enchufando bloques. Los desarrolladores pueden integrar el editor Blockly en sus propias aplicaciones web para crear una gran interfaz de usuario para los usuarios novatos . Un usuario podría crear un programa como este:

Expresión de blockly:



Entonces Blockly generaría el código correspondiente en JavaScript , Python , Dart, o algún otro lenguaje de programación:

Expresión en código de programación:

```
1   if(Conteo > 256){
2     Conteo = 0;
3     imprimir("Terminó el juego");
4   }
```

La aplicación puede ejecutar el código si lo deseas. Desde el punto de vista del desarrollador de aplicaciones, Blockly es básicamente un área de texto en el que el usuario escribe sintácticamente código perfecto.

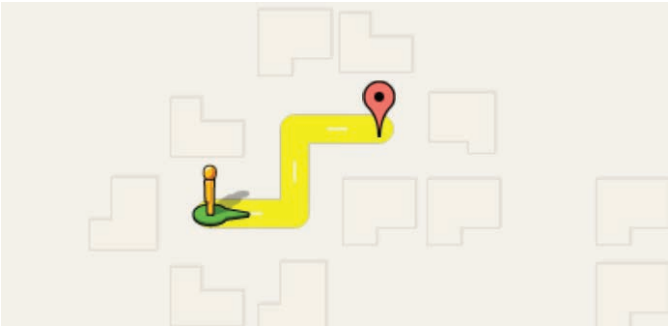
Para informarse más acerca de Blockly puede visitar el sitio web: https://developers.google.com/blockly/

EJERCICIOS EN BLOCKLY

¿Como podemos hacer que el personaje llegue a su destino?

Para hacer que el personaje pueda llegar a su destino con Blockly disponemos de una serie de funciones que nos servirán como herramientas para lograr el objetivo.

Expresión gráfica:

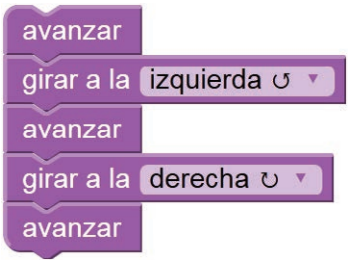


Nuestra lista de funciones disponibles son:

- `avanzar()`
- `girar(direccion)`

En la función `girar` podemos pasar un parámetro que es la dirección, es decir que podemos girar a la izquierda o la derecha con solo especificarlo.

Expresión de blockly:



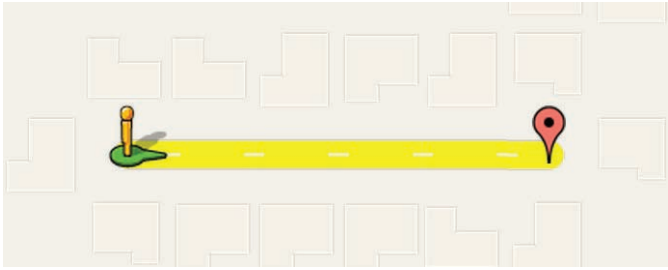
Expresión en código de programación:

```
1   avanzar();
2   girar("izquierda");
3   avanzar();
4   girar("derecha");
5   avanzar();
```

Segundo ejemplo

Esta vez para hacer que el personaje llegue a su destino tenemos que repetir la función `avanzar` varias veces.

Expresión gráfica:



Pero esta vez en vez de repetir la función `avanzar` vamos a utilizar uno de los bucles que conocemos como el `while`.

Expresión de blockly:

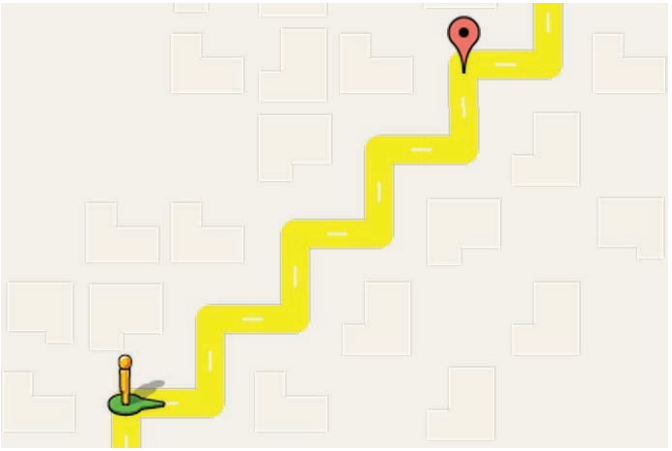
Expresión en código de programación:

```
1   while(no_ha_llegado()){
2     avanzar();
3   }
```

Tercer ejemplo

Ahora haremos un algoritmo un poco más extenso para repetir no sólo avanzar sino también los giros y en que dirección tiene que girar.

Expresión gráfica:



Expresión de blockly:

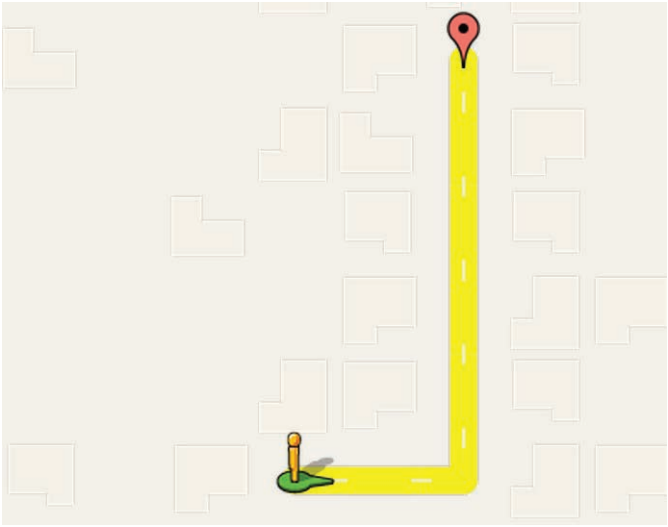
Expresión en código de programación:

```
1   while(no_ha_llegado()){
2     avanzar();
3     girar("izquierda");
4     avanzar();
5     girar("derecha");
6   }
```

Cuarto ejemplo

Para hacer que el personaje llegue a su objetivo esta vez, tenemos que colocar código fuera y dentro del bucle `while`.

Expresión gráfica:



Expresión de blockly:

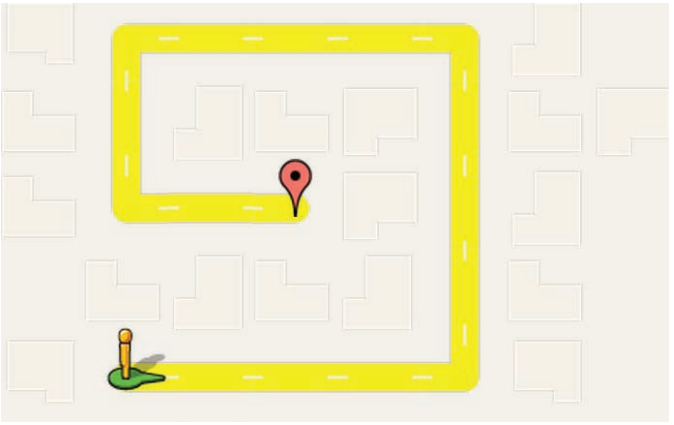
Expresión en código de programación:

```
1   avanzar();
2   avanzar();
3   girar("izquierda");
4   while(no_ha_llegado()){
5     avanzar();
6   }
```

Quinto ejemplo

Al listado de funciones que teníamos disponibles se le ha agregado una función nueva que se llama `hay_camino` que nos devuelve verdadero si hay camino, o falso si no lo hay, también podemos especificar en cual dirección, para esto tendremos que pasar un parámetro que sería la dirección en la cual queremos saber si hay camino o no.

Expresión gráfica:



El resultado de esta función lo utilizaremos con la condicional `if` para ejecutar una acción si hay camino en una de las direcciones que necesitamos.

Expresión de blockly:

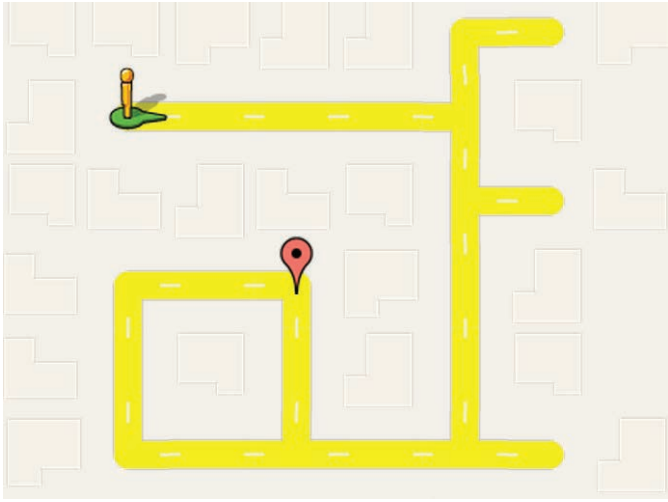
Expresión en código de programación:

```
1   while(no_ha_llegado()){
2     avanzar();
3     if(hay_camino("izquierda")){
4       girar("izquierda");
5     }
6   }
```

Sexto ejemplo

Continuamos con diversos retos para hacer que nuestro amigo llegue a su destino, en este caso podemos hacerlo con todo lo que ya hemos aprendido hasta ahora.

Expresión gráfica:



Expresión de blockly:

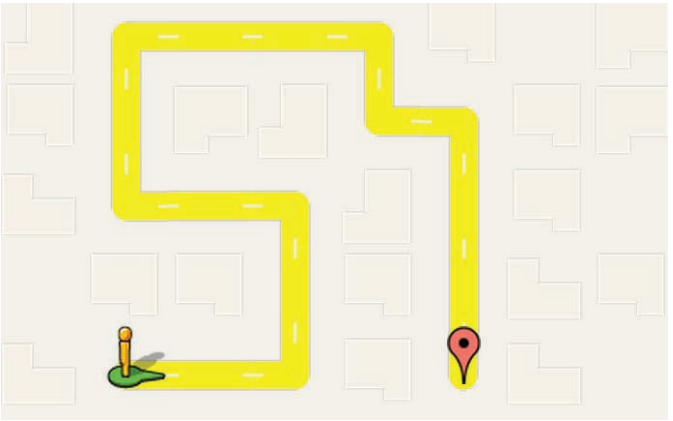
Expresión en código de programación:

```
1   while(no_ha_llegado()){
2     avanzar();
3     if(hay_camino("derecha")){
4       girar("derecha");
5     }
6   }
```

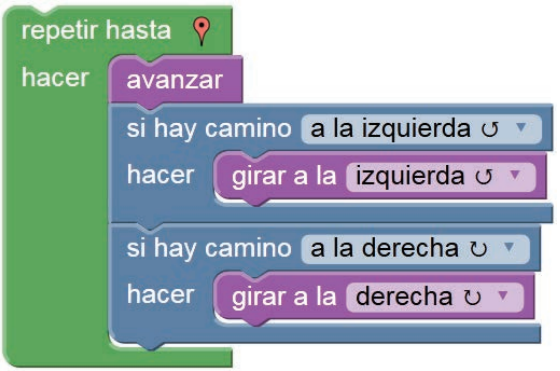
Séptimo ejemplo

Aquí tendríamos que hacer un algoritmo un poco diferente, pero de igual forma no es necesario agregar nada nuevo sino organizarlo de la forma más conveniente para cumplir con el objetivo.

Expresión gráfica:



Expresión de blockly:



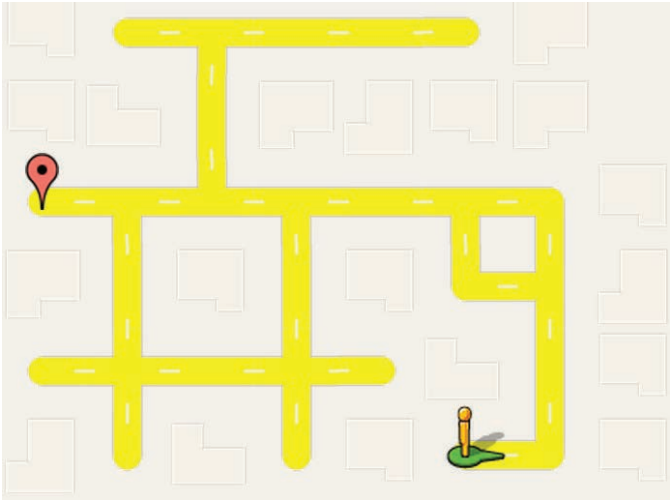
Expresión en código de programación:

```
1   while(no_ha_llegado()){
2     avanzar();
3     if(hay_camino("izquierda")){
4       girar("izquierda");
5     }
6     if(hay_camino("derecha")){
7       girar("derecha");
8     }
9   }
```

Octavo ejemplo

En esta práctica vamos a tener que agregar la parte del `else` a nuestro algoritmo en la condicional para logra que nuestro amigo llegue a su destino.

Expresión gráfica:



Expresión de blockly:

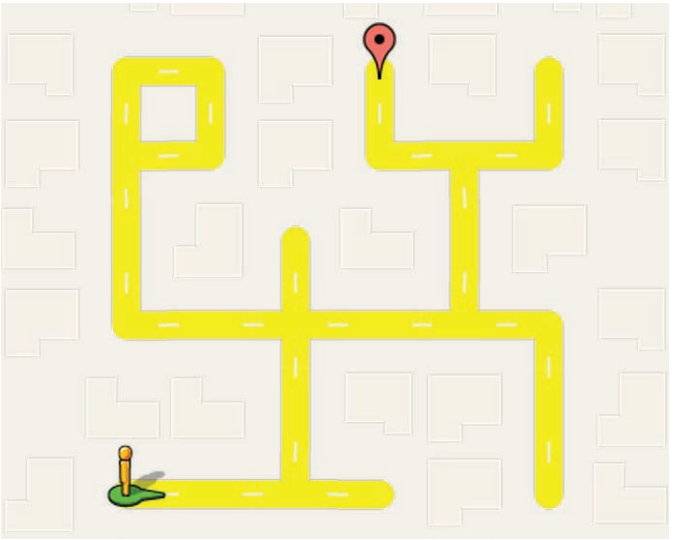
Expresión en código de programación:

```
1   while(no_ha_llegado()){
2     if(hay_camino("enfrente")){
3       avanzar();
4     } else {
5       girar("derecha");
6     }
7   }
```

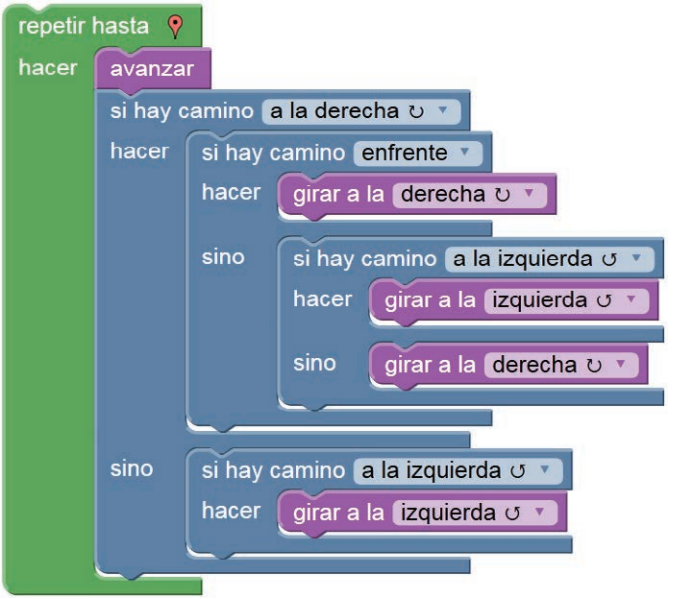
Noveno ejemplo

Este podríamos decir que es el algoritmo más complejo de todos los ejercicios que hemos visto hasta ahora, pero aunque este sea el más largo, no tiene nada nuevo, sino que simplemente tiene condicionales dentro de condicionales para ejecutar acciones más complejas.

Expresión gráfica:



Expresión de blockly:



Expresión en código de programación:

```
while(no_ha_llegado()){
  avanzar();
  if(hay_camino("derecha")){
    if(hay_camino("enfrente")){
      girar("derecha");
      } else {
        if(hay_camino("izquierda")){
          girar("izquierda");
        } else {
          girar("derecha");
        }
      }
  } else {
    if(hay_camino("izquierda")){
      girar("izquierda");
    }
  }
}
```

¿Cómo podemos hacer que el ave lleve el gusano al nido?

Esta práctica es muy similar a las anteriores. El objetivo es hacer que el ave tome el gusano y lo lleve al nido sin tocar la línea gris. Para esto tenemos a disposición dos funciones una de ellas se llama `no_tiene_gusano` que retorna verdadero si el ave no ha tomado el gusano y falso si el ave ya tomó el gusano.

La otra se llama `rumbo`, esta hace avanzar el ave en la dirección que queramos pero debemos especificar la misma en grados, para esto debemos pasar una parámetro.

Expresión gráfica:

Expresión de blockly:

Expresión en código de programación:

```
1   if(no_tiene_gusano()){
2     rumbo(0);
3     } else {
4     rumbo(90);
5   }
```

En este ejercicio tenemos un reto nuevo muy similar al anterior, para poder cumplir con el objetivo utilizamos las mismas funciones que ya conocemos.

Expresión gráfica:

Expresión de blockly:

Expresión en código de programación:

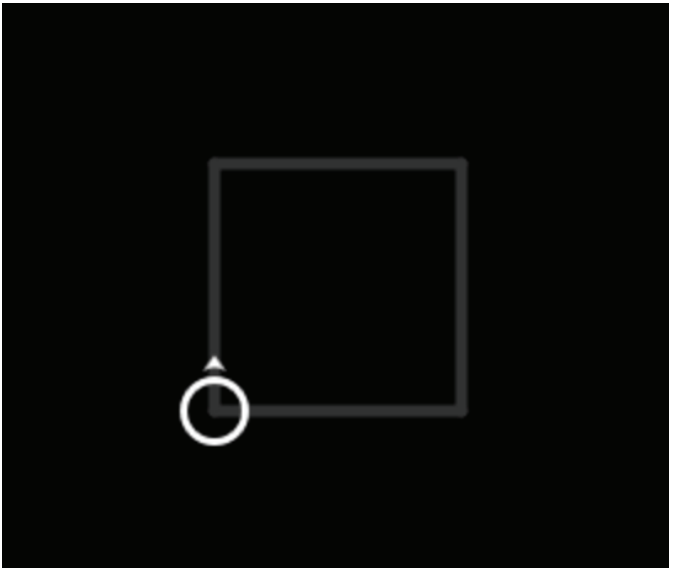
```
1   if(no_tiene_gusano()){
2     rumbo(300);
3     } else {
4     rumbo(60);
5   }
```

Ejercicios de dibujo

Los siguientes ejercicios presentan objetivos diferentes pero utilizando los mismos conocimientos de las practicas anteriores. Veamos como dibujamos con ellos.

Para dibujar disponemos ahora de unas funciones nuevas, una se llama **`avanzar`** y requiere como parámetro la distancia que se quiere avanzar. La otra se llama **`girar`**, esta requiere dos parámetros, uno es la dirección y el otro es el angulo.

Expresión gráfica:



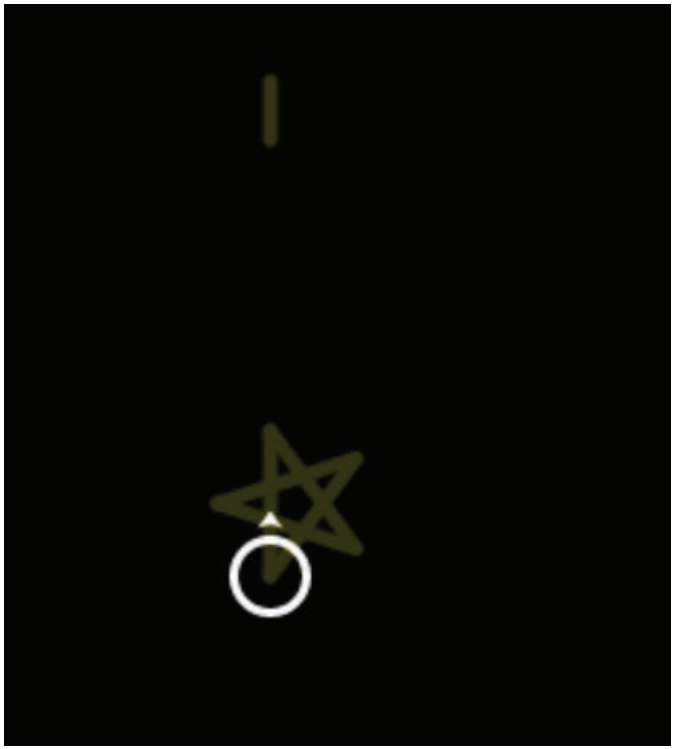
Expresión de blockly:

Expresión en código de programación:

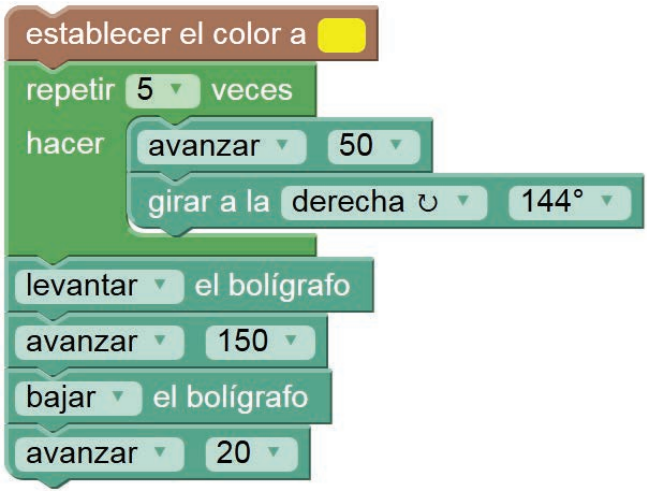
```
1   for(n = 1; n <= 5; n++){
2     avanzar(100);
3     girar("derecha", 72);
4   }
```

En esta practica se agregan más elementos como por ejemplo la función establecer_color, con esta podemos cambiar el color del lápiz con el parámetro color, también tenemos levantar y bajar que sirven para retirar el lápiz del papel y colocarlo.

Expresión gráfica:



Expresión de blockly:



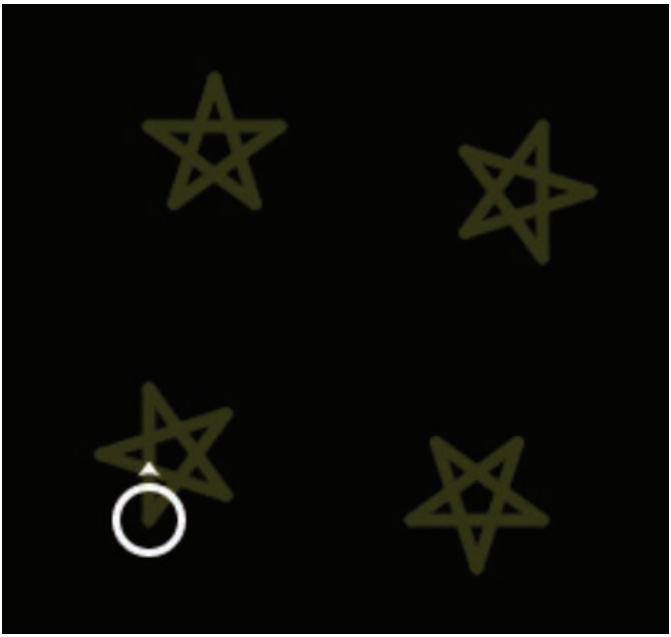
Expresión en código de programación:

```
1   establecer_color("yellow");
2   for(n = 1; n <= 5; n++){
3     avanzar(50);
4     girar("derecha", 144);
5   }
6   levantar();
7   avanzar(150);
8   bajar();
9   avanzar(20);
```

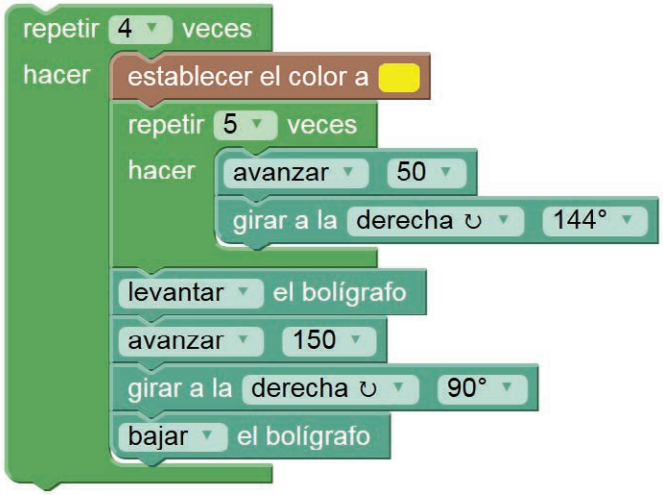
Este es un caso muy similar con la diferencia de que la estrella se dibuja simultáneamente cuatro veces. Esto se logra repitiendo cuatro veces el dibujo de una estrella.

Como la estrella se dibuja con un bucle vamos a necesitar un bucle dentro de otro para lograr el objetivo.

Expresión gráfica:



Expresión de blockly:



Expresión en código de programación:

```
1   establecer_color("yellow");
2   for(n = 1; n <= 4; n++){
3     for(e = 1; e <=5; e++){
4       avanzar(50);
5       girar("derecha", 144);
6     }
7     levantar();
8     avanzar(150);
9     girar("derecha", 90);
10    bajar();
11  }
```

Para trabajar de una forma más directa y que puedas hacerlo desde tu computadora con los ejercicios que acabamos de ver y muchos otros más te recomiendo visitar la página web de Blockly games en: https://blockly-games.appspot.com.

Hemos visto algunos ejercicios de Maze, Bird y Turtle, pero existen muchos más en los que puedes probar todo lo que haz aprendido y probarte a ti mismo como programador que ya eres.