PROGRAMACIÓN CON SCRATCH.

FUNDAMENTOS DE PROGRAMACIÓN.

1. INTRODUCCIÓN.

1.1. ¿QUÉ ES SCRATCH?

Un **programa** es simplemente un conjunto de **instrucciones** que le dicen a un ordenador (o a cualquier otra máquina programable) lo que tiene que hacer. Estas instrucciones se escriben usando un **lenguaje de programación**. Muchos lenguajes de programación son *textuales*, lo que significa que las instrucciones tienen la apariencia de comandos en formato texto (típicamente abreviaturas del inglés). Por ejemplo, para escribir "Hello!" por pantalla, la instrucción necesaria en distintos lenguajes de programación es:

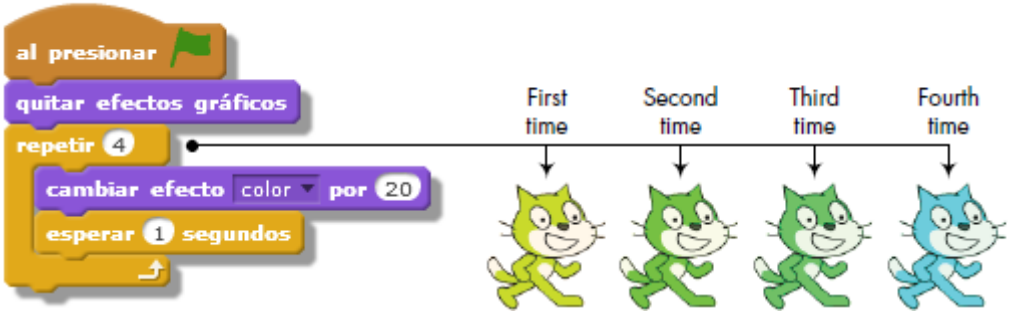
```
print('Hello!')                          Lenguaje Python
std::cout << "Hello!" << std::endl;       Lenguaje C++
System.out.print("Hello!");               Lenguaje Java
```

Aprender a programar en estos lenguajes implica aprender reglas sintácticas que pueden ser muy difíciles para los principiantes. Por el contrario, **Scratch** es un entorno de programación *visual* desarrollado por el Instituto Tecnológico de Massachusetts (MIT) para hacer el aprendizaje de la programación más sencillo y divertido. El sitio web de la plataforma Scratch es https://scratch.mit.edu/. Para utilizar el entorno de trabajo on – line, basta con clicar en "crear" (esta opción necesita tener instalado Adobe Flash Player). También puede usarse sin conexión a internet instalando la herramienta "Scratch Editor Offline" en nuestro ordenador (https://scratch.mit.edu/download).

Como ya hemos dicho, las instrucciones de Scratch no son textuales. En vez de ello, usa bloques gráficos que al interconectarlos entre sí forman programas. Por ejemplo, para hacer que el gato diga "Hello!", el programa que debemos escribir es simplemente:

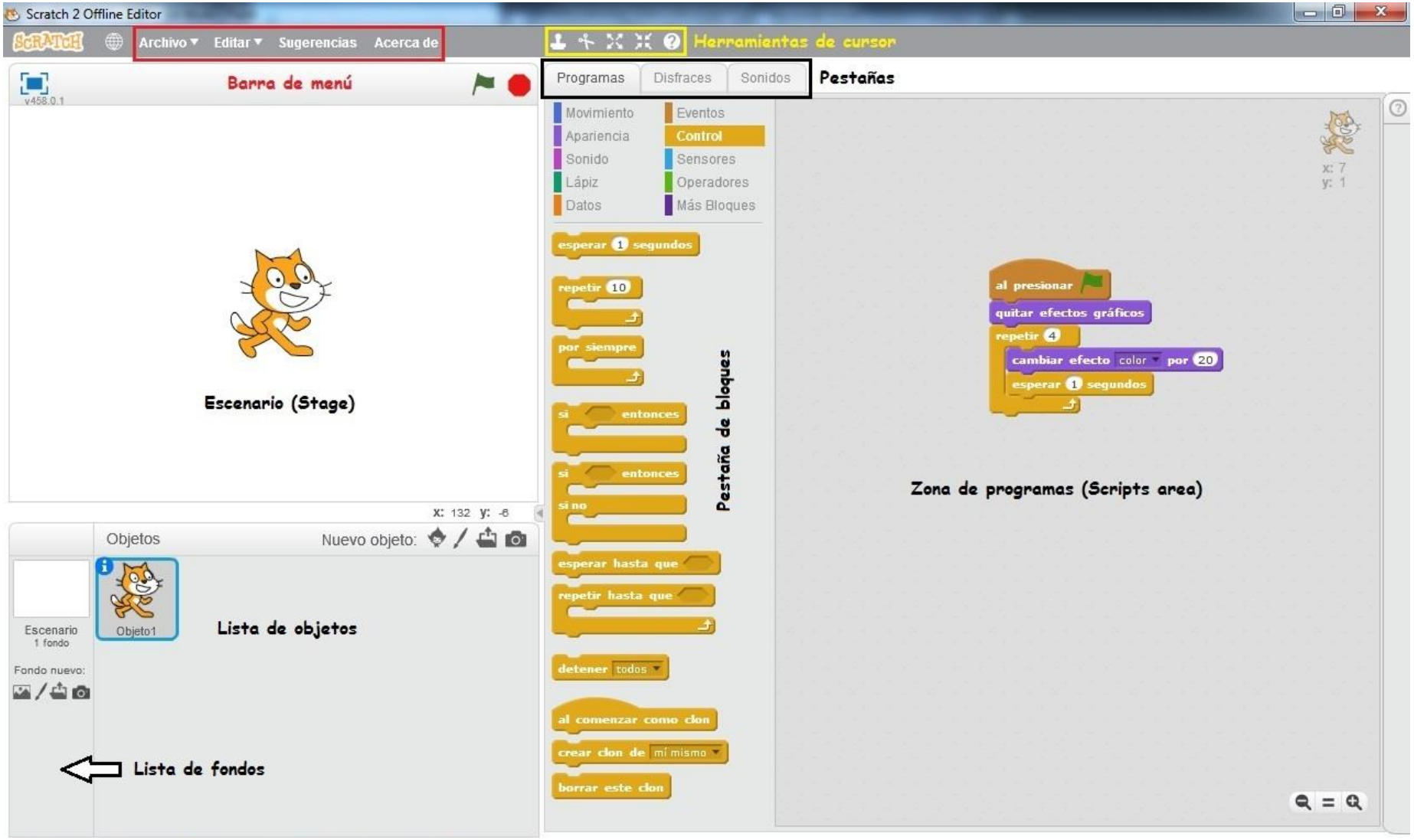
El gato que vemos en la figura es lo que en Scratch se denomina **objeto** (sprite). Los objetos entienden y obedecen las instrucciones que nosotros les proporcionamos. El bloque violeta de la figura le dice al gato que muestre el texto "Hello!" en un bocadillo de diálogo. Muchas de las aplicaciones que desarrollaremos en Scratch contienen múltiples objetos, y deberemos usar bloques para indicarle a cada objeto qué debe hacer (moverse, girar, decir cosas, reproducir un sonido, realizar una operación matemática, etc.).

Para programar en Scratch basta con unir bloques de color como si fueran piezas de un LEGO. A las pilas que generamos uniendo bloques se les denomina **programas** (scripts). La figura muestra un programa que cambia cuatro veces el color del objeto gato.



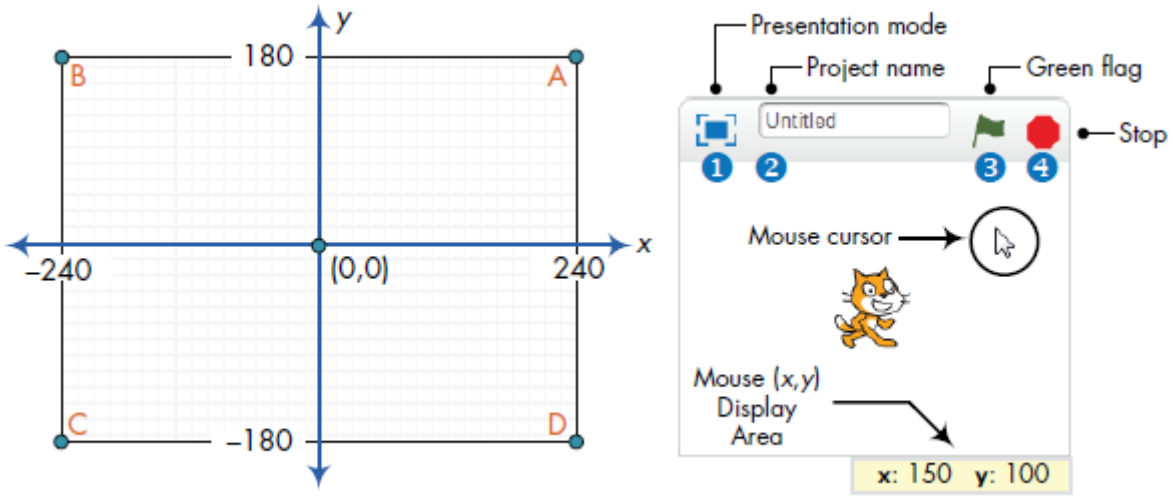
1.2. EL ENTORNO DE PROGRAMACIÓN SCRATCH.

En la pantalla del entorno de programación Scratch vemos tres partes principales, el **escenario** (stage), la **lista de objetos**, y la **pestaña de programas** (que incluye los bloques disponibles para la categoría seleccionada, y la zona de programas). También distinguimos algunas pestañas adicionales, para disfraces y sonidos, de las cuales hablaremos más adelante. Ahora vamos a comentar estas tres partes más detenidamente.



EL ESCENARIO.

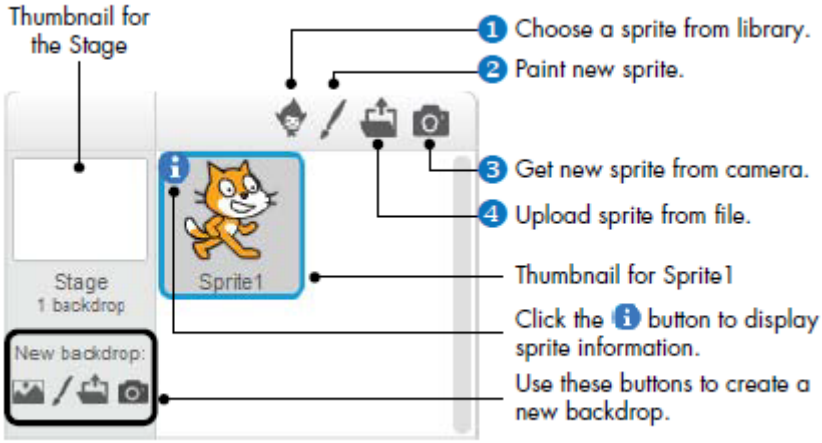
El **escenario** (stage) es la zona en la que los objetos se mueven, dibujan, e interactúan. El escenario tiene un tamaño de 480 pasos de ancho por 360 pasos de alto (ver figura). El centro del escenario está en las coordenadas $(x, y) = (0,0)$, y desde este origen, la coordenada x varía entre -240 y 240, y la coordenada y entre -180 y 180. Para hallar las coordenadas (x, y) de un punto cualquiera sobre el escenario, basta con mover el cursor a ese punto y observar los números que aparecen en la parte inferior derecha del escenario.



La barra sobre el escenario incluye varios controles importantes: El botón (1) pone el programa en modo presentación, y el escenario pasa a ocupar todo el monitor. La zona (2) muestra el nombre del proyecto actual. Los botones de la bandera (3) y del círculo rojo (4) permiten arrancar y parar el programa.

LA LISTA DE OBJETOS.

La **lista de objetos** muestra los nombres y miniaturas de todos los objetos de nuestro proyecto. Los proyectos nuevos comienzan con un escenario en blanco y un objeto disfrazado de gato.

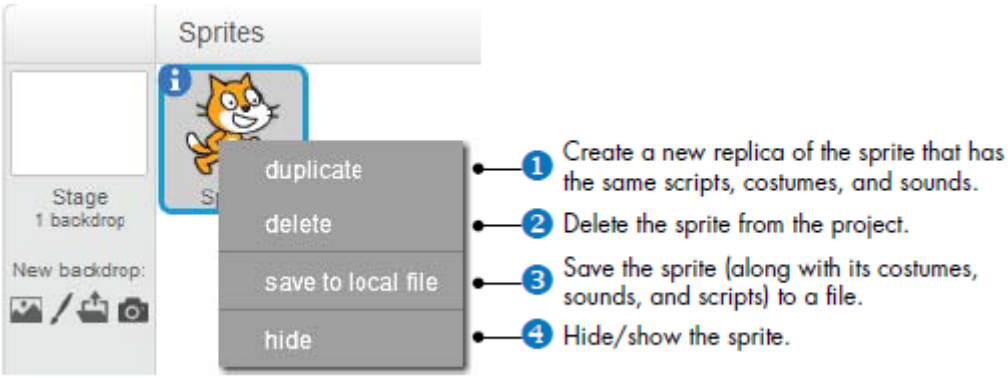


Los botones en la barra sobre la lista de objetos nos permiten añadir nuevos objetos a nuestro proyecto de una de estas cuatro formas: (1) seleccionándolo de la biblioteca de objetos; (2) dibujándolo mediante el editor gráfico de Scratch; (3) tomando una fotografía con una cámara conectada a nuestro ordenador; o (4) seleccionando un archivo en nuestro ordenador (una imagen descargada de internet, un dibujo que hayamos hecho en Paint, etc.).

Cada objeto de nuestro proyecto tiene sus propios programas, disfraces, y sonidos. Podemos seleccionar cualquier objeto y ver sus programas, disfraces, y sonidos clicando sobre la miniatura del objeto en la lista

de objetos, o haciendo doble clic en el propio objeto sobre el escenario. La miniatura del objeto seleccionado siempre aparece resaltada con borde azul en la lista de objetos. Al seleccionar un objeto, podemos acceder a sus programas, disfraces, y sonidos clicando sobre una de las tres pestañas que están encima del área de programas.

Ahora, si clicamos con el botón derecho del ratón sobre la miniatura de un objeto, se abre el menú contextual mostrado en la figura. La opción duplicar (1) copia el objeto (junto con sus programas, disfraces, y sonidos) y le da un nombre diferente. La opción borrar (2) elimina el objeto del proyecto. La opción guardar a un archivo local (3) permite exportar un objeto a nuestro ordenador como un archivo *.sprite2*. (Para importar a otro proyecto un archivo previamente exportado, basta con clicar el botón "cargar objeto desde archivo" en la barra sobre la lista de objetos). La opción esconder/mostrar (4) permite indicar si un objeto es visible o no en nuestro proyecto.



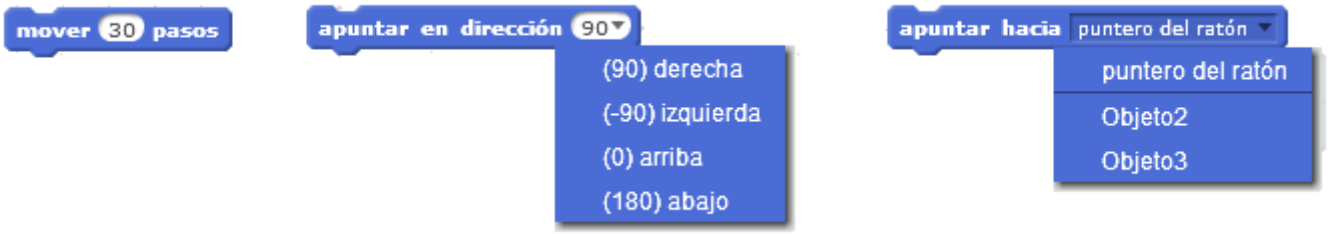
Además de haber miniaturas de los objetos, también tenemos una miniatura del fondo actual de nuestro escenario. El **fondo** (backdrop) es la imagen que actúa como fondo de nuestro escenario. El escenario posee sus propios programas, fondos, y sonidos. Cuando comenzamos con un nuevo proyecto, el escenario comienza por defecto con un fondo blanco, pero podemos añadir nuevos fondos con cualquiera de los cuatro botones bajo la miniatura del escenario. Para seleccionar el escenario con la intención de ver y editar los programas, fondos, y sonidos del escenario, basta con clicar sobre la miniatura del escenario.

LA PESTAÑA DE BLOQUES (PROGRAMAS).

Los bloques disponibles en Scratch se dividen en 10 **categorías**: movimiento, apariencia, sonido, lápiz, datos, eventos, control, sensores, operadores, y más bloques. Cada categoría de bloques es de un color, para poder diferenciarlos más fácilmente. Scratch ofrece más de 100 bloques, aunque algunos de ellos solo aparecen bajo ciertas condiciones (como por ejemplo, los bloques de la categoría "datos").

A modo de ejercicio, puedes probar algunos bloques básicos para ver qué hacen. Por ejemplo, el bloque "mover (10) pasos" de la categoría "movimiento" hace que el objeto se mueva 10 pasos sobre el escenario. Si lo añadimos otra vez a nuestro programa, el objeto se moverá otros 10 pasos. El bloque "decir (Hello!) por (2) segundos" de la categoría "apariencia" hace que el objeto diga "Hello!" en un bocadillo de diálogo durante dos segundos.

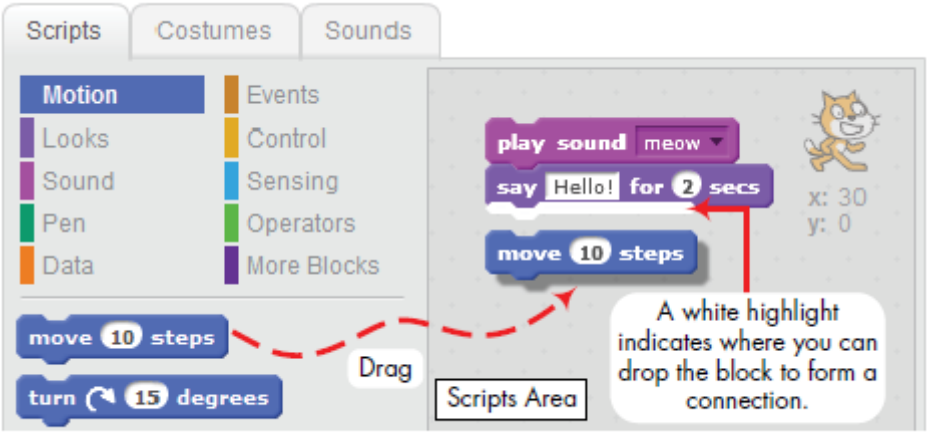
Algunos bloques requieren la entrada de datos (también llamados *argumentos*) que le dicen al bloque lo que debe hacer. El número 10 en el bloque "mover (10) pasos" es un ejemplo. (Aquí indicaremos que un bloque requiere de un argumento mediante un paréntesis vacío en el nombre del bloque, por ejemplo, "mover ( ) pasos"). La siguiente figura muestra las tres formas en las que los bloques nos permiten introducir o cambiar sus argumentos:



(1) Podemos cambiar el número de pasos en el bloque "mover (10) pasos" clicando en la zona blanca donde vemos el 10, e introduciendo un número distinto (30, en este caso).
(2) También podemos clicar en la flecha hacia abajo para ver una lista con una selección de todas las opciones disponibles y seleccionar una, como por ejemplo, en el bloque "apuntar en dirección (90)". Este comando en particular también tiene un área blanca editable para introducir manualmente un valor cualquiera.
(3) Otros bloques, como "apuntar hacia ( )" nos fuerzan a elegir un valor del menú desplegable.

ZONA DE PROGRAMAS.

Para hacer que los objetos hagan cosas, hemos de programarlos arrastrando bloques desde la pestaña de bloques a la zona de programas, e interconectarlos entre sí. Al arrastrar un bloque a la zona de programas, una línea de énfasis blanca nos indica dónde podemos colocarlo para formar una conexión válida con otro bloque. Ello elimina los errores sintácticos que la gente suele cometer al usar lenguajes de programación textuales.



No hace falta tener el programa terminado para comprobar cómo funciona. Al clicar en cualquier lugar del programa, éste se ejecuta de principio a fin, aunque el programa no esté terminado.

También podemos desmantelar una pila de bloques para comprobar si un fragmento del programa funciona correctamente. Esta será una estrategia muy útil para entender programas que sean muy largos. Para

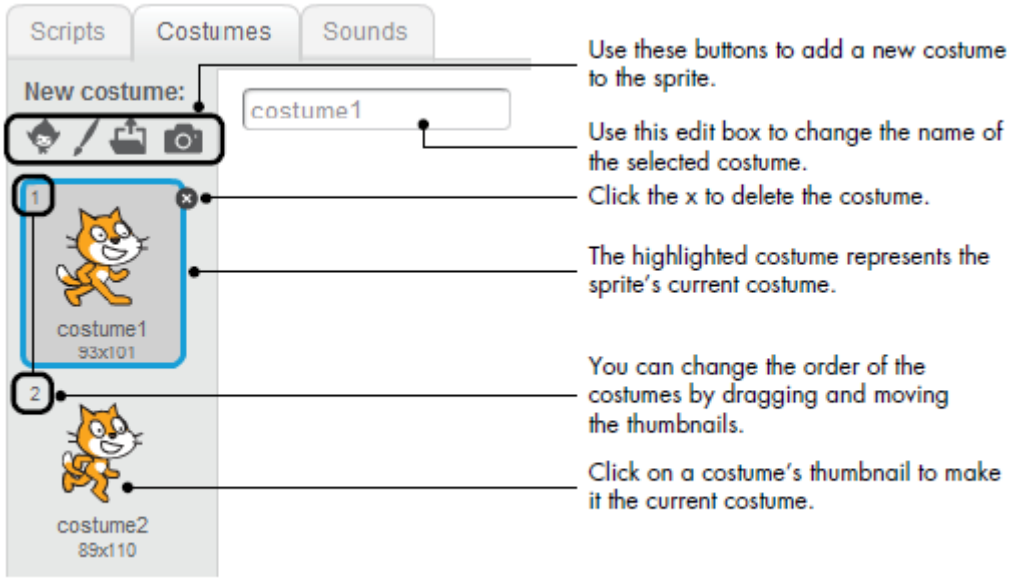
mover toda una pila de bloques, agarramos y arrastramos el bloque situado en la cima de la pila que queremos mover. Para desenganchar un bloque dentro de una pila de bloques, junto con todos los bloques que hay bajo él, lo agarramos y lo arrastramos.

Estas características nos permiten construir nuestro proyecto paso a paso. Podemos conectar pequeñas pilas de bloques, asegurarnos de que funcionan correctamente, y combinar con otras pilas que previamente hayamos testeado para formar programas más largos.

También podemos copiar toda una pila de bloques de un objeto a otro. Para ello, basta con arrastrar la pila desde la zona de programas del objeto origen a la miniatura del objeto destino en la lista de objetos.

PESTAÑA DE DISFRACES.

Podemos cambiar la apariencia de un objeto cambiando su disfraz, que es simplemente una imagen. La pestaña de disfraces contiene todo lo que necesitamos para organizar los disfraces de un objeto. Es algo así como su armario ropero. El armario puede contener muchos disfraces, pero el objeto solo puede llevar puesto un disfraz en cada momento. Por ejemplo, para cambiar el disfraz del objeto gato, clicamos en la miniatura del gato en la lista de objetos para seleccionarlo. En la pestaña "disfraces" vemos que el gato tiene 2 disfraces: disfraz1 y disfraz2. El disfraz resaltado con borde azul representa el disfraz actual del objeto.

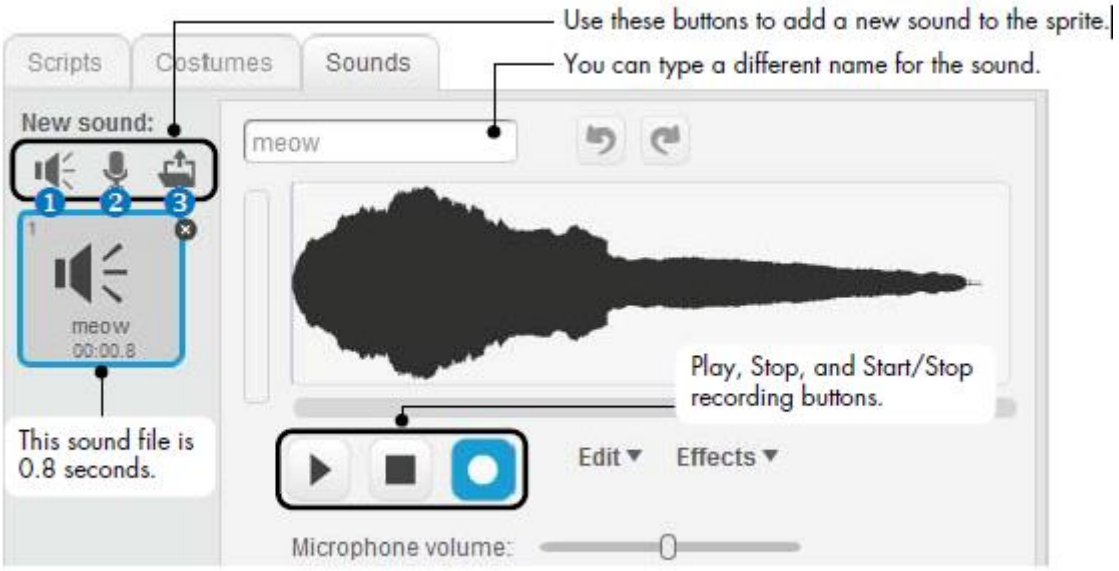


Si hacemos clic con el botón derecho del ratón sobre la miniatura de un cierto disfraz, aparece un menú desplegable con tres opciones: (1) duplicar, (2) borrar, y (3) guardar a un archivo local. La primera opción crea un nuevo disfraz idéntico al duplicado. La opción de borrar elimina el disfraz seleccionado. La última opción nos permite guardar el disfraz en nuestro ordenador como un archivo. Una vez guardado, podemos importar ese disfraz para usarlo en otro proyecto, usando el botón "cargar disfraz desde archivo" (tercer botón de la figura).

PESTAÑA DE SONIDOS.

Los objetos también pueden reproducir sonidos que amenicen nuestros programas. A un objeto se le pueden asociar varios sonidos. Por ejemplo, si nuestro proyecto contiene un objeto que representa un misil, podemos hacer que el misil reproduzca distintos sonidos dependiendo de la velocidad a la que se mueve, de lo lejos que esté, o de si impacta contra el blanco o falla.

Como vemos en la figura, los botones de la pestaña de sonidos nos permiten organizar los diferentes sonidos que un objeto puede reproducir. (Scratch incluso incorpora una herramienta de edición de sonidos, que aquí no explicaremos, pero que recomendamos probar).



Normalmente, solo utilizaremos los tres botones numerados en la figura. El botón (1) nos permite elegir un sonido de la biblioteca de sonidos de Scratch, el botón (2) sirve para grabar un sonido nuevo mediante un micrófono, y el botón (3) es para importar un archivo de sonido que tengamos guardado en nuestro ordenador. Scratch solo puede leer archivos de sonido de tipo MP3 y WAV.

PESTAÑA DE FONDOS.

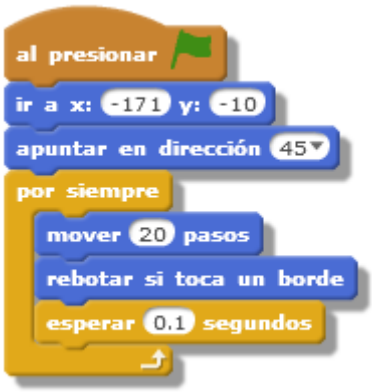
Cuando seleccionamos la miniatura del escenario en la lista de objetos, el nombre de la pestaña central cambia de "disfraces" a "fondos". Esta pestaña sirve para organizar los distintos fondos del escenario. Recordemos que la imagen de fondo del escenario la podemos cambiar a voluntad. Por ejemplo, si estamos creando un juego, puede que queramos comenzar mostrando un fondo con las instrucciones del juego, y después cambiar al fondo donde se desarrolla la partida. La pestaña de fondos es idéntica a la pestaña de disfraces.

INFORMACIÓN DEL OBJETO.

Podemos acceder a la zona de información de un objeto clicando en la pequeña "i" sobre la miniatura del objeto en la lista de objetos. La zona de información muestra el nombre del objeto, su posición y dirección actuales, su estilo de rotación, su estado de visibilidad, y si puede o no ser arrastrado cuando el programa está en modo presentación.

La caja de edición del nombre permite cambiar el nombre del objeto. Las coordenadas (x, y) muestran la posición actual del objeto; observa lo que ocurre al cambiar la posición del objeto sobre el escenario. La

línea de dirección indica la dirección en la que se moverá el objeto en respuesta a un bloque de movimiento. Arrastra la línea y observa cómo rota el objeto. Los tres estilos de rotación ("en todas direcciones", "izquierda - derecha", y "no rotar") controlan la forma en la que aparece el disfraz cuando el objeto cambia su dirección. Para entender el efecto de estos tres botones, crea el programa mostrado en la figura y clica en cada uno de los botones para ver qué ocurre:



La casilla "puede ser arrastrado" indica si el objeto puede o no ser arrastrado (usando el ratón) en el modo presentación. Cambia a modo presentación con esta casilla activada y desactivada, e intenta arrastrar el objeto a través del escenario para entender el efecto de esta opción. Por último, la casilla "mostrar" nos permite mostrar o esconder el objeto durante la fase de diseño del programa.

LA BARRA DE HERRAMIENTAS.

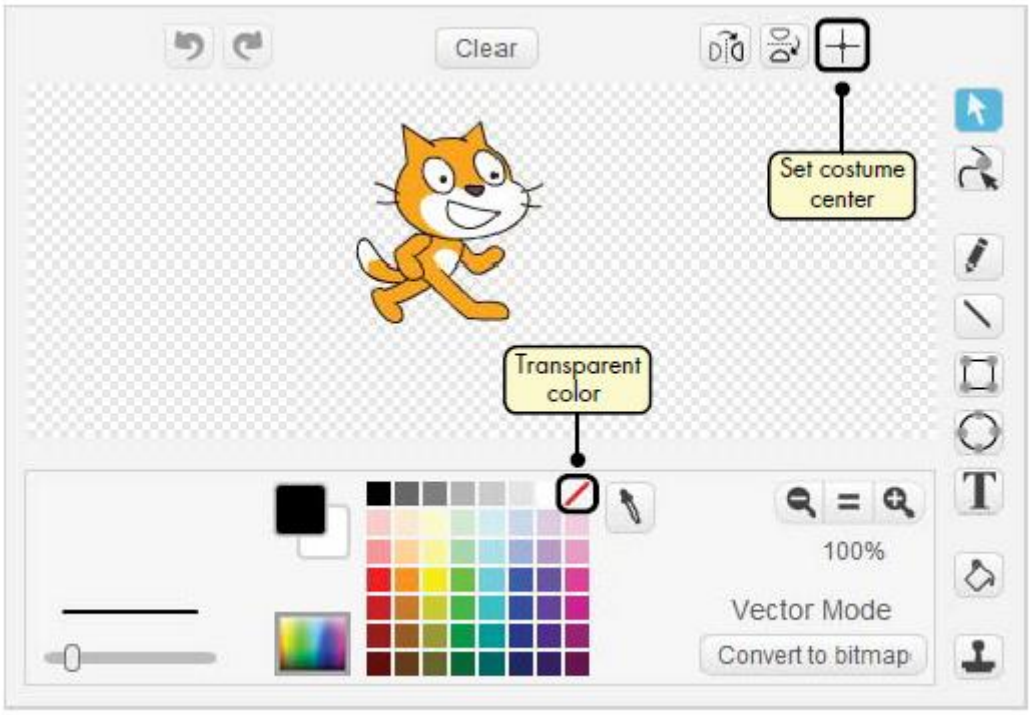
Vamos a echar un rápido vistazo a la barra de herramientas de Scratch. Podemos usar los botones de "duplicar" y "borrar" para copiar y borrar objetos, disfraces, sonidos, bloques, o programas. El botón "crecer" hace los objetos más grandes, mientras que el botón "encoger" los hace más pequeños. Para volver al cursor en forma de puntero, clica en una zona vacía sobre la pantalla. El menú de "idioma" nos permite cambiar el idioma de la interfaz de usuario.

En el menú de "archivo" podemos crear proyectos nuevos, abrir un proyecto ya existente en nuestro ordenador, y guardar un proyecto en nuestro ordenador. Los proyectos de Scratch 2 se guardan como archivos con extensión *.sb2*. Por último, en el menú "editar", la opción "recuperar borrado" nos devolverá el último bloque, programa, objeto, disfraz, o sonido que hayamos borrado; la opción "escenario pequeño" encoge el escenario y deja más espacio para la zona de programas; y la opción "modo turbo" incrementa la velocidad de ejecución de los programas del proyecto.

EDITOR GRÁFICO DE SCRATCH.

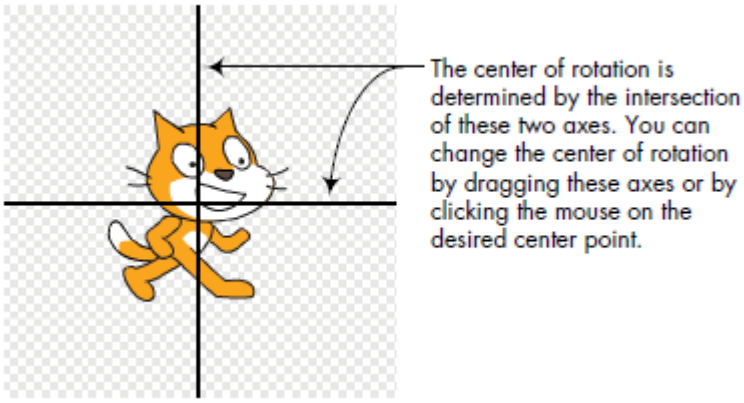
Podemos usar el editor gráfico de Scratch para crear o editar disfraces y fondos. (Aunque, por supuesto, tenemos la libertad de elegir nuestro programa de edición de imágenes favorito).

El editor gráfico de Scratch tiene múltiples opciones y herramientas, pero aquí nos centraremos en dos características principales: la opción "ajustar el centro del disfraz", y la opción "ajustar color transparente".

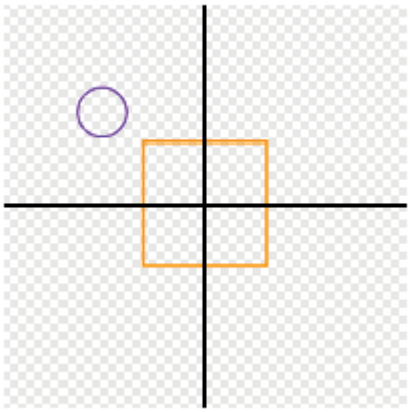


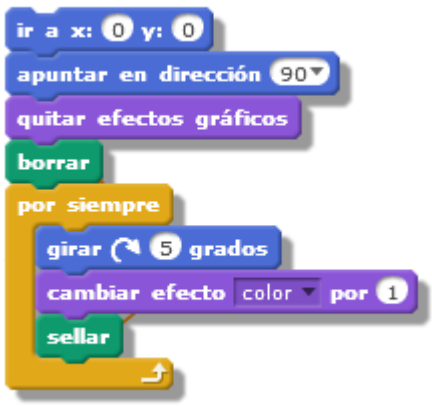
AJUSTAR EL CENTRO DE UN DISFRAZ.

Cuando le indicamos a un objeto que gire, lo hará en relación a un punto de referencia: el centro de su disfraz. El botón "fijar el centro del disfraz" nos permite elegir ese centro. Al clicar en ese botón, aparecen dos ejes cuya intersección indica el centro del disfraz. Para cambiar el centro de un disfraz, basta con arrastrarlo a la nueva posición. Para esconder los ejes, presiona la tecla ESC del ordenador.



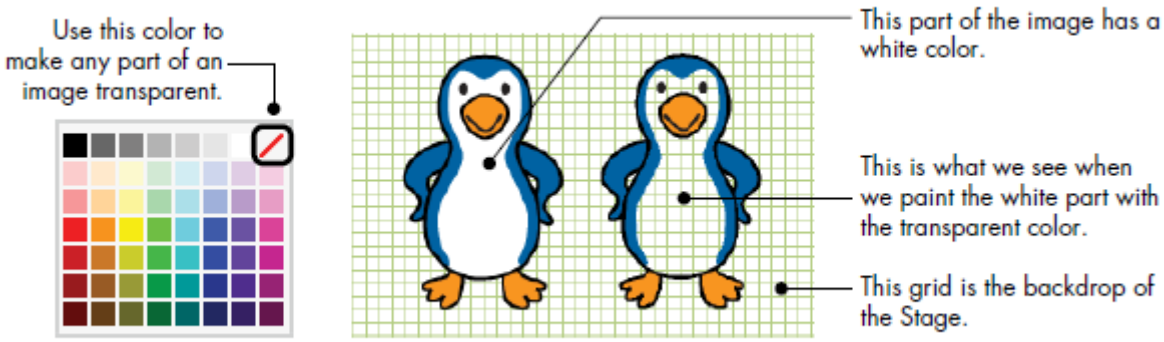
A modo de ejemplo, abre el archivo **centroRotacion.sb2**. Esta aplicación contiene un solo objeto con el disfraz y el programa mostrados en la figura. El centro del disfraz está ubicado en el centro del cuadrado. Ejecuta el programa y observa el patrón que dibuja. A continuación, edita el disfraz y reubica su centro en el centro del círculo, y ejecuta el programa de nuevo para ver cómo cambia el patrón dibujado.





AJUSTAR COLOR TRANSPARENTE.

Cuando dos imágenes se solapan, la imagen superior ocultará una parte de la imagen inferior. De forma similar, los objetos cubren parte del escenario sobre el que se mueven. Si queremos ver cómo es el escenario tras una imagen, hemos de usar el editor gráfico para hacer, al menos, una parte de esa imagen transparente (ver figura).



Para hacer que algo sea invisible, en la paleta de colores clicamos en el cuadrado con una diagonal roja y pintamos con color "trasparente".

1.3. PRIMEROS PASOS CON SCRATCH.

Ahora que ya conocemos el entorno de trabajo de Scratch, vamos a ponernos manos a la obra con dos actividades sencillas.

ACTIVIDAD GUIADA 1: HOLA MUNDO.

1) Selecciona el objeto "gato" en la lista de objetos, y clica sobre la pestaña "programas".
2) Usa el bloque "al presionar (bandera verde)" de la categoría "eventos" para indicar que el programa del gato comenzará al presionar el botón de la bandera.
3) Usa el bloque "tocar sonido (miau)" de la categoría "sonidos" para hacer que el gato maúlle.
4) Usa el bloque "decir (¡Hola mundo!) por (2) segundos" de la categoría "apariencia" para que el gato hable.
5) Ejecuta el programa clicando sobre el botón de la bandera. Observa cómo el gato hace lo que le has indicado en el programa.
6) Modifica el programa para que, en vez de comenzar al clicar en el botón de la bandera, comience cuando cliquemos sobre el propio gato. Modifica de nuevo el programa para que comience al presionar una tecla cualquiera del ordenador.
7) Guarda el programa en tu carpeta de trabajo con el nombre **guiada1.sb2**.

ACTIVIDAD GUIADA 2: EL BAILE DE MR. WIGGLY.

1) Crea un nuevo proyecto: archivo → nuevo.
2) Cambia el fondo del escenario: Selecciona el escenario y agrega desde la biblioteca el fondo "chalkboard". Observa que ahora tienes dos fondos disponibles, fondo1 y chalkboard.
3) Añade a Mr. Wiggly: Desde la biblioteca de Scratch (carpeta "fantasía"), agrega el objeto "nano". Verás que ahora hay dos objetos disponibles, objeto1 (el gato) y nano (Mr. Wiggly). Borra el objeto gato. Desplaza al objeto "nano" para que quede sobre el suelo del escenario.

4) Añade la música para el baile de Mr. Wiggly: selecciona el escenario, y agrega desde la biblioteca el sonido "eggs". Ahora nuestro escenario tiene dos sonidos disponibles: pop y eggs.
5) Haz el programa del escenario:

- El programa comienza al clicar el botón de la bandera.
- Añadimos un bucle infinito para que el programa repita indefinidamente las instrucciones contenidas dentro del bucle. Para ello, usa el bloque "por siempre" de la categoría "control".
- Dentro del bloque "por siempre", añadimos la instrucción "tocar sonido (eggs)" de la categoría "sonido".
- Dentro del bloque "por siempre", añadimos la instrucción "esperar (16) segundos" de la categoría "control".

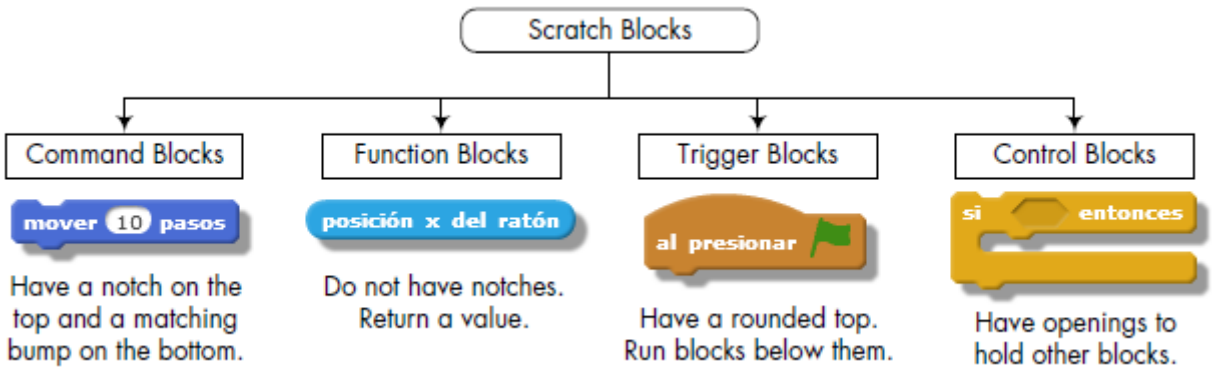
6) Haz el programa de Mr. Wiggly:

- El programa comienza al clicar en la bandera.
- Añadimos un bucle infinito "por siempre".
- Pongamos a Mr. Wiggly a bailar: Dentro del bucle infinito, Mr. Wiggly empieza moviéndose 25 pasos y esperando 1 segundos. Esto lo hace 2 veces.
- A continuación, y aún dentro del bucle infinito, Mr. Wiggly se mueve −25 pasos (hacia la izquierda) y espera 1 segundos. Esto lo hace 4 veces.
- Y para volver a su posición inicial, Mr. Wiggly se mueve 25 pasos (hacia la derecha) y espera 1 segundos. Esto lo hace 2 veces. Al final, el personaje se habrá movido un total de 8 veces: 2 hacia la derecha, 4 hacia la izquierda, y otras 2 hacia la derecha.
- Con esto hemos terminado con el "baile". Además de bailar, Mr. Wiggly cambiará de color a cada pasada del bucle. Dentro del bucle infinito, y a continuación, añade el bloque "cambia efecto (color) por (25)" de la categoría "apariencia"
- Concluye el bucle infinito añadiendo el bloque "pensar (Hmm... ¡Una vez más!) por (1) segundos".

7) Para parar el programa, pincha en el botón del punto rojo junto al botón de la bandera.
8) Guarda el proyecto en tu carpeta de trabajo con el nombre **guiada2.sb2**.

1.4. LOS BLOQUES DE SCRATCH.

Como vemos en la figura, Scratch incluye 4 tipos de bloques: **bloques de comando** (command blocks), **bloques de función** (function blocks), **bloques de activación** (trigger blocks), y **bloques de control** (control blocks). Los bloques de comando y los bloques de control (también llamados bloques apilables) tienen protuberancias en su parte inferior y/o hendiduras en su parte superior. Estos bloques pueden interconectarse para formar pilas de bloques. Los bloques de activación (también llamados sombreros) tienen su parte superior redondeada porque siempre se ubican en la parte alta de una pila. Los bloques de activación asocian eventos a los programas. Estos bloques esperan a que ocurra un evento, como presionar un botón o una tecla, y ejecutan los bloques que hay bajo ellos cuando ese evento se produce. Por ejemplo, todos los programas que comienzan con el bloque "al presionar la bandera verde" se ejecutarán cuando el usuario clique en el botón de la bandera.



Los bloques de función no tienen protuberancias ni hendiduras, y no pueden usarse como elementos constituyentes de una pila de bloques. Son bloques que siempre devuelven algún valor (un número, un texto,

etc.), y se usan como entradas para otros bloques. La forma de estos bloques indica el tipo de dato que devuelven. (Los bloques de función con bordes redondeados devuelven números o cadenas de texto, y los hexagonales indican si algo es verdadero o falso).

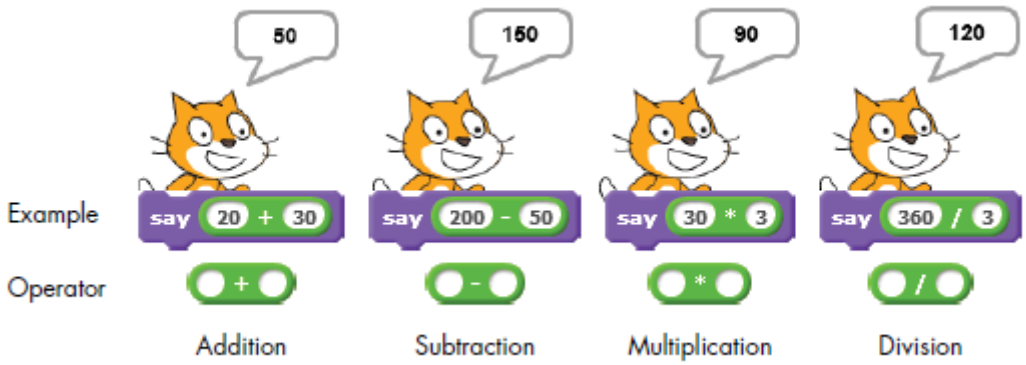
Algunos bloques de función tienen una casilla a su lado. Si activamos la casilla, en el escenario aparecerá un monitor que muestra el valor actual del bloque. Por ejemplo, selecciona un objeto y activa la casilla junto al bloque "posición en x" de la categoría "movimiento". Si ahora arrastras ese objeto por el escenario, verán que el monitor muestra la coordenada x del objeto.

1.5. OPERADORES ARITMÉTICOS Y FUNCIONES.

En esta última sección del capítulo, vamos a echar un vistazo rápido a los operadores y funciones aritméticas soportadas por Scratch, todos ellos disponibles en la categoría "operadores".

OPERADORES ARITMÉTICOS.

Scratch proporciona las cuatro operaciones aritméticas básicas de suma $(+)$, resta $(-)$, multiplicación $(*)$, y división $(/)$. La figura muestra los bloques usados para realizar estas operaciones:



Scratch también soporta el operador "( ) mod ( )", que devuelve el resto de una división de dos números. (Por ejemplo, $10\ mod\ 3$ devuelve 1, porque el resto de dividir 10 entre 3 es 1). El módulo es útil, por ejemplo, para comprobar la divisibilidad de un entero entre otro entero más pequeño. Un módulo de 0 indica que el entero mayor es divisible entre el entero más pequeño. En base a esto, ¿se te ocurre cómo comprobar si un número es par o impar?

Otro operador útil es el operador "redondear ( )", que redondea números decimales al entero más próximo usando las reglas del redondeo. Por ejemplo, redondear$(3{,}1) = 3$, redondear$(3{,}5) = 4$, y redondear$(3{,}7) = 4$.

NÚMEROS ALEATORIOS.

Cuando adquiramos más práctica en programación, probablemente necesitaremos generar números aleatorios, especialmente si programamos juegos y simulaciones. Con este objetivo en mente, Scratch proporciona el bloque "número al azar entre ( ) y ( )".

El bloque "número al azar" proporciona un número aleatorio cada vez que lo usamos. Contiene dos huecos en blanco que nos permiten introducir un rango para el número aleatorio que queremos obtener, y Scratch solo elegirá valores aleatorios entre esos dos límites (ambos incluidos). La tabla a continuación muestra algunos ejemplos de uso del bloque "número al azar".

NOTA: Observa las salidas de "número al azar entre (0) y (1)" y "número al azar entre (0) y (1,0)". El primer caso nos dará o un 0 o un 1, pero el segundo nos dará un número decimal entre 0 y 1. Si cualquier

datos de entrada del bloque "número al azar entre ( ) y ( )" contiene un decimal, la salida también será un decimal.

Example	Possible Outcome
número al azar entre 0 y 1	{0, 1}
número al azar entre 0 y 10	{0, 1, 2, 3, ... , 10}
número al azar entre -2 y 2	{–2, –1, 0, 1, 2}
10 * número al azar entre 0 y 10	{0, 10, 20, 30, ... , 100}
número al azar entre 0 y 1.0	{0, 0.1, 0.15, 0.267, 0.3894, ... , 1.0}
número al azar entre 0 y 100 / 100	{0, 0.01, 0.12, 0.34, 0.58, ... , 1.0}

FUNCIONES MATEMÁTICAS.

Scratch también soporta un gran número de funciones matemáticas. El bloque "(raíz cuadrada) de ( )" agrupa un total de 14 funciones matemáticas, que pueden seleccionarse de la lista desplegable que se abre al clicar en (raíz cuadrada), incluyendo la propia raíz cuadrada, el valor absoluto, las funciones trigonométricas seno, coseno, y tangente (con sus inversas), logaritmos, exponenciales, etc.

EJERCICIO 0. OPERACIONES MATEMÁTICAS.

a) Usa los bloques matemáticos apropiados y el bloque "decir ( )" para calcular y mostrar por pantalla:

1. La raíz cuadrada de 32.
2. El resultado de redondear $99{,}459$.
3. El valor absoluto de 14 y de $-32{,}2$.
4. El valor de 10^3.
5. El logaritmo de 1000.

b) Ídem:

Expression	Value
3 + (2 × 5)	
(10 / 2) – 3	
7 + (8 × 2) – 4	
(2 + 3) × 4	
5 + (2 × (7 – 4))	
(11 – 5) × (2 + 1) / 2	
5 × (5 + 4) – 2 × (1 + 3)	
(6 + 12) mod 4	
3 × (13 mod 3)	
5 + (17 mod 5) – 3	

c) Usa el bloque "decir ( )" y los bloques apropiados para calcular la media de los números 90, 95, y 98.
d) Usa el bloque "decir ( )" y los bloques apropiados para convertir $60°F$ a Celsius ($°C$). (Pista: $T(°\text{C}) = (5/9) \times (T(°F) - 32)$).

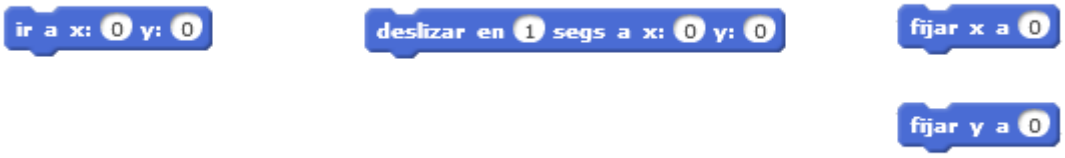
2. MOVIMIENTO Y DIBUJO.

2.1. BLOQUES DE MOVIMIENTO.

Para mover a los objetos sobre el escenario, Scratch ofrece un amplio abanico de instrucciones. Todas ellas están disponibles en los bloques de la categoría "movimiento".

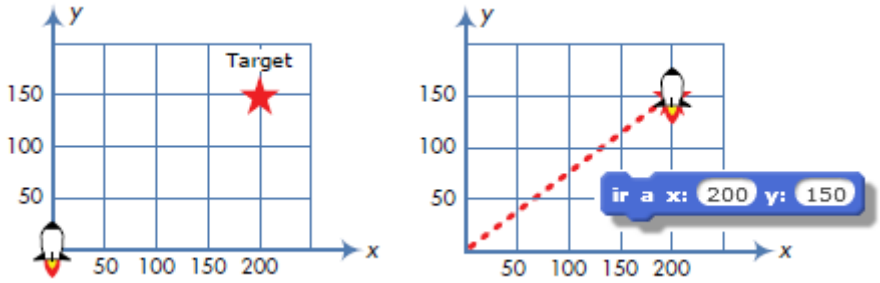
MOVIMIENTO ABSOLUTO.

Los cuatro bloques para controlar el **movimiento absoluto** son:



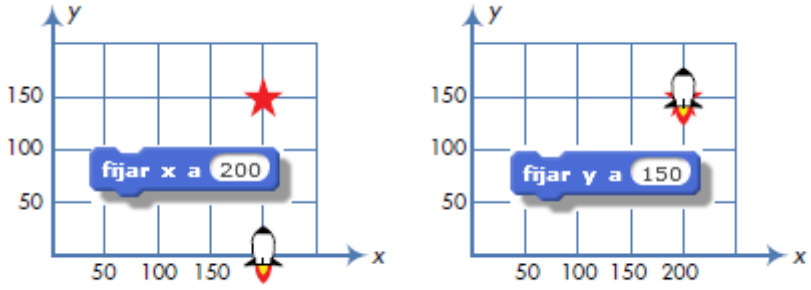
Por ejemplo, para hacer que el cohete vaya a la posición $(x, y) = (200, 150)$, tenemos varias opciones:

a) En un solo paso, con el bloque "ir a x: ( ) y: ( )":



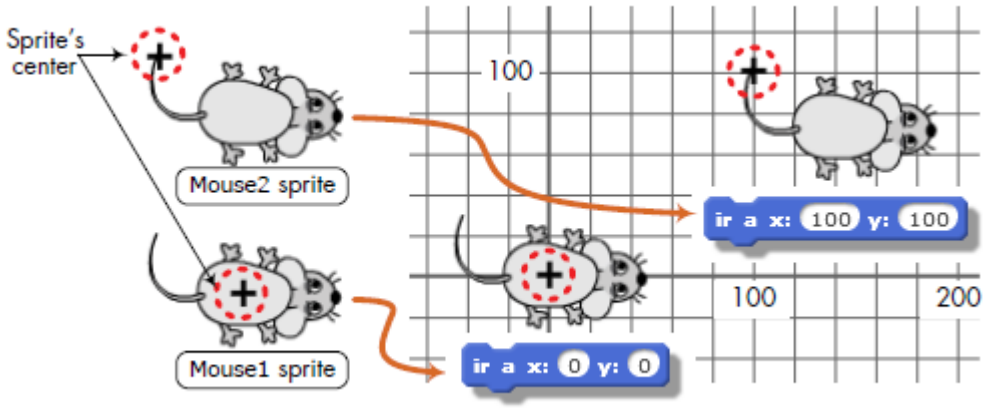
Con esta instrucción, el cohete no se girará para mirar hacia el destino; solo se moverá a lo largo de una línea invisible que conecte el punto inicial $(0, 0)$ con el punto final $(200, 150)$. Notar que, con el bloque "ir a x: ( ) y: ( )", el objeto salta directamente al destino. Si queremos que el objeto realice el viaje de forma progresiva, controlando su velocidad, debemos usar el bloque "deslizar en ( ) segs a x: ( ) y: ( )".

b) En dos pasos, con los bloques "fija x a ( )" y "fija y a ( )":



NOTA: Las instrucciones de movimiento actúan referidas al centro del objeto, donde quiera que esté situado. Por ejemplo, enviar a un objeto al punto $(x, y) = (100, 100)$ mueve al objeto de forma que su centro se ubique en las coordenadas $(100, 100)$, tal y como ilustra la figura. Por lo tanto, cuando

dibujemos o importemos un disfraz para un objeto, y luego tengamos la intención de moverlo, debemos prestar atención a la localización de su centro.

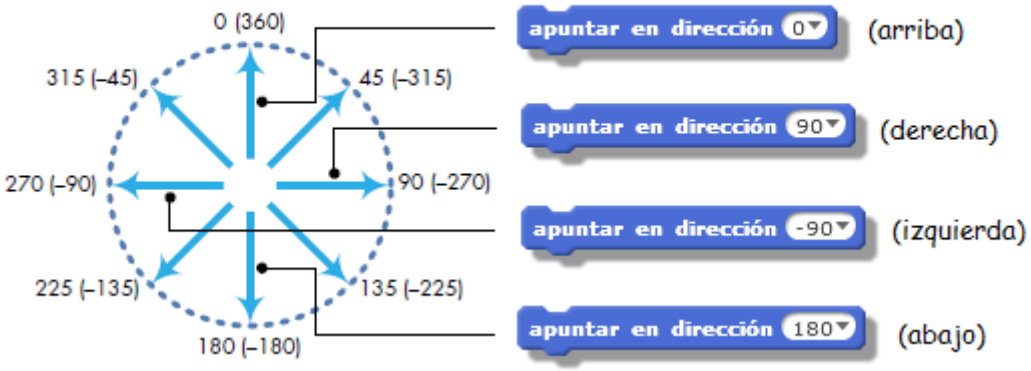


MOVIMIENTO RELATIVO.

Si no conocemos las coordenadas actuales de un objeto, y queremos desplazarlo una cierta distancia respecto su localización actual, lo más conveniente es usar las instrucciones de **movimiento relativo**:

Por ejemplo, para llevar al cohete desde una posición inicial con coordenadas desconocidas, hasta una posición final con coordenadas también desconocidas, deberíamos decirle lo siguiente: muévete 3 pasos, gira 90 grados hacia la derecha, y muévete otros 2 pasos (ver figura).

Las órdenes "mover" y "girar" son ejemplos de comandos de movimiento relativo. La primera orden "mover" hace que el cohete se mueva tres pasos hacia arriba desde la posición en la que estaba. La orden "girar" lo rota 90° desde la dirección en la que estaba mirando (hacia arriba) y lo deja apuntando hacia la derecha. En ambos casos, los movimientos dependen de (son relativos a) la posición y dirección previas. La convención de direcciones usadas en Scratch es la siguiente:

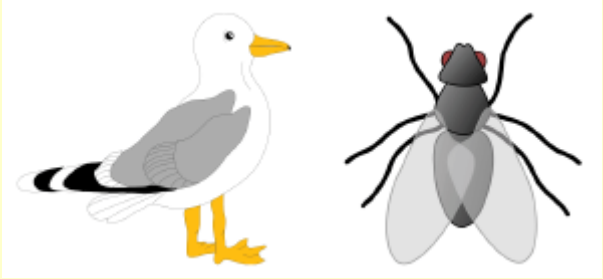


Para hacer que un objeto apunte en una dirección particular, usamos el comando "apuntar en dirección ( )". Para elegir arriba, abajo, derecha, o izquierda, bastan con clicar en la flechita dentro del área blanca del bloque, y seleccionar una de las opciones de la ventana desplegable. Para hacer que el objeto apunte en otras direcciones, escribimos el valor del ángulo deseado en el área de edición blanca del bloque. Incluso podemos usar ángulos negativos (por ejemplo, escribir -45 o 315 hará que el objeto mire hacia arriba y hacia la izquierda en ambos casos).

NOTA: Recuerda que podemos conocer la dirección actual en la que mira el objeto en el área de información del objeto. También podemos activar la casilla junto al bloque "dirección" (en la categoría "movimiento") para ver por pantalla la dirección actual del objeto.

Dirección y disfraces.

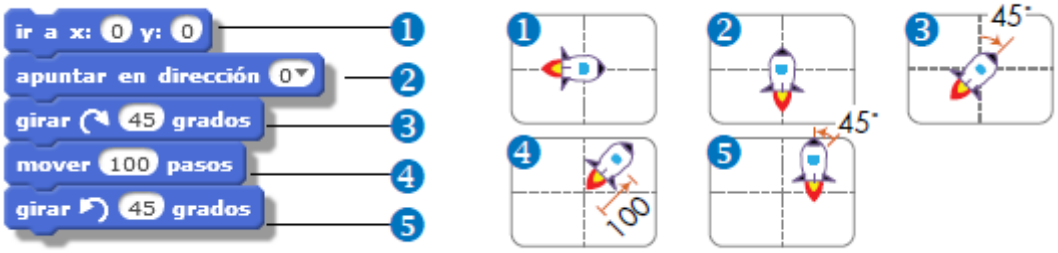
El bloque "apuntar en dirección ( )" no sabe nada acerca del disfraz del objeto. Por ejemplo, consideremos los dos objetos de la figura:



El disfraz del objeto pájaro se ha construido para que mire a la derecha, y el del insecto para que mire hacia arriba. ¿Qué crees que pasará si usamos el comando "apuntar en dirección (90)" (esto es, apuntar hacia la derecha) en ambos objetos?

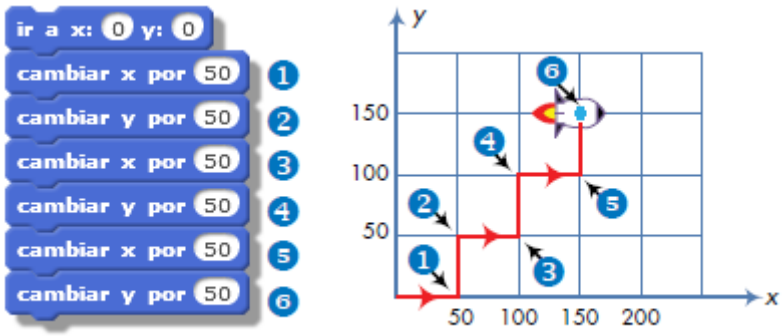
Puede que pensemos que el insecto girará para mirar hacia la derecha, pero de hecho, ninguno de los dos se girará. Aunque $90°$ se etiqueta como "derecha", esa dirección realmente se refiere a la *orientación original* del disfraz en el editor gráfico. Así pues, como el insecto aparece mirando hacia arriba en el editor gráfico, seguirá mirando hacia arriba cuando le digamos que apunte en la dirección $90°$. Si queremos que el objeto responda al comando "apuntar en dirección ( )" como en la figura de la convención de direcciones adoptadas en Scratch, debemos dibujar el disfraz del objeto para que mire hacia la derecha en el editor gráfico (como el disfraz del pájaro en la figura previa).

Ahora que ya sabemos cómo funcionan las direcciones en Scratch, vamos a ver cómo se usan los bloques de movimiento relativo "mover ( ) pasos", "girar ↺ ( ) grados", y "girar ↻ ( ) grados". Por ejemplo, el siguiente programa hace que (1) el centro del cohete se mueva a la posición $(0,0)$ (el centro del escenario). (2) A continuación, hacemos que el cohete apunte en la dirección 0 grados (hacia arriba). Después, en (3) lo giramos 45 grados a favor de las agujas del reloj, y en (4) lo hacemos moverse 100 pasos en su dirección actual. Por último, en (5) giramos el cohete 45 grados hacia la izquierda para dejarlo mirando hacia arriba.



A veces, solo querremos mover el objeto horizontalmente y/o verticalmente desde su posición actual. Para ello usamos los bloques "cambiar x por ( )" y "cambiar y por ( )". Observa el siguiente programa a modo de

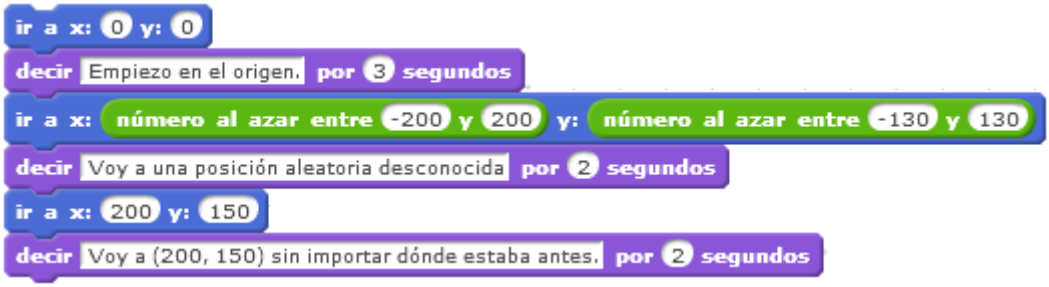
ejemplo. Después de que el objeto se mueva al centro del escenario, posición $(0,0)$, el comando "cambiar x por (50)" en el paso (1) suma 50 unidades a su coordenada x para enviarlo 50 pasos hacia la derecha. Después, en (2), el comando "cambiar y por (50)" suma 50 otras 50 unidades a la coordenada y de su posición actual, haciendo que el objeto se mueva 50 pasos hacia arriba. Los otros comandos funcionan de manera similar. Incrementando en pasos de 50 sus coordenadas x e y, el objeto termina en la posición $(150,150)$.



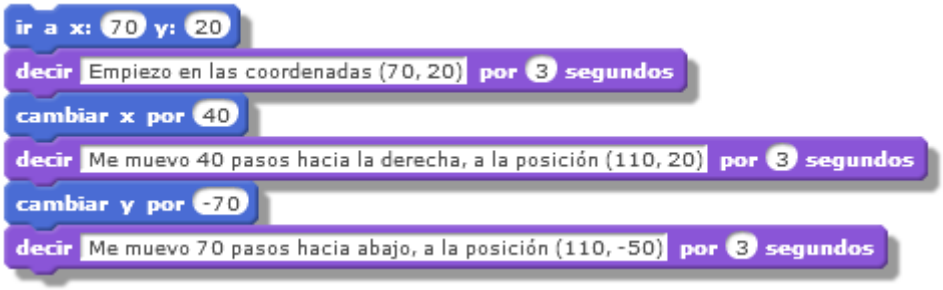
MOVIMIENTO ABSOLUTO VS. MOVIMIENTO RELATIVO.

Aunque todas las instrucciones de la categoría "movimiento" sirven para mover (o girar) un objeto, es importante entender las diferencias entre los bloques de movimiento absoluto y los de movimiento relativo. Los comandos de *movimiento absoluto* nos permiten mover objetos a posiciones específicas sobre el escenario. Por el contrario, los comandos de *movimiento relativo* sirven para mover objetos en relación a su posición y dirección actuales.

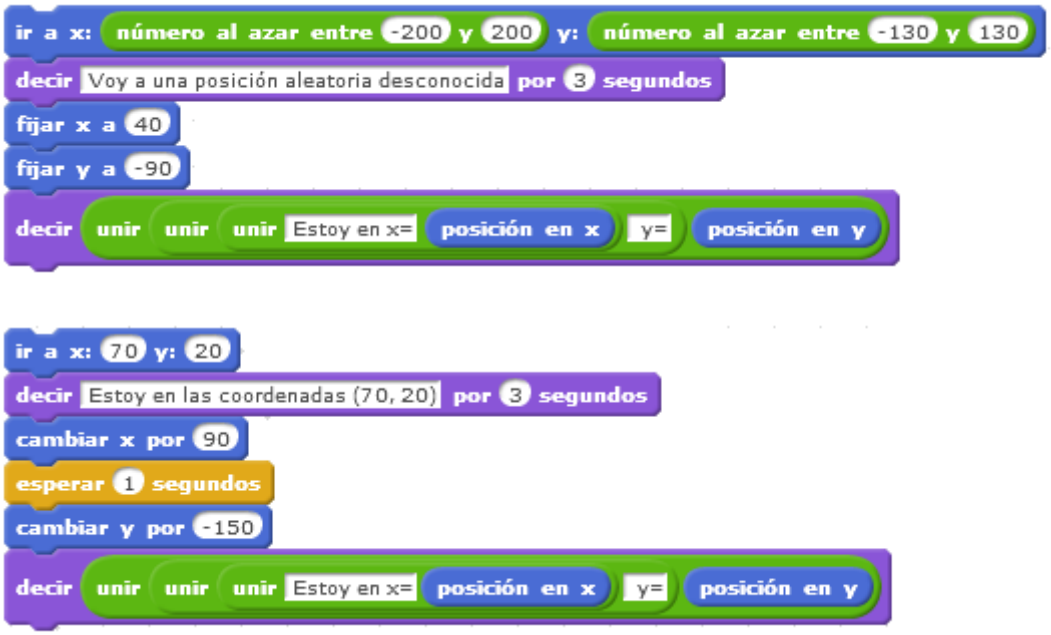
A modo de ejemplo, imaginemos que un objeto se localiza en una posición aleatoria desconocida en el escenario. Al ejecutar el bloque "mover a x: (200) y: (150)", el objeto se traslada a la posición $(200,150)$, independientemente de su posición previa. Prueba el siguiente programa para verificar este comportamiento:



Imaginemos ahora que un objeto se localiza en la posición $(x,y)=(70,20)$. El bloque "cambiar x por (40)" varía el valor de su coordenada x en 40 unidades, trasladando al objeto 40 pasos hacia la derecha hasta la posición $(x,y)=(110,20)$. Si a continuación ejecutamos un bloque "cambiar y por (-70)", la coordenada y de su posición actual cambia en -70 unidades, lo que hace que el objeto se traslade 70 pasos hacia abajo hasta la posición $(x,y)=(110,-50)$. Ejecuta el programa de la figura para verificar este comportamiento (para ello, observa los valores de sus coordenadas (x,y) en el área de información del objeto).

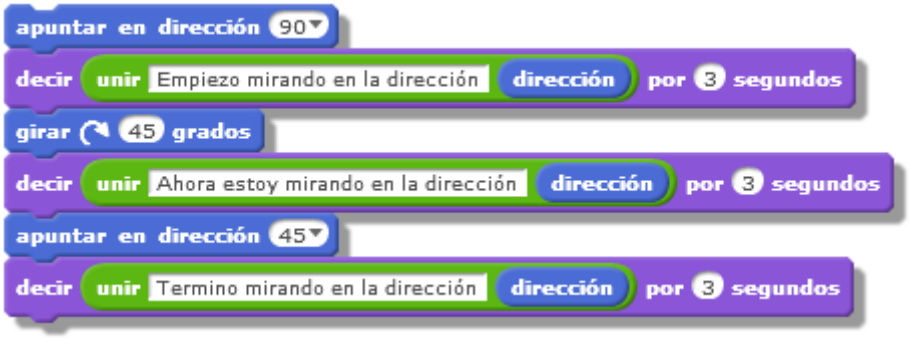


Ahora, vamos a comparar algunos bloques de movimiento que parecen similares, pero cuyo funcionamiento es bien distinto. Comencemos comparando los bloques "fijar x a ( )" y "cambiar x por ( )" (y los bloques correspondientes para la coordenada y) mediante estos dos programas:

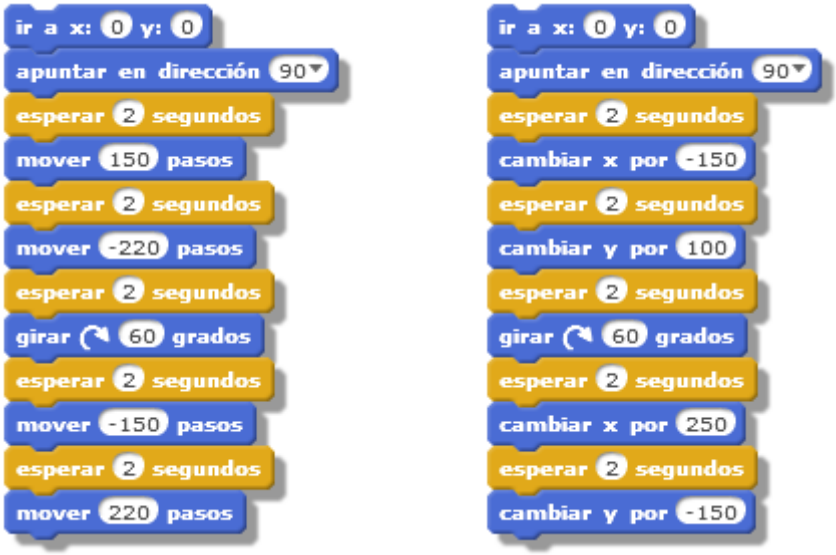


El primer programa utiliza los bloques "fijar a", que fijan los valores de las coordenadas x e y del objeto a unos valores específicos. En este programa, el objeto termina en la posición $(x, y) = (40, -90)$ *siempre*, e independientemente de su posición inicial desconocida. Por el contrario, el segundo programa utiliza los bloques "cambiar por", que toman los valores actuales de las coordenadas x e y del objeto y los cambian en unos valores determinados. En este caso, el objeto termina en una posición que está 90 pasos hacia la derecha y 150 pasos hacia *abajo* desde su localización inicial. Como la posición inicial era $(70, 20)$, el objeto termina en la posición $(160, -130)$.

Continuemos comparando las dos formas de cambiar la dirección de un objeto. Habitualmente, los objetos de Scratch comienzan mirando hacia la derecha (dirección $90°$). Supón que estamos programando uno de estos objetos (por ejemplo, el gato de Scratch). Comenzamos ejecutando el comando "girar ↻ (45) grados". El bloque "girar ↻ ( ) grados" gira el objeto un cierto número de grados en relación a su dirección actual. Como el objeto estaba mirando hacia la derecha, girarlo $45°$ a favor de las agujas del reloj hace que pase a apuntar en la dirección $90° + 45° = 135°$, esto es, hacia abajo y a la derecha. A continuación, ejecutamos el bloque "apuntar en dirección (45)". El objeto queda mirando hacia arriba y hacia la derecha, en la dirección $45°$. Esto es así porque el bloque "apuntar en dirección ( )" fija una cierta dirección para el objeto, independientemente de la dirección en la que estuviese mirando antes. (Por lo tanto, el bloque "apuntar en dirección ( )" es realmente una instrucción de movimiento absoluto, aunque lo hayamos presentado en la sección de movimiento relativo).



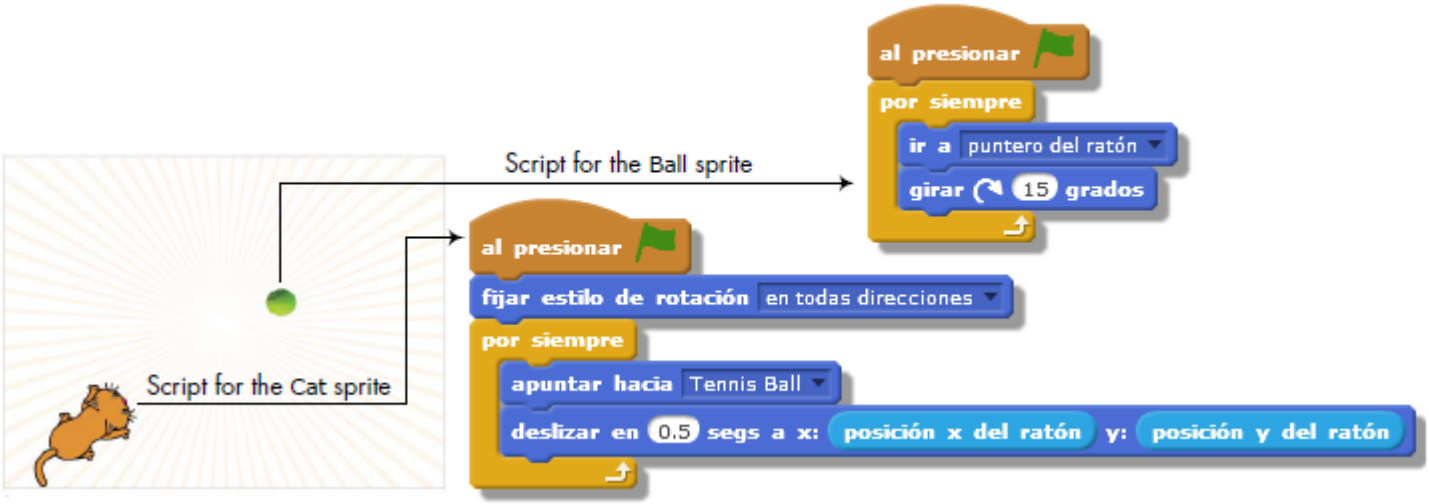
Por último, vamos a comparar el bloque "mover ( ) pasos" con los bloques "cambiar x por ( )" y "cambiar y por ( )". El primer programa de la figura utiliza el bloque "mover", que desplaza al objeto un cierto número de pasos en la dirección en la que está apuntando (o en la dirección opuesta, si el número de pasos es negativo). Por el contrario, el segundo programa utiliza los bloques "cambiar por" para modificar los valores actuales de las coordenadas x e y en las cantidades indicadas. Observar que, en este segundo caso, los bloques "cambiar x por ( )" y "cambiar y por ( )" siempre desplazan al objeto horizontalmente y verticalmente, independientemente de la dirección en la que esté mirando.



OTROS BLOQUES DE MOVIMIENTO.

Estos son los últimos bloques de movimiento que nos queda por explicar:

El bloque "fijar estilo de rotación ( )" sirve para indicar lo que hace el disfraz del objeto al cambiar de dirección. El bloque "rebotar si toca un borde" hace que el objeto rebote cuando llega al borde del escenario. Ya vimos estos bloques en funcionamiento en la sección "información del objeto" (sección 1.2).



Para ver en acción los otros dos comandos, probemos la siguiente aplicación: Tenemos dos objetos, el gato (objeto "cat2" de la biblioteca de Scratch) y la pelota (tennis ball). También tenemos un fondo (rays).

Cuando clicamos en el icono de la bandera, el objeto pelota siempre apunta en la dirección del puntero del ratón. El objeto gato siempre apunta hacia la pelota, y se dirige hacia las coordenadas x e y del ratón mediante el bloque "deslizar". Los bloques de función "posición x del ratón" y "posición y del ratón" se encuentran en la categoría "sensores".

EJERCICIO 1: INICIALES.

Haz que la inicial de tu nombre cambie de color y gire ininterrumpidamente al hacer clic sobre ella. Guarda el proyecto en tu carpeta de trabajo con el nombre **ejercicio1.sb2**. Utiliza un escenario y un objeto similares a los mostrados en la figura (por supuesto, con la letra inicial de tu nombre).

AMPLIACIÓN: Ahora haz que la inicial de tu nombre cambie de tamaño, de manera que crezca y mengüe constantemente. PISTA: Necesitarás un bucle infinito, dentro del cual hay dos bucles "repetir ( )". Usa el bloque "cambiar tamaño por ( )" de la categoría "apariencia" para modificar el tamaño del objeto. (Un número positivo dentro de este bloque hace que el objeto crezca, y un número negativo hace que el objeto se encoja). Guarda el proyecto en tu carpeta de trabajo con el nombre **ejercicio1(2).sb2**.

EJERCICIO 3: ACUARIO VIRTUAL.

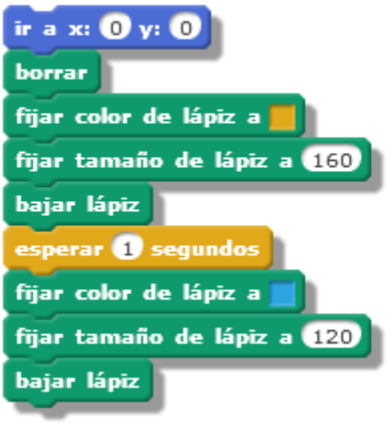
Vamos a crear un proyecto en el que una serie de animales acuáticos (peces, pulpos, estrellas de mar, etc.) nadan y/o se mueven en un acuario. Para ello, sigue los siguientes pasos:

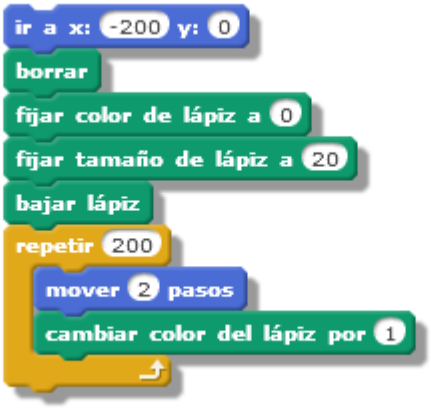
1) Añade los personajes necesarios (al menos cinco) y un fondo apropiado.
2) Añade al escenario un archivo de audio apropiado (agua, burbujas, etc.).
3) Anima a los personajes: unos irán más rápidos, otros más lentos; unos se moverán en una dirección, otros en otra; algunos sólo andan por el suelo, otros nadan por el agua; puedes hacerlos moverse, girar y moverse, sólo girar; etc. (Todo depende del tipo de animal que estés animando). Por supuesto, todos ellos deben responder adecuadamente al llegar al borde del acuario.
4) Programa al escenario para que siempre esté reproduciendo el sonido acuático que has seleccionado.
5) Al terminar, guarda el proyecto en tu carpeta de trabajo con el nombre **ejercicio3.sb2**.

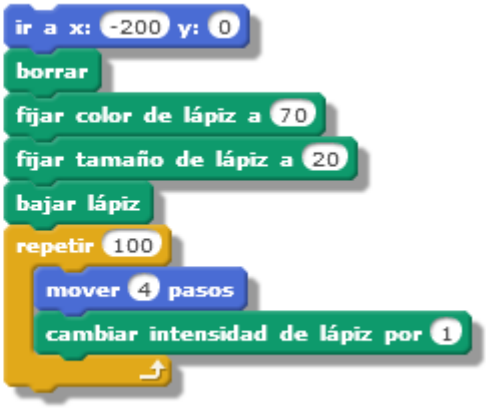
2.2. BLOQUES DE DIBUJO.

Después de aprender las instrucciones de movimiento, sería interesante ver el camino que sigue el objeto al desplazarse. Para ello, disponemos de las instrucciones de dibujo en la categoría "lápiz".

Todos los objetos disponen de un lápiz invisible, que pueden bajar y subir a voluntad. Si el lápiz está abajo, el objeto va dejando un trazo conforme se mueve. Si el lápiz está arriba, el objeto se mueve sin dejar rastro alguno. El resto de comandos de la categoría "lápiz" permiten ajustar el tamaño, el color, y otras características del lápiz. Para ver los comandos de dibujo en acción, prueba los siguientes programas:







NOTA: Al clicar con el ratón sobre el cuadrado coloreado del bloque "fijar color del lápiz a □", el puntero del ratón se convierte en una mano. Si ahora movemos la mano por la pantalla, el color del cuadrado va cambiando al color de la zona por la estamos pasando. Al clicar de nuevo, el cuadrado fija su color al color seleccionado en la pantalla.

NOTA: En el bloque "fijar tamaño del lápiz a ( )", el tamaño máximo del lápiz es 200. No intentes fijar un tamaño superior; comprobarás que el lápiz no se hace más grande.

En el primer programa, cambia el orden en el que dibujas los puntos naranja (tamaño 160) y azul (tamaño 120), esto es, pinta primero el azul, y luego el naranja. ¿Qué ocurre? ¿Qué orden debes seguir al dibujar puntos concéntricos de distinto tamaño?

Observa que la salida del segundo programa te indica qué números se corresponden con qué colores en el bloque "fijar color de lápiz a ( )". Fíjate que el 0 se corresponde con el rojo, el 25 con el naranja, el 35 con el amarillo, el 45 con el verde, el 100 con el azul claro, el 130 con el azul oscuro, y el 160 con el violeta. Este bloque solo acepta valores entre 0 y 200, con los que puede cubrir todos los colores del espectro visible.

Para terminar, comprueba qué pasa en el tercer programa si, dentro del bucle, sustituimos el bloque "cambiar intensidad de lápiz por ( )" por el bloque "cambiar tamaño de lápiz por ( )".

EJERCICIO 4: ARCOÍRIS CIRCULAR.

Escribe un programa que reproduzca esta salida por pantalla. Guarda el proyecto como **Ejercicio4.sb2** en tu carpeta de trabajo.

EJERCICIO 4(B). MOVIMIENTO Y DIBUJO.

Utiliza los bloques de las categorías "lápiz" y "movimiento" para que un objeto dibujante (por ejemplo, arrow1) conecte en el orden indicado las siguientes series de puntos, y revele el dibujo resultante. Guarda el proyecto como **Ejercicio4b.sb2** en tu carpeta de trabajo.

a. (30,20), (80,20), (80,30), (90,30), (90,80), (80,80), (80,90), (30,90), (30,80), (20,80), (20,30), (30,30), (30,20)

b. (−10,10), (−30,10), (−30,70), (−70,70), (−70,30), (−60,30), (−60,60), (−40,60), (−40,10), (−90,10), (−90,90), (−10,90), (−10,10)

ACTIVIDAD GUIADA 3: LA CUCARACHA PINTORA.

Vamos a crear un programa que dibuje líneas en pantalla mediante un objeto que se mueva y gire en el escenario con las teclas ← → ↑ ↓ . La tecla (↑) hace que el objeto se mueva hacia adelante 10 pasos. La tecla (↓) hace que el objeto se mueva hacia atrás 10 pasos (recuerda que retroceder 10 pasos es lo mismo que avanzar -10 pasos). Cada vez que presionemos la tecla (→) el objeto girará hacia la derecha $10°$, y cada vez que presionemos (←) el objeto girará hacia la izquierda $10°$.

En primer lugar, arranca un nuevo proyecto de Scratch, y reemplaza el objeto gato por otro objeto que muestre claramente si está apuntando hacia la izquierda, derecha, arriba, o abajo. Los objetos "beetle" o "cat2" (de la carpeta "animales") son buenas opciones, pero elige el que quieras.

A continuación, vamos a crear el código del objeto. Necesitarás 5 programas en total. El primer programa comienza al presionar el botón de la bandera, y sirve para configurar la posición inicial del objeto, $(x, y) = (0, 0)$, hacerlo apuntar hacia arriba, determinar el color del lápiz (elige uno cualquiera) y el tamaño del lápiz (3), limpiar el escenario, y bajar el lápiz. Los otros cuatro programas sirven para decirle al objeto qué hacer al presionar las teclas de las flechas. Para ello, necesitaremos el bloque "al presionar tecla ( )" de la categoría de "eventos". Por ejemplo:

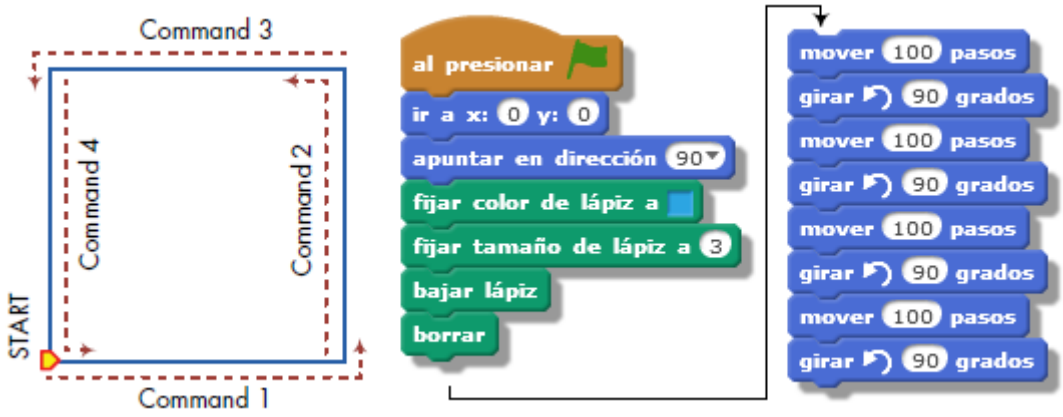
Prueba tu aplicación para asegurarte de que funciona, y guarda el proyecto como **guiada3.sb2**. A continuación, modifica la aplicación para que el trazo se haga más ancho al presionar la tecla A y más estrecho al presionar la tecla E. Piensa en otras formas de modificar la aplicación, como cambiar el trazo de color, de intensidad, etc.

2.3. BUCLES: EL PODER DE LA REPETICIÓN.

En los programas, a menudo es necesario repetir el mismo conjunto de instrucciones varias veces. Pero replicar el código a repetir hace que los programas sean largos, lentos, y difíciles de entender. Para evitar este problema, existen los **bucles**:

Ya nos hemos encontrado previamente con el bucle "por siempre". Este bucle permite repetir indefinidamente las instrucciones que pongamos dentro de él. Por el contrario, el bucle "repetir ( )" permite repetir la ejecución de un cierto conjunto de instrucciones un cierto número (limitado) de veces.

Por ejemplo, si queremos dibujar el cuadrado mostrado en la figura, podríamos hacerlo ejecutando el siguiente código repetitivo:

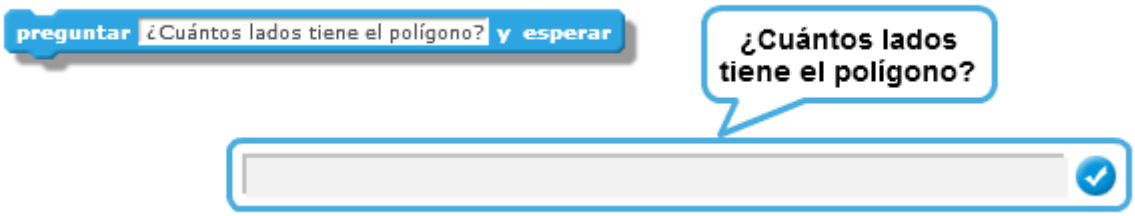


Sin embargo, la segunda parte del programa (a la derecha) puede simplificarse enormemente usando un bucle "repetir":

ACTIVIDAD GUIADA 4: POLÍGONOS.

El programa para dibujar un cuadrado puede modificarse fácilmente para poder dibujar otros polígonos regulares. Para ello, debemos hacer lo siguiente:

1) Como antes, reemplaza el objeto gato por otro objeto donde sea fácil ver hacia dónde está apuntando (como los objetos "airplane", "arrow1", "cat2", etc.), o crea tu propio objeto.
2) Sitúa al objeto en la posición $(x, y) = (0, -150)$.
3) Ajusta el color del lápiz, el tamaño del lápiz, baja el lápiz, y deja la pantalla limpia.
4) Define una variable llamada "numLados" (número de lados) y otra variable llamada "longLado" (longitud del lado). PISTA: Para crear variables, acude a la categoría "datos", y selecciona la opción "Crear una variable"; en la ventana de diálogo, escribe el nombre de la variable, y selecciona "para todos los objetos". Al crear las variables, verás que en la categoría "datos" aparecen nuevos bloques (el bloque de función que almacena el valor de la variable, los bloques de comando para fijar y cambiar el valor de la variable, etc.).
5) En la categoría "sensores", usa el bloque "preguntar (¿Cuántos lados tiene el polígono?) y esperar". Este bloque le pide al usuario que especifique el número de lados del polígono a dibujar, y queda a la espera hasta que el usuario introduce ese dato. Cuando el usuario responde la espera finaliza, el programa continúa con su ejecución, y la respuesta del usuario queda almacenada en el bloque "respuesta".



6) Ahora, y con el bloque "fijar ( ) a ( )" de la categoría "datos", vamos a configurar el valor de la variable "numLados" usando el valor proporcionado por el usuario, que está almacenado en el bloque "respuesta".

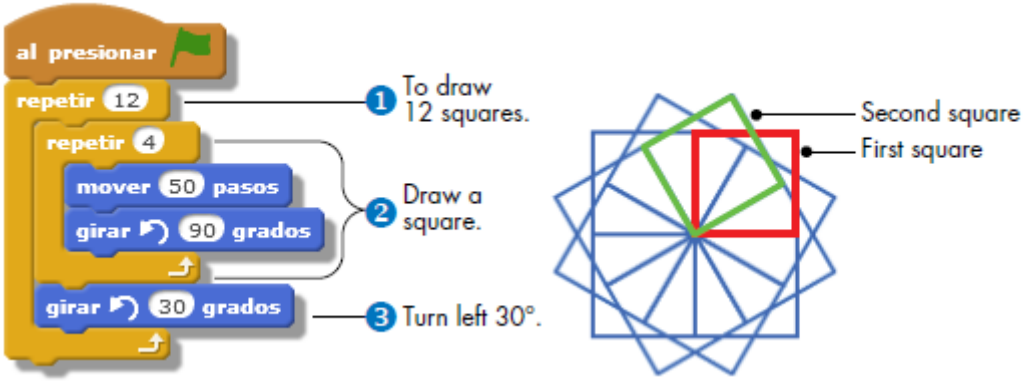
7) Replica los pasos 5 y 6 para pedirle al usuario la longitud de cada lado, y almacenar su respuesta en la variable "longLado".
8) Utiliza un bucle "repetir ( )" para dibujar el polígono. El bucle debe repetirse tantas veces como lados tenga el polígono (este valor está almacenado en la variable "numLados", la cual podemos usar como dato de entrada del bucle "repetir ( )").

A cada pasada, el objeto se mueve una distancia igual al valor de "longLado", y gira hacia la izquierda un número de grados igual a $360/"numLados"$.

9) Guarda el proyecto como **guiada4.sb2**.
10) ¿Qué ocurre cuando el número de lados se hace muy grande? ¿Qué debes hacer con la longitud del lado? ¿Te da esto una idea del procedimiento para dibujar círculos?

ACTIVIDAD GUIADA 5: CUADRADOS ROTADOS.

Utilizando los bucles podemos crear bonitos efectos artísticos simplemente repitiendo un patrón con una secuencia dada. Observa el siguiente programa. (A este programa le faltan las instrucciones para ubicar al objeto en el centro, configurar el color y tamaño del lápiz, bajar el lápiz, y limpiar la pantalla. Añádelas tú). El primer bucle "repetir" se ejecuta 12 veces. A cada pasada del bucle, dibujamos un cuadrado (mediante el segundo bloque "repetir"), y además giramos hacia la izquierda 30 grados para dibujar el siguiente cuadrado. Observa que, para completar toda una vuelta ($360°$), se cumple que $(12\ repeticiones) \times (30°\ por\ repetición) = 360°$. (¿Qué crees que pasaría si cambiásemos a 4 repeticiones y $90°$? ¿Y si fijásemos 5 repeticiones y $72°$? Experimenta con diferentes valores de estos argumentos, y observa qué ocurre).

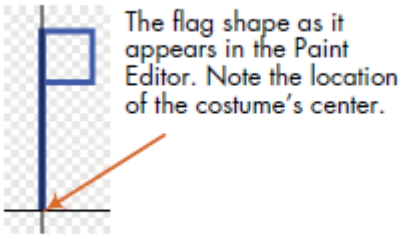


Modifica el programa de forma similar a la actividad guiada previa para permitir al usuario elegir el número de veces que se dibujará el cuadrado (esto es, el número de veces que se repite el primer bucle). Su respuesta determinará el valor de una variable ("numGrados") que indicará cuántos grados hay que girar para dibujar el siguiente cuadrado. Calcula este valor mediante una operación de Scratch. Guarda el proyecto como **guiada5.sb2**.

AMPLIACIÓN: Cambia el programa para permitir al usuario elegir el número y la longitud de los lados del polígono regular que vamos a rotar. Necesitarás crear varias variables para almacenar las sucesivas respuestas del usuario.

SELLOS.

En la actividad previa usaste los bloques "girar" y "repetir" para convertir formas simples (polígonos) en patrones complejos. ¿Y si queremos rotar formas más complejas? Para ello, podemos dibujar un objeto nuevo con el editor de objetos de Scratch, y usar el bloque "sellar" de la categoría "lápiz" para hacer múltiples copias del objeto en el escenario. A modo de ejemplo, abre el editor gráfico de Scratch, y dibuja el objeto bandera mostrado en la figura. (Observa que el centro del disfraz está en el extremo del mástil de la bandera).

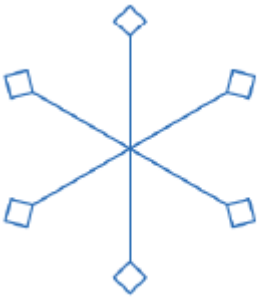


A continuación, ejecuta el programa de la figura para dibujar las aspas de un molino. El bloque "repetir" se ejecuta 8 veces; a cada pasada, el bloque "sellar" estampa una copia del disfraz sobre la pantalla, y a continuación, el programa gira el objeto $45°$ hacia la izquierda. (Para que este programa funcione, debemos usar el bloque "fijar estilo de rotación ( )" con la opción "en todas direcciones", para permitir que la bandera se voltee mientas gira).

Para terminar, agrega el bloque "cambiar efecto ( ) por ( )" de la categoría "apariencia" y experimenta con los distintos efectos gráficos (cambios de color, arremolinamiento, ojo de pez, etc.) para conseguir patrones más impactantes. Por cierto, para que funcione la instrucción "cambiar efecto (color) por ( )", el color de la bandera en el editor gráfico no puede ser negro.

EJERCICIO 5: PRACTICAMOS CON SELLOS.

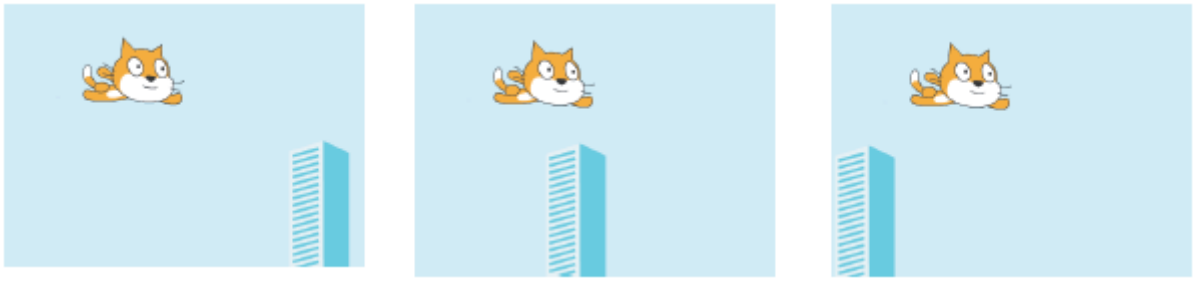
Crea los objetos y programas que te permitan dibujar los dos patrones mostrados en la figura. Piensa bien dónde te conviene ubicar el centro de cada objeto. Reflexiona también sobre cómo desplazar *horizontalmente* las copias del objeto en el segundo ejercicio (concretamente, no te conviene usar el bloque "mover ( ) pasos"). NOTA: Para evitar que ambos dibujos se solapen por pantalla, podemos ocultar los dos objetos, y hacer que sus programas comiencen con eventos distintos (por ejemplo, bandera verde y tecla espacio). Guarda el proyecto como **Ejercicio5.sb2**.



EJERCICIO 6: GATO VOLADOR.

1) Escoge un personaje volador cualquiera (cat 1 flying, bat1, bat2, butterfly1, etc.), un fondo (un cielo azul), y el objeto "edificio" (buildings). Tendrás dos objetos (el personaje y el edificio), y un fondo para el escenario (además del fondo blanco). Elimina todo objeto y fondo que no necesites.

2) Mueve el paisaje para que parezca que el personaje vuela. Para ello, escribe el siguiente programa para el objeto "edificio": Al clicar en la bandera verde, el programa arranca un bucle infinito. Dentro del bucle, el programa ubica al objeto en la zona no visible de la parte derecha de la pantalla, $(x, y) = (250, 0)$. A continuación, y dentro del bucle, un bucle "repetir ( )" cambia gradualmente la posición horizontal del objeto para que se mueva de derecha a izquierda (en pasos de -5). Si el edificio se desplaza en pasos de 5 unidades, el bucle debe repetirse 100 veces. El programa funcionará bien cuando consigas que el edificio aparezca a la derecha, se desplace hacia la izquierda, y al desaparecer en el extremo izquierdo vuelva a aparecer en el extremo derecho. Si es necesario, varía los argumentos del bucle "repetir" y de las instrucciones de movimiento para conseguirlo.

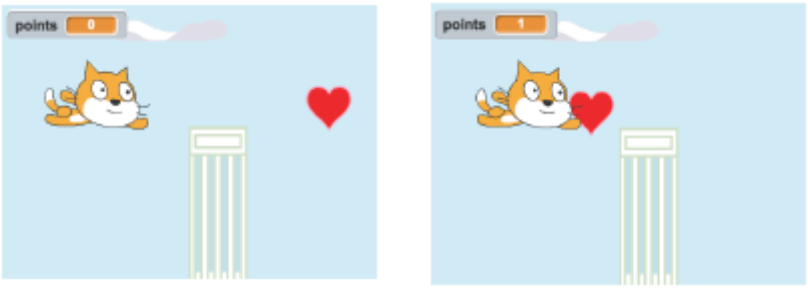


3) Haz el paisaje más variado. Modifica ligeramente el programa para que el objeto "edificio" cambie de disfraz (bloque "siguiente disfraz" de la categoría "apariencia") cada vez que complete su viaje de derecha a izquierda. (El objeto "buildings" tiene 10 disfraces).

4) Haz que tu personaje se mueva al presionar las teclas ↑ ↓ ← →. Por ejemplo:

5) Añade nubes flotando en el cielo: Haz que el objeto "clouds" viaje de derecha a izquierda (esto es, a lo largo de la coordenada x), comenzando en $x = 250$ con una coordenada y aleatoria entre 1 y 180. Cada vez que completa el viaje de ida, las nubes deben cambiar de disfraz (el objeto clouds dispone de 4 disfraces distintos). Como en el paso 3, deberás anidar dos bucles: un bucle "por siempre" y un bucle "repetir". Si haces que las nubes se muevan hacia la izquierda en pasos de -10, el bucle deberá repetirse unas 50 veces.

6) Agrega corazones u otros objetos voladores para recolectarlos. Como en el paso 3, haz que el objeto "heart" *siempre* comience en una posición aleatoria (bloque "ir a (posición aleatoria)" de la categoría "movimiento"), y fija su coordenada $x = 250$. Con esto, harás que el objeto siempre aparezca en el extremo derecho del escenario, en una posición y aleatoria. A continuación, haz que el objeto *siempre* viaje de derecha a izquierda en pasos de -15, con un bucle que se repita 32 veces.



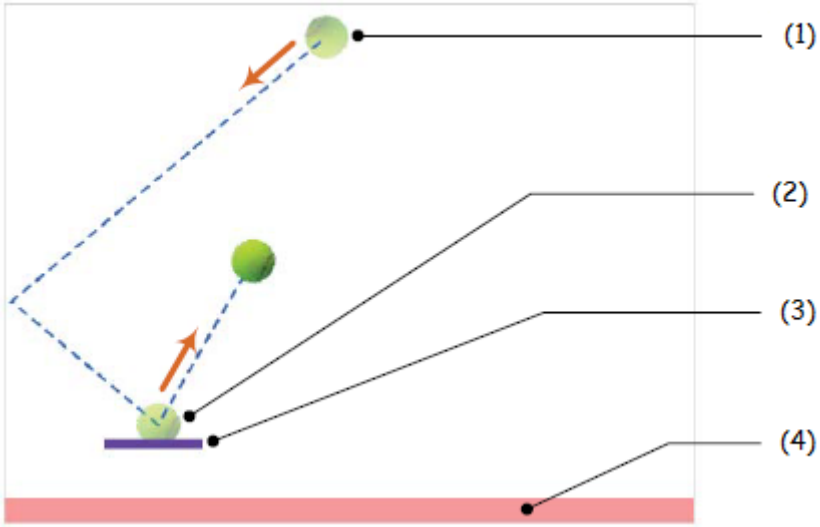
7) Acumula puntos. En la categoría "datos", crea una variable llamada "puntos" que lleve la cuenta de los corazones que vas recolectando. A continuación, crea un nuevo programa para el personaje volador que comience al clicar en la bandera verde. Este programa empieza fijando el valor inicial de la variable "puntos" a 0, bloque "fijar ( ) a ( )" de la categoría "datos". A continuación, arranca un bucle "por siempre", dentro del cual el personaje comprueba si está tocando el objeto corazón. En caso afirmativo, cambia la variable "puntos" en una unidad, y después espera un segundo (para evitar que la puntuación siga aumentando conforme el corazón pasa a través del personaje volador). Para hacer todo esto, necesitarás el bloque "si ( ) entonces" de la categoría "control", el bloque "¿tocando ( )?" de la categoría "sensores", y el bloque "cambiar ( ) por ( )" de la categoría "datos". El esquema básico de esta parte del programa es:

8) Guarda el proyecto en tu carpeta de trabajo con el nombre **ejercicio6.sb2**.

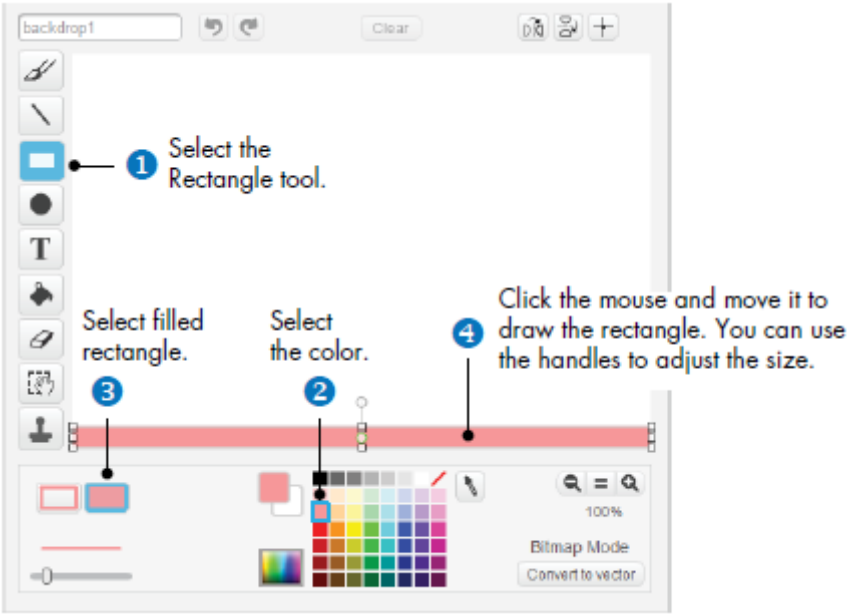
AMPLIACIÓN:
a) Haz que el corazón desaparezca cuando el gato lo coja. Para ello necesitarás los bloques "esconder" y "mostrar" de la categoría "apariencia". Piensa que quieres esconder el corazón *si* el corazón toca al personaje volador, y volver a mostrarlo cuando el corazón vuelve a aparecer a la derecha del escenario.
b) Puedes mejorar el proyecto haciendo que el juego termine si el personaje volador colisiona contra un edificio. Añade el código necesario para lograrlo. (Pista: inspírate en el código del apartado 7, y echa un vistazo a la instrucción "detener ( )" de la categoría de "control").
c) Haz el juego más interesante y difícil añadiendo otros obstáculos voladores (asteroides, pájaros, aviones, etc.) contra los que pueda colisionar nuestro personaje, lo que también implicará el fin de la partida.

EJERCICIO 7. JUEGO DE PELOTA.

El funcionamiento del juego es el siguiente: (1) Cuando el juego comienza, la pelota empieza en la posición indicada en la figura, y se mueve hacia abajo con un ángulo aleatorio. Por supuesto, la pelota rebota al llegar al borde del escenario. (2) Al impactar contra la raqueta, la pelota rebota hacia arriba con un ángulo aleatorio. (3) La raqueta se mueve en horizontal usando el ratón del ordenador. (4) Si la pelota toca el suelo, la partida termina.



Paso 1: Prepara el escenario. Selecciona la miniatura del escenario (con fondo blanco), acude a la pestaña de "fondos", y modifica el fondo blanco con el editor gráfico de Scratch para añadir un delgado y alargado rectángulo de color en la parte baja (ver figura previa). El proceso para dibujar este rectángulo está ilustrado en la siguiente figura. ¿Para qué dibujamos este rectángulo de color? Más adelante, para comprobar si la raqueta falla el golpe, usaremos el bloque "¿tocando el color ( )?" de la categoría "sensores", que permite detectar si la pelota toca ese color.



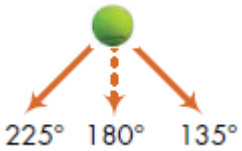
Paso 2: Añade la raqueta y la pelota.

- Raqueta: Usa el editor gráfico para dibujar el objeto raqueta. La raqueta es un rectángulo estrecho y corto, cuyo centro es aproximadamente el centro del rectángulo. Nombra a este objeto "raqueta", y a su disfraz "raqueta1".
- Pelota: Agrega la pelota (tennis ball) desde la biblioteca de objetos.

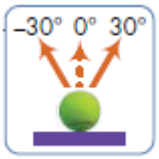
Paso 3: Comienza el juego y haz que los objetos se muevan. Para todos los objetos, el programa arranca al clicar en la bandera verde. Los programas para la raqueta y la pelota empiezan así:

- Raqueta: Sitúa a la raqueta en la posición inicial $(x, y) = (0, -120)$, y a continuación, haz que la coordenada x de la raqueta *siempre* sea la posición indicada por el ratón. Para ello, deberás usar el bloque de función "posición x del ratón" (categoría "sensores") como entrada del bloque "fijar x a ( )" (categoría "movimiento").

- Pelota: Sitúa a la pelota en la posición inicial $(x, y) = (0,\ 160)$. A continuación, haz que apunte en una dirección aleatoria entre 135 y 225 grados. Después de comenzar apuntando en una dirección aleatoria, la pelota *siempre* se irá moviendo en pasos de tamaño 12, y rebotando al llegar al borde del escenario.



Paso 4: Haz que la pelota rebote contra la raqueta. Completa el bucle infinito de la pelota añadiendo un bloque "si ( ) entonces" de la categoría "control". Con este bloque, indica que si la pelota toca la raqueta, la primera debe pasar a apuntar en una dirección aleatoria entre -30 y 30 grados.



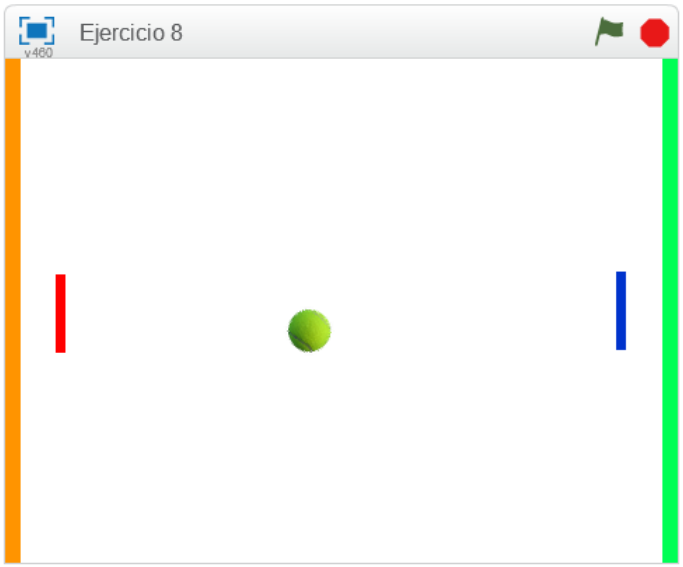
Paso 5: Haz que el juego termine si la pelota toca la franja de color. Antes o después del bloque "si ( ) entonces" previo, añade otro bloque "si ( ) entonces" independiente para parar el programa y los objetos, bloque "detener ( )", si la pelota toca el color rosa, bloque "¿tocando el color ( )?" de la categoría "sensores".

NOTA: Al clicar con el ratón sobre el cuadrado coloreado dentro del bloque "¿tocando el color ( )?", el puntero del ratón se convierte en una mano. Cuando movamos la mano sobre la pantalla, y cliquemos en el rectángulo rosa en la parte baja del escenario, el cuadrado coloreado del bloque "¿tocando el color ( )?" debería volverse de ese mismo color rosa. Así es como se define el color a detectar dentro del bloque "¿tocando el color ( )?".

Paso 6: Sonidos. Añade un sonido cada vez que la pelota golpee la raqueta. Selecciona la pelota de la lista de objetos, clica en la pestaña de sonidos, y busca un sonido adecuado de la biblioteca (o usa el sonido "pop" que tiene por defecto). Completa el programa de la pelota para que reproduzca ese sonido cada vez que la pelota impacte contra la raqueta. ¡Ya has terminado! Guarda el proyecto en tu carpeta de trabajo con el nombre **ejercicio7.sb2**.

EJERCICIO 8. TENIS A DOS JUGADORES.

Haz un programa para jugar al tenis a dos jugadores. Necesitarás tres objetos: dos raquetas idénticas de distinto color, y una pelota. El juego es similar al anterior, solo que ahora las raquetas se mueven en vertical, no en horizontal. El jugador 1 controla la raqueta roja con el ratón, mientras que el jugador 2 controla la raqueta azul con las flechas ↑ y ↓ del teclado. (También podemos hacer que el jugador 1 controle la raqueta roja con las teclas W y S, si no queremos usar el ratón). Ahora hay dos franjas de color (con distintos colores), una a la derecha y otra a la izquierda. El juego termina cuando la pelota toca cualquiera de las dos franjas. Guarda el proyecto en tu carpeta de trabajo con el nombre **ejercicio8.sb2**.



NOTA 1: La posición inicial para la pelota es $(x, y) = (0, 0)$, para la raqueta roja es $(x, y) = (-200, 0)$ y para la raqueta azul es $(x, y) = (200, 0)$.

NOTA 2: La pelota comienza orientándose en una dirección aleatoria entre 45 y 325 grados, y moviéndose en pasos de 12.

NOTA 3: Al chocar contra la raqueta roja, la pelota rebota y se orienta en una dirección aleatoria entre 40 y 140 grados. Al chocar contra la raqueta azul, la pelota rebota y se orienta en una dirección aleatoria entre 220 y 320 grados.

NOTA 4: Configura dos sonidos diferentes para cuando la pelota golpee la raqueta roja o la raqueta azul.

AMPLIACIÓN:
a) Crea un marcador para llevar el tanteo. Necesitarás dos variables, una para llevar la puntuación del jugador rojo, y otra para llevar la puntuación del jugador azul. Deberás actualizar en un punto el tanteo de

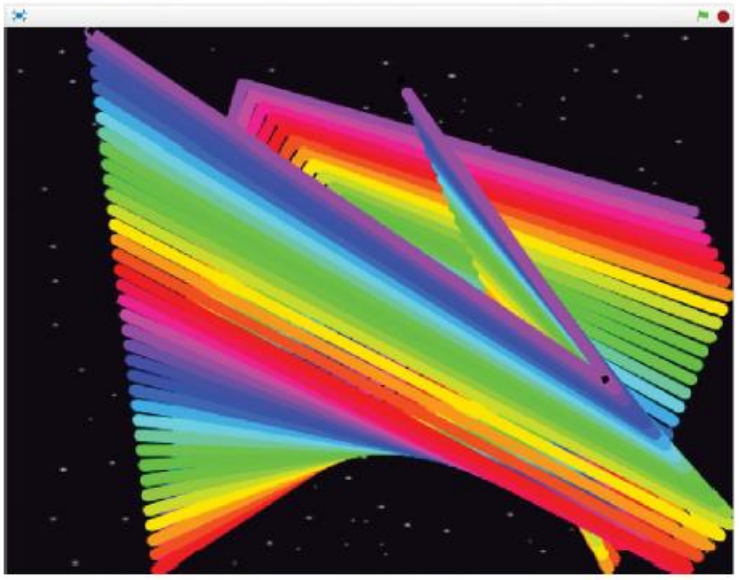
cada jugador cuando la pelota toque la franja vertical tras el jugador contrario. Detén el partido unos instantes cuando cualquier jugador se apunte un nuevo tanto, y tras la pausa, la pelota deberá volver a empezar como al principio (ver notas 1 y 2).
b) Permite que el usuario pueda configurar la velocidad de la pelota al principio del juego. La velocidad habitual son pasos de tamaño 12 (ver nota 2).
c) Haz que ambos jugadores puedan presionar una tecla de forma que la pelota desaparezca durante 1 segundo para después volver a aparecer. Esto hará que el juego sea más complicado.

2.4. PROYECTOS SCRATCH.

En esta sección vamos a desarrollar unos proyectos en Scratch que nos ayudarán a ir más allá en nuestra comprensión de los comandos de movimiento y de dibujo. Algunos de los programas usan bloques que aún no hemos visto; aquí solo los explicaremos brevemente, y los estudiaremos con mayor profundidad en capítulos posteriores.

PROYECTO 1. LÍNEAS DE ARCOIRIS EN EL ESPACIO.

En este primer proyecto vas a crear una "V" de arcoíris que vuele a través del espacio y deje una colorida estela tras ella (ver figura).



1) Elimina los objetos no deseados, y agrega el fondo "stars" de la biblioteca de Scratch.

2) Con el editor gráfico de Scratch, crea un objeto con la forma de un pequeño punto rojo. (Usa la herramienta pincel).

3) Haz el programa del punto rojo: Al clicar en la bandera, apunta en una dirección aleatoria entre −180 y 180 grados. Un bucle infinito le hace moverse en pasos de 10, rebotando si llega al borde del escenario. (Si al rebotar contra la pared observas algo extraño, tal vez debas fijar adecuadamente el estilo de la rotación).

4) Duplica el objeto punto dos veces hasta tener un total de tres puntos. (Los puntos duplicados también heredarán el código del punto original). Estos tres puntos representarán los tres vértices de la "V" voladora.

5) Ahora vamos a crear un cuarto punto para dibujar las líneas de arcoíris. Este punto se moverá muy rápido entre los tres puntos rebotadores, mientras dibuja la línea conforme se mueve. Este proceso se

repetirá indefinidamente, pero transcurridos 10 segundos, la pantalla se limpiará para volver a empezar. Para ello, comienza duplicando uno de los puntos rebotadores para crear el punto dibujante, y elimina el código heredado por este cuarto punto. Así tendremos 4 puntos (ver figura).

6) Escribe el código para el punto dibujante (punto4):

- Programa 1: Al ciclar en la bandera, limpiamos la pantalla. Ajustamos el tamaño de su lápiz a 8. Después, y dentro de un bucle infinito, subimos el lápiz, llevamos al punto 4 hasta la posición del punto1, bajamos el lápiz, y a continuación, lo llevamos a la posición del punto2, e inmediatamente después a la del punto3; por último, cambiamos el color del lápiz por 10.
- Programa 2: Como debemos limpiar la pantalla tras 10 segundos, al clicar en la bandera, y dentro de un bucle infinito, el punto4 espera 10 segundos, y a continuación, limpia la pantalla.

7) Si prefieres que no se vean los puntos conforme se mueven, añade a todos los programas el bloque "esconder" de la categoría "apariencia". (Esto se podría haber evitado dibujando los puntos muy pequeños).

8) Si al clicar sobre la bandera verde mantenemos pulsada la tecla SHIFT (mayúsculas), Scratch se pone en **modo turbo**, y la aplicación se ejecuta a toda velocidad. (Normalmente, Scratch redibuja la pantalla tras ejecutar cada instrucción. Pero en modo turbo, Scratch solo redibuja la pantalla después de haber ejecutado varias instrucciones, lo que acelera la ejecución del programa. Si la ejecución es lo suficientemente rápida, el ojo humano no es capaz de percibir estos cortes).

9) Guarda el proyecto como **proyecto1.sb2**.

AMPLIACIÓN 1: En lugar de una "V" voladora, vamos a hacer un triángulo volador. Para ello, modifica ligeramente el programa1 del punto4 para "cerrar" la "V" y crear un triángulo.

AMPLIACIÓN 2: En lugar de una "V" o de un triángulo, vamos a crear dos líneas voladoras. Para ello necesitarás 4 puntos rebotadores (punto1, punto2, punto3, punto 4) y dos punto dibujantes (punto5 y punto 6). El punto 5 irá entre los puntos 1 y 2, y el punto 6 entre los puntos 3 y 4.

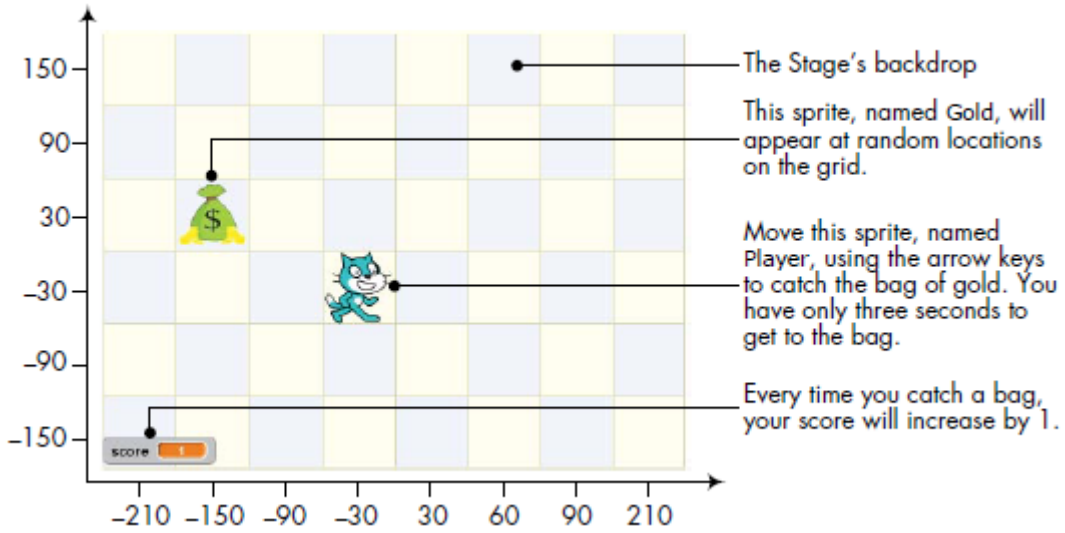
PROYECTO 2. CONSIGUE EL DINERO.

Crea un juego en el que el jugador mueve un personaje usando las teclas de las flechas. El objetivo es coger todas las bolsas de oro que pueda. Las bolsas aparecerán en posiciones aleatorias sobre una cuadrícula. Si el jugador no coge la bolsa en tres segundos, ésta se moverá a otro lugar. Para empezar con el proyecto, abre el archivo **Proyecto 2_sinCodigo.sb2**. Este proyecto ya tiene los objetos ("player" y "gold") y el fondo ("grid60") que necesitamos. Tú tendrás que escribir el código de los programas de control.

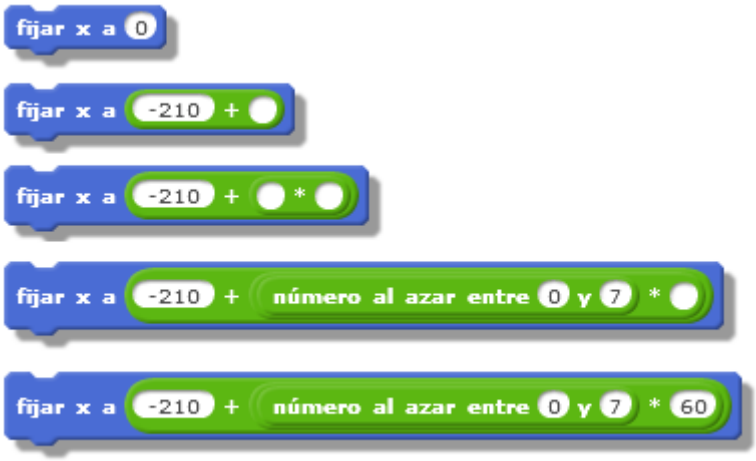
> NOTA: Necesitarás referirte a la figura para entender los números que usaremos en los programas. Acude a ella siempre que sea necesario.

Comenzamos creando el código para el personaje del jugador. Necesitarás crear 5 programas. En el programa de arranque, cuando el jugador clica en la bandera, el personaje se va a la posición inicial $(x, y) = (-30, -30)$ y apunta hacia la derecha. Ahora, deberás hacer otros 4 programas prácticamente iguales: Cuando el jugador presione una tecla de flecha, el programa correspondiente cambia la dirección

del personaje (en la dirección de la flecha), reproduce un sonido breve (pop), y mueve al personaje 60 pasos. Por supuesto, el personaje rebota al llegar al borde del escenario.



Ahora vamos con el código de la bolsa de oro. Comenzamos creando una variable llamada "score" (puntuación). Al clicar en la bandera, la variable "score" comienza a cero. A continuación, ponemos un bucle que se repetirá 20 veces, para mostrar un total de 20 bolsas de oro (este número no es obligatorio, cámbialo si quieres). A cada pasada del bucle, aparecerá una bolsa de oro en una posición aleatoria (para ello, fijamos el valor de la coordenada x a $-210 + (número\ al\ azar\ entre\ 0\ y\ 7) \times 60$, y el valor de la coordenada y a $-150 + (número\ al\ azar\ entre\ 0\ y\ 5) \times 60$). (Para entender de dónde salen estas fórmulas, ver nota1 más abajo). IMPORTANTE: La figura muestra los pasos detallados para construir correctamente el bloque que fija el valor de la coordenada x. (El bloque para fijar el valor de y es similar). Móntalo en el orden indicado, o las bolsas podrían no aparecer correctamente sobre la cuadrícula, o ni siquiera aparecer.



NOTA 1: Para hacer que la bolsa aparezca de forma aleatoria en una casilla cualquiera de las 48 existentes, la posición en x de la bolsa puede ser una cualquiera de entre $-210, -150, -90, \ldots, 210$. Estos números están espaciados 60 pasos, por lo que podemos obtener cada una de las posiciones x desde -210 como:

$$x = -210 + (0 \times 60)$$

$$x = -210 + (1 \times 60)$$

$$x = -210 + (2 \times 60)$$

...

$$x = -210 + (7 \times 60)$$

Después de aparecer en una casilla aleatoria, la bolsa de oro le dará al jugador un tiempo límite de 3 segundos para cogerla. Para controlar el tiempo, el programa de la bolsa reinicia el cronómetro de Scratch a cero, bloque "reiniciar cronómetro" de la categoría "sensores". A continuación, la bolsa espera hasta que el personaje la agarre al tocarla, o hasta que el cronómetro exceda los 3 segundos. Para ello, necesitarás el bloque "esperar hasta que ( )" de la categoría "control", y el bloque "( ) o ( )" de la categoría "operadores", el bloque "¿tocando ( )?" de la categoría "sensores", el bloque "( ) > ( )" de la categoría "operadores", y el bloque "cronómetro" de la categoría "sensores". Los pasos para construir este macrobloque se muestran en la figura:

Cuando cualquiera de estas dos condiciones se cumple el programa finaliza su espera, y continúa como indicamos:
(1) Si la bolsa toca al personaje "player", entonces reproduce el sonido "waterDrop", y cambiamos el valor de la variable "score" en 1 (el personaje ha cogido una bolsa de oro más).
(2) Por otro lado, si el valor de la variable "score" se hace igual a 20, entonces la bolsa dice "¡Has ganado!" durante 1 segundo, y el programa finaliza, bloque "detener (todos)".

NOTA 2: Al igual que el número de bolsas, que no tenía que ser 20, el tiempo que espera una bolsa antes de desaparecer no tiene por qué ser de 3 segundos. De hecho, el juego se hace más difícil si bajamos el tiempo a 2 ó 1 segundos. Prueba varias posibilidades.

Ya hemos terminado. Guarda el proyecto como **Proyecto 2.sb2**.

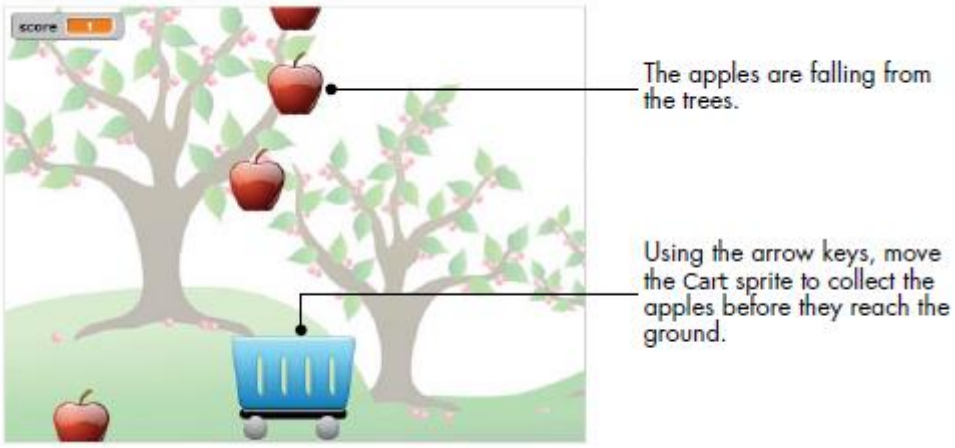
PROYECTO 3: ATRAPANDO MANZANAS.

En este juego, las manzanas aparecen en posiciones horizontales aleatorias en lo alto del escenario, en instantes de tiempo aleatorios, y caen hacia el suelo. El jugador debe mover el carrito y atrapar las manzanas antes de que lleguen al suelo. Cada manzana vale 1 punto.

Como antes, abre el archivo **Proyecto 3_sinCodigo.sb2**. Este proyecto ya tiene los objetos ("Cart" y "Apple") y los escenarios ("Apple_Stage_2") que necesitamos. Tú tendrás que escribir el código.

Puede que nos sorprenda ver que solo hay un objeto manzana. Después de todo, habrá muchas manzanas cayendo al mismo tiempo (ver figura). Por suerte, Scratch incorpora los bloques de **clonar** (categoría "control") que nos permiten crear muchas copias de un objeto.

El archivo también incluye la variable "score" (puntuación), para llevar la cuenta de las manzanas que vayamos cogiendo.



Comenzamos con el programa del carrito:

Programa 1 (carrito): Cuando el usuario clica en la bandera, llevamos al carrito al centro de la parte baja del escenario, coordenadas $(x, y) = (0, -180)$. A continuación, el programa *siempre* comprueba *si* el jugador está presionando las teclas derecha o izquierda, y mueve el carrito en consecuencia. Mediante prueba y error, comprobamos que la mejor forma de mover el carrito es en pasos de 30. Necesitarás estos bloques:

Ahora vamos con el código para el objeto "manzana". Necesitaremos dos programas:

Programa 1 (manzana): Este programa sirve para crear todos los clones del objeto "manzana".

El programa comienza al clicar en la bandera. Primero fijamos el valor de la variable "score" a cero. A continuación, hacemos que el objeto se haga visible mediante el bloque "mostrar" de la categoría "apariencia". Ahora empieza un bucle con 30 repeticiones, para hacer que caigan un total de 30 manzanas. A cada pasada del bucle, el objeto manzana se mueve a una posición horizontal aleatoria en la parte superior del escenario. A continuación, la manzana ejecuta el bloque "crear clon de (mí mismo)" de la categoría "control" para crear una copia de sí misma, y aguarda durante un intervalo de tiempo aleatorio entre $0{,}1$ y $1{,}5$ segundos antes de comenzar con la siguiente pasada del bucle. Fuera del bucle, y después de completar las 30 pasadas, el programa esconde el objeto manzana ejecutando el bloque "esconder".

Este programa funcionará cuando, al clicar en la bandera verde, veamos que se van creando copias de la manzana en la parte superior del escenario, pero en posiciones horizontales aleatorias. Además, el tiempo que pasa entre la creación de una copia y la siguiente también será aleatorio.

Programa 2 (manzana): Este programa es el que controla todos los clones creados en el programa previo, y nos permite hacerlos caer, hacerlos desaparecer, y llevar la cuenta de la puntuación.

Con el bloque "al comenzar como clon" de la categoría "control", cada clon de la manzana ejecutará este programa nada más ser creado. El programa empieza con un bucle "por siempre". Dentro del bucle, el clon

se mueve hacia abajo 10 pasos. A continuación, comprobamos si el clon está tocando al carro, y en ese caso, incrementamos la variable "score" en 1 unidad, reproducimos un sonido ("fairydust"), y borramos ese clon (bloque "borrar este clon" de la categoría "control"). Por otro lado, y aún dentro del bucle, comprobamos si la posición en y del clon es menor que -180 (esto es, si el clon ha llegado al suelo), y en ese caso, reproducimos el sonido "alien creak2" y borramos el clon. Si el clon no ha sido capturado por el carrito ni ha llegado al suelo, es porque aún está cayendo, y el bucle vuelve a ejecutarse una vez más.

NOTA IMPORTANTE: Esta forma de gestionar los clones de la manzana es la técnica que siempre usaremos cuando necesitemos crear múltiples copias (clones) de un objeto. En general, necesitaremos dos programas: (1) el programa que crea los clones (que empieza al pinchar en la bandera verde), y (2) el programa que le dice a los clones lo que deben hacer (que empieza con el bloque "al comenzar como clon").

AMPLIACIÓN: Modifica el programa previo para incluir un tercer objeto (por ejemplo, una manzana envenenada, o una bomba, etc.) que caiga de tanto en tanto junto con las manzanas, pero que al cogerla, haga que el usuario pierda la partida (o le reste 2 puntos a su puntuación total, o cualquier otra cosa que se te ocurra).

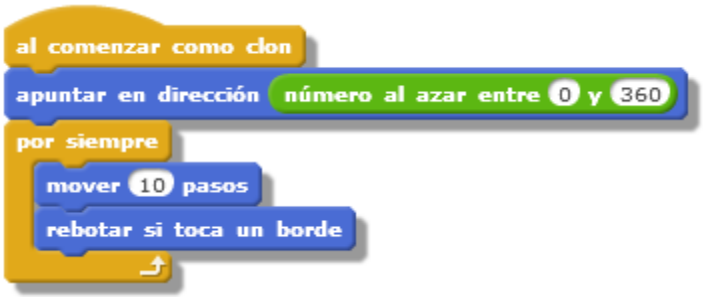
MÁS SOBRE EL CLONADO DE OBJETOS.

En el proyecto 3 hemos probado las instrucciones de clonado. Vamos a hablar un poco más sobre el clonado de objetos. Cualquier objeto puede crear un clon de sí mismo o de otro objeto con el bloque "crear un clon de ( )". (El propio escenario también puede crear clones mediante la misma instrucción). Un objeto clonado hereda el estado del objeto original (el objeto "maestro") en el momento en el que es clonado (esto es, la posición y dirección originales, el disfraz, el estado de visibilidad, color de lápiz, tamaño de lápiz, efectos gráficos, etc.). Los clones también heredan los programas del objeto "maestro". Por ejemplo, prueba el código de la figura (2 programas para el objeto gato), y observa qué ocurre.

Debemos tener cuidado al usar la instrucción "crear un clon" en un programa que no comience con el botón de la bandera, ya que podríamos terminar con más clones de los que pretendíamos en un principio. Por ejemplo, en el siguiente programa, al presionar por primera vez la tecla espaciadora, tendremos dos objetos (el maestro y el clon), pero al presionarla una segunda vez, ya tendremos cuatro (los dos de antes, más sus respectivos clones).

Este problema puede resolverse clonando objetos solo en programas que comiencen al clicar sobre la bandera verde. Estos programas son ejecutados únicamente por el objeto maestro.


al presionar tecla espacio
crear clon de mí mismo


al comenzar como clon
apuntar en dirección número al azar entre 0 y 360
por siempre
mover 10 pasos
rebotar si toca un borde

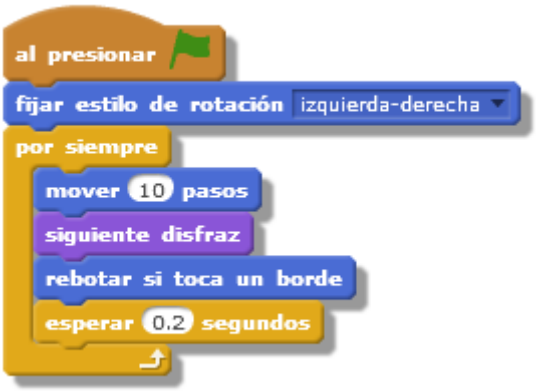
3. APARIENCIA Y SONIDO.

3.1. LA CATEGORÍA APARIENCIA.

Aunque podemos dibujar sobre el escenario mediante las instrucciones de dibujar (categoría "lápiz"), los disfraces proporcionan una forma mucho más sencilla de añadir gráficos a nuestros programas. Los comandos de la categoría "apariencia" nos permiten manipular disfraces para crear animaciones, añadir bocadillos de diálogo y pensamiento, aplicar efectos gráficos, y cambiar la visibilidad de un objeto.

CAMBIAR DISFRACES PARA ANIMAR UN OBJETO.

Sabemos mover un objeto de un punto a otro, pero los objetos estáticos se mueven de forma poco natural. Si usamos los distintos disfraces que posee un mismo objeto y cambiamos entre ellos lo suficientemente rápido, conseguiremos que el objeto se mueva de forma más realista. Por ejemplo, agrega el objeto "Avery walking" de la biblioteca (este objeto tiene 4 disfraces), y añádele el siguiente código:



La instrucción clave del programa es el bloque "siguiente disfraz", que le dice al objeto que se ponga el siguiente disfraz de su lista de disfraces. (Los números dentro de los bloques de "mover" y de "esperar" nos permiten controlar la rapidez de la animación).

Esta animación la podríamos haber conseguido con las instrucciones de dibujar, pero habríamos necesitado un programa muy largo y complejo. Sin embargo, y una vez hemos dibujado los disfraces, programar una animación es muy sencillo. Podemos usar el editor gráfico de Scratch o nuestro programa de dibujo favorito para crear los disfraces que necesitemos.

EJERCICIO 10. CARA SONRIENTE.

Abre el archivo **Ejercicio 10_sinCodigo.sb2**, donde encontrarás un objeto con 5 disfraces (la cara sonriente), y un escenario con 4 fondos. Tu tarea es programar la siguiente aplicación: Cada vez que cliquemos con el ratón sobre la imagen de la cara, el objeto "cara" cambiará al siguiente disfraz de su lista. Al mismo tiempo, la "cara" también cambiará el escenario a uno de sus cuatro fondos de forma aleatoria (usa el bloque "cambiar fondo a ( )" de la categoría "apariencia"). Por otro lado, cuando el escenario cambie a su fondo 4 (stage4), la "cara" detectará este evento (usando el bloque "cuando el fondo cambie a ( )" de la categoría "eventos"), y se deslizará hacia la esquina superior derecha del escenario, y de vuelta al centro. Guarda el proyecto como **Ejercicio 10.sb2**.

EJERCICIO 11. SEMÁFORO.

Abre el archivo **Ejercicio 11_sinCodigo.sb2**, y crea un programa para hacer que el semáforo siempre esté cambiando (de forma realista) entre el rojo, el verde, y el ámbar, usando el bloque "cambiar disfraz a ( )". Recuerda: Un semáforo "real" empieza en rojo, cambia al verde, después al ámbar, y finalmente vuelve al rojo. Esta secuencia se está repitiendo para siempre. La luz ámbar está encendida un tiempo más corto en comparación con las otras dos.

Guarda el proyecto como **Ejercicio 11.sb2**.

AMPLIACIÓN: Al cambiar del verde al rojo pasando a través del ámbar, puedes hacer que la luz ámbar parpadee durante unos instantes. Para ello, deberás crear un nuevo disfraz con las tres luces apagadas. (Duplica uno de los disfraces, borra la luz encendida, y copia una de las luces apagadas).

OBJETOS QUE HABLAN Y PIENSAN.

Los bloques "decir ( )" y "pensar ( )" hacen que nuestros objetos hablen o piensen como los personajes de un cómic. La frase que escribamos aparecerá dentro de un bocadillo sobre el objeto, y el mensaje se muestra de forma permanente. Para borrar el mensaje, usamos los bloques "decir ( )" y "pensar ( )" sin texto. Para mostrar un mensaje durante un tiempo limitado, utilizamos los bloques "decir ( ) por ( ) segundos" y "pensar ( ) por ( ) segundos".

EJERCICIO 12. DISCUSIÓN.

Crea un nuevo proyecto de Scratch y añade los personajes "cat1" (el gato de Scratch) y "gobo". Añade también el fondo "stage1". Mediante sendos programas, ubica a los dos personajes cara a cara sobre la tarima del escenario, y haz que ambos comiencen una discusión sin final, como muestra la figura. Ajusta la temporización y esperas de los personajes para que los diálogos sean realistas y estén sincronizados. Guarda el archivo como **Ejercicio 12.sb2**.

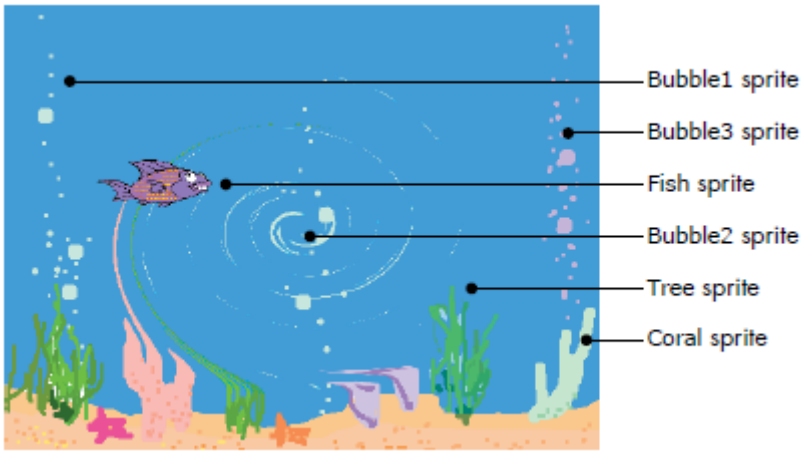
EFECTOS DE IMAGEN.

El comando "establecer efecto ( ) a ( )" permite aplicar distintos efectos gráficos a disfraces y fondos. Los nombres de estos efectos son color, ojo de pez, remolino, mosaico, etc. Para eliminar el/los efecto/s ajustado/s, debemos usar el bloque "quitar efectos gráficos". Prueba los diferentes efectos ejecutando el programa **Efectos gráficos.sb2**. Observa un par de aplicaciones prácticas de estos efectos probando el programa **Efectos gráficos (2).sb2**.

También podemos usar el comando "cambiar efecto ( ) por ( )" para reajustar el efecto en vez de establecerlo directamente. Por ejemplo, si hemos fijado el efecto "desvanecer" a 40, cambiarlo en 60 ajustaría el efecto a un total de 100, lo que haría que el objeto desapareciera por completo.

EJERCICIO 12(B). *ACUARIO*.

Abre el archivo **Aquarium.sb2**. La aplicación contiene seis objetos. Prueba diferentes efectos gráficos para animar el acuario. Algunas sugerencias:
1) Usa el efecto "remolino" sobre el escenario. Comienza con un número grande como 100 ó 200, y cambia progresivamente su valor para darle al agua un aspecto ondulado.
2) Cambia los disfraces de los objetos "bubble1" y "bubble2" a la velocidad adecuada para animar las burbujas.
3) Mueve el pez a través del escenario mientras cambias su disfraz.
4) Aplica el efecto "desvanecer" al objeto "tree".
5) Usa el efecto "color" sobre los objetos "coral" y "bubble3".
6) Añade algunos efectos más, y guarda el archivo como **Ejercicio 12b.sb2**.



EJERCICIO 13. *GATO FANTASMA*.

Programa al gato de Scratch para *animarlo* de forma que se mueva de derechas a izquierdas y rebote, mientras que aparece y desaparece como un fantasma. El efecto desvanecer acepta valores de 0 a 100, donde 0 implica mostrar totalmente el disfraz, y 100 hacerlo desaparecer por completo. Para que desaparezca tendrás que ir aumentando el efecto desvanecer en incrementos de 10, y para que vuelva a aparecer deberás reducirlo en incrementos de −10. Guarda el archivo como **Ejercicio 13.sb2**.

TAMAÑO Y VISIBILIDAD.

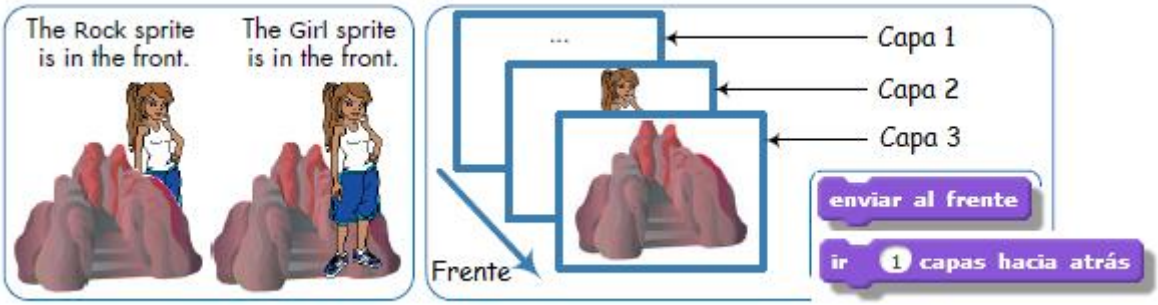
A veces necesitaremos cambiar el tamaño de un objeto, o controlar cuándo aparece en nuestro programa. Por ejemplo, puede que queramos que los objetos que estén más cerca parezcan más grandes, o tal vez queramos mostrar un objeto que nos indique las instrucciones solo al principio del juego.

Para aumentar o reducir el tamaño de un objeto, usamos los bloques "fijar tamaño a ( ) %" y "cambiar tamaño por ( )". El primero ajusta el tamaño de un objeto a un porcentaje de su tamaño *original*, y el segundo modifica el tamaño de un objeto en una cierta cantidad relativa a su tamaño *actual*. Para ver estas instrucciones en acción, prueba el programa de la figura.

Para hacer que un objeto aparezca o desaparezca, debemos usar los bloques "mostrar" o "esconder", respectivamente.

CAPAS.

Los dos últimos bloques de la categoría "apariencia" afectan al orden en el que los objetos se muestran sobre el escenario. Este orden determina qué objetos están visibles cuando se superponen.



Por ejemplo, en la figura tenemos dos objetos: la chica y la roca. Si queremos que la chica esté detrás de la roca, debemos enviar el objeto roca a la capa de delante, o enviar al objeto chica a la capa de atrás. Para ello, Scratch proporciona los comandos "enviar al frente" e "ir ( ) capas hacia atrás". El primero le dice a Scratch que dibuje al objeto en la capa delantera, y el segundo envía el objeto hacia atrás tantas capas como especifiquemos.

Para ver estas instrucciones en acción, prueba la aplicación **capas.sb2**. Tenemos 4 objetos que se mueven sobre el escenario. Las teclas "o" (orange), "b" (blue), "y" (yellow), y "p" (pink) sirven para enviar ese objeto al frente.

3.2. LA CATEGORÍA SONIDO.

En esta sección estudiaremos los bloques presentes en la categoría "sonido". Comenzaremos con los bloques que permiten incorporar archivos de audio a un proyecto, y controlar su reproducción. Después veremos los bloques que nos permiten tocar tambores y otros instrumentos musicales, controlar su volumen, y cambiar la velocidad (*tempo*) a la que se tocan los tambores y las notas musicales.

REPRODUCIR ARCHIVOS DE SONIDO.

Scratch solo permite reproducir archivos de sonido de tipo WAV o MP3. Los bloques que nos permiten reproducir estos archivos son "tocar sonido ( )", "tocar sonido ( ) y esperar", y "detener todos los sonidos". Los dos primeros sirven para reproducir un sonido. Sin embargo, el primero permite que las demás instrucciones se ejecuten mientras se está reproduciendo el sonido, mientras que el segundo detiene la ejecución de las instrucciones subsiguientes mientras el sonido se esté reproduciendo. El último bloque sirve para interrumpir todos los sonidos.

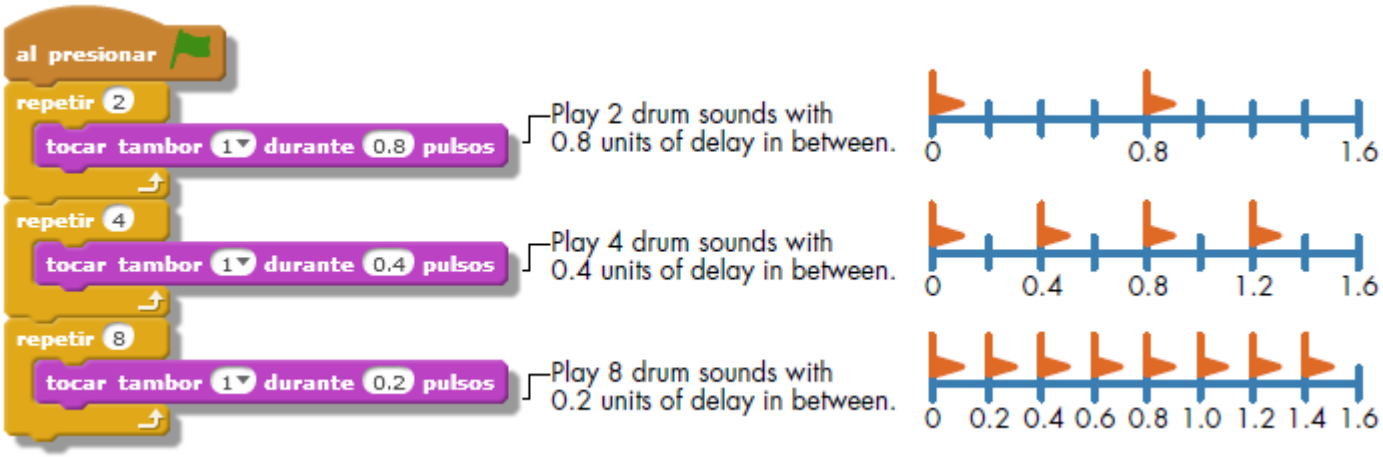
Para añadir música de fondo a un proyecto, basta con reproducir un archivo de audio repetidamente. Hay varias formas de hacerlo. La primera es sencilla, pero produce una pausa en el sonido entre dos reproducciones consecutivas. El segundo es más flexible, porque nos permite configurar la longitud del fragmento del sonido que vamos a reproducir.

EJERCICIO 13(B): LOBO AULLADOR.

Abre el archivo Wolf.sb2. Al clicar en la bandera, el lobo reproducirá el sonido "wolfHowl", que dura unos 4 segundos. Crea un programa en el que el lobo cambie de disfraz en sincronía con el sonido. (PISTA: Añade un bloque "esperar" con un retardo adecuado después de cada cambio de disfraz). Guarda el archivo como **ejecicio13b.sb2**.

TOCAR TAMBORES Y OTROS SONIDOS.

Normalmente querremos reproducir cortos sonidos cuando en el programa ocurra algo, por ejemplo, cuando un personaje impacte contra un obstáculo, cuando finalicemos un nivel, o cuando termine la partida. Para ello, podemos usar el bloque "tocar tambor ( ) durante ( ) pulsos", el cual permite tocar uno de los 18 tambores disponibles usando un cierto número de pulsos. También podemos añadir pausas con el bloque "silencio por ( ) pulsos". Prueba el bloque "tocar tambor" con el siguiente programa:



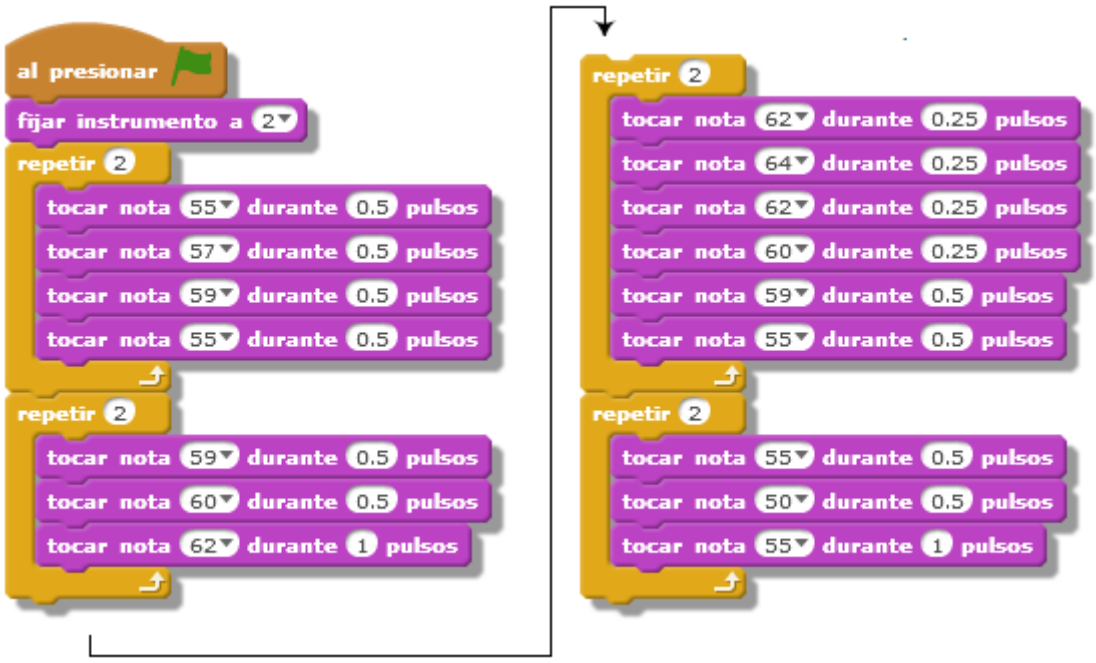
Este programa necesita una explicación: Cada "repetir" toca el mismo tambor usando distintos números de pulsos. Si imaginamos que el eje temporal está dividido en intervalos de $0{,}2$ *unidades temporales*, el primer bucle toca dos sonidos de tambor separados $0{,}8$ unidades de tiempo. El segundo bucle toca cuatro sonidos de tambor separados $0{,}4$ unidades temporales, y el tercer bucle toca ocho sonidos de tambor separados $0{,}2$ unidades temporales. Los tres bucles tardan el mismo tiempo en completarse ($1{,}6$ segundos), por lo que estamos tocando el tambor un número diferente de veces durante el mismo intervalo temporal.

Hemos dicho *unidades temporales* en vez de *segundos* porque el tiempo que tarda cada bucle en completarse depende del **tempo**, el cual puede ajustarse mediante la instrucción "fijar tempo a ( ) ppm". (Por defecto, el tempo está ajustado a 60 pulsos por minuto (ppm), esto es, 1 pulso por segundo). Como el tempo es de 60 ppm, cada bucle del ejemplo tarda $1{,}6$ segundos en completarse. Si ajustásemos el tempo a

120 ppm, cada bucle tardaría 0,8 segundos en completarse. A 30 ppm, cada bucle duraría 3,2 segundos, y así sucesivamente.

COMPONER MÚSICA.

Scratch también permite tocar notas musicales para componer música. El bloque "tocar nota ( ) durante ( ) pulsos" toca la nota que elijamos, con el número de pulsos que especifiquemos. El bloque "fijar instrumento a ( )" le dice a Scratch qué instrumento debe tocar. Vamos a usar estos comandos para crear una canción:



EJERCICIO 14. TOCO UNA CANCIÓN.

Busca en internet la partitura de una canción corta, donde las notas estén codificadas como A, B, C, D, E, F, G, y no como Do, Re, Mi, Fa, Sol, La, Si. (Si no encuentras ninguna, acude al archivo **canciones.rar** y elige una). Crea una aplicación que reproduzca esa canción en Scratch de la manera más virtuosa posible. Guarda el archivo como **Ejercicio14.sb2**.

CONTROLAR EL VOLUMEN.

Imagina que quieres que un sonido se extinga poco a poco en respuesta a algún evento (por ejemplo, un cohete que, tras despegar, se aleja hacia el espacio profundo). Scratch incorpora una serie de comandos para controlar el volumen de los archivos de audio, los sonidos de tambor, y las notas musicales. El bloque "fijar volumen a ( ) %" fija el volumen de un sonido a un porcentaje de su volumen original. Pero cuidado: este comando solo afecta al sonido del objeto que lo use, por lo que si queremos reproducir varios sonidos al mismo tiempo con diferentes volúmenes, deberemos usar varios objetos. El bloque "cambiar volumen por ( )" aumenta o reduce el volumen en el número que insertemos. El bloque "volumen" sirve para mostrar por pantalla el volumen de un objeto. Estos bloques son útiles, por ejemplo, cuando queremos cambiar el volumen dependiendo de lo cerca o lejos que esté un personaje de su objetivo (como en la búsqueda del tesoro), para hacer que ciertas partes de una canción se toquen más fuertes que otras, para simular una orquesta tocando diferentes instrumentos simultáneamente (con diferentes volúmenes), etc.

AJUSTAR EL TEMPO.

Los tres últimos bloques de la categoría "sonido" están relacionados con el tempo, o rapidez, con el que se reproducen los tambores y las notas. El tempo se mide en pulsaciones por minuto (ppm). A mayor tempo, más rápido se reproducirán las notas y tambores.

Scratch permite elegir un tempo específico con el comando "fijar tempo a ( ) ppm". También podemos decirle a un objeto que acelere o ralentice el tempo en una cierta cantidad mediante el comendo "cambiar tempo por ( )". Si queremos ver el tempo de un objeto por pantalla, activa la casilla junto al bloque de función "tempo". Para probar estos comandos, abre el archivo **tempoDemo.sb2**, y ejecútalo.

EJERCICIO 15. GATO MAULLADOR.

Abre el archivo **ejercicio 15_sinCodigo**, donde encontrarás un objeto gato y un escenario con un camino que se aleja entre los árboles. Crea un programa que, al clicar en la bandera, haga que el gato recorra el camino entre los árboles, mientras su tamaño se reduce progresivamente conforme se aleja. Además, haz otro programa que haga que el gato maúlle, pero que sus maullidos se oigan cada vez más bajos conforme se aleja. Guarda el archivo como **Ejercicio15.sb2**.

EJERCICIO 16. NATURALEZA.

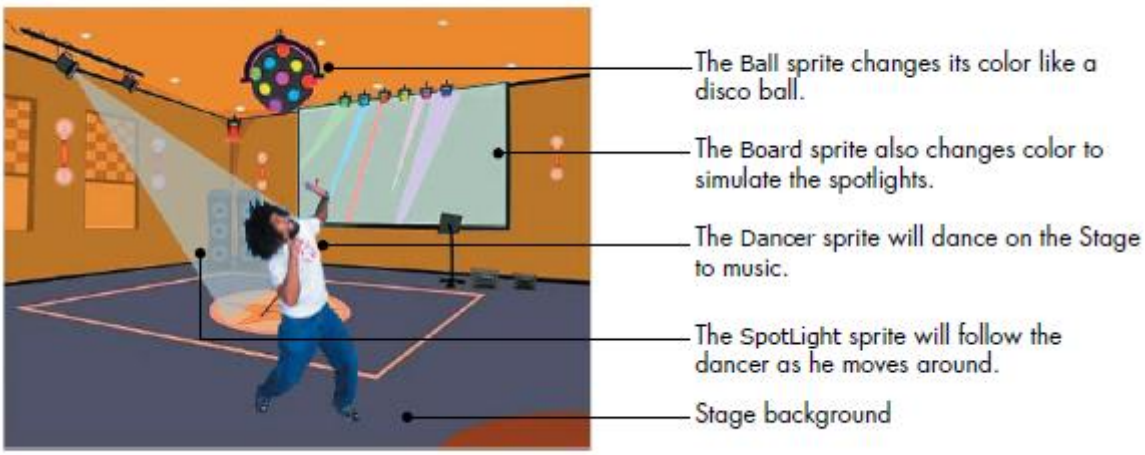
Abre el archivo **Nature.sb2**. La aplicación contiene tres objetos, una foca (seal), un pato (duck) y un pájaro (bird). Anima la escena usando tanto movimiento como sonido:

1) El objeto pájaro tiene dos disfraces para crear un efecto de vuelo. Escribe un programa para hacer que el pájaro vuele a través del escenario, y reproduzca el sonido "bird" en instantes de tiempo aleatorios.
2) El objeto pato tiene 12 disfraces que muestran al pato sacando un pez del agua y comiéndoselo. Escribe un programa para hacer que el pato se mueva a través del escenario, y reproduzca el sonido "duck" en instantes de tiempo aleatorios.
3) El objeto foca tiene 4 disfraces que la muestran jugando con una pelota. Crea un programa para que la foca se mueva jugando por el escenario y reproduzca el sonido "seal" en instantes de tiempo aleatorios.
4) Guarda el archivo como **ejercicio16.sb2**.

3.3. PROYECTOS SCRATCH.

PROYECTO 4. BAILANDO EN EL ESCENARIO.

En este proyecto, animarás a un personaje para que baile en un escenario.



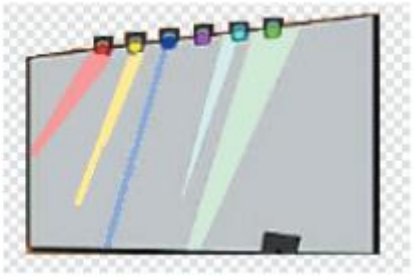
1) Crea un nuevo proyecto, y añade el fondo "party room" y el personaje "Dan". Borra los fondos y objetos por defecto, porque no los necesitaremos.

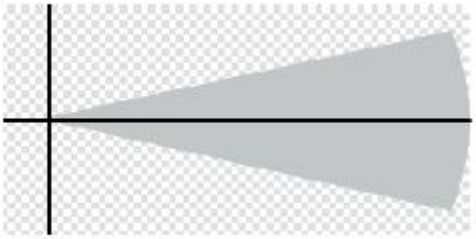
2) Agrega la música de fondo al escenario. El archivo que debes buscar se llama "medieval1" de la carpeta "bucles de música". Este archivo dura $9{,}6$ segundos.

3) Escribe un programa al escenario para que reproduzca continuamente la música, de forma que el archivo de audio se reanude sin cortes y sin saltos cada vez que vuelva a comenzar.

4) Crea el programa del bailarín. El programa comienza ubicando al personaje sobre el escenario, posición $(x, y) = (0, -60)$, y quitando todos los efectos gráficos. A continuación, y de forma indefinida, el bailarín dará 20 pasos de tamaño 1 hacia la derecha mientras cambia el efecto ojo de pez en $+1$ unidades a cada paso. Después, cambia de disfraz, y da 20 pasos de tamaño 1 hacia la izquierda mientras cambia el efecto ojo de pez en -1 unidades a cada paso. Por último, cambia de nuevo el disfraz, y la coreografía se repite.

5) Crea los objetos bola, tablón, y foco (ver figuras), y escribe sus programas.





Objeto "bola":

Guarda en tu carpeta el archivo del escenario, ciclando sobre el escenario, y seleccionando la opción "save picture of stage". (Asegúrate que el bailarín no está tapando el tablón del fondo). Ahora, y en la lista de objetos, clica en la opción "cargar objeto desde archivo", y selecciona el archivo que has guardado antes. Llama a este objeto "bola", y edita su disfraz en editor gráfico de Scratch borrándolo todo excepto la bola de colores. (Usa la herramienta lupa para hacer zoom y ajustarte bien a los bordes). (Cuidado: no cambies el centro de este nuevo objeto, ya que queremos que se superponga sobre la bola del escenario).

El programa de la bola comienza limpiando todos los efectos gráficos, y a continuación, cambia el efecto color en 1 unidades de forma indefinida.

Objeto "tablón":

Crea el objeto tablón de la misma forma que creaste el objeto bola. Agrega algunos colores a los reflejos de las luces sobre el tablón, como en la figura de los apuntes. El programa del tablón comienza limpiando todos los efectos gráficos, lo envía dos capas hacia atrás, y a continuación, cambia el efecto color en 1 unidades de forma indefinida.

Objeto "foco":

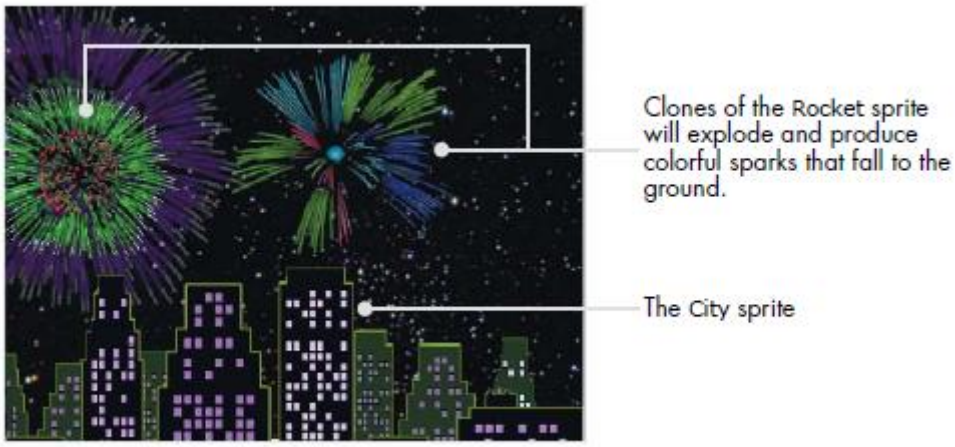
Crea el objeto foco partiendo de cero mediante el editor gráfico de Scratch. Para ello, selecciona la opción "dibujar nuevo objeto" de la lista de objetos. El centro del disfraz lo ubicarás en la punta del cono que representa el haz de luz.

El programa del foco comienza aplicándole el efecto desvanecer a 30, para hacerlo transparente. Después envía al objeto una capa hacia atrás, para ubicarlo detrás del bailarín. A continuación lo posiciona de forma que la punta del haz de luz coincida con la posición del foco en la esquina superior izquierda del escenario. Por último, un bucle infinito hace que el haz de luz siempre apunte hacia el bailarín, espere 0,1 segundos, y cambie de color en pasos de 10.

6) Guarda el proyecto en tu carpeta con el nombre **proyecto 4.sb2**.

PROYECTO 5: FUEGOS ARTIFICIALES.

En este proyecto vamos a crear una escena animada de fuegos artificiales. Los cohetes lanzados explotarán en instantes de tiempo aleatorios, produciendo chispas que caerán descendiendo lentamente.



1) Abre el archivo **Proyecto 5_sinCodigo.sb2**. Esta aplicación tiene dos objetos: la ciudad (city) y el cohete (rocket). El objeto cohete estará constantemente creando clones que explotarán en el cielo. Además, el cohete tiene 8 disfraces, donde el primero solo es un pequeño punto que lanzaremos hacia el cielo. Cuando el cohete llegue a su destino, que se elige al azar, cambiará a uno de los otros 7 disfraces (también elegido al azar) para simular la explosión inicial. Usaremos un efecto gráfico para hacer que la explosión parezca más realista.

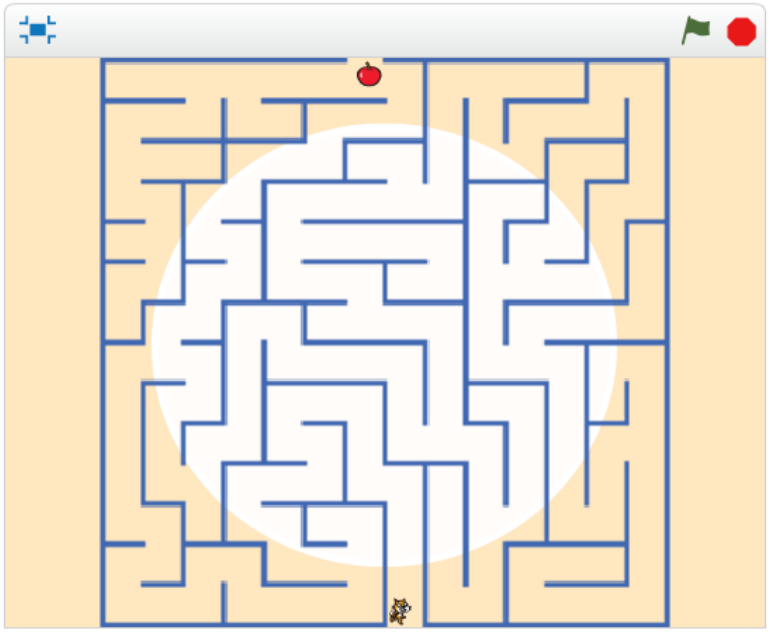
2) Escribe el programa para el objeto "cohete". El programa empieza al clicar en la bandera verde. Después de esconder el objeto cohete, empieza a crear clones de sí mismo de forma indefinida en instantes de tiempo aleatorios (espera aleatoria de entre 0,2 y 2,5 segundos). Como los clones heredan el estado de objeto maestro, también estarán escondidos al principio.

3) Ahora le decimos a los cohetes clonados lo que deben hacer: Cada cohete clonado comienza poniéndose su primer disfraz (el punto rojo). Entonces el clon se mueve a una posición horizontal aleatoria en la parte baja del escenario, se muestra, y se desliza (bloque "deslizar") a una posición aleatoria sobre el escenario (en algún lugar por encima de los edificios). Esta parte del programa es la que lanza al cohete (el punto rojo). Cuando el clon del cohete llega a su posición final en el cielo, explota. Para ello, elige uno de los disfraces del 2 al 8 de forma aleatoria, y reproduce un breve sonido de tambor (el sonido de la explosión). La explosión comienza siendo pequeña (fijando su tamaño a un 20% de su tamaño inicial) y se expande (usa un bucle "repetir" de 20 pasadas para incrementar el tamaño progresivamente en 4 unidades a cada pasada). Al salir del bucle, el clon se borra a sí mismo.

4) Guarda el archivo como **proyecto5.sb2**.

PROYECTO 6. EL CORREDOR DEL LABERINTO.

Vamos a hacer una aplicación para guiar al gato de Scratch a través de un laberinto hasta llegar a su objetivo, una deliciosa manzana. Para ello moveremos al gato mediante las teclas de las flechas, y bloquearemos su avance cuando intente pasar a través de las paredes.



1) Descarga la imagen del laberinto. Para ello, ve a la lista de objetos, y selecciona "cargar objeto desde archivo". Selecciona el archivo **laberinto.sprite2**. Se trata de un objeto con 8 disfraces, que se corresponden con 8 laberintos diferentes.

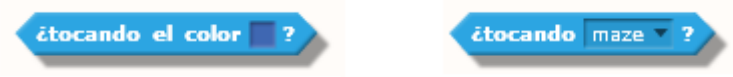
2) Cambia el fondo del escenario: Agrega el fondo "light" de la biblioteca.

3) Vamos a crear el programa del objeto laberinto. El programa empieza al clicar en la bandera verde. Mediante un bloque "cambiar disfraz a ( )", el programa fija el disfraz "maze1" (el primer laberinto). Además, y para asegurarnos de que el laberinto quede centrado en el escenario, ubicamos al objeto en la posición $(x, y) = (0, 0)$.

4) Seguimos con el programa del gato: En primer lugar, y nada más presionar la bandera verde, ajusta el tamaño del gato a un 15% de su tamaño original, de forma que quepa en los pasillos del laberinto. A continuación, ubica al gato en la salida del primer laberinto. Ahora, anima al gato y haz que avance en la dirección apropiada (en pasos de 4) al presionar las flechas del teclado (esto ya lo has hecho antes en varios ejercicios).

5) Hasta ahora, el gato avanza en pasos de 4 unidades atravesando paredes y muros. Ahora queremos remediar este defecto. Por ejemplo, si al moverse hacia la derecha toca un muro del laberinto, entonces el gato debería moverse hacia la izquierda 4 unidades. Modifica el programa previo para evitar que el gato atraviese muros al moverse hacia arriba, hacia abajo, hacia la derecha, y hacia la izquierda.

NOTA: Para detectar si el gato toca un muro, puedes hacerlo detectando si toca el color azul del muro, o detectando si toca el objeto "laberinto" (maze). (Sin embargo, si optas por detectar el color azul del muro, asegúrate de seleccionar el color de forma muy precisa. De no ser así, el gato seguirá atravesando los muros).



6) Añade el objetivo al final del laberinto. El final del laberinto lo marca la ubicación de una manzana. Añade al proyecto el objeto "apple". El programa de la manzana simplemente reduce su tamaño en un 25% de su tamaño original, y la ubica en la salida del laberinto1, posición $(x, y) = (-10, 170)$.

7) Detecta cuándo el gato llega a la manzana. Si el gato toca la manzana, entonces dice "¡Gané!" durante unos segundos, vuelve a la salida del laberinto, y envía un mensaje al objeto laberinto para que cambie de disfraz y se cargue el siguiente laberinto (ver nota a continuación). Crea el programa para hacer todo esto.

> NOTA: Scratch permite que los objetos se envíen mensajes entre sí mediante el bloque "enviar ( )" de la categoría "eventos". En el bloque, selecciona "nuevo mensaje" y escribe el mensaje que el gato va a enviar ("siguiente laberinto").

8) Crea un pequeño programa para que, cuando el objeto laberinto reciba el mensaje del gato, bloque "al recibir ( )" de la categoría "eventos", cambie al siguiente disfraz de su lista.

9) Guarda el proyecto como **proyecto 6.sb2**.

AMPLIACIÓN 1: MODO DOS JUGADORES.

Modifica el proyecto para añadir un segundo jugador. Los dos jugadores correrán uno contra el otro. El jugador 1 comienza en la parte inferior del laberinto y corre hacia la superior, y el jugador 2 comienza en la parte superior y corre hacia la inferior.

El jugador 2 necesitará llegar a su propia manzana en la parte inferior. Para ello, duplica el objeto manzana, y edítala para convertirla en una manzana verde (herramienta "colorear" del editor gráfico de Scratch). Crea un programa para la manzana 2, que será esencialmente igual al de la manzana 1, pero que la ubicará en la parte inferior del laberinto.

El personaje del jugador 2 será una réplica del personaje del jugador 1. Para diferenciarlos, modifica los disfraces del personaje 2 y píntalos de azul. Por supuesto, el gato 1 y el gato 2 tendrán prácticamente el mismo código. Modifica el programa de control del gato azul para poder moverlo con las teclas WASD, para que comience en la parte superior del laberinto, y para que su objetivo sea llegar a la manzana verde.

Jugador 1 Jugador 2

Por último, los personajes vuelven a sus posiciones iniciales cuando tocan sus respectivas manzanas en primer lugar, pero ahora también cuando el otro gato les gana la partida. Notar que sabemos qué gato llega el primero a su manzana porque es el primero en enviar el mensaje "siguiente laberinto". Por consiguiente, cuando un gato reciba dicho mensaje, es porque ha perdido la partida, y debe volver a su posición inicial.

AMPLIACIÓN 2. TRAMPAS.

Hasta ahora, recorrer el laberinto es muy fácil para el jugador. Vamos a hacer la partida más difícil añadiendo trampas. Las trampas sacan unas cuchillas cuando se activan, y después las repliegan cuando se desactivan. Si un jugador toca las cuchillas cuando están fuera, el jugador avanzará más despacio, lo cual le dará ventaja a su contrincante.

Crea un nuevo objeto llamado "trampa". El objeto tendrá 2 disfraces que deberás dibujar, uno con las cuchillas replegadas y otro con las cuchillas sacadas. (Pinta las cuchillas de un color gris que no hayas usado para ninguna otra cosa en el escenario).

NOTA: Asegúrate de que la trampa es lo suficientemente pequeña como para caber en el laberinto. Otra opción es dibujarla grande, y cambiar su tamaño en el programa de control.

Como querremos varias trampas en el laberinto, lo primero que debemos hacer es escribir un programa para crear varios clones del objeto trampa. En concreto, escribe un programa que, nada más comenzar la partida, muestre la trampa, cree 6 clones, y a continuación esconda la trampa.

El programa para cada clon creado comienza ubicándolo en una posición aleatoria (x, y) entre -170 y 170 para ambas coordenadas. Tras una espera de 2 segundos, y mediante un bucle infinito, vamos cambiando a intervalos de tiempo aleatorios entre los dos disfraces de la trampa (cuchillas replegadas, cuchillas sacadas). Para ello, ponemos el disfraz sin cuchillas, esperamos un tiempo aleatorio de entre 5 y 8 segundos, cambiamos al disfraz con cuchillas, esperamos un tiempo aleatorio de entre 1 y 4 segundos, y así constantemente.

Ahora modificamos el código de los gatos. Las cuchillas las hemos pintado de color gris, y no hay nada más en el escenario de ese color. Por lo tanto, cuando cualquier gato esté tocando ese color (es decir, cuando esté tocando las cuchillas), se parará para decir "¡Ay!" durante dos segundos.

AMPLIACIÓN 3. TIEMPO LIMITADO.

Hasta ahora, los jugadores disponen de un tiempo ilimitado para llegar a coger sus respectivas manzanas. El juego sería un poco más difícil si les damos un tiempo límite para alcanzar la manzana (digamos, 30 segundos). Modifica la aplicación para que, al inicio de la partida, arranque un cronómetro que les da a los jugadores un tiempo máximo de 30 segundos. Agotado este tiempo, y si ningún jugador ha conseguido llegar a su manzana, ambos pierden y la partida termina.

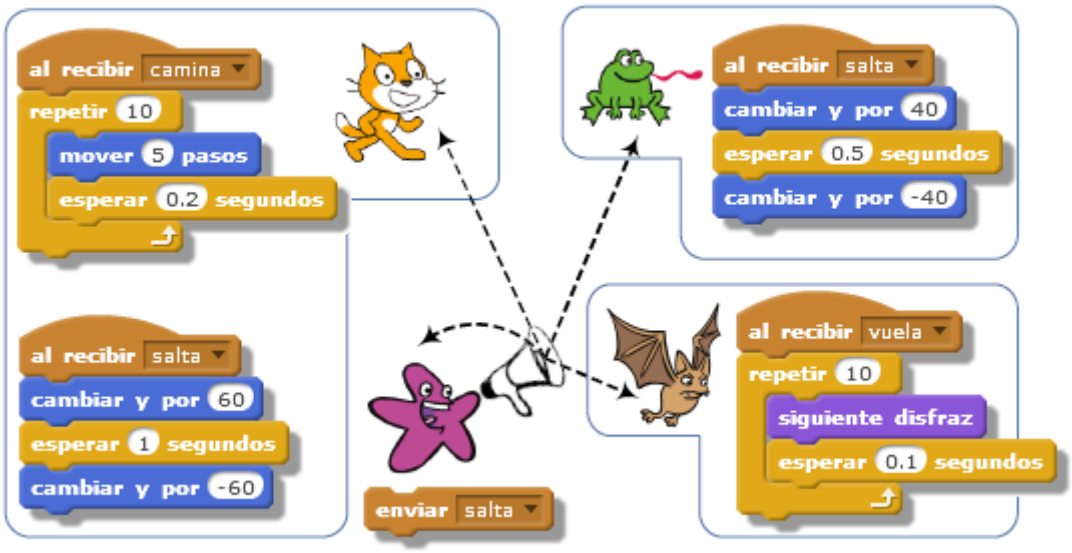
4. PROCEDIMIENTOS.

En lugar de construir nuestros programas como una única gran pieza, habitualmente es más conveniente dividir el problema a resolver en piezas independientes más pequeñas y manejables, llamadas *procedimientos*. Un **procedimiento** es una secuencia de comandos que realiza una función específica dentro de un programa. La utilización de procedimientos hace que nuestros programas sean más fáciles de escribir, de testear, y de depurar.

4.1. ENVÍO Y RECEPCIÓN DE MENSAJES.

Hasta ahora, casi todas las aplicaciones que hemos desarrollado solo contenían un objeto (o varios objetos que operaban de forma independiente entre sí). Sin embargo, muchas aplicaciones requieren de múltiples objetos que trabajen juntos. Necesitamos una forma de sincronizar las tareas asignadas a cada uno de los objetos. En esta sección aprenderemos a usar el mecanismo de envío de mensajes entre objetos para coordinar los trabajos realizados por los distintos objetos de nuestra aplicación. (En las antiguas versiones de Scratch, ésta era la única forma de implementar procedimientos. Más adelante veremos que la versión actual de Scratch permite crear "bloques customizados" para estructurar programas grandes en varios procedimientos más pequeños).

Cualquier objeto de nuestra aplicación puede enviar un mensaje (al cual podemos darle el nombre que queramos) usando los bloques "enviar (mensaje1)" y "enviar (mensaje1) y esperar" de la categoría de "eventos". El envío de un mensaje activará cualquier programa de cualquier objeto que comience con un bloque "al recibir (mensaje1)". Esto es, todos los objetos escuchan el mensaje enviado, pero solo responderán aquellos que tengan un programa que comience con el correspondiente bloque "al recibir ( )".

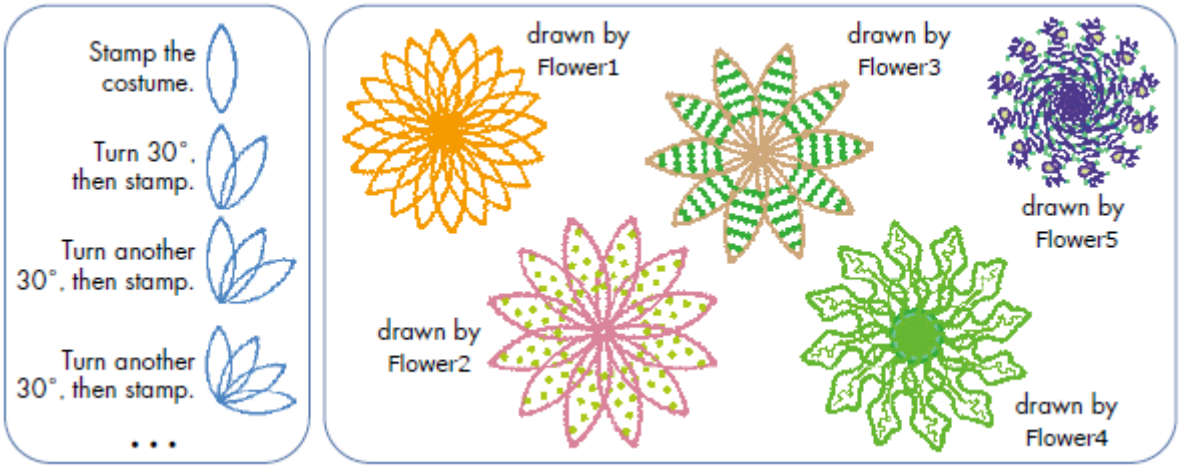


Por ejemplo, en la aplicación de la figura tenemos cuatro objetos: la estrella de mar, el gato, la rana, y el murciélago. La estrella de mar envía el mensaje "salta", y este mensaje se difunde a todos los objetos, incluyendo la propia estrella. En respuesta a este mensaje, tanto el gato como la rana ejecutarán los programas que comienzan con el bloque "al recibir (salta)". Observa que cada objeto salta de forma distinta, ejecutando su propio programa. El murciélago también recibe el mensaje, pero no actúa en respuesta porque no le hemos dicho que haga nada cuando lo reciba. Notar también que el gato no ejecuta el programa para caminar, porque este mensaje no se ha enviado.

El bloque "enviar ( ) y esperar" funciona como el bloque "enviar ( )", con la diferencia de que el primero espera a que todos los receptores del mensaje hayan terminado de ejecutar sus correspondientes bloques "al recibir ( )" antes de continuar con su programa.

ACTIVIDAD GUIADA 7. FLORES.

Para ver la forma en la que varios objetos responden a un mismo mensaje, vamos a crear una aplicación que dibuje varias flores en el escenario en respuesta a un clic de ratón. Abre el archivo **Flowers.sb2**. La aplicación tiene un fondo azul y 5 objetos "flower" (de flower1 hasta flower5), donde cada objeto tiene un disfraz con fondo transparente y centro ubicado en su extremo. Los objetos "flower" se encargarán de dibujar cinco flores en el escenario. Cuando un objeto reciba un mensaje para dibujar su flor, estampará múltiples copias rotadas de su disfraz sobre el escenario.



El programa comienza al clicar con el ratón sobre el escenario. El escenario detectará este evento (con el bloque "al presionar escenario" de la categoría "eventos"). En respuesta, borra la pantalla y envía el mensaje "dibujar". Los cinco objetos "flower" responden a este mensaje ejecutando un programa similar al siguiente:

1) El programa comienza con el bloque "al recibir (dibujar)".

2) A continuación, ajusta el efecto "color" a un número aleatorio entre 0 y 100, ajusta el efecto "brillantez" a un número aleatorio entre −20 y 20, y ajusta el tamaño del disfraz a un valor aleatorio de entre un 80% y un 100% de su tamaño original.

3) Envía al objeto a la posición $x = -160$ e $y = (número\ aleatorio\ entre\ -100\ y\ 100)$.

4) Por último, repite 20 veces la acción de estampar una copia de sí mismo (bloque "sellar") y de girar hacia la derecha 18 grados. (Notar que $20\ repeticiones\ \times 18° = 360°$).

Por supuesto, el paso (3) variará para cada objeto, ya que deberemos ubicarlos en unas coordenadas x distintas para que las flores no se superpongan. Por ejemplo, si para flower1 elegimos $x = -160$, para el resto podemos fijar $x = -80$, $x = 0$, $x = 80$, y $x = 160$. El paso (4) también debemos ajustarlo adecuadamente para cada objeto, ya que el número de repeticiones y la cantidad de grados girados pueden ser diferentes en cada caso.

5) Guarda el proyecto como **Guiada7.sb2**.

EJERCICIO 17. CUADRADOS DE COLOR.

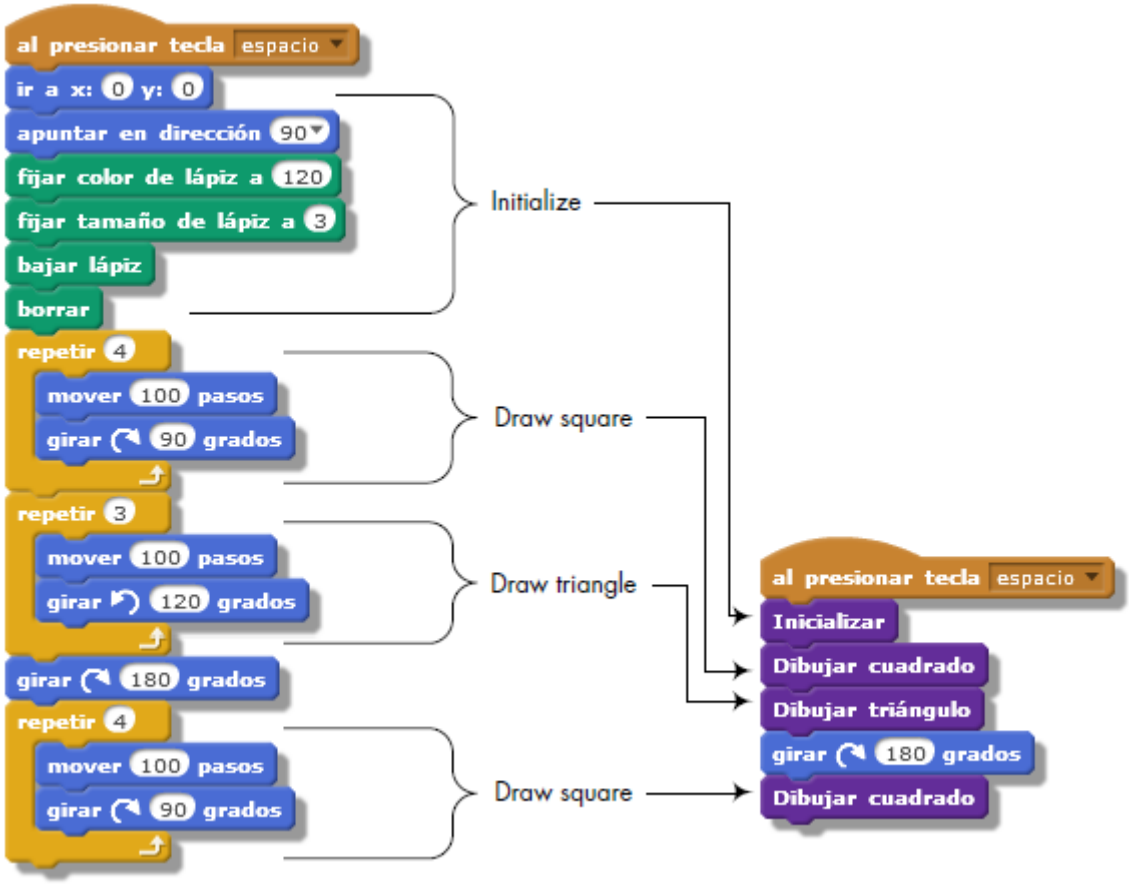
Para poner en práctica el envío y la recepción de mensajes, vamos a crear una aplicación que dibuje cuadrados con un color aleatorio. Abre el archivo **Ejercicio 17_sinCodigo.sb2**. La aplicación incluye un objeto lápiz y un fondo de color. Cuando el usuario clique con el ratón sobre el escenario, el escenario detectará este evento y enviará un mensaje al que llamaremos "cuadrado". (Añade también un programa

para el escenario que borre la pantalla al clicar en el botón verde). Ahora, cuando el objeto lápiz reciba este mensaje, se moverá a la posición actual del ratón, y allí dibujará un cuadrado con un color aleatorio. (El bloque "fijar color del lápiz a ( )" acepta como entrada un número entre 1 y 200, que determina el color del lápiz). Al terminar, guarda el archivo como **Ejercicio 17.sb2**.

4.2. ESTRUCTURAR PROGRAMAS GRANDES EN PIEZAS MÁS PEQUEÑAS.

Ahora que ya sabemos cómo funciona el envío y la recepción de mensajes, vamos a presentar la programación estructurada como método para reducir la complejidad de los programas grandes.

Hasta ahora, los programas que hemos desarrollado son relativamente cortos y sencillos. Pero antes o después, nos enfrentaremos al reto de escribir programas más complejos que contengan cientos de bloques. La **programación estructurada** permite simplificar el proceso de escribir, entender, y mantener programas informáticos. En lugar de tener un único gran programa, este método consiste en dividir el problema en piezas más pequeñas, donde cada una de ellas se encarga de resolver una parte de la tarea global.



Así, considera el programa mostrado en la figura, que simplemente hace un dibujo sobre el escenario. Como vemos, podemos dividir el proceso completo en bloques lógicos más pequeños, cada uno con una función. Por ejemplo, los primeros seis bloques del programa largo sirven para inicializar el objeto. El primer bucle "repetir" dibuja un cuadrado, el segundo bucle un triángulo, y así sucesivamente. Usando la programación estructurada, podemos agrupar bloques relacionados bajo un nombre representativo para formar procedimientos. Una vez tenemos escritos estos procedimientos, podemos utilizarlos en una cierta secuencia para resolver el problema que tenemos entre manos. Además, los procedimientos nos evitan tener que escribir el mismo código varias veces: por ejemplo, en nuestro proyecto hemos de dibujar el cuadrado dos veces, por lo que hemos usado el procedimiento "dibujar cuadrado" en dos ocasiones.

El uso de procedimientos nos permite aplicar la estrategia de "divide y vencerás" a la resolución de problemas complejos. La estrategia consiste en dividir un problema grande y difícil en piezas

independientes más pequeñas y sencillas, para resolver estas piezas individualmente. Después de resolver estas piezas por separado, las juntamos todas de una forma tal que nos permita resolver el problema original.

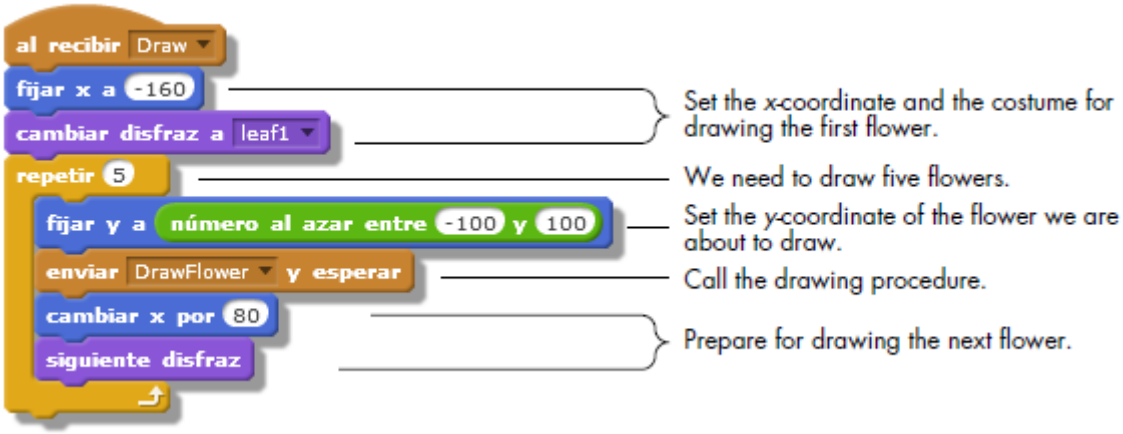
Ahora, ¿cómo se crean estos procedimientos? Hay dos formas de hacerlo. (1) Antes de la nueva versión de Scratch no podíamos construir los bloques que hemos mostrado en la figura. La única forma de emular procedimientos era mediante el mecanismo de envío y recepción de mensajes. (2) La nueva versión de Scratch incorpora la característica de crear y añadir bloques customizados que permiten implementar los procedimientos necesarios.

En esta sección comenzaremos mostrando la forma antigua de hacer las cosas. Después explicaremos el método de construir nuestros propios bloques, y de hecho, será el método que usaremos en el resto del libro.

CREAR PROCEDIMIENTOS MEDIANTE EL ENVÍO DE MENSAJES.

Abre el proyecto **Flowers2.sb2**, que contiene la nueva versión de la actividad guiada 7 resuelta previamente. De nuevo, el escenario envía el mensaje "draw" cuando detecta un clic del ratón. Pero esta vez, el programa solo usa un objeto en lugar de cinco. El único objeto de este proyecto, llamado "flower", tiene cinco disfraces distintos (leaf1, leaf2, … , leaf5). El objeto "flower" llama a un procedimiento para dibujar una flor con cada disfraz. Como solo tenemos un objeto, solo necesitamos una copia del código para dibujar una flor (y no los cinco programas duplicados que teníamos en la versión previa). Esto hace que el programa sea más pequeño y que el código sea más fácil de entender.

Cuando el objeto recibe el mensaje "draw", ejecuta el programa "draw" mostrado en la figura:

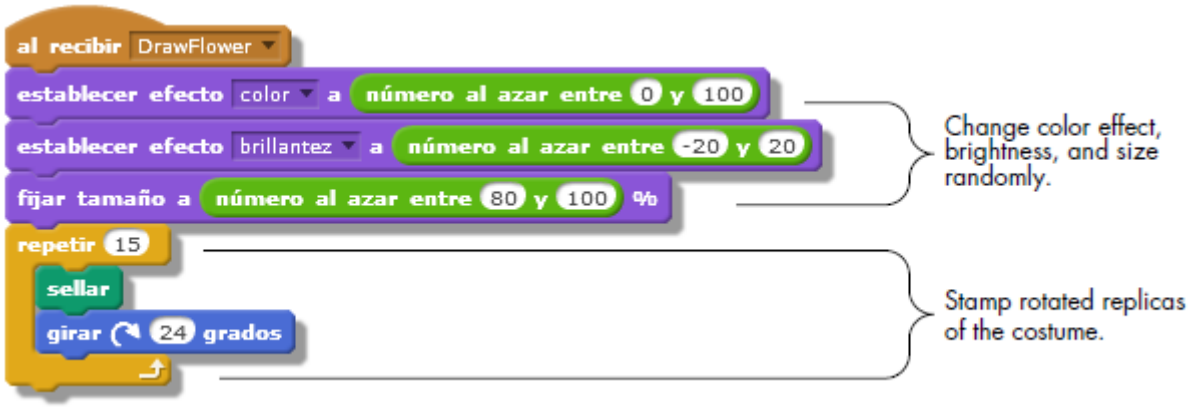


El programa fija el valor de la coordenada x y selecciona el primer disfraz para dibujar la primera flor. Después, comienza un bucle para dibujar las cinco flores. A cada pasada, el objeto fija el valor de la coordenada y de la flor, y llama al procedimiento "drawFlower" enviándose a sí mismo el mensaje "drawFlower". Esta llamada interrumpe la ejecución del programa hasta que se completa el procedimiento "drawFlower". A continuación, el programa "draw" retoma su ejecución, ajustando la coordenada x y cambiando el disfraz para preparar el dibujo de la siguiente flor.

El procedimiento "drawFlower" se muestra en la siguiente figura. Este programa ajusta un valor aleatorio a los efectos "color" y "brillantez", y al tamaño del objeto. A continuación, estampa versiones rotadas del disfraz actual para dibujar una flor.

Notar que mientras la primera versión del programa contenía cinco objetos y cinco programas casi idénticos, esta segunda versión hace lo mismo usando un único objeto (con cinco disfraces) que llama a un

procedimiento para dibujar las cinco flores. Por supuesto, esta nueva versión es más corta y más fácil de entender.

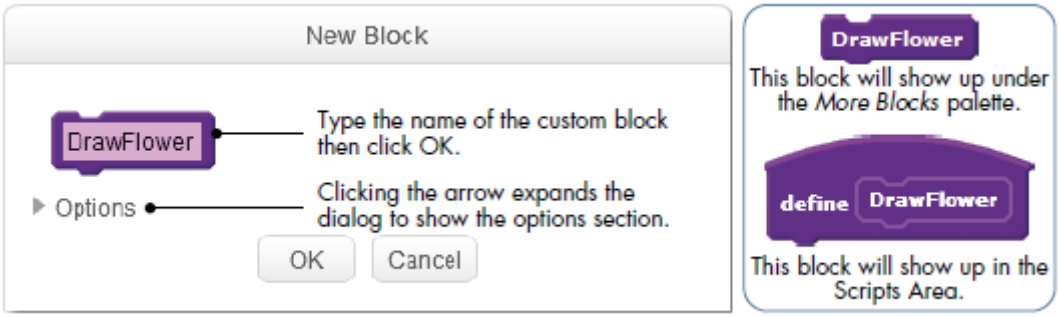


CONSTRUIR NUESTROS PROPIOS BLOQUES.

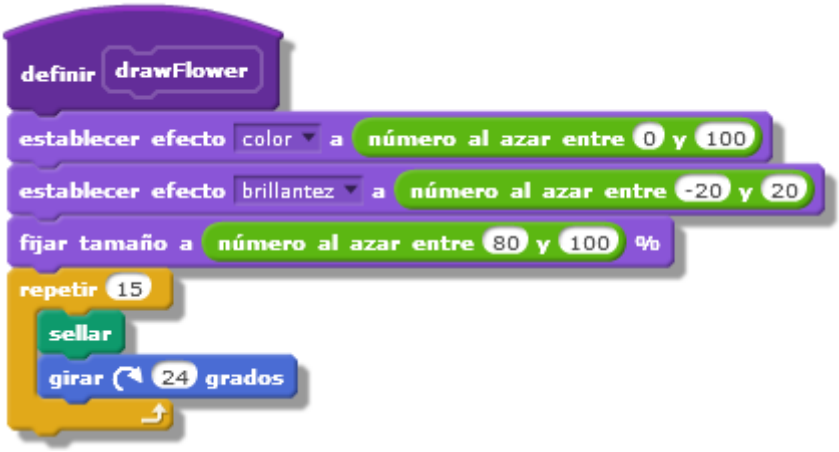
En la nueva versión de Scratch podemos crear nuestros propios bloques para implementar procedimientos. Después de que hayamos creado un nuevo bloque, éste aparecerá en la categoría "más bloques", y podremos usarlo como cualquier otro bloque de Scratch.

Para aprender a crear nuevos bloques, vamos a modificar el programa **Flowers2.sb2** que discutimos en la sección previa, usando un bloque customizado para crear el procedimiento "drawFlower":

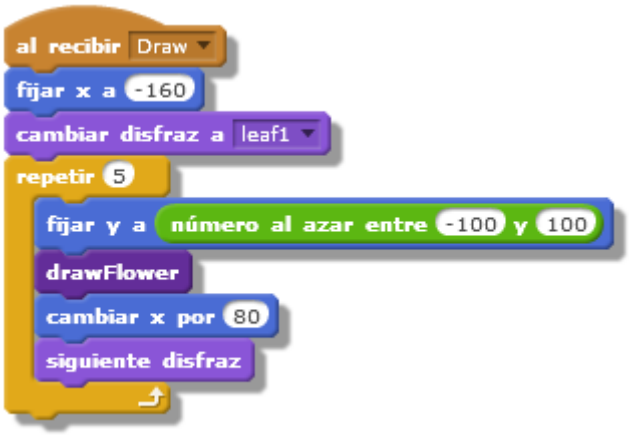
1) Abre el archivo **Flowers2.sb2**. En el menú, selecciona "archivo" → "guardar como", y guarda una copia del proyecto con el nombre **Flower3.sb2**. A continuación, clica en la miniatura del objeto "flower" para seleccionarlo. Ahora, ve a la categoría "más bloques" y clica en la opción "crear un bloque". Aparecerá la ventana de diálogo mostrada en la figura. Escribe "drawFlower" como nombre del bloque y clica en OK. Verás que ahora hay un bloque llamado "drawFlower" en la categoría de "más bloques", y en el área de programas aparece un bloque llamado "definir (drawFlower)".



2) Separa el código conectado al bloque "al recibir (drawFlower)" y conéctalo al bloque "definir (drawFlower)", como muestra la figura. Con esto hemos creado el procedimiento "drawFlower", esta vez implementado en la forma de un bloque customizado. Borra el bloque "al recibir (drawFlower)".



3) Ahora que ya hemos creado el procedimiento "drawFlower", solo nos queda llamarlo desde el programa que gestiona la recepción del mensaje "draw". Modifica dicho programa como muestra la figura, donde simplemente hemos reemplazado el bloque "enviar (drawFlower) y esperar" por nuestro nuevo bloque customizado "drawFlower".



El programa ya está terminado. Pruébalo para verificar que funciona igual que antes.

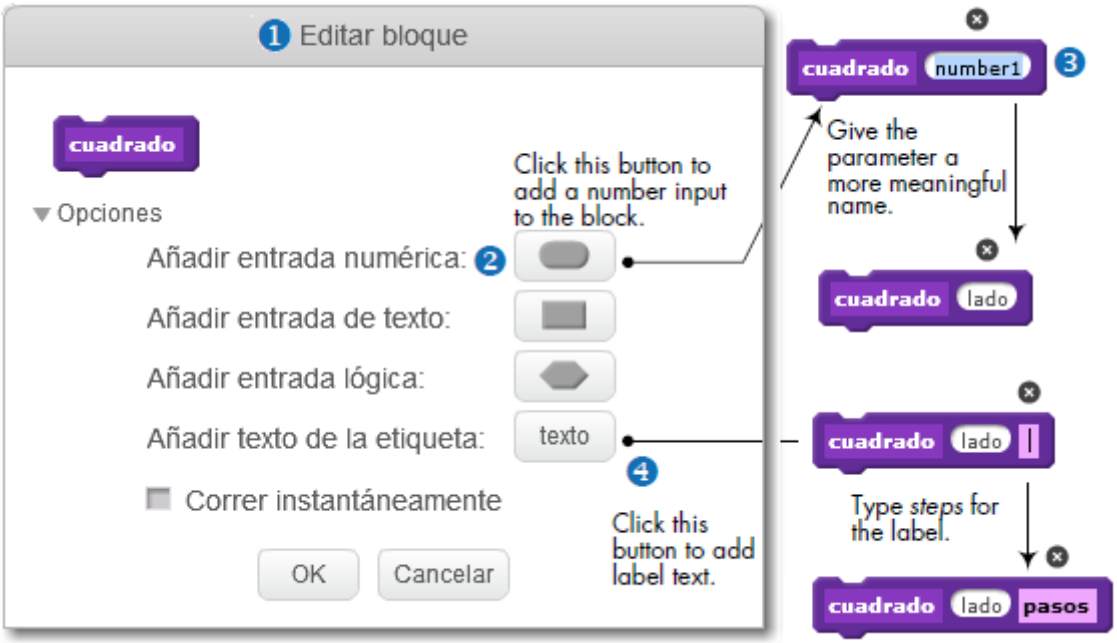
NOTA: Una vez creado un bloque customizado, Scratch nos ofrece una opción para acelerar la ejecución del programa. Para ello, clica en el bloque "definir (drawFlower)" en el área de programas, y en el menú emergente, selecciona "editar". A continuación, despliega las opciones, elige "correr instantáneamente", y clica en OK. Ahora clica con el ratón en el escenario para ejecutar el programa, y observa lo que ocurre. Verás que las flores se construyen casi instantáneamente, en vez de hacerlo progresivamente como ocurría hasta ahora. La razón es la siguiente: el procedimiento "drawFlower" contiene muchos bloques que cambian la apariencia del objeto, como "establecer efecto (color)", "establecer efecto (brillantez)", "fijar tamaño", y "sellar". Tras ejecutar cada uno de estos bloques, Scratch suele hacer una pausa para refrescar (redibujar) la pantalla. Por esta razón podíamos ver el proceso de dibujado de las flores antes de habilitar la opción "correr instantáneamente". Pero después de habilitarla, los bloques se ejecutan sin hacer una pausa para refrescar la pantalla, lo que permite que el procedimiento se ejecute más rápido.

PASARLE PARÁMETROS A UN BLOQUE CUSTOMIZADO.

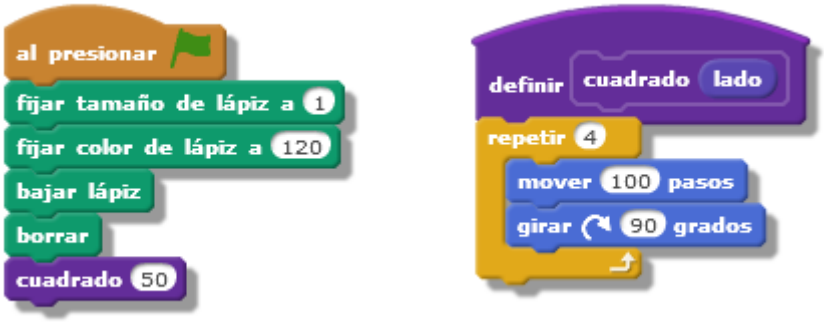
Ahora que ya sabemos crear bloques customizados, vamos a aprender cómo crear bloques que acepten datos de entrada para funcionar. Vamos a comenzar creando un bloque customizado llamado "cuadrado", cuya misión es dibujar un cuadrado de longitud 100 píxeles (ver figura).

Notar que el procedimiento "cuadrado" tiene una utilidad muy limitada, ya que la longitud del lado está fija a 100 píxeles. ¿Y si queremos dibujar cuadrados con un lado de longitud, digamos, 70, 50, 200, etc.? ¿Hacemos un procedimiento distinto para cada caso: cuadrado70, cuadrado50, cuadrado200? Evidentemente, la mejor solución es tener un procedimiento al cual le podamos pasar la longitud del lado.

De hecho, hemos estado usando esta técnica desde el capítulo 1. Por ejemplo, el bloque "mover ( ) pasos" incorpora un espacio que nos permite insertar un número para especificar cuántos pasos queremos movernos. Por tanto, lo que nosotros debemos hacer es añadir un hueco en nuestro bloque "cuadrado", que le permita al usuario especificar la longitud del lado introduciendo un *parámetro*. La figura ilustra cómo modificar el bloque "cuadrado" para conseguirlo:



(1) Primero, clicamos en el bloque "cuadrado" en la categoría "más bloques" (o en el bloque "definir (cuadrado)" en el área de programas), y seleccionamos "editar". En la ventana emergente expandimos las opciones. (2) Queremos que nuestro bloque acepte un parámetro numérico para poder especificar la longitud del lado, así que clicamos en "añadir entrada numérica"; con ello, añadimos al bloque un espacio para poder pasarle la longitud del lado. (3) Ahora, cambiamos el nombre por defecto de la entrada numérica, de number1 a un nombre con significado, como "lado". Con ello, el programa queda así:

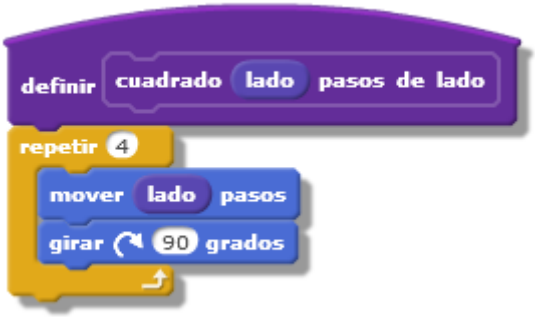


Técnicamente, esto es todo lo que necesitamos. Ahora bien, ¿cómo sabe el usuario qué significa el dato numérico que debe pasarle al bloque "cuadrado"? ¿Se trata del área del cuadrado, su perímetro, su diagonal, su lado, o qué? Por ejemplo, imagina que el bloque "deslizar" se hubiese diseñado de esta forma:

¿Cómo sabría el usuario para qué sirven los datos numéricos introducidos en el primer, segundo, y tercer huecos? Para evitar este problema, los diseñadores de Scratch añadieron unas etiquetas de texto junto a cada hueco que explican para qué sirve el dato introducido en ese hueco:

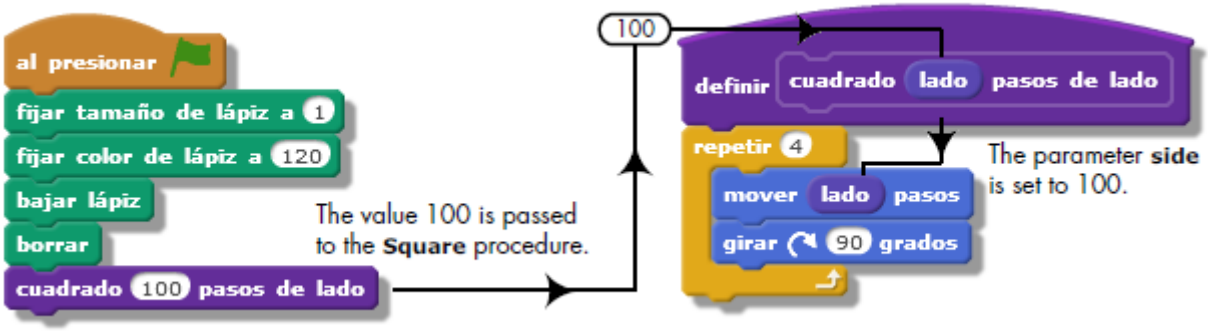
Nosotros debemos hacer algo parecido para nuestro bloque "cuadrado", esto es, debemos añadir una etiqueta de texto que describa el uso o significado del parámetro "lado". Para ello, clicamos en "añadir texto de la etiqueta", para incluir una etiqueta de texto al lado del parámetro "lado" (4). Tecleamos "pasos de lado" y clicamos en OK. El programa quedará como:

Si ahora examinamos la definición del procedimiento "cuadrado" en el área de programas, veremos un pequeño bloque llamado "lado" dentro del encabezado del procedimiento, que le dice al procedimiento que debe leer el dato numérico introducido por el usuario en el bloque "cuadrado". Pero el bloque "mover" dentro del procedimiento aún tiene un valor fijo de 100. Para reemplazar ese valor fijo de 100 por el valor introducido por el usuario, todo lo que debemos hacer es arrastrar el bloque "lado" desde el encabezado hasta el hueco de datos del bloque "mover":



El bloque "lado" que aparece dentro del encabezado del procedimiento "cuadrado" se denomina **parámetro**. Podemos pensar en un parámetro como en un marcador o localizador: Nosotros queríamos que el procedimiento "cuadrado" pudiese dibujar cuadrados con un lado cualquiera sin tener que insertar un dato fijo dentro de nuestro procedimiento, sino usando un parámetro general llamado "lado". Gracias a ello, los usuarios pueden especificar la longitud del lado (en pasos) cuando usan el bloque "cuadrado ( ) pasos del lado" al llamar a este procedimiento.

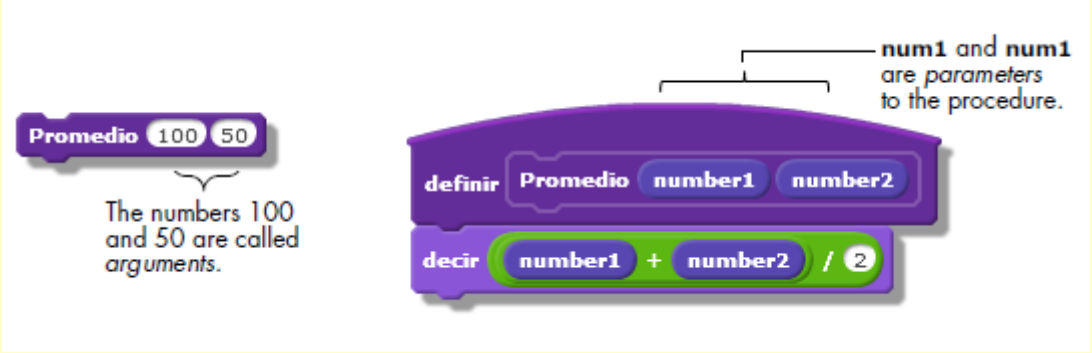
Por ejemplo, en el programa de la figura el usuario le pasa el número 100 (un *argumento*) al procedimiento "cuadrado". Cuando se ejecuta el procedimiento "cuadrado", su parámetro "lado" se ajusta a 100, y este valor se usa para reemplazar todas las ocurrencias del bloque "lado" dentro del procedimiento.



Podemos ampliar el procedimiento "cuadrado" para hacer que acepte el color del cuadrado como segundo argumento (parámetro "num_color", de número de color), para ajustar el grosor del cuadrado como tercer argumento (parámetro "tam_lapiz", de tamaño lápiz), etc.

Parámetros vs. argumentos.

Aunque muchos programadores usan los términos *parámetro* y *argumento* con el mismo significado, los dos términos son, de hecho, distintos. Observa el procedimiento "Promedio" mostrado en la figura más abajo:

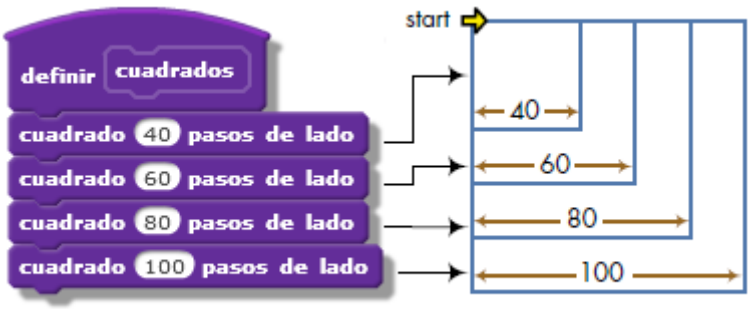


En su definición, este procedimiento tiene dos parámetros, number1 y number2. Un **parámetro** define una entrada a un procedimiento. A este procedimiento lo llamamos mediante el bloque "Promedio" mostrado a la izquierda, pasándole unos valores numéricos o expresiones dentro de los dos huecos disponibles. Los valores 100 y 50 son los **argumentos** del procedimiento. Por supuesto, el número de argumentos en la llamada al procedimiento debe coincidir con el número de parámetros en la definición del procedimiento. Cuando llamamos al procedimiento "Promedio", los parámetros number1 y number2 reciben los valores 100 y 50, respectivamente, debido a su posición en la llamada y en la definición del procedimiento.

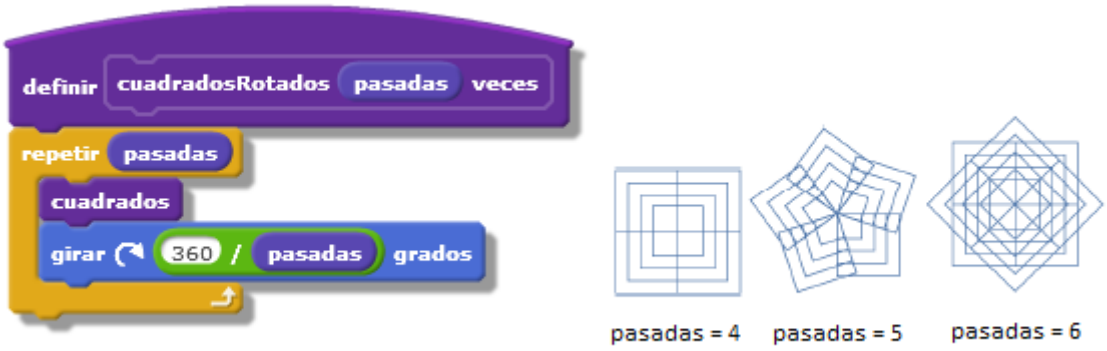
PROCEDIMIENTOS ANIDADOS.

Puede que nos estemos preguntando si un procedimiento puede llamar a otro procedimiento. La respuesta es afirmativa. Hemos indicado antes que los procedimientos deben construirse para realizar una tarea bien definida. Si queremos ejecutar múltiples tareas, es perfectamente legítimo hacer que un procedimiento llame a otro dentro de su definición.

Para ver cómo funciona este método, comencemos con el procedimiento "cuadrado" que construimos en la sección previa. Ahora hemos creado un procedimiento llamado "cuadrados" (con "s" al final) que dibuja cuatro cuadrados, cada uno más grande que el anterior (ver figura). Esto lo hemos hecho llamando al procedimiento "cuadrado" 4 veces en la definición del procedimiento "cuadrados", donde cada llamada usa un argumento diferente.

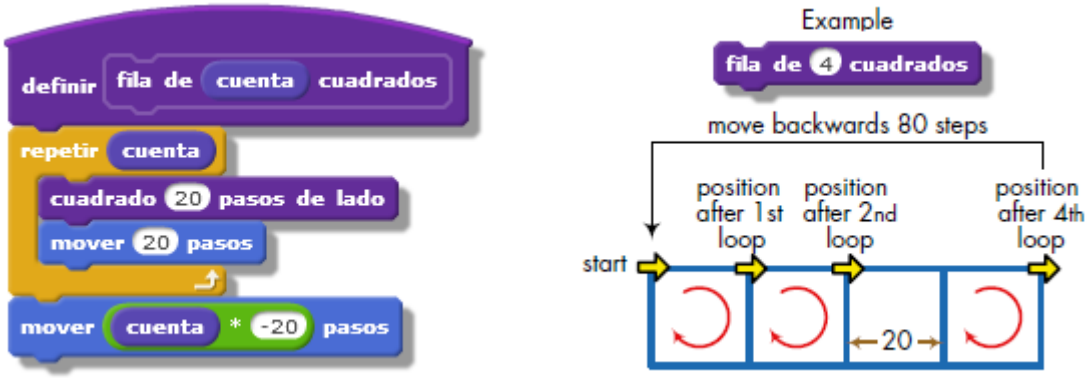


Ahora, vamos a usar el procedimiento "cuadrados" dentro de otro procedimiento, llamado "cuadradosRotados", para crear algunos patrones interesantes. Este procedimiento llama al procedimiento "cuadrados" varias veces, girando las formas dibujadas en un cierto ángulo en cada llamada:

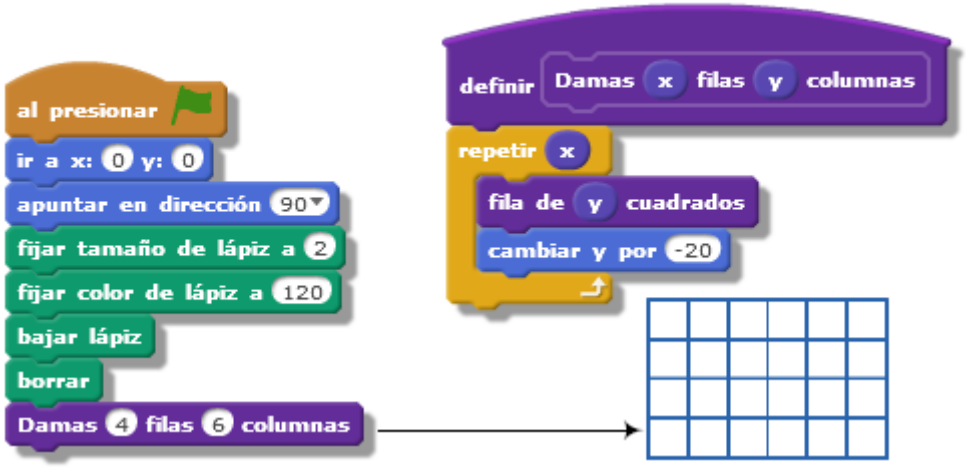


Notar que, en este ejemplo, el parámetro "pasadas" se usa dos veces: una para determinar el número de repeticiones del bucle, y otra para calcular el ángulo de giro después de llamar al procedimiento "cuadrados".

Veamos otro ejemplo que demuestra la potencia de los procedimientos anidados. Comenzaremos con el procedimiento "cuadrado" y terminaremos con un damero. Para ello, creamos un procedimiento llamado "fila" que dibuje una fila de cuadrados (de lado fijo igual a 20, por simplicidad). El número de cuadrados a dibujar se especifica mediante un parámetro llamado "cuenta":



(La última instrucción simplemente lleva al objeto dibujante a su posición inicial, dejándolo preparado para dibujar una nueva fila). Ahora, para dibujar otra fila de cuadrados bajo la que ya hemos dibujado, basta con mover el objeto 20 pasos hacia abajo y volver a llamar al procedimiento "fila". Repitiendo este proceso podemos dibujar tantas filas como queramos. El procedimiento "Damas" hace precisamente eso:



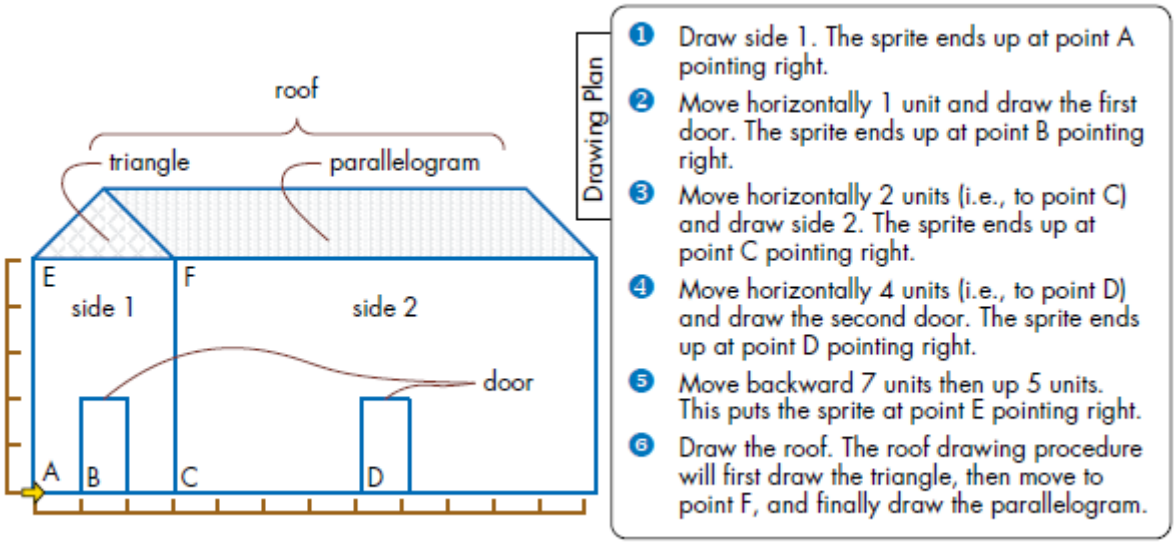
Este procedimiento tiene dos parámetros: el número x de filas y el número y de columnas del damero que queremos construir. Después de dibujar cada fila, el procedimiento mueve el objeto 20 pasos hacia abajo, para poder dibujar la siguiente fila de cuadrados.

4.3. TRABAJAR CON PROCEDIMIENTOS.

Ahora ya sabemos la gran utilidad que supone dividir nuestro programa en partes más pequeñas, para resolver cada una de ellas por separado. En esta sección vamos a discutir cómo realizar esta división. Primero exploraremos la técnica **arriba – abajo**, que consiste en dividir programas grandes en piezas modulares con una estructura lógica evidente. Después presentaremos la técnica **abajo – arriba**, consistente en construir programas complejos combinando procedimientos ya existentes.

DIVIDIR PROGRAMAS EN PROCEDIMIENTOS.

Para mostrar la forma de aplicar la técnica arriba – abajo, consideremos cómo podríamos dibujar una casa similar a la mostrada en la figura:



Algunas posibilidades serían las siguientes:

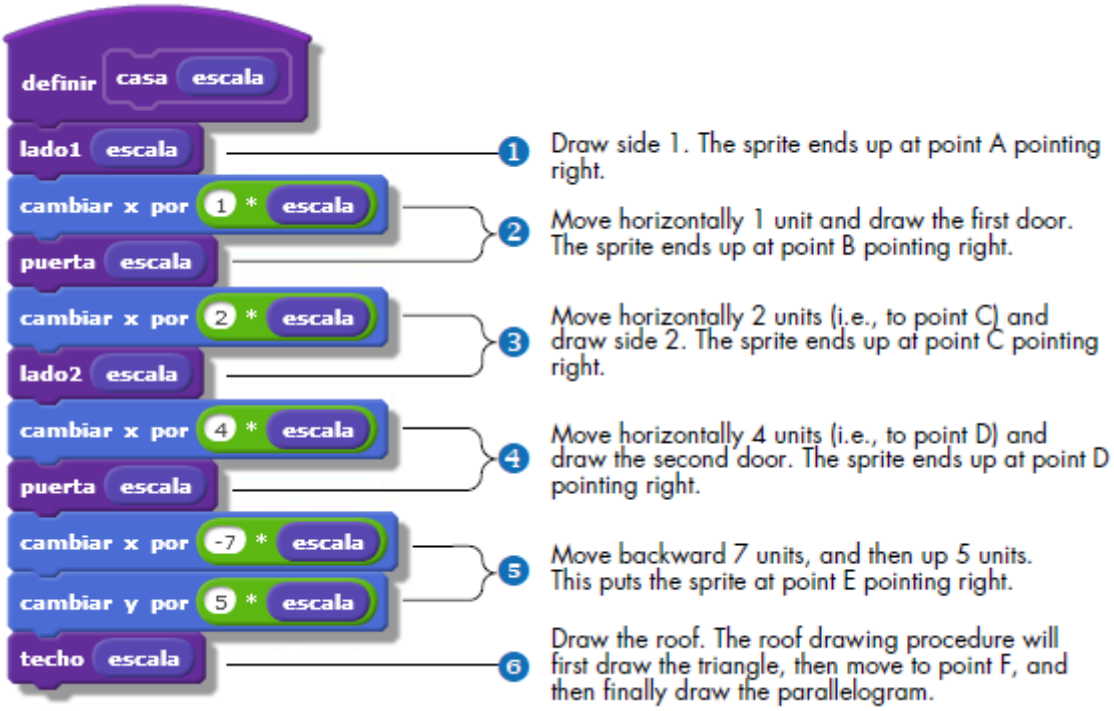
- Imaginar que la casa está hecha de líneas rectas, y dibujar cada línea. Dibujar todas las líneas una a una sería una tarea compleja.
- Imaginar que la casa está formada por 6 figuras: lado1 (side1), lado2 (side2), dos puertas, un triángulo, y un paralelogramo. Dibujar una a una cada forma también sería una tarea compleja.
- Como las dos puertas son iguales, podemos definir un procedimiento para dibujar la puerta y llamarlo 2 veces.
- Podemos imaginar que el triángulo y el paralelogramo constituyen una sola unidad, el techo. Podemos crear un procedimiento para dibujar el techo.
- Podemos imaginar que el lado side1 y su puerta constituyen una sola unidad, el lado frontal. Una tarea del programa sería dibujar el lado frontal.
- …

Hay muchas posibilidades más, pero esto es suficiente para hacernos una idea. El método consiste en agrupar tareas en piezas pequeñas con las que podamos tratar, y centrarnos en cada tarea por separado. Si nos encontramos con piezas similares, debemos intentar buscar una solución común y aplicarla a todas esas piezas.

Con todo esto en mente, nuestro plan para dibujar la casa está resumido en la figura: comenzamos con el objeto apuntando hacia la derecha en el punto A. A continuación, creamos un programa que realice los pasos

indicados: Escribimos un procedimiento ("lado1") para dibujar el lado izquierdo de la casa (paso 1). También escribimos tres procedimientos ("puerta", "lado2", y "techo") para dibujar las dos puertas, el lado derecho de la casa, y el techo (pasos 2, 3, 5, y 6), y conectamos estos procedimientos con los comandos de movimiento apropiados.

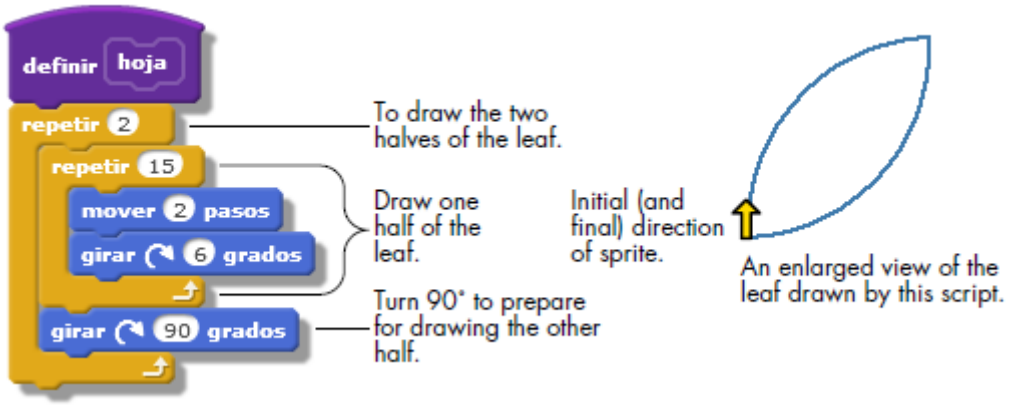
El procedimiento "casa" está mostrado en la figura. Este procedimiento recibe un único parámetro ("escala") que especifica la unidad de longitud (esto es, el factor de escalado) para dibujar la casa. Notar que el procedimiento "puerta" se usa dos veces. Notar también que el procedimiento "techo" se usa para dibujar todo el techo, y que puede contener diferentes sub-procedimientos para dibujar los distintos elementos del techo.



La figura también muestra los procedimientos individuales para dibujar los distintos componentes de la casa. Se trata de procedimientos que dibujan figuras geométricas simples, tal y como hicimos en el capítulo 2. Los procedimientos "lado1", "puerta", y "lado2" dibujan unos rectángulos de 3×5, 1×2, y 9×5 (escalados por el factor de escala), respectivamente. El procedimiento "techo" incorpora dos sub-procedimientos ("triángulo" y "paralelogramo") para dibujar las dos partes del techo. Observar que el factor "escala" se ha usado de forma consistente en todos los procedimientos. Este parámetro define el tamaño de la unidad horizontal y vertical en la figura inicial, y permite dibujar casas más grandes o más pequeñas llamado al procedimiento "casa" con un argumento diferente.

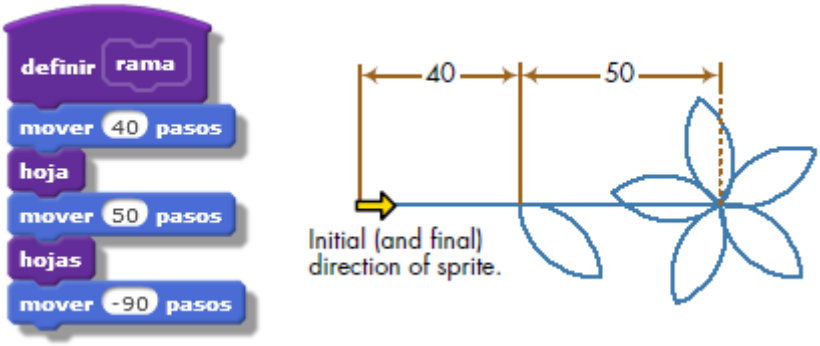
CONSTRUIR PROGRAMAS A PARTIR DE PROCEDIMIENTOS.

Para mostrar la técnica abajo – arriba, comenzaremos con un procedimiento muy sencillo, llamado "hoja", que dibuja una única hoja de una planta. El bucle "repetir (2)" dibuja las dos mitades de la hoja, mientras que el bucle "repetir (15)" sirve para dibujar cada mitad, como una serie de 15 cortos segmentos de línea con un giro de $6°$ entre uno y otro.



A continuación, el procedimiento "hojas" (con "s" final) repite la llamada al procedimiento "hoja" 5 veces, para dibujar 5 versiones giradas (en $72°$) de la misma hoja. (Como siempre, notar que $72° \times 5 = 360°$).

Ahora podemos usar los procedimientos "hoja" y "hojas" para dibujar una rama. El procedimiento "rama" mostrado en la figura mueve al objeto dibujante hacia adelante 40 pasos, dibuja una sola hoja (llamando al procedimiento "hoja"), se mueve 50 pasos más hacia adelante, dibuja 5 hojas (llamando al procedimiento "hojas"), y finalmente retorna a su posición inicial.



Por último, el procedimiento "flor" usa un bucle para llamar al procedimiento "rama" 6 veces y dibujar seis versiones rotadas (en $60°$) de la misma rama:

Podríamos seguir dibujando formas más complicadas reutilizando procedimientos más sencillos, pero la idea ya debería estar clara. Si así lo quisiéramos, ahora podríamos crear un procedimiento que dibujase todo un árbol de flores, y luego otro que dibujase un jardín lleno de árboles de flores, etc.

En definitiva, la idea que debemos extraer del ejemplo es que, sea cual sea la complejidad del problema a resolver, siempre podemos construir una solución juntando una serie de piezas más pequeñas y manejables. Trabajando de esta forma, comenzaremos con unos procedimientos muy cortos que resuelven problemas muy sencillos, y los usaremos para crear procedimientos más y más sofisticados.

EJERCICIO 18. TU NOMBRE.

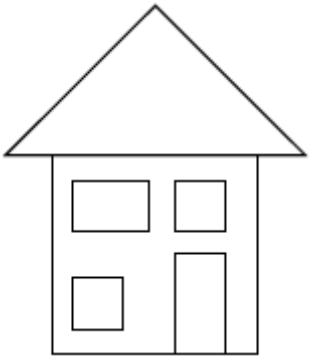
Escribe diferentes procedimientos para dibujar cada letra de tu nombre. Nombra a cada procedimiento con la letra que dibuja. Después, escribe un programa que llame a todos estos procedimientos para dibujar tu nombre sobre el escenario. Guarda el archivo como **Ejercicio18.sb2**.

EJERCICIO 19. PROCEDIMIENTOS FÍSICOS Y MATEMÁTICOS.

a) Escribe un procedimiento que convierta una temperatura de grados Celsius (grados centígrados) a kelvin. Para obtener la relación de conversión puedes usar el hecho de que los $-273\ °C$ se corresponden con los $0\ K$ (temperatura llamada "cero absoluto").
b) Escribe un procedimiento que convierta unidades de volumen (m^3) a unidades de capacidad ($litros$). La correspondencia es $1\ dm^3 = 1\ l$.
c) Escribe un procedimiento que convierta de km/h a m/s, la unidad SI para la velocidad.
d) Escribe un procedimiento para computar el área de un círculo ($A = \pi r^2$) dado su radio r. Usa $\pi = 3{,}1416$.

EJERCICIO 20. *CASA*.

Escribe un programa para dibujar la casa mostrada en la figura. Comienza escribiendo pequeños procedimientos para dibujar las distintas partes de la casa (puerta, techo, ventanas, etc.). Después, combina todos estos procedimientos para crear la casa completa. Guarda el archivo como **Ejercicio20.sb2**.

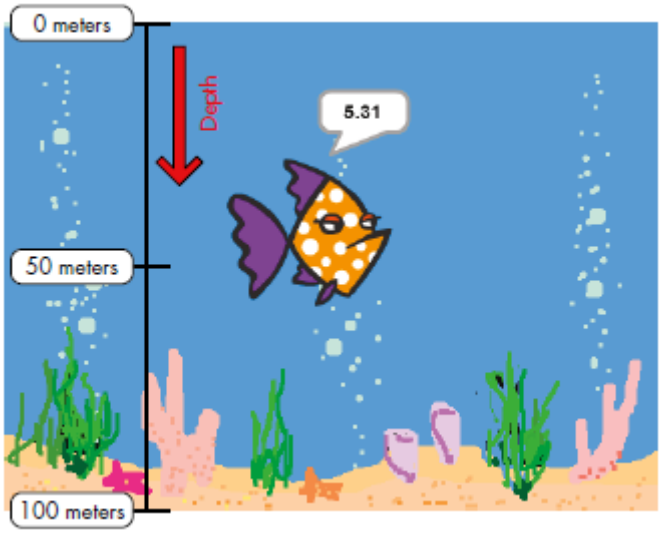


EJERCICIO 21. COPO DE NIEVE.

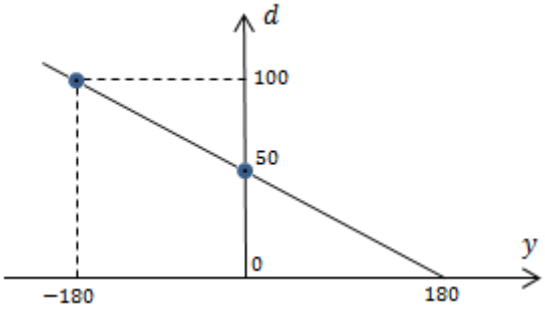
Escribe un programa que dibuje por pantalla el copo de nieve mostrado en la figura. PISTA: Crea un procedimiento para dibujar cada una de las puntas del copo, y otro procedimiento para crear el copo (el cual llamará seis veces al procedimiento de las puntas). El programa principal ajustará las características del lápiz, y llamará al procedimiento copo una sola vez. Guarda el archivo como **Ejercicio21.sb2**.

EJERCICIO 22. PRESIÓN BAJO EL AGUA.

En este ejercicio simularás la presión que experimenta un pez bajo el agua. Asume que la presión P sentida por el pez (en atmósferas) está relacionada con su profundidad d (en metros por debajo de la superficie) por la fórmula $P = 0{,}1d + 1$. El archivo **Ejercicio22_sinCodigo** contiene los objetos y los fondos necesarios. Crea un programa mediante el cual el pez siempre dice qué presión siente mientras nada a lo largo del escenario. Para nadar, el programa principal (que empieza al clicar sobre la bandera) hace que el pez apunte en la dirección $45°$, y entonces, se va moviendo a lo largo del acuario, rebotando contra los bordes.



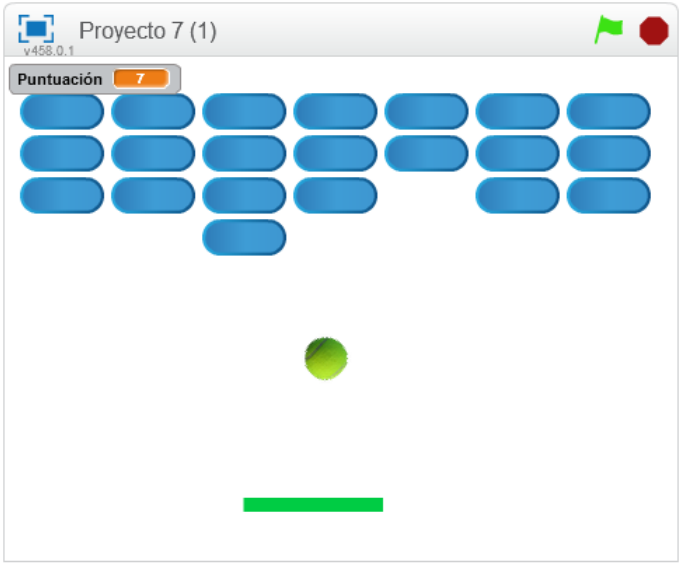
NOTA: ¿Cómo sabemos a qué profundidad bajo la superficie está el pez? Recuerda que la coordenada y en la que se encuentra el pez varía desde 180 (arriba del escenario) hasta -180 (abajo del escenario). Por su parte, de la figura vemos que la posición $y = 180$ se corresponde con una profundidad $d = 0\,m$, la posición $y = 0$ (el centro del escenario) con una profundidad $d = 50\,m$, y la posición $y = -180$ con una profundidad $d = 100\,m$. Si representamos la profundidad d como una función de la coordenada y del pez podemos ver que la relación entre ambas es lineal (función afín), y que la ecuación de dicha recta es $d = -\frac{50}{180}y + 50$.



4.4. PROYECTOS SCRATCH.

PROYECTO 7. ARKANOID.

El Arkanoid es un videojuego clásico en el que una raqueta en la parte inferior del escenario hace rebotar una pelota que rompe unos bloques situados en la parte alta del escenario: https://www.youtube.com/watch?v=44rceRqY8_k. El objetivo de este proyecto es hacer una versión de este famoso videojuego:



1) Construye una raqueta que se mueva de derecha a izquierda sobre el escenario.

Borra el objeto gato, y crea el objeto raqueta (dibuja un nuevo objeto con la forma de un delgado rectángulo verde). El centro del objeto estará en el centro del rectángulo. Cuando más corta dibujes la raqueta, más difícil será el juego.

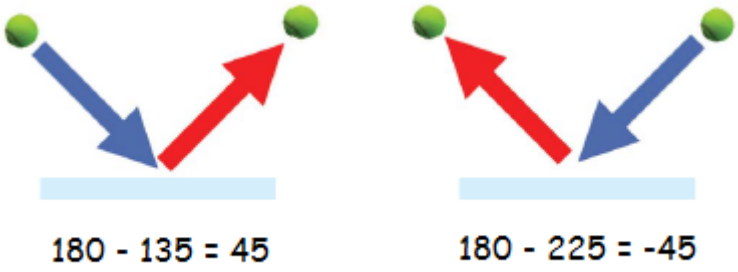
A continuación, programa la raqueta para que, al clicar en la bandera verde, comience en la posición $(x, y) = (130, -140)$. A continuación, el programa hace que la raqueta siempre acuda a la posición con la coordenada x marcada por la ubicación del puntero del ratón, y con la coordenada y fija a -140.

2) Construye una pelota que rebote contra las paredes.

Añade a la lista de objetos la pelota "tennis ball" de la biblioteca de objetos (categoría "deportes"). Escribe un programa que, al clicar en la bandera verde, haga que la pelota comience en la posición $(x, y) = (0, 0)$. A continuación, apunta en la dirección $135°$ (hacia abajo y hacia la derecha). Transcurrido 1 segundo, la raqueta siempre se moverá en pasos de 10, rebotando si llega al borde del escenario.

3) Haz que la pelota rebote contra la raqueta.

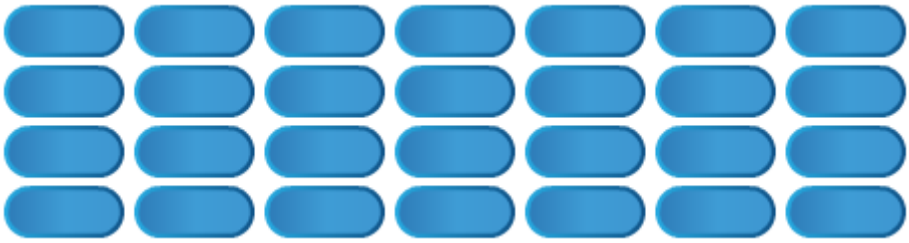
Completa el programa de la pelota: si en su movimiento la pelota toca a la raqueta, la primera llama al procedimiento "rebotar". Este procedimiento no tiene parámetros, y simplemente hace que la pelota rebote hacia arriba tras tocar la raqueta. Para determinar la dirección en la que rebotará la pelota, pensemos lo siguiente (ver figura): Si la pelota viaja hacia abajo y hacia la derecha (dirección $135°$), ha de rebotar hacia arriba y hacia la derecha (dirección $45°$). Notar que se cumple que $180° - 135° = 45°$. Por otro lado, si la pelota viajaba hacia abajo y hacia la izquierda (dirección $225°$), ha de rebotar hacia arriba y hacia la izquierda (dirección $-45°$). Notar que, en este caso, también se cumple que $180° - 225° = -45°$.



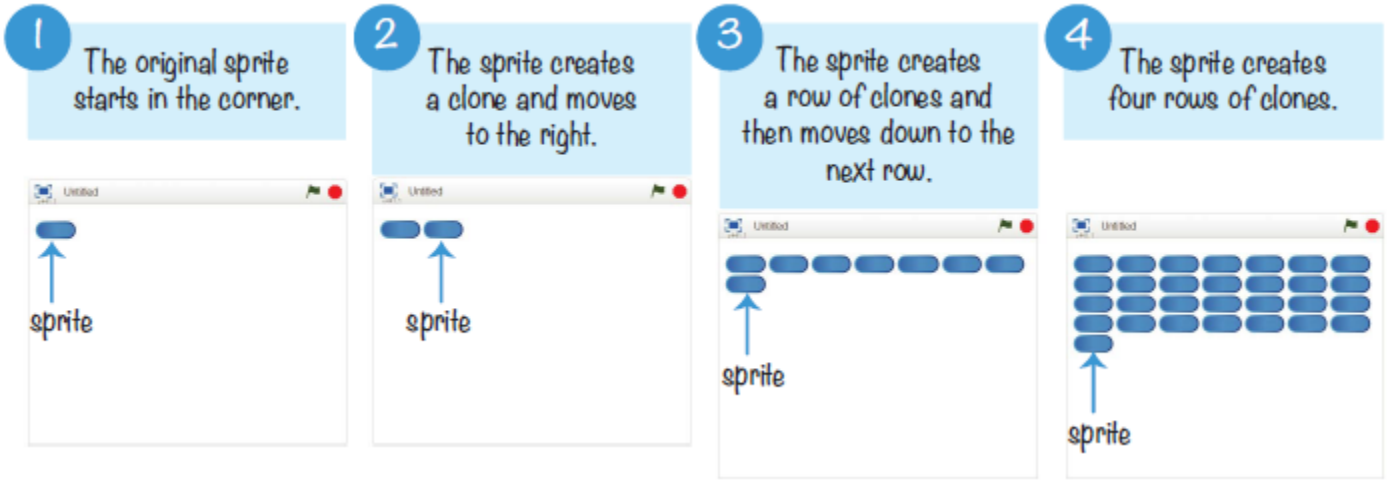
Ahora, la dirección actual en la que viaja la pelota está almacenada en el bloque "dirección" de la categoría "movimiento". Por lo tanto, la dirección en la que debe pasar a apuntar la pelota al rebotar siempre es:

4) Construye el objeto ladrillo y clónalo.

Añade al proyecto el objeto "button2" de la biblioteca de Scratch. A continuación, crea una nueva variable llamada "puntuación", clicando en la opción "crear una variable" de la categoría "datos". Esta variable estará disponible para todos los objetos.



El programa del ladrillo comienza al clicar en la bandera verde. El programa esconde el objeto, fija el valor de la variable "puntuación" a cero, ajusta su tamaño al 50%, y lo ubica en la posición $(x, y) = (-200, 140)$. A continuación, el programa va a crear un "muro de ladrillos" en la parte superior, de 7 ladrillos de ancho x 4 ladrillos de alto (ver figura). Ello lo conseguiremos con un bucle doble anidado: Cada pasada de un bucle "repetir (7)" creará un clon del ladrillo, y lo ubicará a una distancia horizontal de 65 pasos del clon previo. Con este bucle creamos los 7 bloques de cada fila. A continuación, cada pasada de un bucle repetir (4) creará una nueva fila de 7 bloques, volverá a la posición horizontal inicial $x = -200$, y bajará 30 pasos para crear la siguiente fila de 7 bloques.



5) Haz que la pelota rebote en los ladrillos.

Completa el código del ladrillo: Para cada clon creado (bloque "al comenzar como un clon"), el programa muestra el clon, y de forma indefinida comprueba si está tocando a la pelota. De ser así, el programa cambia el valor de la variable "puntuación" en 1 unidad, y borra ese clon.

Completa el programa de la pelota: si la pelota toca el ladrillo, debe rebotar contra él. Para ello, llama al procedimiento "rebotar" que ya definimos previamente.

6) Consigue que la partida finalice cuando la raqueta no consigue devolver la pelota.

Crea un nuevo programa para la pelota: Al clicar en la bandea verde, la pelota comprueba de forma indefinida si llega a una posición vertical por debajo de la raqueta, $posición\ en\ y < -140$. (La raqueta ha fallado el golpeo de la pelota). En ese caso, la pelota envía el mensaje "fin de partida", que hará que aparezca el objeto "game over" sobre el escenario.

Crea el objeto "game over": Para ello, acude al editor gráfico de Scratch, y utiliza la herramienta de texto para escribir en rojo el texto GAME OVER sobre un fondo transparente.

Crea el código de control del objeto "game over": Este objeto necesita dos programas. (1) El primer programa empieza al clicar en la bandera verde, y simplemente centra al objeto en el escenario, posición $(x, y) = (0, 0)$, y lo esconde. (2) El segundo programa comienza cuando el objeto "game over" recibe el mensaje "fin de partida". En ese caso, el objeto se muestra, y para la partida deteniendo al resto de objetos.

7) Haz que la partida finalice cuando hayas roto todos los ladrillos.

Crea el objeto "ganaste", de forma similar a como creaste el objeto "game over", pero ahora con letra de color verde.

Crea el programa de control para el objeto "ganaste": el programa empieza al clicar en la bandera verde, y comienza centrando al objeto en el escenario, y escondiéndolo. A continuación, fija la puntuación a cero, y queda a la espera hasta que la variable "puntuación" se haga igual a 28 (porque hay $7 \times 4 = 28$ ladrillos en total), bloque "esperar hasta que ( )" de la categoría de "control". Cuando la puntuación se hace igual a 28, el programa muestra el objeto, y finalizamos la partida deteniendo todos los objetos.

8) Guarda el proyecto como **Proyecto 7 (1).sb2**.

AMPLIACIÓN 1: Dibuja un fondo para tu escenario.

Bajo la miniatura del fondo blanco, clica en la opción "dibujar nuevo fondo". En el editor gráfico de Scratch, y con intercambiador de colores, elige dos colores púrpuras distintos (claro y oscuro). Usando los distintos gradientes de la herramienta de relleno, y clicando aleatoriamente sobre el lienzo del editor, crea un fondo como el mostrado en la figura.

AMPLIACIÓN 2: Añade música.

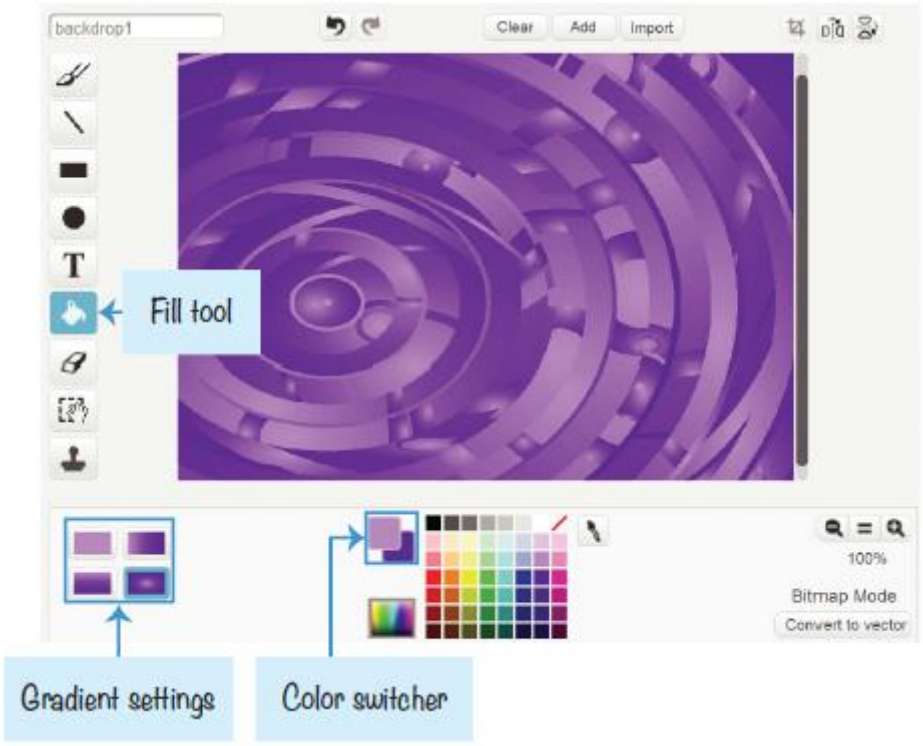
Añade al escenario el sonido "dance celebrate" de la biblioteca de Scratch. A continuación, añade un programa al escenario para que este sonido se esté reproduciendo ininterrumpidamente y sin cortes.

AMPLIACIÓN 3: Haz que la raqueta brille con diferentes colores al golpear la pelota.

Para ello, deberás completar programa a la raqueta de forma que, si toca la pelota, realice una serie de 10 cambios de color.

AMPLIACIÓN 4: Añade un sonido distinto cada vez que rompemos un ladrillo.

Añade al objeto ladrillo los sonidos "laser1" y "laser2" de la biblioteca de Scratch. Completa el programa del ladrillo para que, al desaparecer, toque aleatoriamente el sonido "laser1" o el sonido "laser2".



AMPLIACIÓN 5: Añade un sonido a la pelota cada vez que impacte contra la raqueta.

AMPLIACIÓN 6: Añade una estela tras la pelota.

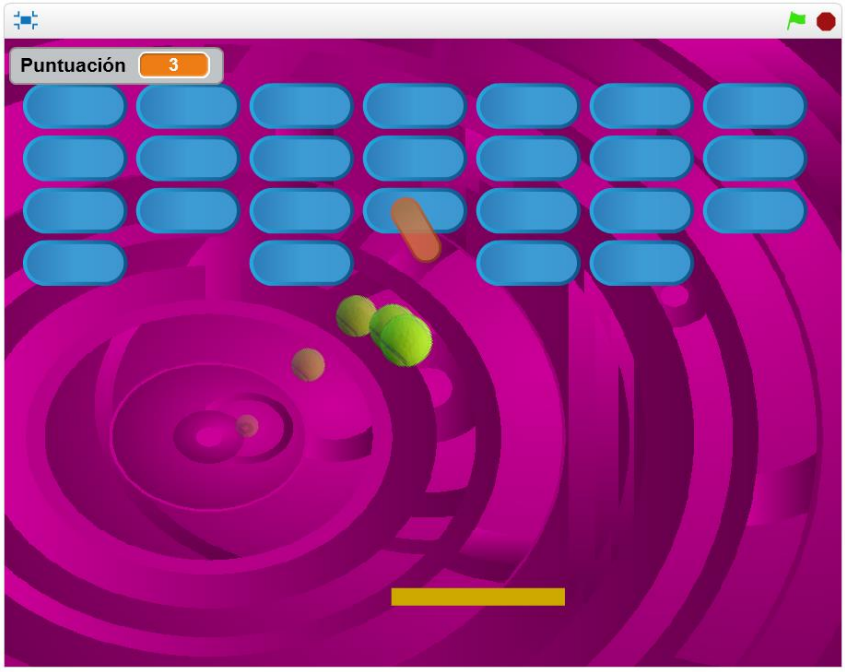
Vamos a añadir un rastro de clones tras la pelota conforme se mueve por el escenario, como la cola de un cometa. Pero no podemos usar clones del objeto pelota, porque rebotarían igual que la pelota maestra cuando ésta rebotase. En vez de ello, creamos un nuevo objeto pelota (volviendo a tomar el objeto "Tennis ball" de la biblioteca de Scratch), y éste será el objeto que clonaremos.

El código para este objeto "pelota2" consta de dos programas: (1) El primer programa comienza al clicar en la bandera verde, y simplemente oculta el objeto. (2) El segundo programa actúa sobre los clones de la pelota 2 (bloque "al comenzar como clon"), y comienza mostrado al clon creado. A continuación, acude a la posición de la pelota 1, y comienza un bucle de 15 pasadas que hace que el clon se encoja y se desvanezca, cambiando el tamaño del clon en -4 y el efecto "desvanecer" en 5. Tras el bucle, el programa borra el clon.

También tenemos añadir un programa al código de la pelota1, que se encargará de crear los clones de la pelota2. Este programa empieza al clicar en la bandera verde, y comienza enviando el objeto pelota1 al frente. A continuación, un bucle infinito espera 0,1 segundos y crea clones de la pelota 2 de forma indefinida.

AMPLIACIÓN 7: Añade una animación a la ruptura de ladrillos.

Modifica el código del objeto ladrillo para que los clones del ladrillo tengan una salida animada, en lugar de simplemente desaparecer al ser golpeados por la pelota. Para ello, y cuando la pelota les toque, añade un bucle de 10 repeticiones en el que, para cada pasada, se les aplique un cambio en el efecto "color" de 25, un cambio en el efecto "desvanecer" de 5, un cambio de tamaño de -4, un cambio en su coordenada y de 4, y un giro de $15°$ hacia al derecha.



AMPLIACIÓN 8:
Modifica el videojuego para hacerlo más difícil cuando ya hayas roto un buen número de ladrillos los ladrillos. Para ello, consigue que la raqueta se haga más pequeña y que la pelota se mueva más rápida cuando hayas roto, digamos, la mitad de los ladrillos.

5. VARIABLES.

A menudo necesitaremos crear programas que puedan leer y recordar valores, y tomar decisiones basadas en ciertas condiciones. Este capítulo se ocupa del primero de estos dos objetivos. La toma de decisiones será el objetivo del siguiente capítulo.

Ya nos habremos dado cuenta de que los programas pueden procesar y manipular diferentes tipos de datos. Estos datos pueden actuar como entradas (por ejemplo, el número 10 en el bloque "mover (10)", o el texto "hola!" en el bloque "decir (hola!)"), o como salidas (por ejemplo, en los bloques "número al azar" o "distancia a ( )"). En los programas más complejos, necesitaremos almacenar y modificar datos para completar ciertas tareas. El manejo de los datos en Scratch se gestiona usado **variables** y **listas**. Este capítulo solo explora el uso de las variables.

5.1. TIPOS DE DATOS EN SCRATCH.

A menudo, los programas manipulan diferentes tipos de datos, como números, textos, imágenes, etc., para producir información útil. Es muy importante conocer los tipos de datos que ofrece Scratch, y las operaciones que se pueden realizar con ellos. Scratch permite usar tres tipos de datos: datos lógicos (booleanos), números, y cadenas.

Un dato **booleano** solo puede tomar dos valores posibles: VERDADERO (true) o FALSO (false). Estos datos se usan para comprobar una o más condiciones, y en función del resultado, hacer que el programa haga una cosa u otra. Hablaremos mucho más de los booleanos en el próximo capítulo.

Una variable **numérica** puede almacenar tanto enteros como decimales. (Scratch no distingue entre ambos, y los llama simplemente *números*). Podemos redondear un decimal al entero más cercano mediante el bloque "redondear ( )" de la categoría "operadores". También podemos obtener un entero a partir de un decimal usando las funciones "techo" y "piso" del bloque "(raíz cuadrada) de ( )". Por ejemplo, "(techo) de (3,2)" da como resultado 4, y "(piso) de (3,7)" da como resultado 3.

Una **cadena** (string) es una secuencia de caracteres, incluyendo letras (mayúsculas y minúsculas), números (del 0 al 9), y otros símbolos (como +, −, &, @, $, #, etc.). Los datos tipo cadena se usan para almacenar nombres, direcciones, títulos de libros, fechas, etc.

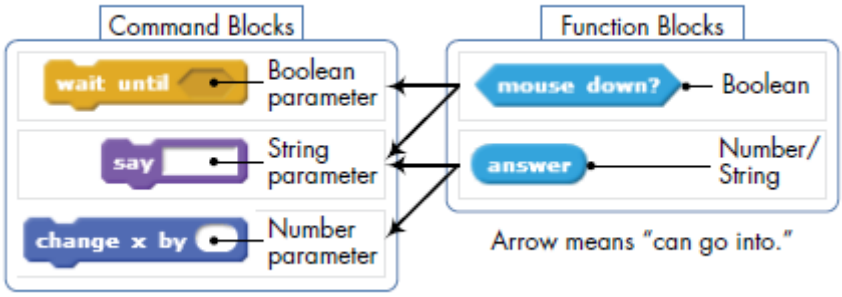
TIPOS DE DATOS QUE ACEPTAN LOS BLOQUES.

Notar que los bloques de función de Scratch y los huecos para parámetros de los bloques de comando tienen formas distintas (rectángulos con bordes redondeados, rectángulos con bordes marcados, y hexágonos).[1] La forma del hueco para parámetros de un bloque de comando está relacionada con el tipo de datos que ese bloque acepta. De forma similar, la forma de un bloque de función está relacionada con el tipo de datos que ese bloque devuelve. La figura muestra las distintas posibilidades. (Ahora bien, y a modo de ejemplo, notar que un bloque de función de bordes redondeados puede devolver tanto un número como una cadena, y notar que un hueco para parámetros con forma de rectángulo marcado acepta bloques de función con forma de hexágono y de rectángulo redondeado).

Sin embargo, no vale la pena aprendernos qué formas se relacionan con qué datos. Scratch impide que el usuario pueda introducir, digamos, un dato de tipo numérico en un bloque que requiera un dato de tipo booleano. También impide que el usuario pueda insertar un bloque de función numérico (por ejemplo,

[1] Para recordar qué era un bloque de comando y un bloque de función, revisar la sección 1.4.

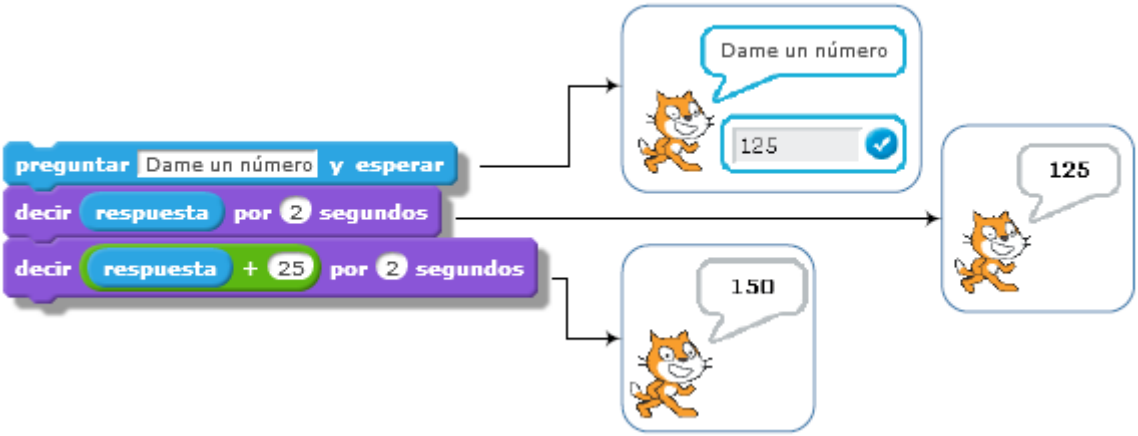
"posición y del ratón") en un bloque de comando que requiera un parámetro booleano (por ejemplo, "esperar hasta que ( )". Inténtalo, verás que es imposible.



CONVERSIÓN AUTOMÁTICA DE DATOS.

Muchos de los bloques de función que hemos usado hasta ahora ("posición en x", "posición x del ratón", "número al azar entre ( ) y ( )", etc.) dan como resultado un dato numérico, y por lo tanto, sus bordes son redondeados. Por lo tanto, usar esos bloques dentro de un bloque de comando que requiera un parámetro numérico (hueco redondeado) es perfectamente válido.

Pero algunos bloques de función de borde redondeado pueden almacenar tanto números como cadenas, como por ejemplo, el bloque "respuesta" de la categoría "sensores", o el bloque "unir" de la categoría "operadores". Por lo tanto, y a modo de ejemplo, ¿qué pasa cuando insertamos un bloque "respuesta" que almacene un número en un bloque que requiera una cadena, como el bloque "decir ( )"? Afortunadamente, Scratch siempre intenta convertir automáticamente un tipo de datos en otro, si es necesario. En este ejemplo, Scratch convertiría el dato numérico almacenado en "respuesta" en un dato de tipo cadena, tal y como requiere el bloque "decir".



La siguiente figura muestra otro ejemplo de conversión automática de datos, donde Scratch convierte el dato booleano almacenado en el bloque de función "¿tecla (espacio) presionada?" en el dato tipo cadena que requiere el bloque "pensar ( )".

5.2. INTRODUCCIÓN A LAS VARIABLES.

Imagina que quieres programar una versión software del juego de feria Whac-a-Mole, donde unos topos salen de unos agujeros, y nosotros debemos golpearlos con un mazo. En nuestra versión, un objeto saldrá en una localización aleatoria en el escenario, permanece visible durante un tiempo, y vuelve a desaparecer. Tras una breve espera, aparece de nuevo en una localización diferente. El jugador debe clicar con el ratón en el objeto tan pronto como aparezca. Cada vez que lo consigue, gana un punto. El problema a afrontar como programadores es: ¿cómo podemos seguir la puntuación del jugador? ¡Pues con variables!

¿QUÉ ES UNA VARIABLE?

Una **variable** es un espacio de memoria con nombre. Podemos pensar en ella como una caja etiquetada que almacena datos (números, textos, etc.) para que un programa pueda acceder a ellos cuando los necesite.

Cuando creamos una variable, el programa reserva la memoria suficiente para alojar el valor de la variable, y etiqueta la memoria reservada con el nombre de esa variable. Tras crear la variable, podemos usar su nombre en el programa para referirnos al valor que representa. Por ejemplo, si tenemos una variable llamada "lado" que contenga el número 50, podemos construir un comando como "mover $(3 * lado)$ pasos". Cuando Scratch ejecute este comando, localizará la variable llamada "lado" en memoria, tomará su contenido (el número 50), y usará este valor para reemplazar la etiqueta "lado" dentro del bloque "mover $(3 * lado)$ pasos".

En nuestro juego Whac-a-Mole, necesitamos una forma de recordar la puntuación del jugador. Para hacerlo, podemos reservar algo de espacio en memoria para almacenar la puntuación. También debemos darle a ese espacio de memoria una etiqueta única, digamos "puntuación", que nos sirva para encontrar ese espacio de memoria siempre que necesitemos consultar o cambiar el valor que almacena.

Cuando el juego comience, le diremos a Scratch que debe "fijar (puntuación) a (0)". Así, Scratch buscará la variable llamada "puntuación" y le adjudicará un valor de 0. Además, cada vez que el jugador clique sobre el objeto, le diremos a Scratch que debe "cambiar (puntuación) por (1)". En respuesta al primer clic, Scratch buscará la variable "puntación" nuevamente, encontrará el valor 0, y lo cambiará a 1. La siguiente vez que el jugador clique sobre el objeto, Scratch volverá a incrementar el valor de la variable "puntuación" en 1, lo que resultará en un valor total de 2.

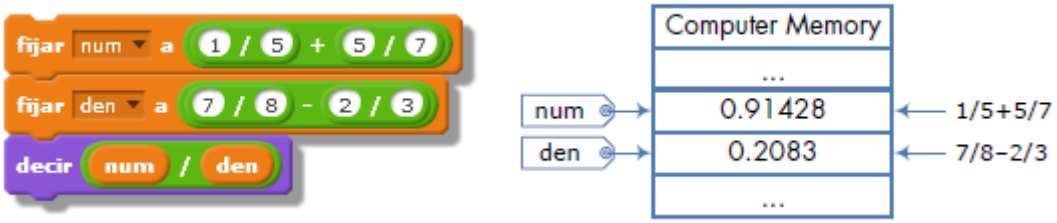
Otro uso importante de las variables es almacenar resultados intermedios a la hora de evaluar expresiones algebraicas. Este proceso funciona de forma similar a como nosotros hacemos nuestros cálculos mentales: Si nos piden hallar el resultado de $2 + 4 + 5 + 7$, probablemente comencemos haciendo $2 + 4$, y memorizando la respuesta (6). Después sumamos el 5 a la respuesta previa (que está almacenada en nuestra memoria), y memorizamos el nuevo resultado (11). Finalmente, sumamos el 7 al resultado anterior y obtenemos una respuesta final de 18. Para ilustrar cómo usar variables como almacenamiento temporal, suponer que queremos escribir un programa que calcule:

$$\frac{(1/5) + (5/7)}{(7/8) - (2/3)}$$

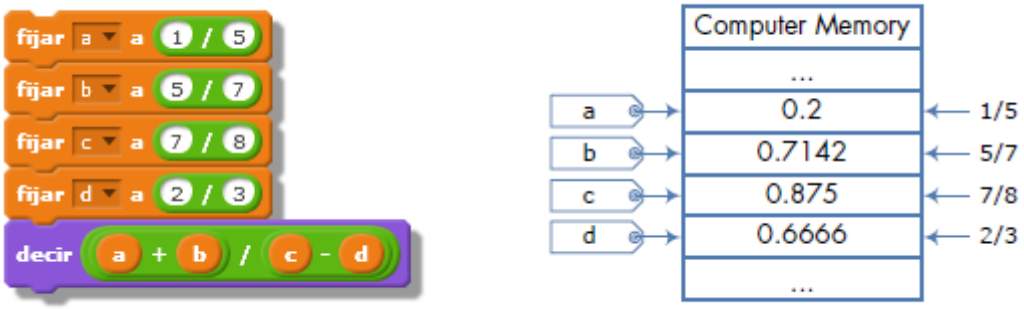
Podríamos evaluar la expresión con un solo comando, pero el programa sería difícil de leer:

Una forma más limpia de escribir el programa es evaluar el numerador y el denominador individualmente, y después usar el bloque "decir" para mostrar el resultado de su división. Para ello, creamos las variables

"num" (numerador) y "den" (denominador), y fijamos sus valores como muestra la figura. A continuación, el bloque decir toma los valores almacenados por "num" y "den" y los divide (ver figura):



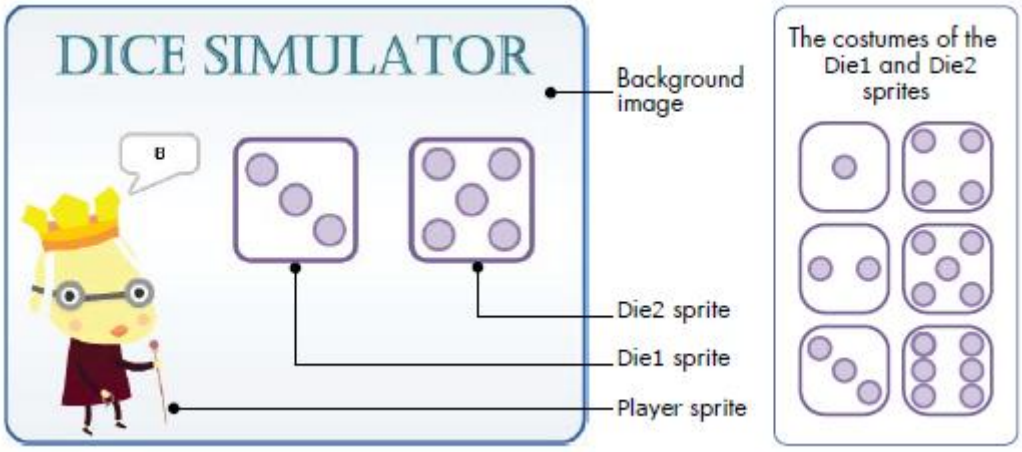
Otra posibilidad sería definir 4 variables, a, b, c, y d, cada una de las cuales almacene las 4 fracciones de la expresión. El bloque "decir" haría la suma de a y b, y la dividiría por la resta de c y d (ver figura).



Las 4 expresiones proporcionan el mismo resultado final, pero cada implementación lo hace con un estilo diferente. El primer programa es corto, pero difícil de leer. El tercer programa divide el cálculo en muchos pasos con un gran nivel de detalle, pero también podría ser difícil de leer. La segunda solución también divide el cálculo en varios pasos, pero hace que el programa sea fácil de entender. Parece que la segunda opción es la más adecuada.

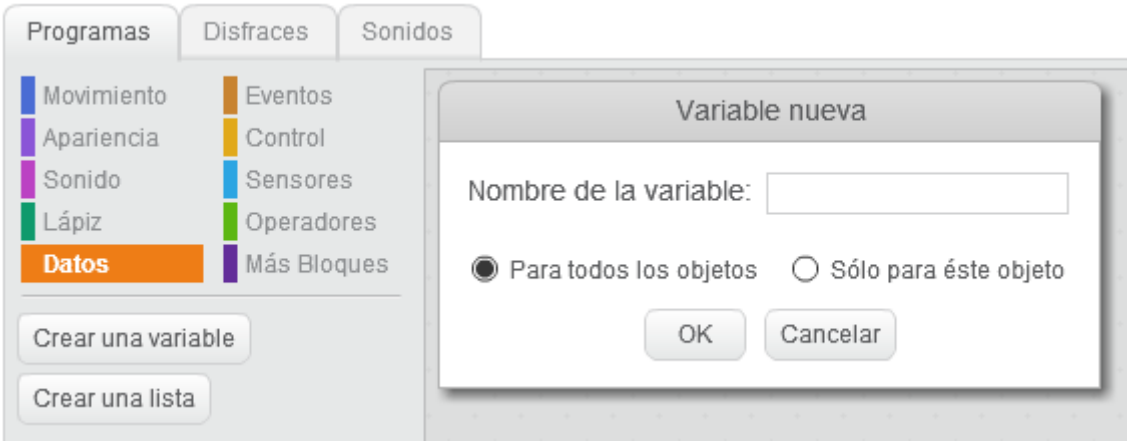
CREAR Y USAR VARIABLES.

Para poner en práctica la creación y uso de variables, vamos a desarrollar una aplicación que simule el lanzamiento de dos dados y que muestre su suma:

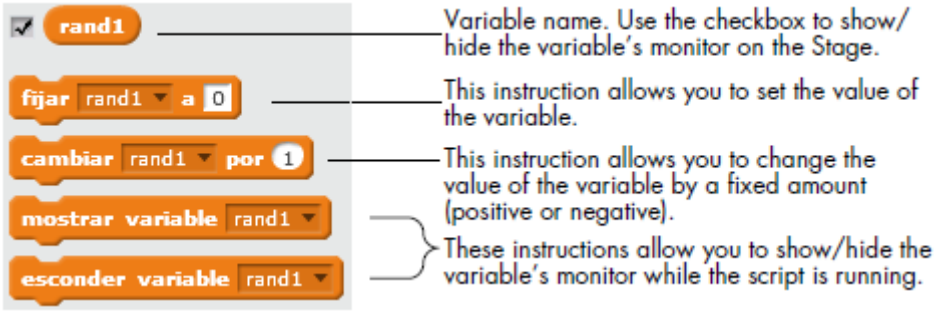


Abre el archivo **diceSimulator_sinCodigo.sb2**. El proyecto incluye 3 objetos: "player", "die1", y "die2". El objeto "player" controla la simulación: al clicar en la bandera verde, este objeto genera dos números aleatorios entre 1 y 6, y guarda esos valores en dos variables llamadas "rand1" y "rand2". A continuación, envía el mensaje "girar" a los objetos "dice1" y "dice2" para que muestren los valores generados aleatoriamente. Después de una espera de 1 segundo, el objeto "player" suma "rand1" y "rand2" y muestra el resultado con el bloque "decir".

Vamos a crear los programas necesarios. En primer lugar, selecciona el objeto "player". Acude a la categoría "datos" y clica en la opción "crear una variable". En la ventana de diálogo emergente, escribe el nombre de la variable y su **ámbito**, esto es, qué objetos pueden reescribir o cambiar el valor de la variable. En este caso, el nombre de la variable será "rand1" y el ámbito "para todos los objetos". Clica en OK para terminar.



Después de crear la variable, en la categoría "datos" aparecerán una serie de bloques nuevos:



Repite el proceso indicado más arriba para crear la variable "rand2". Ahora, la categoría "datos" debería contener un segundo bloque llamado "rand2". Ahora que hemos creado las dos variables, vamos a construir el programa para el objeto "player". El contenido de este programa ya lo hemos apuntado unos párrafos más arriba. Dale forma.

Ya hemos dicho antes que, tras fijar el valor de las variables "rand1" y "rand2", el objeto "player" envía el mensaje "girar", para notificar a los dados "die1" y "die2" que deben comenzar una ronda de 20 cambios aleatorios de disfraz (para simular que los dados se ponen a girar). Tras esta ronda, los dados terminan poniéndose el disfraz especificado por las variables "rand1" y "rand2", respectivamente. (Notar que cada dado tiene 6 disfraces, que se corresponden con las 6 caras de un dado). Por ejemplo, si la variable "rand1" tenía el valor 5, el disfraz que al final debe ponerse el dado "die1" es el de los 5 puntos. Así, el programa para el dado1 es:

Observar que, para que el dado "die1" termine poniéndose el disfraz adecuado, al segundo bloque "cambiar disfraz a ( )" le hemos pasado como parámetro la variable "rand1", cuyo valor fijó el objeto "player" antes de enviar el mensaje "girar". El programa para el dado "die2" es esencialmente idéntico.

El programa para nuestro simulador de dados ya está terminado. Pruébalo clicando en la bandera verde. Guarda el proyecto como **diceSimulator.sb2**.

EL ÁMBITO DE LAS VARIABLES.

El **ámbito** de una variable determina qué objetos pueden reescribir (esto es, cambiar) el valor de esa variable.

Podemos fijar el ámbito de una variable al crearla, seleccionando una de las dos opciones disponibles. La opción "sólo para este objeto" crea una variable que solo puede cambiar el objeto al que pertenece. El resto de objetos aún podrán leer y usar el valor de esa variable, pero no podrán reescribirlo. Por ejemplo, en el proyecto de la figura el objeto "gato" tiene una variable, llamada "cuenta", cuyo ámbito es "sólo para este objeto". El objeto pingüino puede leer y mostrar el valor de esta variable (fijado el objeto gato), pero Scratch no proporciona ningún bloque que le permita cambiar su valor. Notar que para mostrar el valor de la variable "cuenta", hemos usado el bloque "decir ( )", pasándole como parámetro el bloque de función "(posición en x) de (gato)" de la categoría sensores. Al seleccionar gato como segundo parámetro de este bloque, el primer parámetro nos permitirá elegir un atributo del objeto gato, incluida una de sus variables.

Escoger el ámbito "sólo para este objeto" es una buena elección cuando queremos que una variable solo pueda ser actualizada por el objeto al que pertenece, evitando así que otros objetos puedan causar modificaciones indeseadas en su valor.

A las variables con ámbito "sólo para este objeto" se las denomina **variables locales**, y se dice que su ámbito es local. Distintos objetos pueden tener variables locales con el mismo nombre sin que se produzca ningún conflicto. Por ejemplo, si tenemos dos objetos "coche" en un juego de carreras, cada uno podría tener una variable local llamada "velocidad" que determinase la rapidez del coche en el escenario. Cada coche podría cambiar el valor de su variable "velocidad" de manera independiente a la del otro coche. (Así, si fijásemos la velocidad del coche1 a $50\ km/h$, y la velocidad del coche2 a $70\ km/h$, el segundo debería moverse más rápido que el primero).

Por otro lado, a las variables con ámbito "para todos los objetos" pueden ser leídas y cambiadas por cualquier objeto de nuestra aplicación. Estas variables, también llamadas **variables globales**, son útiles para la comunicación y la sincronización entre distintos objetos. Por ejemplo, si un juego tiene tres botones que permiten al usuario seleccionar uno de los tres posibles niveles donde jugar, podemos crear una variable global llamada "nivel", y hacer que cada objeto botón ajuste esta variable a un valor diferente al ser clicado.

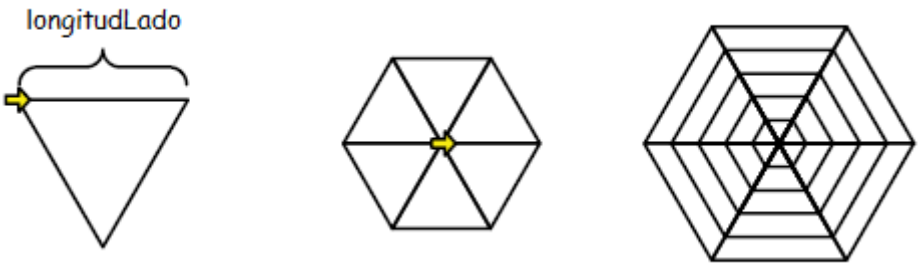
CAMBIAR VARIABLES.

Scratch proporciona dos bloques de comando que nos permiten cambiar el valor de una variable. El bloque "fijar ( ) a ( )" asigna directamente un nuevo valor a una variable, independientemente de su contenido

actual. Por otro lado, el bloque "cambiar ( ) por ( )" cambia al valor de una variable en una cierta cantidad en relación a su valor actual. Los tres programas de la figura muestran distintas formas de usar estos comandos para lograr la misma salida: la variable "sum" acaba almacenando el valor 5.

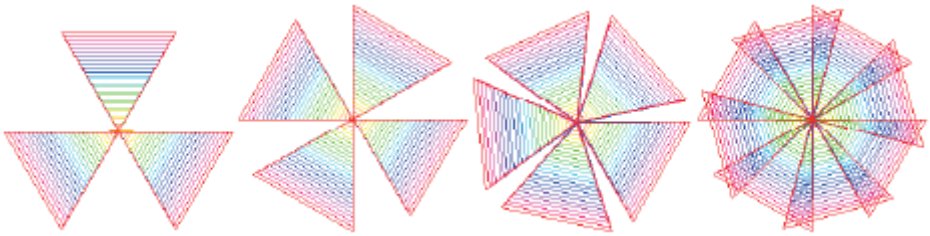
EJERCICIO 23. TELARAÑA.

Podemos dibujar una telaraña dibujando varios hexágonos de tamaño cada vez mayor, como muestra la figura (la flecha indica la dirección inicial del objeto dibujante). Para ello, crea un procedimiento "triángulo" que dibuje un triángulo equilátero, con un parámetro que sea la longitud del lado. A continuación, crea un procedimiento "hexágono" (con un parámetro que sea la longitud del triángulo) que llame al procedimiento "triángulo" seis veces, con un giro a la derecha de $60°$ (esto es, $360°/6$) tras cada llamada. A continuación, crea la variable "longitudLado" y el procedimiento "telaraña", el cual comienza ajustando el valor de "longitudLado" a cero. A continuación, un bucle "repetir" (con tantas repeticiones como tamaño queramos para la telaraña, en la figura hemos usado 5 repeticiones) cambia el valor de "longitudLado" en 15 unidades, y llama al procedimiento "hexágono" pasándole como parámetro la variable "longitudLado". El programa principal comienza al clicar en la bandera, y simplemente esconde el objeto dibujante, lo ubica en el centro del escenario, lo orienta hacia la derecha, fija el color y el tamaño del lápiz (120 y 1, respectivamente), baja el lápiz, limpia la pantalla, y llama al procedimiento "telaraña". Guarda el programa como **Ejercicio 23.sb2**.



EJERCICIO 24. MOLINILLO.

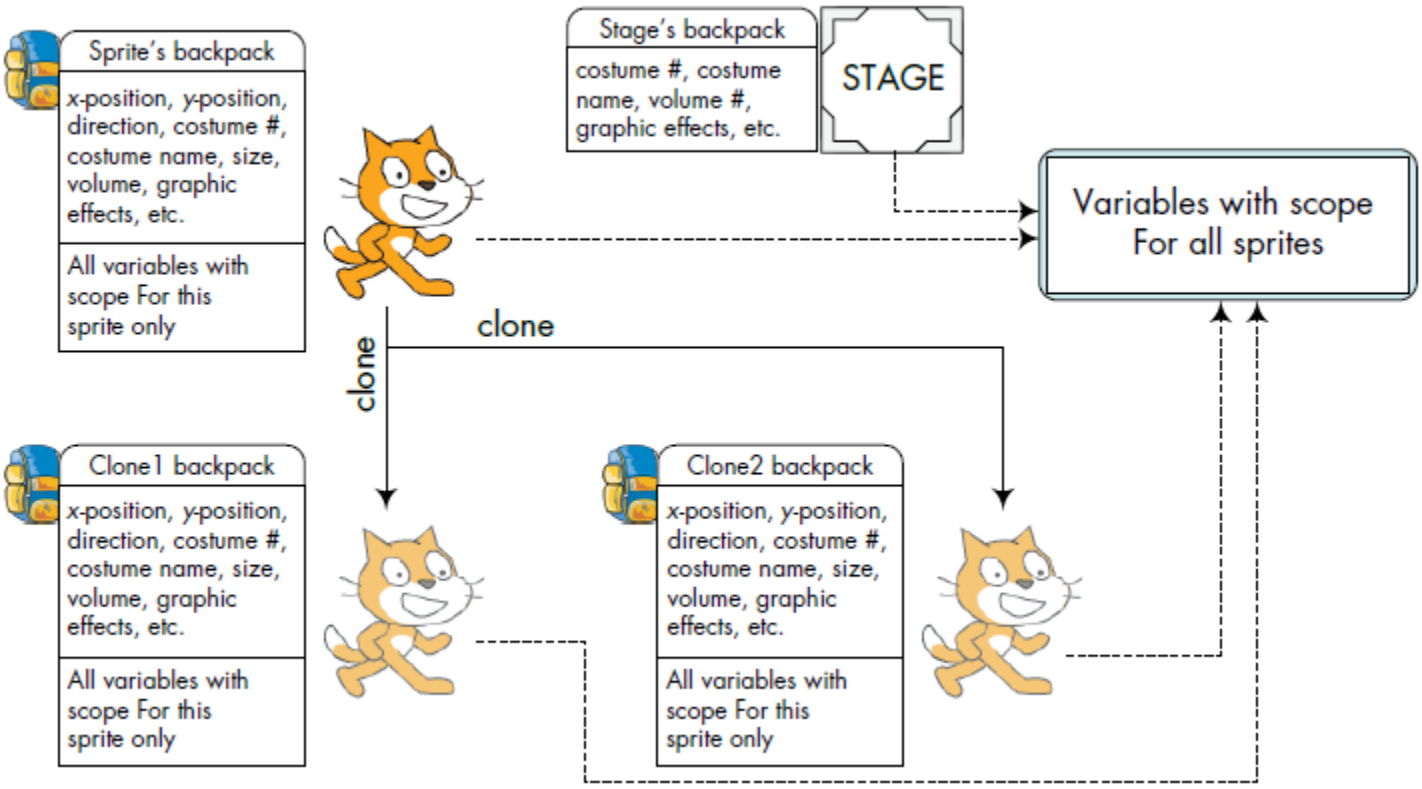
En este ejercicio vamos a hacer un molinillo de colores. El procedimiento es similar al del ejercicio anterior, pero en este caso usaremos una variable para controlar el número de repeticiones triangulares. El procedimiento resultante, al que llamaremos "palas" (y que es muy similar al procedimiento "hexágono") llamará al procedimiento "triángulo" tantas veces como indique esa variable. A continuación, el procedimiento "molinillo" funciona de forma similar al procedimiento "telaraña", pero esta vez también cambia el color del lápiz a cada pasada del bucle, para crear un bonito efecto arcoíris. El programa principal es muy parecido al del ejercicio previo. La figura muestra varias salidas para diferentes valores de la variable que controla el número de repeticiones triangulares. Guarda el programa como **Ejercicio 24.sb2**.



VARIABLES EN CLONES.

Cada objeto tiene una lista de propiedades asociadas, como su posición x actual, su posición y actual, su dirección, etc. Podemos imaginarnos esa lista como una mochila donde el objeto guarda los valores actuales de dichas propiedades. Cuando creamos para un objeto una variable con el ámbito "solo para este objeto", esa variable se añade a la lista de propiedades del objeto (a la mochila del objeto).

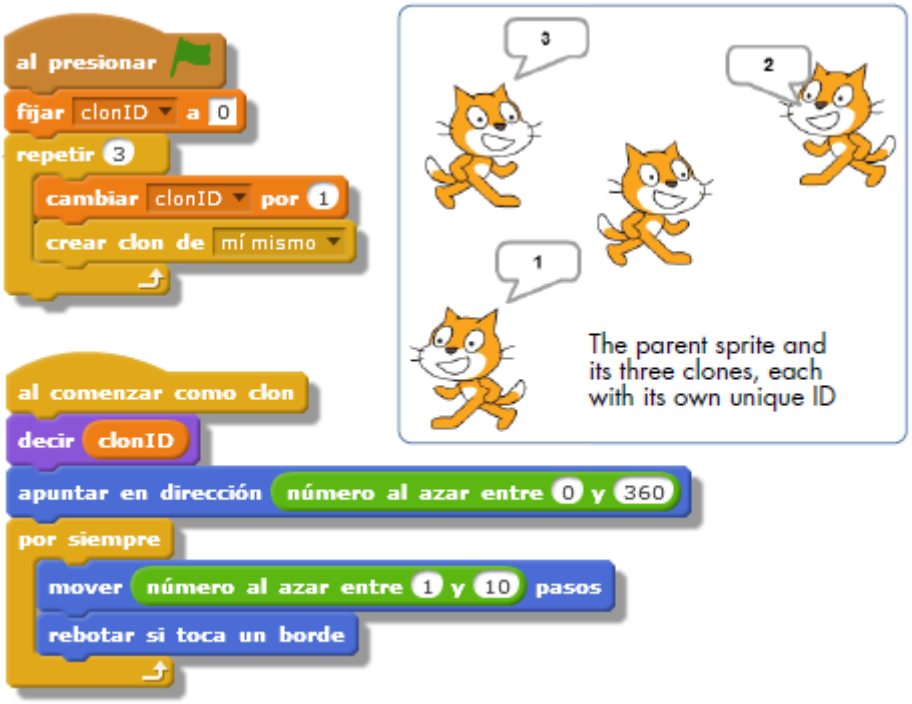
Cuando clonamos un objeto, el clon hereda una copia de todos los atributos del objeto maestro, incluyendo sus variables. Una propiedad heredada comienza con un valor idéntico a la propiedad del maestro, en el instante en el que se crea el clon. Pero después, los atributos y las variables del clon evolucionan de forma independiente, y sus cambios no afectan al maestro. Además, los subsiguientes cambios en los atributos del objeto maestro tampoco afectan a las propiedades de los clones.



Por ejemplo, digamos que el objeto maestro posee una variable local llamada "velocidad", cuyo valor actual es 10. Al crear un clon del maestro, el nuevo objeto también tendrá una variable local llamada "velocidad" con un valor de 10. Después de eso, si el objeto padre cambia su velocidad a 20, el valor de la velocidad del clon seguirá siendo 10.

Podemos usar este hecho para distinguir los distintos clones de nuestra aplicación. Para ello, abre el programa **CloneIDs.sb2**. El objeto maestro (u objeto padre) posee una variable llamada "clonID" (un número de identificación para sus clones).

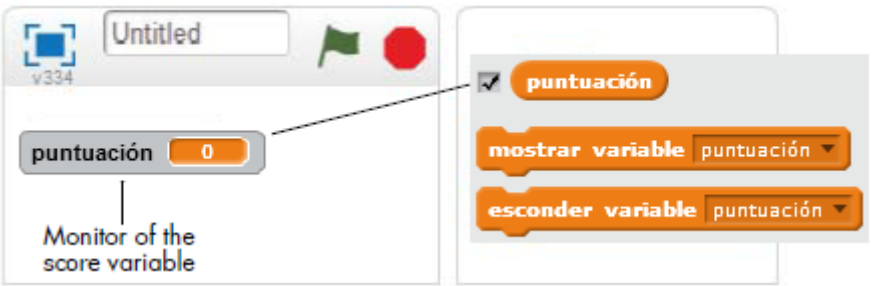
Al clicar la bandera, el objeto maestro comienza un bucle que crea tres clones, y ajusta su variable "clonID" a un valor distinto (1, 2, y 3) antes de crear cada clon. Después, cada clon nace con su propia copia de "clonID" inicializada a un valor diferente. De esta forma, podríamos usar un bloque "si ( ) entonces" para comprobar el número de identificación del clon, y hacer que cada clon realice una acción distinta.



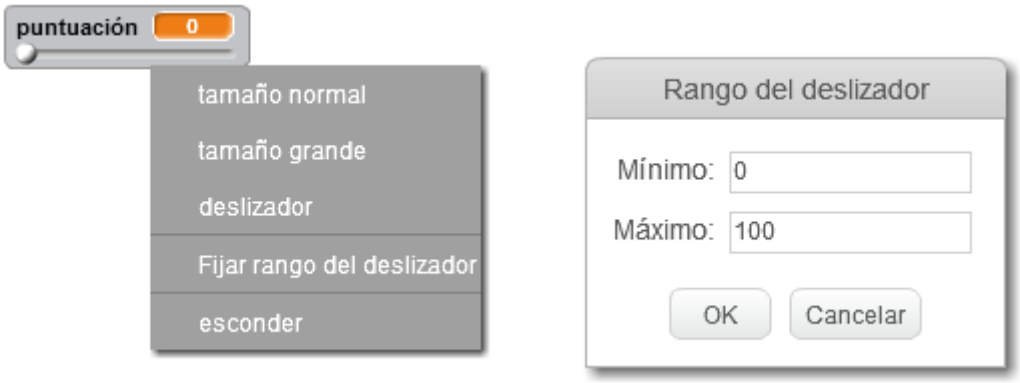
Ahora veamos la forma en la que los clones interactúan con las variables globales (variables con ámbito "para todos los objetos"). Recordar que las variables globales pueden leerlas y cambiarlas todos los objetos, incluyendo los clones. El programa **ClonesAndGlobalVars.sb2** usa este hecho para detectar cuándo desaparecen todos los clones del objeto maestro. En el programa, el maestro ajusta la variable "numClones" a 5 y crea 5 clones. A continuación, espera a que "numClones" tome el valor 0 para anunciar el final del programa (game over). Los clones aparecen en instantes de tiempo y localizaciones aleatorias, dicen "Hola!" durante dos segundos, y desaparecen. Antes de desaparecer, cada clon disminuye el valor de la variable "numClones" en 1 unidad.

5.3. MOSTRAR LOS MONITORES DE LAS VARIABLES.

Al crear una variable, en la categoría "datos" aparecerá un bloque de función con el nombre de esa variable, junto al cual veremos una casilla. Si activamos la casilla, en el escenario aparecerá un monitor que siempre muestra el valor actual de esa variable:



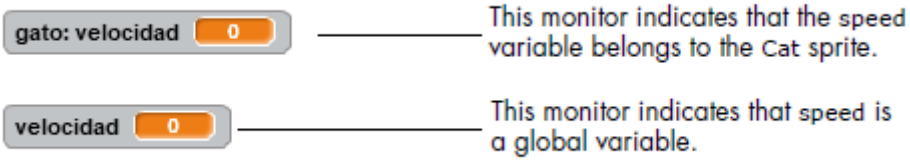
Esta funcionalidad es muy útil cuando queremos seguir el valor de una variable, y comprobar si toma el valor correcto. También podemos cambiar la visibilidad de un monitor desde el propio programa, usando los bloques "mostrar variable ( )" y "esconder variable ( )".



Los monitores pueden usarse como lectores o como controles, porque permiten mostrar y cambiar el valor de una variable, respectivamente. Haz doble clic sobre el monitor para pasarlo a tamaño grande, otro doble clic para pasarlo a modo deslizador, y un tercer doble clic para volver a dejarlo en modo normal. (También podemos acceder a todas estas opciones haciendo clic con el botón derecho del ratón sobre el monitor). Al elegir el modo deslizador, también aparecerá la opción "fijar rango del deslizador", para indicar los valores máximo y mínimo a los que puede llegar el deslizador.

El deslizador nos permite cambiar el valor de la variable mientras el programa está en ejecución, lo que facilita a los usuarios interactuar con nuestra aplicación. Por ejemplo, el programa de la figura (para el escenario) le permite al usuario, arrastrando el deslizador de la variable "colorEscenario", controlar el color de fondo del escenario.

NOTA: El monitor de una variable también indica su ámbito. Si una variable pertenece exclusivamente a un objeto, el monitor de la variable debe mostrar el nombre del objeto delante del nombre de la variable. En la figura, el monitor "gato: velocidad" indica que la variable "velocidad" pertenece al objeto gato. Si la variable "velocidad" fuese global, su monitor sólo indicaría "velocidad".



USAR MONITORES DE VARIABLES EN PROGRAMAS.

La posibilidad de usar monitores como indicadores y controles nos ofrece multitud de aplicaciones, tanto en juegos como en simulaciones y programas interactivos.

EJERCICIO 25. LEY DE OHM.

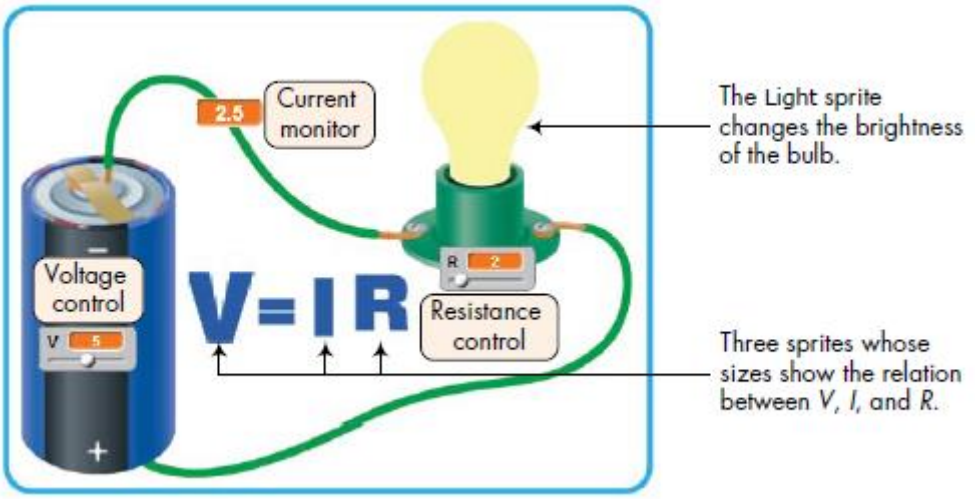
Como sabrás de electricidad, la ley de Ohm dice que cuando se aplica un voltaje V entre los extremos de un componente eléctrico de resistencia R, la intensidad I de la corriente que circula a su través es:

$$I = \frac{V}{R}$$

Vamos a construir una aplicación que le permita al usuario cambiar los valores de V y R usando controles con deslizador. Una vez fijados esos valores, la aplicación muestra el correspondiente valor de la intensidad I de la corriente.

Abre el archivo **Ejercicio 25_sinCodigo.sb2**. Este archivo incluye un escenario con el dibujo de un circuito, y 5 objetos: la letra V, el símbolo $=$, la letra I, la letra R, y una luz superpuesta sobre la bombilla del escenario.

Comienza el ejercicio creando 3 variables globales: V, I, y R. Ubica sus monitores en las posiciones y la apariencia mostradas en la figura. El deslizador para el voltaje V tiene un rango de $[0, 10]$, y el deslizador del resistor un rango de $[1, 10]$. Cuando el usuario cambia V o R con los deslizadores, la aplicación calcula el valor correspondiente de la intensidad de la corriente I que circula por el circuito. El brillo de la bombilla cambia en proporción al valor de la corriente que pasa a su través: a más corriente, más brillo. Los tamaños de las letras V, I, y R también cambian para indicar los tamaños relativos de estas cantidades.



1) Vamos a crear el programa principal, que es el del escenario: Este programa empieza al clicar en la bandera verde, y comienza fijando los valores de las variables V y R a 0 y 1, respectivamente. A continuación, y de forma indefinida (para siempre) fija el valor de la variable I a V/R (ley de Ohm), y envía el mensaje "actualizar" al resto de objetos y queda en espera.

2) Cuando los objetos "Volt", "Equal", "Current", y "Resistance" reciben el mensaje "actualizar", cada uno de ellos cambia su tamaño (desde un 100% a un 200% de su tamaño original) en relación a los valores actuales de sus respectivas variables. Eso lo hacemos con el bloque adecuado, pasándole como argumento el siguiente bloque de función:

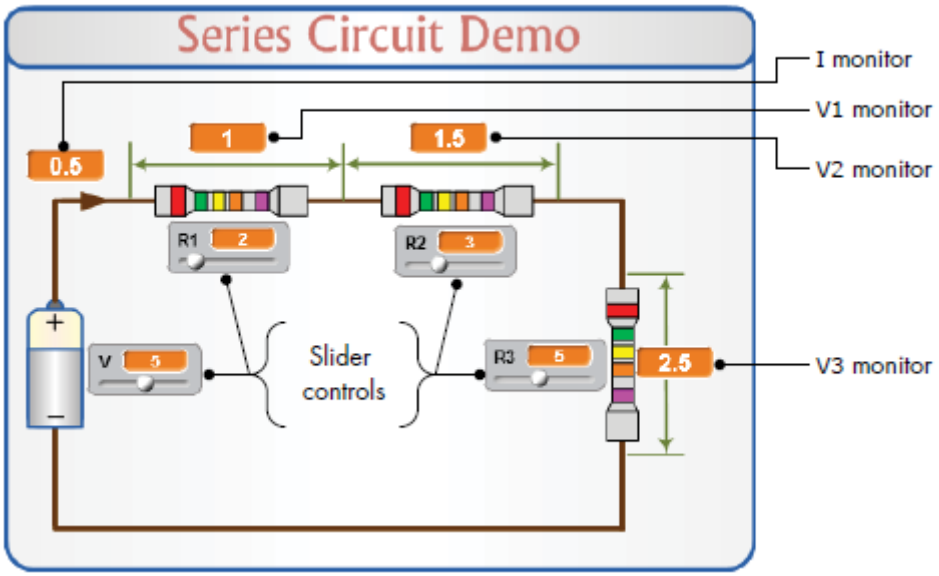
(El ejemplo es para el objeto "Volt"). Por su parte, el objeto "light" fija el efecto "desvanecer" a un valor apropiado para que su nivel de transparencia sea proporcional al valor de I. Este valor proporcional es:

Por supuesto, la aplicación debe empezar (al clicar sobre la bandera verde) fijando el tamaño de los objetos "Volt", "Equal", "Current", y "Resistance" a su tamaño original (100%), y el efecto "desvanecer" del objeto "Light" a su valor inicial (100). Añade los programas necesarios a los códigos de todos estos objetos.

3) Guarda el ejercicio como **Ejercicio 25.sb2**.

EJERCICIO 26. CIRCUITO SERIE.

Abre el archivo **Ejercicio 26 (1)_sinCodigo.sb2**. Se trata de un proyecto sin objetos, pero con un escenario que muestra un circuito serie con una pila conectada a tres resistores. Para tu comodidad, este proyecto ya cuenta con una serie de variables predefinidas: I, R_1, R_2, R_3, R_{tot}, V, V_1, V_2, y V_3. El usuario puede fijar a voluntad el voltaje V de la pila (entre 0 y 10 voltios), y los valores de resistencia R_1, R_2, y R_3 de los tres resistores (entre 1 y 10), usando los deslizadores correspondientes. El objetivo de este proyecto es calcular y mostrar mediante monitores los valores de la intensidad de corriente I que circula por el circuito, así como los voltajes entre los extremos de cada resistor en serie, V_1, V_2, y V_3 (ver figura).



Como sabrás de electricidad, las ecuaciones que gobiernan el comportamiento de cualquier circuito serie son:

$$R_{tot} = R_1 + R_2 + R_3$$

$$I = \frac{V}{R_{tot}}$$

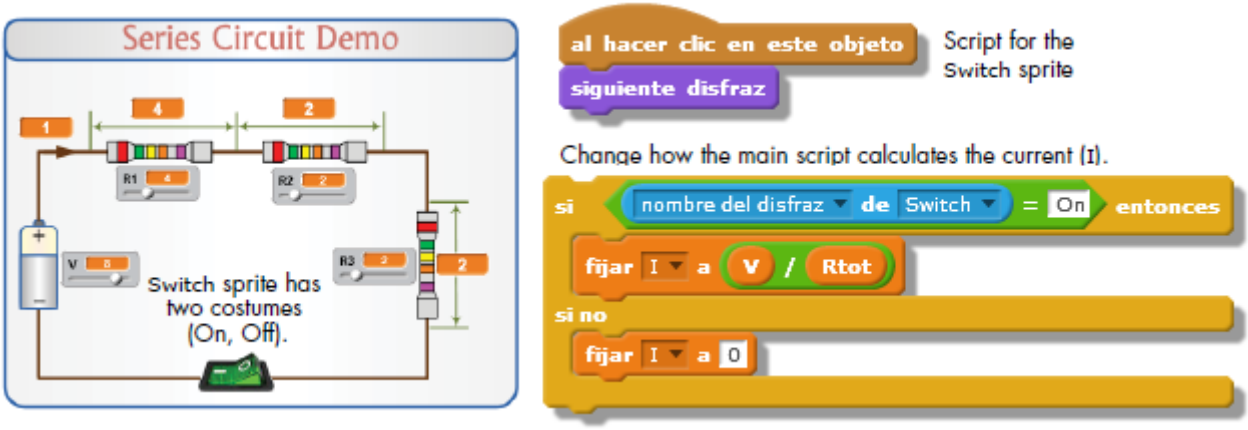
$$V_1 = I\,R_1, \qquad V_2 = I\,R_2, \qquad V_3 = I\,R_3$$

$$V = V_1 + V_2 + V_3$$

Guarda el ejercicio como **Ejercicio 26 (1).sb2**.

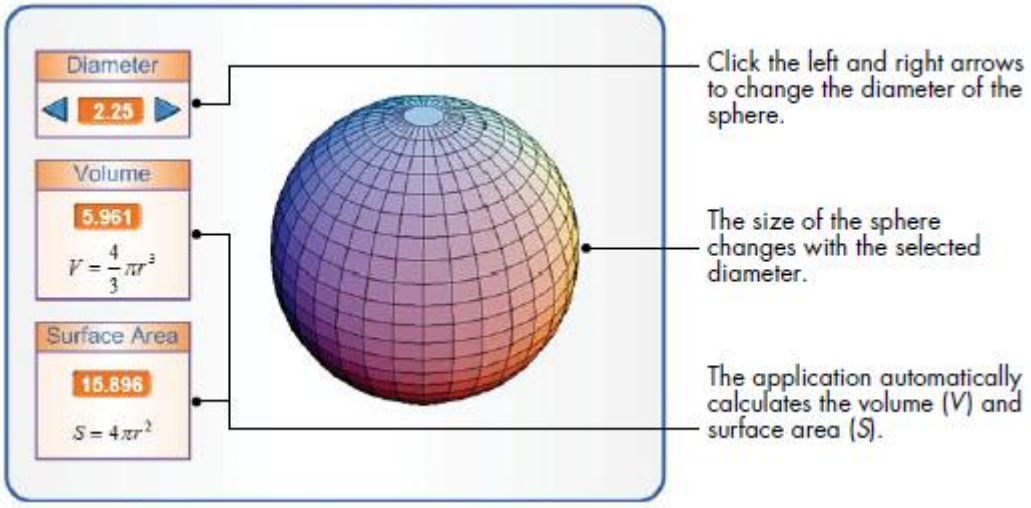
AMPLIACIÓN: Puedes construir una interesante mejora de esta aplicación añadiendo la imagen de un interruptor que abra o cierre el circuito. Si el interruptor está abierto, no hay corriente circulando por el circuito. Si el interruptor está cerrado, la corriente circula por el circuito serie, y éste funciona. Usa el

archivo **Ejercicio 26 (2)_sinCodigo.sb2** y las pistas que te mostramos más abajo para implementar esta mejora. Guarda el ejercicio como **Ejercicio 26 (2).sb2**.



EJERCICIO 27: ÁREA Y VOLUMEN DE UNA ESFERA.

El objetivo es hacer una aplicación interactiva para calcular el volumen y el área de una esfera, dado su diámetro ($diametro = 2 \times radio$). El usuario inserta el diámetro clicando en unos botones en la interfaz de usuario. Para que la aplicación sea visualmente atractiva, el tamaño de la esfera mostrada en el escenario también cambia en proporción al diámetro elegido. La figura muestra la interfaz de usuario de la aplicación, que ya está construida en el archivo **Ejercicio 27_sinCodigo.sb2**.



La aplicación contiene tres objetos (los botones "up" y "down" y la esfera), un fondo con las tres ventanas de diámetro, volumen y área, y 4 variables (el volumen V, el área S, el diámetro, y el radio r).

1) Los programas de los botones "up" y "down" comienzan cuando el usuario clica sobre ellos, y simplemente reportan si han sido clicados, enviando los mensajes "arriba" y "abajo", respectivamente.

2) El objeto esfera tiene 9 disfraces que representan esferas de diámetros $1,\ 1.25,\ 1.5,\ 1.75, \ldots, 3$. Cuando la esfera recibe los mensajes "arriba" o "abajo", ejecuta los programas:

El bloque de función "# de disfraz" de la categoría "apariencia" almacena el número de disfraz que el objeto lleva puesto actualmente. Por consiguiente, y a modo de ejemplo, el programa de la derecha arranca cuando el usuario clica en el botón "up" para incrementar el diámetro. Lo primero que hace es comprobar si el disfraz actual no es el último (el de la esfera más grande, de diámetro 3). En ese caso, cambia el disfraz del actual al siguiente más grande, y llama al procedimiento "recalcular", para obtener el volumen y la superficie de la nueva esfera a partir del nuevo diámetro seleccionado.

El procedimiento "recalcular" comienza fijando el valor de la variable diámetro a partir del número de disfraz actual, mediante la fórmula:

$$diametro = 1 + 0.25 \times (numero\ disfraz - 1)$$

El número de disfraz varía de 1 a 9, y los valores de los diámetros correspondientes son 1, 1.25, 1.5, 1.75, … , 3. Esta fórmula permite hacer exactamente esa conversión. A continuación, el procedimiento fija el valor de la variable radio a partir de la variable diámetro (ya deberías saber cómo hacerlo). Por último, el procedimiento fija los valores de las variables V y S usando las fórmulas para el volumen y el área de una esfera, $V = \frac{3}{4}\pi r^3$ y $S = 4\pi r^2$. (Toma $\pi = 3{,}14$).

3) Guarda el ejercicio como **Ejercicio 27.sb2**.

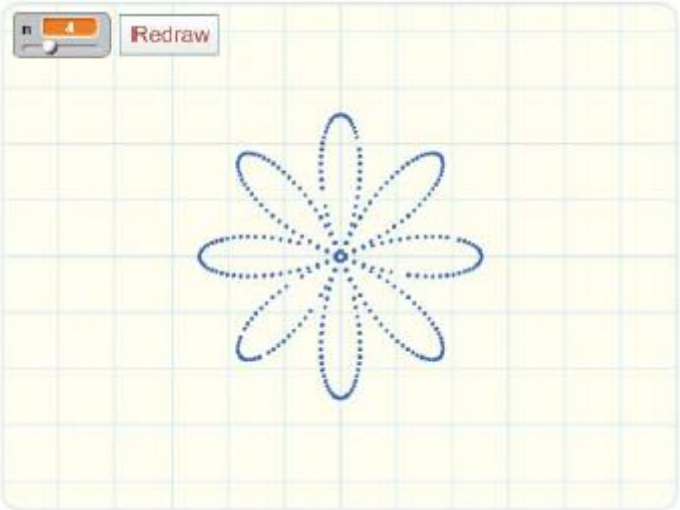
EJERCICIO 28. ROSA DE N HOJAS.

En este ejercicio crearás una aplicación que dibuje una rosa con múltiples hojas. El proceso para dibujarla es el siguiente:
1) Comenzamos en el origen del escenario.
2) Apuntamos al objeto dibujante en una cierta dirección. Como los ángulos se suelen representar con la letra griega θ ("zeta"), la variable que indica la dirección del objeto la denominaremos "zeta".
3) Movemos el objeto r pasos y dibujamos un único punto sobre el escenario. Después de hacerlo, subimos el lápiz, y volvemos al origen.
4) Cambiamos el ángulo "zeta" en una cierta cantidad (por ejemplo, $1°$), y repetimos los pasos 2 a 4.

La relación entre la distancia r y el ángulo "zeta" es:

$$r = a \times \cos(n \times \theta)$$

, donde a es un número real y n es un entero. Esta ecuación permite dibujar una rosa cuyo tamaño y número de hojas vienen controlados por a y n, respectivamente. Esta ecuación también incluye a la función trigonométrica coseno ($\cos$), que encontraremos en el bloque "(raíz cuadrada) de ( )" de la categoría "operadores".



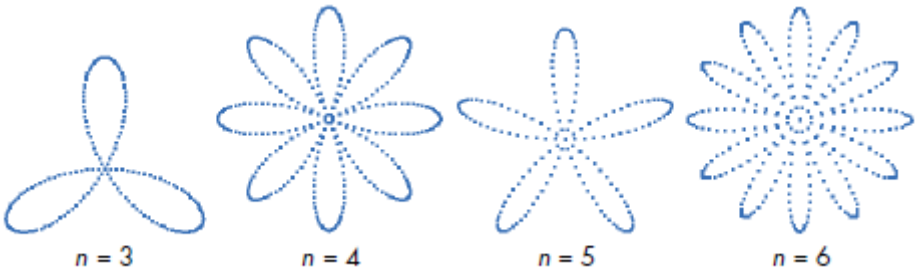
Abre el archivo **Ejercicio 28_sinCodigo.sb2**, donde encontrarás la interfaz de usuario de esta aplicación. Esta aplicación contiene 2 objetos: el botón "redraw" (redibujar), y un objeto oculto que actúa como objeto dibujante. Además, la aplicación también tiene definidas tres variables: a, n, y zeta. El usuario controla el número de hojas de la rosa mediante el monitor con deslizador de la variable n. Una vez fijado el valor de n, clica en el botón "redraw" para redibujar la rosa. Vamos con los programas:

a) El programa del botón es sencillo: cuando el usuario clica sobre el botón, éste simplemente envía el mensaje "redibujar".

b) Cuando el objeto dibujante recibe el mensaje "redibujar", se va al centro del escenario, fija el color y el tamaño del lápiz (120 y 3, respectivamente), sube el lápiz, borra el escenario, fija el valor de la variable a (por ejemplo, a 100), y llama al procedimiento "rosa".

c) El procedimiento "rosa" no tiene parámetros. Comienza fijando el valor inicial de la variable "zeta" a 0 grados. A continuación comienza un bucle de 360 repeticiones. A cada pasada del bucle, el procedimiento sube el lápiz, acude al centro del escenario, apunta en la dirección "zeta", se mueve $r = a \times \cos(n \times \theta)$ pasos, baja el lápiz para dibujar un punto en esa localización, y actualiza el valor de la variable "zeta" en 1 grado. (El número de repeticiones del bucle es 360 porque $360 \times 1° = 360°$).

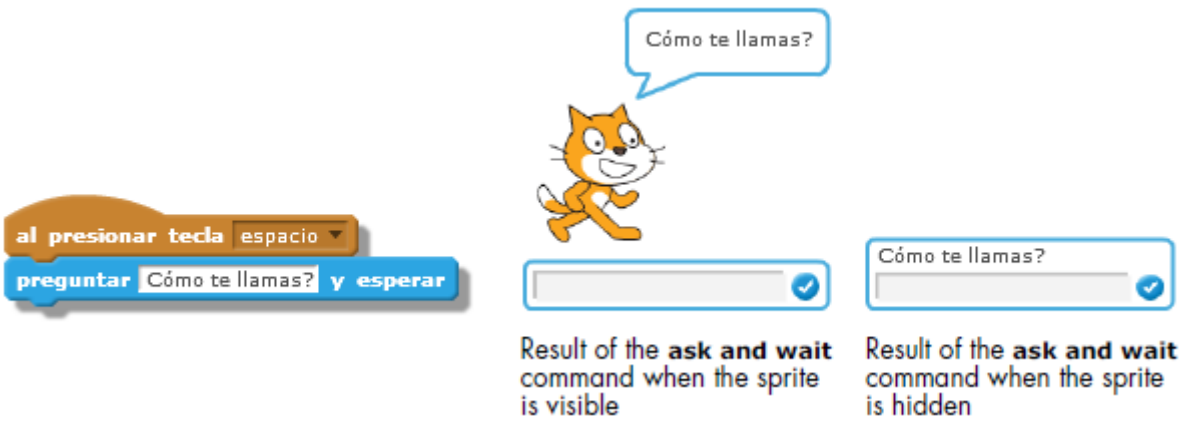
La figura muestra las salidas para distintos valores de n. ¿Puedes hallar la relación entre el valor de n y el número de hojas?



AMPLIACIÓN: Añade otro monitor con deslizador para permitir al usuario cambiar el valor de la variable a, y modifica los programas como creas conveniente. También puedes modificar el procedimiento "rosa" para que acepte a como parámetro de entrada. (Ver subsección "pasarle parámetros a un bloque customizado" de la sección 4.2).

5.4. OBTENER DATOS DE LOS USUARIOS.

Imagina que quieres crear una aplicación para enseñar matemáticas básicas a los niños. Por ejemplo, podríamos tener un objeto que muestre una suma, y le pida al usuario que introduzca el resultado. ¿Cómo harías para leer la respuesta introducida por el usuario, y ver si es correcta?

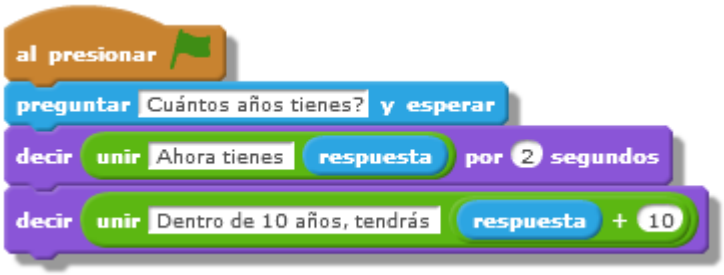


La categoría "sensores" incluye el bloque "preguntar ( ) y esperar" para leer los datos insertados por el usuario. A este bloque se le pasa como argumento una cadena, normalmente la pregunta que le hacemos al usuario. La ejecución de este bloque produce una salida ligeramente distinta dependiendo de la visibilidad del objeto, esto es, de si el objeto está oculto o no (ver figura). La salida mostrada a la derecha de la figura también es la que aparece si el bloque "preguntar ( ) y esperar" está en un programa del escenario.

Después de ejecutar el bloque "preguntar ( ) y esperar", el programa queda en espera hasta que el usuario presiona la tecla ENTER o clica en el botón azul junto a la caja de entrada de datos para el usuario. Cuando eso ocurre, Scratch almacena el dato introducido por el usuario en el bloque "respuesta", y continúa con la ejecución del programa con el bloque que está justo después del bloque "preguntar ( ) y esperar".

A modo de ejemplo, abre el archivo **LeerDatosUsuario.sb2**, y observa cómo funciona:

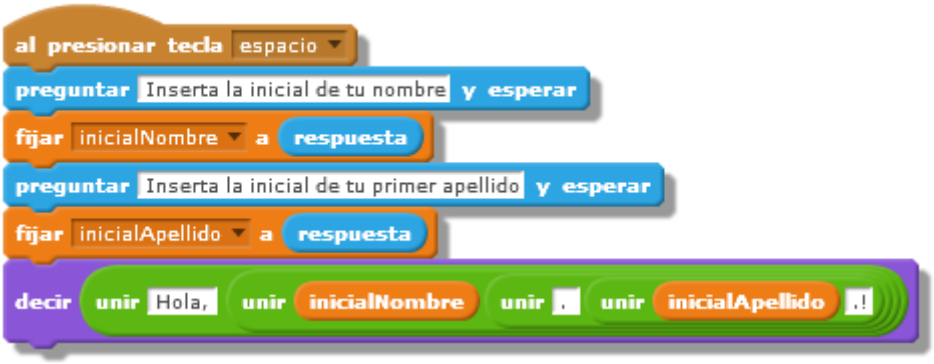
Leer números: Al clicar en la bandera verde, el programa pregunta al usuario su edad y queda en espera. Cuando el usuario inserta su edad, este dato queda almacenado en el bloque de función "respuesta", que utilizamos como entrada en sendos bloques "decir" (junto con varios bloques "unir") para mostrar por pantalla su edad actual, y su edad dentro de 10 años.



NOTA: El bloque "unir ( ) ( )" sirve para combinar (concatenar) dos cadenas. Por ejemplo, el comando:

, hace que el gato diga por pantalla "Hola mundo!". (Notar que hemos añadido un espacio en blanco al final de la cadena "Hola" para separar las dos cadenas). Para ver más ejemplos de uso del bloque "unir ( ) ( )", ejecuta el archivo **Bloque_unir.sb2**.

Leer caracteres: El segundo programa comienza al presionar la tecla espaciadora del teclado. El programa le pide al usuario la inicial de su nombre y espera. El dato insertado por el usuario se almacena en el bloque "respuesta", y a continuación, el programa fija el valor de la variable "inicialNombre" usando el valor almacenado en "respuesta". El programa repite el proceso previo para leer y almacenar la inicial del primer apellido del usuario en la variable "inicialApellido". A continuación, mediante un bloque "decir" y una combinación de bloques "unir", el objeto saluda al usuario mostrando sus iniciales separadas por puntos.



Realizar operaciones aritméticas: El último programa comienza al clicar en el gato, y le pide al usuario dos números, que almacena en las variables "num1" y "num2". A continuación, y mediante un bloque decir (junto con una combinación de bloques "unir"), mostramos por pantalla una cadena de texto con la multiplicación de los dos números y el símbolo =, y a continuación, el resultado de dicha multiplicación.

EJERCICIO 29. ECUACIÓN CUADRÁTICA.

Los ejemplos previos muestran varias formas de usar los bloques "preguntar y esperar" y "respuesta" para escribir programas que reciban datos del usuario y resuelvan una variedad de problemas. Ahora, y a modo de práctica, te proponemos construir un programa para hallar las raíces (o soluciones) de una ecuación cuadrática de la forma $ax^2 + bx + c = 0$, para unos valores a, b, y c cualesquiera proporcionados por el usuario. Tienes libertad total para diseñar la interfaz de usuario, procura ser original y creativo. Guarda el ejercicio como **Ejercicio 29.sb2**.

EJERCICIO 30. TEOREMA DE PITÁGORAS.

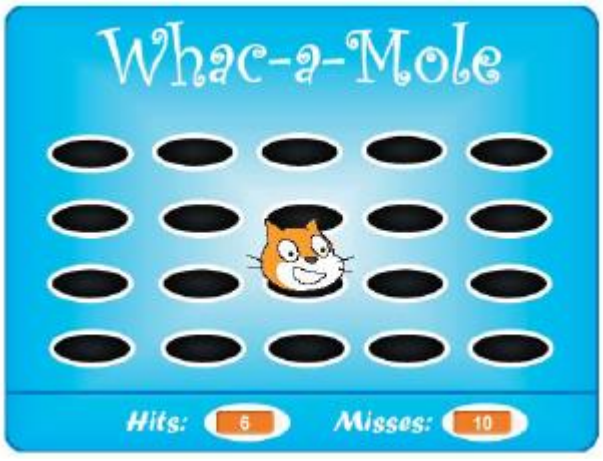
Escribe un programa que le pida al usuario las longitudes de los dos catetos de un triángulo rectángulo, y calcule la longitud de la hipotenusa. Diseña también la interfaz de usuario. Guarda el ejercicio como **Ejercicio 30.sb2**.

EJERCICIO 31. ÁREA Y VOLUMEN DE UN PARALELEPÍPEDO.

Escribe un programa que le pida al usuario la longitud L, la anchura W, y la altura H de un paralelepípedo regular (una caja). El programa calculará y mostrará el área y el volumen de esa figura. (Pista: $Volumen = L \times W \times H$; $Area = 2 \times [(L \times W) + (L \times H) + (H \times W)]$. Guarda el ejercicio como **Ejercicio 31.sb2**.

EJERCICIO 32: WHAC-A-MOLE.

En este ejercicio vamos a completar el juego Whac-a-Mole del que hablamos al principio del capítulo (sección 5.2).



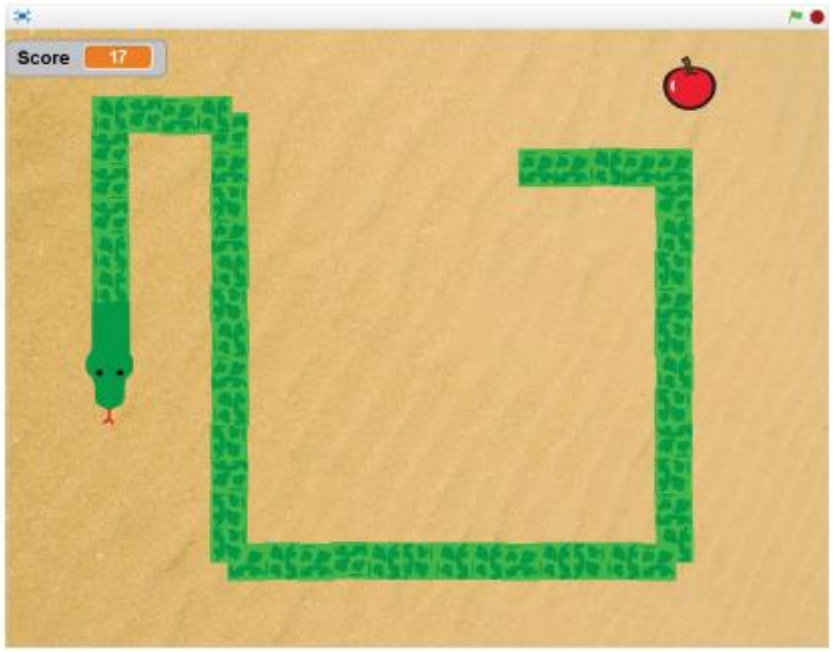
El archivo **Whac-a-Mole.sb2** contiene una implementación parcial del programa. No intentes entender el contenido del bucle "por siempre", aún no hemos llegado a ello. Basta con saber que al clicar en la bandera verde, el bucle mueve el objeto gato de forma aleatoria sobre los agujeros. Añade dos programas (uno para el gato y otro para el escenario) para cambiar los valores de las variables, "hits" (aciertos) y "misses" (fallos) de forma apropiada. Necesitarás usar los bloques de activación "Al hacer clic en este objeto" y "Al presionar escenario" de la categoría

"Eventos". Además, añade algunos efectos de sonido para hacer el programa más divertido. También puedes añadir una condición que finalice el juego después de que un temporizador o de que el número de aciertos o fallos lleguen a unos valores determinados.

5.5. PROYECTOS SCRATCH.

PROYECTO 8. LA SERPIENTE.

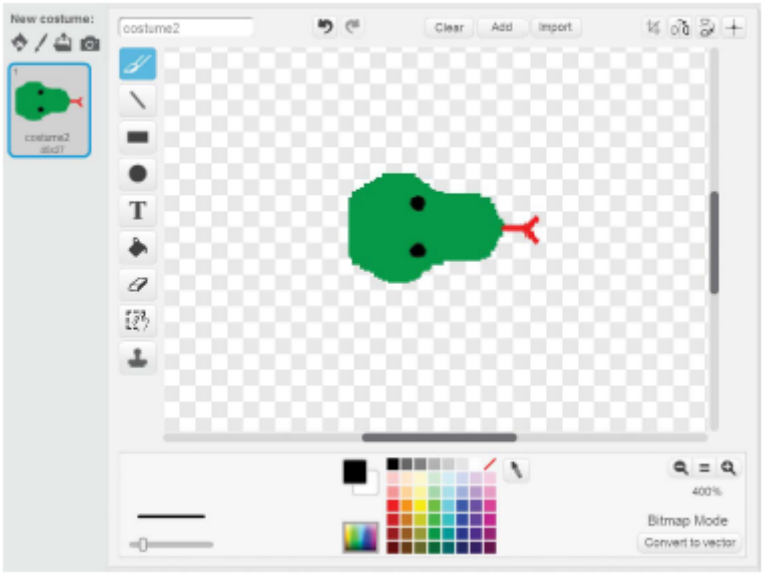
Vamos a crear nuestra versión del videojuego clásico en el que un jugador mueve una serpiente hacia unas manzanas que aparecen en el escenario. Cuantas más manzanas come, más y más larga se hace la serpiente, y más difícil es evitar que se choque contra ella misma o contra los bordes del escenario. El jugador no puede ralentizar el movimiento de la serpiente, y la partida termina cuando la serpiente se choca.



0) Añade como fondo del escenario el archivo **Proyecto 8_arena.jpg**, y utilízalo en lugar del fondo blanco.

1) Haz una cabeza de serpiente que se mueva con las teclas de las flechas.

Primero creamos la cabeza de la serpiente, utilizando el editor gráfico de Scratch. Dibuja una cabeza de serpiente parecida a la de la figura, que apunte hacia la derecha. Después de dibujarla, cambia el nombre del objeto a "cabeza".



Usa los botones de crecer y encoger de la barra de menú para aumentar o reducir el tamaño de la cabeza que has dibujado. El tamaño final debería ser similar al de la primera figura.

Ahora añadimos el código de control de la cabeza. Al clicar en la bandera verde, el programa posiciona a la cabeza en la posición $(x, y) = (0, -100)$, y la hace apuntar hacia arriba. A continuación, fija el valor de una variable "puntuación" a cero, y la serpiente comienza a moverse indefinidamente en pasos de 10 unidades. Además, el usuario debe controlar la dirección en la que apunta la cabeza serpiente, mediante las teclas de las flechas.

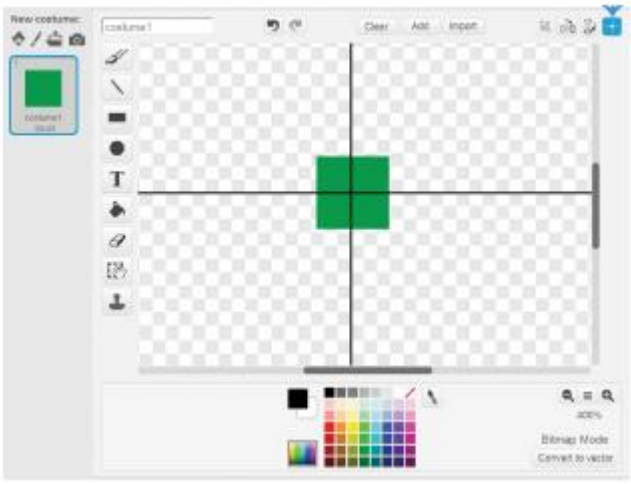
2) Haz que aparezcan y desaparezcan las manzanas.

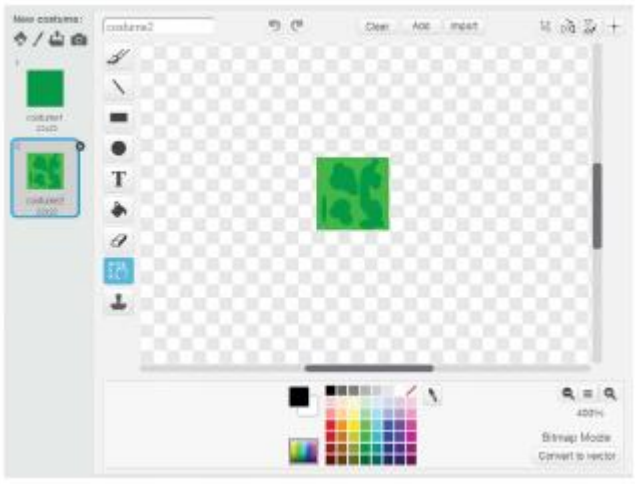
Primero, añade al proyecto el objeto "manzana" de la biblioteca de objetos de Scratch.

Crea el programa de la manzana: al clicar en la bandera verde, el programa ajusta el tamaño de la manzana a un 50% de su tamaño original, para que adquiera el aspecto mostrado en la primera figura. A continuación, la manzana comprueba indefinidamente si toca la cabeza de la serpiente. En ese caso, cambia el valor de la variable "puntuación" en 1 unidad, y desaparece de su posición actual yendo a una posición (x, y) aleatoria (x entre -210 y 210, e y entre -150 y 150).

3) Haz un cuerpo que aparezca detrás de la cabeza de la serpiente.

Acude al editor gráfico de Scratch para crear el disfraz del objeto "cuerpo". Primero, dibuja un pequeño cuadrado usando el mismo color que el de la cabeza de la serpiente. Ubica su centro en el centro del cuadrado. Renombra al disfraz como "disfraz1", y al objeto como "cuerpo"





Ahora vamos a crear un segundo disfraz para el objeto "cuerpo". Para ello, duplica el "disfraz1" clicando con el botón derecho del ratón sobre él; aparecerá un nuevo disfraz idéntico llamado "disfraz2". Cambia el color del cuadrado del disfraz2 a un verde más claro. Usaremos este verde claro para detectar si la serpiente se choca con ella misma. Si quieres, añade algunas manchas de color verde oscuro para crear una textura similar a la de la piel de una serpiente (pero sin llegar a pintar el borde exterior del cuadrado, que debe ser verde claro).

Crea el código para el cuerpo de la serpiente: Queremos que el objeto "cuerpo" siempre esté siguiendo al objeto "cabeza", y que genere clones de sí mismo.

Programa 1: este programa comienza al clicar en la bandera, y empieza cambiando al disfraz1. A continuación, se ejecuta un bucle infinito, dentro del cual el programa lleva el cuerpo a la posición de la cabeza (bloque "ir a ( )" de la categoría "movimiento"), espera $0{,}01$ segundos, y crea un clon. Con ello, estaremos generando una estela de cuerpos tras la cabeza de la serpiente.

Programa 2: El programa de los clones comienza haciendo que cada clon apunte en una dirección aleatoria entre $0°$, $90°$, $180°$, y $270°$, ver figura. (Esta rotación aleatoria hace que los segmentos del cuerpo parezcan ligeramente distintos).

A continuación, el programa espera $0{,}1$ segundos y cambia al disfraz2. Ahora debemos ir borrando los clones para que la serpiente no siga creciendo y creciendo sin medida. Para ello, el clon espera un periodo de tiempo igual a $puntuacion/5$ antes de ser borrado.

Notar que todos los clones esperan esta misma cantidad de tiempo antes de ser borrados, por lo que los primeros clones creados son los primeros en ser borrados. Además, comer manzanas incrementa el valor de la variable "puntuación", por lo que la cantidad de tiempo que espera cada clon antes de ser borrado también aumenta con el número de manzanas comidas. Este tiempo de espera más prolongado hace que la serpiente parezca más larga, porque hay más clones del cuerpo que permanecen en el escenario. Así, cuentas más manzanas coma la serpiente, más larga se hace.

4) Detecta si la serpiente se choca con ella misma o con el escenario.

Modifica el programa principal de la cabeza: (1) Si la cabeza toca el color verde claro, es porque se ha chocado contra ella misma, y en ese caso, envía el mensaje "game over". (2) Si la cabeza toca el borde del escenario, es porque se ha chocado contra él, y también envía el mensaje "game over".

El receptor del mensaje es la propia cabeza. Al recibir el mensaje "game over", la cabeza dice "Ay!" durante 2 segundos, y para el programa (bloque "detener ( )" de la categoría "control").

NOTA IMPORTANTE: ¿Recuerdas el bloque "esperar $(0{,}1)$ segundos" que hace que los clones del cuerpo esperen un poco antes de cambiar de disfraz? Notar que el nuevo clon creado cambia al disfraz2, que tiene un color verde claro. Este color es el que la cabeza usa para detectar si la serpiente se ha chocado contra ella misma. Como los clones del cuerpo se crean en la misma posición que la cabeza, están tocando a la cabeza nada más aparecer. Esta es la razón por la que queremos pausar la detección del choque cuando se crea el clon. Sin esta pausa, el objeto cabeza pensaría que se ha chocado contra el clon del cuerpo recién creado, porque estaría tocando un color verde claro.

5) Guarda el archivo como **Proyecto 8 (1).sb2**.

AMPLIACIÓN 1: Añade un segundo tipo de fruta con más puntuación.

Vamos a añadir un segundo tipo de fruta que incremente la puntuación en 3 puntos cuando la serpiente se la coma. Para ello:
a) Añade el objeto "fruit platter" de la biblioteca de Scratch.
b) Añade el código para este objeto. Necesitaremos dos programas:

Programa 1: Al clicar la bandera verde, el programa esconde el objeto, ajusta su tamaño al 50% de su tamaño original, y a continuación, arranca un bucle infinito que le hace esperar 10 segundos, lo ubica en una posición aleatoria sobre el escenario (x entre -210 y 210, e y entre -150 y 150), y lo muestra.

Programa 2: Al clicar la bandera verde, el programa comienza un bucle infinito, donde comprueba si el objeto plato de fruta está tocando la cabeza. De ser así, el programa esconde la fruta, y cambia la puntuación en tres unidades.

Guarda el archivo como **Proyecto 8 (2).sb2**.

6. TOMA DE DECISIONES.

Los programas que hemos desarrollado hasta ahora seguían un modelo de ejecución muy simple: Comenzaban con la primea instrucción, la ejecutaban, seguían con la siguiente instrucción, la ejecutaban, y continuaban así hasta que llegaban al final del programa. Los bloques de comando de este tipo de programas se ejecutan secuencialmente, sin esquivar ni saltar ningún bloque.

Sin embargo, en muchas situaciones es necesario alterar este flujo secuencial en la ejecución de un programa. Por ejemplo, en una aplicación para enseñar matemáticas básicas a los niños, probablemente querremos hacer cosas distintas si el usuario acierta o falla al dar su respuesta. En este capítulo exploraremos los bloques que permiten la toma de decisiones, y crearemos programas que usan estos bloques para comprobar datos de entrada y realizar diferentes acciones en función de esa comprobación.

6.1. OPERADORES DE COMPARACIÓN.

Nosotros tomamos decisiones a diario, y cada decisión normalmente conlleva la realización de una acción. Por ejemplo, podemos pensar: "Si este coche vale menos de 6.000 €, lo compro". Entonces, preguntamos el precio del coche, y decidimos si lo compramos o no.

Scratch también permite la toma de decisiones. Usando **operadores de comparación**, podemos comparar los valores de dos variables o expresiones para determinar si una es mayor que, menor que, o igual que la otra. (A los operadores de comparación también se los llama *operadores relacionales*, porque comprueban las relaciones existentes entre dos valores). Los tres operadores relacionales soportados por Scratch se muestran en la figura:

Operador	Significado	Ejemplo
	mayor que	 ¿Es el precio mayor que 6000?
	menor que	 ¿Es el precio menor que 6000?
	igual que	 ¿Es el precio igual a 6000?

Observar que todos los bloques relacionales de la tabla tienen forma hexagonal. Como recordaremos del capítulo 5, eso significa que el resultado de evaluar uno de esos bloques es un valor *booleano*, el cual puede ser VERDADERO o FALSO. Por eso, a estas expresiones también se las denomina *expresiones booleanas*. Por ejemplo, la expresión "precio < 6000" comprueba si el valor de la variable "precio" es menor que 6000. Si resulta que "precio" es menor que 6000, el bloque devuelve el valor VERDADERO, y en caso contrario, devuelve el valor FALSO. Así, podemos usar esta expresión para construir una condición de decisión de la forma: "Si (precio < 6000), entonces compro el coche".

Pero antes de estudiar el bloque "si ( ) entonces", vamos a analizar la forma en la que Scratch evalúa las expresiones booleanas.

Booleanos en el mundo real.

La palabra *booleano* se usa en honor al matemático del siglo XIX George Boole, que inventó un sistema algebraico basado exclusivamente en dos valores: 0 y 1 (o VERDADERO y FALSO). El álgebra booleana terminaría convirtiéndose en la base de la ciencia informática moderna. En la vida real, estamos continuamente usando expresiones booleanas para tomar decisiones. Los ordenadores también las usan para determinar qué rama de un programa van a seguir. Por ejemplo, los sistemas de seguridad domésticos suelen programarse para hacer sonar una alarma si el código de seguridad insertado es erróneo (codigoCorrecto = FALSO), o para desactivar la alarma si el código insertado es correcto (codigoCorrecto = VERDADERO). El ordenador a bordo de nuestro vehículo activará automáticamente los airbag cuando detecte que ha ocurrido una colisión (colisión = VERDADERO). Nuestro teléfono móvil puede mostrar un icono cuando el nivel de batería sea bajo (bateríaBaja = VERDADERO), y quitar el icono cuando el nivel de carga de la batería sea aceptable (bateríaBaja = FALSO).

EVALUAR EXPRESIONES BOOLEANAS.

Digamos que fijamos dos variables x e y a los valores $x = 5,\ y = 10$. La tabla a continuación muestra algunos ejemplos que usan los bloques relacionales de Scratch.

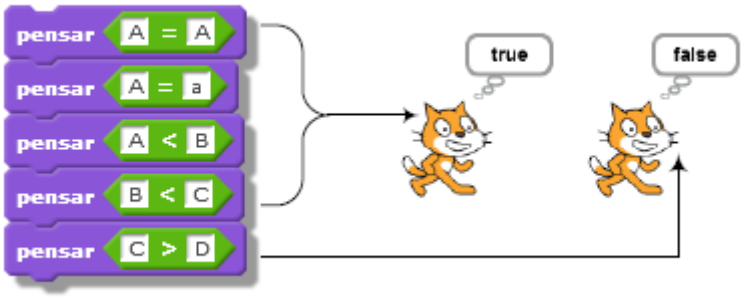
Statement	Meaning	z (output)	Explanation
	z = is(5 < 10)?	z = true	because 5 is less than 10
	z = is(5 > 10)?	z = false	because 5 is not more than 10
	z = is(5 = 10)?	z = false	because 5 is not equal to 10
	z = is(10 > 2*5)?	z = false	because 10 is not more than 10
	z = is(5 = 5)?	z = true	because 5 is equal to 5
	z = is(10 < 5 + 6)?	z = true	because 10 is less than 11

Estos ejemplos revelan algunos hechos importantes: (1) Los bloques relacionales sirven para comparar tanto variables individuales como expresiones completas. (2) El resultado de una comparación siempre es VRDADERO (true) o FALSO (false), es decir, un valor booleano. (3) El bloque "$x = y$" no significa "fija $x = y$"; en realidad, pregunta "¿es x igual a y?". Así, cuando se ejecutamos "fijar z a $(x = y)$" el valor de x sigue siendo 5.

COMPARAR LETRAS Y CADENAS.

Imagina un juego en el que el usuario debe adivinar una letra secreta entre la A y la Z. El juego lee la apuesta del usuario, la compara con la letra secreta, y le indica al usuario que debe refinar su conjetura basándose en el orden alfabético. Por ejemplo, si la letra secreta fuese la G, y el usuario conjeturase la B, el juego debería decir algo así como "Después de la B". ¿Cómo podríamos comparar la letra secreta con la letra insertada por el usuario para decidir qué mensaje mostrar?

Los operadores relacionales de Scratch también permiten comparar letras. Como muestra la figura, Scratch compara letras basándose en su orden alfabético. Como la letra A viene antes que la B en el alfabeto, la expresión $A < B$ devuelve VERDADERO. (Sin embargo, es importante notar que Scratch no diferencia entre mayúsculas y minúsculas: la letra A es la misma que la letra a; por consiguiente, la expresión $A = a$ también devuelve VERDADERO).



Sabido esto, podemos comprobar la apuesta del usuario usando el siguiente conjunto de condicionales:

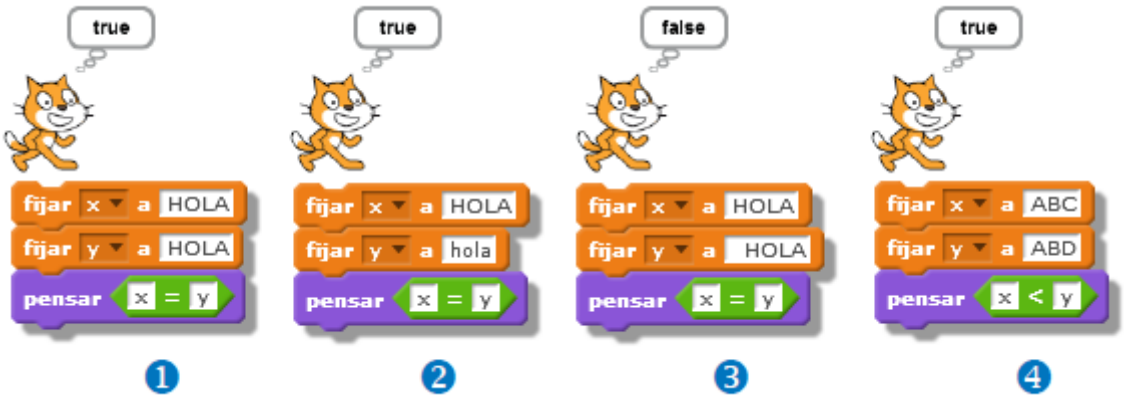
$$\text{SI } (respuesta = letraSecreta), \text{ENTONCES } decir\ Correcto$$

$$\text{SI } (respuesta > letraSecreta), \text{ENTONCES } decir\ Antes\ que\ \langle respuesta \rangle$$

$$\text{SI } (respuesta < letraSecreta), \text{ENTONCES } decir\ Despues\ que\ \langle respuesta \rangle$$

Un **condicional** es un enunciado de la forma "Si la condición es cierta, entonces realiza esta acción". En la siguiente sección veremos cómo construir condicionales en Scratch. Pero ahora, vamos a explicar los operadores relacionales un poco más.

¿Y si el código secreto contiene más de una letra? Por ejemplo, puede que el juego consista en adivinar el nombre de un animal. Afortunadamente, también podemos usar los operadores relacionales de Scratch para comparar cadenas. Ahora bien, ¿cómo procesa Scratch una comparación como $elefante > ratón$? La figura ilustra la situación:



Un cuidadoso estudio de las situaciones mostradas en la figura muestra que:

- Scratch compara cadenas de caracteres sin diferenciar entre mayúsculas y minúsculas (ver caso 2).
- Scratch considera los espacios en blanco en sus comparaciones (ver caso 3).
- Al comparar las cadenas "ABC" y "ABD" (ver caso 4), Scratch primero considera el primer carácter en las dos cadenas. Como en este caso ambos son iguales (letra A), Scratch examina el segundo carácter en ambas cadenas. Al volver a ser iguales (letra B), Scratch sigue con el tercer carácter, y como la letra C "es menor" que la letra D, Scratch considera que la primera cadena es menor que la segunda.

Por lo tanto, deberíamos ver que la expresión $elefante > ratón$ se evaluará como FALSO, a pesar de que los elefantes son más grandes que los ratones. Según las reglas de comparación de cadenas en Scratch, la cadena "elefante" es menor que la cadena "ratón", porque la letra e viene antes que la letra r en el alfabeto.

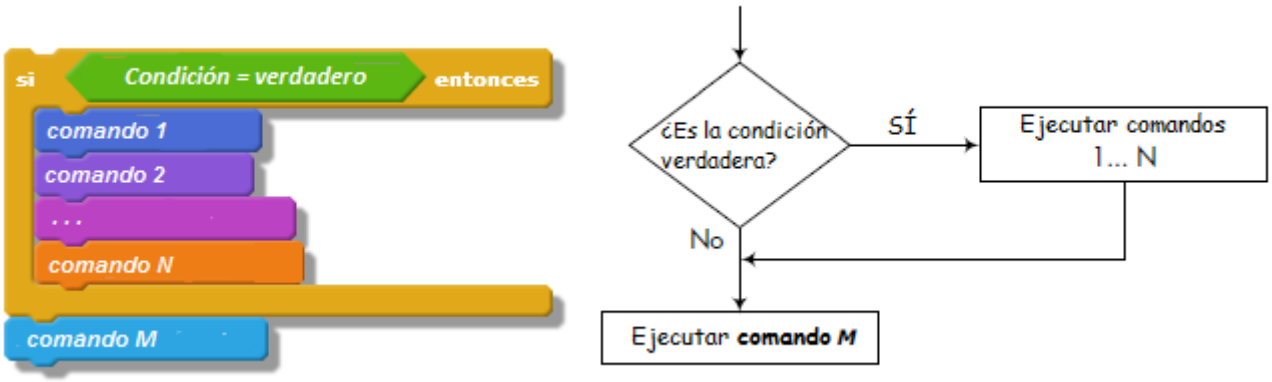
6.2. ESTRUCTURAS DE DECISIÓN.

Ahora que ya entendemos qué son los operadores relacionales, y cómo los usa Scratch para comparar números y cadenas, vamos a aprender a usar los bloques condicionales.

La categoría "control" de Scratch contiene dos bloques que nos permiten tomar decisiones y controlar las acciones que realizan nuestros programas: el bloque "si ( ) entonces" y el bloque "si ( ) entonces / si no". Usando estos bloques, podemos formular una pregunta, y realizar acciones basadas en la respuesta.

EL BLOQUE "SI ( ) ENTONCES".

El bloque "si ( ) entonces" es una estructura de decisión que nos permite especificar si una serie de comandos debería o no ejecutarse, dependiendo de si se cumple o no una condición (ver figura).



En la figura, la forma hexagonal dentro del bloque "si ( ) entonces" representa un bloque de decisión que proporciona una respuesta SI/NO (o VERDADERO/FALSO) a una pregunta formulada. Si la condición dentro del "si ( ) entonces" es verdadera, el programa ejecuta los comando contenidos dentro del cuerpo antes de seguir con el comando que viene tras el bloque "si ( ) entonces" (el comando M en la figura). Si la condición es falsa, el programa se salta esos comandos y va directamente al comando M.

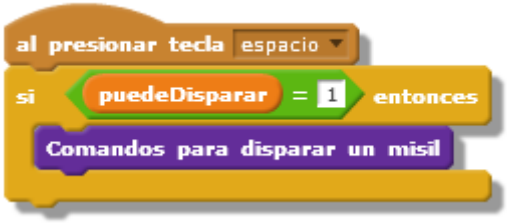
A modo de ejemplo, ejecuta el programa mostrado en la figura. Este programa ejecuta un bucle "por siempre" que mueve el objeto por el escenario, cambia su color *si* está en la mita derecha del escenario, y lo hace rebotar en los bordes.

USAR VARIABLES COMO INDICADORES.

Imagina que estamos creando un juego de combates espaciales donde el objetivo es destruir una flota de naves enemigas. El jugador controla una nave con las teclas de las flechas y dispara misiles con la tecla espaciadora. Si la nave del jugador recibe un cierto número de disparos, pierde su capacidad de disparar. En ese caso, al presionar la tecla espaciadora la nave no debería disparar más misiles. Evidentemente, al presionar la tecla espaciadora, el programa debe verificar el estado del sistema de disparo de la nave para decidir si el jugador puede o no disparar.

Este tipo de comprobaciones suelen realizarse mediante **indicadores** (flags), que son variables que se usan para indicar si ha ocurrido o no un evento de interés. Se pueden usar dos valores cualesquiera para describir el estado de un evento, pero es práctica habitual usar el 0 (o FALSO) para indicar que ese evento no ha ocurrido, y el 1 (o VERDADERO) para indicar que sí se ha producido.

En nuestro juego, podemos usar un indicador llamado "puedeDisparar" para indicar el estado de la nave. Un valor de 1 significa que la nave puede disparar misiles, y un valor de 0 que no puede. En base a esto, el programa que gestiona la tecla espaciadora sería algo como:



Al principio del juego, deberíamos inicializar el valor de la variable "puedeDisparar" a 1, para indicar que al comienzo de la partida la nave puede disparar misiles. Cuando la nave sea alcanzada un cierto número de veces, deberíamos ajustar el valor de "puedeDisparar" a 0, para indicar que el sistema de ataque de la nave ya no es operativo. En ese punto, presionar la tecla espaciadora ya no dispararía ningún misil.

Aunque podemos llamar a nuestros indicadores como queramos, es recomendable darles nombres que reflejen su naturaleza booleana (verdadero / falso). La tabla muestra algunos ejemplos.

Example	Meaning and Possible Course of Action
fijar juegoComenzado a 0	El juego aún no ha comenzado. Ignorar cualquier tecla.
fijar juegoComenzado a 1	El juego ya ha comenzado. Empezar a precesar las teclas presionadas.
fijar finalDePartida a 0	La partida aún no ha terminado. Mostrar el tiempo restante.
fijar finalDePartida a 1	La partida ya ha terminado. Ocultar el display con el tiempo restante.
fijar alcanceDetectado a 0	La nave no ha sido alcanzada por el enemigo. No hacer sonar la alarma.
fijar alcanceDetectado a 1	La nave ha sido alcanzada. Hacer sonar la alarma.

EL BLOQUE "SI ( ) ENTONCES / SI NO"

Imagina un juego para enseñar matemáticas básicas a los niños. El juego presenta por pantalla una suma, y le pide al usuario que introduzca la respuesta. El usuario recibe un punto por una respuesta correcta, y pierde un punto por una respuesta incorrecta. Esto podríamos hacerlo con dos sentencias SI:

SI *respuesta correcta*, ENTONCES *sumar un punto al marcador*

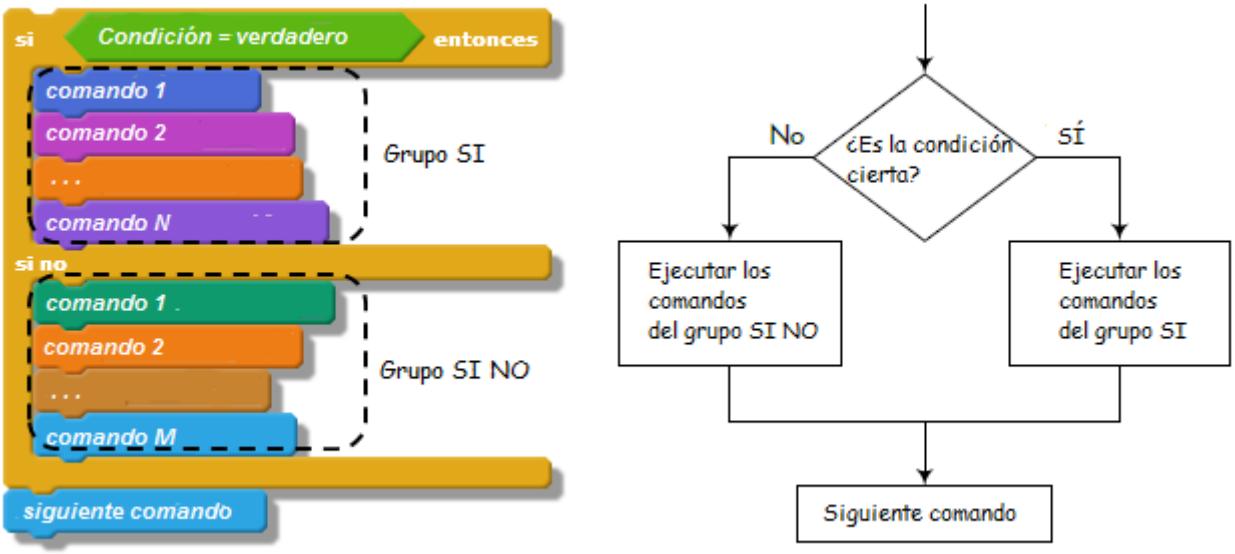
SI *respuesta incorrecta*, ENTONCES *restar un punto al marcador*

Pero también podemos simplificar esta lógica combinando las dos sentencias SI en una sola sentencia SI / SI NO:

SI *respuesta correcta*, ENTONCES
sumar un punto al marcador

SI NO
restar un punto al marcador

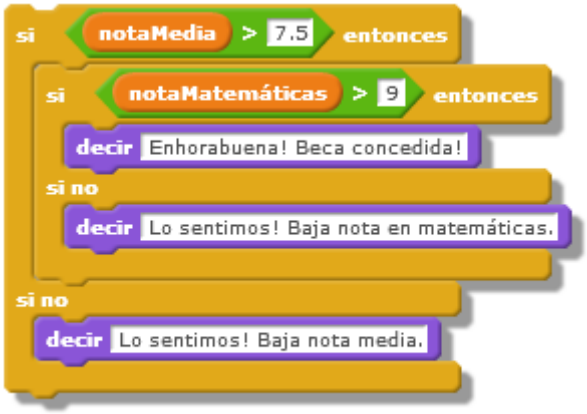
La estructura de un bloque "si ( ) entonces / si no" y su diagrama de flujo se muestran en la figura. La condición se comprueba una sola vez. Si esta condición es cierta, se ejecutan los comandos de la parte SI del bloque. Si la condición es falsa, se ejecutan los comandos de la parte SI NO. El programa solo ejecutará uno de los dos grupos de comandos del bloque "si / si no". A estas rutas alternativas en el flujo de ejecución de un programa también se las denomina *ramas*.



La figura muestra un ejemplo sencillo del uso del bloque "si ( ) / si no". Este programa usa el operador "( ) mod ( )" (el módulo, esto es, el resto de una división) para determinar si el número introducido es par o impar.

BLOQUES "SI" Y "SI / SI NO" ANIDADOS.

Si queremos comprobar más de una condición antes de realizar una acción, podemos anidar varios bloques "si" (o bloques "si / si no") unos dentro de otros para efectuar la comprobación múltiple deseada. Así, por ejemplo, consideremos el programa de la figura, el cual determina si un estudiante debería recibir una beca. Para recibirla, el estudiante debe tener una nota media mayor que 7,5 y una nota por encima de 9 en matemáticas:

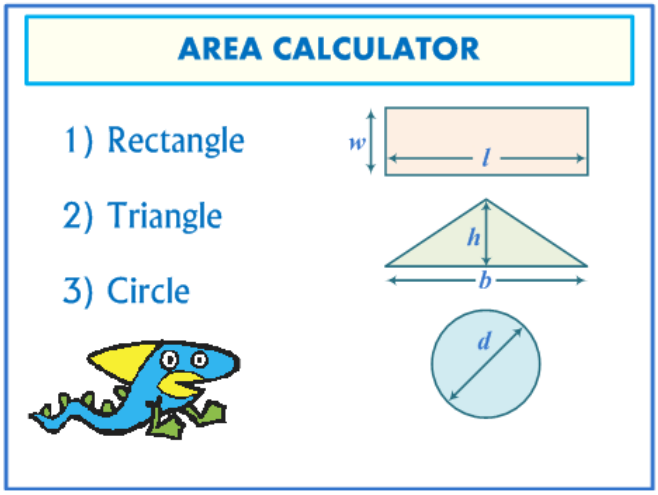


PROGRAMAS ACCIONADOS POR MENÚ.

En esta sección vamos a estudiar un uso muy habitual de los bloques "si" anidados: vamos a aprender cómo escribir programas que presentan al usuario múltiples opciones y que actúan en función de la opción elegida.

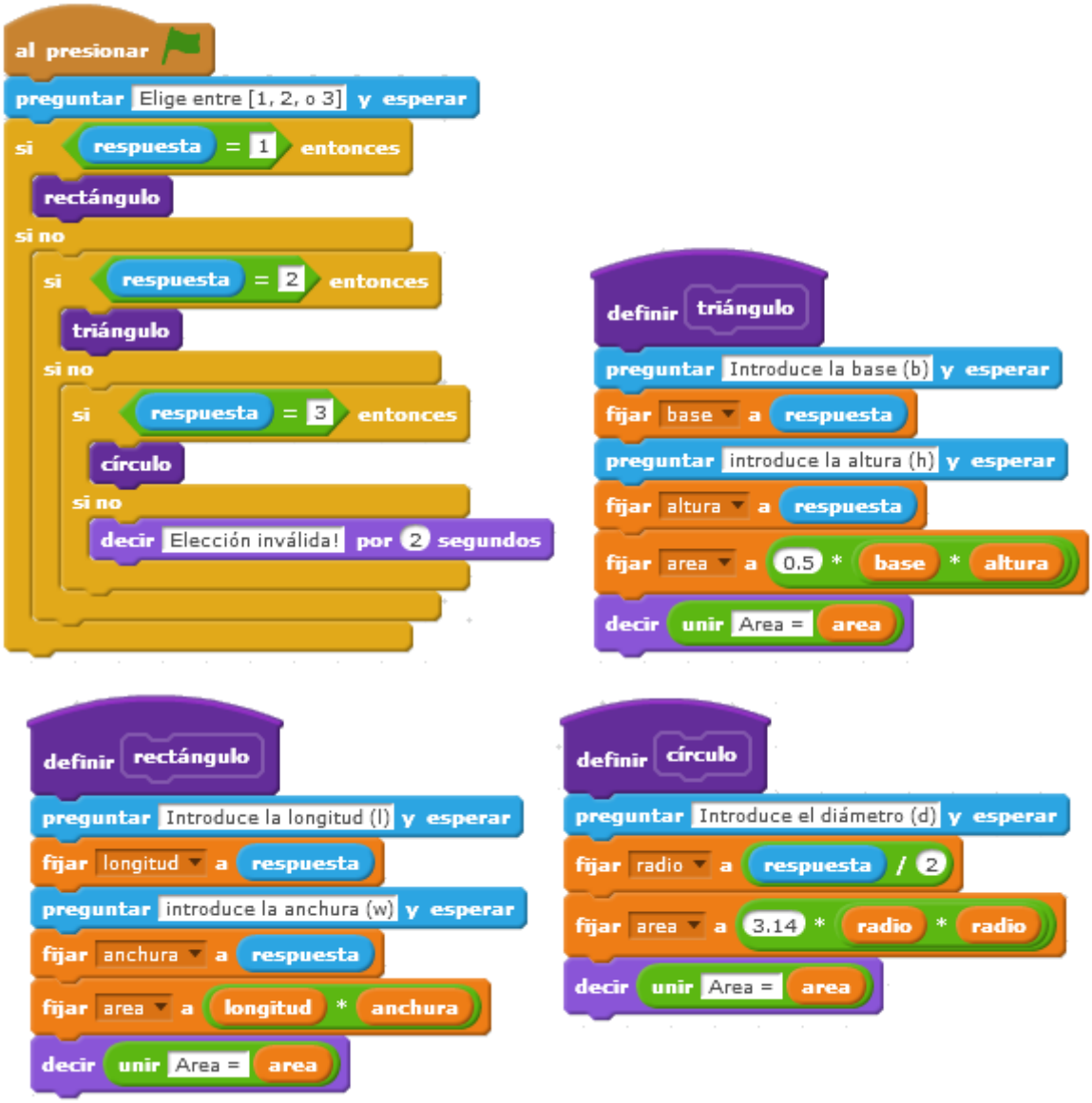
Como sabemos, al arrancar algunos programas, estos comienzan mostrando una lista (o menú) de opciones disponibles, y esperan a que hagamos una elección. A veces, el usuario interactúa con estos programas insertando un número que se corresponde con la opción deseada. Estos programas usan una secuencia de bloques "si / si no" anidados para determinar la elección del usuario y actuar en consecuencia.

A modo de ejemplo, vamos a discutir un programa que calcula el área de diferentes formas geométricas: (1) un rectángulo, (2) un triángulo, y (3) un círculo.



Abre el archivo **AreaCalculator.sb2**. La interfaz de usuario de esta aplicación contiene una imagen de fondo con las opciones disponibles (1, 2, y 3), y el objeto "tutor", que le pide al usuario una elección, realiza el cálculo, y muestra el resultado. El programa principal (del objeto "tutor"), y los procedimientos a los que llama, se muestran en la siguiente figura.

Después de pedirle al usuario que elija entre 1, 2, o 3, el objeto "tutor" espera a que el usuario haga una elección, y usa tres bloques "si ( ) / si no" para procesarla. Si el usuario introdujo una elección válida (esto es, un número entre 1, 2, y 3), el programa llama al procedimiento apropiado para calcular el área de la figura elegida.



6.3. OPERADORES LÓGICOS.

En la sección previa aprendimos a usar bloques "si" o "si / si no" anidados para comprobar múltiples condiciones, pero eso también podemos hacerlo con los **operadores lógicos**.

Operator	Meaning
y	El resultado es VERDADERO si y solo si las dos expresiones son verdaderas.
o	El resultado es VERDADERO si una de las dos (o las dos) expresiones son verdaderas.
no	El resultado es VERDADERO si la expresión es falsa.

Usando operadores lógicos podemos combinar dos o más expresiones relacionales para producir un solo resultado de tipo verdadero/falso. Por ejemplo, la expresión lógica $(x > 5)\ y\ (x < 10)$ está formada por dos expresiones lógicas que están combinadas con el operador lógico Y (and). Así, podemos pensar en $(x > 5)$ y

en $(x < 10)$ como los dos operandos del operador Y; el resultado de esta operación es VERDADERO si y solo si ambos operandos son verdaderos. La tabla muestra los tres operadores lógicos disponibles en Scratch con una pequeña explicación de su significado.

EL OPERADOR Y.

El operador Y (and) recibe dos expresiones como parámetros. Si ambas expresiones son verdaderas, el operador Y devuelve VERDADERO; en otro caso, devuelve FALSO.

X	Y	X and Y
true	true	true
true	false	false
false	true	false
false	false	false

A modo de ejemplo, digamos que estamos creando un juego en el que el jugador obtiene 200 puntos extra cuando su puntuación alcanza los 100 puntos en el primer nivel. El nivel en el que está el juego se controla mediante una variable llamada "nivel", y la puntuación mediante otra variable llamada "puntuación". La figura muestra cómo comprobar estas condiciones (1) usando bloques "si" anidados, y (2) con el operador Y. Como vemos, el operador Y proporciona una forma más concisa de realizar la misma comprobación.

EL OPERADOR O.

El operador O (or) también recibe dos expresiones como parámetros. Si cualquiera de las dos es verdadera, el operador O devuelve VERDADERO. Solo devuelve FALSO si las dos expresiones son falsas.

X	Y	X or Y
true	true	true
true	false	true
false	true	true
false	false	false

A modo de ejemplo, suponer que los jugadores de un juego tienen un tiempo limitado para llegar al siguiente nivel. También comienzan con una cierta cantidad de energía que se agota conforme recorren el nivel actual. El juego termina si el jugador no consigue llegar al siguiente nivel en el tiempo permitido, o si agota toda su energía antes de llegar al siguiente nivel. El tiempo disponible se controla con una variable llamada "tiempo", y la energía actual del jugador mediante la variable "energía". La figura muestra cómo comprobar

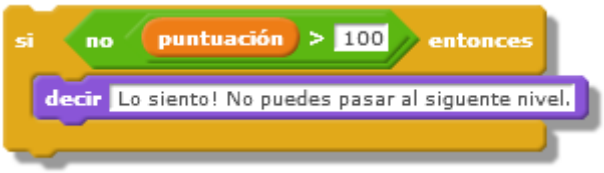
si la partida ha terminado usando (1) bloques "si / si no" anidados, y (2) con el operador O. Notar de nuevo que el operador O proporciona una forma más concisa de testear múltiples condiciones.

EL OPERADOR NO.

El operador NO (not) sólo recibe una expresión como entrada. El resultado del operador es VERDADERO si la expresión es falsa, y FALSO si la expresión es verdadera.

X	not X
true	false
false	true

Volviendo al ejemplo previo, digamos que el jugador no puede pasar al siguiente nivel si la puntuación no es mayor que 100 puntos. Esta sería una buena oportunidad para usar el operador NO (ver figura). Este programa puede entenderse de la siguiente forma: "Si la puntuación no es mayor que 100, ejecutar los comandos dentro del bloque "si"".



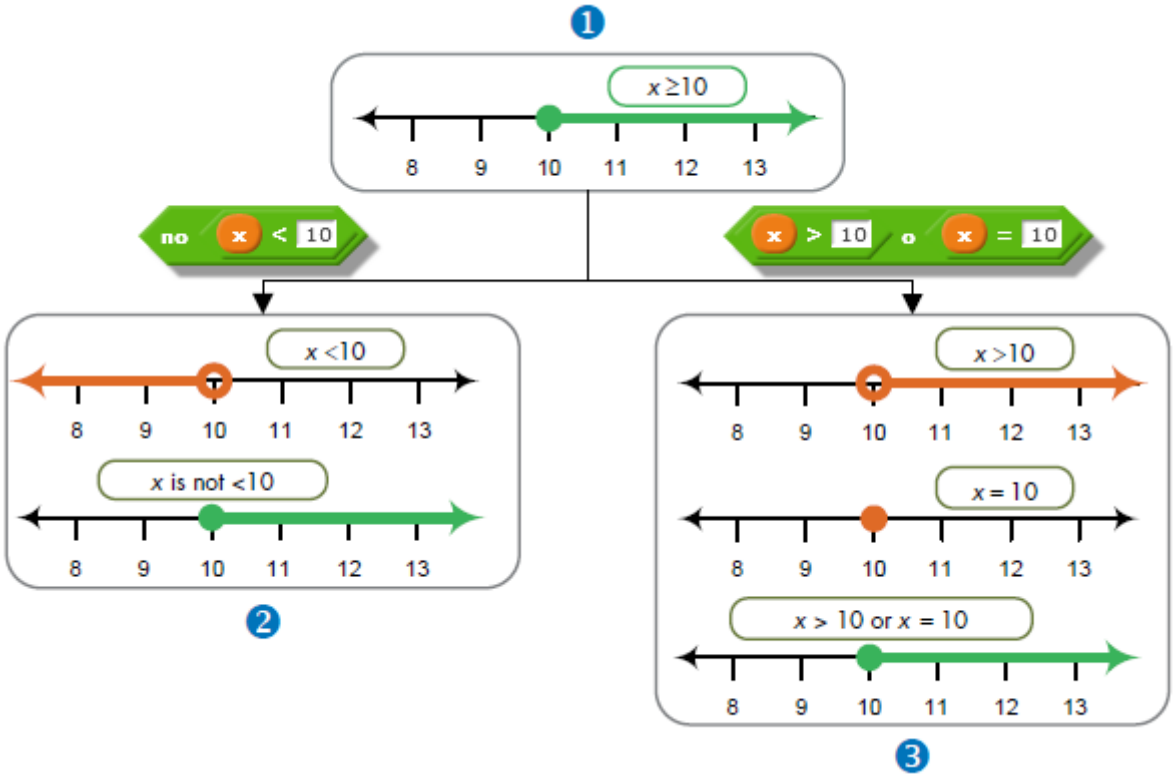
En efecto, si el valor de la variable "puntuación" es 100 o menor, la expresión lógica se evalúa a verdadero, y entonces se ejecuta el comando "decir". Notar que la expresión lógica "no (puntuación > 100)" es equivalente a la expresión "puntuación < 100".

USAR OPERADORES LÓGICOS PARA COMPROBAR RANGOS NUMÉRICOS.

Para validar los datos numéricos introducidos por el usuario, o filtrar datos de entrada erróneos, podemos usar operadores lógicos para determinar si un número está dentro o fuera de un cierto rango numérico.

Expression	Value
(x > 10) and (x < 20)	Evaluates to true if the value of **x** is greater than 10 and less than 20.
(x < 10) or (x > 20)	Evaluates to true if the value of **x** is less than 10 or greater than 20.
(x < 10) and (x > 20)	Always false. **x** can't be both less than 10 and greater than 20.

Aunque Scratch no ofrece soporte directo para manejar los operadores ≥ (mayor o igual que) y ≤ (menor o igual que), podemos usar los operadores lógicos estudiados antes para implementar estas comprobaciones. Por ejemplo, imaginar que queremos comprobar la condición $x \geq 10$ en nuestro programa. La figura muestra las distintas formas de hacerlo:



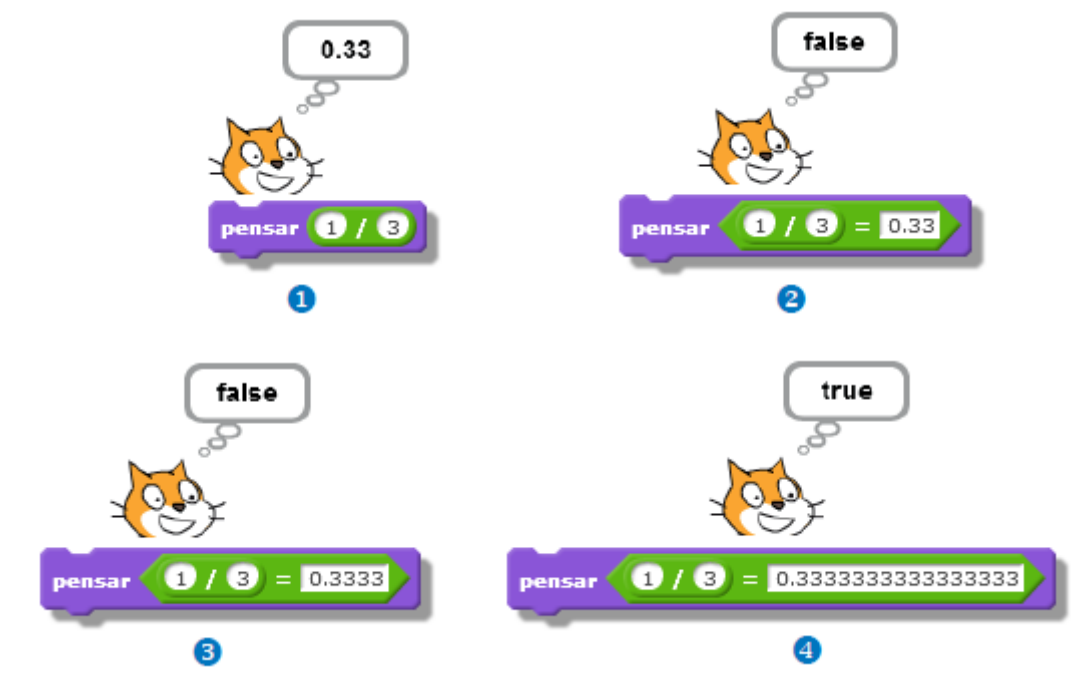
La siguiente tabla muestra más ejemplos de cómo usar los operadores lógicos y relacionales de Scratch para expresar rangos que contengan los operadores ≥ ó ≤.

Expression	Implementation Using Logical Operators
$x \geq 10$	no x < 10
$x \geq 10$	x > 10 o x = 10
$x \leq 10$	no x > 10
$x \leq 10$	x < 10 o x = 10
$10 \leq x \leq 20$	no x < 10 y no x > 20
$10 \leq x \leq 20$	x > 10 o x = 10 y x < 20 o x = 20

Comparar números decimales.

Debemos tener mucho cuidado al usar el operador "igual a" al comparar números decimales. Debido a la forma en la que estos números se almacenan en la memoria del ordenador, la comparación puede ser imprecisa.

A modo de ejemplo, considera los bloques mostrados en la figura. El resultado de dividir $1/3$ es $0,\hat{3} = 0{,}3333\ldots$ (es decir, un decimal periódico puro con infinitos cifras decimales). Como los ordenadores usan una cantidad de espacio fija para almacenar el resultado, la fracción $1/3$ no puede representarse de forma exacta en un ordenador. Aunque en (1) Scratch nos dice que el resultado de la división es $0{,}33$, el valor almacenado internamente en realidad tiene una precisión mucho mayor. Por consiguiente, los resultados de las dos primeras comparaciones (2 y 3 en la figura) evalúan a FALSO. Solo la última comparación (expresando el resultado con 16 decimales) evalúa a VERDADERO.



Dependiendo de la situación, podemos usar uno de los siguientes métodos para evitar este tipo de errores:

- Usar los operadores "menor que" (<) y "mayor que" (>) en lugar del operador "igual a" (=) cuando sea posible.
- Usar el bloque "redondear ( )" para redondear los dos números que queremos comparar, y entonces comparar la igualdad de los números redondeados.
- Testea la diferencia absoluta entre los dos valores que estás comparando. Por ejemplo, en lugar de comprobar si $x = y$, podemos chequear si la diferencia absoluta entre x e y está dentro de una tolerancia aceptable, usando un bloque similar a:

6.4. EJERCICIOS SCRATCH.

EJERCICIO 33. NOTAS DESTACADAS.

Escribe un programa que le pida al usuario 5 notas de entre 1 y 10. El programa contará el número de notas que son mayores o iguales que 7.

EJERCICIO 34. EL MAYOR DE TRES.

Escribe un programa que le solicite al usuario tres números. El programa determinará e indicará por pantalla cuál es el mayor de los tres.

EJERCICIO 35. PRECIO DE VENTA.

Una empresa vende 5 tipos de productos cuyos precios de venta son los mostrados en la tabla. Escribe un programa que le pida al usuario el número de producto y la cantidad vendida. El programa calculará y mostrará por pantalla el precio de venta total.

Product Number	Retail Price
1	$2.95
2	$4.99
3	$5.49
4	$7.80
5	$8.85

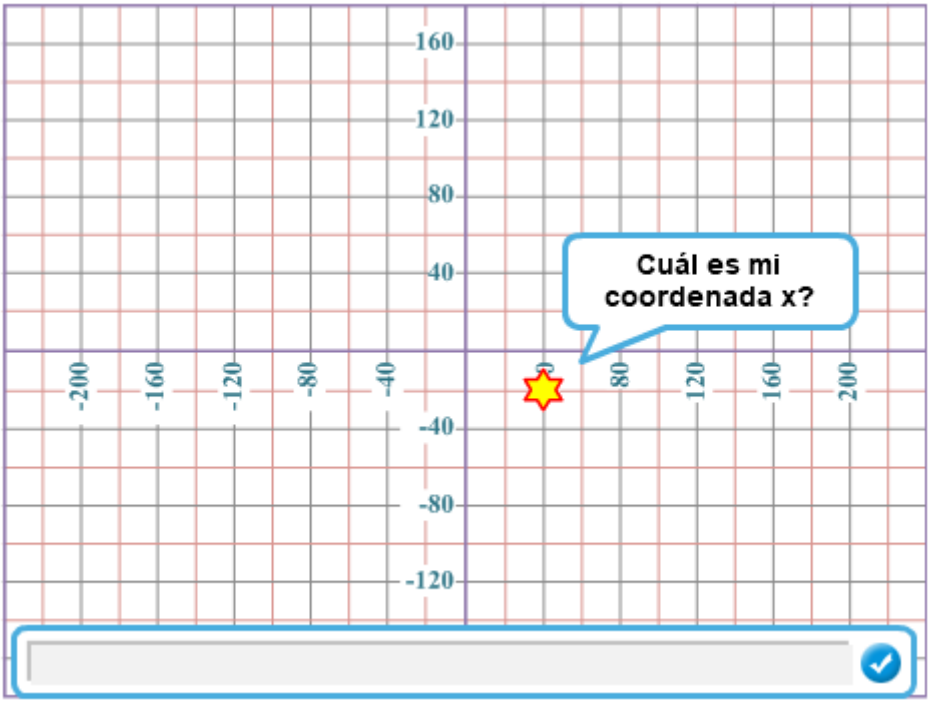
EJERCICIO 36. TERNAS PITAGÓRICAS.

Como sabemos, el teorema de Pitágoras afirma que si a y b son las longitudes de los catetos de un triángulo rectángulo y c es la longitud de la hipotenusa (el lado más largo), se cumple que $a^2 + b^2 = c^2$. Escribe un programa que reciba tres números del usuario y que determine si esos tres números podrían representar los tres lados de un triángulo rectángulo. Comprueba tu programa para $a = 3,\ b = 4$, y $c = 5$.

6.5. PROYECTOS SCRATCH.

PROYECTO 9. ADIVINA MIS COORDENADAS.

Vamos a desarrollar un juego para comprobar el conocimiento del usuario de las coordenadas Cartesianas (x, y). El juego contiene un objeto estrella ("star") que se mueve a un punto aleatorio sobre el escenario cada vez que se ejecuta el programa. Entonces, el programa le pregunta al usuario las coordenadas del objeto. El juego chequea la respuesta y da un mensaje de acierto o error.



Abre el archivo **Proyecto 9_sinCodigo.sb2**. Este archivo incluye el objeto "estrella", y un fondo con los ejes x e y convenientemente numerados. Además, el proyecto ya incluye las variables "X", "Y", y "punto".

Vamos a crear el programa de la estrella:

1) El programa comienza al presionar la bandera verde. Lo primero que hace es elegir las coordenadas x e y de la estrella de forma aleatoria. La coordenada x se elige al azar entre los valores $\{-220, -200, -180, \ldots, 220\}$. (Esto puede conseguirse seleccionando aleatoriamente un entero entre -11 y 11, y multiplicando el resultado por 20). De forma similar, la coordenada y se elige aleatoriamente entre los valores $\{-160, -140, -120, \ldots, 160\}$. (Piensa tú como obtener este conjunto de valores posibles). Estos valores aseguran que la estrella caerá en una de las intersecciones del mallado del escenario. Los valores aleatorios para las coordenadas x e y se almacenan en las variables "X" e "Y", respectivamente.

2) A continuación, el programa envía a la estrella a la posición aleatoria (X, Y) obtenida en el paso previo.

3) El programa le pregunta al usuario cuál es la coordenada x de la estrella, y espera su respuesta. Si el valor proporcionado es el correcto, el programa se mueve al paso 4. Si el valor proporcionado es incorrecto, el programa llama al procedimiento "respuestaCorrecta", para mostrar las coordenadas correctas de la estrella.

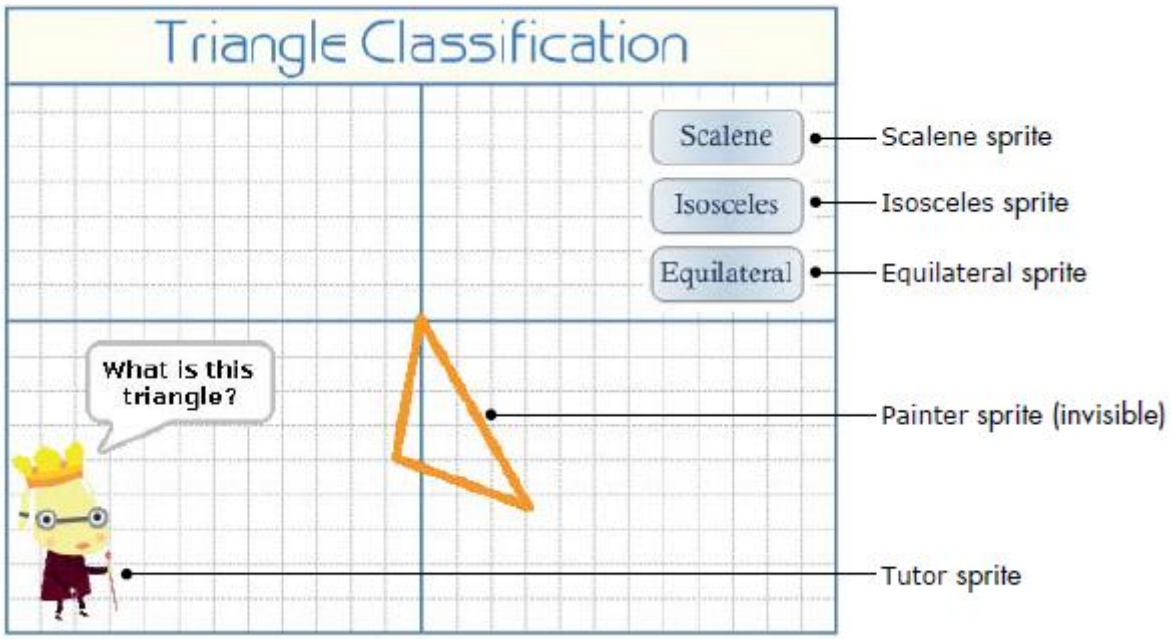
4) Si la coordenada x introducida por el usuario es correcta, el programa le pregunta por el valor de la coordenada y, y espera su respuesta. Si el valor de la coordenada y también es correcto, el programa muestra el mensaje "¡Bien hecho!". Si el valor de la coordenada y es incorrecto, el programa llama al procedimiento "respuestaCorrecta".

5) El procedimiento "respuestaCorrecta" simplemente fija el valor de la variable "punto" al valor (X, Y), siendo "X" e "Y" las variables que almacena las coordenadas x e y donde se localiza la estrella. (Probablemente necesitarás hacer uso de varios bloques "unir"). A continuación, el procedimiento simplemente dice "Has fallado. La respuesta correcta es: " e incluye el valor de la variable "punto".

6) Comprueba si el programa funciona correctamente y guárdalo como **Proyecto 9.sb2**.

PROYECTO 10. JUEGO DE CLASIFICACIÓN DE TRIÁNGULOS.

Dependiendo de la longitud de sus lados, los triángulos se clasifican en escalenos, isósceles, y equiláteros. En este proyecto desarrollaremos un juego que pregunte al usuario sobre estos conceptos.



El juego dibuja un triángulo y le pregunta al jugador que especifique de qué tipo es. El interfaz de usuario del juego está disponible en el archivo **Proyecto 10_sinCodigo.sb2**. El archivo incluye 5 objetos. Los objetos "escaleno", "isósceles", y "equilátero" son botones que el usuario presiona para elegir la respuesta correcta, y el objeto dibujante (que es invisible) dibuja el triángulo en pantalla. El objeto "tutor" es el que controla el juego: determina qué triángulo hay que dibujar cada partida, y comprueba la respuesta del usuario. Empezaremos creando el código del objeto "tutor":

1) Al clicar en el icono de la bandera verde, el programa principal entra en un bucle infinito, que comienza fijando la variable "selección" a 0 (para indicar que el usuario no ha respondido todavía), muestra una nueva pregunta (llamando al procedimiento "nuevaPregunta", ver paso 2), espera la respuesta del usuario (bucle "esperar hasta que ("selección" > 0)", ver paso 5), y terminada la espera, comprueba la respuesta (llamando al procedimiento "comprobarRespuesta", ver paso 3).

2) El procedimiento "nuevaPregunta" comienza fijando el valor de la variable "tipo" a un número al azar entre 1, 2, y 3, para determinar aleatoriamente qué tipo de triángulo se dibujará en pantalla. A continuación, el programa asigna a la variable "nombre" el valor "escaleno" si la variable "tipo" es igual a 1, el valor "isósceles" si "tipo" es igual a 2, y el valor "equilátero" sin "tipo" es igual a 3. Después envía un mensaje con el contenido de la variable "nombre" y queda a la espera. Este mensaje le dirá al objeto dibujante qué tipo de triángulo debe dibujar, ver paso 4. Para terminar, el programa formula la pregunta "¿Qué tipo de triángulo es este?", para pedirle al usuario que especifique el tipo del triángulo dibujado, clicando en el botón adecuado.

3) El procedimiento "comprobarRespuesta" es muy sencillo. Simplemente toma la variable "tipo" (el número al azar que usamos para determinar el tipo de triángulo que se dibujó en pantalla) y comprueba si es igual a la variable "selección" (donde almacenaremos el botón que presione el usuario para especificar el tipo de triángulo mostrado en pantalla). Si son iguales, el tutor dirá "¡Muy bien!" durante 2 segundos. En caso contrario, el tutor dirá "¡No! Es un triángulo "nombre"". (Recordemos que, en la variable "nombre", se almacena el nombre del tipo de triángulo que se ha dibujado por pantalla).

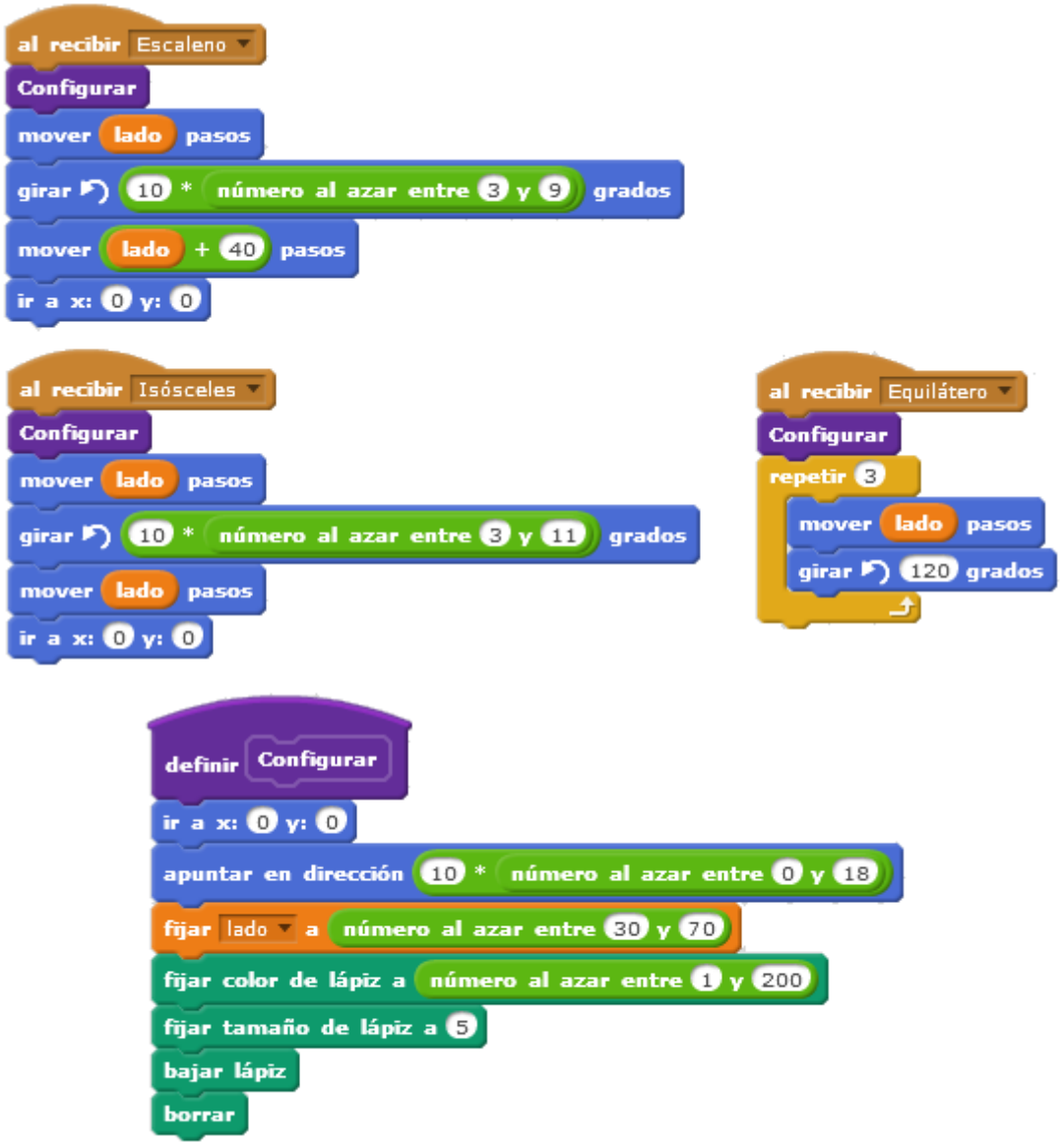
4) Ahora, vamos con el código para el objeto dibujante. Recordemos que el objeto "tutor" determina aleatoriamente qué tipo de triángulo debe dibujarse en pantalla. El objeto "tutor" envía un mensaje con el contenido "escaleno", "isósceles", o "equilátero", que recibe el objeto dibujante para saber qué triángulo debe dibujar. Así pues, el código del objeto dibujante incluye tres programas, cada uno de los cuales sirve para dibujar un triángulo escaleno, isósceles, o equilátero. Todos estos programas comienzan llamando al procedimiento "configurar", que se encarga de fijar (aleatoriamente) la orientación, tamaño, y color del triángulo dibujado. El código completo del objeto dibujante se muestra en la figura. (Notar que el algoritmo para dibujar cada tipo de triángulo es distinto en cada caso).

5) Después de que el objeto tutor le indique al objeto dibujante qué triángulo debe mostrar en pantalla (procedimiento "nuevaPregunta"), y de que el triángulo efectivamente se dibuje, el tutor queda en espera hasta que el usuario clica en uno de los botones de respuesta (bloque "esperar hasta que ("selección" > 0)"), momento en el que el valor de la variable "selección" se actualiza a 1 (escaleno), 2 (isósceles), o 3 (equilátero). Los objetos botón se encargan de actualizar este valor. Por ejemplo, el programa del objeto botón "escaleno" comienza cuando el usuario clica en ese botón, y simplemente ajusta el valor de la variable "selección" a 1. (El botón "isósceles" fijará "selección" a 2, y el botón "equilátero" fijará "selección" a 3).

Una vez que el correspondiente botón actualiza la variable "selección" a un valor > 0, el programa del tutor finaliza su espera y llama al procedimiento "comprobarRespuesta", del que ya hemos hablado antes.

AMPLIACIÓN:

- Haz que el juego lleve la puntuación. Añade un punto cada vez que el usuario acierte, y resta un punto cada vez que falle.
- Añade una opción para que el usuario pueda quitar el juego.
- Define un criterio para la finalización del juego. Por ejemplo, puedes sustituir el bucle infinito del programa principal del tutor por un bucle "repetir" que se ejecute, digamos, 20 veces. También podemos terminar el juego si el usuario responde erróneamente en 3 ocasiones.
- Haz que ocurra algo interesante mientras se está ejecutando del juego. Por ejemplo, puedes crear una variable llamada "númeroEspecial" y asignarle un valor aleatorio al comenzar el juego. Cuando el número de respuestas correctas se haga igual a este número, el juego podría dar puntos extra, reproducir una canción, hacer que el tutor cuente un chiste o haga un baile gracioso, etc.
- Aplica a los botones algún tipo de efecto visual cuando el puntero se deslice sobre ellos, o cuando cliquemos sobre ellos.

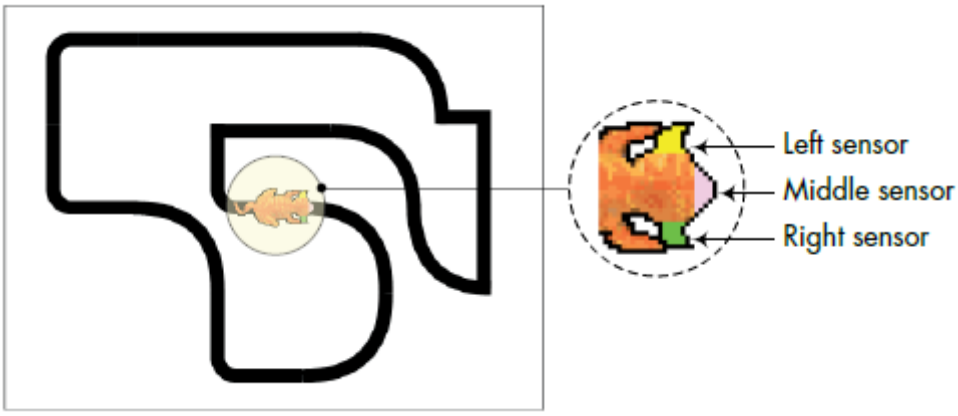


PROYECTO 11. SEGUIDOR DE LÍNEA.

En este proyecto vamos a hacer que un objeto siga un camino negro trazado sobre el escenario. Abre el archivo Proyecto 11_sinCodigo.sb2, donde encontrarás un fondo con el camino negro ya dibujado, y un objeto gato un tanto especial: En efecto, si miramos de cerca el objeto gato, nos daremos cuenta de que la nariz y las dos orejas están pintadas con colores diferentes (ver figura).

La estrategia será usar la nariz y las orejas del gato como sensores para detectar la línea negra bajo ellos. El algoritmo para seguir la línea negra será el siguiente:

- Si la nariz del gato (color rosa) está tocando la línea negra, el gato avanza hacia adelante.
- Si la oreja izquierda del gato (color amarillo) está tocando la línea negra, el gato gira en contra de las agujas del reloj (hacia su izquierda), y se mueve adelante a una rapidez reducida.
- Si la oreja derecha del gato (color verde) está tocando la línea negra, el gato gira a favor de las agujas del reloj (hacia su derecha), y se mueve adelante a una rapidez reducida.



Por supuesto, la rapidez exacta de avance (el número de pasos que se mueve el gato) y los ángulos de giro pueden ser diferentes, dependiendo del camino a seguir, y deben fijarse mediante prueba y error. En el camino negro de la figura, la rapidez de avance si la nariz toca la línea es de 2 pasos, mientras que la rapidez de avance si las orejas tocan la línea es de $0{,}5$ pasos. En ambos casos, el giro hacia la derecha o hacia la izquierda es de 5 grados.

El programa del gato comienza al clicar en la bandera verde, y empieza ubicando al gato en una zona de la línea que discurra hacia la derecha (ver figura). A continuación, hace que el gato apunte hacia la derecha, y de forma indefinida, aplica el algoritmo que hemos explicado antes. (PISTA: La aplicación de este algoritmo requiere la utilización de tres bloques "si / si no" anidados)

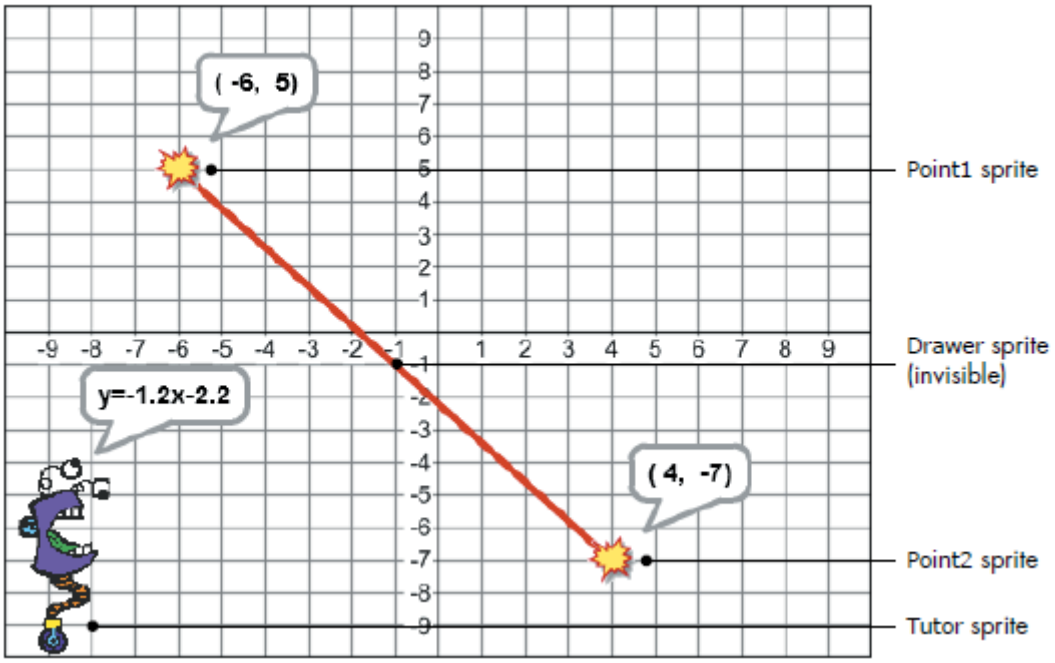
Para construir este programa, necesitaremos usar un nuevo comando: el bloque "color ( ) tocando ( )" de la categoría "sensores". Este bloque comprueba si un determinado color del objeto (que se especifica en el primer espacio en blanco) está tocando otro color (especificado en el segundo espacio en blanco). Si el color del objeto está tocando el segundo color especificado, el bloque devuelve VERDADERO; en caso contrario, devuelve FALSO. Estos dos colores pueden elegirse clicando en el espacio en blanco correspondiente del bloque, y después, clicando en cualquier punto del entorno Scratch para seleccionar el color deseado.

PROYECTO 12. ECUACIÓN DE UNA LÍNEA.

La ecuación de la línea que une los dos puntos $P = (x_1, y_1)$ y $Q = (x_2, y_2)$ es $y = mx + b$, donde $m = (y_2 - y_1)/(x_2 - x_1)$ es la pendiente de la línea y b es la denominada "ordenada en el origen". Una línea vertical tiene una ecuación de la forma $x = k$, y una línea horizontal tiene una ecuación de la forma $y = k$, donde k es una constante.

En este proyecto desarrollaremos una aplicación que halle la ecuación de la línea que une dos puntos en el plano Cartesiano. El archivo **Proyecto 12_sinCodigo.sb2** incorpora la interfaz de usuario de la aplicación. El usuario arrastra sobre el escenario dos objetos que representan dos puntos de la línea, y la aplicación muestra automáticamente la ecuación de la línea que los conecta. La aplicación contiene 4 objetos: "punto1"

y "punto2" (los dos puntos de la recta), el objeto dibujante (un objeto oculto que dibuja una línea recta entre esos dos puntos), y el objeto "tutor" (el responsable de computar y mostrar la ecuación de la línea).



El código de los dos puntos es muy similar. Ambos objetos utilizan una lógica que restringe las localizaciones de los objetos a los puntos de intersección de la rejilla en la pantalla. Básicamente, cuando el usuario arrastra el objeto "punto1", el programa actualiza las variables "X1" e "Y1" que almacenan sus coordenadas y envía el mensaje "redibujar". De forma similar, cuando el usuario arrastra el objeto "punto2", el programa actualiza las variables "X2" e "Y2" que alojan sus coordenadas y envía el mismo mensaje. Las cuatro variables "X1", "Y1", "X2", e "Y2" solo pueden tomar valores enteros en el rango de -9 a 9. Por su dificultad, los programas de los puntos ya están construidos en el archivo Proyecto 12_sinCodigo.sb2.

Vamos ahora con el código para el objeto dibujante ("drawer"):

1) Cuando el juego comienza al clicar en la bandera verde, el dibujante fija el color del lápiz (a rojo), fija el tamaño del lápiz (4), baja el lápiz, limpia la pantalla, y se oculta.

2) Cuando el objeto dibujante recibe el mensaje "redibujar", va a la posición del objeto "punto1", limpia la pantalla, y después va a la posición del objeto "punto2". De esta forma, dibuja una línea recta que conecta "punto1" y "punto2".

Continuamos con el código del objeto "tutor":

1) El objeto "tutor" también ejecuta un programa al recibir el mensaje "redibujar". (1) Primero, comprueba si las coordenadas x e y de los dos puntos son las mismas. En ese caso no hay línea que procesar, y el objeto simplemente dice "mismo punto". (2) Si los dos puntos son diferentes pero sus coordenadas x son iguales tenemos una línea vertical, y el programa muestra una ecuación de la forma $x = constante$. (3) Si los dos puntos son diferentes pero sus coordenadas y son iguales tenemos una línea horizontal, y el programa muestra una ecuación de la forma $y = constante$. (4) En caso contrario, los dos puntos forman una línea recta cuya ecuación tiene la forma $y = mx + b$. Entonces, el programa llama al procedimiento "computar" para hallar la pendiente m y la ordenada en el origen b, y a continuación, llama al procedimiento "mostrarEcuación" para poner la ecuación en un formato adecuado y mostrársela por pantalla al usuario.

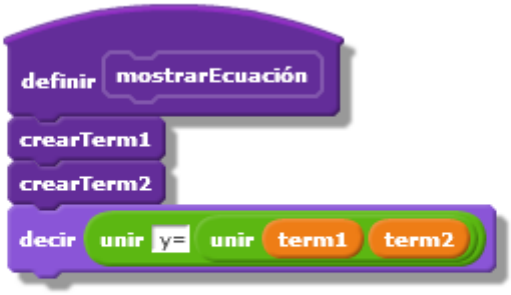
2) El procedimiento "computar" comienza calculando la pendiente m y la ordenada en el origen b:

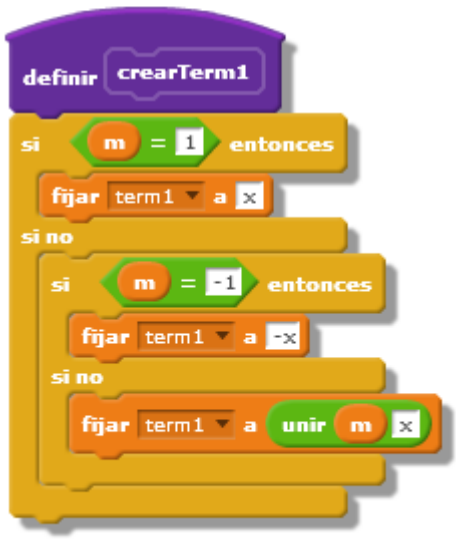
$$m = \frac{(y_2 - y_1)}{(x_2 - x_1)} \qquad b = y_1 - mx_1$$

A continuación, toma esos valores calculados y los redondea a la centésima parte más cercana:

3) El procedimiento "mostrarEcuación" usa dos variables ("term1" y "term2"), y los subprocedimientos correspondientes ("crearTerm1" y "crearTerm2") para formatear adecuadamente la ecuación a mostrar (ver figura). El procedimiento "mostrarEcuación" considera los siguientes casos especiales al formatear la ecuación de la línea:

- Si la pendiente es 1, ajusta "term1" a x (en vez de $1x$).
- Si la pendiente es -1, ajusta "term1" a $-x$ (en vez de $-1x$).
- "term2" se formatea usando el signo apropiado (más o menos) de la ordenada en el origen.
- Si la ordenada en el origen es 0, la ecuación tendrá la forma $y = mx$.



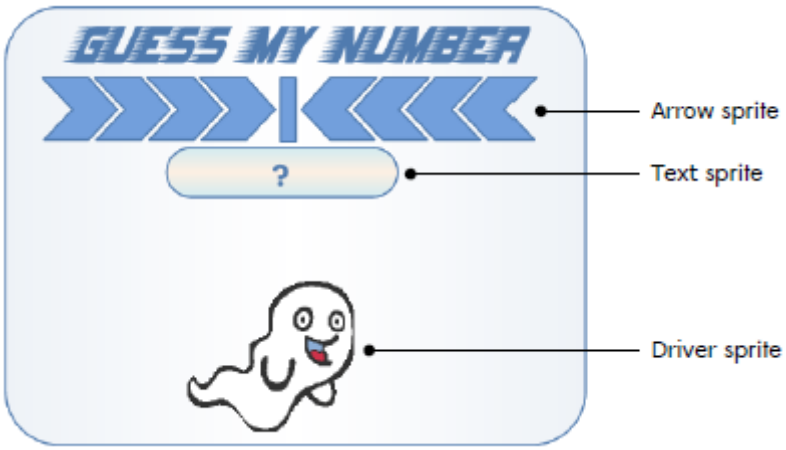


6.6. PROYECTOS LIBRES.

Ahora ya sabes lo suficiente para poder realizar una serie de proyectos libres que te permitirán poner en práctica tus conocimientos de programación en Scratch. Al contrario que hasta ahora, estos proyectos ya no están dirigidos, y deberás ser tú quien desarrolle libremente el código necesario para hacer funcionar la aplicación.

PROYECTO LIBRE 1. ADIVINA EL NÚMERO.

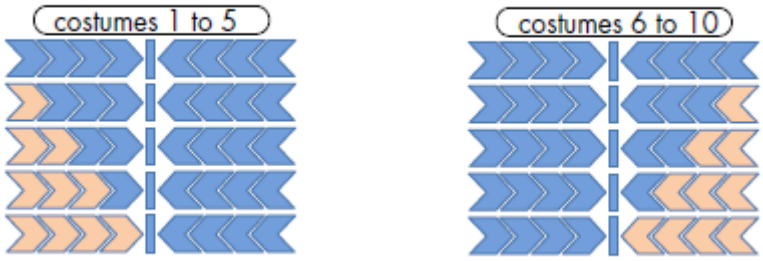
Esta aplicación elige aleatoriamente un entero entre 1 y 100 y le pide al usuario que adivine ese número. Cuando el usuario responde, el programa le dice si su conjetura es mayor o menor que el número secreto, mostrando por pantalla "demasiado alto" o "demasiado bajo". El jugador dispone de 6 oportunidades para adivinar el número secreto. Si el usuario consigue acertar el número, gana la partida; en otro caso, la pierde. La interfaz de esta aplicación está contenida en el archivo **Libre 1_sinCodigo.sb2**.



El objeto "presentador" (driver) controla el flujo de la aplicación. El objeto "flecha" (arrow) proporciona una información animada sobre cómo ajustar nuestra siguiente apuesta. El objeto "texto" (text) muestra el estado del juego tras cada nueva apuesta.

El juego comienza al clicar la bandera verde. En respuesta, el objeto "presentador" obtendrá el número secreto aleatorio, y en 6 ocasiones (como máximo), le pedirá al usuario su apuesta, y la comparará con el número secreto. Si el usuario acierta (antes de la 6ª apuesta), envía el mensaje "ganó" y queda en espera a que el objeto receptor del mensaje termine su tarea para finalizar la partida. Si la apuesta del usuario es mayor que el número secreto, el "presentador" envía el mensaje "alto" y queda en espera a que el objeto receptor termine su tarea, y en caso contrario (esto es, si la apuesta es menor que el números secreto) envía el mensaje "bajo" y queda en espera a que el objeto receptor realice su tarea. Cuando el usuario agota sus 6 intentos sin acertar el número secreto, el programa envía el mensaje "perdió" y queda en espera. Tras volver de esta última espera, el programa termina.

El objeto "texto" dispone de 5 disfraces ("?", "demasiado alto", "demasiado bajo", "ganaste", y "perdiste"). El primero es su disfraz por defecto, el segundo es el que se pone si la apuesta del usuario es demasiado alta, el tercero es el apropiado si la apuesta es demasiado baja, y los dos restantes son los disfraces que se pone cuando el usuario gana o pierde la partida. Programa al objeto texto para que se ponga el disfraz adecuado en cada ocasión (al clicar la bandera verde, al recibir uno u otro mensaje, etc.).



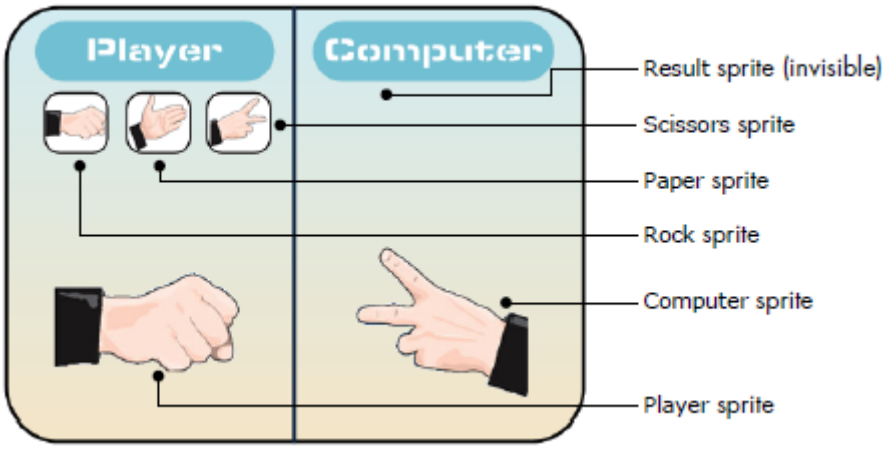
Por último, el objeto "flecha" tiene 10 disfraces, para realizar un efecto de animación tras cada apuesta errónea. Debemos programarlo de forma que comience por defecto con el disfraz1. Si la apuesta del usuario es demasiado baja (recepción del mensaje "bajo"), debemos animar la flecha para que la parte

izquierda del objeto se ilumine de izquierda a derecha (disfraces 1 a 5) varias veces, indicando que el número secreto es mayor que el que el usuario ha conjeturado. Pero si recibe el mensaje "alto", debemos animar la parte derecha de la flecha para que se ilumine de derecha a izquierda (disfraces 6 a 10) varias veces, indicando que el número secreto es menor que el que el usuario ha conjeturado.

Con esto has terminado. Guarda el proyecto como **Libre 1.sb2**.

PROYECTO LIBRE 2. PIEDRA, PAPEL, O TIJERA.

En este proyecto crearás un juego de piedra, papel, o tijeras, en el que el usuario juega contra el ordenador. La interfaz de este juego está disponible en el archivo **Libre 2_sinCodigo.sb2**.

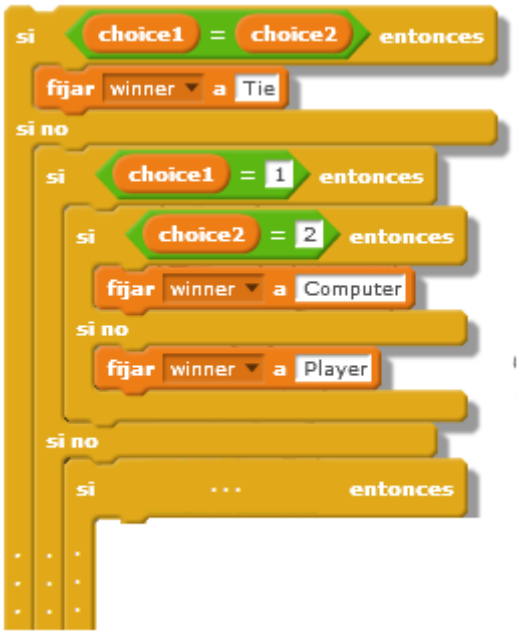


El juego empieza cuando el usuario clica en uno de los tres botones de piedra (rock), papel (paper), o tijeras (scissors). Cada botón fija el valor de la variable "choice1" al valor que identifica la elección del usuario ($piedra = 1$, $papel = 2$, y $tijeras = 3$), y a continuación, envía el mensaje "empezar" para decirle al resto de objetos que el usuario ya ha elegido.

El mensaje "empezar" lo recibe el escenario, que envía el mensaje "nueva ronda" y queda a la espera. Este mensaje sirve para indicarles a los objetos "jugador" (player) y "ordenador" (computer) que muestren sus respectivas selecciones. A continuación, llama al procedimiento "comprobarResultado" para determinar cuál de los dos ha ganado esta ronda. Después, envía el mensaje "mostrar resultado" y queda a la espera, hasta que el objeto "resultado" (result), que es el receptor del mensaje, muestre el resultado de la partida por pantalla. Finalmente, el escenario envía el mensaje "game over" y queda a la espera. Este mensaje indica al resto de objetos que deben prepararse para la próxima ronda.

El mensaje "nueva ronda" lo reciben los seis objetos del proyecto. Los objetos "piedra", "papel", "tijeras", y "resultado" simplemente se ocultan al recibirlo. Por su parte, los objetos "jugador" y "ordenador" comienzan ejecutando una serie de 5 rápidos cambios de disfraz (un simple efecto visual divertido), y a continuación, muestran el disfraz correspondiente a su elección. Mientras que el disfraz de "jugador" viene dado por la variable "choice1" (cuyo valor lo fija el botón que el usuario haya clicado), el disfraz que selecciona el "ordenador" se elige aleatoriamente (un número al azar entre 1 y 3), y ese dato se almacena en la variable "choice2".

Una vez que el jugador y el ordenador han hecho y mostrado su elección, y como ya hemos mencionado antes, el escenario llama al procedimiento "comprobarResultado". Mediante una serie de condicionales anidados, este procedimiento compara los valores de "choice1" y "choice2 y fija el valor de la variable "winner" como corresponda: empate (tie), jugador (player), u ordenador (computer). A modo de ejemplo, la figura muestra una parte de este conjunto de condicionales anidados. (Recuerda que 1 se corresponde con piedra, 2 con papel, y 3 con tijeras).



Después de comprobar el resultado y ajustar el valor de la variable "winner", el escenario envía el mensaje "mostrar resultado". Este mensaje lo recibe el objeto "resultado", que muestra uno de sus tres disfraces de acuerdo con el valor de la variable "winner". (Por supuesto, el objeto "resultado" comienza la partida oculto, para ponerse el disfraz adecuado solo al final de la ronda).

Finalmente, cuando el objeto "resultado" termina su tarea, el escenario envía el mensaje "game over" y espera. En respuesta a este mensaje, los tres objetos botón vuelven a mostrarse, y el objeto "resultado" vuelve a ocultarse. Así, el juego queda listo para la próxima ronda.

Con esto hemos terminado. Guarda el proyecto como **Libre 2.sb2**.

AMPLIACIÓN 1: Modifica el juego para llevar la cuenta de cuántas rondas gana el jugador, cuántas gana el ordenador, y cuántas terminan en empate.

AMPLIACIÓN 2: Modifica el proyecto para convertirlo en un juego para dos jugadores. Por ejemplo, un jugador puede realizar su elección con las teclas 1, 2, y 3 del teclado, y el otro con las teclas ←, ↑, y →.

7. MÁS SOBRE BUCLES.

Recordemos que las *estructuras repetitivas*, también llamadas *bucles*, son comandos de programación que le dicen a un ordenador que ejecute repetidamente una instrucción o un conjunto de instrucciones. Los bucles más sencillos son los *bucles definidos* (o *bucles controlados por contador*), los cuales repiten una secuencia de instrucciones un número de veces determinado. Otros tipos de bucles se repiten hasta que ocurre una cierta condición; estos bucles se denominan *bucles indefinidos* o *bucles controlados por condición*. Otros bucles, llamados *bucles infinitos*, se repiten para siempre.

En este capítulo aprenderemos las diferentes estructuras de repetición disponibles en Scratch, y presentaremos el bloque "detener" para terminar bucles infinitos. Además, aprenderemos a usar bucles para validar datos introducidos por el usuario. Este capítulo también discutirá los *bucles anidados*, y tratará la *recursividad* (el mecanismo mediante el cual un procedimiento se llama a sí mismo) como forma alternativa de conseguir la repetición.

7.1. OTROS BUCLES DE SCRATCH.

Como vimos en el capítulo 2, los bucles permiten repetir la ejecución de una instrucción o de un conjunto de instrucciones en un programa. Scratch soporta los tres bloques de repetición mostrados en la figura:

Ya hemos usado dos de estos bloques, el bloque "repetir ( )" y el bloque "por siempre". En esta sección presentaremos el tercer bloque de repetición, el bloque "repetir hasta que ( )", y también explicaremos algunos términos técnicos de todos los bucles en general.

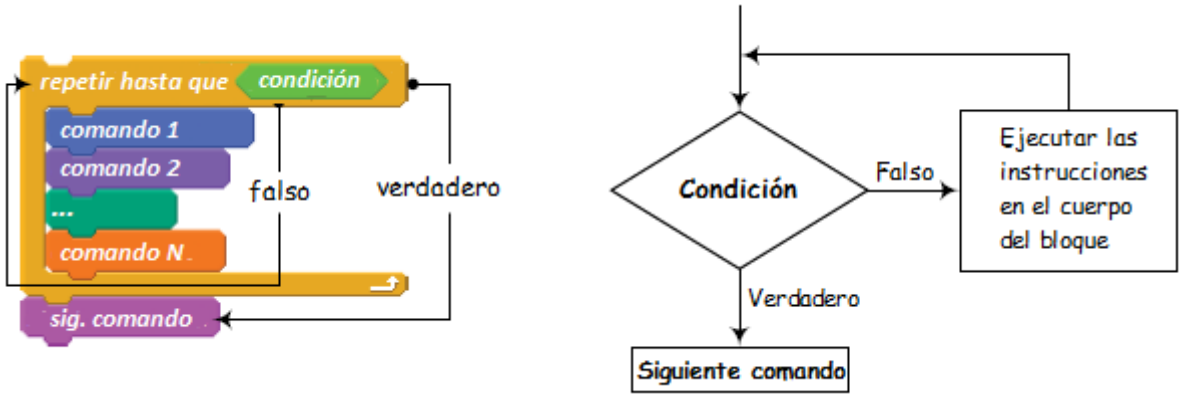
Cada repetición de un bucle se denomina una **iteración**, y a menudo, el término **cuenta** se usa para describir el número de veces que se repite un bucle. El bucle "repetir ( )" es un bucle controlado por contador porque repite las instrucciones que contiene un determinado número de veces. Este bucle lo usamos cuando conocemos el número de veces que debe repetirse el bucle, como por ejemplo, cuando queremos dibujar un polígono con un cierto número de lados.

Por otro lado, el bloque "repetir hasta que ( )" es un bucle controlado por condición. Las instrucciones dentro del bucle se repiten o no dependiendo de la veracidad o falsedad de la condición bajo prueba. Este bloque lo usamos cuando no sabemos cuántas veces debe repetirse el bucle, y solo sabemos que queremos que la repetición continúe hasta que se cumpla una cierta condición. Por ejemplo, podríamos decir "repite el comando "preguntar" hasta que el usuario introduzca un entero positivo", o también, "repite el disparo de proyectiles hasta que el nivel de energía del jugador caiga por debajo de un cierto valor".

EL BLOQUE "REPETIR HASTA QUE ( )".

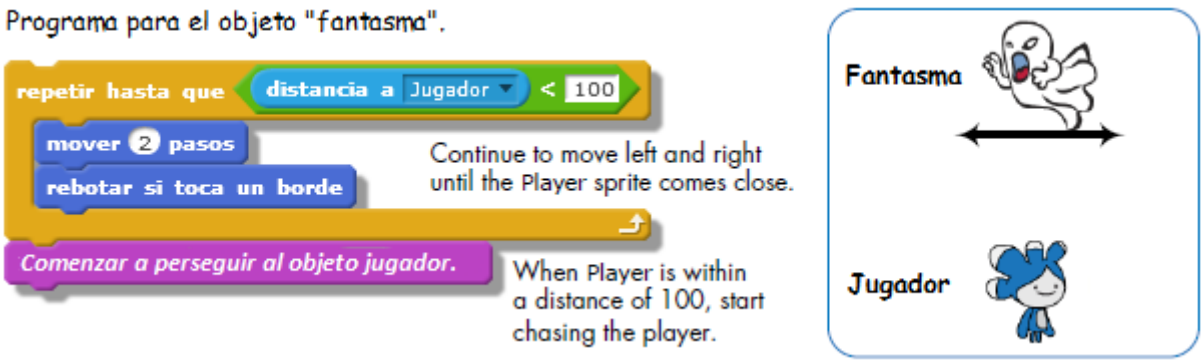
Imagina que estamos desarrollando un juego que le plantea al usuario una pregunta matemática básica. Si la respuesta es incorrecta, el juego vuelve a hacer la misma pregunta para darle al jugador otra oportunidad más. En otras palabras, el juego pregunta la misma cuestión *hasta* que el jugador introduce la respuesta correcta. Evidentemente, el bloque "repetir ( )" no es el apropiado para esta tarea, porque no sabemos cuántas oportunidades necesitará el jugador para introducir la respuesta correcta. El bloque "repetir hasta que ( )" es el adecuado en esta situación. La figura muestra la estructura del bloque "repetir hasta que ( )".

Este bloque contiene una expresión booleana cuyo valor se comprueba en la entrada del bloque. Si la expresión es falsa, se ejecutan las instrucciones dentro del bucle. Tras ejecutar el último comando dentro del bucle, éste comienza otra vez, y la expresión booleana se comprueba de nuevo. Si la expresión sigue siendo falsa, las instrucciones dentro del bucle se ejecutan una vez más. Este bucle se repite hasta que la expresión booleana se hace verdadera. Cuando esto ocurre, los comandos dentro del bucle ya no se ejecutan más, y el programa continúa con el comando que va justo después del bucle.



Notar que si la condición booleana ya es cierta cuando el programa ejecuta el bucle por primera vez, los comandos dentro del bucle no se ejecutarán. Notar también que el bucle no terminará hasta que algún comando (ya sea dentro del bucle o en alguna otra parte activa de la aplicación) haga que la condición booleana se vuelva verdadera. Si el resultado de la condición nunca se vuelve verdadero, entraremos en un bucle infinito.

La siguiente figura muestra un ejemplo práctico de uso del bloque "repetir hasta que ( )". Siempre que el objeto "jugador" esté más lejos de 100 pasos del objeto "fantasma", el fantasma seguirá moviéndose en su dirección actual (horizontal en este ejemplo), rebotando al llegar a los bordes. Si la distancia entre los objetos se hace menor de 100 pasos, el bloque "repetir hasta que ( )" terminará, y el objeto "fantasma" comenzará a perseguir al objeto "jugador". El bloque "distancia a ( )" lo encontramos en la categoría "sensores".



EJERCICIO 37. PERSECUCIÓN FANTASMAL.

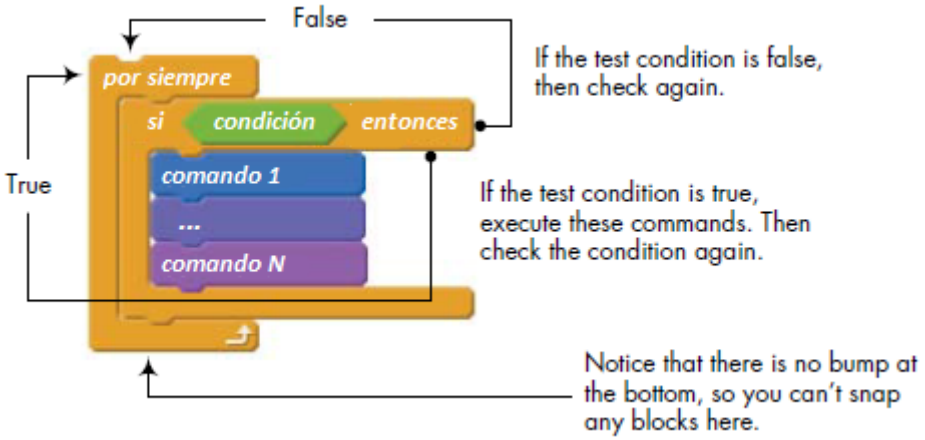
Crea una aplicación para completar el ejemplo previo. Primero, busca y agrega los dos objetos a tu proyecto, y fija su tamaño a un 60% de su tamaño original. El usuario simplemente controla al objeto "jugador" con las flechas del teclado. El objeto "fantasma" actúa como sigue: (1) Como hemos dicho en el ejemplo, siempre que el jugador esté más lejos de 100 pasos, el fantasma se moverá horizontalmente desde su posición inicial en $(x, y) = (-40, 80)$, rebotando al llegar a los bordes. (2) Cuando la distancia se haga más pequeña que 100 pasos, el fantasma siempre apuntará hacia el jugador y se moverá hacia él. Si el fantasma toca al jugador, el programa termina.

AMPLIACIÓN: Piensa cómo cambiarías la condición del bucle "repetir hasta que ( )" para hacer que el fantasma también persiga al jugador cuando la posición en y del jugador esté fuera de un cierto rango (por ejemplo, de -50 a 50).

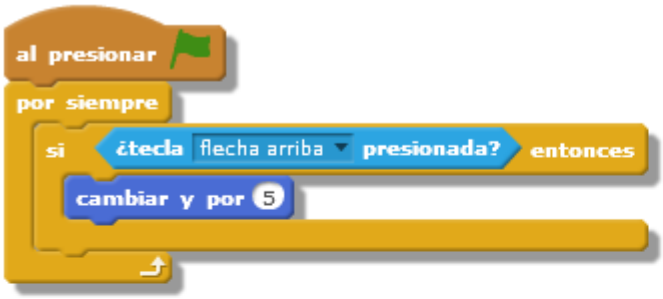
CONSTRUIR UN BLOQUE "SI" INFINITO.

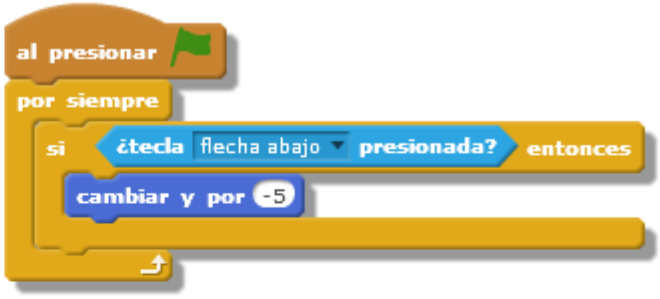
Los bucles infinitos son útiles en multitud de ocasiones: reproducir una música de fondo, animar objetos cambiando sus disfraces continuamente, etc. El bucle "por siempre" es un *bucle infinito incondicional* porque no tiene una condición que controle la ejecución de los comandos que contiene.

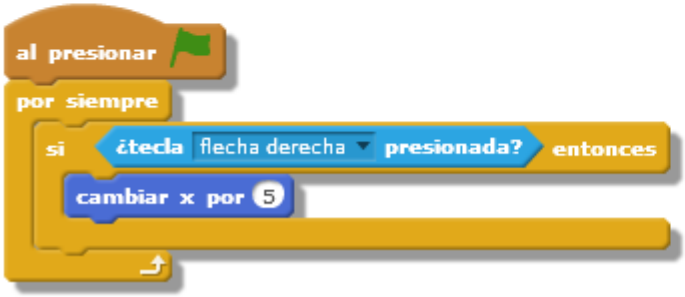
Eso podemos arreglarlo anidado un bloque "si" dentro de un bloque "por siempre" para crear un bucle infinito condicional, como el mostrado en la figura. La condición del bloque "si" se chequea al principio de cada iteración, y los comandos que contiene solo se ejecutan cuando la condición evalúa a verdadero. Notar que, como se supone que el bloque "por siempre" debe de ejecutarse indefinidamente, no podemos poner más bloques tras él.

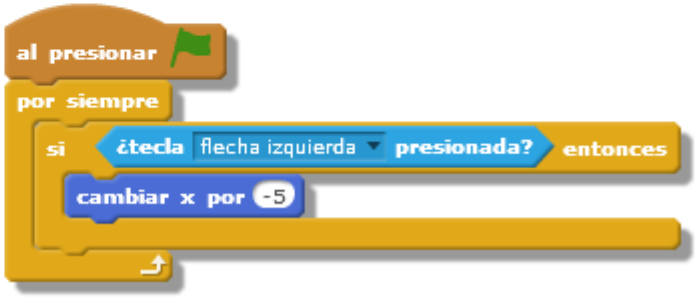


La combinación "por siempre"/"si" se usa de forma muy frecuente para controlar el movimiento de un objeto con las flechas del teclado:



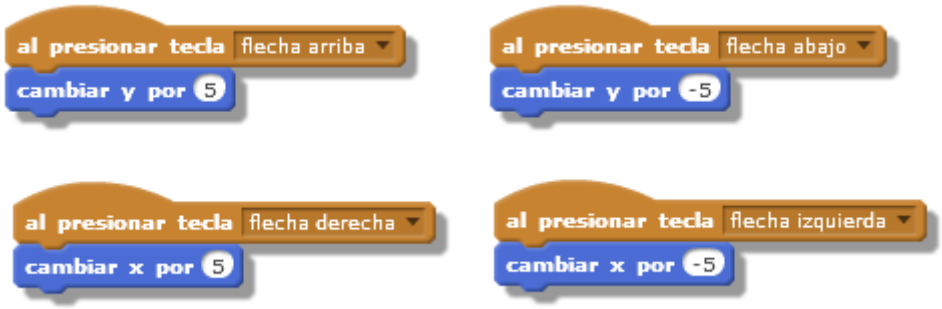






Crea este programa en Scratch y ejecútalo. Observa que si presionamos las teclas arriba y derecha simultáneamente, el objeto se moverá diagonalmente arriba y a la derecha. Prueba otras combinaciones de teclas para ver cómo responde esta aplicación.

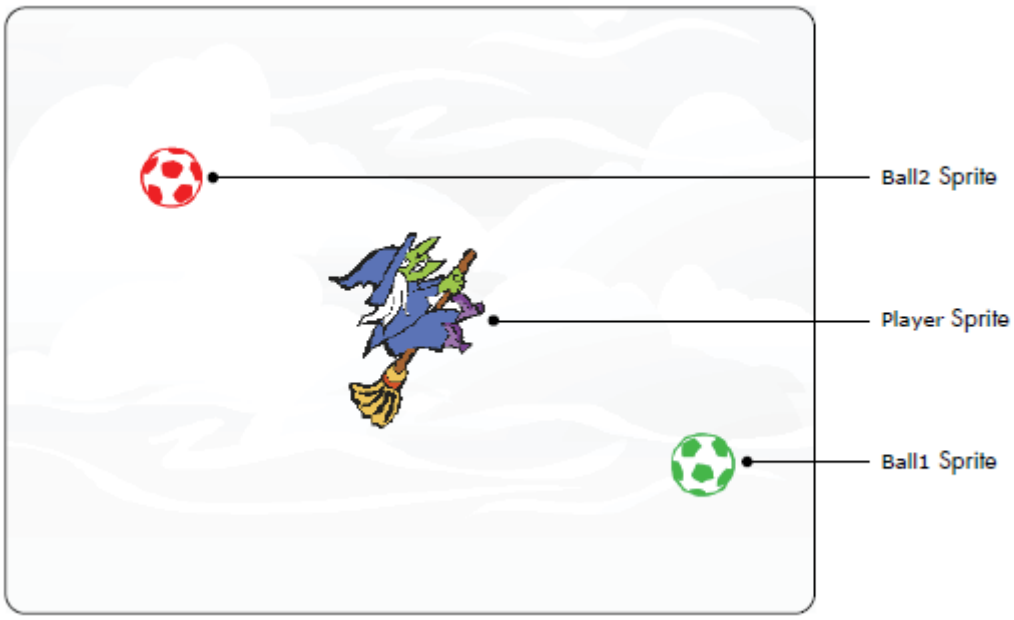
La siguiente figura muestra otra forma de controlar el movimiento de un objeto con las flechas del teclado. ¿Cuál de los dos métodos es más sensible al presionar las teclas? ¿Qué ocurre en este segundo método si presionamos dos teclas simultáneamente? Ahora, agrupa los cuatro bloques "si" del primer método dentro de un único bloque "por siempre". ¿Qué ocurre si presionas dos teclas al mismo tiempo? ¿Cómo cambia el comportamiento del objeto?



7.2. COMANDOS DE PARADA.

Digamos que estamos escribiendo un programa para hallar el primer entero menor que 1000 que es divisible entre 3, 5, y 7. Podemos escribir un código que compruebe los números 999, 998, 997, y así sucesivamente, uno tras otro, en un bucle. Queremos *parar* la búsqueda al encontrar el número que estamos buscando (945 en este caso). ¿Cómo le decimos a Scratch que termine el bucle y pare el programa? Para ello podemos usar el comando "detener ( )" de la categoría "control". La figura muestra las tres opciones que admite este bloque:

La primera opción termina de forma inmediata el programa que ejecuta el bloque. La segunda opción para *todos* los programas que estén en ejecución en nuestra aplicación; es equivalente al botón rojo ubicado sobre el escenario. (Notar que no podemos conectar más bloques bajo el bloque "detener" cuando usamos una de estas dos opciones; la base del bloque tiene forma lisa). La tercera opción le permite al objeto (o al escenario) terminar todos sus programas excepto en que ejecutó el bloque "detener". La base de este bloque presenta una protuberancia, lo que significa que podemos conectar más bloques bajo él para ejecutarlos después de que el bloque "detener" suspenda el resto de programas del objeto.



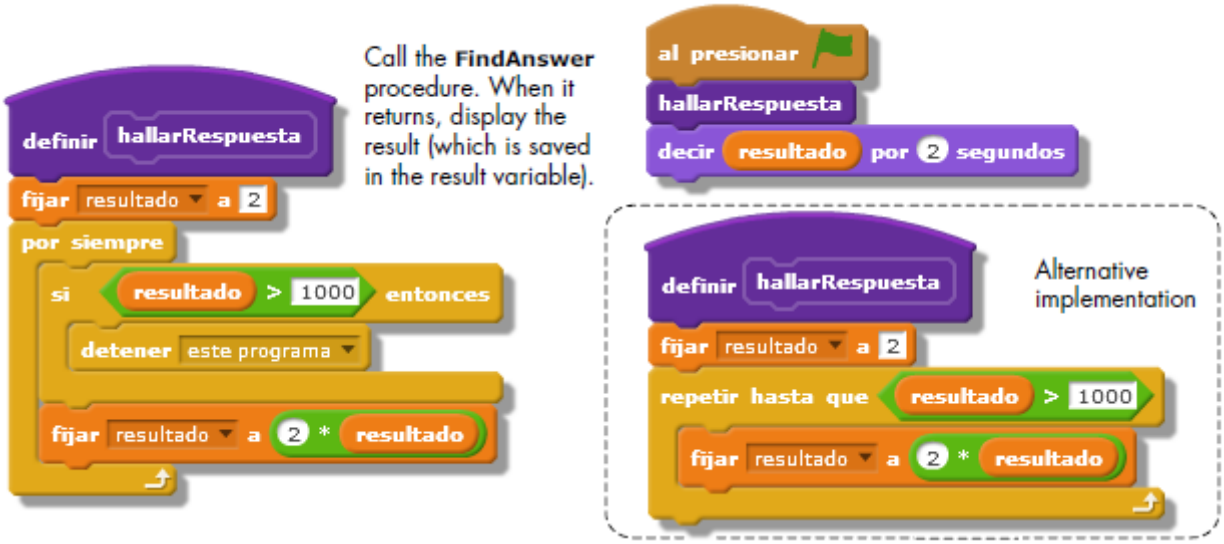
A modo de ejemplo, abre el archivo **detenerDemo.sb2** (ver figura). Los dos balones se mueven por el escenario y persiguen a la bruja. El jugador mueve al objeto bruja con el teclado, e intenta evitar que los balones la alcancen. Si el balón rojo toca a la bruja, la partida termina. Si el balón verde toca a la bruja deja de perseguirla, pero el balón rojo pasará a moverse un poco más rápido, lo que hará más difícil la partida. Los programas para controlar al balón verde (izquierda) y al balón rojo (derecha) se muestran en la figura:

Observa que cuando el balón verde toca al jugador, incrementa la variable "rapidez" (que especifica la rapidez con la que se mueve el balón rojo), e invoca al comando "detener (este programa)" para finalizar el programa. El resto de programas deberían continuar su ejecución normalmente. Ahora, si el balón rojo toca al jugador, su programa ejecuta el comando "detener (todos)", que hace que paren todos los programas de la aplicación.

También podemos usar el bloque "detener" para terminar un procedimiento en cualquier punto de su ejecución, y hacerlo volver al programa que lo ha llamado. Hablaremos sobre ello en la siguiente sección.

TERMINAR UN BUCLE COMPUTACIONAL.

Imagina que queremos hallar la primera potencia de 2 que es mayor que 1000. Para ello, escribimos un *procedimiento* que compruebe los números 2^1, 2^2, 2^3, 2^4, y así sucesivamente, mediante un bucle. Cuando encontremos el número que buscamos, queremos que el programa muestre la respuesta por pantalla y pare el procedimiento. La figura muestra dos formas de hacerlo:



El procedimiento de la izquierda inicializa la variable "resultado" a 2, que es la primera potencia de 2 que debemos comprobar ($2^1 = 2$), y después entra en un bucle infinito para buscar la respuesta. En cada iteración comprueba el valor de "resultado", y si es mayor que 1000, el procedimiento invoca el comando "detener (este programa)" para parar y volver al programa que lo llamó (también mostrado en la figura). Si "resultado" no es mayor que 1000, se ejecuta el comando tras el bloque "si", el cual se encarga de multiplicar "resultado" por 2 para calcular la siguiente potencia de 2. Si ejecutamos este programa, veremos que el bloque "si" encuentra que "resultado" es 2 en la primera iteración, 4 en la segunda, 8 en la tercera, etc. Este proceso continúa hasta que "resultado" termina haciéndose mayor que 1000, momento en el que el procedimiento para y vuelve al programa que lo llamó, donde simplemente se muestra por pantalla el valor de "resultado" mediante un bloque "decir".

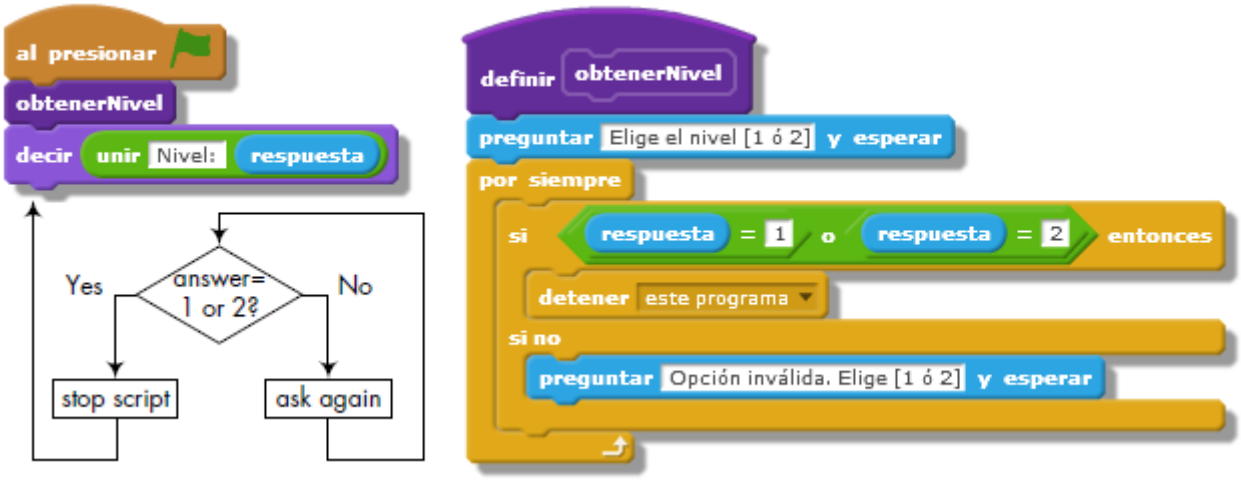
A la derecha, tenemos una implementación alternativa del mismo procedimiento. Aquí hemos usado un bloque "repetir hasta que ( )" que permanece en el bucle hasta que "resultado" se hace mayor que 1000. Como en la primera implementación, el bucle continúa multiplicando por 2 el valor de "resultado" hasta que se hace mayor que 1000. Cuando esto ocurre, el bucle termina de forma natural, y el procedimiento vuelve al programa que lo llamó. Notar que, en este caso, no hemos necesitado usar un bloque "detener".

El bloque "detener" también es útil cuando necesitamos validar los datos que introduce el usuario. Este es el objetivo de la siguiente sección.

VALIDAR DATOS INTRODUCIDOS POR EL USUARIO.

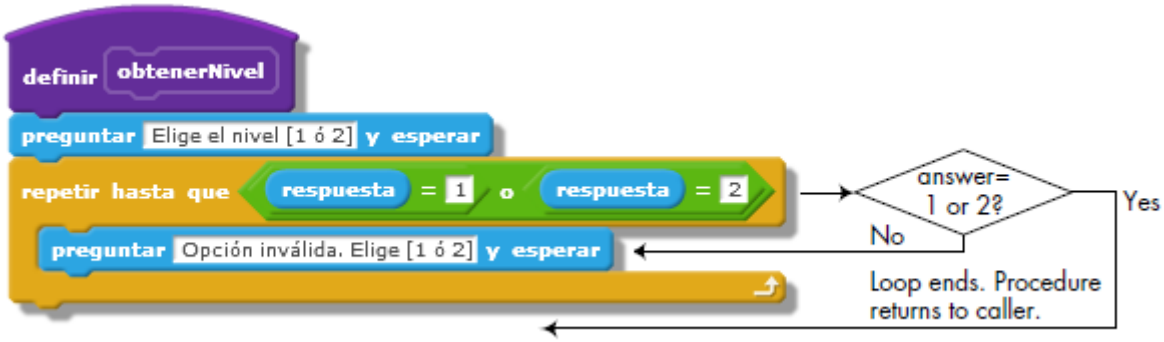
Cuando escribimos una aplicación que lea datos introducidos por el usuario, siempre debemos comprobar que los datos insertados son válidos antes de empezar a procesarlos. Las estructuras de repetición nos ayudan a realizar esta tarea. Si los datos proporcionados por el usuario no son válidos, podemos usar un bucle para mostrar un mensaje de error y pedirle al usuario que introduzca un valor correcto.

A modo de ejemplo, digamos que estamos creando un juego con dos niveles, y necesitamos que el usuario seleccione en qué nivel desea jugar. Los únicos datos de entrada válidos son 1 y 2. Si el usuario inserta un número distinto, hemos de ofrecerle la posibilidad de introducir un valor correcto. Una forma de hacerlo es ésta:



El procedimiento "obtenerNivel" le pide al usuario que elija un nivel (1 ó 2) y comprueba la respuesta dentro de un bucle "por siempre". Si la respuesta del usuario no es válida, el bucle le pide al usuario que vuelva a elegir el nivel. Si el usuario introduce un número válido, el procedimiento llama al bloque "detener (este programa)" para terminar el bucle y finalizar el procedimiento. Cuando esto ocurre, el programa principal (que ha estado esperando pacientemente a que el procedimiento "obtenerNivel" realice su tarea) continúa ejecutando en bloque "decir".

La siguiente figura muestra una forma alternativa de construir el procedimiento "obtenerNivel" mediante un bloque "repetir hasta que ( )":



Este procedimiento le pide al usuario que introduzca su elección, y espera una respuesta. Si el usuario introduce 1 ó 2, la condición del bloque "repetir hasta que ( )" evalúa a verdadero, lo que hace que el bucle termine de forma natural, y el procedimiento finaliza. Por otro lado, si el usuario introduce un valor distinto de 1 ó 2, la condición del bucle evalúa a falso, y se ejecuta el comando "preguntar" dentro del bucle. Este comando espera la respuesta del usuario, y el bloque "repetir hasta que ( )" continuará pidiéndole al usuario que haga una elección hasta que introduzca un dato válido. De nuevo, notar que esta implementación no requiere un bloque "detener".

7.3. CONTADORES.

En ocasiones necesitaremos llevar un control del número de iteraciones que realiza un bucle. Por ejemplo, si queremos que el usuario solo tenga tres oportunidades para introducir una contraseña correcta, debemos contar los intentos, y cerrar el programa tras el tercer intento fallido. Podemos manejar estas situaciones usando una variable (habitualmente llamada *contador de bucle*) que cuente el número de iteraciones por las que pasa el bucle. Veamos algunos ejemplos.

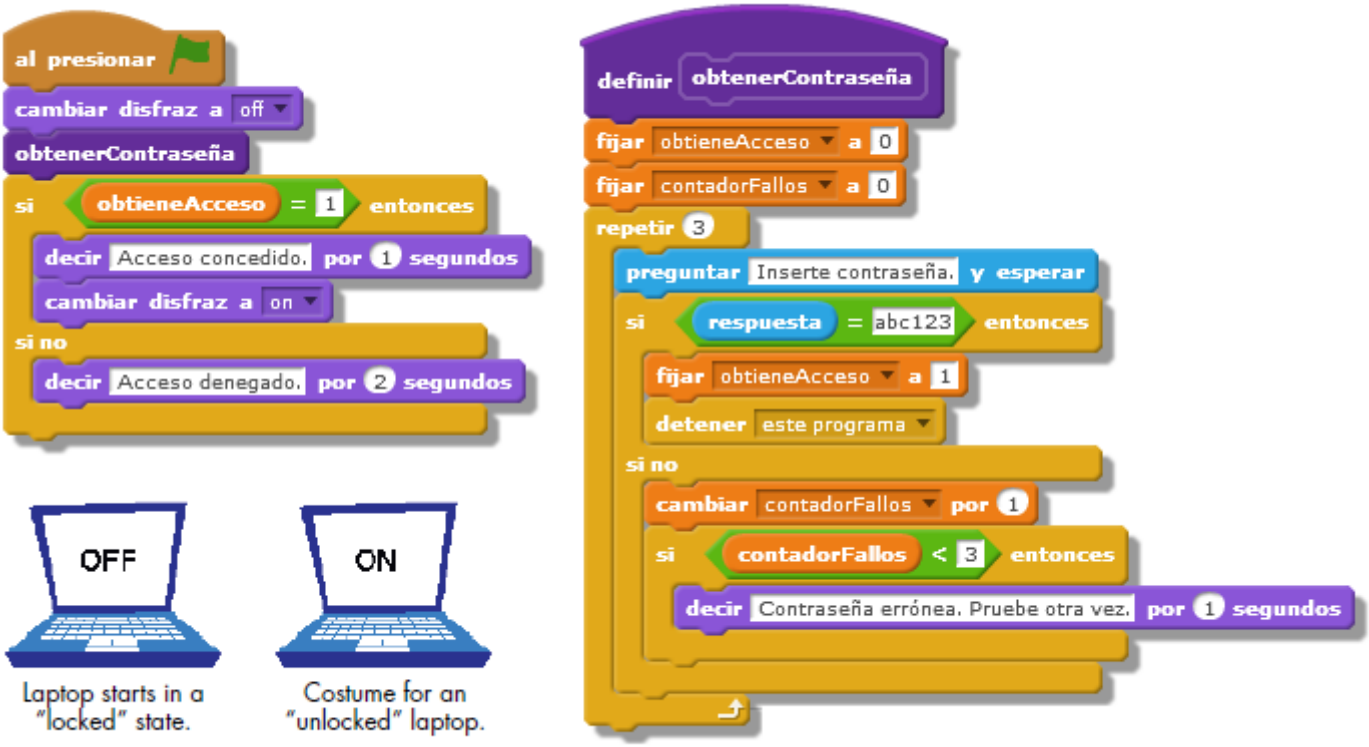
COMPROBAR UNA CONTRASEÑA.

El programa de la figura le pide al usuario que introduzca una contraseña para desbloquear un portátil. El objeto "portátil" tiene dos disfraces, ver figura: el disfraz con el texto OFF indica que el ordenador está bloqueado, y el disfraz con el texto ON indica que está desbloqueado. El usuario no obtendrá acceso al portátil si introduce tres veces una contraseña errónea.

Al pinchar en la bandera verde, el objeto "portátil" cambia al disfraz "off", y llama al procedimiento "obtenerContraseña" para autenticar al usuario. Este procedimiento le devuelve al programa principal una comprobación de la contraseña introducida, usando el indicador "obtieneAcceso". Cuando volvemos del procedimiento al programa principal, el bloque "si / si no" comprueba el indicador "obtieneAcceso" para decidir si el usuario debería o no acceder al sistema. Si "obtieneAcceso" vale 1, lo que indica que el usuario introdujo la contraseña correcta, el bloque "si" ejecuta un comando "decir" que muestra por pantalla el texto "Acceso concedido", y cambia el disfraz del objeto "portátil" a "on". En caso contrario, el programa muestra el texto "Acceso denegado" y el objeto portátil continúa con el disfraz "off".

El procedimiento "obtenerContraseña" comienza fijando el valor del indicador "obtieneAcceso" a 0, para registrar que aún no ha recibido una contraseña válida, e inicializa la variable "contadorFallos" (nuestro contador de bucle) a 0. A continuación, ejecuta un bucle repetir con un máximo de 3 iteraciones. A cada iteración, el procedimiento le solicita al usuario que introduzca una contraseña, y queda en espera. Si el usuario introduce la contraseña correcta ($abc123$ en este caso), el procedimiento fija el valor del indicador "obtieneAcceso" a 1, y se para a sí mismo ejecutando el comando "detener (este programa)" para retornar al programa que lo llamó. En caso contrario, si el usuario no ha agotado los tres intentos, se muestra un

mensaje de error, y el procedimiento le da al usuario otra oportunidad. Cuando el usuario falla tres veces consecutivas, el bucle "repetir (3)" termina automáticamente, y el procedimiento vuelve al programa principal con el valor del indicador "obtieneAcceso" todavía a 0.



Abre el archivo **compruebaContraseña.sb2** y ejecútalo. ¿Qué ocurre si insertas la contraseña $aBC123$ (en lugar de $abc123$)? ¿Qué te dice este hecho sobre la comparación de cadenas en Scratch?

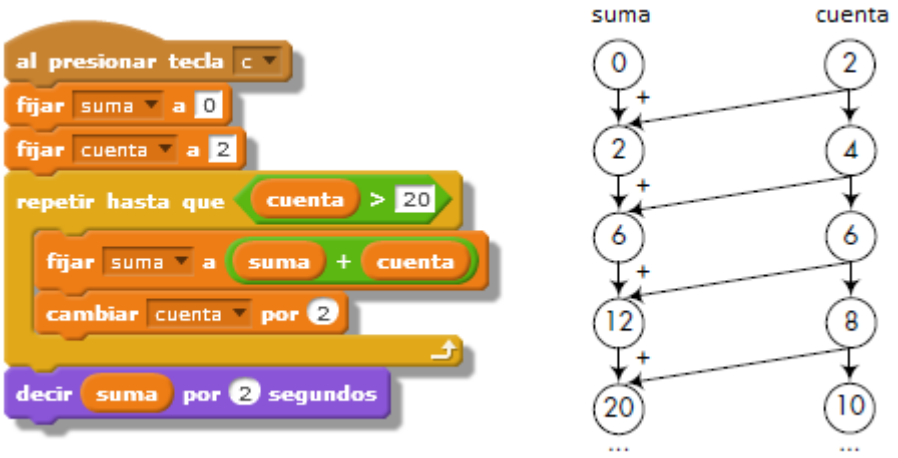
EJERCICIO 38. COMPRUEBA CONTRASEÑA.

Abre el archivo **compruebaContraseña.sb2** e intenta implementar el procedimiento "obtenerContraseña" usando un bucle "repetir hasta que ( )". Guarda el archivo resultante como **compruebaContraseña (2).sb2**.

CONTAR CON UN INCREMENTO CONSTANTE.

Por supuesto, no siempre tenemos que incrementar nuestros contadores en una unidad cada vez que el bucle realiza una nueva iteración. Por ejemplo, el programa de la izquierda hace que un objeto cuente desde 5 hasta 55 en incrementos de 5. El programa de la derecha hace que el objeto cuente hacia abajo desde 99 hasta 0 en decrementos de 11 (o en incrementos de -11), esto es: $99,\ 88,\ 77, \ldots,\ 11,\ 0$.

Para ver cómo usar esta técnica de conteo en una aplicación práctica, imagina que queremos hallar la suma de todos los enteros pares desde 2 hasta 20 (ambos incluidos), esto es: $2+4+6+8+\cdots+20$. El programa de la figura hace exactamente eso:



El programa arranca inicializando la variable "suma" a 0 y la variable "cuenta" a 2, y después entra en un bucle condicional que se repite hasta que "cuenta" se hace mayor que 20. A cada iteración del bucle, el valor de "suma" se actualiza al valor actual de "suma" más el valor de "cuenta", y además, el valor de la variable "cuenta" se incrementa en 2 para obtener el siguiente entero par de la secuencia (ver figura).

Por cierto, ¿qué crees que pasará si le pedimos a Scratch que repita un bucle 2,5 veces? Los tres ejemplos más abajo muestran cómo maneja Scratch este tipo de situaciones.

Por supuesto, no tiene sentido repetir 2,5 veces la ejecución de una serie de instrucciones, pero Scratch no produce error al introducir esos valores en un bucle "repetir ( )". En lugar de lanzar un mensaje de error, Scratch simplemente redondea el contador decimal de un bucle al entero más próximo.

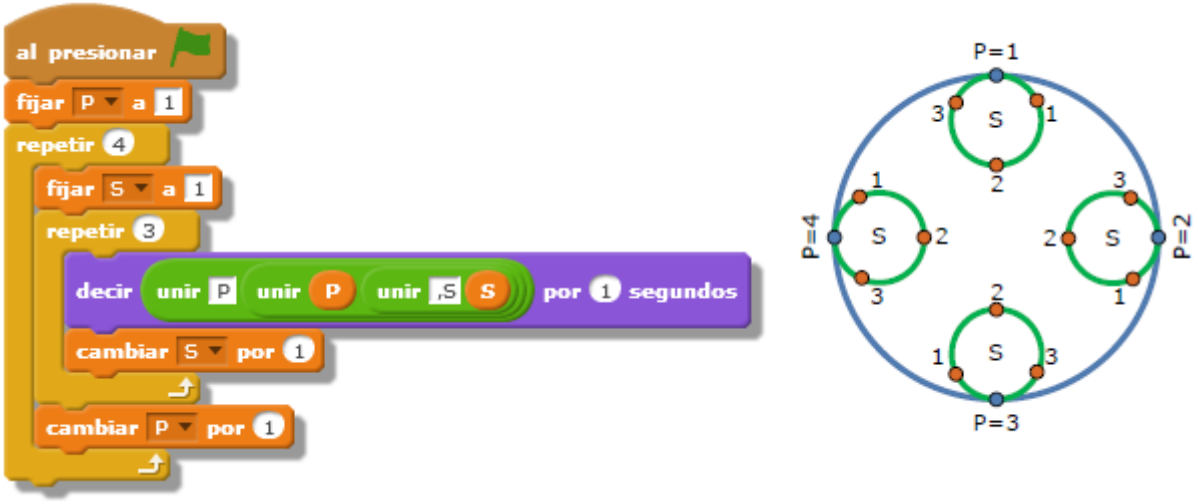
7.4. UNA REVISIÓN DE LOS BUCLES ANIDADOS.

¿Recuerdas la actividad guiada 5 sobre cuadrados rotados que hiciste en el capítulo 2 (sección 2.3)? Allí usamos dos bucles anidados para dibujar esos cuadrados. Un bucle (el *bucle interno*) era el responsable de dibujar el cuadrado, mientras que el otro bucle (el *bucle externo*) controlaba el número de rotaciones. En esta sección aprenderemos a usar el concepto de contadores de bucle en conjunción con los bucles anidados para crear iteraciones en dos (o más) dimensiones. Esta técnica es esencial en programación, y como veremos, puede usarse en multitud de aplicaciones.

A modo de ejemplo, digamos que un restaurante ofrece 4 tipos de pizza (P1, P2, P3, y P4) y tres tipos de ensaladas (S1, S2, y S3). Existen 12 combinaciones posibles para elegir. El dueño del restaurante quiere

imprimir un menú que liste todas las combinaciones de pizza y ensalada posibles, junto con sus precios combinados y sus contenidos calóricos. Veamos cómo usar bucles anidados para generar una lista de todas las combinaciones. (Dejaremos el cálculo de los precios y del contenido calórico como ejercicio).

Si lo piensas, verás que sólo necesitamos dos bucles: un bucle (el bucle externo) para pasar por todos los tipos de pizza, y otro bucle (el bucle interno) para pasar por todos los tipos de ensaladas. El bucle externo comienza con P1, mientras que el interno recorre S1, S2, y S3. A continuación, el bucle externo se mueve a P2, y el bucle interno vuelve a recorrer S1, S2, y S3. Este proceso continúa hasta que el bucle externo haya pasado por los cuatro tipos de pizza. La figura muestra una implementación de esta idea:



El programa usa dos bucles y dos contadores. El contador del bucle externo ("repetir (4)") se denomina "P", y el contador del bucle interno ("repetir (3)") se llama "S". En la primera iteración del bucle externo (para el que $P = 1$), el valor del contador "S" se fija a 1, y el bucle interno se repite tres veces. A cada repetición, este bucle ejecuta un comando "decir" para mostrar los valores actuales de "P" y "S", y a continuación incrementa "S" en 1 unidad. Por con siguiente, la primera iteración del bucle externo hace que el objeto diga "P1, S1", luego "P1, S2", y por último "P1, S3".

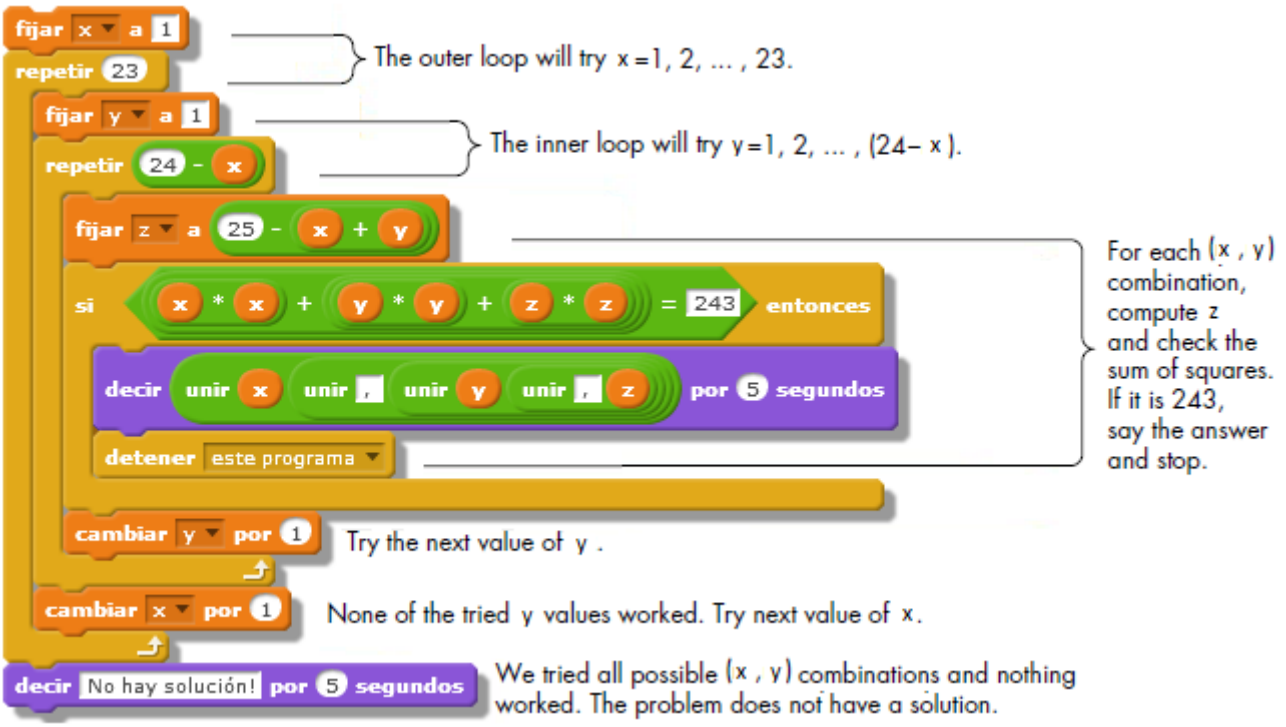
Cuando el bucle interno termina después de repetirse tres veces, "P" se incrementa en 1 unidad, y comienza la segunda iteración del bucle externo. El valor de "S" se reinicia a 1, y se ejecuta de nuevo el bucle interno. Esto hace que el objeto diga "P2, S1", "P2, S2", y "P2, S3". El proceso continúa de forma similar, haciendo que el objeto diga "P3, S1", "P3, S2", y "P3, S3", y finalmente "P4, S1", "P4, S2", y "P4, S3".

Ahora que ya sabemos para qué sirven los bucles anidados, vamos a aplicar esta técnica para resolver un interesante problema matemático. Queremos escribir un programa para hallar tres enteros positivos x, y, y z (distintos de cero) tales que $x + y + z = 25$ y además $x^2 + y^2 + z^2 = 243$. Como los ordenadores son muy eficientes realizando tareas repetitivas, nuestro plan será probar todas las combinaciones posibles de tres números (técnica llamada *búsqueda exhaustiva*) y dejar que sea el ordenador el que haga todo el trabajo.

En base a la primera ecuación, el primer número, x, puede tener un valor cualquiera entre 1 y 23, porque debemos sumarle dos números (distintos de cero) para obtener 25. (Puede que nos hayamos dado cuenta de que x no puede ser mayor que 15, porque $16^2 = 256$, resultado que es mayor que 243. Pero por el momento vamos a ignorar la segunda ecuación, y vamos a ajustar el límite superior del bucle a 23).

Ahora, el segundo número, y, puede tomar un valor cualquiera entre 1 y $24 - x$. Por ejemplo, si x es 10, el valor máximo posible de y es 14, porque z debe ser como mínimo 1. Análogamente, si conocemos x e y, podemos computar z como $25 - (x + y)$. A continuación, también debemos comprobar si la suma de los cuadrados de esos tres números es 243. Y si es así, hemos terminado.

La figura muestra el programa ya terminado:

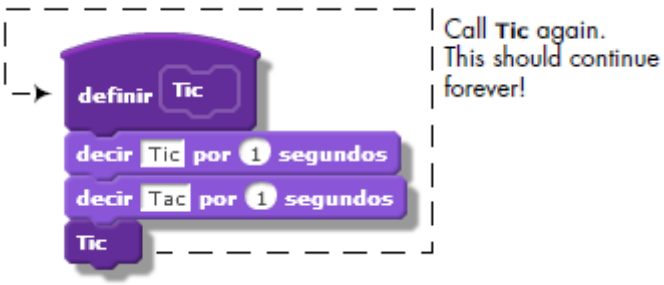


El bucle externo prueba todos los valores de x entre 1 y 23. Para cada valor de x, el bucle interno prueba todos los valores de y entre 1 y $(24 - x)$. Para cada combinación de x e y, el programa ajusta $z = 25 - (x + y)$, y después comprueba si la suma de los cuadrados de estos tres números es 243. Si ése es el caso, el programa muestra la respuesta por pantalla y termina. Si ninguna de las combinaciones se ajusta a los requerimientos, el programa indica por pantalla que el problema no tiene solución, y termina.

7.5. RECURSIVIDAD: PROCEDIMIENTOS QUE SE LLAMAN A SÍ MISMOS.

Las estructuras de repetición que hemos presentado hasta ahora nos permiten repetir una instrucción o un conjunto de instrucciones mediante iteración. Otra técnica que podemos usar para producir repetición es la **recursividad**.

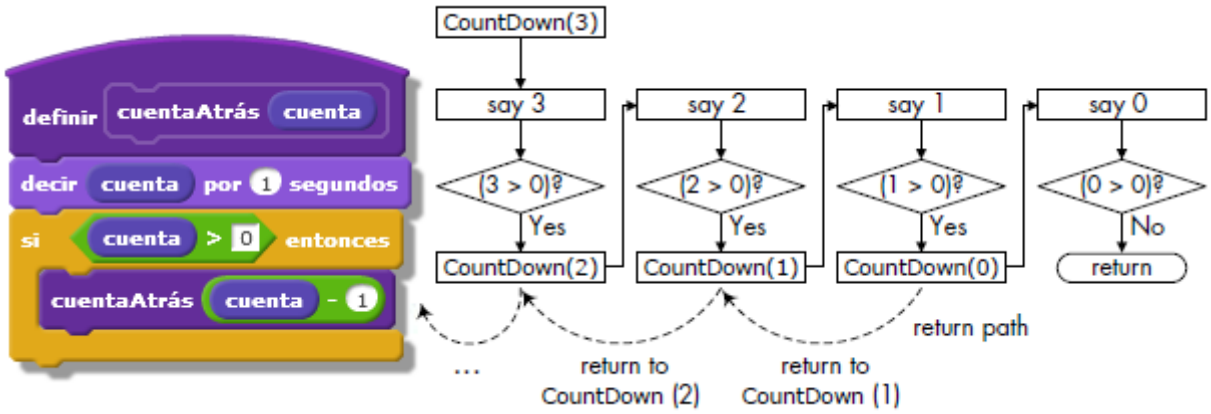
La recursividad consiste en que un procedimiento se llame a sí mismo, bien directamente, o bien indirectamente a través de otro procedimiento (por ejemplo, A llama a B, B llama a C, y C llama a A). Vamos a demostrar este concepto considerando el ejemplo de la figura:



El procedimiento "Tic" ejecuta dos comandos "decir" (el primero dice "Tic" y el segundo dice "Tac"), y después, se llama a sí mismo de nuevo. La segunda llamada hace la misma cosa, y de esta forma, el objeto estará diciendo "Tic Tac" para siempre, a no ser que una acción externa al procedimiento lo detenga. Notar que haciendo que un procedimiento se llame a sí mismo hemos conseguido repetir los dos comandos "decir"

de forma indefinida sin usar bloques de bucles. La forma de recursividad usada en este ejemplo se denomina *recursividad de cola*, porque la llamada recursiva se localiza justo al final del procedimiento. (Scratch también permite llamadas recursivas antes de la última línea, pero nosotros no estudiaremos ese tipo de recursividad).

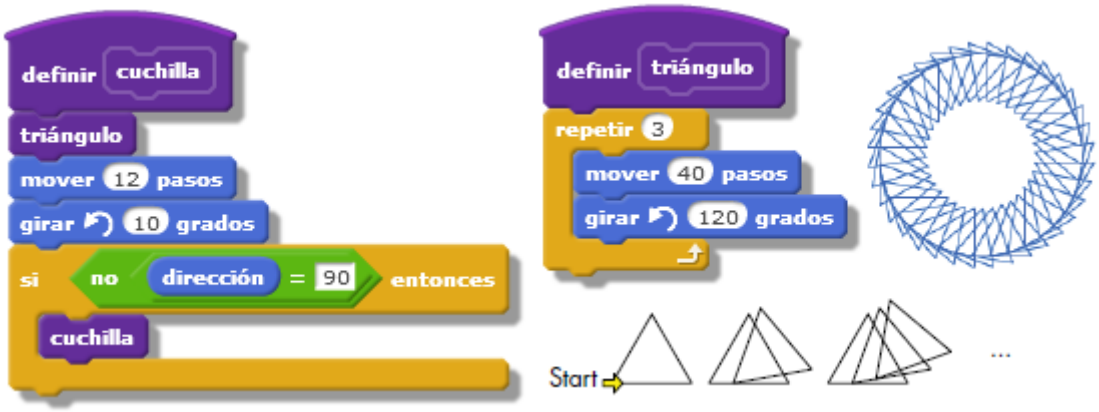
Como la recursividad infinita no suele ser una buena idea, debemos controlar el control de la ejecución de un procedimiento recursivo con condicionales. Por ejemplo, el procedimiento podría incluir un bloque "si" que determine si debería hacerse o no la llamada recursiva. La figura muestra el procedimiento recursivo "cuentaAtrás" (countDown), el cual realiza una cuenta atrás desde un cierto número inicial hasta cero:



Veamos cómo funciona este procedimiento cuando lo llamamos pasándole un argumento de 3. Cuando el procedimiento comienza, el valor del argumento "cuenta" está fijado a 3, y el comando "decir" muestra el número 3 por pantalla. A continuación, el procedimiento comprueba si el valor de "cuenta" es mayor que cero. Como 3 es mayor que 0, el procedimiento se llama a sí mismo pasándose como argumento la resta "cuenta"-1, cuyo valor actual será 2.

En la segunda llamada, el procedimiento comienza con un valor para "cuenta" de 2. El comando "decir" muestra el número 2 por pantalla, y se llama a sí mismo pasándose un argumento de $2-1=1$. Este proceso continúa hasta que el procedimiento se llama a sí mismo con un argumento de 0. Después de mostrar el número 0 por pantalla, el procedimiento comprueba si "cuenta" es mayor que cero. Como la expresión lógica dentro del bloque "si" evalúa a falso, no se hacen más llamadas recursivas, y el procedimiento retorna al programa principal.

Ahora que ya conocemos los principios de la recursividad de cola, vamos a aplicarla a programas más interesantes. A modo de ejemplo, consideremos el procedimiento "cuchilla" mostrado en la figura:



Supongamos que el objeto dibujante que ejecuta este procedimiento (la flecha amarilla en la figura) comienza en algún lugar del escenario apuntando en la dirección $90°$ (hacia la derecha). Después de dibujar un triángulo equilátero (procedimiento "triángulo"), el objeto se mueve 12 pasos hacia delante y gira $10°$ en

contra de las agujas del reloj. Entonces, el procedimiento comprueba la nueva dirección del objeto. Si el objeto *no* está apuntando en la dirección 90°, el procedimiento se llama a sí mismo otra vez para dibujar el siguiente triángulo de la secuencia. En caso contrario, no se realiza la llamada recursiva, y el procedimiento termina después de dibujar el disco de sierra mostrado en la figura.

Para ejemplos tan sencillos como el que acabamos de mostrar, seguramente sería más sencillo usar un bloque "repetir" para conseguir la repetición deseada. Pero como hemos mencionado antes, hay muchos problemas que son más fáciles de resolver usando recursividad en vez de iteración.

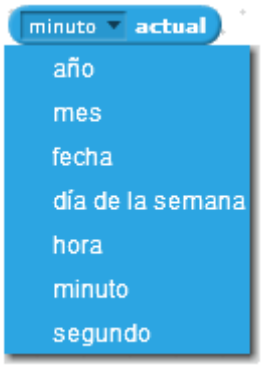
EJERCICIO 39. MI BLOQUE "DECIR ( ) ( ) VECES".

¿Qué hace el siguiente procedimiento? Impleméntalo y llámalo (desde un programa principal) con diferentes argumentos para comprobar tu respuesta.

7.6. PROYECTOS SCRATCH.

PROYECTO 13. RELOJ ANALÓGICO.

El bloque "( ) actual" de la categoría "sensores" puede reportar al año, mes, fecha, días de la semana, hora, minuto, o segundo actuales:

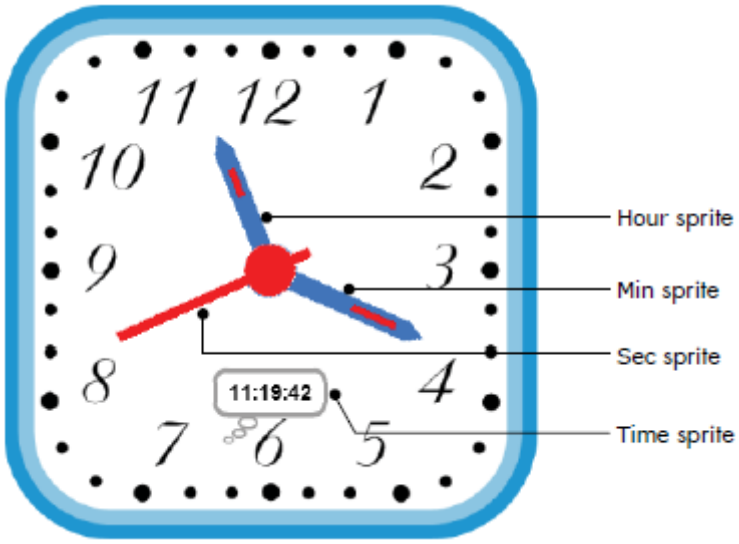


En este proyecto usaremos este bloque para construir el reloj analógico mostrado en la figura. Abre el archivo **Proyecto 13_sinCodigo.sb2**. La aplicación contiene 4 objetos: los objetos "seg", "min", y "hora", que representan las tres agujas del reloj, y el objeto "tiempo" (un pequeño punto blanco), que muestra por pantalla la hora en formato digital (mediante un bocadillo de pensamiento, ver figura).

El reloj arranca al pinchar en la bandera verde. En respuesta, los 4 objetos comienzan un bucle "por siempre" para actualizar sus respectivos estatus basándose en la hora actual del sistema (el ordenador).

1) Comencemos escribiendo el programa para el objeto "seg": El número de segundos reportados por el bloque "(segundos) actual" varía entre 0 y 59. Cuando el sistema reporta 0 segundos, el objeto "sec"

debería apuntar hacia arriba (dirección $0°$); a los 15 segundos, el objeto "sec" debería apuntar hacia la derecha (dirección $90°$), y así sucesivamente. De aquí se deduce que la aguja "sec" debería girar $6°$ (esto es, $360°$ dividido entre 60 segundos) por cada segundo transcurrido. Por lo tanto, la dirección en la que debe apuntar la aguja "sec" viene dada por la relación "(segundo) actual" $\times 6$. El programa para el objeto "sec" simplemente debe empezar al clicar en la bandera verde, y mediante un bucle infinito, hacer que la aguja siempre apunte en la dirección "(segundo) actual" $\times 6$ y espere 1 segundo antes de volver a cambiar de dirección.



2) De nuevo, el número de minutos reportados por el bloque "(minutos) actual" varía entre 0 y 59. Y de nuevo, cuando el sistema reporta 0 minutos, el objeto "min" debería apuntar hacia arriba (dirección $0°$); a los 15 minutos debería apuntar hacia la derecha (dirección $90°$), etc. Por consiguiente, el programa para la aguja "min" es totalmente análogo al de la aguja "seg", solo que ahora, la dirección en la que debe apuntar viene dada por "(minuto) actual" $\times 6$.

3) Vamos con la aguja horaria. El bloque "(hora) actual" reporta la hora del sistema como un número entre 0 y 23. Queremos que la aguja horaria apunte en la dirección $0°$ (arriba) a las 0 horas, que apunte en la dirección $30°$ a las 1 horas, en la dirección $60°$ a las 2 horas, en la dirección $90°$ (derecha) a las 3 horas, etc. Ahora bien, si la hora actual son, digamos, las 11:50, no queremos que la aguja horaria apunte exactamente a las 11, sino más cerca de las 12. Podemos hacer este ajuste tomando también en consideración los minutos actuales.

Como cada hora (o 60 minutos) se corresponde con un giro de $30°$ de la aguja *horaria* en la esfera del reloj, cada minuto equivale a un giro de $2°$. Por consiguiente, a cada minuto debemos ajustar el ángulo de la aguja *horaria* por el número de minutos actuales divididos por 2.

Por consiguiente, el programa del objeto "hora" comienza al pinchar en la bandera verde, y a continuación, entra en un bucle infinito. A cada iteración, el programa fija la variable local "ángulo" a la dirección en la que debería apuntar la aguja cada hora en punto, a saber, "(hora) actual" $\times 30$. A continuación, y conforme transcurre la hora actual, la aguja horaria debería desplazarse lentamente hacia la dirección de la siguiente hora. Para ello, cambiamos el valor de la variable "ángulo" por un valor igual a "(minuto) actual"$/2$. A continuación, el bucle hace que la aguja apunte en la dirección de "ángulo", y el bucle termina realizando una espera de 60 segundos.

4) El programa para el objeto "tiempo" es trivial. Es simplemente una serie de bloques "unir" anidados para construir una cadena de la forma $hora{:}\,minuto{:}\,segundo$ y mostrarla por pantalla mediante un bocadillo de pensamiento.

El proyecto ya está terminado. Guárdalo con el nombre **Proyecto 13.sb2**.

AMPLIACIÓN: Al igual que para la aguja horaria, modifica el programa para hacer que la aguja minutera también se mueva de forma progresiva, en vez de saltar cada minuto como lo hace ahora. Además, cambia el programa del objeto "hora" para que muestre una cadena de la forma "3:25:00 PM" (formato 12 horas) en vez de hacerlo de la forma "15:25:00" (formato 24 horas).

PROYECTO 14. CAZANDO PÁJAROS.

Vamos a programar un juego sencillo en el que el jugador dispara contra unos pájaros. La interfaz de usuario la encontraremos en el archivo **Proyecto 14_sinCodigo.sb2** (ver figura).

Como vemos, el juego contiene 5 objetos: "pájaro1", un clon de "pájaro1", "pájaro2", "tirador" (el coche), y "bala". El jugador puede mover horizontalmente el objeto "tirador" usando las flechas del teclado. Al presionar la tecla espaciadora, el tirador dispara una bala verticalmente hacia el cielo. Si el jugador alcanza al pájaro1 o a su clon, gana un punto. El pájaro 2 es una especie protegida, y el jugador no debe disparar contra él; si la bala alcanza a pájaro2, la partida finaliza. El jugador dispone de un minuto para derribar tantos pájaros como pueda.

Cada pájaro usa dos disfraces. Al cambiar entre los dos disfraces, animaremos los pájaros para que parezca que están batiendo sus alas. El escenario tiene dos fondos, llamados "inicio" y "final". El fondo "final" es idéntico al fondo "inicio", pero con el mensaje "Game Over" sobreimpreso sobre la imagen.

1) Comencemos con el código para el escenario, que consta de dos programas.

Programa 1: Al clicar en la bandera, el escenario se pone el disfraz "inicio", reinicia un cronómetro (bloque "reiniciar cronómetro" de la categoría "sensores"), y entra en un bucle infinito. A cada iteración actualiza el valor de la variable global "tiempoRestante", redondeando la resta del tiempo transcurrido en el cronómetro de Scratch al valor inicial de 60 segundos. Para ello, usamos el bloque:

Todavía dentro del bucle, el escenario comprueba si "tiempoRestante" llega a cero, y en su caso, envía el mensaje "GameOver".

Programa 2: Cuando el escenario recibe el mensaje "GameOver", espera $0{,}1$ segundos, cambia al disfraz "final", y finaliza la partida.

2) Continuamos con el código del objeto "tirador". Al pinchar en la bandera, el programa comienza ubicando al objeto en el centro de la parte inferior del escenario (ver figura). Mediante un bucle infinito, el programa detecta si el usuario presiona las teclas ← o → para mover al tirador en la dirección correspondiente.

3) Vamos con el código del objeto "pájaro1", que consta de varios programas:

Programa 1: Al pinchar en la bandera verde, el objeto se muestra, crea un clon de sí mismo (para crear el segundo pájaro1), se ubica arriba y a la izquierda del escenario (coordenadas $(x, y) = (-240, 120)$), y llama al procedimiento "comenzar", que sirve para mover al objeto horizontalmente de izquierda a derecha a lo largo del escenario, mientras lo anima para que mueva las alas.

Programa 2: el clon también empieza en la parte izquierda del escenario (pero a menor altura, coordenadas $(x, y) = (-240, -25)$), y también llama al procedimiento "comenzar".

Procedimiento "comenzar": este procedimiento usa un bucle infinito para mover al pájaro1 y a su clon horizontalmente a lo largo del escenario, de izquierda a derecha a pasos aleatorios de entre 5 y 20, mientras cambia constantemente de disfraz para animarlo. Si el bucle detecta que el pájaro está en extremo derecho del escenario ("posición en x" > 239), lo lleva de vuelta al extremo izquierdo ($x = -240$).

Programa 3: El pájaro1 y su clon también son destinatarios del mensaje "GameOver". Al recibirlo, simplemente se ocultan.

4) El código para pájaro2 es muy similar al del pájaro1. Al pinchar en la bandera verde, pájaro2 se mueve de izquierda a derecha, comenzando desde una altura de 40. A continuación se muestra y ejecuta un bucle infinito similar al del procedimiento "comenzar" del pájaro1. El pájaro simplemente se mueve hacia la derecha batiendo las alas y volviendo al extremo izquierdo cuando llega al extremo derecho. Por supuesto, pájaro2 también responde al mensaje "GameOver" ocultándose al recibirlo.

5) Hasta ahora, no hemos hecho nada para hacer que el tirador dispare. Aquí es donde entra en juego el objeto "bala", cuyo código consta de dos programas.

Programa 1: Al pinchar en la bandera verde, el programa inicializa a cero las variables globales "disparos" (número de balas disparadas) e "impactos" (número de pájaros alcanzados). A continuación, hace que el objeto bala apunte hacia arriba y lo oculta. Luego entra en un bucle infinito para comprobar constantemente si el jugador ha presionado la tecla espaciadora, bloque "tecla ( ) presionada" de la categoría "sensores". En ese caso, el programa incrementa en 1 la variable "disparos", crea un clon de la bala (que el programa2 moverá hacia el cielo), y espera $0{,}5$ segundos (para evitar que el jugador dispare otra bala demasiado pronto).

Programa 2: el programa del clon de la bala comienza llevándola a la posición del centro del objeto "tirador", y la muestra. A continuación, y mediante un bucle "repetir hasta que ( )", mueve a la bala hacia arriba en incrementos de 10 pasos. Cuando la coordenada y de la bala excede el valor 160, es porque ha llegado al extremo superior del escenario sin alcanzar a ningún pájaro. En ese caso, el programa sale del bucle "repetir hasta que ( )" y el programa borra el clon.

Dentro del bucle "repetir hasta que ( )", y conforme la bala se mueve hacia arriba, debemos comprobar si la bala alcanza a algún pájaro. Si la bala toca al pájaro1 (o a su clon), el programa incrementa en 1 la variable "impactos", y reproduce un sonido (el que te parezca más adecuado). Si la bala toca al pájaro2, el programa envía el mensaje "GameOver" para indicar el final de la partida. En ambos casos, el programa borra el clon de la bala porque ha terminado su recorrido.

El proyecto ya está terminado. Guárdalo con el nombre **Proyecto 14.sb2**.

AMPLIACIÓN: Tal y como está, el juego es plenamente funcional, pero puedes añadir muchas mejoras. Algunas sugerencias:

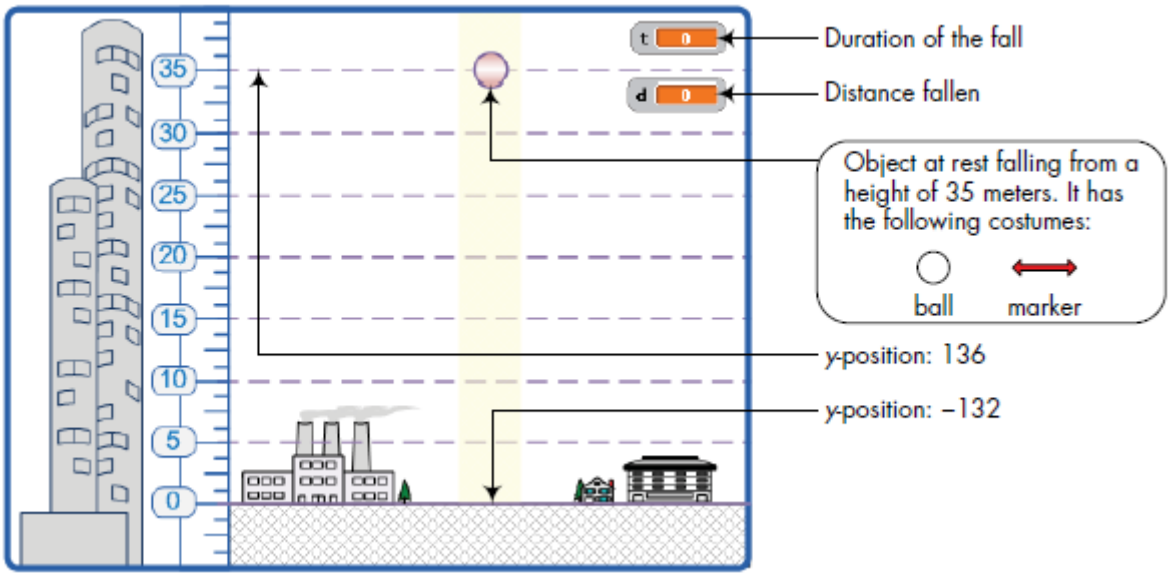
- Dale al jugador un número de balas limitado, e incorpora un marcador que cuente el número de disparos fallados.
- Añade más pájaros que se muevan con distintas velocidades (y si quieres, que tengan tamaños más reducidos). Recompensa al jugador con más puntos si alcanza a los pájaros más veloces y pequeños.
- Modifica el juego para que puedan participar dos jugadores.

PROYECTO 15. SIMULACIÓN DE UN MOVIMIENTO DE CAÍDA LIBRE.

En este proyecto desarrollaremos una aplicación que simule el movimiento de un objeto en caída libre. Ignorando los efectos de la flotabilidad y de la resistencia del aire, cuando un objeto es liberado desde una cierta altura partiendo del reposo (velocidad inicial cero), la distancia d que cae el objeto (en metros) durante un intervalo de tiempo t (en segundos) viene dada por la fórmula:

$$d = \frac{gt^2}{2}$$

, donde $g = 9{,}8\ m/s^2$ es la aceleración de la gravedad. El objetivo de esta simulación es mostrar la posición del objeto en caída en los instantes $0{,}5\ s,\ 1{,}0\ s,\ 1{,}5\ s,\ 2{,}0\ s$, y así sucesivamente, hasta que el objeto llegue al suelo. La interfaz de usuario está disponible en el archivo **Proyecto 15_sinCodigo.sb2** (ver figura).



En nuestra aplicación dejaremos caer una pelota en reposo desde una altura de 35 metros. Despejando este dato en la fórmula, vemos que el objeto llegará al suelo tras un tiempo de $t = \sqrt{(2 \times 35)/9{,}8} = 2{,}67\ s$. La aplicación incluye un objeto "pelota" que tiene dos disfraces: "pelota" y "marca". Cuando llega el instante de mostrar la posición de la pelota en caída, el objeto cambia momentáneamente al disfraz "marca", estampa una copia del disfraz en la posición actual, y vuelve al disfraz "pelota".

Vamos a empezar con el código para la pelota:

1) Comenzamos construyendo el procedimiento "inicializar", que lleva al objeto "pelota" a su posición inicial, $(x, y) = (70, 136)$, le pone el disfraz "pelota", y quita el bocadillo de diálogo y limpia del escenario las marcas de la simulación previa. A continuación, inicializa las variables "t" y "contador" a cero. (La variable "t" representa la duración de la caída, y "contador" servirá para llevar un registro del número de repeticiones de un bucle).

2) Construimos el procedimiento "sellarMarca", al que llamaremos para sellar una marca en algunas localizaciones específicas en ciertos instantes de tiempo durante la caída de la pelota. Este procedimiento simplemente cambia al disfraz "marca", sella el disfraz en pantalla, y vuelve al disfraz "pelota".

3) El programa principal lanza la simulación, que comienza al pinchar en la bandera verde. Este programa empieza llamando al procedimiento "inicializar".

Después, entra en un bucle infinito para calcular algunos parámetros de la simulación en diferentes instantes de tiempo. El bucle realiza esos cálculos para actualizar la posición de la pelota cada $0{,}05\ s$ y asegurar el movimiento continuo de la pelota en pantalla. Cada $0{,}05\ s$, actualizamos el valor de la variable "t", calculamos la distancia d que ha caído la pelota (usando la fórmula), e incrementamos en 1 el valor de la variable "contador".

Todavía dentro del bucle, el programa comprueba con un bloque "si / si no" si la pelota ha llegado al suelo (lo que ocurre cuando la pelota ha caído 35 metros o más ($d \geq 35$), ver sección 6.3 para recordar cómo evaluar esta condición lógica):

- En caso afirmativo (grupo "si"), el programa fija la coordenada y de la pelota a la coordenada y en la que se localiza el suelo ($y = -132$), muestra por pantalla la duración de la caída ($2{,}67\ s$), y termina la simulación.

- En caso contrario (grupo "si no"), el programa ajusta la posición vertical de la pelota de acuerdo con la distancia caída. Como una altura de 35 metros se corresponde con 268 píxeles en el escenario (ver figura), una regla de tres nos dice que una caída de d metros se corresponde con $268 \times (d/35)$. Por consiguiente, la posición vertical y de la pelota se obtiene restando esta cantidad a la posición y inicial, que es 136.

  Además, y como la duración de cada iteración es de $0{,}05\ s$, el bucle necesita 10 iteraciones para completar $0{,}5\ s$, que es el intervalo tras el cual sellamos una nueva marca. Por consiguiente, y todavía dentro del grupo "si no", el programa comprueba si el contador toma los valores 10, 20, 30, etc. (esto es, si "contador" es divisible entre 10). En ese caso, el objeto "bola" llama al procedimiento "sellarMarca" para estampar una marca en esa localización.

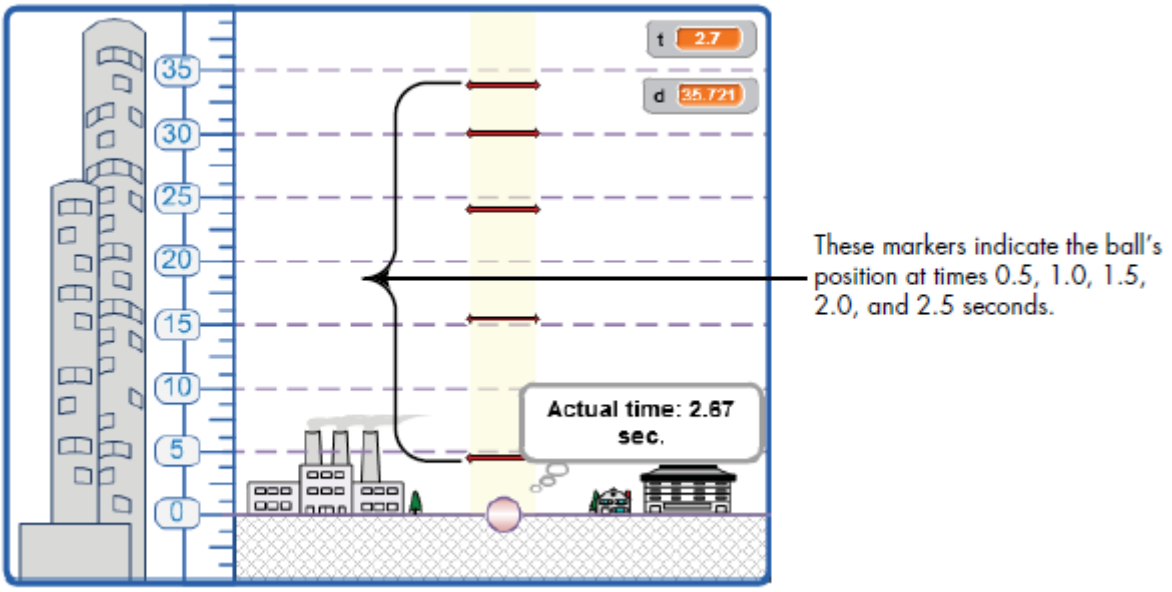
La siguiente figura muestra el resultado de la ejecución de nuestra simulación. Observa que la distancia que la pelota cae tras cada nuevo intervalo de $0{,}5\ s$ es cada vez mayor. Esto es así porque, debido a la gravedad, el objeto se acelera (esto es, su velocidad aumenta con el tiempo) a razón de $9{,}8\ m/s^2$.

AMPLIACIÓN: Intenta convertir esta aplicación en un juego en el que el jugador deja hacer la pelota para alcanzar a un objeto que se mueve horizontalmente en el suelo. Puedes añadir un marcador, cambiar la rapidez del objetivo, o incluso dar la opción de jugar la partida en otro planeta, en la Luna, etc. (Para hacer esto último, deberás cambiar la aceleración gravitacional).

Algunos datos:

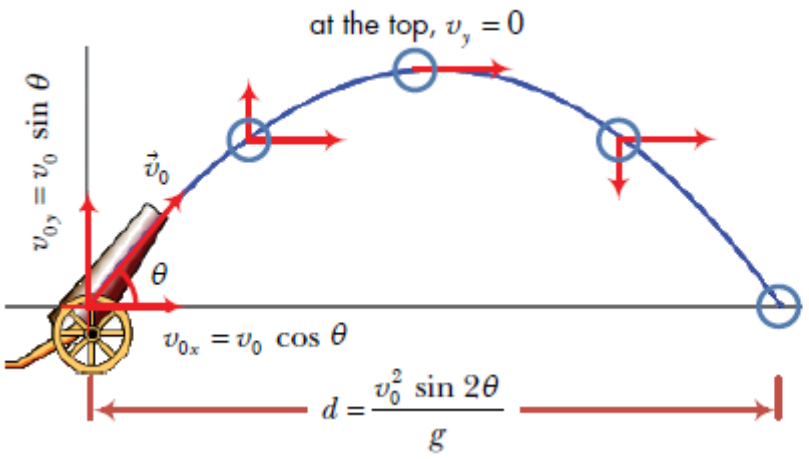
Planeta	Aceleración gravitacional (m/s^2)
Mercurio	2,8
Venus	8,9
Tierra	**9,81**
Marte	3,71
Júpiter	22,9
Saturno	9,1
Urano	7,8
Neptuno	11,00

Fuente: https://es.wikipedia.org/wiki/Anexo:Datos_de_los_planetas_del_sistema_solar (dato "gravedad ecuatorial").



PROYECTO 16. SIMULACIÓN DE MOVIMIENTO DE PROYECTIL.

Imagina que un cañón dispara un obús con una cierta velocidad inicial $\vec{v}_0$ y con un cierto ángulo θ desde la horizontal (ver figura). Podemos analizar la trayectoria del proyectil resolviendo su vector velocidad inicial $\vec{v}_0$ en sus componentes horizontal y vertical, v_{0x} y v_{0y}, en diferentes instantes de tiempo. Ignorando la resistencia del aire, la componente horizontal permanece constate, pero la componente vertical se ve afectada por la gravedad. Cuando se combinan los movimientos correspondientes a estas dos componentes, la trayectoria resultante es una parábola. Vamos a examinar las ecuaciones que gobiernan el movimiento de proyectil (despreciando la resistencia del aire).



maximum height

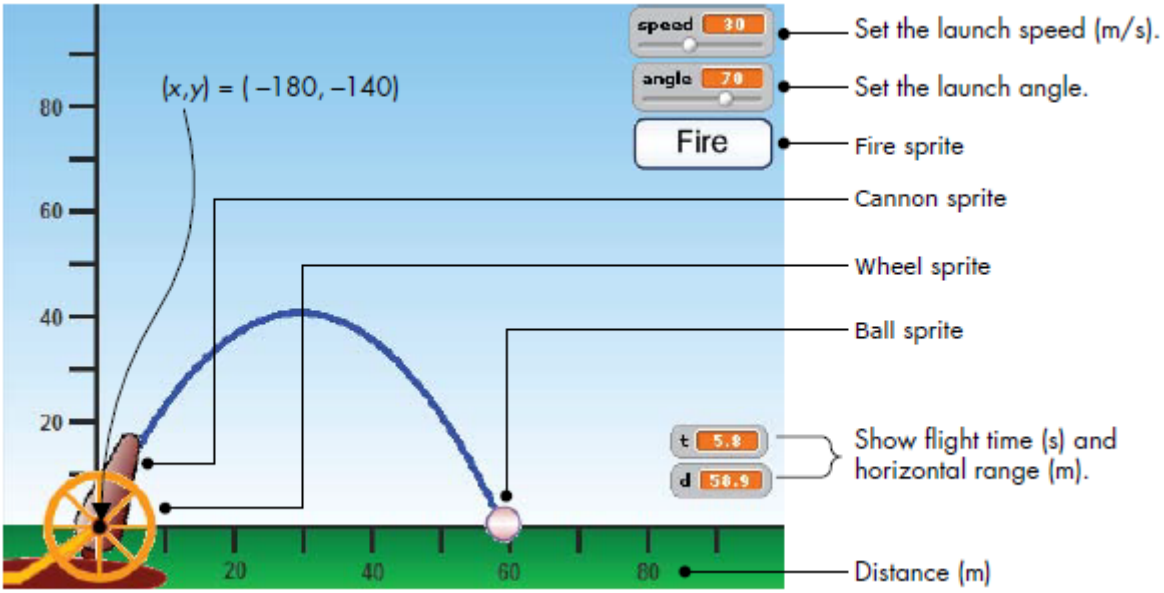
$$h = \frac{(v_0 \sin\theta)^2}{2g}$$

travel time

$$t = \frac{2v_0 \sin\theta}{g}$$

El origen de nuestro sistema de coordenadas es el punto en el que el proyectil comienza su vuelo, por lo que el valor de la coordenada x del proyectil en cualquier instante de tiempo es $x(t) = v_{0x}t$, y el de la coordenada y es $y(t) = v_{0y}t - \frac{1}{2}gt^2$, donde $v_{0x} = v_0 \cos\theta$ es la componente x de $\vec{v}_0$, $v_{0y} = v_0 \sin\theta$ es la componente y de $\vec{v}_0$, y $g = 9{,}8\ m/s^2$ es la aceleración gravitacional. Usando estas ecuaciones, podemos calcular el tiempo total de vuelo t, la altura máxima h alcanzada, y el alcance horizontal d del proyectil. Estas ecuaciones están escritas en la figura.

Esta información es todo lo que necesitamos para simular el movimiento de un proyectil. La interfaz de usuario está disponible en el archivo **Proyecto 16_sinCodigo.sb2**.



La aplicación contiene 4 objetos: El objeto "rueda" proporciona el eje de rotación del cañón, mientras que el objeto "cañón", que rota de acuerdo con el deslizador "ángulo", proporciona una indicación visual del ángulo de disparo. El objeto "disparo" (fire) es un botón que presiona el usuario para disparar el proyectil, y el objeto "bola" contiene el programa principal para calcular las coordenadas del proyectil y dibujar su trayectoria. El usuario proporciona el ángulo de disparo θ y la rapidez inicial v_0 usando dos controles con deslizador. El objeto "bola" comienza en la posición $(x, y) = (-180, -140)$ y dibuja la trayectoria parabólica para los parámetros especificados. Los monitores "t" y "d" muestran el tiempo de vuelo y el alcance horizontal durante el vuelo del proyectil.

1) La simulación comienza al clicar en la bandera verde. El código del objeto cañón es complicado, y el archivo lo proporciona ya construido. No vale la pena explicar los detalles, y nos basta con saber que los programas hacen apuntar el cañón en la dirección especificada por el usuario mediante el control con deslizador llamado "ángulo". La dirección de lanzamiento especificada queda almacenada en la variable "ángulo". (El usuario también puede especificar el ángulo de disparo clicando y arrastrando el propio cañón).

2) Cuando el usuario pincha en el botón "disparo", este objeto envía el mensaje "Fuego", que será recibido y procesado por el objeto "Bola". El programa del objeto botón también está construido en el archivo.

3) Vamos con el código para el objeto "Bola", que consta de dos programas:

Procedimiento "inicializar": Este procedimiento se encarga de mover al objeto "Bola" a la capa delantera, y la ubica en el punto de lanzamiento, posición $(x, y) = (-180, -140)$. Después, baja el lápiz, fija el tamaño del lápiz (4), y limpia de pantalla todos los trazos previos. A continuación, el procedimiento calcula los valores de las componentes horizontal (componente x) y vertical (componente y) de la velocidad inicial $\vec{v}_0$, que almacena en las variables "vx" y "vy", respectivamente. Recordemos que las fórmulas para las componentes

horizontal y vertical son $v_{0x} = v_0 \cos\theta$ y $v_{0y} = v_0 \sin\theta$. (El valor de la rapidez inicial v_0 lo especifica el usuario mediante el mando con deslizados llamado "rapidez", y queda almacenado en la variable con el mismo nombre. Por su parte, θ es el ángulo de lanzamiento (almacenado en la variable "ángulo"), y $\cos$ y $\sin$ son las funciones trigonométricas coseno y seno, respectivamente). Finalmente el procedimiento también inicializa el valor de la variable "t" a cero ($t = 0$).

Programa principal: El programa principal de este objeto comienza al recibir el mensaje "Fuego", y lo primero que hace es llamar al procedimiento "inicializar". A continuación, el programa entra en un bucle infinito, que calcula y actualiza la posición de la bola de cañón cada $0{,}02\ s$. Para ello, en primer lugar calcula la distancia vertical dy recorrida por la bola en cada intervalo:

$$dy = v_{0y}t - \frac{gt^2}{2}$$

Ahora, si dy es negativo, es porque la bola ha llegado al suelo, y en ese caso, el programa detiene la simulación. Si dy no es negativo, se calcula la distancia horizontal "d":

$$d = v_{0x}t$$

Una vez calculadas dy y d, el programa las escala en consonancia con el tamaño del escenario en el que mueve la bola. En la dirección vertical, disponemos de 320 pasos (desde -140 hasta 180), lo que se corresponde con $100\ m$, y en la dirección horizontal tenemos 420 pasos (desde -180 hasta 240), lo que también se corresponde con $100\ m$. Esto significa que una distancia vertical de dy metros es equivalente a $320 \times dy/100$ pasos, y que distancia horizontal de d metros es equivalente a $420 \times d/100$ pasos. Entonces, las coordenadas x e y actuales de la bola desde su posición inicial en $(x, y) = (-180, -140)$, variables "xPos" e "yPos" respectivamente, serán:

$$xPos = -180 + 420 \times d/100$$

$$yPos = -140 + 320 \times dy/100$$

De esta forma, el programa calcula los nuevos valores de "xPos" e "yPos" para este nuevo intervalo, y envía a la bola a esas coordenadas. Finalmente, el programa incrementa la variable "t" en una pequeña cantidad de $0{,}02\ s$, y el bucle se repite con este nuevo valor de "t" para calcular la siguiente posición de la bola.

El proyecto ya está terminado. Guarda el archivo como **Proyecto 16.sb2**. A modo de ejemplo, si la bola se dispara con un ángulo de $70°$ con una rapidez inicial de $30\ m/s$, el tiempo de vuelo total es de $5{,}75\ s$, y el alcance horizontal es de $59\ m$.

AMPLIACIÓN: Intenta convertir esta simulación en un juego. Por ejemplo, puedes mostrar un objeto a una altura aleatoria en el extremo derecho del escenario, y decirle al usuario que intente alcanzarlo de un cañonazo. Si el jugador falla el disparo, el juego puede proporcionar algunas pistas para reajustar el ángulo de disparo y la rapidez inicial.

PROYECTO 17. BALONCESTO CON GRAVEDAD.

Muchos videojuegos, como Mario Bros. y otros, muestran una vista lateral de la escena, donde la parte baja de la pantalla es el suelo y los personajes se muestran de perfil. Estos juegos tienen gravedad: los personajes pueden saltar, y después caen hasta que llegan de nuevo al suelo. En este proyecto construiremos un juego de baloncesto con gravedad. El jugador saltará y lanzará la pelota hacia un aro, y la pelota y el jugador volverán a caer por efecto de la gravedad.

1) Hacemos que el gato caiga.

Abre un archivo nuevo de Scratch, y llámalo **Proyecto 17.sb2**. Comienza renombrando al objeto "Sprite1" (Objeto1) como "gato". Ahora, agrega el fondo "brick wall1" desde la librería de Scratch.

Selecciona al objeto "gato", y crea una variable local llamada "velocidad y". (Recordemos que la velocidad es una cantidad vectorial que indica la rapidez con la que algo se está moviendo, y en qué dirección lo está haciendo. "Velocidad y" es la componente vertical de la velocidad del gato. Cuando "velocidad y" sea un número positivo, el gato se moverá hacia arriba, y cuando "velocidad y" sea un número negativo, el gato se moverá hacia abajo).

Ahora, la gravedad hace que los objetos se *aceleren* hacia abajo, esto es, hacia el suelo. En este juego, el objeto "gato" debe moverse hacia abajo al caer, y la rapidez a la que lo hace deberá cambiar conforme se ejecuta el programa. Para ello, vamos a comenzar añadiendo al gato el siguiente programa: Al clicar en la bandera verde, inicializamos el valor de "velocidad y" a cero, y a continuación, arranca un bucle infinito. A cada pasada del bucle, cambiamos el valor de la coordenada y del gato (su posición vertical) por el valor almacenado en "velocidad y", y a continuación, variamos el valor de "velocidad y" en -2 unidades. De esta forma, y a cada nueva iteración del bucle, la posición en y del gato cambiará (negativamente) más y más rápido, haciendo que el gato caiga más y más rápido.

2) Añade el código para detectar el suelo y parar.

El programa recién escrito hace que el gato caiga, pero también queremos que deje de caer al llegar al suelo. Para ello, modifica el programa previo como sigue:

Dentro del bucle "por siempre", haz que la variable "velocidad y" del gato cambie en -2 unidades solo *si* la posición en y del gato es mayor que -130 (esto es, si el gato aún no ha llegado al suelo de nuestro escenario). De esta forma, si el gato aún no ha llegado al suelo, seguirá cayendo. En caso contrario, debemos fijar la coordenada y del gato a un valor de -130 (para ubicarlo justo sobre el suelo), y además, fijar el valor de "velocidad y" de nuevo a cero, para evitar que siga cayendo a través del suelo.

3) Haz que el gato salte.

Después de conseguir hacer que el gato caiga por efecto de la gravedad, hacer que el gato salte es muy fácil.

El programa comenzará al presionar la tecla ↑ del teclado. A continuación, comprobamos *si* al presionar esa tecla, el gato está sobre el suelo (sabremos si el gato está en el suelo porque, en ese caso, su coordenada y es igual a -130). De ser así, cambiamos el valor de la variable "velocidad y" en 20 unidades (recuerda que una velocidad positiva hace que el gato suba).

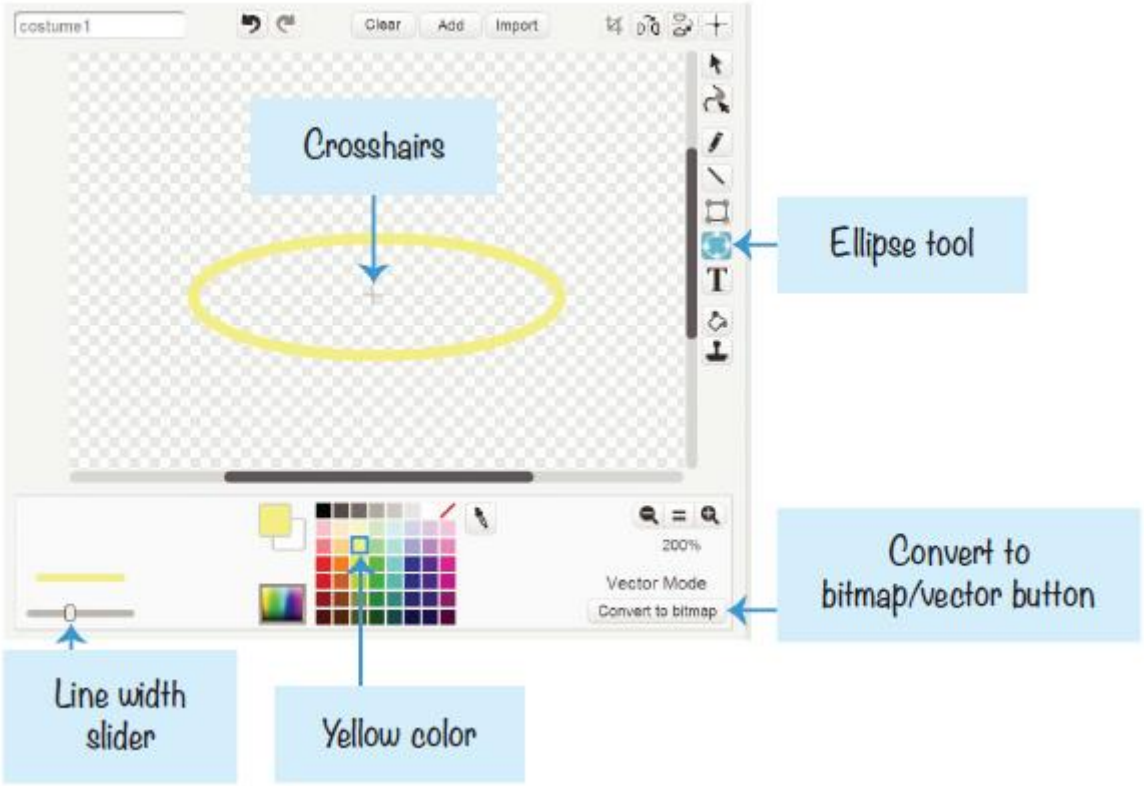
De esta forma, y cada vez que presionemos la tecla ↑ (pero solo si el gato está en el suelo), "velocidad y" cambiará en 20 unidades, y el gato saltará. Pero al saltar, el programa previo entrará en ejecución, y modificará el valor de "velocidad y" en -2 unidades a cada iteración del bucle. Por consiguiente, aunque al principio el gato salta con una "velocidad y" igual a 20, tras la primera iteración pasará a hacerlo a 18, a la siguiente a 16, y así sucesivamente. Cuando "velocidad y" se vuelva 0, el gato estará en el punto más alto de su salto. A continuación, y a cada nueva pasada del bucle, "velocidad y" cambia en -2 unidades, por lo que se hace más y más negativa, y el gato comienza a caer hasta que llega de nuevo al suelo.

4) Consigue que el gato se mueva hacia la derecha y hacia la izquierda.

A continuación, escribe un programa para que el gato se mueva hacia la derecha y hacia la izquierda al presionar las teclas → y ←, respectivamente. El gato se moverá en pasos de 10, y además, debemos animar su movimiento usando los dos disfraces que tiene disponibles. Ya debería saber cómo hacer este programa.

5) Haz un aro volador para encestar la pelota.

En primer lugar, crea el objeto "aro" mediante el editor gráfico de Scratch. Asegúrate de fijar el centro del disfraz en el centro de la elipse del aro. Dibuja el aro con un tamaño relativo similar al de la primera figura.



A continuación, cambia el nombre del objeto a "aro", y añádele el sonido "cheer" de la biblioteca de sonidos de Scratch. Este será el sonido que reproducirá cada vez que el jugador consiga una canasta.

Vamos a crear el código del aro, que tiene dos programas:

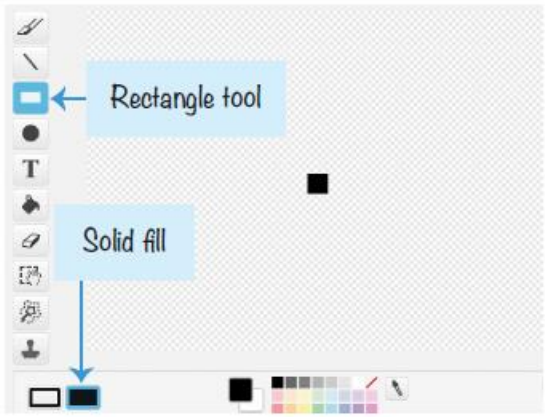
Programa 1: Al pinchar en la bandera verde, el aro siempre se estará deslizando a una posición (x, y) aleatoria (coordenada x entre -240 y 240, coordenada y entre -50 y 50).

Programa 2: Cuando el objeto "pelota" toque al objeto "sensor" y la aplicación detecte una canasta, la pelota enviará el mensaje "canasta" (ver pasos 6 y 7). Entonces, al recibir ese mensaje, el aro reproducirá el sonido "cheer" y mostrará por pantalla el texto "¡Canasta!" durante dos segundos.

6) Crea el objeto "sensor".

Veamos cómo determinar si el jugador encesta o no la pelota dentro del aro. Podríamos hacer un programa que simplemente compruebe si la pelota está tocando el aro, pero como el aro es tan ancho, si la pelota solo toca un borde por fuera, eso se contaría como una canasta. En realidad, queremos que se detecte una canasta solo cuando la pelota pasa por el centro del aro.

Para ello vamos a crear un "sensor", aunque el término apropiado en programación de videojuegos es *hitbox*. Un **hitbox** es una pequeña área rectangular que determina si dos objetos han colisionado entre sí. Para crear el "sensor" (hitbox), acudimos al editor gráfico de Scratch, y dibujamos un pequeño cuadrado negro:



El código del sensor es sencillo: el programa arranca al pinchar en la bandera verde, y mediante un bucle infinito, el sensor siempre acude a la posición del aro (esto es, al centro del aro). Por supuesto, el sensor no debe ser visible, por lo que el programa también debe ocultarlo. Esto puedes hacerlo, al menos, de dos formas, a saber, usando el bloque "esconder", o el bloque "fijar efecto (desvanecer) a (100)".

7) Crea y programa la pelota.

En primer lugar, agrega el objeto "basketball" de la biblioteca de objetos de Scratch, y añádele el sonido "pop" de la biblioteca de sonidos. Cambia el nombre del objeto a "pelota".

Ahora, selecciona el objeto "pelota" y crea una nueva variable local llamada "velocidad y". Aunque los nombres son iguales, las variables "velocidad y" de la pelota y del gato son independientes, porque las hemos definido locales. A continuación, crea una nueva variable *global* llamada "puntuación jugador1", y activa la casilla adyacente para mostrar por pantalla un monitor de esa variable.

Vamos con el código de la pelota, que consta de dos programas:

Programa 1: Este programa empieza al clicar en la bandera verde, y simplemente fija el valor de la variable "puntuación jugador1" a 0, y oculta el objeto pelota.

Programa 2: Este programa arranca al presionar la tecla espaciadora, y comienza reproduciendo el sonido "pop" y ubicando a la pelota en la posición del gato. A continuación, fija el valor de la variable "velocidad y" de la pelota a 24, y muestra la pelota. Después, un bucle "repetir hasta que ( )" hace que la pelota caiga hasta llegar al suelo. Para ello, el bucle se repetirá hasta que la posición en y de la pelota sea < -130 (esto es, hasta que la pelota llegue al suelo). A cada pasada del bucle, cambiamos la posición en x de la pelota en 8 unidades, la posición en y en una cantidad igual al valor de "velocidad y", cambiamos el valor de "velocidad y" en -2 unidades, y giramos la pelota $6°$ hacia la derecha. El programa sale del bucle cuando la pelota llega al suelo, y en ese momento debe desaparecer, por lo que la última instrucción debe ser esconder el objeto "pelota".

8) Detecta si el gato hace canasta.

Vamos a modificar el programa que acabamos de escribir para comprobar si el objeto "pelota" está tocando al objeto "sensor". Así es como sabremos si el gato ha conseguido una canasta, momento en el cual deberíamos incrementar en 1 el valor de la variable "puntuación jugador1". Pero hay una complicación: no deberíamos contar una canasta si la pelota atraviesa el aro hacia arriba.

Recordemos que si el valor de la variable "velocidad y" es positivo, la pelota se está moviendo hacia arriba. Si el valor de "velocidad y" es instantáneamente cero, la pelota no se está moviendo ni hacia arriba ni hacia abajo. Pero si el valor de "velocidad y" es negativo, la pelota está cayendo.

Así pues, modifica el programa de la pelota, añadiendo dentro del bucle un condicional que compruebe si la pelota está tocando al sensor, y además, si el valor de "velocidad y" es negativo. En ese caso, cambiamos el valor de "puntuación jugador1" en 1 unidad, y enviamos el mensaje "canasta". (Recordemos que cuando recibe el mensaje "canasta", el aro reproduce un sonido de celebración, y muestra el texto "¡Canasta!" por pantalla).

9) Arregla el error de programación relativo a la puntuación.

En informática, un **bug** es el término que se usa para designar un problema que hace que el programa se comporte de forma inesperada.

Tal vez hayas notado que la variable "puntuación jugador1" se incrementa en varios puntos del programa al encestar una sola canasta. El bucle "repetir hasta que ( )" continua repitiéndose hasta que la pelota llega al suelo, por lo que todo este código es para un solo lanzamiento del balón. El bucle comprueba varias veces si la pelota está tocando al sensor y cayendo, pero el valor de "puntuación jugador1" solo debería incrementarse en 1 unidad la primera vez. Ahora mismo, esto no ocurre, por lo que tenemos un error de programación, esto es, un *bug*.

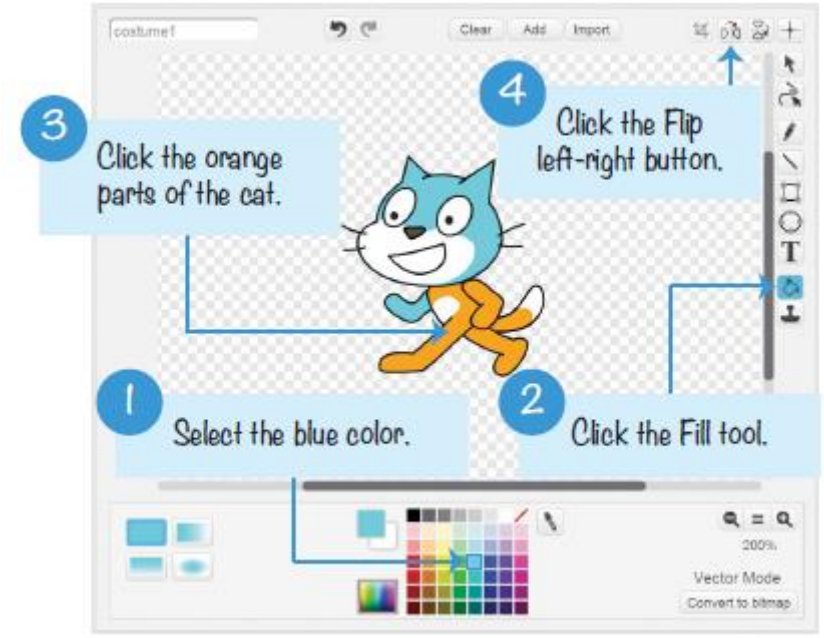
Para arreglar este problema, vamos a crear una nueva variable, llamada "consigueCanasta", que registre la primera vez que la pelota toca el sensor para un solo lanzamiento. A continuación, modifica el programa de la pelota: Al principio del programa (cuando el jugador acaba de presionar la tecla espaciadora), inicializamos el valor de "consigueCanasta" a "no" (el jugador acaba de lanzar la pelota, la pelota no ha podido llegar al aro y todavía no puede haber hecho una canasta). Además, completa la condición del bloque "si" para que solo entre si la pelota está tocando el sensor, si la "veocidad y" es negativa, y además, si la variable "consigueCanasta" toma el valor "no". De esta forma, la primera vez que la pelota detecte que hay canasta, incrementamos la puntuación, y cambiamos el valor de "consigueCanasta" a "sí" (para que no vuelva

a entrar en el condicional en sucesivas iteraciones). La variable "consigueCanasta" se reinicializará a cero la próxima vez que el jugador presione la tecla espaciadora para lanzar una nueva pelota.

AMPLIACIÓN 1: MODO DOS JUGADORES.

Vamos a mejorar el juego de baloncesto añadiendo un segundo jugador. Esta tarea será sencilla, porque el código del segundo jugador será casi idéntico.

1) Duplica los objetos "gato" y "pelota". A continuación, en el editor gráfico de Scratch, colorea los dos disfraces del objeto "gato2" para hacerlos azules. Después, y también en el editor, clica en el botón "voltear horizontalmente" para hacer que el "gato2" mire hacia la izquierda (haz esto para ambos disfraces).



2) Modifica el código del objeto "gato2": la teclas para mover el gato hacia la derecha y hacia la izquierda serán d y a, respectivamente. La tecla para saltar será w.

3) Modifica el código del objeto "pelota2".
El segundo jugador necesita su propia variable para contar su puntuación, a la que llamaremos "puntuación jugador2". Además, modifica el programa de "pelota2" para que la tecla para lanzar la pelota sea la e, inicializa adecuadamente las variables apropiadas, haz que la pelota2 acuda al objeto "gato2" al presionar la tecla de lanzamiento, y por último, consigue que la pelota2 vuele hacia la izquierda (en vez de hacerlo hacia la derecha como la pelota1).

PROYECTO 18. ASTEROIDES.

Asteroides es un videojuego clásico de Atari del año 1979. El jugador pilota una nave espacial que avanza *por el espacio*, y su objetivo es destruir los asteroides que se encuentra mientras esquiva los fragmentos resultantes. (Como veremos, añadir la especificación "por el espacio" hace que el juego sea más interesante).

En vez de controlar directamente hacia dónde se mueve la nave, el jugador la impulsará como si de un disco de hockey sobre hielo se tratase. Como la nave espacial tiene inercia, se deslizará sobre el escenario. Para ralentizar la nave, el jugador debe impulsarla en dirección opuesta. Se requiere de cierta habilidad para mover la nave sin perder el control, pero eso solo es una parte de la diversión del juego. La otra parte será destruir los asteroides.

1) Haz una nave espacial que se vea impulsada por el espacio.

Crea un nuevo proyecto de Scratch, y llámalo **Proyecto 18.sb2**. En primer lugar, vamos a poner el fondo de nuestro escenario. De la biblioteca de Scratch, selecciona el fondo "stars". No usaremos el objeto "gato" en este proyecto, así que bórralo de la lista de objetos.

Vamos a agregar el objeto "Nave". Para ello, acude al archivo **Proyecto 18.rar** y descomprímelo en una carpeta. A continuación, vuelve a Scratch y pincha en el botón "cargar objeto desde archivo" junto al texto "Nuevo objeto" en la lista de objetos. Selecciona la imagen **naveEspacial.png** de la carpeta donde descomprimiste el archivo RAR. Cambia el nombre del objeto a "nave".

Con el objeto "nave" seleccionado, y en la categoría "datos", crea las variables *locales* "velocidad x" y "velocidad y". Crea también las variables globales "puntuación" y "jugadorEstáVivo".

A continuación, vamos a crear el primer programa de la nave. Este código define la posición inicial de la nave y los valores iniciales de las variables, y también contiene la lógica que define los controles del usuario.

Programa 1 (nave): El programa arranca al clicar en la bandera verde, y comienza ubicando la nave en el origen y mostrándola por pantalla. A continuación, ajusta los valores iniciales de las variables: "jugadorEstáVivo" a "sí", "puntuación" a 10 (más tarde entenderemos por qué inicializamos la puntación a 10), "velocidad x" a 0, y "velocidad y" a 0. Después empieza un bucle infinito, dentro del cual comprobamos si el usuario presiona los controles de movimiento, y actuamos en consecuencia. El usuario utilizará como controles las teclas WASD. Si presiona la tecla d, la "velocidad x" de la nave aumentará en $0{,}5$ unidades. Si presiona la tecla a, la "velocidad x" de la nave cambiará en -0.5 unidades. Completa tú el código de las teclas w y s para cambiar la "velocidad y" de la nave. Por último, y todavía dentro del bucle, cambiamos la coordenada x de la nave con el valor almacenado en "velocidad x", y la coordenada y de la nave con el valor almacenado en "velocidad y".

Puede que nos preguntemos por qué el código para controlar el movimiento de la nave difiere del código de otros programas previos. La razón es la siguiente: En este programa, el acto de presionar una de las teclas WASD suma o resta $0{,}5$ unidades a las variables "velocidad x" o "velocidad y". Entonces, al final del bucle, el programa cambia las posiciones x e y de la nave usando los valores de estas variables. Y éste es el punto

crucial: Incluso después de haber soltado la tecla, las variables siguen reflejando la posición actualizada, y la nave continúa moviéndose.

2) Consigue que la nave pase de un borde al borde opuesto.

Hasta ahora, la nave se detiene al llegar a los bordes del escenario. Esto es así porque Scratch siempre evita que los objetos se muevan fuera del escenario, lo que suele ser útil en la mayoría de los programas. Pero en el juego de los asteroides queremos que, conforme los objetos van saliendo por un extremo del escenario, vayan apareciendo en el extremo opuesto (ver figura).

Añade un segundo programa para la nave que logre este comportamiento.

Programa 2 (nave): El programa comienza al clicar en la bandera verde. A continuación, arranca un bucle infinito, donde comprobamos si el objeto llega a los extremos derecho, izquierdo, superior, o inferior, y actuamos en consecuencia. Por ejemplo, para comprobar si la nave llega al extremo izquierdo, chequeamos si la posición en x de la nave es menor que -235, y en ese caso, fijamos la posición x a 235. Completa el programa actuando análogamente en todos los extremos.

3) Añade pequeños impulsos aleatorios al movimiento de la nave.

Tal y como hemos programado a la nave hasta ahora, el control de su movimiento ya es complicado. Pero para hacer el juego aún más desafiante, añadiremos a la nave pequeños impulsos aleatorios, de forma que la nave no pueda simplemente permanecer estática en el centro del escenario sin moverse en absoluto. Para ello, añadiremos un tercer programa a la nave:

Programa 3 (nave): Este programa arranca al clicar en la bandera verde, y comienza con un bucle infinito. Tras esperar 1 segundo, el programa simplemente cambia el valor de "velocidad x" y de "velocidad y" en un número aleatorio entre $-1{,}5$ y $1{,}5$, y esto lo hace a cada pasada del bucle. (Esto significa que, a cada segundo, el movimiento de la nave recibirá un impulso aleatorio).

4) Apunta con el ratón y dispara proyectiles con la tecla espaciadora.

El código que controla el movimiento de la nave ya está completo. Ahora vamos a hacer que la nave dispare proyectiles de energía. Estos proyectiles harán explotar a los asteroides *en el espacio*.

Empezamos creando el objeto "proyectil". Para ello, añadimos de la biblioteca de Scratch el objeto "ball", y cambiamos el nombre del objeto a "proyectil". Además, agregamos al objeto el sonido "laser1" de la biblioteca de sonidos de Scratch.

Cada vez que disparemos tendremos que hacer clones del objeto "proyectil", pero el objeto original y los clones ejecutarán programas distintos. Solo hay un objeto "proyectil", pero el jugador debe ser capaz de disparar muchos proyectiles al mismo tiempo. Para ello, crearemos clones del proyectil original, para que pueda haber varios proyectiles volando simultáneamente por pantalla. El objeto original permanecerá oculto; todos los proyectiles que aparezcan por pantalla serán clones.

Crea una variable *local* llamada "soyUnClon", para poder discriminar qué proyectil es el original y cuáles son clones. El proyectil original ajustará el valor de esta variable a "no", y los clones a "sí". Escribe los siguientes programas para el objeto "proyectil":

Programa 1 (proyectil): Al clicar en la bandera verde, el programa comienza inicializando la variable "soyUnClon" a "no", ajusta su tamaño a un 10% de su tamaño original, y oculta el objeto.

Programa 2 (proyectil): Este programa permitirá disparar proyectiles cada vez que presionemos la tecla espaciadora. Para ello, el programa arrancará al presionar la tecla espaciadora, y comienza comprobando *si* el jugador está vivo (variable "jugaadorEstáVivo") *y* si el objeto "proyectil" *no* es un clon (variable "soyUnClon"). En caso de cumplirse esta condición múltiple, el programa envía al proyectil original a la posición de la nave, lo hace apuntar en la dirección del ratón, y crea un nuevo clon. (Notar que este programa solo le permite al usuario disparar proyectiles si está vivo, y solo permite que sea el proyectil original el que cree clones de sí mismo).

Programa 3 (proyectil): En este programa haremos que los clones del proyectil se muevan a la posición del ratón tras ser disparados. Al comenzar como un clon, el proyectil fija el valor de la variable "soyUnClon" a "si", se muestra, reproduce el sonido "laser1", y arranca un bucle de 50 repeticiones. A cada iteración de bucle, el clon se mueve 10 pasos, y ejecuta un código (idéntico al de la nave) que le permite desaparecer de un extremo y aparecer en el extremo opuesto. Al terminar las 50 iteraciones, el proyectil de energía decae y desaparece, para lo cual debemos borra el clon. (De esta forma nos aseguramos de que cada proyectil solo tiene un tiempo de vida limitado, para que no esté moviéndose para siempre por el escenario).

4) Crea asteroides que floten en el espacio.

Ahora vamos a crear los asteroides que flotarán por el espacio hasta que un proyectil de energía impacte contra ellos. Al ser alcanzados, los asteroides se romperán repetidamente en dos trozos más pequeños hasta que sean lo suficientemente pequeños como para vaporizarse.

Empezaremos creando el objeto "asteroide". Para ello, pincha en el botón "cargar objeto desde archivo" y selecciona el archivo **asteroid.png** en la carpeta donde descomprimiste el archivo RAR. Renombra el objeto como "asteroide".

Con el objeto "asteroide" seleccionado, acude a la categoría datos y crea una variable local llamada "impactos". Haz lo mismo para crear las variables *locales* "velocidad x", "velocidad y", y "rotación". Más adelante usaremos la variable "impactos" para contabilizar el número de veces que ha sido alcanzado un asteroide, así como su tamaño.

Ahora vamos a escribir el código que hace que el objeto "asteroide" cree nuevos clones que aparecen por pantalla con una velocidad y una rotación aleatorias. Con ello crearemos un impredecible enjambre de asteroides *en el espacio*.

Programa 1 (asteroide): Este programa es el que actúa sobre el asteroide original. Al clicar en la bandera verde, el programa comienza ocultando el objeto, fijando su tamaño al 100% de su tamaño original, y fijando el valor inicial de la variable "impactos" a 0. A continuación arranca un bucle infinito. A cada pasada del bucle, el programa ubica al asteroide en una posición con coordenadas $x = -235$ e y un número aleatorio entre -175 y 175, crea un nuevo clon de sí mismo, y espera un tiempo aleatorio de entre 8 y 12 segundos.

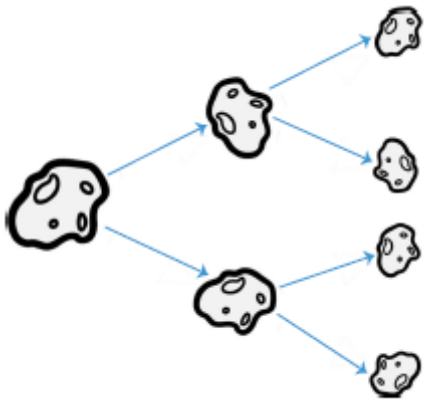
Programa 2 (asteroide): Este programa hace que los clones creados empiecen a moverse. Cuando se crea un nuevo clon, el programa comienza mostrándolo, y ajustando los valores iniciales de las variables "velocidad x", "velocidad y", y "rotación" a un número aleatorio entre -3 y 3. A continuación, un bucle infinito gira al

clon un número de grados igual al valor de "rotación", cambia su coordenada x en un valor igual a "velocidad x", y cambia su coordenada y en un valor igual a "velocidad y".

Programa 3 (asteroide): Al igual que para la nave y para los proyectiles, este programa hará que los clones pasen de un extremo al extremo opuesto. El programa es idéntico a los que ya hemos creado previamente.

5) Haz que los asteroides se partan en dos al ser alcanzados.

Cuando un clon del proyectil alcance a un asteroide, éste creará dos clones más pequeños de sí mismo, haciendo que parezca que el asteroide se ha partido en dos.



En primer lugar, vamos a la pestaña de sonidos y añadimos desde la biblioteca de Scratch el sonido "chomp", que se reproducirá siempre que un clon del proyectil impacte contra un asteroide. A continuación, añade el siguiente código al objeto "asteroide":

Programa 4 (asteroide): Al comenzar como clon, el programa empieza arrancando un bucle infinito. A cada iteración, el programa comprueba si el clon está tocando un proyectil de energía. En ese caso, realiza todas estas tareas: En primer lugar, reproduce el sonido "chomp", envía el mensaje "asteroide alcanzado", y cambia los valores de las variables "puntuación" e "impactos" en 2 y 1 unidades, respectivamente. A continuación, comprueba si el número de impactos recibidos por el asteroide es igual a 4, y en su caso, borra el clon. Por último, cambia el tamaño del clon en -25, crea dos clones de sí mismo (de tamaño más reducido), y borra el clon "padre". Con esto termina el condicional, y también el bucle.

Notar que, según este código, cuando un clon ya ha sido alcanzado 4 veces, el programa borra ese clon y ya no crea dos nuevos clones más pequeños. De esta forma evitamos que los asteroides se estén duplicando indefinidamente. (Sin embargo, si queremos que el número de asteroides crezca exponencialmente, solo tenemos que aumentar el número máximo de impactos que el asteroide puede recibir).

A continuación, seleccionamos el objeto "proyectil" de la lista de objetos, y lo completamos con el siguiente programa:

Programa 4 (proyectil): Cuando el asteroide es alcanzado por un clon del proyectil, el primero envía el mensaje "asteroide alcanzado". Todos los clones del proyectil recibirán este mensaje, pero solo debemos borrar el clon que, en ese momento, esté tocando al asteroide. De esta forma, el clon que ha impactado contra el asteroide desaparece de pantalla. Escribe el programa que efectúa esta tarea.

6) Lleva la cuenta de la puntuación y construye un temporizador.

El juego de los asteroides se vuelve más y más difícil cuantos más asteroides haya volando por pantalla. Una buena estrategia de juego es deshacerse en primer lugar de los asteroides más pequeños antes de disparar a los más grandes. Pero para meter más presión al jugador, vamos a hacer que la variable "puntuación" disminuya en 1 unidad a cada segundo, de forma que si llega a cero, el jugador también pierde la partida.

(Quizás ahora entendamos por qué inicializamos la variable puntuación a 10). De esta forma, obligamos al jugador a apresurarse para destruir más asteroides y ganar más puntos.

En primer lugar, vamos a crear el objeto "tiempo terminado". Para ello, clicamos en el botón "dibujar nuevo objeto", y en el editor gráfico de Scratch, usando la herramienta de texto, escribimos con letras rojas y mayúsculas el mensaje "el tiempo se ha acabado".

Ahora, escribe el siguiente programa para el objeto "tiempo terminado":

Programa 1 (tiempo terminado): El programa empieza al clicar en la bandera verde, y lo primero que hace es ocultar el objeto. A continuación, a cada pasada de un bucle infinito, esperamos 1 segundo, cambiamos la variable "puntuación" en -1 unidades (le restamos 1 unidad), y comprobamos si la "puntuación" se ha hecho igual a cero. En ese caso, mostramos el objeto por pantalla, y detenemos la aplicación.

7) Haz que la nave explote al chocar contra un asteroide.

La partida termina si el jugador no destruye asteroides con la suficiente rapidez como para evitar que su puntuación caiga a cero, pero también si la nave es alcanzada por un asteroide. Vamos a añadir el código para detectar este suceso y para mostrar por pantalla una explosión animada.

En primer lugar, debemos agregar a nuestro proyecto el objeto "explosión", que constará de 8 disfraces. Este objeto (junto con sus disfraces) están disponibles en el archivo **explosión.sprite2**. Para cargarlo en nuestra aplicación, clica en el botón "cargar objeto desde archivo" y selecciona el archivo indicado.

Ahora, selecciona el objeto "explosión" de la lista de objetos, y añádele el siguiente código:

Programa 1 (explosión): Este programa empieza al clicar en la bandera verde, y simplemente oculta el objeto, porque en circunstancias normales, no debe aparecer por pantalla.

Programa 2 (explosión): La explosión permanece oculta hasta que recibe el mensaje "explotar". Cuando esto ocurre, reproduce el sonido "alien creak2", acude a la posición de la nave, se pone el disfraz1, se muestra, y mediante un bucle repetir va cambiando a sus otros 7 disfraces (para crear una animación de explosión). Cuando termina la ronda de cambios de disfraz, se oculta, y espera 2 segundos.

El mensaje "explotar", que activa la animación del objeto "explosión", lo envía el objeto "nave" cuando es alcanzado por un asteroide. Por lo tanto, debemos añadir a la nave el código necesario para ello:

Programa 4 (nave): El programa se activa al clicar en la bandera verde, y mediante un bucle infinito, comprueba constantemente si la nave está tocando un asteroide. En ese caso, ocultamos el objeto "nave" (para reemplazarlo por la animación de la explosión), ajustamos el valor de la variable "jugadorEstáVivo" a "no", enviamos el mensaje "explotar" y quedamos a la espera de que el objeto "explosión" ejecute su programa, tras lo cual, paramos la aplicación.

8) Con esto, el proyecto ya está acabado. Guárdalo como **Proyecto 18.sb2**.

AMPLIACIÓN: MUNICIÓN LIMITADA.

Una de las características que pueden hacer que el juego se vuelva fácil con el tiempo es que podemos disparar tantos proyectiles como queramos (uno cada vez que presionamos la tecla espaciadora). Para obligar al jugador a apuntar cuidadosamente y no disparar a lo loco, podemos limitar el número de proyectiles.

Para ello, podemos definir una nueva variable, llamada "energía", que disminuya en 1 unidad cada vez que el jugador dispare un proyectil de energía. Si la "energía" llega a cero, el jugador ya no podrá disparar más proyectiles. Sin embargo, permitiremos que el valor de la variable "energía" vaya aumentando lentamente con el tiempo para que el jugador pueda volver a disparar al cabo de un rato.

Crea la variable "energía" con ámbito global. Esta variable debe empezar con un valor de 10 al comienzo del juego, y disminuir en 1 unidad cada vez que el jugador dispare un proyectil. El jugador debería ser capaz de disparar solo si el valor de "energía" es mayor que 0. Completa el código del objeto "proyectil" para incorporar estas funcionalidades:

Programa 5 (proyectil): Este programa simplemente se encarga de inicializar el valor de la variable energía a 10, e irla aumentando lentamente siempre que su valor sea menor que el valor máximo de energía que se puede disponer, que también es 10. El programa se activa al clicar en la bandera verde, y comienza fijando el valor de "energía" a 10. A continuación, el programa arranca un bucle infinito, y a cada iteración, espera $0{,}4$ segundos, y comprueba si el nivel de energía es menor que 10. En ese caso, incrementa el valor de energía en 1 unidad.

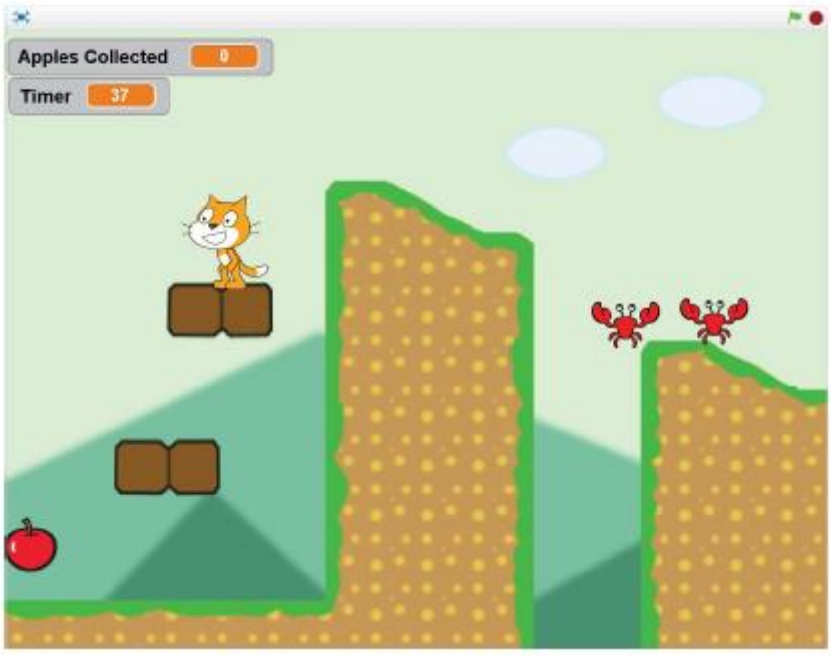
Ahora, tendrás que modificar el programa 2 del proyectil, para asegurarnos de que el jugador solo pueda disparar si el nivel de energía es mayor que cero. Además, y a cada disparo, debemos reducir el nivel de energía en 1 unidad.

PROYECTO 19. JUEGO DE PLATAFORMAS.

El primer videojuego de Mario Bros. fue desarrollado en 1985 por Nintendo. Hoy día es su franquicia más famosa y uno de los juegos más influyentes de todos los tiempos. Como el personaje corre, salta, y va de una plataforma a otra, a este tipo de videojuegos se les conoce como *juegos de plataformas*.

En este proyecto construiremos un videojuego en el que el gato desempeña el papel de Mario. El jugador podrá hacer que el gato se mueva y salte en un único nivel mientras recoge manzanas y trata de evitar que unos cangrejos se las roben. La partida tiene una duración limitada: el jugador solo dispone de 45 segundos para recoger todas las manzanas que pueda mientras trata de evitar a los cangrejos.

Verás que este proyecto es, de lejos, el más ambicioso de todos los que hemos hecho hasta ahora, y requerirá el uso de todas las herramientas y conceptos de programación que hemos estudiado.

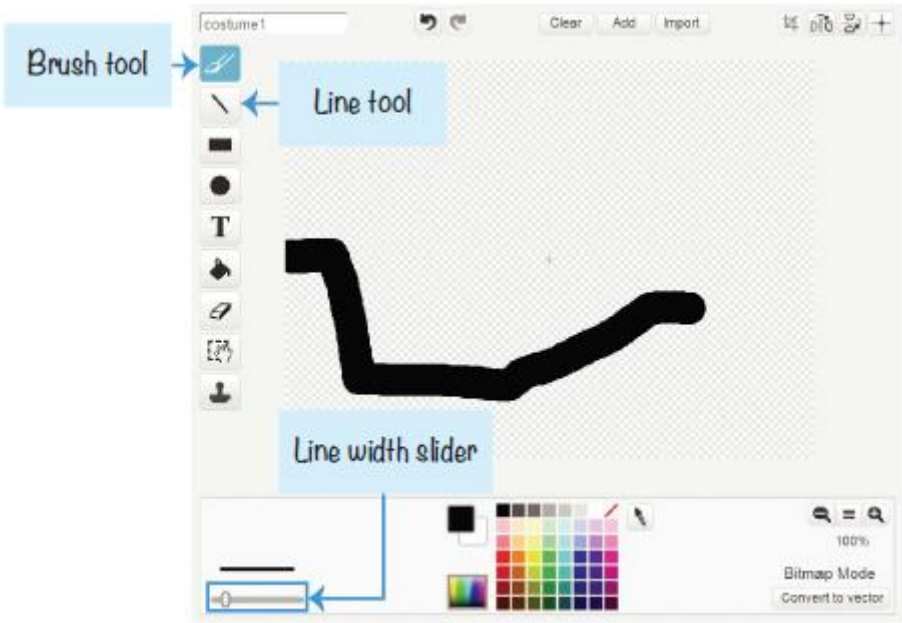


A. Crea la gravedad, la caída, y el aterrizaje.

Para comenzar, vamos a añadir el código para gestionar la gravedad, la caída, y el aterrizaje, de forma similar a como lo hicimos en el proyecto 17 del baloncesto con gravedad. La única diferencia (importante) es que, en el juego de plataformas, el gato aterriza cuando toca un objeto "suelo", en lugar de hacerlo al tocar la parte inferior del escenario. Codificar este hecho es complicado, porque queremos que le suelo tenga rampas y plataformas.

1) Crea el objeto "suelo".

En los primeros pasos de nuestro videojuego vamos a utilizar un objeto "suelo" provisional muy sencillo, simplemente para ver si el código que escribimos funciona o no. Para crear este objeto "suelo" de pruebas, clicamos en el botón "dibujar nuevo objeto". En el editor gráfico de Scratch, usa las herramientas pincel y/o línea para dibujar la forma del suelo. Asegúrate de dibujar una rampa de pendiente suave a la derecha y una rampa con pendiente pronunciada a la izquierda, similares a las de la figura. Al terminar, renombra el objeto como "suelo".



2) Añade el código para la gravedad y el aterrizaje.

Selecciona el objeto "gato", y crea una variable local llamada "velocidad y". A continuación, añade el siguiente código para el gato:

Programa 1 (gato): Al clicar en la bandera verde, ajustamos el estilo de rotación "izquierda - derecha", y fijamos el valor inicial de "velocidad y" a 0. A continuación, arranca un bucle infinito, que en cada iteración cambia el valor de "rapidez y" en -2 unidades, y cambia la coordenada y del gato con el nuevo valor de "velocidad y". Después, y aún dentro del bucle infinito, arranca un bucle interno que se repite hasta que el gato no esté tocando el suelo. A cada pasada, el bucle ajusta el valor de "velocidad y" a cero (para que el gato deje de caer) y cambia la coordenada y del gato en 1 unidad (para elevarlo ligeramente). Con esto finalizan el bucle interno y el externo.

Este programa realiza dos acciones dentro del bucle infinito: En primer lugar, y por efecto de la gravedad, el objeto gato cae más y más rápido hasta llegar al suelo. En segundo lugar, el programa hace que el gato deje de caer, y lo eleva ligeramente en el caso de que haya caído a una cierta profundidad por debajo de la superficie del suelo. Así es como conseguimos que el gato siempre termine su caída en la parte superior del suelo, sin importar cuál sea la forma del objeto "suelo".

En definitiva, este programa consigue que el gato caiga, llegue hasta el suelo, deje de caer, y se eleve ligeramente si es necesario, hasta terminar de pie justo sobre la parte superior del suelo independientemente de forma del objeto suelo (ver figura).

3) Haz que el gato camine y se traslade de un extremo al opuesto.

Programa 2 (gato): El gato también debe caminar hacia la derecha y hacia la izquierda mediante las teclas WASD. Añade el código necesario para conseguirlo. Por ejemplo, siempre que presionemos la tecla A, el gato debería mirar hacia la izquierda, y moverse horizontalmente en esa dirección 6 pasos. (Para ello, cambia el valor de la coordenada x del gato en 6 o -6 unidades, según el caso).

Programa 3 (gato): Este programa debe conseguir que el gato aparezca en la parte alta del escenario ($y = 170$) si cae hacia la parte baja del escenario en una ubicación donde no hay suelo ($y < -170$). El código es muy similar al que ya usaste en el juego de los asteroides, pero además de cambiar la coordenada y del objeto, también tendrás que fijar el valor de la variable "velocidad y" a 0. Más adelante escribiremos el código para pasar de un extremo lateral al otro.

4) Elimina el retardo que se produce al elevar al gato.

El problema de nuestro programa es que el proceso de elevar al objeto "gato" cuando cae a una cierta profundidad dentro del suelo es muy lento. El programa debería ejecutarse mucho más rápido, de forma que el usuario no vea cómo se eleva el objeto tras caer dentro del suelo. Para corregir este defecto, vamos a usar un procedimiento que se encargue de la tarea.

Crea el procedimiento "gestionaSuelo", y selecciona la opción "correr instantáneamente". Define el procedimiento con el siguiente código:

Procedimiento "gestionaSuelo": el procedimiento incluye el bucle "repetir hasta que ( )" que ya utilizamos en el programa 1 del gato, y que se encargaba de detener la caída del gato y de elevarlo hasta que dejase de tocar el suelo.

Como este código lo hemos transferido a un procedimiento, elimínalo del programa 1 del gato, y en su lugar, efectúa una llamada al procedimiento "gestionaSuelo".

Con esto conseguimos que la parte del código encargada de elevar al gato se ejecute en "modo turbo" (esto es, sin refrescar la pantalla). De esta forma, parecerá que el gato se eleva a la parte alta de suelo instantáneamente.

B. Gestiona correctamente las pendientes pronunciadas y los muros.

El objeto "suelo" puede tener rampas ascendentes y descendentes por las que el gato debería poder caminar. Esta situación es totalmente nueva, porque hasta ahora, todos los objetos de nuestros proyectos

previos caminaban por un suelo perfectamente liso. El problema que presenta nuestro código actual es que el gato puede caminar por la pendiente pronunciada de la izquierda con la misma facilidad que lo hace por la suave pendiente de la derecha, y esto no es muy realista. Queremos conseguir que una pendiente pronunciada bloquee el avance del gato, y para ello, debemos introducir unos cambios en el código.

5) Añade el código para gestionar las pendientes pronunciadas.

Necesitamos cambiar el código que hace que el gato camine, y añadir algo de código nuevo. Para ello, vamos a usar un nuevo procedimiento llamado "caminar" que recibirá como entrada un parámetro denominado "pasos".

(Asegúrate de activar la opción "correr instantáneamente" al definir este procedimiento). También necesitaremos crear una nueva variable local llamada "elevaciónDelSuelo", que usaremos para determinar si la pendiente es demasiado pronunciada como para que el gato pueda ascenderla.

Para empezar, vamos a ver cómo modificar el programa 2 que hace que el gato camine:

Programa 2 (gato): Este programa le permitía al gato caminar hacia la derecha al presionar la tecla d y hacia la izquierda al presionar la tecla a. Por ejemplo, al presionar la tecla a, el gato simplemente apuntaba hacia la izquierda, y se movía 6 pasos (cambiando el valor de la coordenada x en -6 unidades). Pero ahora, en vez de cambiar la coordenada x del gato, el programa hará una llamada al procedimiento "caminar", pasándole como argumento al número de pasos (6 o -6, según el caso):

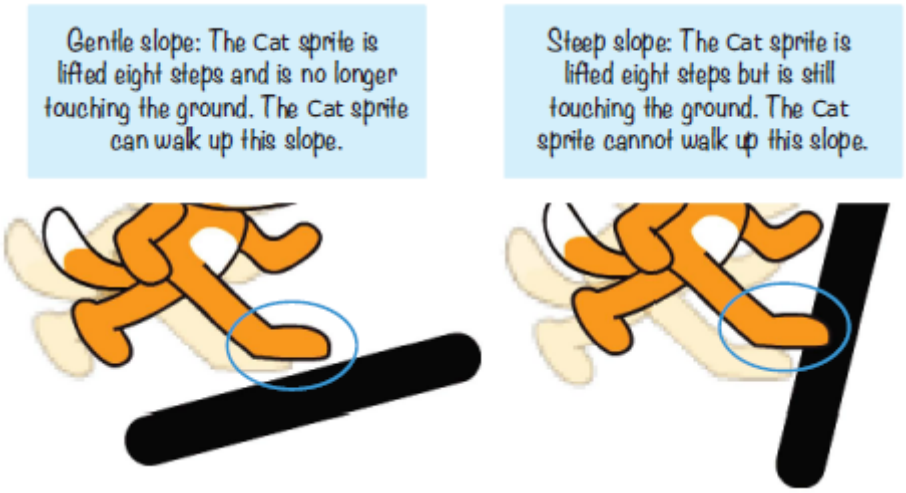
Por supuesto, ahora debemos escribir el código que le da contenido y funcionalidad al procedimiento "caminar ( )":

Procedimiento "caminar ( )": El procedimiento comienza cambiando la coordenada x del gato en un número de unidades igual al valor del argumento "pasos", tal y como hacíamos antes. A continuación, el procedimiento inicializa la variable "elevaciónDelSuelo" a 0, y arranca un bucle que se repetirá hasta que el gato no esté tocando el suelo o hasta que el valor de "elevaciónDelSuelo" se haga igual a 8. A cada iteración, el bucle cambia el valor de la coordenada y en una unidad, y modifica el valor de la variable "elevaciónDelSuelo" en una unidad. Cuando el procedimiento sale del bucle, ejecuta un condicional, en el que comprueba si el valor de "elevaciónDelSuelo" es igual a 8. En ese caso, cambia el valor de la coordenada x del gato en una cantidad de $-1 \times$ "pasos", y el valor de la coordenada y en -8. Con esto finaliza el condicional y el procedimiento.

Vamos a explicar este procedimiento: El código dentro del bucle usa la variable "elevaciónDelSuelo" para determinar si una pendiente es lo suficientemente pronunciada como para bloquear el avance del gato (este código también sirve para bloquear el avance del gato por los muros). La variable "elevaciónDelSuelo" comienza a 0, y se incrementa en 1 cada vez que el bucle eleva al gato cambiando su coordenada y en 1 unidad. Este bucle continúa ejecutándose repetidamente hasta que el gato ya no está tocando el suelo o hasta que la variable "elevaciónDelSuelo" se hace igual a 8.

Si "elevaciónDelSuelo" es menor que 8, la pendiente no es muy pronunciada, y el gato puede caminar por ella, por lo que el procedimiento no tiene que hacer nada más.

Pero si "elevaciónDelSuelo" es igual a 8, el bucle deja de repetirse. Esta situación se interpretaría como "el objeto ha sido elevado hasta una altura de 8, pero todavía está tocando el suelo, por lo que debe estar en una zona de pendiente muy pronunciada", ver figura. En ese caso, debemos deshacer la elevación y el desplazamiento horizontal que le hemos dado al gato. Esa es la tarea de las instrucciones dentro del bloque condicional. (Notar que al cambiar el valor de la coordenada x en una cantidad $-1 \times$ $(el\ valor\ del\ argumento\ "pasos")$ deshacemos el desplazamiento horizontal que le habíamos dado al gato en la primera instrucción del procedimiento).



C. Haz que el gato pueda dar saltos altos y bajos.

Una vez terminado el código que le permite al gato andar, vamos con el código que le permitirá saltar. En el proyecto del baloncesto, esto lo hacíamos cambiando el valor de la variable "velocidad y" a un número positivo. Pero eso implicaba que el jugador siempre saltaba una altura igual al valor de la variable. Sin embargo, en muchos juegos de plataformas el jugador puede dar un salto más pequeño o más alto presionando levemente o sostenidamente el botón de salto. Ése es el objetivo de este apartado.

6) Añade el código para saltar.

Comenzamos creando una variable local llamada "enElAire", cuyo valor ajustaremos a 0 siempre que el gato esté en el suelo. Pero el valor de "enElAire" aumentará (y después disminuirá) siempre que el gato salte (para después caer de nuevo al suelo). Cuanto mayor sea el valor de "enElAire", más lejos estará el gato del suelo.

A continuación, añade el siguiente código para el gato:

Programa 4 (gato): Al clicar en la bandera verde, el programa arranca un bucle infinito. En cada iteración, el bucle comprueba si el usuario ha presionado la tecla W y si el valor de "enElAire" es menor que 8. En ese caso, el programa ajusta el valor de "velocidad y" a 14.

A continuación, vamos a modificar dos de los programas que ya hemos escrito para el gato, para añadir la variable "enElAire" que limita lo alto que el gato puede llegar a saltar:

Programa 1 (gato): Éste es el programa que hace que el gato caiga más y más rápido por efecto de la gravedad, y que detiene su caída (y lo asciende penetra dentro del suelo para dejarlo sobre la superficie del suelo). Dentro del bucle infinito, debemos añadir un bloque que en cada iteración cambie el valor de la variable "enElAire" en 1 unidad.

Procedimiento "gestionaSuelo": La tarea de este procedimiento era eliminar el retardo que ocurría al elevar al objeto para dejarlo sobre la superficie del suelo. Dentro del bucle, debemos añadir un bloque que ajuste el valor de la variable "enElAire" a 0.

¿Qué conseguimos con estas modificaciones? Cuando el jugador presiona la tecla W por primera vez para que el gato salte, el programa 14 hace que la variable "velocidad y" tome el valor 14. Esto hace que el bucle dentro del programa 1 cambie la coordenada y del objeto con ese valor positivo de "velocidad y", moviéndolo hacia arriba. Durante las primeras pasadas del bucle el valor de la variable "enElAire" se estará incrementando, pero todavía será menor que 8. Por consiguiente, si el jugador sigue presionando la tecla W, el valor de "velocidad y" seguirá ajustándose a 14 en vez de disminuir por acción del bloque "cambiar (velocidad y) por (-2)". Con esto conseguimos que el gato vaya salte más alto si el jugador mantiene presionada la tecla W. Pero al final, el valor de "enElAire" se hará mayor o igual que 8, momento en el que no importará si el jugador sigue presionando la tecla W o no. Recuerda que el programa 4 solo ajusta el valor de "velocidad y" a 14 si el jugador presiona la tecla W y si el valor de "enElAire" es menor que 8.

Llegados a este punto, el código del programa 1 hará que el valor de "velocidad y" comience a disminuir, y el gato terminará cayendo cuando su valor se haga negativo. Cuando el gato llega al suelo entra en juego el procedimiento "gestionaSuelo", donde la variable "enElAire" se resetea a 0.

D. Añade el código para detectar el techo.

Ahora mismo el gato puede andar sobre el suelo, y las paredes bloquean su movimiento. Pero si el jugador hace saltar al gato cuando está bajo una plataforma, el gato la atravesará y se verá elevado hacia arriba. Para resolver este problema, debemos hacer algunos ajustes al código que hace subir al gato para añadir la detección del techo.

7) Añade una plataforma al objeto "suelo".

Modifica el objeto "suelo" para añadir una plataforma como la mostrada en la figura. Asegúrate que está lo suficientemente alta como para que el gato pueda caminar bajo ella, pero también lo suficientemente baja como para que el gato pueda saltar y su cabeza llegue a golpearla.

8) Añade el código de detección del techo.

El problema reside en el procedimiento "gestionarSuelo". Este código asume que el gato está cayendo desde arriba, y cuando el gato toca el suelo, el procedimiento lo eleva hacia arriba para dejarlo justo encima. Pero el objeto "suelo" debería representar cualquier parte sólida a través de la cual el gato no pueda pasar, techos incluidos. Por consiguiente, debemos cambiar el código para conseguir que, si el gato está saltando hacia arriba y toca el objeto "suelo", deje de ascender porque su cabeza ha chocado contra

un techo. Sabemos cuándo el gato se está moviendo hacia arriba porque su "velocidad y" es mayor que cero. Por lo tanto, vamos a modificar el procedimiento "gestionarSuelo" para añadirle un nuevo parámetro de entrada booleano llamado "subiendo".

(Asegúrate de activar la opción "correr instantáneamente"). A continuación, modifica el procedimiento "gestionarSuelo ( )" como se te indica:

Procedimiento "gestionarSuelo ( )": Dentro del bucle "repetir hasta que ( )", añadimos un condicional "si / si no" que verifica el valor de la variable booleana "subiendo". Si su valor es VERDADERO, el procedimiento cambia el valor de la coordenada y del gato en -1 unidades. En caso contrario, el procedimiento cambia el valor de la coordenada y del gato en $+1$ unidades y ajusta el valor de la variable "enElAire" a 0. Fuera del condicional, pero aún dentro del bucle, el procedimiento ajusta el valor de "velocidad y" a 0. Con esto, termina el bucle y también el procedimiento.

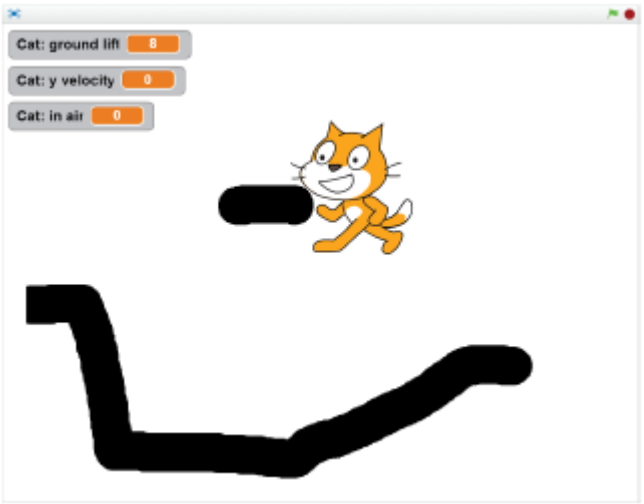
¿Qué conseguimos con esto? Si el gato se está moviendo hacia arriba (subiendo = VERDADERO), el hecho de cambiar su coordenada y en -1 unidad hace que el gato sienta que se ha golpeado la cabeza. En caso contrario, el procedimiento se comporta como hasta ahora, elevando al gato para que quede justo sobre la superficie del suelo.

A continuación, debemos modificar la llamada al procedimiento "gestionarSuelo ( )" en el programa 1 del gato. Añadiremos una condición booleana para determinar si el objeto se está moviendo hacia arriba:

El bloque "gestionarSuelo $(velocidad\ y > 0)$" fija el valor del argumento "subiendo" a VERDADERO si el valor de la variable "velocidad y" es mayor que 0 (esto es, si el objeto está saltando y moviéndose hacia arriba). Si "velocidad y" no es mayor que 0 es porque el objeto está cayendo o inmóvil, lo que hace que el argumento "subiendo" se fije a FALSO.

Esta es la forma en la que el procedimiento "gestionaSuelo ( )" decide si debería ejecutar el bloque "cambiar y por (-1)" (para evitar que el gato pueda pasar a través del techo) o el bloque "cambiar y por (1)" (para elevar al gato y sacarlo de dentro del suelo). En cualquier caso, si el gato está tocando el suelo (que es lo que está pasando si el código dentro del bloque "repetir hasta que (no (¿tocando (suelo)?))" se está ejecutando), la variable "velocidad y" debería fijarse a 0 para que el gato deje de caer o de saltar.

E. Añade un *hitbox* al objeto "gato".

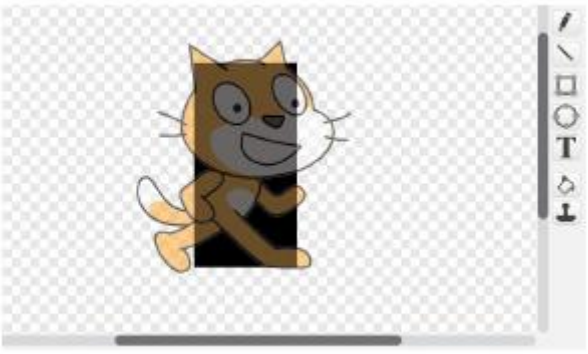


El juego tiene otro problema. Para detener la caída del gato, el código comprueba si el objeto "gato" está tocando el objeto "suelo". Pero el gato podría tocar el suelo con cualquier parte de su cuerpo, incluso con sus bigotes o mejillas. Por ejemplo, en la figura el gato no cae porque su mejilla ha "aterrizado" sobre la plataforma, y este comportamiento no es muy realista.

Por suerte, podemos arreglar este problema usando un **hitbox** como el que ya usamos en el juego de baloncesto.

9) Añade un hitbox al disfraz del objeto "gato".

Clica en la pestaña "disfraces" del objeto "gato", y selecciona la opción "pintar nuevo disfraz". Dibuja un rectángulo negro que cubra la mayor parte (pero no toda) del área de los otros dos disfraces. Observa la figura para hacerte una idea del tamaño:



Llama a este disfraz "hitbox". ¿Cómo vamos a usarlo? Siempre que el código del juego vaya a comprobar si el objeto "gato" está tocando el objeto "suelo", cambiaremos el disfraz del gato al rectángulo negro antes de hacer la comprobación, y tras hacerla, volveremos a cambiar al disfraz habitual. De esta forma, será el hitbox quien determine si el gato está tocando el suelo.

Estos cambios de disfraz los gestionaremos mediante un procedimiento donde la opción "correr instantáneamente" esté habilitada, para evitar mostrar el disfraz del rectángulo por pantalla.

10) Añade el código del hitbox.

Modifica el procedimiento "gestionarSuelo" como se indica:

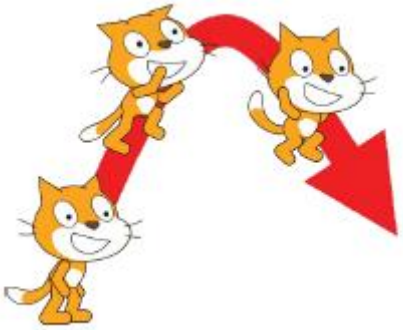
Procedimiento "gestionarSuelo": Ahora, lo primero que debe hacer el procedimiento es cambiar al disfraz "hitbox", y lo último es cambiar al disfraz habitual (disfraz1).

Como el disfraz "hitbox" es un simple rectángulo sin partes salientes que puedan "atascarse" en las plataformas, el juego se comportará ahora de una forma más natural.

F. Anima el proceso de andar.

Como de costumbre, podemos conseguir que el gato ande de forma animada cambiando entre sus dos disfraces por defecto (disfraz1 y disfraz2). Sin embargo, un usuario de Scratch que se hace llamar "griffpatch" (https://scratch.mit.edu/users/griffpatch/) ha creado una serie de disfraces para andar que permiten una animación más interesante:

Este usuario también ha creado disfraces para permanecer de pie, saltar, y caer:



11) Añade los nuevos disfraces al objeto "gato".

Los disfraces para el gato están disponibles en el archivo RAR llamado **Proyecto 19.rar**. Descomprime este archivo en una carpeta. Una vez descomprimidos, los archivos que necesitamos son de **walk1.png** a **walk8.png** para los disfraces de andar, y **stand.svg**, **jump.svg**, y **fall.svg** para los disfraces de permanecer de pie, saltar, y caer.

Una vez descomprimidos los archivos, vuelve a Scratch, selecciona el objeto "gato" y clica en la pestaña "disfraces". A continuación, elige la opción "importar disfraz desde archivo" y ve importando uno a uno todos los disfraces. Borra los disfraces originales "disfraz1" y "disfraz2" porque ya no los usaremos. (Por supuesto, no borres el disfraz "hitbox"). Finalmente, pon los disfraces exactamente en este orden (esto es importante):

1. Stand.
2. Jump.
3. Fall.
4. Walk1.
5. Walk2.
6. Walk3.
7. Walk4.
8. Walk5.
9. Walk6.
10. Walk7.
11. Walk8.
12. Hitbox.

Cada disfraz no solo viene identificado por un nombre (walk1, fall, o hitbox), sino también por un número. El número que identifica a cada disfraz se basa en el orden en el que se muestran los disfraces en la pestaña "disfraces". Por ejemplo, el disfraz más arriba en la pila se llama "stand", pero también se le identifica como disfraz 1. El siguiente disfraz de la pila se llama "jump", pero también se le identifica como disfraz 2, etc.

12) Crea el procedimiento "mostrarDisfrazCorrecto".

Con todos estos disfraces a nuestra disposición, será un poco difícil determinar qué disfraz debemos mostrar y cuándo hacerlo. Como siempre, aquí usaremos la técnica de cambiar rápidamente entre fotogramas estáticos para conseguir una imagen animada, igual que ocurre en un libro animado (folioscopio).

Para seguirles la pista a los disfraces, crea dos variables locales llamadas "fotograma" y "fotogramasPorDisfraz". A continuación, modifica el programa 1 del gato para ajustar los valores iniciales de estados dos nuevas variables a 0 y 4, respectivamente.

Ahora, cuando el jugador mueve al gato hacia la derecha o hacia la izquierda, queremos que la variable "fotograma" se incremente. Por su parte, la variable "fotogramasPorDisfraz" controlará lo rápido o lento que se reproduce la animación.

Vamos a modificar el código del procedimiento "caminar ( )" para incrementar la variable "fotograma" en una cantidad que calcularemos a partir del valor de "fotogramasPorDisfraz":

Procedimiento "caminar ( )": Como último comando de este procedimiento, añade un bloque que cambie el valor de la variable "fotograma" en una cantidad igual a $1/fotogramasPorDisfraz$.

Ahora, cuando el gato permanezca de pie inmóvil (esto es, sin moverse hacia la derecha o hacia la izquierda), la variable "fotograma" debe reinicializarse a 0. Modifica el Programa 2 del gato para realizar esta tarea:

Programa 2 (gato): Añade un nuevo bloque "si" que compruebe si la tecla a no se ha presionado y si la tecla d tampoco se ha presionado. En ese caso, fijamos el valor de la variable "fotograma" a 0.

Ahora vamos a escribir el código para determinar qué disfraz hay que mostrar. Usaremos este código en varios lugares en los programas que ya hemos escrito, por lo que lo más conveniente es usar un procedimiento.

Crea un nuevo procedimiento llamado "mostrarDisfrazCorrecto". Asegúrate de activar la opción "correr instantáneamente". Agrega al procedimiento el siguiente código:

Procedimiento "mostrarDisfrazCorrecto": El procedimiento comienza con un bloque "si / si no", donde comprueba si la variable "enElAire" es menor que 3:

- En caso de que la condición evalúe a VERDADERO (esto es, dentro del grupo SI), comprobamos (mediante otro bloque "si / si no" anidado) si el valor actual de la variable "fotograma" es igual a 0. En ese caso, cambiamos al disfraz "Stand" (el disfraz número 1). En caso contrario, cambiamos a un disfraz cuyo número calcularemos como:

- En caso de que la condición evalúe a FALSO (esto es, dentro del grupo SI NO), comprobamos (mediante un bloque "si / si no" interno) si el valor actual de la variable "velocidad y" es mayor que 0. En ese caso, cambiamos al disfraz "jump". En caso contrario, cambiamos al disfraz "fall".

Con esto termina el bloque "si / si no" externo, y el procedimiento.

¿Cómo funciona este procedimiento? Si el objeto "gato" está en el suelo (o acaba de empezar a saltar, o está cayendo cerca del suelo, de forma que "enElAire" < 3) queremos mostrar el disfraz "stand" (de pie), o uno de los disfraces "walk" (andar). Recuerda que al pinchar en la bandera verde, el programa 2 del gato vuelve a fijar el valor de "fotograma" a 0 si el jugador no presiona las teclas a o d. Así pues, cuando "fotograma" es igual a 0, el procedimiento muestra el disfraz "stand". En caso contrario, debemos calcular cuál de los 8 disfraces "walk" debemos mostrar. Este cálculo lo realiza el bloque "(piso) de $((4) + (fotograma)$ mod $(8))$", usando los números identificativos de los disfraces basados en su orden en la pestaña de "disfraces". Veamos cómo funciona este complicado bloque matemático:

La variable "fotograma" se va incrementando en el procedimiento "caminar ( )". Cuando "fotograma" toma un valor de 0 a 7, queremos que muestre los disfraces con números del 4 al 11. Ésta es la razón por la que hacemos la suma $(4) + (fotograma)$. Pero cuando "fotograma" llegue a 8, queremos volver al disfraz 4, no pasar al disfraz 12. El bloque "( ) mod ( )" nos permite efectuar este retorno: Como (8) mod (8) $= 0$, cuando

el valor de "fotograma" se hace igual a 8, la operación $(fotograma)$ mod (8) nos permite volver a mostrar el primer disfraz. Pero tenemos que sumar 4, porque el primer disfraz de andar es el número cuatro. Por último, el bloque "(piso) de ( )" nos permite redondear hacia abajo. En ocasiones, el valor de "fotograma" se fijará a $4{,}25$ ó $4{,}5$, por lo que $4 + fotograma$ dará $8{,}25$ ó $8{,}5$, y lo que queremos es redondear hacia abajo estas cifras para mostrar el disfraz 8.

Por último, el código del grupo SI NO del condicional externo gestiona lo que ocurre si la variable "enElAire" resulta ser mayor o igual que 3. En ese caso, comprobamos el valor de "velocidad y" para ver si el gato está cayendo ("velocidad y" < 0) o saltando ("velocidad y" > 0), y ponemos el disfraz adecuado. Y con esto, termina el procedimiento.

Ahora que hemos definido el procedimiento "mostrarDisfrazCorrecto", debemos cambiar los bloques "cambiar disfraz a (disfraz1)" en los procedimientos "gestionarSuelo ( )" y "caminar ( )" por las correspondientes llamadas al procedimiento "mostrarDisfrazCorrecto".

G. Crea el nivel.

Ahora vamos a cambiar el sencillo fondo blanco y negro de nuestro escenario por un fondo que muestre un nivel real. Lo bueno del código que hemos escrito hasta ahora es que funcionará para objetos "suelo" de cualquier forma o color. Por consiguiente, si cambiamos el fondo del escenario (por ejemplo, para jugar en distintos niveles) no tendremos que reprogramar el objeto "gato".

13) Descarga y agrega el fondo para el escenario.

Selecciona el objeto "suelo" y clica en la pestaña "disfraces". En "disfraz nuevo", clica en la opción "cargar disfraz desde archivo" y selecciona el archivo **platformerBackdrop.png** de la carpeta donde descomprimiste el archivo RAR. Después de cargar este archivo, puedes borrar el disfraz previo.

Sin embargo, no es suficiente con añadir el archivo platformerBackdrop.png como disfraz para el objeto "suelo", también debemos cargarlo como fondo del escenario. Selecciona la miniatura del fondo en la lista de objetos, clica en la pestaña de "fondos", y pincha en la opción "cargar fondo desde archivo". Igual que antes, selecciona el archivo platformerBackdrop.png. Necesitamos cargar este archivo en ambos lados porque, en el siguiente paso, borraremos del objeto "suelo" todas las partes que no sean suelos o plataformas. Queremos que el objeto "suelo" solo conserve aquellas zonas sobre las que el gato puede andar.

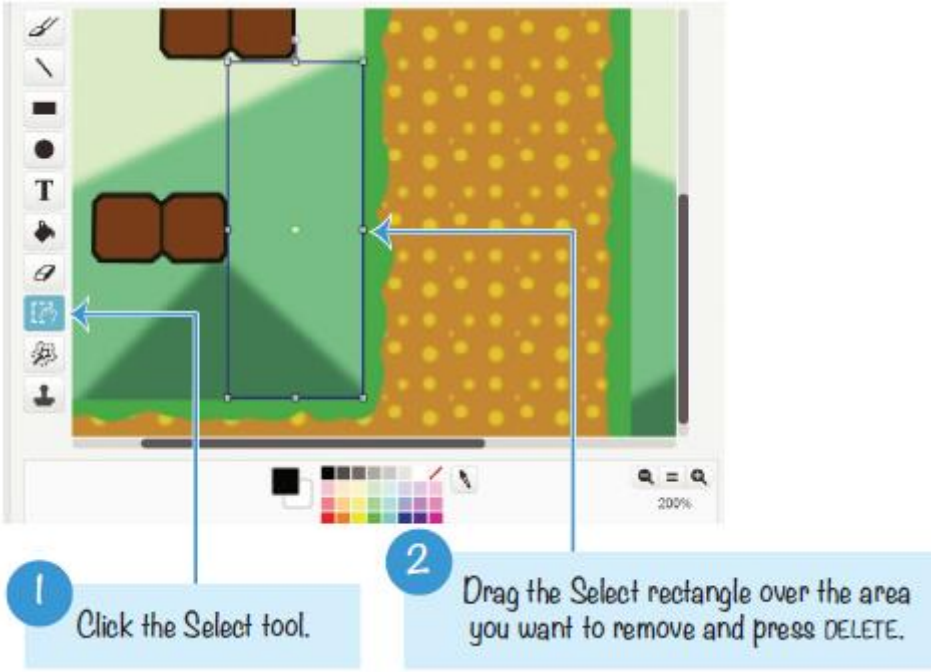
Por cierto, que para este nuevo escenario debemos reducir el tamaño del gato para ajustarlo al tamaño del fondo. Para ello, añade al programa 1 del gato una instrucción que fije el tamaño del gato a un 30% de su tamaño original.

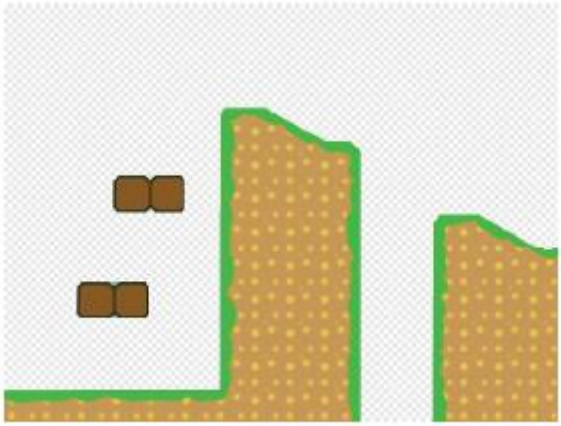
14) Crea un disfraz "hitbox" para el objeto "suelo".

Recordemos que el código de nuestro juego de plataformas se fundamenta en que el objeto "gato" toque al objeto "suelo". El disfraz del objeto "suelo" es un hitbox, y si este disfraz es un rectángulo que ocupe todo el escenario, nuestro código pensará que todo el escenario es un suelo sólido. Por lo tanto, debemos eliminar las partes del objeto "suelo" que no son plataformas.

La forma más sencilla de hacer esto es acudir al editor gráfico de Scratch para modificar el disfraz del objeto "suelo". Pincha en la herramienta "seleccionar", y arrastra hasta seleccionar un rectángulo en la parte del disfraz que quieras borrar. Después de seleccionar un área, presiona en la tecla DEL (o SUPR) del teclado para eliminarla (ver figura).

Usa la herramienta "borrar" para borrar áreas que no son rectangulares. Si te equivocas, pincha en el botón "deshacer" en la parte superior del editor gráfico. Continua eliminando todas las partes innecesarias del fondo hasta que solo queden las plataformas y los suelos (ver figura).





15) Añade el código del objeto "suelo".

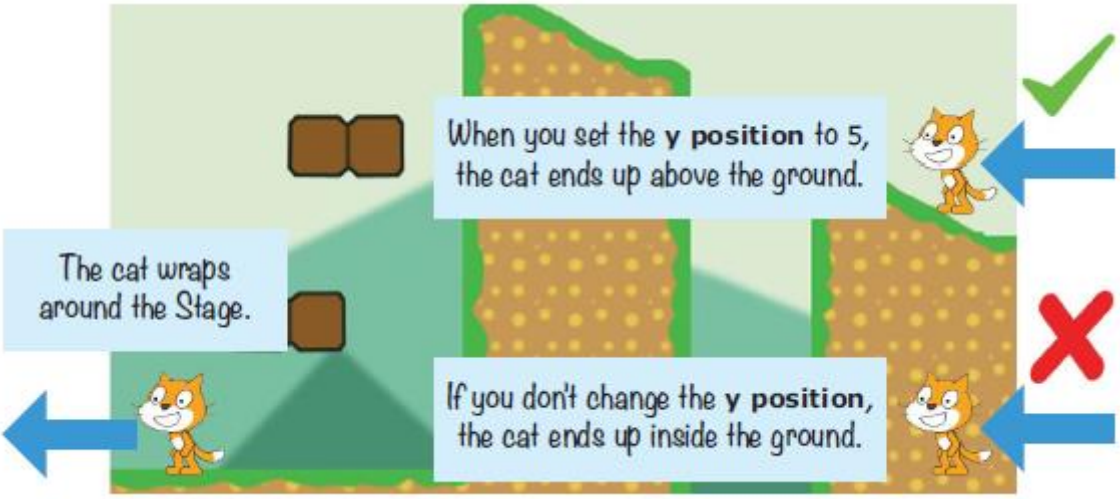
El fondo del escenario se usa para mostrar la apariencia de las plataformas y del fondo. Por su parte, el objeto "suelo" se usa para identificar qué partes del escenario son plataformas sobre las que el gato puede caminar. Añade el siguiente código al objeto "suelo":

Programa 1 (suelo): Al clicar en la bandera verde, este programa simplemente centra el objeto en el centro del escenario, y fija el efecto desvanecer a 100.

Con este programa, alineamos perfectamente el disfraz del objeto "suelo" con el fondo del escenario. (Como el disfraz y el fondo provienen del mismo archivo, podemos alinearlos ajustando el centro del objeto con el centro del fondo, que está en la posición $(0, 0)$). Además, como el *dibujo* del disfraz no tiene relevancia (solo importa la *forma* del disfraz), el programa hace el dibujo del disfraz totalmente transparente. Como el disfraz está alineado con el fondo, seguiremos viendo las plataformas del fondo. De esta forma, el fondo del escenario mostrará la apariencia del nivel, mientras que el disfraz del objeto "suelo" simplemente actuará como un hitbox.

16) Añade el resto del código que le permite al gato pasar de un extremo a otro.

Observa que nuestro nuevo nivel tiene un par de plataformas flotantes y una colina con un foso en el centro. El programa 3 ya consigue que que el gato aparezca en la parte alta del escenario si cae hacia la parte baja a través del foso. Ahora también hay que añadir el código para pasar de un extremo lateral al otro.



Pero cuidado: ten en cuenta que si el gato camina hacia el extremo izquierdo del escenario, su coordenada x es menor que -230. En ese caso, debemos trasladarlo al extremo derecho fijando el valor de su coordenada x a 230. Pero además, cuendo el gato se mueve hacia el extremo izquierdo, su coordenada y es menor que 5. Si al atravesar el extremo izquierdo simplemente lo trasladamos al extremo derecho, el gato terminará dentro del suelo (ver figura). Para evitarlo, antes de trasladarlo debemos comprobar si el valor de su coordenada y es menor que 5, y en ese caso, además de fijar el valor de su la coordenada x a 230, también debemos fijar el valor de su coordenada y a 5 para ponerlo sobre el suelo del extremo derecho.

Programa 5 (gato): Este programa arranca al clicar en la bandera verde. Si el gato se mueve hacia el extremo derecho ($x > 230$), debemos trasladarlo al extremo izquierdo ($x = -230$). Y si el gato se mueve hacia el extremo izquierdo ($x < -230$), debemos trasladando al extremo derecho ($x = 230$, teniendo en cuenta las considereacines respecto a la coordenada y que ya hemos indicado en el párrafo previo)

H. Añade a los cangrejos enemigos y a las manzanas.

El juego de plataformas ya está casi completo. El usuario puede hacer que le gato camine, salte, caiga, y permanezca de pie sobre las plataformas. Solo nos queda fijar el objetivo de la partida: Para ello añadiremos una manzana que aparezca en una ubicación aleatoria en el escenario, y también unos cuentos enemigos que intenten tocar al gato y robarle las manzanas que ha recolectado.

17) Añade el objeto "manzana" y su código.

Agrega el objeto "Apple" de la biblioteca de Scratch, y cambia su nombre a "manzana".

Como en otros proyectos previos, usaremos una variable para llevar la cuenta del número de manzanas que recoge el gato. Define una variable global llamada "manzanasRecogidas".

A continuación, agrega este código al objeto "manzana":

Programa 1 (manzana): Al clicar la bandera verde, el programa empieza finado el valor de la variable "manzanasRecogidas" a 0, y ajustando el tamaño del objeto al 50% de su tamaño original. A continuación arranca un bucle infinito. A cada pasada del bucle, el programa fija el efecto desvanecer a 100 (para hacer

la manzana invisible), y lleva a la manzana a una posición aleatoria (con x entre -240 y 240, e y entre -180 y 80). Por supuesto, la posición aleatoria en la que aparece la manzana no debe estar contenida dentro del suelo, por lo que ahora arranca un bucle condicionado interno que continúa reubicando a la manzana en una posición aleatoria hasta que la manzana no esté tocando al objeto "suelo". Tras ubicar a la manzana en una posición aleatoria fuera del suelo, ajustamos el efecto desvanecer a 0 (para mostrar la manzana), y la manzana espera hasta ser tocada por el gato. Cuando el gato la toca, cambiamos el valor de la variable "manzanasRecogidas" en 1 unidad, y terminamos el bucle infinito.

18) Crea el objeto "cangrejo".

Vamos a añadir a los enemigos cangrejos, cuya función es intentar tocar al gato para robarle una a una las manzanas que ha recogido.

Selecciona el objeto "gato" de la lista de objetos, haz clic con el botón derecho del ratón, y elige la opción "duplicar". Los enemigos usarán el mismo código que el gato para poder saltar y caer sobre las plataformas. (Lo que sí haremos después será quitar el código que permite controlarlos mediante el teclado, y lo reemplazaremos por un código que los mueve aleatoriamente). Renombra a este objeto duplicado como "cangrejo". Después, en la pestaña de disfraces, pincha en el botón "selecciona un disfraz de la biblioteca", y selecciona los disfraces "crab-a" y "crab-b". Borra todos los disfraces del gato, incluyendo el "hitbox".

Por último, crea un nuevo disfraz "hitbox" de un tamaño correcto para el cangrejo. Observa la figura para ver cómo debería quedar:

19) Crea la inteligencia artificial del enemigo.

En los videojuegos, la **inteligencia artificial** (AI) se refiere al código que controla el movimiento de los enemigos y la forma en la que reaccionan ante el jugador. En nuestro juego de plataformas, la inteligencia artificial de los cangrejos es muy poco inteligente: los cangrejos simplemente se moverán de forma aleatoria.

Selecciona el objeto "Cangrejo" y crea una variable local llamada "movimiento". Esta variable almacenará un número que representará los movimientos del cangrejo:

1. Caminar hacia la izquierda.
2. Caminar hacia la derecha.
3. Saltar recto hacia arriba.
4. Saltar hacia la izquierda.
5. Saltar hacia la derecha.
6. Permanecer inmóvil.

Los movimientos del objeto "cangrejo" se decidirán aleatoriamente (usando estos números) y cambiarán de forma frecuente. Añade el siguiente código al objeto "cangrejo":

Programa 1 (cangrejo): Al pinchar en la bandera verde, arranca un bucle infinito. En cada iteración, ajustamos el valor de la variable "movimiento" a un número aleatorio entre 1 y 6. Ahora, comprobamos el valor que ha tomado esta variable:

- Si "movimiento" es igual a 1, volvemos a fijar el valor de la variable movimiento a "izquierda".
- Si "movimiento" es igual a 2, volvemos a fijar el valor de la variable movimiento a "derecha".
- Si "movimiento" es igual a 3, volvemos a fijar el valor de la variable movimiento a "saltar".
- Si "movimiento" es igual a 4, volvemos a fijar el valor de la variable movimiento a "saltar-izquierda".
- Si "movimiento" es igual a 5, volvemos a fijar el valor de la variable movimiento a "saltar-derecha".
- Si "movimiento" es igual a 6, volvemos a fijar el valor de la variable movimiento a "inmóvil".

A continuación, busca todo el código del antiguo objeto "gato" que use los bloques "¿tecla ( ) presionada?" y reemplázalos por bloques que comprueben el valor de la variable "movimiento". Así, modifica el programa que comprueba si el usuario está presionando las teclas A y D para ajustarse a lo siguiente:

Programa 2 (cangrejo): Al pinchar en la bandera verde, arranca un bucle infinito. A cada pasada del bucle, hacemos varias comprobaciones:

- Si la variable "movimiento" es igual a "izquierda" o es igual a "saltar-izquierda", el programa hace que el cangrejo apunte hacia la izquierda, y llama al procedimiento "caminar ( )" pasándole 4 pasos como parámetro.
- Si la variable "movimiento" es igual a "derecha" o es igual a "saltar-derecha", el programa actúa de forma análoga.
- Si la variable "movimiento" es igual a "inmóvil", ajustamos la variable "fotograma" a 0.

Con esto termina el bucle y el programa.

Ahora cambia el programa que comprueba si el usuario está presionando la tecla W para ajustarse a lo siguiente:

Programa 3 (cangrejo): De nuevo, al clicar la banderea verde arranca un bucle infinito. A cada pasada del bucle, comprobamos si el valor de la variable "enElAire" es menor que 8. En ese caso, comprobamos si la variable "movimiento" toma los valores "saltar", "saltar-izquierda", o "salltar-derecha", y en ese caso, ajustamos el valor de "velocidad y" a 14.

Ahora vamos a animar los movimientos del cangrejo. El cangrejo solo dispone de dos disfraces, "crab-a" y "crab-b". Vamos a cambiar entre esos dos disfraces para animar el movimiento de andar. Para ello, modificaremos el procedimiento "mostrarDisfrazCorrecto" del cangrejo:

Procedimiento "mostrarDisfrazCorrecto": si el valor de la variable "fotograma" es igual a 0, cambiamos al disfraz "carb-a". En caso contrario, cambiamos al disfraz cuyo número identificativo viene dado por:

Notar que los números del bloque "(piso) de ((1) + ($fotograma$) mod (2))" han cambiado respecto a los números de bloque para el gato. El primer disfraz es el disfraz 1, y el cangrejo solo tiene dos disfraces, por lo que esos números se han cambiado a 1 y 2, respectivamente.

Finalmente, debemos crear un nuevo programa para que los cangrejos puedan robar las manzanas al jugador:

Programa 4 (cangrejo): Al pinchar en la bandera verde, arranca un bucle infinito donde un condicional externo comprueba si el cangrejo está tocando al gato. En ese caso, un condicional interno comprueba si el valor de "manzanasRecogidas" es mayor que 0. En caso afirmativo, el cangrejo cambia el valor de "manzanasRecogidas" en -1, y dice "tengo una!" durante 2 segundos. En caso negativo, dice "quiero manzanas!" durante 2 segundos.

Con esto hemos terminado con el cangrejo. Pero El juego es más desafiante si añadimos un segundo cangrejo, así que clica con el botón derecho del ratón sobre la miniatura del objeto "cangrejo" en la lista de objetos, y elige la opción "duplicar".

20) Añade el objeto "Tiempo finalizado".

Ya casi hemos acabado. Solo nos queda añadir un temporizador, que obligue al jugador a coger tantas manzanas como pueda durante un tiempo limitado.

En la lista de objetos, pincha en el botón "dibujar nuevo objeto", y dibuja en el editor gráfico de Scratch el texto "tiempo finalizado":

Renombra al objeto como "tiempo finalizado". Crea una variable global llamada "temporizador", y añade al objeto el siguiente código:

Programa 1 (tiempo finalizado): Al clicar en la bandera verde, el programa esconde al objeto, fija el valor de "temporizador" a 45, y arranca un bucle que se repetirá hasta que "temporizador" se haga 0. A cada pasada del bucle, esperamos 1 segundo, y cambiamos "temporizador" por -1. Al terminar el bucle (esto es, al agotarse la cuenta atrás), mostramos al objeto, y paramos la aplicación.

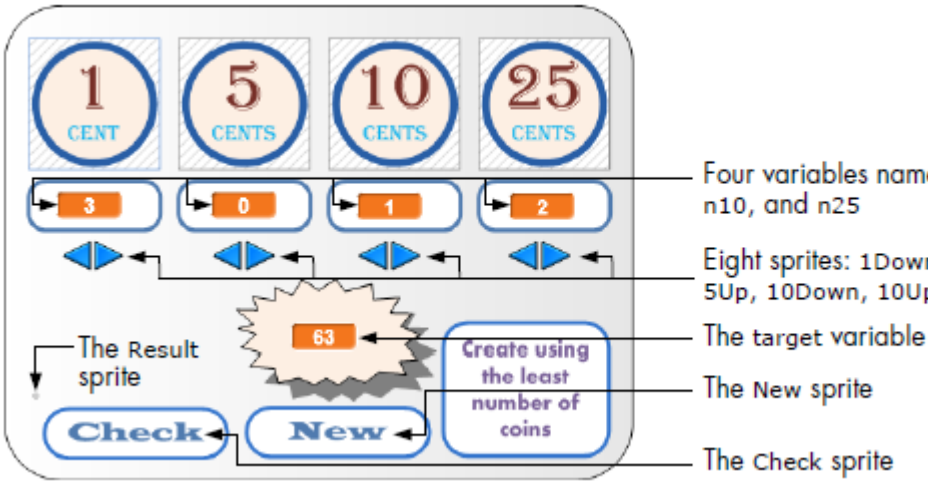
Este código le da al jugador 45 segundos para recoger tantas manzanas como pueda, mientras que evita que los cangrejos se las roben. Al agotarse el tiempo, se termina la partida.

20) ¡¡Ya hemos terminado!! Guarda el juego como **Proyecto 19.sb2** y juega unas cuentas partidas; te lo has ganado.

7.7. PROYECTOS LIBRES.

PROYECTO LIBRE 3. MONEDAS.

Este proyecto consiste en construir un juego educativo para evaluar las habilidades para contar de los estudiantes de los primeros ciclos de primaria. Esta aplicación muestra una cantidad de dinero en céntimos, y el jugador debe hallar la mínima cantidad de monedas necesarias para conseguir esa cantidad. La interfaz de usuario y los objetos necesarios están disponibles en el archivo **Libre3_sinCodigo.sb2**.



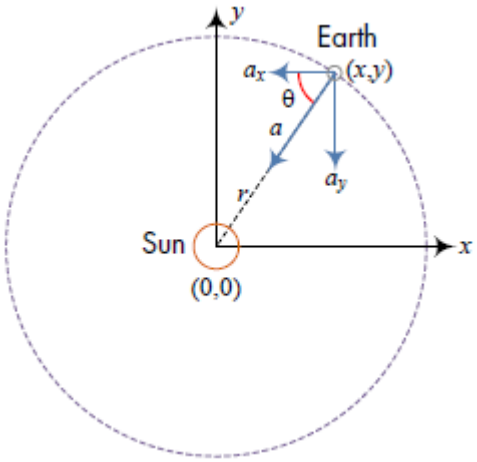
La aplicación comienza cuando el usuario clica en el botón "new", el cual extrae una cantidad aleatoria entre 1 y 250 (que almacena en la variable "target"), y envía el mensaje "nuevo problema", que captura el objeto "result" para limpiar el bocadillo de diálogo que pudiese haber de una partida previa. A continuación, con los botones de las flechas bajo cada tipo de moneda, el usuario introduce cuántas monedas de cada tipo son necesarias (como mínimo) para sumar la cantidad aleatoria. (Esos números quedan almacenados en las variables "n1", "n5", "n10", y "n25"). Después de indicar el número de monedas, el usuario clica en el botón "check", que envía el mensaje "comprobar respuesta" para chequear si la respuesta es correcta.

El mensaje "comprobar respuesta" también lo recibe el objeto "result", que pasa a comparar el número de monedas de cada tipo especificado el usuario (datos almacenados en las variables "n1", "n5", "n10", y "n25"), con el número mínimo correcto de monedas para sumar la cantidad en cuestión (datos almacenados en las variables "c1", "c5", "c10", y "c25"), ver siguiente párrafo. En base a esta comparación, el objeto muestra por pantalla (mediante un bocadillo de diálogo) si la respuesta es correcta, si es correcta pero no con el mínimo número de monedas, o si es incorrecta.

Para computar el mínimo número de monedas necesarias, el objeto "result" acude al procedimiento "calculate", que por su complejidad, mostramos en la figura:

Con esto ya has terminado. Guarda el archivo como **Libre 3.sb2**.

PROYECTO LIBRE 4. MOVIMIENTO PLANETARIO.



Este proyecto libre te propone construir una simulación del movimiento planetario de un sencillo sistema solar que solo contenga un planeta en órbita alrededor del sol. Según la ley de gravitación universal de Newton, la magnitud de la fuerza gravitacional entre el sol y el planeta viene dada por:

$$F = G\frac{Mm}{r^2}$$

, donde M y m son las masas del sol y del planeta, r la distancia entre ellos, y G la constante de gravitación.

Usando la segunda ley de Newton, la aceleración que experimenta el planeta hacia el sol resulta ser:

$$a = \frac{F}{m} = G\frac{M}{r^2}$$

Si el sol está localizado en el origen, $(0,0)$, y la posición instantánea del planeta es (x,y), las componentes x e y de la aceleración son:

$$a_x = a\cos\theta = -G\frac{Mx}{r^3}$$

$$a_y = a\sin\theta = -G\frac{My}{r^3}$$

(El signo menos indica que la aceleración está dirigida hacia el sol). Como la aceleración es el cambio de la velocidad por unidad de tiempo, durante cada intervalo temporal, la velocidad horizontal de planeta, v_x, cambia en una cantidad a_x, y la velocidad vertical, v_y, cambia en una cantidad a_y. Además, y como la velocidad es el cambio en la posición por unidad de tiempo, durante cada intervalo temporal la posición horizontal del planeta, x, cambia en una cantidad v_x, y la posición vertical del planeta, y, cambia en una cantidad v_y. Esto significa que, para cada intervalo de tiempo de nuestra simulación, debemos calcular la distancia del planeta al sol, $r = \sqrt{x^2+y^2}$, cambiar v_x y v_y (mediante a_x y a_y, respectivamente), y calcular la nueva posición del planeta x e y (con los valores de v_x y v_y).

Lo último que debemos hacer es considerar qué unidades se corresponden con la escala de nuestro problema. Midiendo la distancia en unidades astronómicas (donde $1\ AU \approx 1{,}5\times10^{11}\ m$) y el tiempo en años (donde $1\ año \approx 3{,}156\times10^{7}\ s$), tenemos:

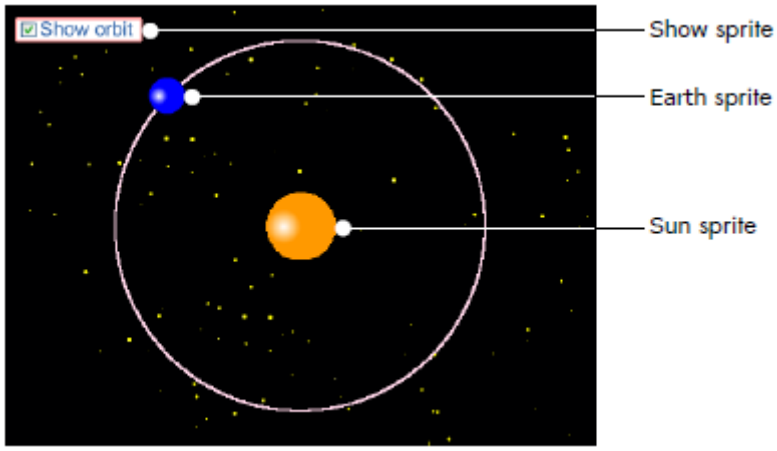
$$M\times G \approx (1{,}99\times10^{30}\ kg)\times\left(6{,}673\times10^{-11}\ \frac{m^3}{kg\cdot s^2}\right) \approx 39{,}2\ AU^3/yr^2$$

Con estos conceptos teóricos, ya disponemos de la información suficiente para construir la simulación que traza la órbita del planeta alrededor del sol. La interfaz de usuario y los objetos necesarios están disponibles en el archivo **Libre4_sinCodigo.sb2**.

La simulación comienza al clicar en la bandera verde. El planeta empieza en la posición $(150,0)$ y arranca con una pequeña velocidad inicial positiva en la dirección y (la velocidad inicial en la dirección x es cero). Con estos valores, podemos poner en marcha el procedimiento anteriormente indicado para calcular las posiciones subsiguientes del planeta, y dibujar su órbita.
El objeto "show" (mostrar) tiene dos disfraces, uno con la casilla clicada y otro con la casilla sin clicar. Al clicar en este botón, cambiamos entre los disfraces y enviamos los mensajes "mostrarOrbita" o

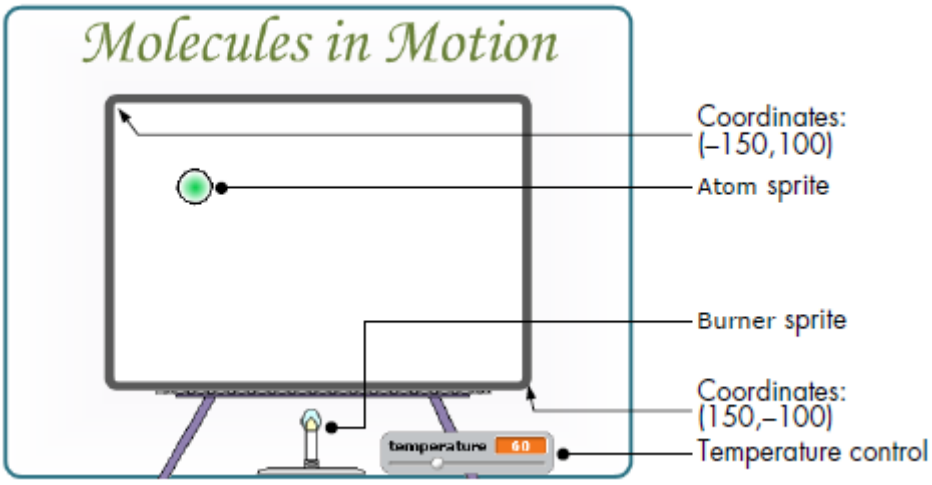
"esconderOrbita", según el caso. Cuando el planeta recibe el mensaje "mostrarOrbita", se pone a dibujar la órbita que recorre, y si recibe "esconderOrbita" deja de dibujar la órbita y limpia la pantalla.

Al terminar el proyecto, guarda el archivo como **Libre 4.sb2**.

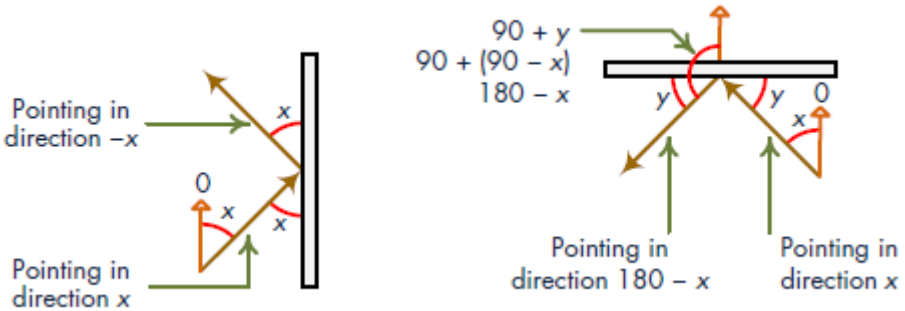


PROYECTO LIBRE 5: MOLÉCULAS EN MOVIMIENTO.

Según la teoría cinética, las moléculas de un gas se están moviendo constantemente de forma aleatoria. Si el gas está dentro de un recipiente, las partículas colisionan continuamente unas con otras, y con las paredes del recipiente. La rapidez del movimiento es directamente proporcional a la temperatura del gas. En este proyecto vamos a simular el movimiento de una sola molécula de gas dentro de un recipiente. La interfaz de usuario está disponible en el archivo **Libre 5_sinCodigo.sb2**.



Cuando el objeto "átomo" colisiona contra una pared del recipiente, debería rebotar con un ángulo igual a su *ángulo de incidencia* (el ángulo con el que impacta contra la pared), como muestra la figura:



La figura de la izquierda muestra que si un átomo moviéndose en una cierta dirección x (en este ejemplo, la dirección $x = 45°$) golpea la pared derecha (o izquierda), rebotará en la dirección $-x$ (en este ejemplo, en la dirección $-45°$). De forma similar, la figura de la derecha muestra que si un átomo apuntando en una cierta dirección x impacta contra la pared superior (o inferior), rebotará en la dirección $180° - x$.

Además del escenario, la aplicación contiene dos objetos, el "átomo" (atom) y el "quemador" (burner). El quemador tiene 4 disfraces, que muestran llamas de diferentes tamaños.

La simulación arranca al clicar en la bandera verde. El programa del quemador simplemente cambia entre los disfraces para animar la llama. Por su parte, el átomo comienza en el origen, y selecciona una dirección aleatoria. A continuación, y de forma indefinida, el átomo calcula el valor de la variable "rapidez" (speed) en función del valor actual de la variable "temperatura" (temperature), el cual se fija con un deslizador. La fórmula a utilizar es:

$$speed = \frac{temperatura}{10}$$

A continuación, el programa comprueba si el átomo ha colisionado contra alguna de las paredes (las ubicaciones de las paredes podemos deducirlas de la primera figura). En caso de colisión, la discusión previa debería aclararnos cómo obtener la dirección de rebote a partir de la dirección de incidencia (esto es, a partir de la dirección actual). Tras rebotar, el objeto se mueve una cierta distancia (especificada por la variable "rapidez") en su nueva dirección.

Cuando termines la simulación, guarda el archivo como **Libre 5.sb2**.

ASPECTOS AVANZADOS: CADENAS Y LISTAS.

8. PROCESADO DE CADENAS.

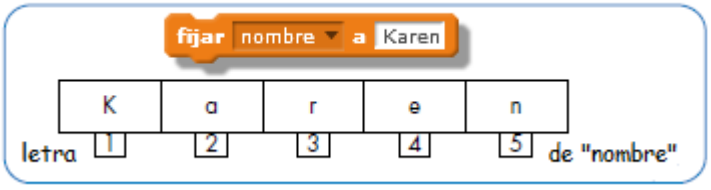
Una cadena es un conjunto de caracteres que Scratch trata como una sola unidad. Con Scratch podemos escribir programas para combinar, comparar, clasificar, encriptar, y para manipular cadenas de muchas otras formas. Ése es precisamente el objetivo de este capítulo.

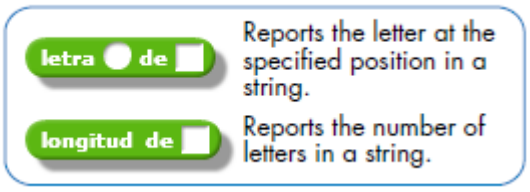
El capítulo empieza con un repaso del tipo de datos cadena. Una vez revisadas las cadenas, escribiremos unos procedimientos para gestionarlas y manipularlas. Estos procedimientos nos permitirán quitar y reemplazar caracteres, insertar y extraer partes de una cadena, y aleatorizar el orden de los caracteres que contiene. A continuación, utilizaremos estos procedimientos y técnicas para desarrollar algunas aplicaciones interesantes.

8.1. UNA REVISIÓN DE LAS CADENAS.

Como ya mencionamos en el capítulo 5, Scratch dispone de tres tipos de datos: booleanos, números, y cadenas. Dicho de forma muy sencilla, una cadena es simplemente una secuencia ordenada de caracteres. Estos caracteres pueden ser letras (minúsculas y mayúsculas), dígitos (del 0 al 9), y otros caracteres especiales (como $+$, $-$, &, @, etc.). Podemos usar cadenas para almacenar nombres, direcciones, números de teléfono, títulos de libros, y cosas por el estilo.

En Scratch, los caracteres de una cadena se almacenan secuencialmente. Por ejemplo, si tenemos una variable llamada "nombre", la ejecución del comando "fijar nombre a (Karen)" hará que los caracteres se almacenen como ilustra la figura:





Podemos acceder a los caracteres individuales de una cadena con el bloque "letra ( ) de ( )" de la categoría "operadores". Por ejemplo, "letra (1) de (nombre)" devuelve el carácter K, y "letra (5) de (nombre)" devuelve el carácter n. Scratch también proporciona el bloque "longitud de ( )", que devuelve el número de caracteres de una cadena. Usando estos dos bloques, junto con los correspondientes bloques "repetir", podemos contar caracteres, examinar múltiples caracteres, y hacer otras muchas cosas útiles, como demostraremos a continuación.

CONTAR CARACTERES ESPECIALES EN UNA CADENA.

El programa de la figura le pide al usuario que introduzca una cadena, y a continuación, cuenta y muestra por pantalla cuántas vocales contiene.

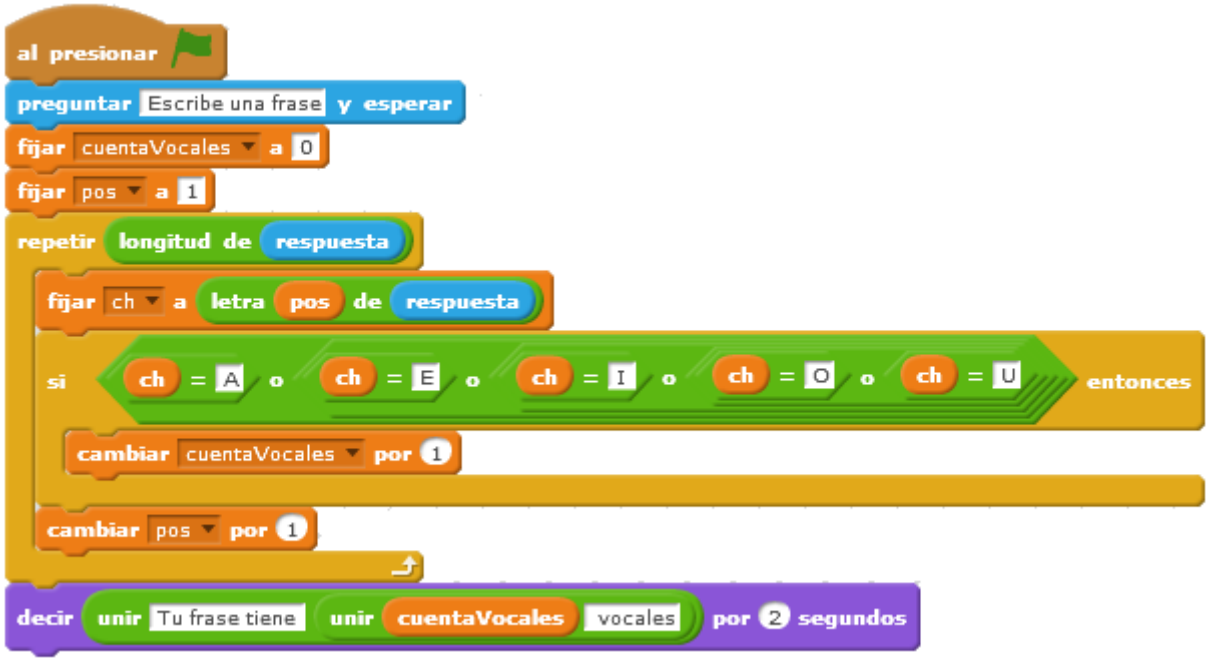
El programa comprueba una a una cada letra de la cadena en busca de vocales. Cada vez que encuentra una vocal, incrementa el valor de la variable "cuentaVocales" en 1 unidad. El programa usa la variable "pos" (de posición) para monitorizar la posición del carácter que está comprobando actualmente.

Al comenzar, el programa le pide al usuario que escriba una frase, y queda a la espera hasta que la introduce. Recordemos que Scratch almacena automáticamente los datos introducidos por el usuario en la variable "respuesta" (categoría "sensores"). A continuación, fija el valor de "cuentaVocales" a 0, y el valor de "pos" a 1 (para acceder al primer carácter de la cadena).

Después, el programa comprueba todos los caracteres de la cadena mediante un bucle "repetir ( )". (El operador "longitud de (respuesta)" reporta el número de caracteres de la cadena introducida por el usuario, y este valor es el número de veces que debe repetirse el bucle). En cada iteración, el programa usa la variable "ch" (de la palabra inglesa *character*) para comprobar cada carácter de la cadena. En la primera iteración, el programa ajusta el valor de "ch" al primer carácter de "respuesta". La segunda iteración ajusta el valor de "ch" al segundo carácter, y así sucesivamente, hasta que el bucle llega al final de la cadena. La variable "pos" nos sirve para acceder al carácter deseado.

Seguidamente, el bloque "si" comprueba si el carácter que estamos examinando es una vocal. Si es una vocal, ya sea mayúscula o minúscula, el valor de "cuentaVocales" se incrementa en 1.

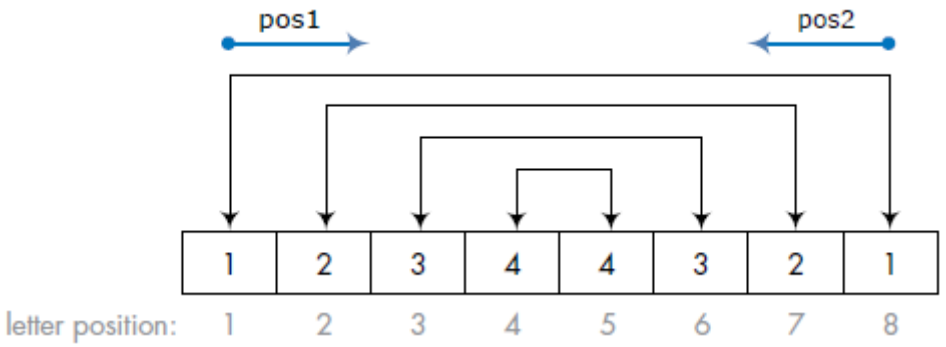
Tras analizar un carácter, el bucle incrementa "pos" en 1 unidad y empieza a leer el siguiente carácter. Cuando se han comprobado todos los caracteres el bucle termina, y el programa muestra por pantalla el número de vocales mediante un bloque "decir ( ) por (2) segundos".



Aplicaremos las técnicas de este ejemplo muchas veces a lo largo del capítulo, así que asegúrate de que lo has entendido perfectamente.

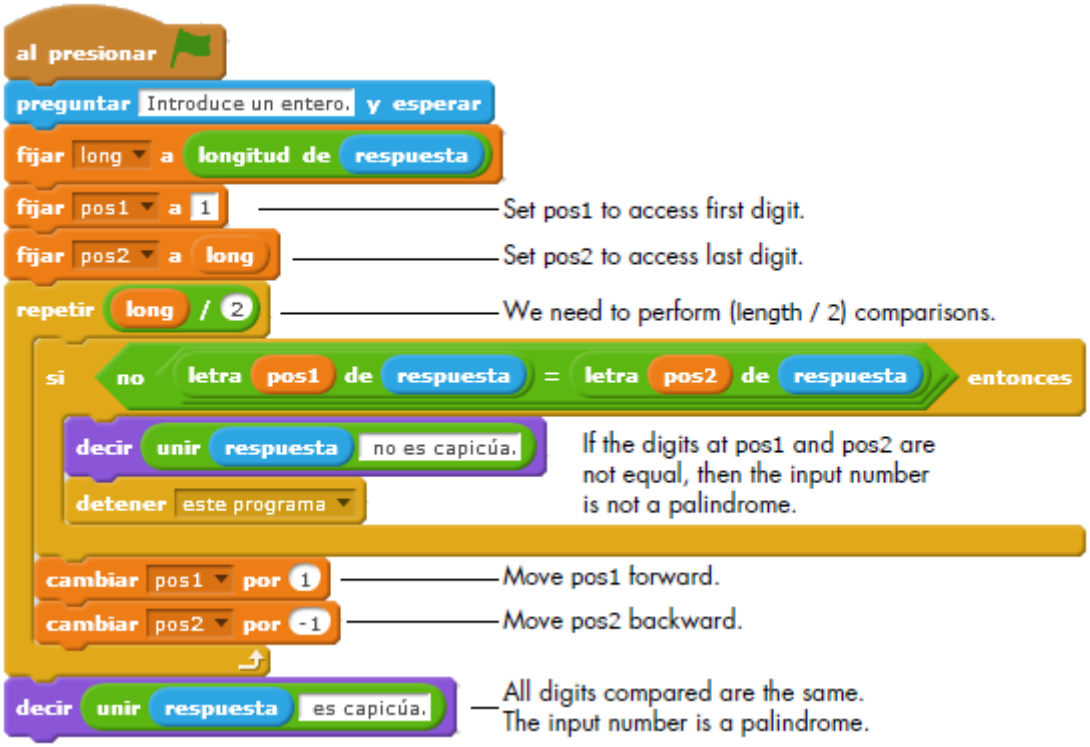
COMPARAR CARACTERES DE CADENAS.

Nuestro segundo ejemplo comprueba si un número entero introducido por el usuario es *capicúa*. Un número capicúa es aquel que se lee igual de izquierdas a derechas que de derechas a izquierdas. Por ejemplo, el número 12721 es un capicúa, pero 34534 no lo es. (Cuando hablamos de textos, el término que se usa es *palíndromo*. Por ejemplo, Hannah es un palíndromo).



Para ilustrar nuestro algoritmo de comprobación de números capicúas, digamos que el usuario introduce el número 12344321 (ver figura). Para ver si un número es capicúa, tenemos que comparar si los dígitos primero y octavo son iguales, si el segundo y el séptimo son iguales, si el tercero y el sexto son iguales, etc. Si alguna de estas comparaciones produce un resultado FALSO (esto es, si los dígitos no son iguales), el número no será capicúa. La figura muestra el código que implementa este método de comprobación.

El programa accede a los dos dígitos que debemos comparar con las variables "pos1" y "pos2", que se mueven en direcciones opuestas. La variable "pos1" comienza en el primer dígito y se mueve hacia adelante, y la variable "pos2" comienza en el último dígito y se mueve hacia atrás. El número de comparaciones a realizar es, como mucho, la mitad del número de dígitos del entero introducido. Si el entero es 12344321, debemos hacer 4 comparaciones. (Esta misma argumentación aplica a un entero con un número impar de dígitos, ya que el dígito central no debe compararse con ninguno). Una vez que el programa decide si el entero es o no capicúa, muestra por pantalla un mensaje con el resultado.



En la siguiente sección exploraremos algunas de las operaciones más comunes que se pueden efectuar con cadenas, y veremos algunas estrategias para escribir procedimientos de manipulación de cadenas en Scratch.

8.2. EJEMPLOS DE MANIPULACIÓN DE CADENAS.

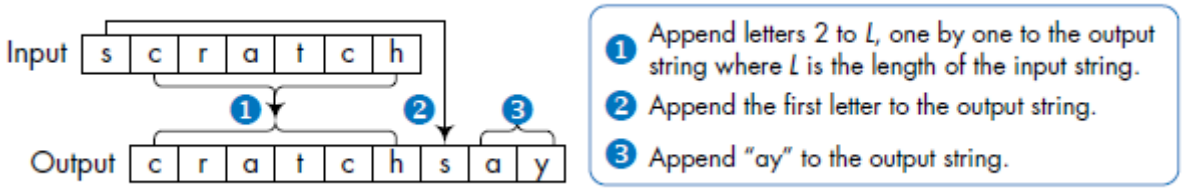
El operador "letra ( ) de ( )" nos permite leer los caracteres individuales de una cadena. Pero si queremos insertar (o borrar) caracteres de una cadena, tenemos que programar los métodos que nos permitan hacerlo.

En Scratch no podemos alterar los caracteres de una cadena, así que la única forma de cambiar una cadena es crear una cadena nueva. Por ejemplo, si queremos cambiar a mayúsculas la primera letra de la cadena "pablo", debemos crear una nueva cadena que contenga la letra "P" seguida del resto de letras, "ablo". La idea es usar el operador "letra ( ) de ( )" para leer los caracteres de la cadena original, y el operador "( ) unir ( )" para adjuntar esas letras a la nueva cadena.

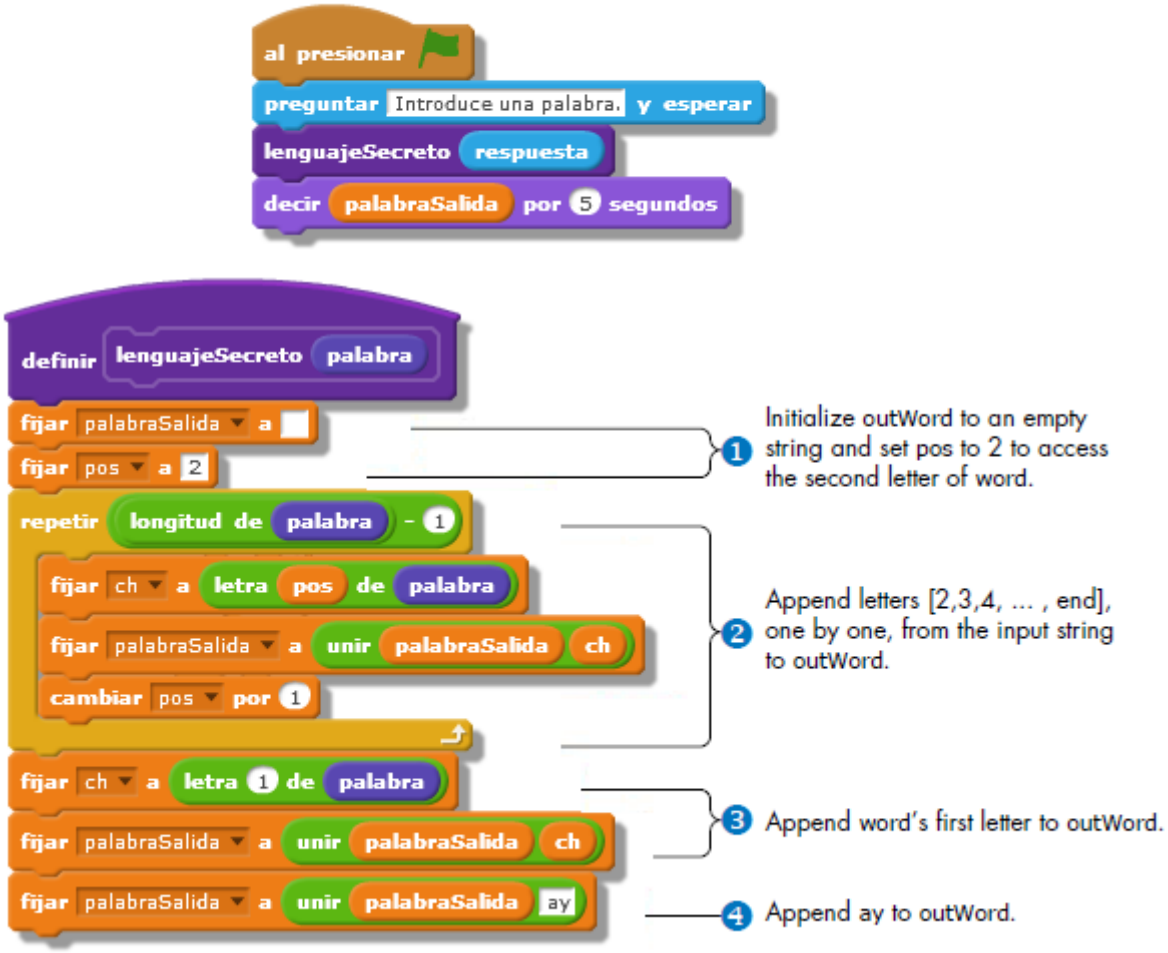
En esta sección desarrollaremos una serie de aplicaciones sencillas para mostrar cómo se usan las técnicas de manipulación de cadenas.

LENGUAJE SECRETO.

¿Y si nuestros objetos pudiesen hablar entre ellos en un lenguaje secreto? Las reglas para crear las palabras de este lenguaje son sencillas: Basta con mover la primera letra al final y añadir las letras "ay". Así, la palabra "seta" se convierte en "etasay", la palabra "pera" se convierte en "erapay", etc. La estrategia que usaremos para convertir una palabra a nuestro lenguaje secreto se ilustra en la figura:



(1) Primero, adjuntamos una a una todas las letras (excepto la primera) desde la palabra de entrada a la palabra de salida. (2) A continuación, añadimos la primera letra de la palabra de entrada a la palabra de salida. (3) Finalmente, añadimos "ay". El procedimiento que implementa estos pasos se muestra en la figura:



El procedimiento usa tres variables: la variable "palabraSalida" aloja la cadena de salida conforme se va construyendo. El contador "pos" (posición) le dice al programa qué carácter de la cadena original hay que agregar a la palabra de salida. Finalmente, la variable "ch" almacena un carácter de la cadena de entrada. El procedimiento recibe como entrada el parámetro "palabra", mediante el cual se le pasa la palabra que queremos traducir a nuestro lenguaje secreto.

En primer lugar, el procedimiento guarda una cadena vacía en la variable "palabraSalida", y ajusta "pos" a 2. (Una cadena vacía es una cadena que no contiene ningún carácter, esto es, una cadena de longitud 0). Después, el procedimiento usa un bloque "repetir ( )" para adjuntar todas las letras, excepto la primera,

desde la cadena de entrada ("palabra") a la cadena de salida ("palabraSalida"). Debemos omitir el primer carácter, por lo que el contador del bucle es una unidad menor que la longitud de la cadena de entrada. A cada iteración del bucle, el procedimiento adjunta un carácter de la cadena "palabra" a la cadena "palabraSalida". Al final del bucle, se adjunta la primera letra de "palabra" a "palabraSalida", junto con las letras "ay".

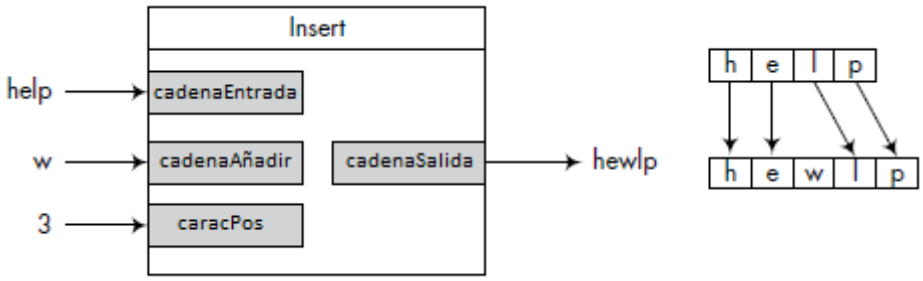
EJERCICIO 40. TRADUCTOR DE FRASES.

Modifica la aplicación previa para que permita traducir toda un frase (por ejemplo, "me llamo Alejandro y soy profesor") a nuestro lenguaje secreto. (PISTA: Llama al procedimiento "lenguajeSecreto ( )" para cada palabra de la frase de entrada). Como ampliación, crea un procedimiento que haga la traducción inversa, esto es, que reciba una palabra escrita en nuestro lenguaje secreto, y la muestre en castellano.

CORRECCIÓN DE PALABRAS DELETREADAS.

En esta sección vamos a desarrollar un juego que genera palabras mal deletreadas y le pide al usuario que las escriba correctamente. El juego generará palabras mal deletreadas insertando una letra aleatoria en una posición aleatoria en una palabra en inglés. (Por supuesto, podría haber más de un deletreo correcto de una palabra incorrecta; por ejemplo, si la palabra original es "wall" y el juego produce "mwall", tanto "mall" como "wall" son palabras correctas. Pero para que nuestro juego sea razonablemente sencillo de programar, vamos a ignorar esa posibilidad).

En primer lugar, vamos a hacer un procedimiento general para insertar caracteres en una posición específica de una cadena. Este procedimiento, llamado "insertar", recibe tres parámetros: la palabra de entrada ("cadenaEntrada"), la cadena (o carácter) a insertar ("cadenaAñadir"), y la posición donde queremos añadir esos nuevos caracteres ("caracPos"). El procedimiento genera una nueva cadena ("cadenaSalida") con "cadenaAñadir" insertada en "cadenaEntrada" en la posición indicada por "caracPos", como ilustra la figura:



El procedimiento funciona como sigue: Añade uno a uno los caracteres de "cadenaEntrada" a "cadenaSalida". Cuando llega a la posición "caracPos", simplemente añade el/los carácter/es de "cadenaAñadir" a "cadenaSalida" antes de adjuntar la letra presente en la posición "caracPos" de "cadenaEntrada". El procedimiento completo se muestra en la figura.

En primer lugar, el procedimiento inicializa "cadenaSalida" a una cadena vacía, y fija la variable "pos" a 1 para acceder a la primera letra de la cadena de entrada. A continuación arranca un bucle "repetir ( )" para adjuntar, una a una, las letras de "cadenaEntrada" a "cadenaSalida": A cada iteración, el procedimiento toma el siguiente carácter de "cadenaEntrada" y lo aloja en la variable "ch". Pero si la posición del carácter actual es igual al valor de "caracPos", el procedimiento adjunta "cadenaAñadir" a "cadenaSalida". En todos los casos, se adjunta "ch" a "cadenaSalida", y "pos" se incrementa en 1 unidad para acceder a la siguiente letra de "cadenaEntrada".

La figura también muestra el programa principal de la aplicación. La cadena "alfa" contiene todas las letras del alfabeto que son necesarias para formar cualquier palabra inglesa. Esta cadena proporcionará una letra aleatoria para insertarla en la palabra que queremos mostrar deletreada incorrectamente. El programa

elige aleatoriamente una palabra de una **lista** (de nombre "wordList"), y la almacena en la variable "palabraEntrada". Aprenderemos mucho más sobre las listas en el siguiente capítulo; por ahora, podemos imaginar esta lista como un banco de palabras. Posteriormente, el programa elige una letra aleatoria ("caracterAleat") de la cadena "alfa", y una posición aleatoria ("posAleat") para ubicar esta letra en "palabraEntrada". A continuación, el programa llama al procedimiento "insertar ( ) dentro de ( ) en la posición ( )" para crear la palabra mal deletreada ("cadenaSalida"). Después de eso, el programa arranca un bucle para obtener la respuesta del usuario. Dentro del bucle, el programa le pide al jugador que introduzca el deletreo correcto, y usa un bloque "si / si no" para comprobar la respuesta. Si la respuesta del jugador se ajusta a la palabra original ("palabraEntrada"), el juego termina; en caso contrario, el jugador puede volver a intentarlo.

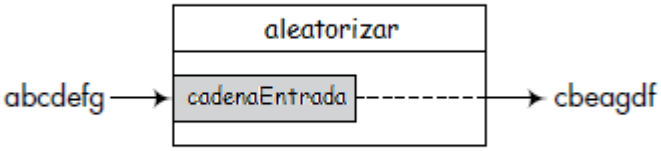
EJERCICIO 41. DOS CARACTERES.

Modifica el juego previo para que la palabra mal deletreada contenga dos letras adicionales en lugar de solo una.

DESCIFRADO.

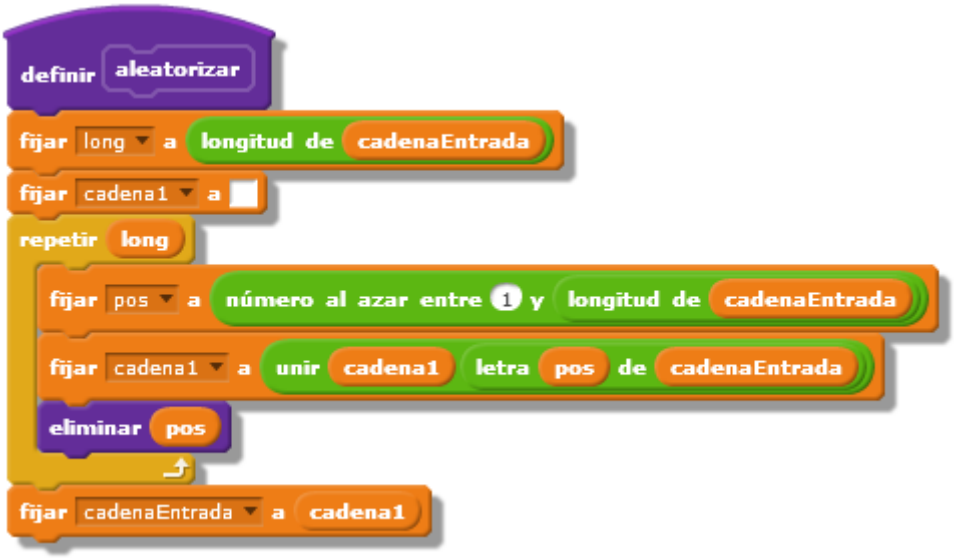
Nuestro último ejemplo comienza con una palabra en inglés, desordena sus letras, y le pide al usuario que adivine cuál es la palabra original.

Comencemos creando un procedimiento que redistribuya los caracteres de una cadena en un orden aleatorio. El programa que llama al procedimiento fija el valor de la cadena de entrada ("cadenaEntrada"), y el procedimiento, llamado "aleatorizar", la modifica desordenando sus caracteres, como ilustra la figura:



Step	Pick random	Append to cadena1	Remove from cadenaEntrada
1	3	c	abdefg
2	2	cb	adefg
3	1	cba	defg
4	2	cbae	dfg
...	...	...	...

El método es el siguiente: el procedimiento toma una letra aleatoria de "cadenaEntrada", y la adjunta a una cadena auxiliar llamada "cadena1". (Se trata de una cadena temporal que comienza vacía, y donde el procedimiento guarda la palabra desordenada conforme la va construyendo). A continuación, elimina esa letra de "cadenaEntrada" para no volver a usarla, y repite todo el proceso hasta que "cadenaEntrada" quede vacía. La implementación de este procedimiento se muestra en la figura:

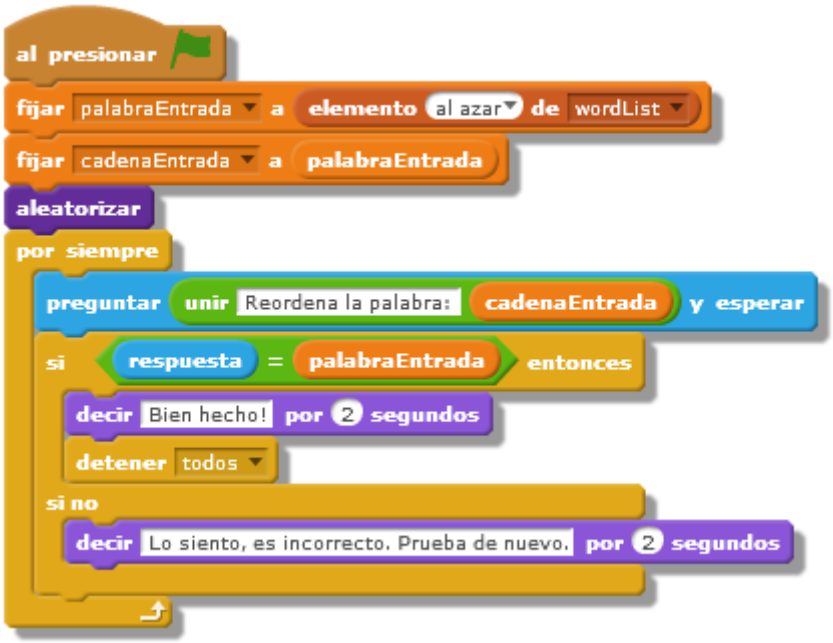


En primer lugar, "aleatorizar" fija el valor de la variable "long" a la longitud de "cadenaEntrada", e inicializa la variable "cadena1" a una cadena vacía. A continuación, el procedimiento arranca un bucle "repetir ( )" para construir la palabra desordenada. El contador del bucle es igual a la longitud de la cadena de entrada. A cada iteración, el bucle elige una posición aleatoria en "cadenaEntrada", y adjunta esa letra a "cadena1". (Notar que usamos el bloque "longitud de (cadenaEntrada)" porque tanto "cadenaEntrada" como su longitud cambiarán conforme le vamos quitando letras para pasárselas a "cadena1"). Después de eso, el procedimiento llama a otro procedimiento, llamado "eliminar ( )", para borrar el carácter de "cadenaEntrada" que acabamos de usar. Cuando el bucle termina de desordenar las letras, el procedimiento fija el valor de "cadenaEntrada" al valor de "cadena1", donde está almacenada la palabra desordenada.

El procedimiento "eliminar ( )" simplemente sirve para eliminar el carácter de "cadenaEntrada" situado en la posición especificada por el parámetro "posCarac". Este procedimiento evita que añadamos dos veces la misma letra a la palabra desordenada.

Este procedimiento usa otra cadena temporal auxiliar, llamada "cadena2", para construir la nueva cadena que queremos crear. El procedimiento comienza vaciando "cadena2", y ajustando el valor del contador de bucle "n" a 1, para acceder al primer carácter de "cadenaEntrada". A continuación, el procedimiento arranca un bucle "repetir ( )" para montar la cadena de salida. A cada pasada del bucle, comenzamos comprobando si el carácter actual es el que queremos borrar o no. Si *no* es el que queremos borrar, lo adjuntamos a "cadena2". A continuación, el bucle incrementa el valor del contador "n" para acceder a la siguiente letra de "cadenaEntrada". Al salir del bucle, el procedimiento termina fijando el valor de "cadenaEntrada" igual al de "cadena2", donde se almacena temporalmente la nueva palabra.

La siguiente figura muestra el programa principal de la aplicación:



El programa selecciona aleatoriamente una palabra de una lista, y la guarda en el variable "palabraEntrada". A continuación, fija el valor de la variable "cadenaEntrada" igual al de "palabraEntrada", y llama al procedimiento "aleatorizar" para desordenar los caracteres de "cadenaEntrada". Después de eso, el programa arranca un bucle para obtener la respuesta del usuario. Dentro del bucle, el programa le pide al

usuario que introduzca la palabra ordenada, y usa un bloque "si / si no" para comprobar la respuesta. Esta parte es idéntica a la del juego previo sobre palabras deletreadas.

8.3. EJERCICIOS SCRATCH.

EJERCICIO 42. Escribe un programa que le pida al usuario que introduzca una palabra, y a continuación, que la diga N veces, donde N es el número de letras que contiene la palabra.

EJERCICIO 43. Escribe un programa que le pida al usuario que introduzca una palabra. Después, el programa determina el número de veces que aparece la letra a en la palabra de entrada.

EJERCICIO 44. Escribe un programa que reciba un sustantivo del usuario. A continuación, el programa produce el plural de esa palabra. PISTA: En la palabra de entrada, comprueba la última letra, y la segunda letra comenzando desde la última (esto es, la antepenúltima letra). Para que el programa sea relativamente sencillo, considera que la regla general para formar el plural es la siguiente: si la palabra termina en l, r, n, d, z, y j, el plural se forma añadiendo "es"; en otro caso se forma añadiendo "s".

EJERCICIO 45. Escribe un programa que obtenga del usuario un solo carácter (entre a y z), y obtenga la posición de ese carácter en el alfabeto ($a = 1$, $b = 2$, $c = 3$, etc.). Las letras minúscula y mayúscula se tratan de la misma forma. (PISTA: define una variable llamada "alfa" que aloje en una cadena todas las letras del alfabeto, y usa un bucle para hallar la posición del carácter de entrada en la cadena "alfa").

EJERCICIO 46. Escribe un programa que le pida al usuario una letra del alfabeto y que muestre por pantalla la letra que la precede.

EJERCICIO 47. Escribe un programa que reciba del usuario un entero positivo, y a continuación, muestre la suma de sus dígitos. Por ejemplo, si el usuario introduce 3582, el programa debería mostrar 18. (Por supuesto, el entero debe tener al menos dos dígitos. Tu programa debe verificar este hecho, y en caso contrario, avisar al usuario y pedirle que vuelva a introducir el entero).

EJERCICIO 48. Escribe un programa que reciba una palabra del usuario y muestre las letras de esta palabra en orden inverso.

EJERCICIO 49. Escribe un programa que obtenga un número del usuario y que inserte un espacio entre cada par de dígitos. Por ejemplo, si el número de entrada es 1234, el programa debería dar 1 2 3 4. (PISTA: construye la variable de salida *uniendo* los dígitos del número de entrada con espacios en blanco).

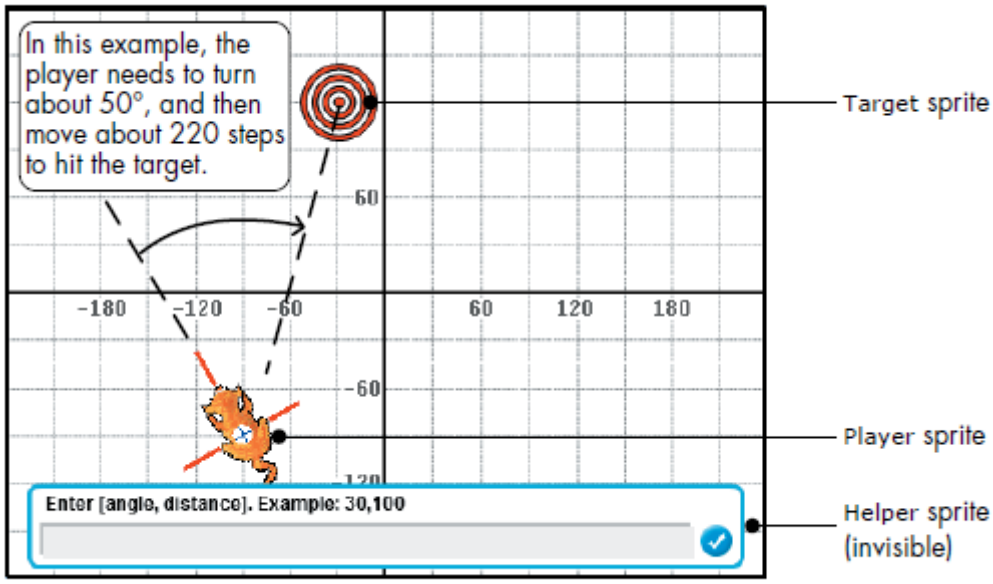
8.4. PROYECTOS SCRATCH.

Los procedimientos que acabamos de ver nos han servido para mostrar los conceptos básicos sobre el procesado de cadenas. (Sin embargo, debemos tener en cuenta que estos ejemplos solo constituyen una pequeña muestra de todas las operaciones que se pueden realizar con las cadenas).

En esta sección usaremos todo lo que hemos aprendido sobre las cadenas para desarrollar algunas aplicaciones interesantes. Y de paso, estas aplicaciones nos permitirán aprender algunos trucos de programación que podremos usar en el futuro en nuestras propias creaciones.

PROYECTO 20. DISPARO (MOVIMIENTO RELATIVO).

Este juego está diseñado para enseñar al usuario el concepto de movimiento relativo de una manera motivadora y divertida. El objetivo del juego es estimar el ángulo de giro y la distancia entre dos objetos en el escenario. La interfaz de usuario está disponible en el archivo **Proyecto 20_sinCodigo.sb2**.



Al empezar el juego, la aplicación posiciona a los objetos "jugador" (player) y "objetivo" (target) en unas localizaciones aleatorias sobre el escenario. A continuación le pide al usuario que estime el ángulo de giro y la distancia que el objeto "jugador" debería moverse para alcanzar al objetivo. Después, el programa mueve al objeto "jugador" de acuerdo con los datos introducidos por el usuario. Si el objeto "jugador" se detiene dentro de un cierto radio alrededor del objetivo, el usuario gana la partida. En caso contrario, el objeto "jugador" vuelve a su posición inicial, y el usuario puede probar otra vez. Comencemos con el programa del objeto jugador.

Programa 1 (jugador): Al pinchar en la bandera verde, el programa envía el mensaje "juegoNuevo" y queda a la espera. Este mensaje lo recibe el objeto "ayudante" (helper), que asigna unas nuevas localizaciones aleatorias a los objetos "jugador" y "objetivo". El objeto "ayudante" ejecutará un procedimiento sencillo (mostrado en la figura) que actualiza las siguientes cinco variables con unos números aleatorios que ubicarán a los objetos "jugador" y "objetivo" sobre el escenario (separados a una cierta distancia):

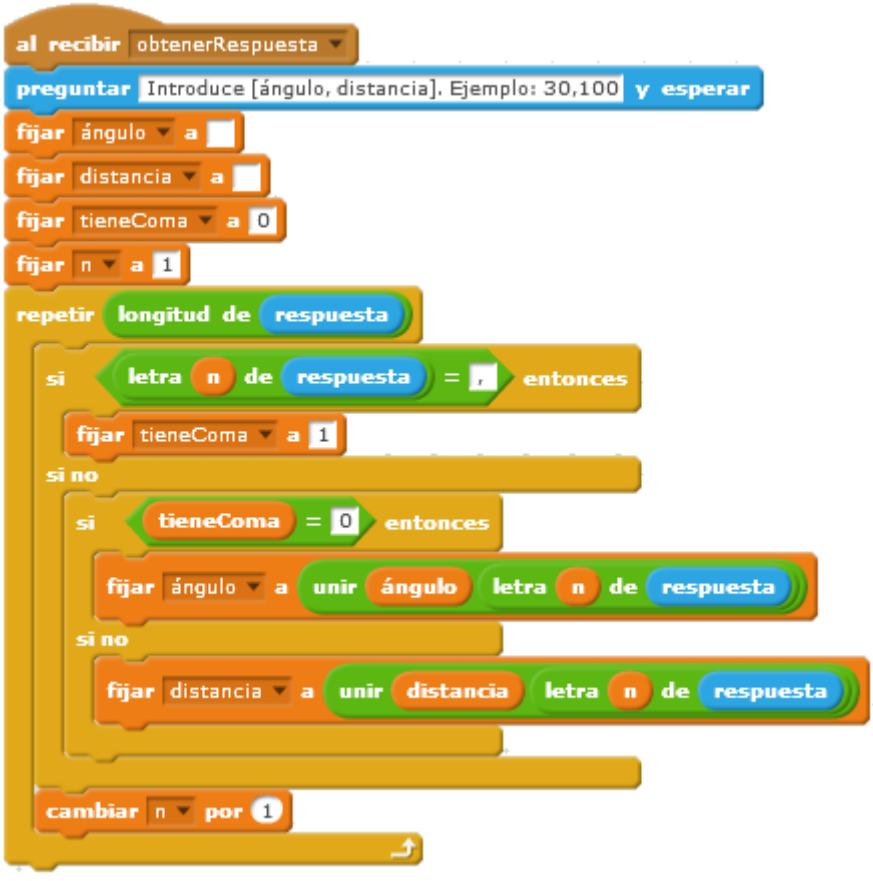
"XJugador" e "YJugador": variables que almacenan las coordenadas x e y del objeto "jugador".
"XObjetivo" e "YObjetivo": variables que almacenan las coordenadas x e y del objeto "objetivo".
"ánguloInicial": variable que almacena la dirección inicial del objeto "jugador".

Un vez que el programa dispone de las nuevas posiciones del "jugador" y del "objetivo", envía el mensaje "comenzarJuego" y queda a la espera. Este mensaje lo recibe el objeto "objetivo", que en respuesta, se ubica en las coordenadas de su nueva ubicación. A continuación, el programa entra en un bucle infinito para darle al jugador varias oportunidades para alcanzar el objetivo. El bucle terminará con un comando "detener (todos)" (presente en el procedimiento "comprobarRespuestas") cuando el jugador alcance al objetivo.

En cada iteración, el bucle fija la posición y dirección iniciales del objeto "jugador" (cuyos valores están almacenados en las variables "XJugador", "YJugador", y "ánguloInicial"), y borra de pantalla todos los trazos de lápiz de una respuesta previa. Después, el bucle envía el mensaje "obtenerRespuesta" y queda a la espera. (En respuesta a este mensaje, el objeto "ayudante" le pide al usuario que introduzca una respuesta. El "ayudante" divide la respuesta del usuario en dos partes (la parte de delante y de detrás de una coma), y

actualiza las variables "ángulo" y "distancia" de forma correspondiente. El programa2 del objeto "ayudante" que se encarga de esta tarea se muestra en la figura). Tras finalizar la espera, el bucle gira hacia la derecha y mueve al objeto "jugador", con su lápiz abajo, según los datos introducidos por el usuario (datos que están alojados en las variables "ángulo" y "distancia"). Con ello, el "jugador" deja un rastro visual de su movimiento, para que el usuario pueda refinar su estimación en la próxima jugada. Finalmente, el bucle ejecuta una llamada al procedimiento "comprobarRespuestas", para ver si las respuestas proporcionadas por el usuario quedan lo suficientemente cerca del objetivo. Con esto termina el bucle y el programa 1 del "jugador".

Programa 1 (ayudante).



Programa 2 (ayudante).

El juego termina si y solo si el objeto "jugador" queda a una corta distancia del objetivo. De ello se encarga el procedimiento "comprobarRespuestas", del que hablaremos más adelante.

Programa 2 (jugador): Hemos dicho que cuando el objeto "jugador" recibe la respuesta del usuario, se mueve de forma correspondiente con su lápiz abajo para dejar un trazo de su movimiento. Este programa arranca al clicar en la bandera, y simplemente configura el color y el tamaño del lápiz (120 y 2), y lo baja.

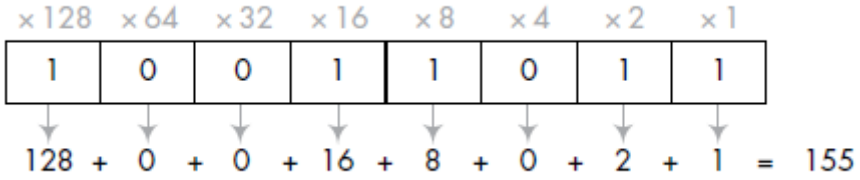
Procedimiento "comprobarRespuestas" (jugador): este procedimiento comprueba si las respuestas proporcionadas por el usuario quedan lo suficientemente cerca del objetivo. El procedimiento comienza comprobando si la distancia al "objetivo" es menor que 20. En ese caso, el objeto dice "¡Ganaste!" durante 2 segundos, y detiene todos los programas. En caso contrario, el objeto dice "Lo siento" durante 2 segundos, y el bucle infinito del programa 1 del jugador comienza de nuevo, de forma que el usuario tiene otra oportunidad para proporcionar unas respuestas más afinadas.

Programa 1 (objetivo): Como ya hemos comentado, este programa empieza cuando el objetivo recibe el mensaje "comenzarJuego" del jugador, y simplemente ubica al objetivo en las coordenadas "XObjetivo" e "YObjetivo", y envía al objeto una capa hacia atrás.

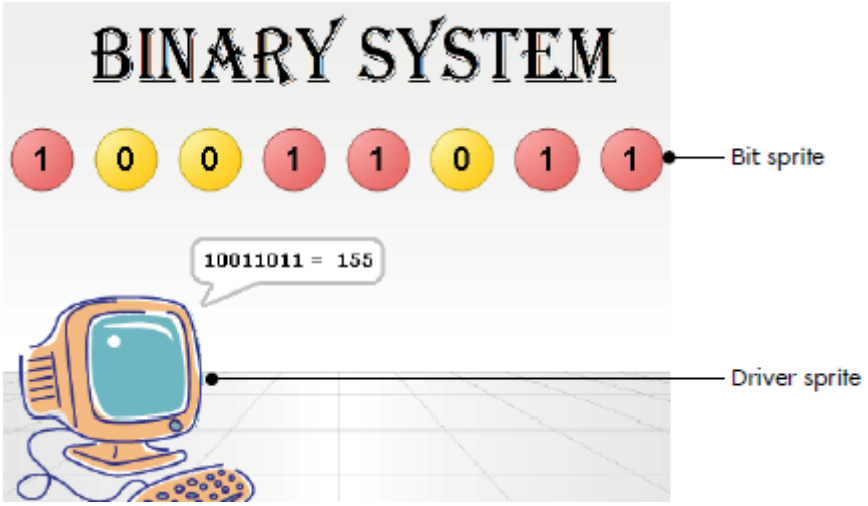
Al terminar, guarda el proyecto como **Proyecto 20.sb2**.

PROYECTO 21. CONVERSOR BINARIO A DECIMAL.

Los números binarios (base-2) solo tienen dos dígitos posibles: 0 y 1. La mayoría de ordenadores operan y se comunican con números binarios. Sin embargo, los humanos preferimos usar números en el sistema decimal (base-10), probablemente porque tenemos 10 dedos. En esta sección desarrollaremos una aplicación que convierta un número binario a su equivalente decimal. Comencemos explicando cómo convertir de binario a decimal. La figura muestra un ejemplo usando el número binario 10011011.



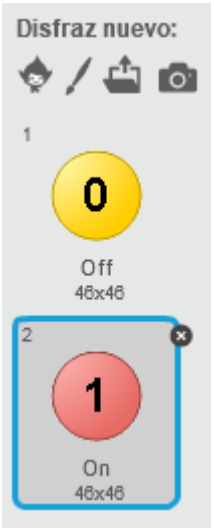
Todo lo que hay que hacer es multiplicar cada dígito binario por su **valor posicional** correspondiente, y después sumar todos esos productos. Los valores posicionales se corresponden con potencias de la base, que se van incrementando de derecha a izquierda, y donde la primera posición tiene una potencia de 0. Como el binario es base 2, el dígito más a la derecha tiene un valor posicional de $2^0 = 1$, así que debemos multiplicar ese dígito por 1. El siguiente dígito se multiplicaría por $2^1 = 2$, el siguiente por $2^2 = 4$, y así sucesivamente.



EJERCICIO 50. CONVIERTE A DECIMAL.

Para comprobar que has entendido el proceso de conversión de binario a decimal, convierte los siguientes números a decimal usando lápiz y papel: (a) 10101, (b) 1101001, y (c) 11000011.

La interfaz de usuario de la aplicación está disponible en el archivo **Proyecto 21_sinCodigo.sb2**. Esta interfaz se muestra en la figura.



El programa le pide al usuario que introduzca un número binario de 8 bits. A continuación, muestra el número binario introducido sobre el escenario mediante el objeto "bit", que usa dos disfraces ("off" y "on") para representar el 0 y el 1. El programa computa el número decimal equivalente, y el objeto "controlador" (driver), que tiene un disfraz de ordenador, muestra ese valor al usuario. Vamos a escribir el código de esta aplicación:

Programa 1 (controlador): El programa comienza al pinchar en la bandera verde, evento que es detectado por el objeto "controlador", que envía el mensaje "inicializar" y queda a la espera. Este mensaje lo recibe el objeto "bit", el cual, como veremos después, prepara la pantalla para la siguiente partida. A continuación, el programa del controlador le pide al usuario que inserte un número binario de 8 dígitos, y almacena la respuesta en la variable "binario". Después envía el mensaje "BinarioADecimal" y queda a la espera. De nuevo, este mensaje lo recibe el objeto "bit", que convierte el número binario a decimal y almacena el resultado en la variable "decimal". Por último, el programa muestra por pantalla el resultado de la conversión.

Programa 1 (bit): Éste es el programa que ejecuta el objeto "bit" en respuesta al mensaje "inicializar". Este programa dibuja por pantalla un patrón de bits que muestra 8 ceros (usando el disfraz "off" del objeto "bit", ver figura).

Para ello, el programa comienza limpiando la pantalla, y fijando el disfraz "off". A continuación, el programa mueve al objeto "bit" a la posición $(210, 65)$ para dibujar el dígito binario más a la derecha. Después, arranca un bucle "repetir (8)" para dibujar los 8 bits del número binario. A cada pasada del bucle, el programa sella una copia del objeto por pantalla, y se mueve hacia la izquierda 60 pasos para sellar la siguiente copia (ver figura). Al salir del bucle, el programa cambia al disfraz "on", y oculta el objeto.

Como veremos en breve, siempre que en la cadena introducida por el usuario aparezca un bit a 1, el programa debería estampar el disfraz "on" del objeto "bit" para poner a 1 el dígito correspondiente.

Cuando el usuario introduce el número binario a convertir, el objeto "bit" recibe el mensaje "BinarioADecimal" y ejecutar el siguiente programa:

Programa 2 (bit): El programa comienza inicializando todas las variables que usará:

- "longitud" es el número de bits del número binario introducido por el usuario, por lo que debe inicializarse al valor de la longitud de la cadena "binario".
- "pos" comienza apuntando al dígito más a la derecha del número binario introducido por el usuario, por lo que su valor debe inicializarse igual al valor del "longitud".
- "peso" empieza con un valor igual al valor posicional del dígito más a la derecha del número binario, esto es, igual a 1.
- "decimal" es la variable donde almacenaremos el resultado de la conversión, y se inicializa a 0.
- "xPos" comienza en la coordenada x de la imagen del dígito binario más a la derecha, esto es, en 210.

A continuación, el programa arranca un bucle "repetir ( )" con un número de iteraciones igual a la longitud de la cadena binaria de entrada. Dentro del bucle, el procedimiento comprueba cada dígito para ver si se trata de un 0 o de un 1 (usando el condicional mostrado en la figura). Si el bucle encuentra un 1, suma el valor actual de "peso" a "decimal", va a la posición de ese dígito (que es $(x, y) = (xPos, 65)$), se muestra, sella una copia del disfraz "on" sobre la imagen previa del dígito a 0, y se vuelve a ocultar.

Fuera del condicional, pero todavía dentro del bucle, el programa actualiza el valor de varias variables para la siguiente iteración:

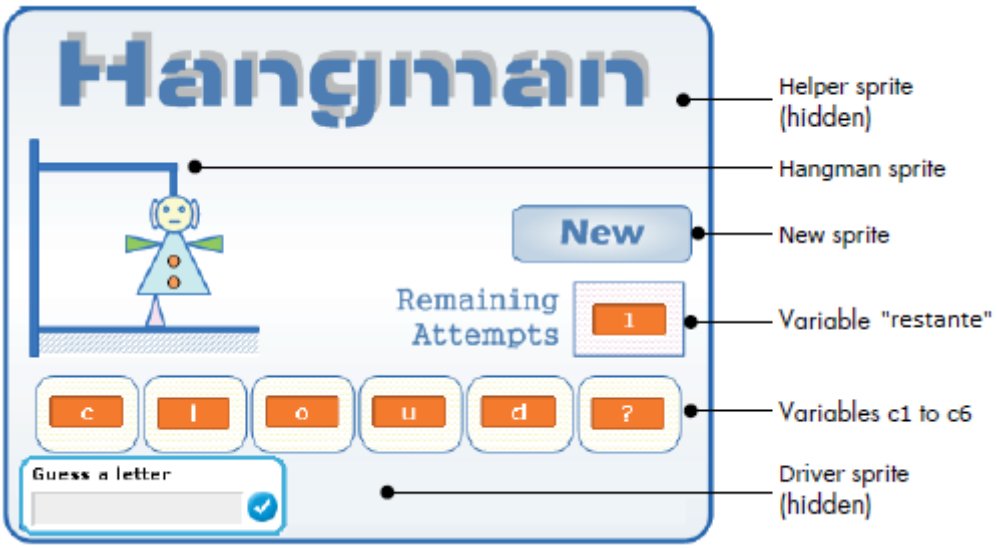
- "pos" se actualiza para apuntar al dígito que está a la izquierda del que acabamos de procesar. Como empezó en la posición del dígito más a la derecha ("pos"$= 8$), para pasar al siguiente a la izquierda debemos cambia su valor actual en -1 unidades.
- "xPos" pasa a tomar el valor de la posición en x de la imagen del siguiente dígito, que está 60 pasos a la izquierda de la imagen actual.
- "peso" comenzó tomando el valor posicional del dígito más a la derecha ("peso"$= 1$). Para actualizar su valor, multiplicamos por 2 su valor actual, lo que significa que irá tomando los valores 1, 2, 4, 8, 16, etc.

Con esto termina el programa, y el proyecto. Guarda tu trabajo con el nombre **Proyecto 21.sb2**.

AMPLIACIÓN: Haz que el objeto "controlador" valide el número binario introducido por el usuario antes de enviar el mensaje "BinarioADecimal" al objeto "bit". Debes verificar (1) que el número introducido es un número binario (esto es, que el número solo contiene unos y ceros), y (2) que la longitud de la cadena de entrada es de, como mucho, 8 dígitos.

PROYECTO 22. JUEGO DEL AHORCADO.

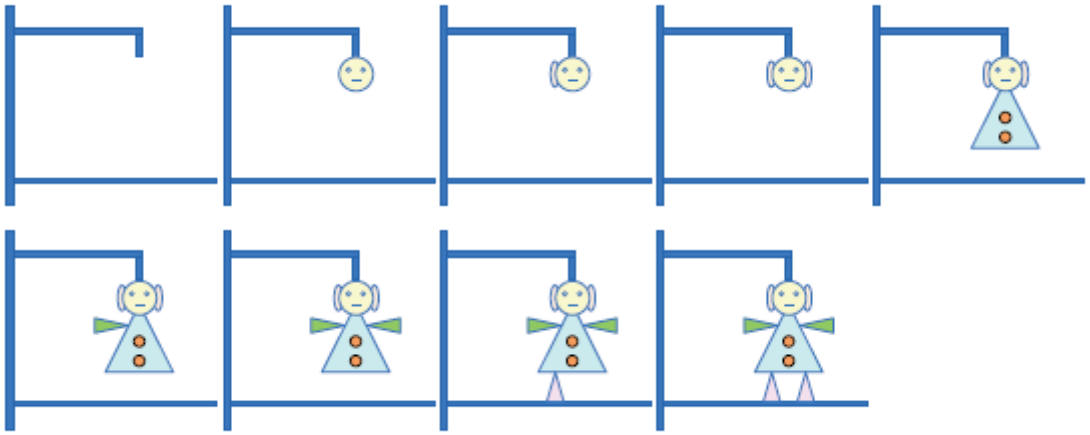
En esta sección vamos a programar el clásico juego del ahorcado. El programa elige al azar una palabra en inglés de 6 letras, y muestra por pantalla un signo de interrogación para cada letra. El jugador tiene 8 oportunidades para adivinar las letras de la palabra. Si el jugador adivina correctamente una de las letras, el programa muestra todas las ocurrencias de esa letra en la palabra secreta. En caso contrario, el programa muestra otra pieza de la figura del ahorcado (la cabeza, el cuerpo, el brazo izquierdo, etc.). Después de 8 fallos, el programa completa la figura, y el jugador pierde la partida. Si el jugador adivina la palabra secreta en menos de 8 intentos, gana la partida. La interfaz de usuario está disponible en el archivo **Proyecto 22_sinCodigo.sb2**, tal y como se muestra en la figura:



La aplicación contiene estos cuatro objetos:

1) "Conductor" (driver): este objeto se oculta cuando comienza la partida, le pide al usuario que adivine una letra, y procesa las respuestas del usuario. Cuando el juego termina, el objeto se muestra con uno de sus dos disfraces ("win" o "lose"):

2) "Ahorcado" (hangman): este objeto muestra progresivamente la imagen del ahorcado. Tiene un total de 9 disfraces, uno por cada parte adicional de la figura del ahorcado (ver figura):



3) "Nuevo" (new): este objeto muestra el botón "new" (nuevo) por pantalla.
4) "Ayudante" (helper): este objeto invisible muestra las letras adivinadas por el jugador, así como el número de intentos restantes. Usa 7 variables con monitores posicionados en las localizaciones adecuadas. El uso de este objeto para actualizar los monitores permite separar el código del juego del código de la interfaz de usuario. Así, y a modo de ejemplo, podríamos cambiar la tipografía de las letras en pantalla sin tener que tocar el resto de la aplicación.

Programa 1 (nuevo): La partida comienza cuando el usuario pulsa el botón "nuevo", momento en el que este objeto envía el mensaje "nuevaPartida" para alertar al objeto "controlador" de que ha empezado una partida nueva.

Programa 1 (controlador): Cuando el objeto "controlador" recibe el mensaje "nuevaPartida", ejecuta un programa que empieza llamando al procedimiento "inicializar", que se encarga de resetear la interfaz de usuario. A continuación, arranca un bucle infinito para leer las letras propuestas por el usuario. A cada pasada del bucle, el "controlador" le pide al usuario que adivine una letra de la palabra secreta, y espera su respuesta. Cuando el usuario inserta una letra, el programa llama al procedimiento "procesarRespuesta", que se encarga de actualizar un indicador (llamado "contieneLaLetra") que informa de si la palabra secreta contiene o no esa letra. Al volver del procedimiento, el programa comprueba el valor del indicador "contieneLaLetra". Si su valor es igual a 0, llama al procedimiento "procesarRespuestaErrónea"; en caso contrario, llama al procedimiento "procesarRespuestaCorrecta".

Hablaremos de todos estos procedimientos más adelante. Por ahora, vamos a escribir el procedimiento "inicializar":

Procedimiento "inicializar" (controlador): durante la inicialización, el objeto "controlador" comienza ocultándose, e inicializando la variable "palabraMostrada" a una cadena con 6 signos de interrogación. (La variable "palabraMostrada" sirve para mostrar por pantalla el progreso de la partida: comienza con la cadena "??????", y se va actualizando cada vez que el usuario adivina una letra de la palabra secreta).

Después, fija el valor de "intentosRestantes" (el número de intentos que le quedan al jugador) a 8. A continuación, elige la "palabraSecreta" de una lista predefinida de palabras de 6 letras (ver figura), y envía el mensaje "actualizar" para que el objeto "ayudante" asigne a sus variables (cuyos monitores están visibles por pantalla) los valores adecuados. Finalmente, el procedimiento envía el mensaje "reiniciar" al objeto "ahorcado". Cuando "ahorcado" recibe este mensaje, cambia a su disfraz de inicio, que muestra una horca vacía.

Programa 1 (ayudante): Recordemos que el objeto "ayudante" se encarga de mostrar las letras adivinadas por el jugador, así como el número de intentos que aún tiene disponibles. El programa de este objeto arranca al recibir el mensaje "actualizar", y simplemente almacena en 6 variables (llamadas "c1", "c2", ... , "c6") las distintas letras que contiene la variable "palabraMostrada", y fija el valor de la variable "restantes" al valor de la variable "intentosRestantes". Este programa está disponible en la figura:

Programa 1 (ahorcado): Este programa comienza al recibir el mensaje "reiniciar", y simplemente cambia al disfraz "start", que muestra un patíbulo vacío.

Ahora, para entender cómo funciona el procedimiento "procesarRespuesta", consideremos un ejemplo sencillo: Supongamos que la palabra secreta es "across", y que estamos en la primera ronda de la partida (lo que significa que el valor de "palabraMostrada" es "??????"). Si el jugador propone la letra "r", el procedimiento "procesarRespuesta" debería ajustar el valor de "contieneLaLetra" a 1 para indicar una letra acertada, actualizar el valor de "palabraMostrada" a "??r???" para mostrar la posición de esa letra en la palabra secreta a adivinar, y fijar el valor de la variable "interrogCont" (el número de signos de interrogación aún presentes en "palabraMostrada") a 5. Cuando "interrogCont" llega a 0, el usuario ha adivinado todas las letras de la palabra secreta.

Procedimiento "procesarRespuesta" (controlador): el procedimiento empieza reseteando el valor del indicador "contieneLaLetra" y el de la variable "interrogCont", ambos a 0. (Este procedimiento incrementará el valor de "interrogCont" en 1 unidad por cada letra desconocida en la palabra secreta). También inicia el valor de la variable temporal "temp", que usará para construir la palabra a mostrar tras cada intentona del jugador, a una cadena vacía. Además, fija el valor de la variable "pos" (el contador del bucle) a 1. A continuación, arranca un bucle "repetir ( )" que se repetirá un número de veces igual a la longitud de "palabraSecreta". Este bucle examina cada letra de "palabraSecreta", usando "pos" como índice. El bucle comienza almacenando en la variable "ch" la letra ubicada en la posición "pos" de la cadena "palabraSecreta". Si la letra que se está examinando es igual a la letra propuesta por el usuario (letra almacenada en la variable de Scratch "respuesta"), el indicador "contieneLaLetra" se fija a 1. En caso contrario, el procedimiento ajusta el valor de "ch" a la letra en la posición correspondiente ("pos") de la

variable "palabraMostrada". En cualquier caso, el programa adjunta "ch" al final de la cadena "temp", como ilustra la figura. Después, el bucle sigue el rastro del número de signos de interrogación en la cadena "palabraMostrada": Si el valor de "ch" es igual a "?", el programa cambia el valor de "interrogCont" en 1 unidad. El bucle termina incrementando el valor de "pos" en 1 unidad, para examinar la siguiente letra de "palabraSecreta". Cuando el bucle termine, la variable "temp" contendrá las seis letras que se deben mostrar por pantalla, teniendo en cuenta la última letra adivinada por el usuario. Así pues, la última instrucción del programa es ajustar el valor de "palabraMostrada" al valor de "temp".

palabraSecreta=across
palabraMostrada=??????
respuesta=r

Iteration	ch	ch	temp
1	a	?	?
2	c	?	??
3	r	r	??r
4	o	?	??r?
5	s	?	??r??
6	s	?	??r???

Cuando el procedimiento "procesarRespuesta" retorna, el programa que lo llamó (programa 1 del "controlador") comprueba el valor de "contieneLaLetra" para ver si el jugador ha respondido correctamente. Si no es así, llama al procedimiento "procesarRespuestaErrónea":

Procedimiento "procesarRespuestaErrónea" (controlador): Este procedimiento comienza enviando el mensaje "respuestaErrónea", para notificar al objeto "ahorcado" que debe mostrar su siguiente disfraz. A continuación, el procedimiento disminuye el valor de "intentosRestantes" en 1 unidad. Ahora, si ya no quedan intentos, el procedimiento muestra por pantalla cuál es la palabra secreta (ajustando el valor de "palabraMostrada" al de "palabraSecreta" y enviando al objeto "ayudante" el mensaje "actualizar"), cambia el disfraz a "lose" y se muestra, y termina la partida. En caso contrario, el procedimiento envía un mensaje "actualizar" al objeto "ayudante" para mostrar cuántos intentos le quedan al usuario.

Programa 2 (ahorcado): Este programa comienza al recibir el mensaje "respuestaErrónea", y simplemente cambia a su siguiente disfraz.

Si la letra propuesta por el usuario es correcta, el programa 1 del controlador llama al procedimiento "procesarRespuestaCorrecta":

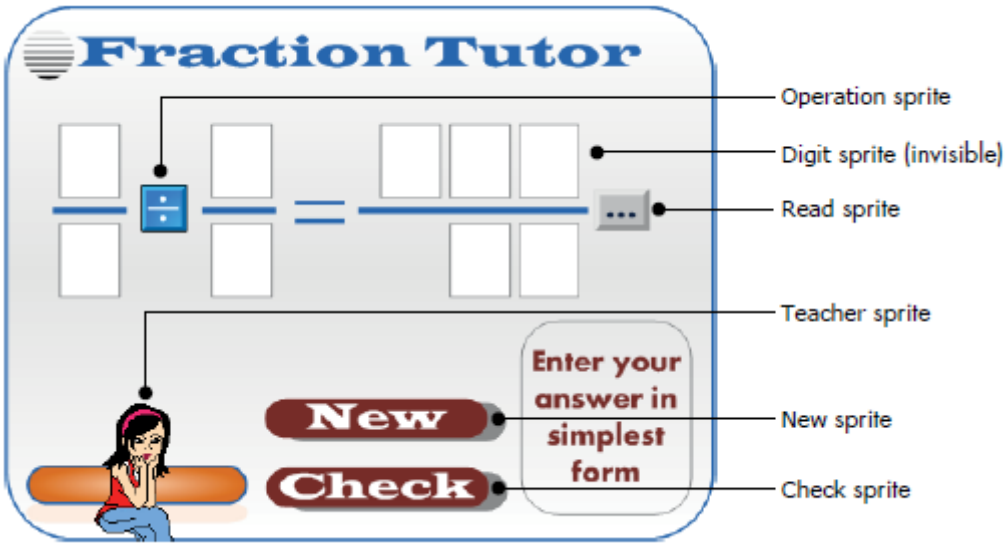
Procedimiento "procesarRespuestaCorrecta" (controlador): Este procedimiento comienza enviando el mensaje "actualizar" para que el "ayudante" muestre la letra que el usuario ha acertado, y a continuación, comprueba el valor de "interrogCont". Si "interrogCont" es 0, el jugador ha adivinado correctamente todas las letras de la palabra secreta, y el objeto "controlador" debe cambiar a su disfraz "win", mostrarse, y terminar la partida.

Con esto hemos terminado. Guarda la aplicación como **Proyecto 22.sb2**.

AMPLIACIÓN: El programa del ahorcado no comprueba los datos introducidos por el usuario. El usuario podría introducir un carácter inválido (una cifra, o un símbolo como @, &, etc.), o incluso toda una palabra. Modifica el programa para que rechace cualquier entrada que no sea válida.

PROYECTO 23. OPERACIONES CON FRACCIONES.

En este último proyecto vamos a desarrollar un juego educativo para enseñar fracciones. La interfaz de este juego está disponible en el archivo **Proyecto 23_sinCodigo.sb2**, y se muestra en la figura:



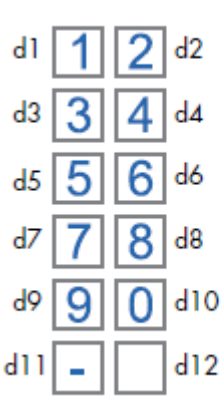
El jugador selecciona una operación (+, −, ×, o ÷) y clica en el botón "new" para que se le proponga un nuevo problema. Cuando el jugador proporciona una respuesta (botón "read") y pincha en el botón "check" (comprobar), el objeto "profesor" (la imagen de la mujer) comprueba esa respuesta y da una contestación.

La aplicación contiene 6 objetos. "Operación" (operation) le permite al usuario elegir la operación matemática a efectuar. "Leer" (read) muestra el botón para introducir la respuesta, "nuevo" (new) muestra el botón de nuevo, y "comprobar" (check) muestra el botón para comprobar la respuesta. El objeto "profesor" comprueba la respuesta del usuario, y un objeto invisible llamado "dígito" (digit) estampa en pantalla los números que se corresponden con el problema en curso.

Cuando el usuario pincha en el botón "new", el objeto "nuevo" ejecuta el siguiente programa:

Programa 1 (nuevo): este programa comienza asignando valores aleatorios entre 1 y 9 al numerador y al denominador de las dos fracciones a operar. Esos números se almacenan en las variables "num1", "den1", "num2", y "den2". Después, el programa envía el mensaje "nuevoProblema" (y queda a la espera) para indicarle al objeto "dígito" que debe estampar esos números por pantalla.

El objeto "dígito" tiene 12 disfraces (llamados "d1", "d2", … , "d12"), como muestra la figura. Cuando este objeto recibe el mensaje "nuevoProblema", estampa los disfraces que representan los numeradores y denominadores de las dos fracciones a operar.



Programa 1 (dígito): Este programa comienza al recibir el mensaje "nuevoProblema". El programa empieza limpiando la pantalla, y mostrando al objeto. A continuación, llama al procedimiento "estampar ( ) en ( ) ( )" para estampar en pantalla los valores de "num1", "den1", "num2", y "den2" en las posiciones (x, y) adecuadas. Finalmente, el programa esconde al objeto. Este programa se muestra en la figura.

Procedimiento "estampar ( ) en ( ) ( )" (objeto "dígito"): este procedimiento usa varios bloques "si / si no" anidados para determinar qué disfraz se corresponde con el dígito a estampar. El procedimiento recibe 3 parámetros: el parámetro "dígito" (el dígito a estampar por pantalla), y las coordenadas "x" e "y" de la localización donde queremos estampar ese dígito:

Definición del procedimiento.

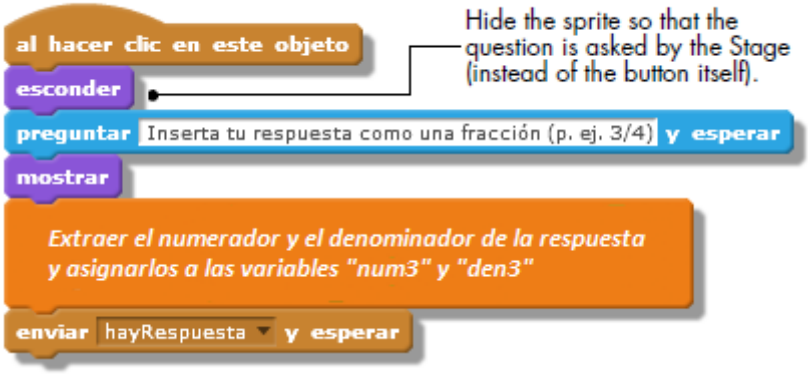
Llamada al procedimiento.

(1) El procedimiento comienza comprobando si el parámetro "dígito" es igual a "0". En ese caso, cambia al disfraz "d10". En caso contrario, realiza una segunda comprobación: (2) Si "dígito" es igual a "–", el procedimiento cambia al disfraz "d11". En caso contrario, realiza una tercera comprobación: (3) Si el valor de "dígito" es mayor que 0 y menor que 10 (esto es, si dígito es igual a 1, 2, 3, …, 9), el procedimiento cambia al disfraz correspondiente (esto es, al disfraz "d1", "d2", "d3", …, "d9"). Esto podemos hacerlo mediante el bloque:

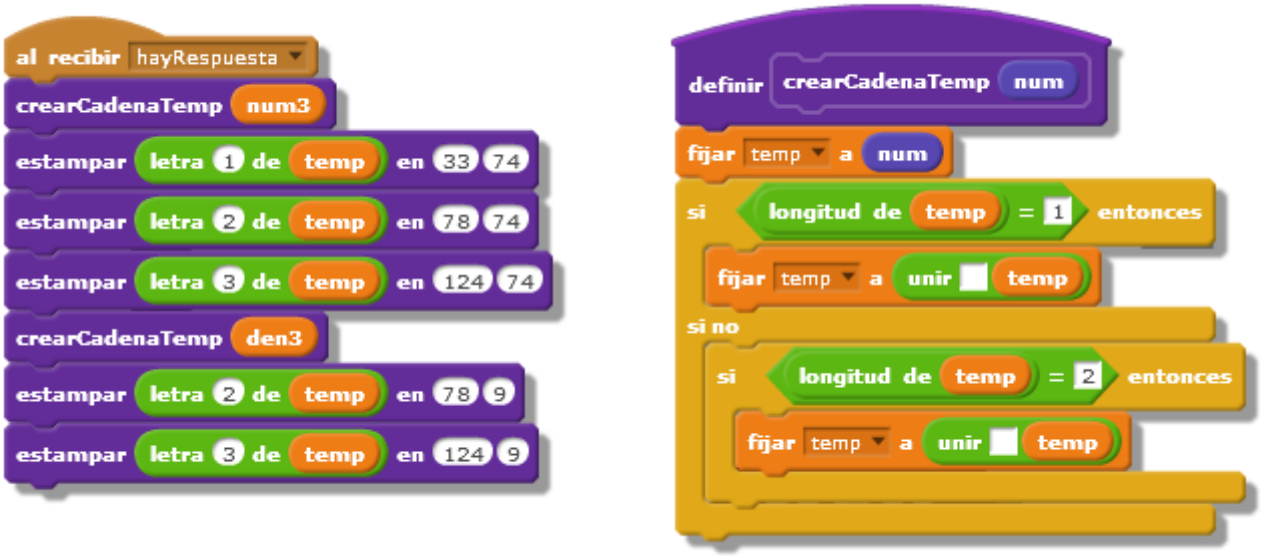
En caso contrario, la única alternativa que nos queda es que el dígito sea un espacio en banco, por lo que el procedimiento debe cambiar al disfraz "d12". Con esto finaliza esta secuencia de condicionales anidados. Tras haber fijado el disfraz adecuado, el procedimiento acude a la posición (x, y) especificada por los parámetros "x" e "y", y sella la imagen del disfraz seleccionado. Con esto termina el procedimiento.

Con todo lo hecho hasta ahora, el problema a resolver se muestra por pantalla. En ese momento, el usuario puede pinchar en el botón "leer" (read) para insertar su respuesta. Vamos a escribir el programa de control de este botón:

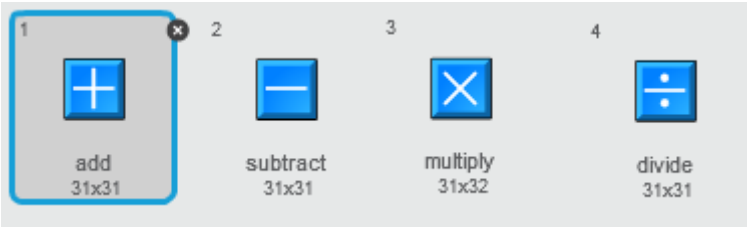
Programa 1 (leer): El programa comienza cuando el usuario clica sobre el objeto. El programa esconde al objeto, y le pide al usuario que introduzca su respuesta en la forma de una fracción (ver figura). A continuación, el programa muestra al objeto, y realiza un procesado de la respuesta para extraer el numerador y el denominador, y asignarlos a las variables "num3" y "den3". (Este procesado es muy similar al que hicimos en el proyecto 20 (disparo) para extraer el ángulo y la distancia de la respuesta, y podemos inspirarnos en él). Finalmente, el programa envía el mensaje "hayRespuesta" para decirle al objeto "dígito" que muestre por pantalla la respuesta del usuario. La figura muestra la estructura general del programa:



Programa 2 y procedimiento "crearCadenaTemp ( )" (objeto "dígito"): Cuando el objeto "dígito" recibe el mensaje "hayRespuesta", estampa los dígitos de "num3" y "den3" en las posiciones correctas de la pantalla, de la misma forma que estampó los numeradores y los denominadores de la dos fracciones a operar. Este programa (y el procedimiento "crearCadenaTemp ( )" que utiliza) se muestran en la figura:



Después de insertar su respuesta, el usuario puede clicar en el botón "comprobar" (check) para ver si la respuesta es correcta. El programa del botón "comprobar" simplemente envía un mensaje "comprobarRespuesta" para informar al resto de objetos de que el usuario quiere comprobar su respuesta. Este mensaje lo recibe y procesa el objeto "profesor", que ejecuta el siguiente programa:



Programa 1 (profesor): Al recibir el mensaje "comprobarRespuesta", el programa comprueba cuál es el número del disfraz actual del objeto "operación" (operation) con el bloque "(# de disfraz) de (operation)" de la categoría "sensores". El disfraz actual de "operación" le dice a "profesor" qué procedimiento ("suma", "resta", "multiplicación", o "división") debe ejecutar. Esto debemos hacerlo con varios condicionales anidados:

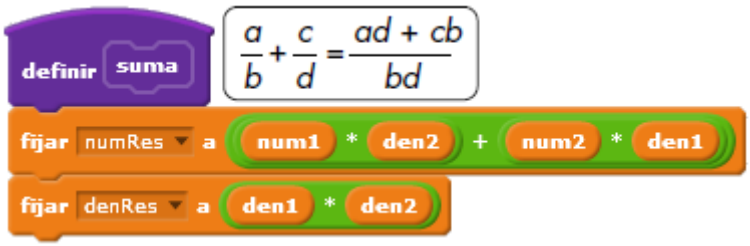
Los procedimientos de "suma", "resta", "multiplicación", y "división" tomarán las variables "num1", "den1", "num2", y "den2" como entradas, y calculará el valor de las variables "numRes" y "denRes", que representan el numerador y el denominador de la respuesta correcta. (Estos 4 procedimientos los construiremos más adelante).

Después de hallar la respuesta correcta, el programa debe simplificar la fracción resultante. (Por ejemplo, la respuesta $2/4$ debe simplificarse a $1/2$). Para realizar esta reducción, el programa halla el máximo común divisor (MCD) de "numRes" y "denRes" mediante el procedimiento "hallarMCD ( ) ( )". (Hablaremos de este procedimiento más adelante). Tras hallar el MCD (valor que queda almacenado en la variable "mcd"), el programa divide "numRes" y "denRes" por ese número, y finalmente llama al procedimiento "darComentario" para mostrar por pantalla si la respuesta del usuario es o no es la correcta. Con esto termina el programa 1 del "profesor".

Ahora vamos a escribir los procedimientos de "suma", "resta", "multiplicación", y "división". Estos procedimientos computan el resultado efectuando una operación de la forma:

$$\frac{num1}{den1}[+,-,\times,\div]\frac{num2}{den2}=\frac{numRes}{denRes}$$

, y almacena el resultado en las variables "numRes" y "denRes". A modo de ejemplo, el procedimiento para efectuar la suma es:



Escribe tú los procedimientos para la resta, la multiplicación, y la división. Como ya deberíamos saber, las técnicas para computar la resta, multiplicación, y división de dos fracciones son:

$$\frac{a}{b}-\frac{c}{d}=\frac{ad-cb}{bd} \qquad \frac{a}{b}\times\frac{c}{d}=\frac{ac}{bd} \qquad \frac{a}{b}\div\frac{c}{d}=\frac{ad}{bc}$$

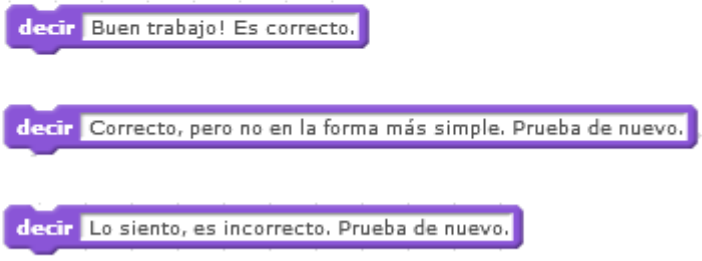
Continuemos con el procedimiento para hallar el máximo común divisor (MCD). Este procedimiento recibe como entradas los parámetros "num1" y "num2", que representan los números cuyo MCD deseamos hallar.

Para entender cómo funciona este procedimiento, consideremos el caso en el que "num1"$=-10$ y "num2"$=6$. Debemos hallar el entero positivo más grande que divida de forma exacta (sin dejar resto) a "num1" y "num2". El procedimiento comienza fijando el valor de "mcd" (la variable que almacena el MCD buscado) igual al valor absoluto *más pequeño* de los dos números (6 en este caso). A continuación, un bucle comprueba los números 6, 5, 4, etc., hasta que tanto "num1" como "num2" sean divisibles (sin resto) por el número que estamos comprobando. Ése es el resultado que estamos buscando. En este ejemplo, "mcd" será igual a 2, porque ambos números (-10 y 6) pueden dividirse por 2 sin dejar resto.

Procedimiento "hallarMCD" (profesor): el procedimiento empieza comprobando si el valor absoluto de "num1" es mayor que el valor absoluto de "num2". En caso afirmativo, ajusta el valor de "mcd" al valor absoluto de "num2"; en caso negativo, ajusta el valor de "mcd" al valor absoluto de "num1". A continuación, arranca un bucle infinito que compruebe si "num1" y "num2" son ambos divisibles por el valor de "mcd". En ese caso, hemos encontrado el valor final del MCD, y el procedimiento se detiene a sí mismo para volver al programa que lo llamó. En caso contrario, el procedimiento cambia el valor de "mcd" en -1 unidades, y el bucle vuelve a repetirse.

El último procedimiento a escribir es el procedimiento "darComentario", que compara la respuesta del usuario (almacenada en las variables "num3" y "den3") con la respuesta correcta (almacenada en "numRes" y "denRes") y muestra por pantalla el mensaje apropiado (ver figura):

Examples	
Correct Answer $\frac{numRes}{denRes}$	User Answer $\frac{num3}{den3}$
$\frac{3}{4}$	$\frac{3}{4}$
$\frac{3}{4}$	$\frac{6}{8}$
$\frac{3}{4}$	$\frac{2}{3}$



Procedimiento "darComentario" (profesor): Este procedimiento simplemente compara los valores de "numRes" y "num3" por un lado, y los valores de "denRes" y "den3" por otro. Si las dos parejas de valores son iguales, la respuesta del usuario es totalmente correcta, y mostramos por pantalla el texto adecuado. En caso contrario, si los productos cruzados son iguales (esto es, si $numRes * den3 = denRes * num3$), es porque la respuesta del usuario es correcta, pero no está simplificada, y mostramos el mensaje correspondiente. Si no se da ninguno de los dos casos previos, es porque la respuesta del usuario no es correcta, y mostramos el mensaje apropiado.

Ya has terminado. Prueba la aplicación y guárdala con el nombre **Proyecto 23.sb2**.

EJERCICIO 51. COMPARAR FRACCIONES.

En este ejercicio vas a crear un juego que le permita al usuario comparar dos fracciones. La interfaz de usuario está disponible en el archivo **Ejercicio 51_sinCodigo.sb2**, y se muestra en la figura. Al clicar el botón "new", el juego elige aleatoriamente dos fracciones para compararlas. El usuario elige mayor que (>), menor que (<), o igual que (=) clicando en el botón "operator". Cuando el usuario clica en el botón "check" (comprobar), el juego comprueba la respuesta y proporciona algún comentario. Completa el código del archivo indicado con los programas necesarios para realizar estas tareas. Cuando termines, guarda el archivo como **Ejercicio 51.sb2**.

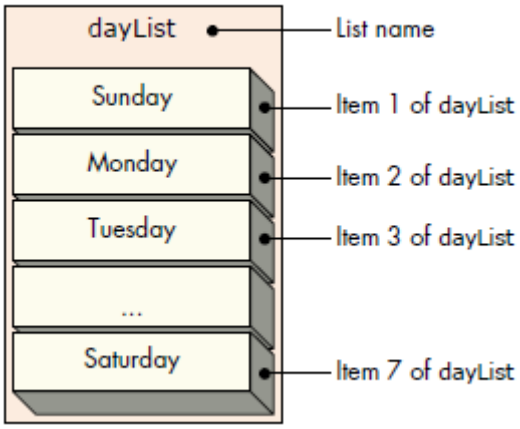
9. LISTAS.

Los programas que hemos escrito hasta ahora usaban variables ordinarias para almacenar datos individuales. Pero las variables no son útiles cuando queremos guardar un conjunto de valores, como por ejemplo, los números de teléfono de nuestros amigos, los nombres de los libros de una biblioteca, o los registros diarios de las temperaturas máxima y mínima en una localidad durante un mes. En efecto, si quisiéramos que nuestro programa recordase los números de teléfono de 20 de nuestros amigos, necesitaríamos 20 variables. Evidentemente, escribir un programa que gestione 20 variables sería muy engorroso. En este capítulo aprenderemos a manejar las *listas*, un tipo de datos propio de Scratch que nos permite agrupar valores relacionados.

El capítulo comienza explicando cómo crear listas en Scratch, qué comandos podemos usar con ellas, y cómo llenarlas con los datos que introduzca un usuario. A continuación hablaremos sobre las listas numéricas y las operaciones más comunes que podemos realizar con ellas, como hallar el valor máximo, el valor mínimo, y el valor promedio de sus elementos. Después, aprenderemos un algoritmo para clasificar y ordenar los elementos de una lista. Finalmente, escribiremos varios programas que ilustran algunas aplicaciones útiles de las listas.

9.1. LISTAS EN SCRATCH.

Una **lista** es como un contenedor donde podemos guardar múltiples valores y acceder a ellos. Podemos imaginar que una lista es una especie de cómoda con muchos cajones, donde cada cajón almacena un solo objeto. Al crear una lista le ponemos un nombre, igual que lo haríamos con una variable. Después, podemos acceder a los elementos individuales de la lista usando su posición en la lista. Por ejemplo, la figura muestra una lista llamada "dayList" que almacena los nombres de los siete días de la semana:

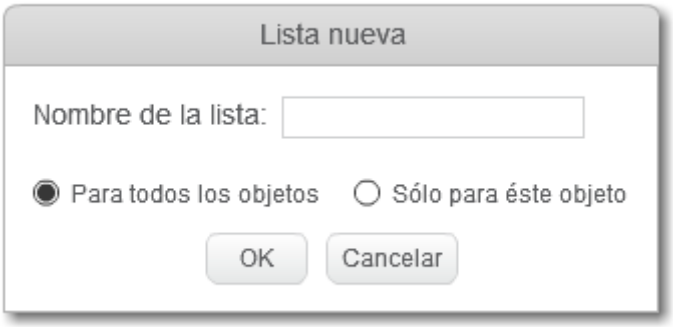


Podemos acceder a los elementos contenidos en la lista usando sus *índices de almacenamiento* (o posiciones). En Scratch, el primer elemento tiene un índice de 1, el segundo un índice de 2, y así sucesivamente. Por ejemplo, como el martes (tuesday) es el tercer elemento de la lista, su índice es 3. Por consiguiente, podemos acceder al tercer elemento de la lista "dayList" usando un comando de la forma "elemento (3) de (dayList)".

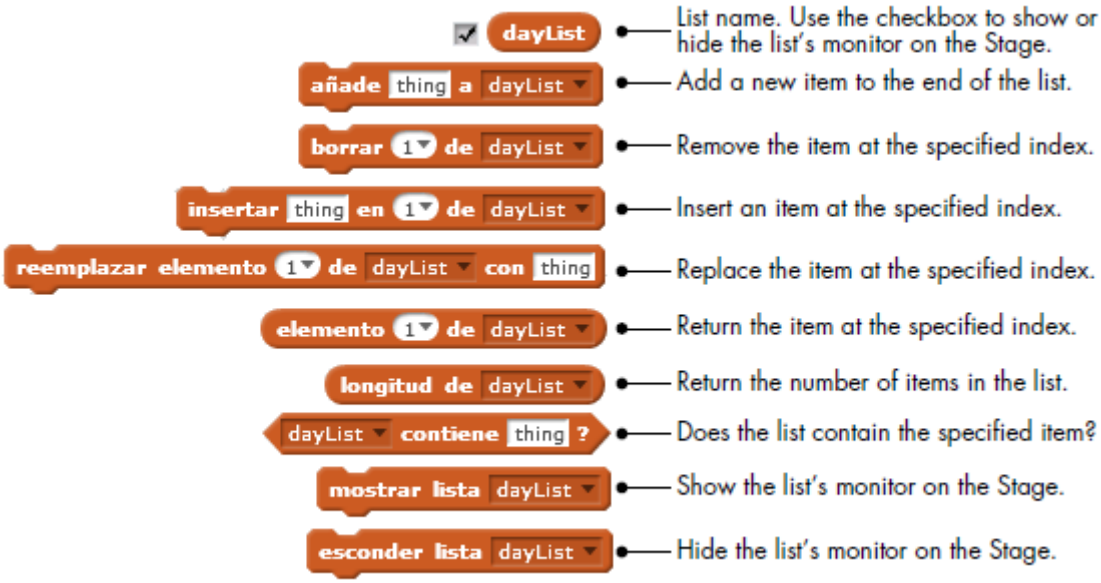
A continuación, veremos cómo crear listas en Scratch. También estudiaremos los comandos que nos permiten gestionar y manipular listas en nuestros programas, y entenderemos la forma en la que responde Scratch a comandos inválidos.

CREAR LISTAS.

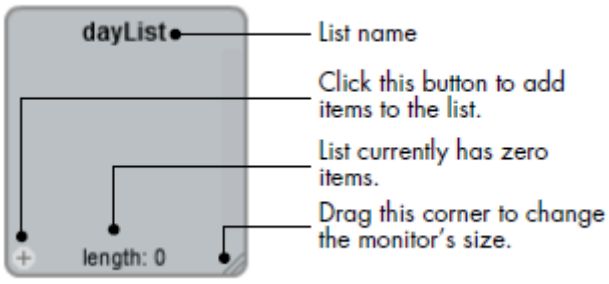
El proceso de crear una lista es casi idéntico al de creación de variables. Selecciona la categoría "datos" y pincha en el botón "crear una lista" para acceder a la ventana de diálogo de la figura. A continuación, escribe el nombre de la lista (nosotros usaremos "dayList") y especifica su ámbito ("para todos los objetos" (global) o "sólo para este objeto" (local)).



Al clicar en el botón "OK", Scratch crea una nueva lista vacía, y muestra los bloques relacionados con las listas, como ilustra la figura:



Al crear la lista, Scratch también muestra su monitor en el escenario, como muestra la figura. La lista estará inicialmente vacía, por lo que su longitud es inicialmente 0. Puedes usar este monitor para añadir elementos a la lista conforme escribes tu programa.



Si conocemos los datos que queremos guardar en la lista (como en el caso de nuestra lista "dayList"), podemos añadirlos en este punto del proceso de creación. La figura muestra cómo añadir días a la lista "dayList" usando su monitor. Pincha en el signo "más" en la esquina inferior izquierda siete veces para crear siete entradas, y a continuación, escribe un día de la semana dentro de cada entrada. Usa la tecla de tabulación (a la izquierda de la tecla de la Q) para navegar a través de los elementos de la lista. Para

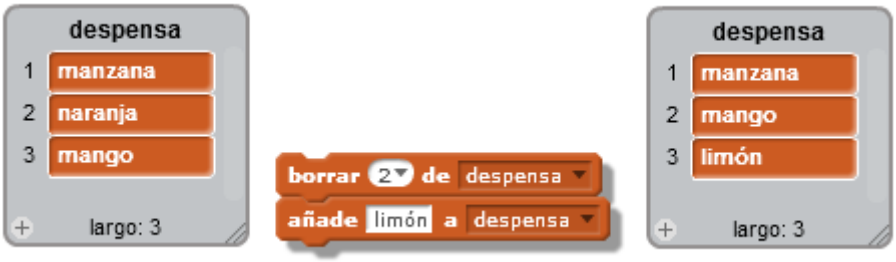
seleccionar un elemento de la lista, basta con clicar sobre él. Al presionar el botón del signo más con un elemento de la lista seleccionado, el nuevo elemento de la lista se añadirá a continuación del elemento actual; si no hay ningún elemento seleccionado, el nuevo elemento se añade al final de la lista. Cuando seleccionamos un elemento de la lista, junto a él aparece un botón con una cruz. Si queremos eliminar ese elemento de la lista, debemos presionar sobre el botón de la cruz del elemento a borrar. Prueba todo lo que hemos comentado en la lista que has creado.

COMANDOS DE LISTAS.

Una de las figuras previas contenía todos los bloques que añade Scratch al crear una nueva lista. En esta sección estudiaremos cómo funcionan esos bloques.

Añadir y borrar.

El comando "añade ( ) a ( )" agrega un nuevo elemento al final de una lista, mientras que el comando "borrar ( ) de ( )" elimina el elemento en la posición especificada.



El programa de la figura muestra estos dos comandos en acción. El programa ejecuta un comando "borrar" para eliminar el segundo elemento de la lista, que es "naranja". A continuación, el programa pone "limón" al final de la lista mediante el bloque "añade".

El comando "añade" es fácil de entender, así que vamos a examinar el comando "borrar" con un poco más de detalle. En el hueco para parámetros de este bloque podemos escribir directamente el índice del elemento que queremos borrar, pero también podemos seleccionar una de las opciones del menú desplegable que aparece al pinchar en la flecha dentro del hueco para parámetros (ver figura). Este menú desplegable nos permite borrar el primer elemento de la lista, el último elemento, o borrar todos los elementos de la lista.

Insertar y reemplazar.

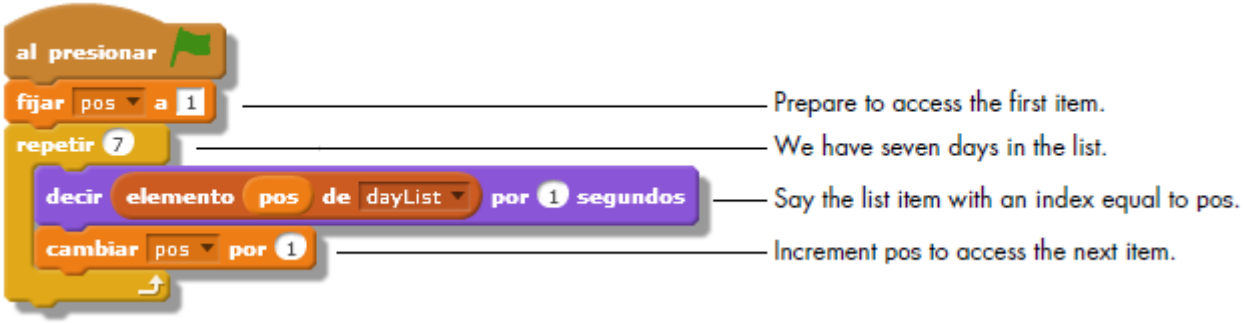
Imagina que queremos guardar alfabéticamente los nombres y números de teléfono de nuestros amigos en una lista, como los contactos de nuestro teléfono móvil. Conforme vamos creando la lista, debemos insertar la información de contacto de cada amigo en la posición adecuada. Pero con posterioridad, si un amigo cambia de número, necesitamos editar la lista para cambiar esa información. Los comandos "insertar" y "reemplazar" son útiles para estas tareas. La figura muestra un ejemplo de utilización de estos comandos:

El comando "reemplazar" sobrescribe el elemento actual en la posición indicada (3 en el ejemplo) con la información especificada (el nuevo número de Ana). El comando "insertar" ubica un nuevo elemento (en el ejemplo, el teléfono de Blas) en la posición indicada (4 en el ejemplo). Notar que al insertar el contacto Blas en la posición 4, los elementos existentes se mueven hacia abajo 1 posición para hacer sitio a la nueva entrada.

Si clicamos en la flecha dentro del hueco para parámetros de los bloques "reemplazar" e "insertar" podremos acceder a un menú desplegable con tres opciones: 1, último, y al azar (ver figura). Si elegimos "al azar", el comando elegirá un índice aleatoriamente. Más adelante veremos algunas aplicaciones útiles de esta opción.

Acceder a los elementos de una lista.

Ya hemos mencionado antes que podemos acceder a cualquier elemento de una lista usando el índice de ese elemento. Por ejemplo, el programa de la figura muestra cómo usar el bloque "elemento ( ) de ( )" para acceder a los elementos de la lista "dayList". Este programa usa una variable llamada "pos" (de posición) para iterar a lo largo de los elementos de la lista, mostrando los contenidos mediante el bloque "decir".



El programa inicializa el valor de "pos" a 1 para acceder al primer elemento de "dayList", y a continuación, entra en un bucle. El bucle se repite 7 veces, el número de elementos en nuestra lista. A cada pasada, el

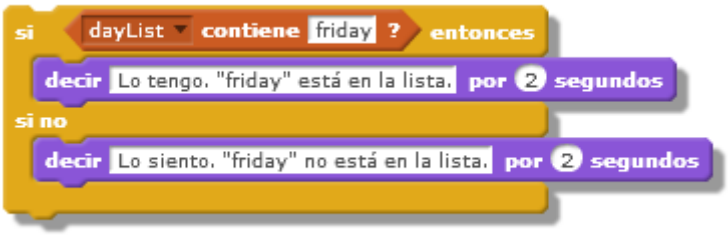
bucle dice el elemento de la lista con un índice igual a "pos", e incrementa el valor de "pos" en 1 unidad para acceder al siguiente elemento.

EJERCICIO 52. LONGITUD DE UNA LISTA.

En el ejemplo previo, borra el número 7 del bucle "repetir (7)" y en su lugar, usa el bloque "longitud de (dayList)". Esto es lo que normalmente haremos para recorrer los elementos de una lista cuando no sabemos cuántos elementos contiene. Además, usa la opción "al azar" del menú desplegable del bloque "elemento ( ) de ( )", y observa qué ocurre.

El bloque "¿( ) contiene ( )?".

Usando el bloque "¿( ) contiene ( )?" podemos comprobar si una lista contiene o no una cierta cadena. Se trata de un bloque booleano que devuelve VERDADERO o FALSO dependiendo de si la lista contiene o no la cadena especificada. El programa de la figura ilustra una aplicación de este bloque. Como "dayList" contiene la cadena "friday", se ejecutará el comando "decir" dentro del bloque "si".



NOTA: El bloque "¿contiene?" no distingue entre mayúsculas y minúsculas. Por ejemplo, el bloque "¿(dayList) contiene (friDAY)?" también se evaluaría a VERDADERO.

LOS LÍMITES DE UNA LISTA.

Los bloques de listas "borrar", "insertar", "reemplazar", y "elemento de" requieren un parámetro de entrada que especifique el índice del elemento al que queremos acceder. Por ejemplo, para borrar el séptimo elemento de la lista "dayList", usamos el bloque "borrar (7) de (dayList)". ¿Pero qué crees que ocurrirá si usamos un índice inválido en uno de estos bloques? Por ejemplo, ¿cómo crees que respondería Scratch si le pidiésemos borrar el octavo elemento de la lista "dayList" (que solo contiene 7 elementos)?

Comando o bloque de función.	Resultado.
elemento 8 de dayList	Devuelve una cadena vacía porque "dayList" solo tiene siete elementos. Lo mismo ocurre si usamos un índice menor que 1
elemento 1.9 de dayList	Scratch ignora el 9 decimal y devuelve el primer elemento de "dayList", que es "sunday". Algo similar ocurriría si pidiésemos el elemento 5.3; Scratch devolvería el quinto elemento, "thursday".
insertar newDay en 10 de dayList	Scratch ignora este comando porque intenta crear un hueco en la lista. La lista queda inalterada.
insertar newDay en 8 de dayList	Esto tiene el mismo efecto que el comando "añade". Añade "newDay" al final de la lista.
borrar 8 de dayList	Este comando se ignora (porque "dayList" solo tiene siete elementos), y la lista queda inalterada.

Técnicamente, es un error intentar acceder a un elemento más allá de los límites de una lista. Sin embargo, y en lugar de mostrar un mensaje de error, Scratch intenta hacer algo coherente con el bloque infractor.

(Por esta razón, la ausencia de mensajes de error en un programa no significa que no haya un error). Scratch no se quejará de que estemos intentando acceder a un índice inválido; en vez de ello, el programa simplemente no se comportará de la manera esperada. La tabla de la figura muestra lo que puede ocurrir cuando intentamos acceder a "dayList" usando un índice fuera del rango del 1 al 7.

Los ejemplos de la tabla muestran que, aunque los bloques de Scratch siempre hacen algo sensato cuando sus entradas son inválidas, puede que no hagan lo que a nuestro programa le interesaría que hubiesen hecho. Debemos asegurarnos de que nuestros programas proporcionan las entradas correctas a dichos bloques para que funcionen tal y como queremos.

9.2. LISTAS DINÁMICAS.

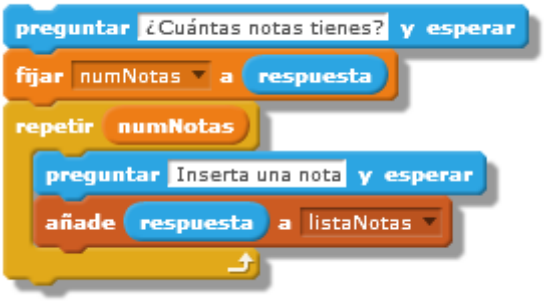
Los ejemplos que hemos visto hasta ahora usaban listas que se habían creado manualmente usando sus monitores. ¿Pero qué pasa si no conocemos cuál será el contenido de una lista al escribir nuestro programa? Por ejemplo, podríamos necesitar crear una lista con los números que vaya introduciendo un usuario, o llenar una lista con valores aleatorios cada vez que se ejecuta un programa. Éste es el objetivo de la presente sección.

Las listas son muy útiles porque pueden crecer o menguar dinámicamente conforme se está ejecutando un programa. A modo de ejemplo, digamos que estamos escribiendo una aplicación en la que el profesor introduce las notas de los alumnos para poder procesarlas. (El profesor podría necesitar la nota máxima, la mínima, el promedio, la mediana, y cosas por el estilo). Sin embargo, el número de alumnos puede ser distinto en cada clase. El profesor podría necesitar 20 notas para la clase 1, 25 notas para la clase 2, etc. ¿Cómo sabe nuestro programa que el profesor ha terminado de introducir notas? En esta sección también responderemos a esta pregunta.

En primer lugar, vamos a mostrar dos formas de rellenar listas con los datos proporcionados por un usuario. Después estudiaremos las listas numéricas y veremos algunas operaciones comunes que se pueden hacer con ellas. Una vez entendamos estos conceptos básicos, estaremos listos para usar estas técnicas en nuestras propias aplicaciones.

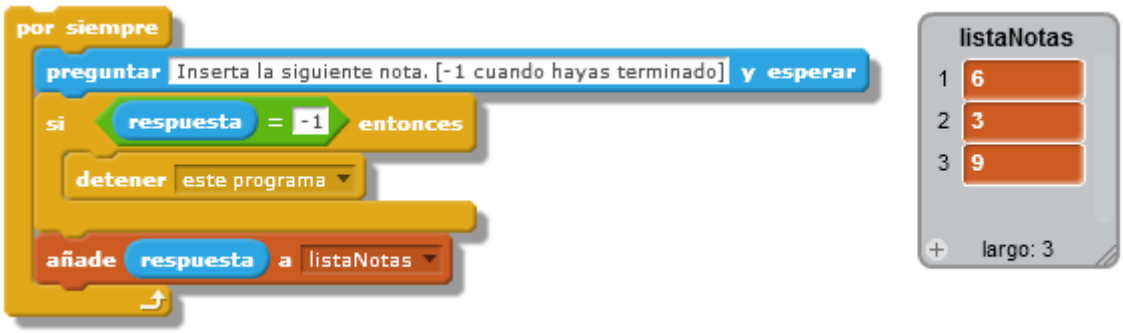
RELLENAR LISTAS CON LOS DATOS DE UN USUARIO.

Hay dos formas con las que podemos rellenar una lista con los datos introducidos por un usuario. En el primer método, el programa comienza preguntando cuántas entradas querrá introducir el usuario, y a continuación, arranca un bucle para ir recopilando los datos que va proporcionando el usuario. La figura muestra un programa que ilustra esta técnica:



La segunda forma de rellenar automáticamente una lista es hacer que el usuario introduzca un valor especial (conocido como **centinela**) para indicar el final de la lista. Por supuesto, el centinela debe elegirse de forma que no coincida con uno de los elementos posibles de la lista. Por ejemplo, si nuestra lista va a almacenar nombres o números positivos, una buena elección para el centinela sería el valor -1. Ahora bien, si el usuario puede introducir números negativos, el valor -1 no sería un centinela adecuado. Para nuestra

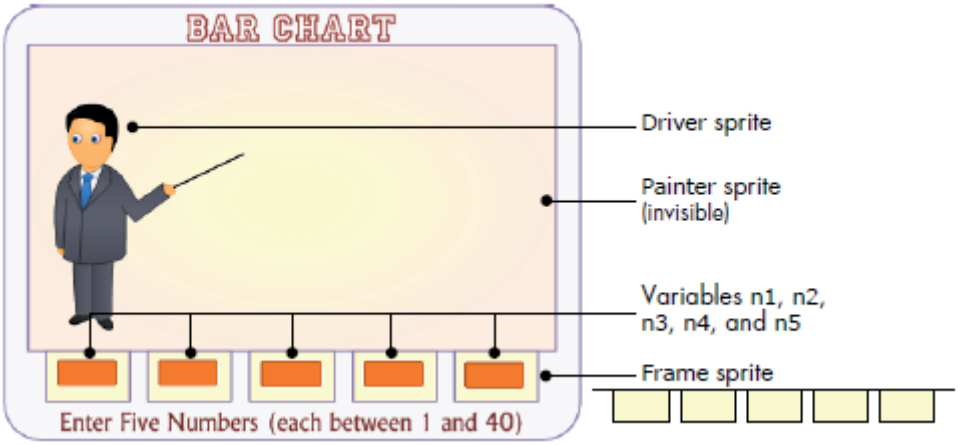
lista de notas, un centinela de -1 funcionaría perfectamente. El programa de la figura usa este centinela para saber cuándo el usuario termina de introducir valores.



Notar que el programa especifica en la pregunta al usuario cuál es el centinela (-1 en este caso). Si el usuario introduce el valor -1, el programa se para, porque sabe que el usuario ha terminado de introducir notas (y ese valor no se incluye en la lista). En caso contrario, el valor introducido se adjunta a la lista, y al usuario se le solicita otra entrada. La figura muestra cómo quedaría "listaNotas" si el usuario introdujese tres notas (6, 3, y 9) seguidas del centinela (-1).

CREAR UN DIAGRAMA DE BARRAS.

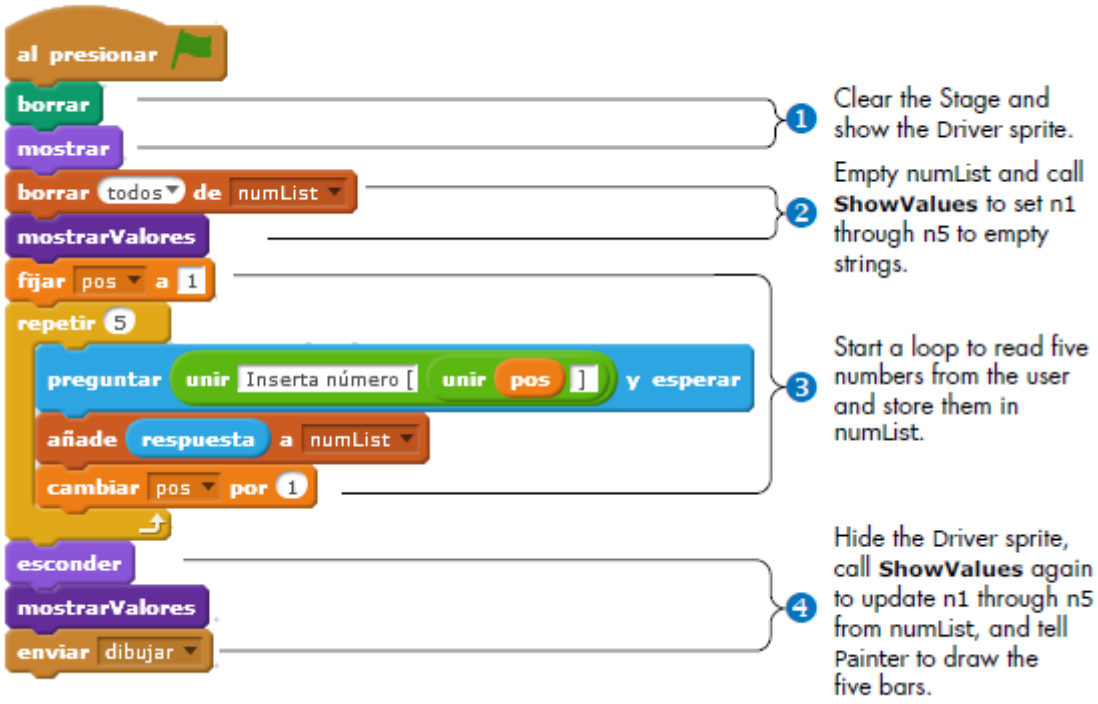
Como ejemplo práctico de la recopilación de los datos de un usuario en una lista, vamos a escribir una aplicación que dibuje un gráfico de barras (también llamado *histograma*) a partir de los números que introduzca el usuario. Por simplicidad, nuestra aplicación solo aceptará cinco números entre 1 y 40. Una vez el programa haya recibido los cinco números, dibujará cinco barras con alturas proporcionales a los valores introducidos. La interfaz de usuario está disponible en el archivo **diagrama de barras_sinCodigo.sb2**.



La aplicación contiene tres objetos. El objeto "controlador" (driver) gobierna el flujo de la aplicación; contiene los programas para aceptar los datos introducidos por el usuario, rellenar la lista, y decirle al objeto "dibujante" (painter) que comience a dibujar las barras. El objeto "dibujante" es un objeto invisible que se encarga de dibujar el diagrama de barras. El "marco" (frame) es un objeto meramente cosmético; se ocupa de ocultar las bases de las barras para hacerlas parecer planas (sin este objeto, las bases de las barras verticales tendrían esquinas redondeadas). Los valores numéricos de las cinco barras se muestran por pantalla mediante cinco variables, llamadas "n1", "n2", …, "n5", cuyos monitores están ubicados en las posiciones adecuadas sobre el escenario. Al pinchar en el icono de la bandera verde para arrancar la aplicación, el objeto "controlador" ejecuta el programa mostrado en la figura.

(1) En primer lugar, el objeto "controlador" aparece en el escenario y limpia las marcas de lápiz previas. De esta forma, si ya hay un diagrama de barras en pantalla, el objeto lo limpia antes de dibujar el nuevo diagrama. (2) A continuación, el programa borra el contenido de la lista "numList" para que pueda almacenar

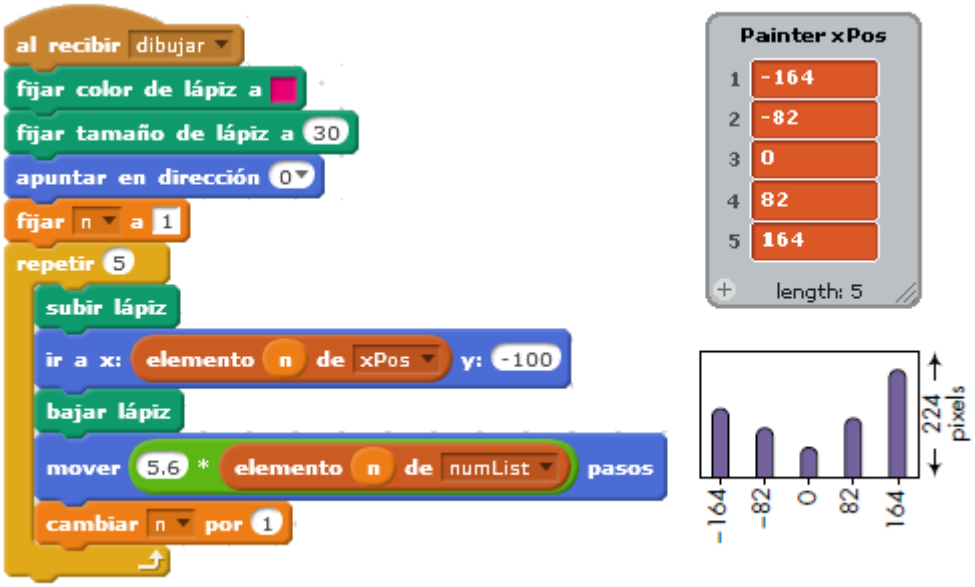
los nuevos valores introducidos por el usuario, y llama al procedimiento "mostrarValores" para fijar los valores de "n1" a "n5" de forma que sus monitores queden en blanco. (3) Cuando el escenario está preparado, el programa entra en un bucle "repetir ( )", que itera cinco veces. Dentro del bucle, el "controlador" le pide al usuario que introduzca un número, y lo añade a "numList". (4) Después de recopilar los cinco números del usuario y guardarlos en la lista, el objeto "controlador" se oculta para dejarle sitio al diagrama de barras. Después llama de nuevo al procedimiento "mostrarValores" para actualizar las variables de "n1" a "n5" con los nuevos valores introducidos por el usuario, y envía el mensaje "dibujar" para que el objeto "dibujante" dibuje las cinco barras.



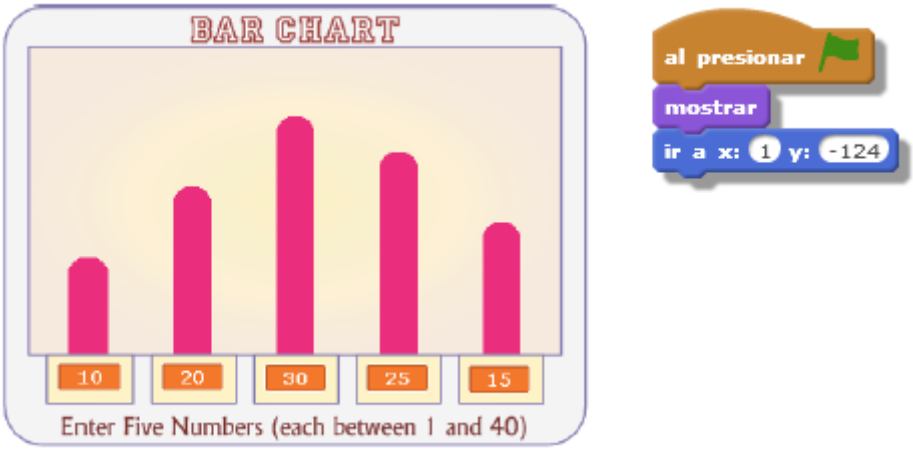
El procedimiento "mostrarValores" se muestra en la figura. Este procedimiento simplemente ajusta los valores de las variables "n1" a "n5" a las entradas correspondientes en la lista "numList". Como la primera llamada a "mostrarValores" se hace inmediatamente después de limpiar "numList", las cinco variables contendrán cadenas vacías. Esto implicará la limpieza de los cinco monitores, que es exactamente lo que queríamos hacer. Cuando "numList" contiene los datos del usuario, la llamada a "mostrarValores" permitirá mostrar esos datos en los monitores.

Ahora vamos a analizar el programa del objeto "dibujante", que se ejecuta cuando este objeto recibe el mensaje "dibujar". La figura muestra este programa. En primer lugar, el objeto ajusta el color del lápiz. A continuación, fija el tamaño del lápiz a un valor grande para dibujar las barras gruesas, y pasa a apuntar en la dirección hacia arriba para dibujar las cinco barras verticales. Posteriormente, el programa arranca un bucle para dibujar las cinco barras. Como conocemos de antemano la posición en x de cada una de las barras (ver figura), hemos creado una lista llamada "xPos" para almacenar esos valores. En cada iteración del

bucle, el objeto "dibujante" se mueve a la posición x de la barra actual, baja el lápiz, y se mueve hacia arriba para dibujar una línea vertical.



La altura de cada línea es proporcional al valor de "numList" correspondiente. El área de nuestro diagrama en pantalla ocupa 224 píxeles de alto, y como 40 es el valor máximo, una entrada de 40 debería tener una barra igual de alta que el área del gráfico (224 píxeles). Para hallar la altura en píxeles de un número cualquiera en "numList", debemos multiplicar ese número por $5{,}6$ (esto es, por $224/40$). La figura muestra la salida de la aplicación después de que el usuario haya introducido unos cuantos datos. Notar que el objeto "marco" cubre las bases redondeadas de los gruesos trazos verticales para que las barras parezcan planas en su parte baja. La figura también muestra el programa del objeto "marco".



Al acabar, guarda el archivo como **diagrama de barras.sb2**. Ejecuta la aplicación varias veces para entender cómo funciona.

EJERCICIO 53. BARRAS DE COLOR.

Cambia la aplicación previa para que cada barra del diagrama se dibuje con un color diferente. PISTA: crea una nueva lista, llamada "color", para que el objeto dibujante guarde el número del color de las cinco barras, y use el comando mostrado en la figura antes de dibujar cada barra:

9.3. LISTAS NUMÉRICAS.

Las listas de números son muy comunes en muchas aplicaciones. Por ejemplo, podemos tener listas de notas en exámenes, de medidas de temperaturas, de precios de productos, y más. En esta sección exploraremos algunas de las operaciones más comunes que podríamos necesitar realizar sobre una lista numérica. En particular, escribiremos procedimientos para hallar el valor máximo, el valor mínimo, y el valor promedio de los números almacenados en una lista.

HALLAR EL MÁXIMO Y EL MÍNIMO.

Imagina que eres un profesor y que necesitas saber cuál es la nota máxima del último examen. Para ello, podríamos escribir un programa para comparar todas esas notas y hallar el valor máximo. El programa de la figura halla el valor máximo de una lista llamada "notas".

El procedimiento "hallarMáximo" comienza ajustando el valor de la variable "notaMax" igual al primer número de la lista. A continuación, inicia un bucle para comparar el resto de números en la lista con el valor actual de "notaMax". Cada vez que encuentra un número mayor que "notaMax", actualiza el valor de "notaMax" a ese número. Cuando el bucle termina, el valor guardado en "notaMax" será el número más grande contenido en la lista.

Para hallar el valor mínimo de una lista podemos usar un algoritmo muy similar. Comenzamos asumiendo que el primer elemento de la lista es el elemento más pequeño, y a continuación, usamos un bucle para comprobar el resto de elementos. Cada vez que encontremos un valor más pequeño, actualizamos la variable que almacena el valor mínimo.

EJERCICIO 54. HALLAR EL MÍNIMO.

Escribe un procedimiento llamado "hallarMínimo" que nos permita determinar la nota mínima de la lista "notas".

HALLAR EL PROMEDIO.

En este ejemplo escribiremos un procedimiento para calcular la nota media de los números almacenados en la lista "notas". Podemos hallar el valor promedio de una secuencia de N números calculando primero su

suma y dividiendo después por el número total de números, N. El procedimiento "hallarPromedio" mostrado en la figura hace exactamente eso:

Este procedimiento usa un bucle para pasar por todas las notas almacenadas en la lista, las suma, y guarda el resultado en una variable llamada "suma". (Esta variable se inicializa a 0 antes de arrancar el bucle). Cuando el bucle termina, el programa calcula el valor promedio dividiendo "suma" entre el número de notas, y guarda el resultado en una variable llamada "promedio".

> NOTA: Presta especial atención a la forma en la que se va acumulando el valor de la variable "suma" dentro del bucle. Este patrón de programación, llamada *patrón de acumulación*, aparece de forma muy habitual en muchas aplicaciones.

EJERCICIO 55. PROCESADO DE NOTAS.

Combina "hallarMáximo", "hallarMínimo", y "hallarPromedio" en un solo procedimiento llamado "procesarNotas", que mostrará al mismo tiempo las notas media, máxima, y mínima de la lista "notas".

9.4. BÚSQUEDA Y CLASIFICACIÓN EN LISTAS.

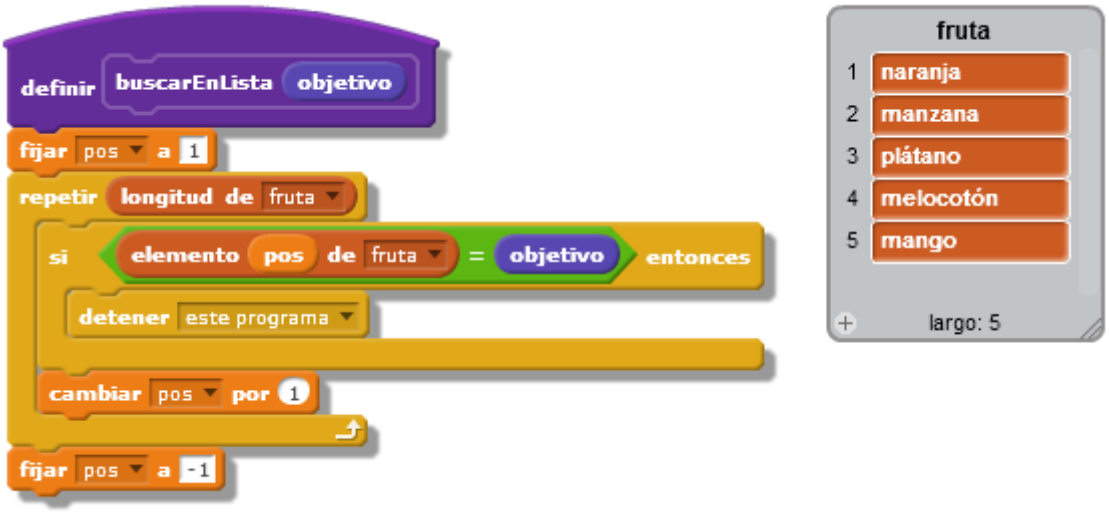
Suponer que tenemos una lista de contactos que no está ordenada. Si quisiéramos organizar los contactos, podríamos clasificarlos alfabéticamente en base a sus nombres. Y si quisiéremos consultar el número de teléfono de uno de nuestros contactos a partir de su nombre, deberíamos buscar en la lista ese nombre. El objetivo de esta sección es presentar las técnicas básicas de programación para buscar y clasificar los elementos de una lista.

BÚSQUEDA LINEAL.

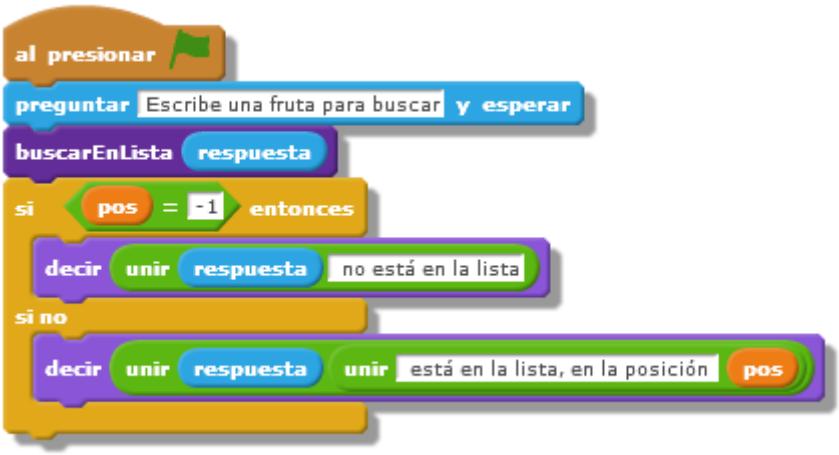
El bloque "¿( ) contiene ( )?" proporciona una forma muy sencilla de comprobar si una lista contiene un elemento específico. Pero si también queremos saber la posición en la lista donde se encuentra el elemento buscado, tenemos que programar la búsqueda nosotros mismos.

Esta sección explica un método de búsqueda en listas, llamado **búsqueda lineal** (o **búsqueda secuencial**). Este método de búsqueda funciona en cualquier lista, esté o no ordenada. Sin embargo, como la búsqueda lineal compara el valor buscado con cada elemento de la lista, puede tardar mucho tiempo en responder si la lista es grande.

Para ilustrar este método de búsqueda, suponer que estamos buscando un elemento específico en una lista llamada "fruta" (ver figura). Si la lista contiene el elemento buscado, también queremos saber la posición exacta de ese elemento en la lista. El procedimiento "buscarEnLista" de la figura realiza una búsqueda lineal para darnos la respuesta.



Comenzando con el primer elemento, "buscarEnLista" compara una a una las frutas de nuestra lista con aquella que estamos buscando (la cual está representada por el parámetro "objetivo"). El procedimiento se detiene tanto si encuentra el valor buscado como si llega al final de la lista sin encontrarlo. Si el programa encuentra el valor que buscamos, la variable "pos" contendrá la localización donde se encontró a ese elemento. En caso contrario, el procedimiento ajusta "pos" a un valor inválido (-1 en este caso) para indicar que el elemento buscado no está en la lista. La siguiente figura muestra un ejemplo de programa principal para llamar a este procedimiento.

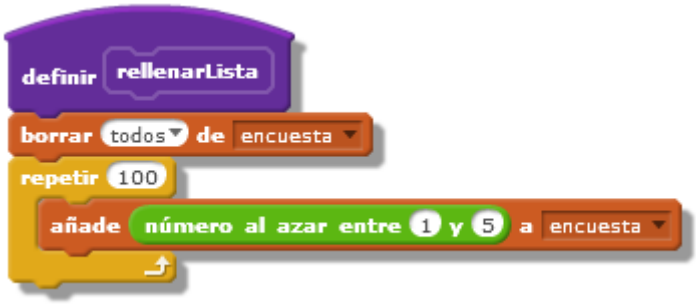


Notar que el valor de "pos" le dice al programa principal dos cosas: (a) Si el elemento buscado está o no en la lista, y (b) cuando el elemento está en la lista, cuál es su posición exacta.

FRECUENCIA DE OCURRENCIA.

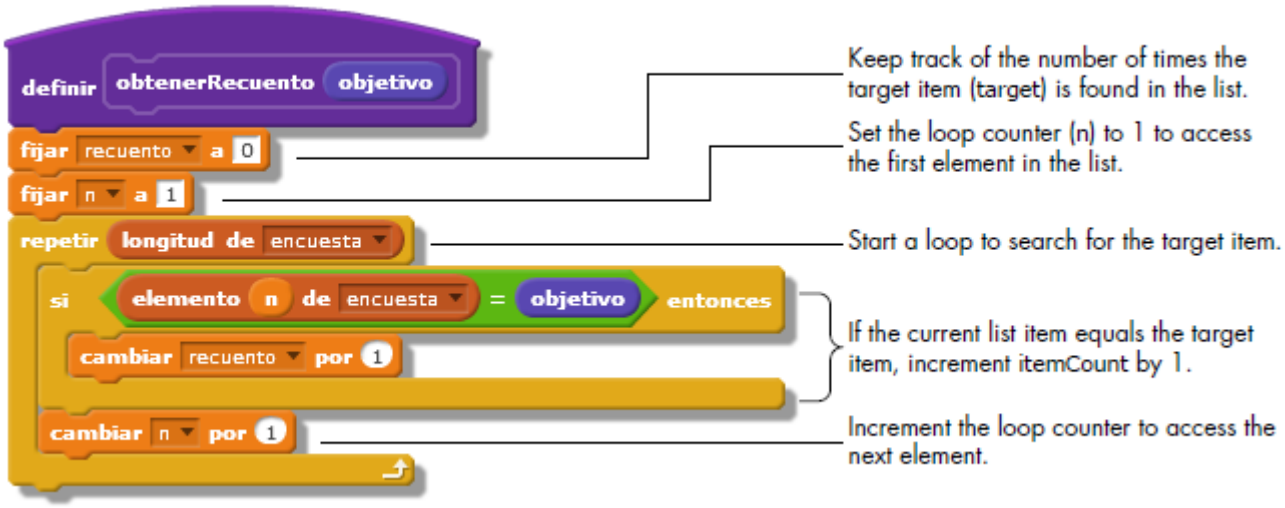
Supón que en el instituto se lleva a cabo una encuesta sobre la calidad de la comida de la cafetería. Los estudiantes del instituto evalúan la calidad mediante una nota del 1 al 5 (siendo 1 = pésima, 2 = mala, 3 = regular, 4 = buena, 5 = excelente). Los votos se introducen en una lista, y se nos pide un programa que procese estos datos. Por ahora, el centro solo quiere saber cuántos estudiantes valoran la calidad de la comida como pésima (esto es, cuántos estudiantes le dieron una nota de 1).

Evidentemente, nuestro programa necesita un procedimiento que cuente cuántas veces aparece el valor 1 en la lista. Para simular los votos de los alumnos, usaremos una lista que contenga 100 votos aleatorios. Ello lo haremos con el procedimiento "rellenarLista" mostrado en la figura. Este procedimiento añade 100 números aleatorios entre 1 y 5 en una lista llamada "encuesta".

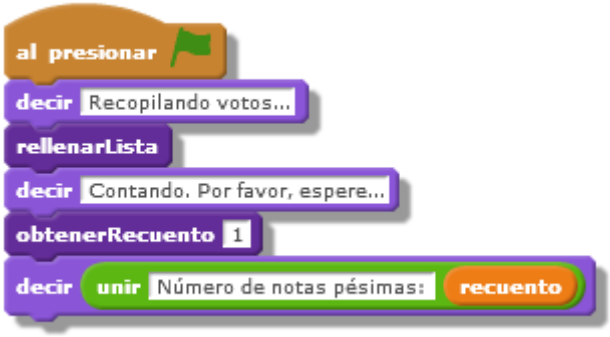


Ahora que tenemos una lista de votos, podemos contar la frecuencia con la que una cierta nota aparece en la lista. Eso lo haremos con el procedimiento "obtenerRecuento ( )" mostrado en la figura.

El parámetro "objetivo" representa el elemento que estamos buscando, y la variable "recuento" almacena el número de veces que encontramos ese elemento en la lista. El procedimiento comienza fijando "recuento" a 0, y a continuación arranca un bucle "repetir ( )" para buscar en la lista el valor especificado en "objetivo". A cada iteración del bucle, el procedimiento comprueba el elemento en la localización indexada por el contador de bucle "n". Si ese elemento es igual al valor de "objetivo", el programa incrementa en 1 el valor de "recuento".



El programa principal que llama al resto de procedimientos sería similar al siguiente:



EJERCICIO 56. ESTUDIO COMPLETO DE LA CAFETERÍA.

Después de estudiar el número de notas pésimas, el director nos pide que comprobemos los resultados de todas las notas (esto es, el número de notas pésimas, malas, regulares, buenas, y excelentes). El director

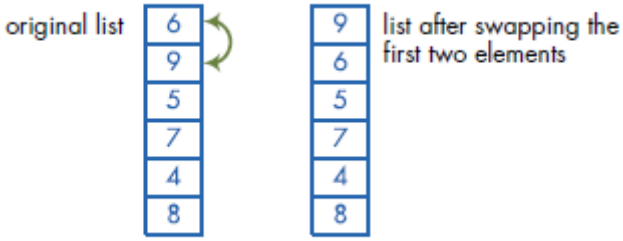
también quiere saber la nota media que se le ha dado al servicio de la cafetería, Modifica el programa previo para darle al director la información solicitada.

ORDENAMIENTO DE BURBUJA.

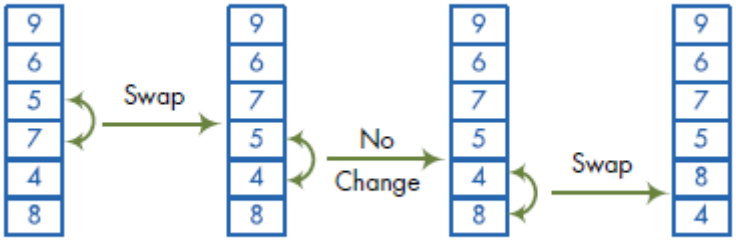
Si tenemos un conjunto de nombres, puntuaciones en un videojuego, o cualquier otra cosa que queramos mostrar en un orden particular (alfabéticamente, del mayor al menor, etc.), deberemos ordenar la lista que lo contiene. Hay muchas formas de ordenar listas, y el ordenamiento de burbuja es uno de los algoritmos más sencillos. (Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas").[2] En esta sección aprenderemos cómo funciona el ordenamiento de burbuja y escribiremos un programa de Scratch que lo implemente.

Suponer que queremos ordenar la lista de números [6 9 5 7 4 8] en orden descendente. Esta secuencia de pasos ilustra cómo funciona el algoritmo de ordenamiento de burbuja:

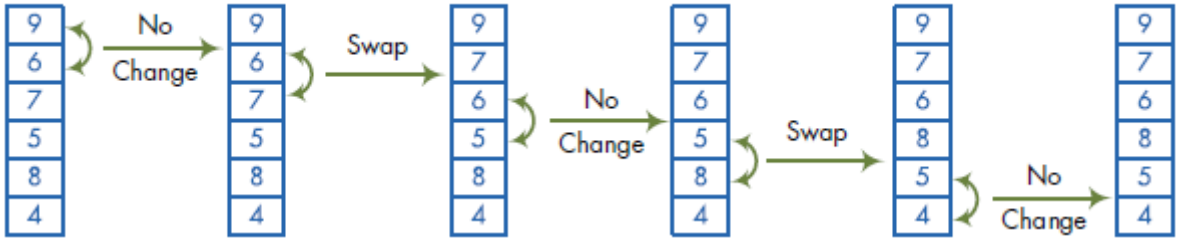
1) Comenzamos comparando los dos primeros elementos de la lista. Como 9 es mayor que 6, podemos intercambiar sus posiciones (ver figura).



2) Ahora podemos comprar el segundo y tercer elementos, que son 6 y 5. Como 6 es mayor que 5, estos dos números ya están ordenador, y podemos seguir con la siguiente pareja.
3) Repetimos este proceso para comparar el tercer y cuarto elementos, el cuarto y el quinto, y finalmente el quinto y el sexto. La figura muestra cómo queda la lista tras estas tres comparaciones:

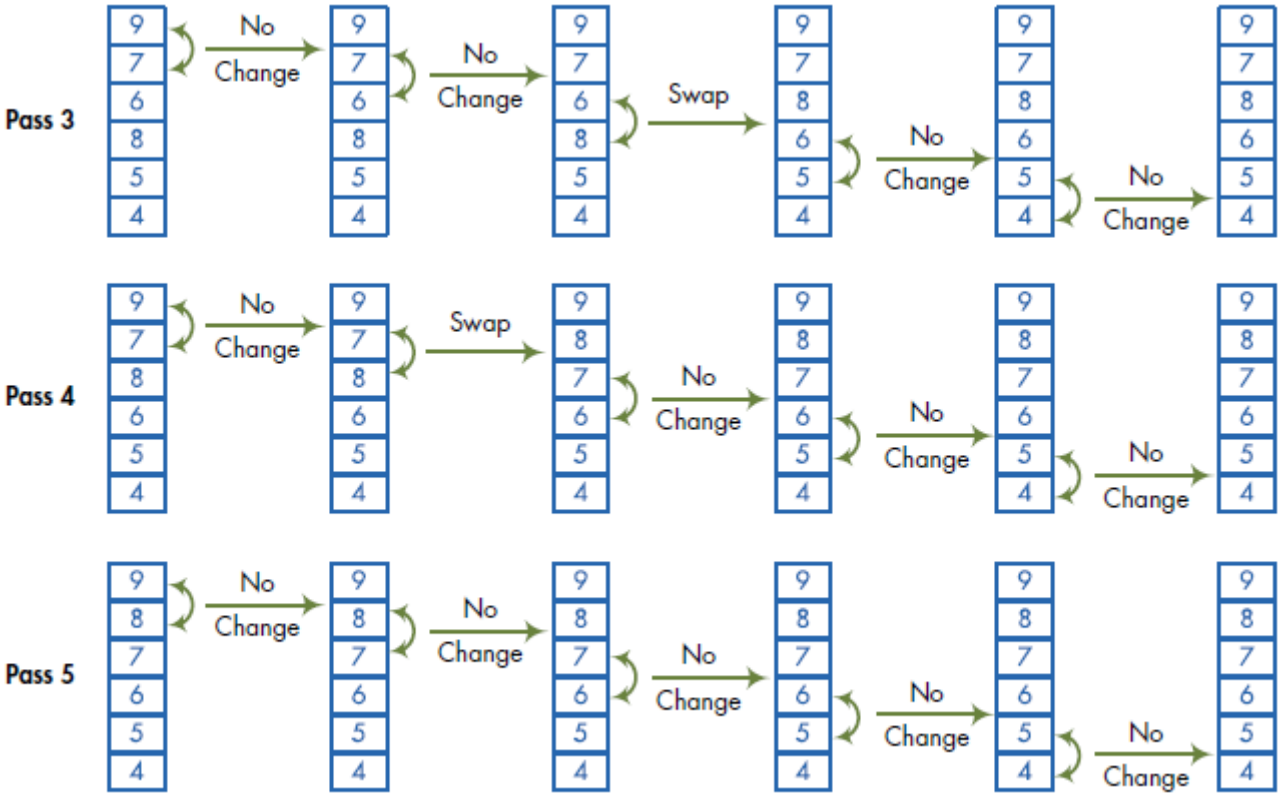


4) Con esto finaliza la primera pasada del ordenamiento de burbuja, pero la lista aún no está ordenada. Necesitamos efectuar una segunda pasada, comenzando con el paso 1. Una vez más, comparamos cada pareja de elementos adyacentes, y los intercambiamos si es necesario. Tras la segunda pasada, la lista queda como muestra la figura:

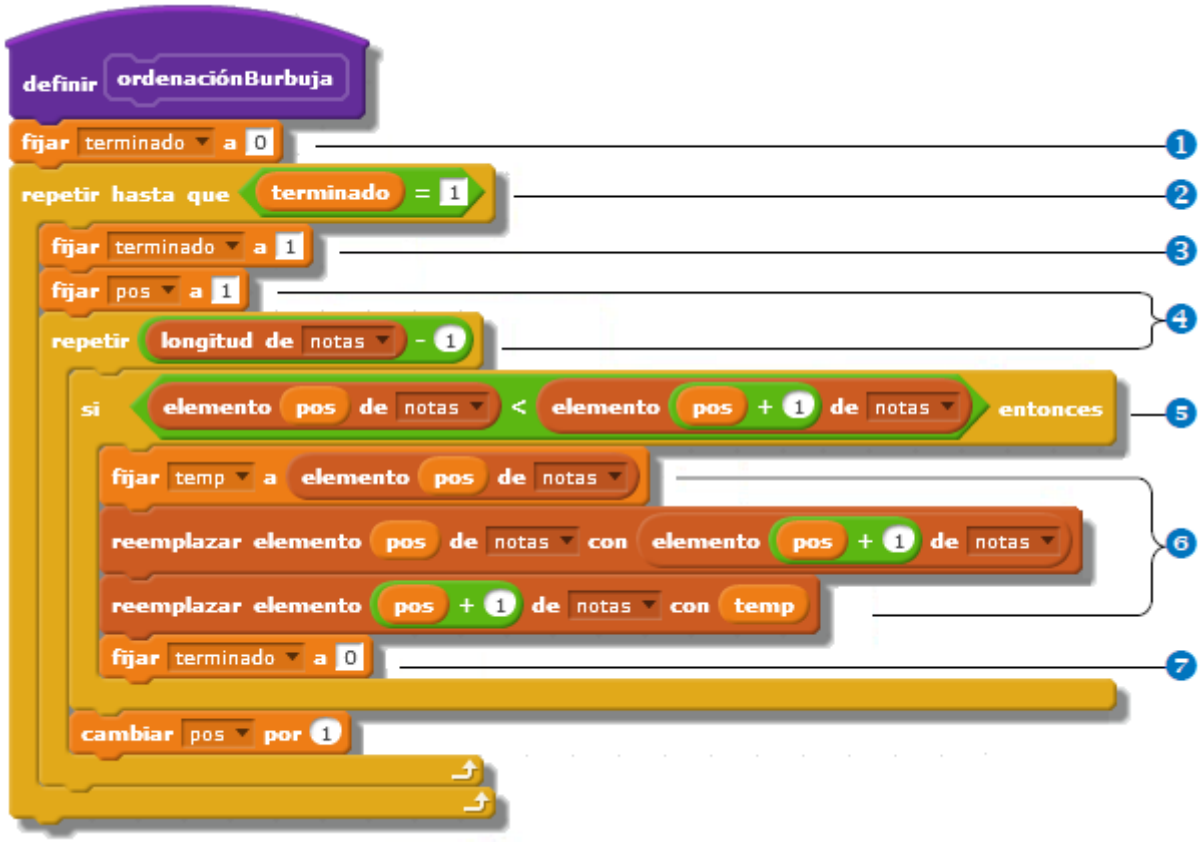


[2] https://es.wikipedia.org/wiki/Ordenamiento_de_burbuja

5) Repetimos el proceso de ordenamiento de burbuja hasta que ya no haya intercambio de números durante una pasada, lo que significa que la lista ya está ordenada. La figura muestra las tres últimos pasadas del algoritmo:



Ahora que ya hemos vistos cómo funciona el algoritmo de ordenamiento de burbuja, vamos a implementarlo en Scratch:



Como muestra la figura, el programa tiene dos bucles. El bucle interno recorre cíclicamente la lista, comparando parejas de elementos e intercambiándolos cuando es necesario. Además, fija el valor de un indicador (llamado "terminado") a 0 cuando se necesita otra pasada. El bucle externo se repite hasta que

"terminado" deja de der 0 y toma el valor 1, porque un valor 0 significa que no aún hemos acabado de ordenar. Si el bucle interno completa una pasada sin intercambiar elementos, el bucle externo saldrá, terminando el procedimiento.

Vamos a examinar el procedimiento con mayor detalle. (1) Como todavía no hemos hecho el ordenamiento, el procedimiento comienza fijando el valor de "terminado" a 0. (2) El bucle externo usa un bloque "repetir hasta que ( )" para recorrer la lista hasta que quede ordenada (esto es, hasta que "terminado" se vuelva 1). (3) Al principio de cada iteración, este bucle fija "terminado" a 1 (es decir, asume que no hemos hecho ningún intercambio de elementos), y también fija la variable "pos" a 1 para comenzar el ordenamiento con el primer número.

(4) A continuación, el bucle interno compara cada pareja de elementos adyacentes en la lista. El bucle necesita realizar $N-1$ comparaciones, donde N es el número de elementos de la lista. (5) Si el elemento en el índice "pos" $+1$ es mayor que el elemento en el índice "pos", estos dos elementos deben intercambiar posiciones en la lista. En caso contrario, el procedimiento suma 1 unidad a "pos" para comparar la siguiente pareja de elementos. (6) Si hay que intercambiar los dos elementos, el procedimiento los intercambia con la ayuda de una variable temporal llamada "temp".

(7) Una vez que la pasada actual termina de recorrer la lista, el bucle interno vuelve a fijar "terminado" a 0 si intercambió alguna pareja de números, o deja "terminado" igual a 1 si no efectuó cambios. El bucle externo continuará hasta que la lista esté ordenada.

Prueba el programa varias veces hasta que entiendas cómo funciona. Para ello, crea una lista llamada "notas" con unas cuantas notas cualesquiera entre 0 y 10. A continuación, añade a la aplicación el procedimiento "desordenar" (que se encarga de desordenar la lista de notas una vez que "ordenaciónBurbujas" las ha ordenado, y el programa principal (que simplemente llama a los procedimiento "desordenar" y "ordenaciónBurbujas").

EJERCICIO 57. ORDENACIÓN DE NOMBRES.

Has una lista de nombres en lugar de una lista de números, y usa el programa previo de ordenamiento de burbuja para ordenar la lista. ¿Funciona el programa tal y como debería? Además, ¿qué cambios debes hacer en el procedimiento para que ordene la lista en orden ascendente?

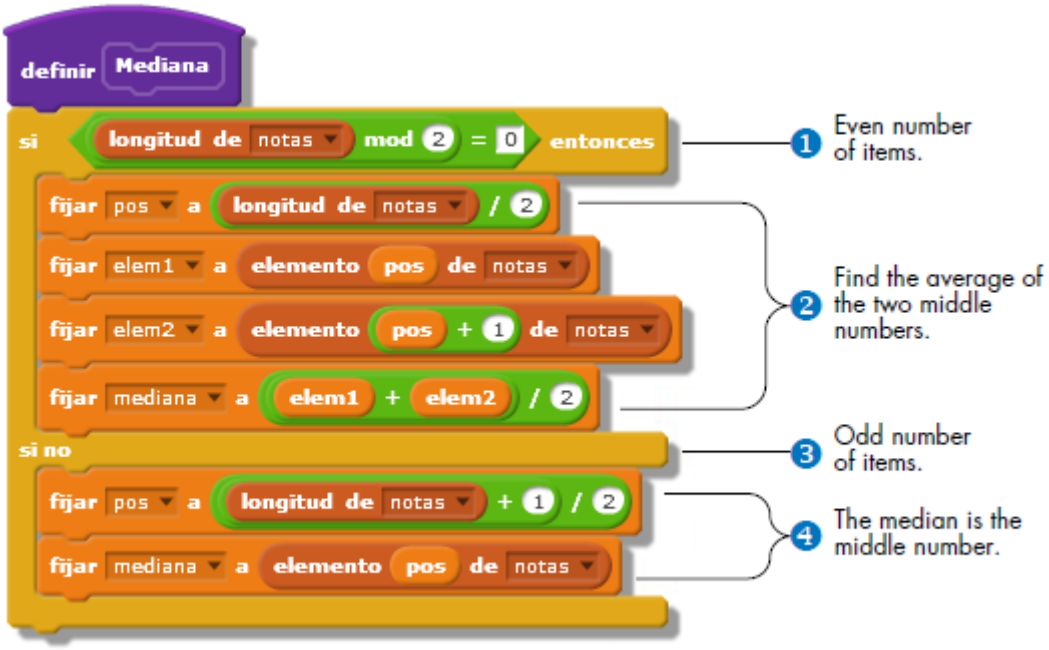
HALLAR LA MEDIANA.

Ahora que ya sabemos ordenar listas, podemos hallar fácilmente cuál es la mediana de una secuencia de números. La **mediana** es el valor intermedio de una secuencia de números ordenados. Si tenemos un número impar de elementos, la mediana es el número en la posición central de la secuencia. Si tenemos un número

par de elementos, la mediana es el promedio de los dos números centrales. Podemos describir cómo obtener la mediana de un conjunto ordenado de N elementos como sigue:

$$mediana = \begin{cases} elemento\ en\ el\ índice\ \dfrac{N+1}{2} & ,si\ N\ es\ impar \\ promedio\ de\ los\ elementos\ en\ \dfrac{N}{2}\ y\ \dfrac{N}{2}+1 & ,si\ N\ es\ par \end{cases}$$

La figura muestra el procedimiento que realiza este cálculo. Este procedimiento asume que la lista ya está ordenada.



El procedimiento usa un bloque "si / si no" para manejar los casos de listas pares e impares. (1) Si el número de elementos es divisible entre 2 (esto es, si la lista tiene un número par de elementos), (2) la variable "mediana" se calcula como el promedio de los dos términos centrales. (3) En caso contrario, las lista tiene un número impar de elementos, (4) y la variable "mediana" se ajusta igual al número en la posición central de la lista.

9.5. EJERCICIOS SCRATCH.

EJERCICIO 58. NÚMEROS PRIMOS.

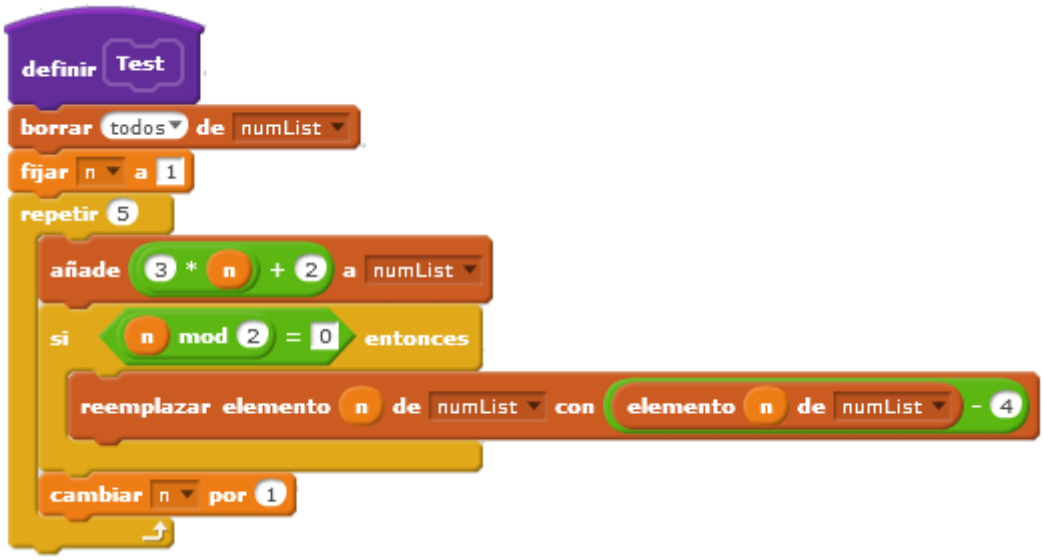
Crea una lista que contenga los 10 primeros números primos. Escribe un programa que muestre estos números usando el bloque "decir".

EJERCICIO 59. CONTACTOS.

Crea tres listas que guarden registros personales. La primera lista almacena nombres, la segunda fechas de nacimiento, y la tercera números de teléfono. Escribe un programa que le pida al usuario el nombre de la persona cuya información personal queremos obtener. Si el nombre de esa persona existe en la primera lista, el programa dirá su fecha de nacimiento y su número de teléfono.

EJERCICIO 60. VALOR ALMACENADO.

¿Cuál es el valor almacenado en "numList" después de ejecutar el programa de la figura? Modifica el programa para comprobar que tu respuesta es correcta.



EJERCICIO 61. DOBLE.

Escribe un programa que doble el valor de todos los elementos almacenados en una lista numérica.

EJERCICIO 62. ALUMNOS Y NOTAS.

Escribe un programa que le pida al usuario que introduzca los nombres de los estudiantes y sus notas y que almacene estos datos en dos listas. Deja de recopilar datos del usuario cuando éste introduzca el valor -1 para un nombre de estudiante.

EJERCICIO 63. TEMPERATURAS.

Escribe un programa que le pida al usuario introducir las temperaturas máxima y mínima de los 12 meses del año. Guarda estos datos en dos listas.

EJERCICIO 64. ELIMINA DUPLICADOS.

Escribe un programa que le pida al usuario 10 enteros. Guarda cada entero introducido en una lista, pero solo si no es un duplicado de un entero previamente introducido.

EJERCICIO 65. PROCESA NOTAS.

Escribe un programa que procese una lista de 50 notas del 0 al 10 (haz un procedimiento para rellenarla automáticamente), y que determine cuántos alumnos obtuvieron un notable (esto es, una nota entre 7 y 9, el 7 incluido pero el 9 no incluido).

9. 6. PROYECTOS SCRATCH.

PROYECTO 24. EL POETA.

Vamos a comenzar con un generador de poemas (en inglés). Este poeta elige palabras aleatoriamente de 5 listas ("artículo" (article), "adjetivo" (adjetive), "nombre" (noun), "verbo" (verb), y "preposición" (preposition)), y las combina según un esquema fijo. Para darles coherencia a nuestros poemas, todas las palabras de estas listas están relacionadas con el amor y la naturaleza. (Por supuesto, podríamos terminar con algún poema absurdo, pero no importa). Las listas y objetos necesarios están disponibles en el archivo **Proyecto 24_sinCodigo.sb2**.

Cada poema está compuesto de tres líneas que siguen las siguientes estructuras:

- Línea 1: artículo, adjetivo, nombre.
- Línea 2: artículo, nombre, verbo, proposición, artículo, adjetivo, nombre.
- Línea 3: adjetivo, adjetivo, nombre.

Con esta construcción en mente, vamos a comenzar con el procedimiento que construye la primera línea del poema:

Procedimiento "hacerLínea1" (poeta): Este procedimiento selecciona una palabra al azar de la lista "article" y la guarda en la variable "línea1". A continuación, el procedimiento adjunta a "línea1" un espacio en blanco, y una palabra al azar de la lista "adjetive". Después, adjunta a "línea1" otro espacio en blanco y una palabra aleatoria de la lista "noun". Por último, el objeto "poeta" dice la línea completa.

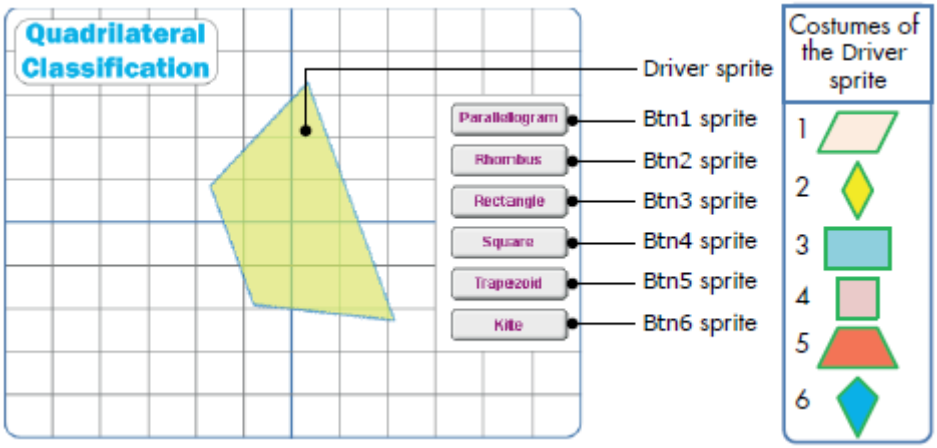
Los dos procedimientos restantes son similares a este. Constrúyelos. Aquí mostramos dos poemas creados por nuestra máquina generadora:

each glamorous road
a fish moves behind each white home
calm blue pond

every icy drop
a heart stares under every scary gate
shy quiet queen

PROYECTO 25. JUEGO DE CLASIFICACIÓN DE CUADRILÁTEROS.

El siguiente proyecto es un juego que explora los diferentes tipos de cuadriláteros (figuras geométricas de 4 lados). El juego muestra una de seis figuras por pantalla (paralelogramo, rombo, rectángulo, cuadrado, trapezoide, o cometa (kite)) y le pregunta al jugador que clasifique la forma pinchando en el botón correcto:



Abre el archivo **Proyecto 25_sinCodigo.sb2**. Como ves, el juego contiene siete objetos: seis para los botones de respuesta ("btn1" a "btn6") y un objeto "controlador" (driver) que contiene el programa principal. Como vemos en la figura, el objeto controlador tiene 6 disfraces que se corresponden con los seis cuadriláteros del juego.

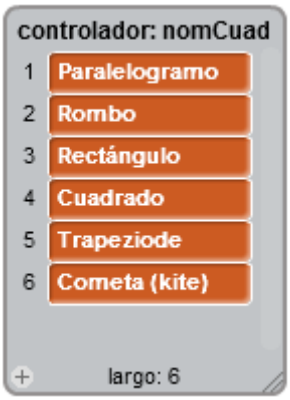
Programa 1 (controlador): Al pinchar en la bandera verde, el controlador va a la capa frontal para que no lo bloquee ningún botón. A continuación, arranca un bucle infinito, donde el programa llama al procedimiento "mostrarForma" para mostrar un cuadrilátero al azar, fija el valor de la variable "elección" a 0 (para indicar que el usuario aún no ha respondido), y queda a la espera (ver figura) hasta que "elección" deja de ser cero (lo que significa que el usuario ya ha pinchado en uno de los botones de respuesta). Cuando el usuario responde, y aun dentro del bucle, el programa llama al procedimiento "comprobarRespuesta" para decirle al jugador si ha acertado o no.

Procedimiento "mostrarForma" (controlador): Lo primero que hace este procedimiento es llevar al objeto "controlador" al centro del escenario, y hacerlo apuntar en una dirección aleatoria. A continuación, le asigna a la variable "forma" un valor aleatorio entre 1 y 6, y cambia el disfraz del objeto a ese valor aleatorio para mostrar el cuadrilátero que el usuario debe identificar. Para que el mallado del fondo siga siendo visible, el procedimiento establece el efecto "desvanecer" del objeto a un valor aleatorio entre 25 y 50. Para hacer que parezca que a cada ronda aparece una forma completamente nueva, el procedimiento también establece el efecto "color" del objeto a un valor aleatorio entre 25 y 75, y redimensiona el objeto a un 80%, 90%, … . o un 150% de su tamaño original, como muestra la figura. Con esto termina el procedimiento.

Veamos ahora los programas de cada uno de los botones de respuesta:

Programa 1 (btn1): Al pinchar sobre este objeto, el programa fija el valor de la variable "elección" a 1, para indicar que el usuario ha respondido que la figura por pantalla es un paralelogramo.

El resto de programas son idénticos, salvo por el valor que asignan a la variable "elección":



Estos programas simplemente fijan el valor de "elección" a un número diferente dependiendo de qué botón presione el usuario. Una vez "elección" contiene la respuesta del usuario, el procedimiento "comprobarRespuesta" del "controlador" puede comparar la respuesta del usuario (almacenada en "elección") con el valor de "forma", que especifica el tipo de cuadrilátero dibujado por pantalla.

Procedimiento "comprobarRespuesta" (controlador): El procedimiento comienza comprobando si el valor de "elección" es igual al valor de "forma". En ese caso, el controlador dice "Bien hecho!" durante 2 segundos.

En caso contrario, reproduce el sonido "alieCreak1", usa la variable "forma" como índice de la lista "nomCuad" (nombre del cuadrilátero) para obtener el nombre correcto de la forma mostrada por pantalla (valor que almacena en la variable "nombre"), y finalmente, cuál es la respuesta correcta (usando la variable "nombre").

Con esto termina el proyecto. Guarda el archivo como **Proyecto 25.sb2**.

AMPLIACIÓN: Tal y como está escrito, el juego continúa indefinidamente. Modifica la aplicación para añadirle al juego un criterio de parada (5 respuestas acertadas, 3 respuestas erróneas, o lo que tú consideres). Además, añade unos marcadores para visualizar y llevar la cuenta del número de aciertos y errores.

PROYECTO 26. EL HECHICERO MATEMÁTICO.

En este proyecto exploraremos cómo usar listas para guardar registros no uniformes (esto es, registros con tamaños diferentes), y cómo usar una lista como índice de otra. Un *registro* es simplemente una colección de datos relacionados sobre una persona, ligar, o cosa. En nuestro caso, cada registro consiste en una respuesta a un acertijo junto con las instrucciones para resolver ese acertijo. Mientras que cada acertijo tiene una sola respuesta válida, el número de instrucciones para resolver el acertijo varía de un acertijo a otro.

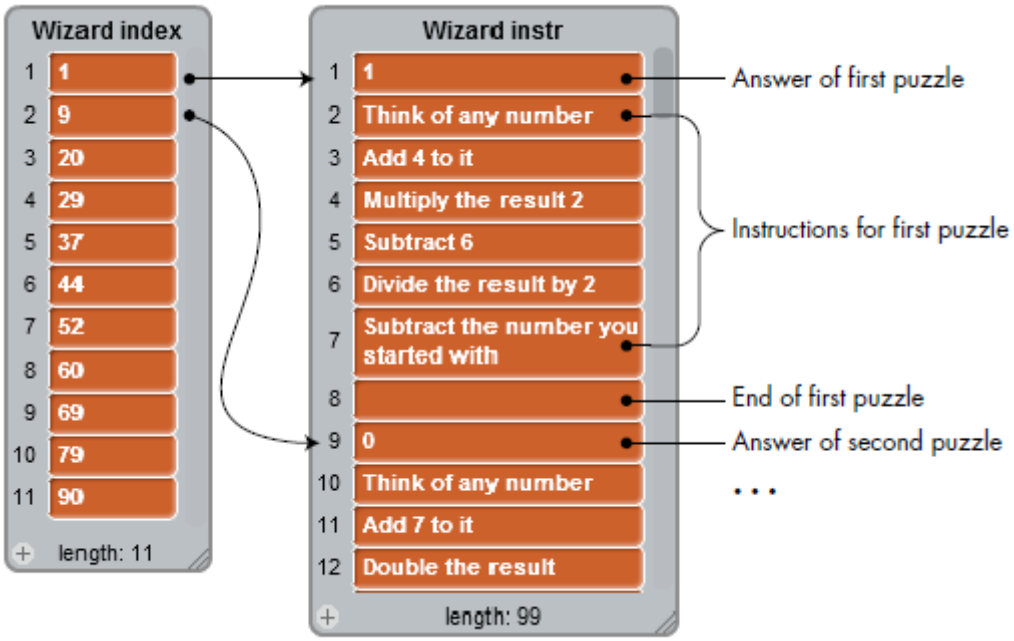
Nuestro hechicero matemático le pide al usuario que piense en un número "secreto" y realice una secuencia de operaciones matemáticas sobre él (doblar su valor, restarle 2, dividir la respuesta por 10, y así sucesivamente). Al final, y después de que el jugador haya efectuado todas las operaciones solicitadas, el mago usará sus poderes mágicos para adivinar el resultado, a pesar de que no sabía a priori el número inicial del usuario. La siguiente figura ilustra cómo funciona el juego:

Wizard's Instruction	Your Number
Think of a number.	2
Add 5.	7
Multiply by 3.	21
Subtract 3.	18
Divide by 3.	6
Subtract your original number.	4

Después de la última instrucción, el hechicero nos dirá que tras seguir las instrucciones, el resultado es 4, a pesar de que el juego no sabía que empezamos con el número 2. ¿Adivinas cuál es el truco?

Abre el archivo **Proyecto 26_sinCodigo.sb2**. La interfaz de usuario se muestra en la figura.

La aplicación contiene tres objetos: el objeto "hechicero" (wizard), que le da al usuario las instrucciones a calcular, y los objetos de los botones "OK" y "New". La aplicación también usa las listas "index" y "instr", ambas de ámbito local para el mago (ver figura).



La lista "inst" (a la derecha) contiene los registros para los 11 acertijos distintos que contiene la aplicación. Cada registro incluye (a) la respuesta al acertijo, (b) las instrucciones, y (c) un elemento vacío para marcar el final de ese registro. Las entradas de la lista "index" (a la izquierda) identifican el primer índice de cada acertijo contenido en la lista "inst". Por ejemplo, el segundo elemento de la lista "index" es 9, lo que significa que el registro del segundo acertijo comienza en la novena posición de la lista "inst", como iustra la figura

La estrategia para desarrollar este juego es la siguiente:

1) Cuando el usuario comienza una nueva partida, la aplicación elige un número al azar entre 1 y 11 (porque nuestra aplicación contiene un total de 11 acertijos posibles).
2) La aplicación consulta la lista "index" para conocer la posición inicial del registro para el acertijo seleccionado.
3) La aplicación accede a la lista "instr" en el índice hallado en el paso 2. El primer elemento en ese registro es la respuesta al acertijo. Los elementos siguientes representan las instrucciones que el mago va diciendo.
4) La aplicación hace que el mago vaya diciendo las instrucciones del acertijo una tras otra, hasta encontrarse con un elemento vacío, lo que simboliza la última instrucción del acertijo. El mago espera a que el usuario presione la tecla "OK" antes de decir una nueva instrucción.
5) La aplicación revela la respuesta al acertijo.

Ahora que sabemos cómo debería funcionar la aplicación, vamos a comenzar con los programas de los dos botones:

Programa 1 (New): Este programa comienza al clicar sobre el objeto, y simplemente envía el mensaje "juegoNuevo".

Programa 1 (OK): Al clicar en el botón "OK" en respuesta a una instrucción, el objeto fija el valor de la variable "clicado" a 1 para informar al objeto "hechicero" de que el jugador ha realizado la instrucción que se le ha solicitado.

Cuando el objeto "hechicero" recibe el mensaje "juegoNuevo", ejecuta el siguiente programa:

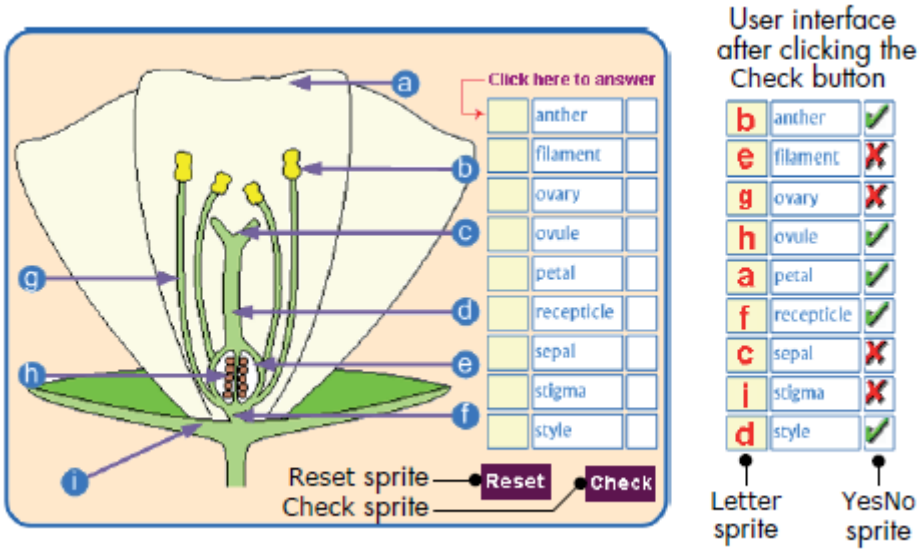
Programa 1 (hechicero): al recibir el mensaje "juegoNuevo", el programa comienza borrando cualquier bocadillo de diálogo de un acertijo previo, e inicializando la variable "clicado" a 0. A continuación, el programa gurda el número del acertijo elegido al azar en la variable "numAcertijo". Después de eso, lee la posición de inicio del acertijo seleccionado de la lista "index", y lo guarda en la variable "pos". Posteriormente, el programa usa la variable "pos" para leer la respuesta al acertijo seleccionado, y la guarda en la variable "respuestaAcerijo". Luego, el programa suma 1 unidad al valor de "pos" para pasar a apuntar a la primera instrucción del acertijo, y arranca un bucle "repetir hasta que ( )" para ir diciendo todas las instrucciones del acertijo en orden. (Este bucle se repetirá hasta que la instrucción leída sea una cadena vacía, lo que marca el final del acertijo, ver figura). A cada pasada del bucle, el hechicero dice la instrucción actual (la instrucción en el índice "pos" de la lista "instr"), fija "clicado" a 0, y espera hasta que "clicado" tome el valor 1 (esto es, hasta que el usuario clique en el botón "OK") antes de moverse a la siguiente instrucción (cambiando "pos" en 1 unidad). El programa sale del bucle cuando encuentra un elemento vacío en la lista "instr"), y el programa termina diciendo la respuesta al acertijo.

Con esto terminamos el proyecto. Guarda el archivo como **Proyecto 26.sb2**.

AMPLIACIÓN: Si eliminas uno de los acertijos de la lista "instr", o cambias el número de instrucciones de uno de los acertijos, debes reconstruir la lista "index" para reconciliarla con los contenidos de la lista "instr". Escribe un procedimiento que rellene automáticamente la lista "index", basándose en los contenidos actuales de la lista "instr". La idea clave es buscar cadenas vacías en la lista "instr", ya que estas cadenas indican el final de un registro de acertijo y el comienzo del siguiente.

PROYECTO 27. ANATOMÍA FLORAL.

En este proyecto construiremos un cuestionario sobre las partes anatómicas de una flor. La interfaz de la aplicación está disponible en el archivo **Proyecto 27_sinCodigo.sb2**, y se muestra en la figura:

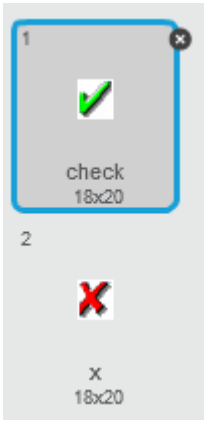


La figura muestra el aspecto de la aplicación al principio del cuestionario, y después de que el programa compruebe las respuestas del usuario. El usuario introducirá en las casillas amarillas las letras que en la figura se corresponden con los nombres de las partes de la flor indicadas, y después pinchará en el botón

"check" para comprobar sus respuestas. El programa compara las respuestas del usuario con las respuestas correctas, y proporciona una corrección usando iconos de marcas verdes y cruces rojas para indicar qué respuestas son correctas y cuáles son erróneas.

Esta aplicación usa tres listas. "correctAns" (respuesta correcta) contiene las letras que se corresponden con las respuestas correctas de las nueve partes de la flor indicadas en el juego. "cellYCenter" contiene las 11 posiciones verticales usadas por los objetos "Letter" (letra) y "YesNo" (SiNo) que les permiten saber dónde estampar sus respectivos disfraces. Cuando el usuario clica con el ratón sobre una de las casillas de respuesta, el escenario detecta este evento y pide una respuesta. El escenario actualiza el correspondiente elemento de la lista "ans" (respuesta) para ajustarse a la letra insertada por el usuario, yb estampar esa letra sobre la casilla de respuesta adecuada.

El objeto "YesNo" posee dos disfraces, la marca verde y la cruz roja. Cuando el usuario clica en el botón "Check" para comprobar sus respuestas, el objeto "YesNo" ejecuta el siguiente programa:



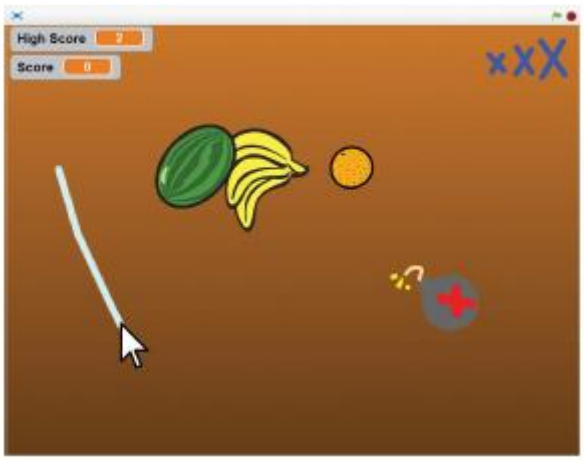
Programa 1 (YesNo): El programa comienza al recibir el mensaje "comprobar", y empieza mostrando al objeto. A continuación, ajusta el valor de la variable "pos" a 1 para acceder al primer elemento de las listas "correctAns" y "ans". A continuación, el programa arranca un bucle "repetir ( )" que realizará tantas iteraciones como elementos tenga la lista "correctAns". (El bucle usa la variable "pos" para indexar los elementos de las listas "correctAns" y "ans"). A cada pasada del bucle, el programa fija el valor de la variable "ch1" al elemento actual de la lista "correctAns" y el valor de la variable "ch2" al elemento actual de la lista "ans". A continuación, comprueba si los valores de "ch1" y "ch2" son iguales (esto es, comprueba uno a uno si los elementos de las listas "correctAns" y "ans" coinciden). Si los dos valores son iguales, cambia al disfraz de la marca verde. Si son distintos, cambia al disfraz de la cruz roja.

En ambos casos, el programa acude a la posición adecuada en el escenario (ver figura), estampa el disfraz seleccionado, y cambia el valor de "pos" en 1 unidad para acceder a los siguientes elementos de las listas "correctAns" y "ans" y compararlos. Con esto termina el bucle "repetir". Ya fuera del bucle, el programa termina ocultando al objeto.

Los programas del resto de objetos ya están escritos en el archivo. Prueba la aplicación, y guarda el archivo como **Proyecto 27.sb2**.

PROYECTO 28. EL NINJA DE LA FRUTA.

"Fruit ninja" fue un popular videojuego de móviles en el que la aplicación lanzaba fruta por el aire, y el jugador debía cortarla deslizando el dedo por pantalla antes de que tocase el suelo. En nuestra versión del videojuego el jugador cortará la fruta clicando y arrastrando el ratón.



A) Construye el fondo de la pantalla de inicio.
Nuestra aplicación tendrá una pantalla de inicio. Cuando el usuario pinche en la bandera verde, la pantalla de inicio mostrará el título del juego. Cuando el jugador pierda, el juego volverá a la pantalla de inicio. (La pantalla de inicio también es un buen lugar para poner tu nombre como desarrollador de la aplicación).

Crea un proyecto nuevo, y guárdalo como **Proyecto 28.sb2**.

1) Dibuja los fondos.
Selecciona la miniatura del escenario, y clica en la pestaña de "fondos". Usa la herramienta "línea" para dibujar los contornos de las letras del título ("Ninja de la fruta"). Después, usa la herramienta "rellenar con color" para rellenar las letras.

> NOTA: El diseño que aquí te proponemos es solo un ejemplo. Siéntete libre para elegir cualquier otro a tu gusto (otros colores de letras, otros colores de fondo, otros gradientes, etc.).

Después de dibujar el título, reemplaza el fondo blanco con un gradiente marrón oscuro. Selecciona el color marrón oscuro en la paleta de colores. (Al seleccionar la herramienta "rellenar con color", aparecen los cuatro gradientes disponibles junto a la paleta de colores). Después clica sobre el intercambiador de color (remarcado en amarillo en la figura), y elige un color marrón claro. Por último, selecciona uno de los gradientes de color para elegir el tipo de gradiente. (Nosotros hemos elegido el gradiente vertical, remarcado en azul en la figura).

Una vez hechos todos estos ajustes, pincha en el lienzo del editor gráfico para rellenar el escenario con el gradiente configurado. (Asegúrate de rellenar también los huecos de las letras, como el hueco de la "R", de la "A", etc. Al final, debería quedarte algo similar a esto:

Cambia el nombre del fondo de "fondo1" a "pantalla de inicio".

Ahora vamos a crear el fondo liso donde se desarrollará la acción del juego. Al comenzar el juego, la aplicación debe esconder la pantalla de inicio y cambiar a este fondo. Clica en el botón "dibujar nuevo fondo" bajo la etiqueta "fondo nuevo". A continuación, dibuja un gradiente marrón para este fondo, como el mostrado en la figura. Cambia el nombre de este fondo de "fondo2" a "pantalla del juego".

2) Agrega el código al escenario.
Selecciona la miniatura del escenario en la lista de objetos, y pincha en la pestaña "programas". Crea dos variables de ámbito global llamadas "puntuación" y "puntuación más alta".

Crea dos nuevos mensajes, llamados "pantalla de inicio" y "comenzar juego". Esto puedes hacerlo clicando en la flechita negra y seleccionando la opción "nuevo mensaje" disponible en los bloques "enviar (mesaje1)" y "al recibir (mensaje1)" de la categoría "eventos".

A continuación, escribe los siguientes programas:

Programa 1 (escenario): al clicar en la bandera verde, el programa fija el valor de "puntuación más alta" a 0, y envía el mensaje "pantalla de inicio".

Programa 2 (escenario): Al recibir el mensaje "pantalla de inicio", fija el valor de puntuación a 0, y cambia al fondo "pantalla de inicio" de la categoría "apariencia".

Programa 3 (escenario): Al recibir el mensaje "comenzar juego", cambia al fondo "pantalla de juego".

B) Haz el rastro del corte.
Nuestra aplicación no usa un objeto para representar al jugador. En vez de eso, el jugador clica y arrastra el ratón sobre el escenario, haciendo que aparezca brevemente un rastro que simula la estela dejada por el filo de una espada.

3) Dibuja el objeto "corte".
Clica en el botón "dibujar nuevo objeto" junto a la etiqueta "nuevo objeto" sobre la lista de objetos. Renombra este nuevo objeto como "corte". En el editor gráfico, dibuja un rectángulo azul claro similar al mostrado:

IMPORTANTE: El color del objeto "corte" no debe ser el mismo que el que usemos en otras frutas. En caso contrario, esas frutas serían capaces de cortar otras frutas.

4) Crea las listas y las variables del objeto "corte".
Vamos a crear una lista que nos permita saber en qué lugar de la pantalla está realizando el corte el jugador. Como el corte realizado por el jugador se dibujará con el lápiz entre muchas posiciones (x, y), necesitamos recurrir a una lista.

Con el objeto "corte" seleccionado, crea dos listas con ámbito local llamadas "corte x" y "corte y". A continuación, crea una nueva variable local llamada "i". (En el siguiente paso veremos qué uso le damos a esta variable). De esta forma, el objeto "corte" dispondrá de dos listas y una variable para dibujar el rastro del corte.

5) Registra los movimientos del ratón.
Usaremos los bloques de la categoría "lápiz" para dibujar líneas para el rastro del corte. Pero antes, necesitamos saber *dónde* dibujar esas líneas. Añade el siguiente programa al objeto "corte".

Programa 1 (corte): Al presionar la bandera verde, el programa comienza borrando todos los contenidos de las listas "corte x" y "corte y". A continuación, arranca un bucle "por siempre". A cada pasada del bucle, el programa comprueba si el jugador está presionando el botón del ratón, bloque "¿ratón presionado?" de la categoría sensores. En ese caso, añade los valores de las posiciones x e y del ratón a las listas "corte x" y "corte y" respectivamente. (De esta forma, y mientras el jugador mantenga el ratón presionado, los valores de las coordenadas x e y por las que se mueve el ratón se van almacenando al final de las listas "corte x" y "corte y"). A continuación, y aún dentro del bucle, el programa comprueba si la longitud de la lista "corte x" es mayor que 4 o si el ratón ya no está presionado (ver figura). En ese caso, borra el primer elemento de las listas "corte x" y "corte y". (De esta forma, si hay más de 4 elementos en las listas, el programa borra el primero de ellos. Esto sirve para que el programa solo se quede con las 4 últimas coordenadas (x, y) del ratón. Este borrado también se hace si el jugador deja de presionar el ratón. Esto sirve para acortar el rastro del corte cuando el jugador suelta el botón del ratón)

Este programa rellena las listas "corte x" y "corte y" con las coordenadas adecuadas para dibujar el rastro del corte. El siguiente programa se encargará de dibujarlo.

6) Construye un procedimiento para dibujar el rastro del cuchillo.

Mediante un bloque customizado, crea un nuevo procedimiento llamado "dibujar corte", y en las opciones, activa la casilla "correr instantáneamente".

Una vez creado este procedimiento (notar que aún no le hemos dotado de contenido), añade el siguiente programa al objeto "corte":

Programa 2 (corte): Al pinchar en la bandera verde, el objeto "corte" se esconde, borra la pantalla, fija el color del lápiz al mismo color azul que usamos al dibujar el disfraz del objeto "corte" (toma este color de la miniatura del objeto en la lista de objetos; de hecho, para eso creamos el rectángulo de color azul como disfraz del objeto), fija el tamaño del lápiz a 6, y a continuación, y con un bucle "por siempre", llama constantemente al procedimiento "dibujar corte".

Ahora vamos a definir el procedimiento "dibujar corte". Queremos que "dibujar corte" dibuje una línea que comience en las primeras coordenadas (x, y) del rastro del corte (coordenadas almacenas en los primeros elementos de las listas "corte x" y "corte y"). La línea irá a las posiciones (x, y) indicadas por los siguientes

valores de las listas "corte x" y "corte y", hasta llegar al último par de valores. Nuestro código deberá asegurarse de que las longitudes de las listas "corte x" y "corte y" siempre son iguales.
Procedimiento "dibujar corte" (objeto "corte"): El procedimiento comienza limpiando la pantalla, y a continuación, comprueba si la longitud de la lista "corte x" es mayor que cero (esto es, si la lista no está vacía). En ese caso, sube el lápiz y entonces se va a las coordenadas (x, y) almacenadas en los primeros índices de las listas "corte x" y "corte y". Después (y aún dentro del condicional), baja el lápiz, y fija el valor de la variable "i" a 2 (para acceder a los segundos elementos de las listas). Posteriormente, el condicional arranca un bucle "repetir hasta que ( )" que usa la variable "i" para controlar dónde debería moverse el lápiz en cada iteración. Este bucle se está repitiendo hasta que "i" se hace mayor que la longitud de "corte x". A cada iteración del bucle, el objeto se mueve a las coordenadas (x, y) almacenadas en las posiciones "i" de las listas "corte x" y "corte y", y cambia el valor de "i" en una unidad para acceder a los siguientes elementos de las listas en la próxima iteración. Cuando el bucle termina, el objeto sube el lápiz, y con ello termina el condicional y el procedimiento.

NOTA: ¡Cuidado! Para obtener la longitud de la lista "corte x", debemos usar el bloque naranja oscuro "longitud de (corte x)" de la categoría "datos", y no el bloque verde "longitud de ( )" de la categoría "operadores".

C) Haz un botón para comenzar el juego.
Mediante un botón de comienzo, el jugador tendrá la posibilidad de decidir cuándo empieza la partida.

7) Crea un objeto de tipo botón para empezar la partida.
Pincha en el botón "selecciona un nuevo objeto de la biblioteca", y selecciona el objeto "button1". Abre el área de información de este nuevo objeto y cambia su nombre a "botón de comienzo".

Clica en la pestaña de "disfraces". Selecciona el color blanco en la paleta de colores, y a continuación, usando la herramienta de texto, escribe "comenzar". A continuación, redimensiona y arrastra el texto para dejarlo sobre la parte inferior del botón:

Vuelve a crear el texto "comenzar" por segunda vez. Rota el texto para que quede boca abajo, y arrástralo para dejarlo en la parte superior del botón.

A continuación, añade el siguiente código para el objeto "botón de comienzo":

Programa 1 (botón de comienzo): Al recibir el mensaje "pantalla de inicio", el botón se muestra, y a continuación arranca un bucle infinito que lo hace girar continuamente en pasos de 2 grados. El bucle también comprueba constantemente si el botón está tocando el color azul del objeto "corte", y en ese caso, oculta al objeto botón, envía el mensaje "comenzar juego", y detiene este programa.

Este programa le permite al usuario comenzar la partida al hacer un corte sobre el botón de inicio.

D) Haz frutas y bombas que se lancen a través de la pantalla.
Las frutas de este juego tienen aspectos distintos, pero todas ellas se comportan de la misma forma. Por ello, usaremos un único objeto con el mismo código, pero con varios disfraces diferentes.

8) Crea el objeto "fruta".
Pincha en el botón "selecciona un objeto de la biblioteca", y selecciona el objeto "apple" de la biblioteca de Scratch. En el área de información del objeto, cambia su nombre a "fruta".

Acede a la pestaña de "disfraces" del objeto "fruta", y pincha en el botón "selecciona un disfraz de la biblioteca". Añade el disfraz "banana". Repite esta operación para añadir los disfraces "orange" y "watermelon-a". De ser necesario, arrastra los disfraces para que queden en el orden indicado en la figura:

Ahora, en la biblioteca de disfraces no hay un disfraz de bomba, así que tendremos que dibujarlo nosotros mismos. Si el jugador corta accidentalmente una bomba, pierde la partida.

En el editor gráfico de Scratch, pincha en el botón "pintar nuevo disfraz". Ahora, dibuja una bomba con una X roja sobre ella, lo más parecida posible a la de la figura. Para ello, usa la herramienta elipse para dibujar el cuerpo de la bomba, la herramienta rectángulo para la parte superior, y la herramienta línea para X roja, la mecha, y las chispas. Juega con el control del grosor de la línea para conseguir líneas más anchas o estrechas, según te convenga. Renombra este disfraz como "bomba".

9) Crea los disfraces de las frutas partidas.
El objeto "fruta" tendrá disfraces de los 4 tipos de frutas, de la bomba, y de las 4 frutas partidas. Este paso es muy sencillo: lo único que tenemos que hacer es duplicar los disfraces de las frutas y separarlos ligeramente para que parezca que se han cortado limpiamente.

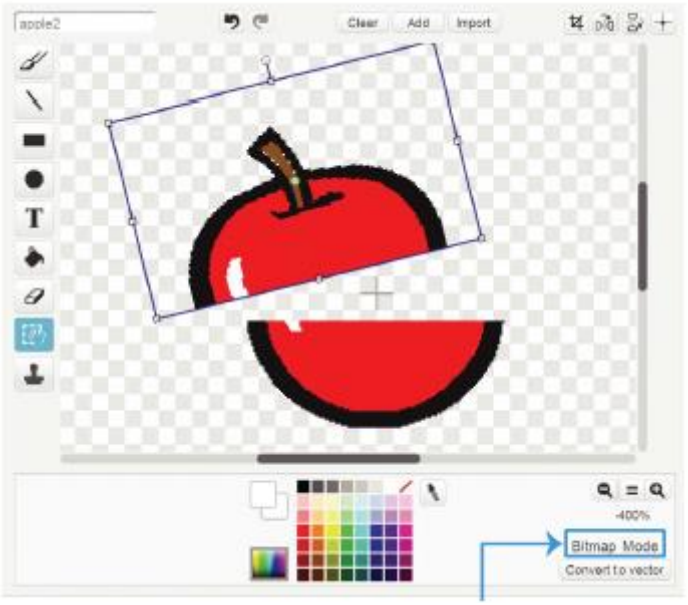
En el editor gráfico, haz clic con el botón derecho del ratón sobre el disfraz de la manzana, y selecciona "duplicar" para crear un segundo disfraz de manzana idéntico al primero. Duplica también el plátano, la naranja, y la sandía. Asegúrate que las réplicas están en el mismo orden que los disfraces originales:

1. Manzana.
2. Plátanos.
3. Naranja.
4. Sandía.
5. Bomba.
6. Manzana2.
7. Plátanos2.
8. Naranja2.
9. Sandía2.
10. Bomba2.

NOTA: Insistimos en que el orden indicado para los disfraces será vital para el código que desarrollaremos a continuación. Vuelve a comprobar que el orden es el correcto.

Para que el código del objeto "fruta" funcione para todas las frutas, el número de disfraz de la fruta cortada debe ser 5 unidades mayor que el número de disfraz de la correspondiente fruta sin cortar. Por ejemplo, el disfraz de la manzana es el disfraz 1, y el de la manzana cortada es el disfraz 6; el disfraz de los plátanos es el disfraz 2, y el de los plátanos cortados es el disfraz 7. Ésta es la razón por la que el orden de los disfraces es muy importante. Además, también necesitaremos un segundo disfraz de bomba, que será el disfraz 10. Duplica el disfraz de la bomba, aunque no tendremos que hacer una versión partida de él.

Ahora, las imágenes de los disfraces de fruta son tipo vectorial, lo que significa que están hechas como una colección de formas simples, y no como un conjunto de píxeles. Aunque las imágenes tipo vectorial tienen una mejor apariencia que las imágenes tipo mapa de bits, no pueden dividirse en dos con las herramientas del editor gráfico de Scratch. Por ello, debemos convertirlas a imágenes tipo mapa de bits.



Para ello, selecciona los disfraces duplicados de las frutas (disfraces 6, 7, 8, y 9), y clica en el botón "convertir a mapa de bits" en la esquina inferior derecha del editor. A continuación, usa la herramienta "seleccionar" para hacer una selección rectangular de la mitad superior de cada disfraz. Arrastra la mitad seleccionada para separarla un poco de la otra mitad, y rótala ligeramente para que parezca que está cortada por la mitad (ver figura). Repite este proceso para todos los disfraces duplicados de las frutas.

10) Añade el código del objeto "fruta".
El objeto "fruta" ya dispone de todos los disfraces que necesita, así que vamos a programar su comportamiento. El objeto "fruta" crea clones de sí mismo, y cada clon elegirá un disfraz aleatorio de entre sus disfraces 1, 2, 3, 4, o 5. Después, los clones se lanzarán a sí mismos hacia arriba por el aire. El clon detectará si ha sido cortado, y cambiará a su disfraz cortado. O bien, si el clon ha elegido aleatoriamente el disfraz de la bomba y el jugador lo corta, la partida termina.

Comencemos creando las variables *locales* "velocidad x", "velocidad y", "rapidez rotación", y "número de frutas". Estas variables las usarán todos los clones para determinar la forma en la que se lanzan hacia arriba y caen hacia abajo.

Si el jugador corta la bomba, el juego reproducirá un ruido. Clica en la pestaña de "sonidos" y a continuación, pincha en el botón "selecciona un sonido de la biblioteca". Elige el sonido "alien creak2".

Una vez creadas las variables y agregados los sonidos, podemos escribir el código que le permite al objeto "fruta" crear múltiples clones de sí mismo a cada segundo. En el paso 11 añadiremos el código que controla el comportamiento de los clones.

Programa 1 (fruta): Este programa empieza al pinchar en la bandera verde, y simplemente esconde el objeto.

Programa 2 (fruta): Este programa comienza al recibir el mensaje "terminar juego", y se encarga de parar el resto de programas del objeto, y de borrar el clon actual.

Programa 3 (fruta): Este programa arranca cuando el jugador corta el botón de inicio, esto es, cuando recibe el mensaje "comenzar juego". Nada más empezar, el programa entra en un bucle infinito. A cada pasada del bucle, el programa espera 1 segundo (los clones se van creando segundo a segundo), y fija el valor de la variable "número de frutas" a un número aleatorio entre 1 y 4 (para lanzar simultáneamente entre 1 y 4 frutas). Después, el bucle infinito entra en un bucle "repetir" interno, que se repite un número de veces igual al valor de "número de frutas". (Este bucle interno creará un nuevo clon cada iteración). Dentro de este bucle, el objeto "fruta" original fija su posición x a un valor aleatorio entre -200 y 200 y su coordenada y a la posición -170 (para ubicar al objeto original en algún lugar en la parte inferior de la pantalla). A continuación, el bucle fija los valores de las variables "velocidad x", "velocidad y", y "rapidez de rotación" a unos valores aleatorios, como muestra la figura (de esta forma, la fruta se lanzará a sí misma de forma aleatoria). Después, el objeto cambia a un disfraz aleatorio entre los disfraces 1 a 5, y por último, crea un clon de sí mismo. (Al crear el clon, la posición, variables, y disfraz actuales del clon son una copia de las del objeto original, por lo que el objeto original estará listo para configurarse aleatoriamente para la creación del siguiente clon). Con esto finaliza el bucle interno, el bucle infinito, y el programa.

NOTA: El bloque "fijar (velocidad x) a ( )" utiliza una ecuación complicada. El valor de "velocidad x" determinará lo lejos que se lanzará la fruta hacia la derecha o hacia la izquierda. Este valor no puede ser muy grande, así que tomamos el valor del bloque "posición en x" del objeto (un número aleatorio entre -200 y 200), y lo dividimos entre 50, lo que proporciona un número aleatorio entre -4 y 4.

Además, siempre querremos que la fruta se lance hacia el centro del escenario, lo que significa que si el objeto "fruta" está a la izquierda del escenario (valor de "posición en x" negativo), el clon debería lanzarse hacia la derecha, esto es, el valor de la variable "velocidad x" debería fijarse a un valor positivo. (Y viceversa, si el objeto "fruta" está a la derecha, el clon debería lanzarse hacia la izquierda). Para ello, debemos multiplicar $"posición\ en\ x"\ /50$ por -1. Finalmente, y para que los lanzamientos varíen de uno a otro, a todas esta operación le sumamos un número aleatorio entre -2 y 2. En el paso 11 veremos la forma en la que los clones usan la variable "velocidad y".

Los valores de "velocidad x" y "velocidad y" se usarán conjuntamente para que el clon se lance siguiendo una trayectoria parabólica. Las matemáticas de las parábolas son muy útiles en ciencia e ingeniería, y probablemente ya las hayas estudiado, pero a priori, no son sencillas. Es por ello que algunos de los bloques que usaremos para recorrer esta trayectoria parabólica pueden resultarnos un poco complicados.

11) Añade el código para los clones del objeto "fruta".

Cuando el programa 3 del objeto "fruta" original cree un nuevo clon de sí mismo, el clon comenzará a ejecutar su propio código para lanzarse y detectar si ha sido cortado.

Para este nuevo programa, necesitaremos crear un mensaje al que llamaremos "fruta fallada".

Programa 4 (fruta): Al crear cada clon, el objeto original está oculto, y lo primero que hace este programa es mostrarlo. A continuación, arranca un bucle "repetir hasta que ( )" que se encarga de la gravedad y de la detección del corte. Este bucle se estará repitiendo mientras el clon esté en el aire, esto es, hasta que la posición en y del clon sea menor que -170. A cada pasada, el bucle hace que el clon caiga más y más rápido por efecto de la gravedad cambiando su "velocidad y" en -1 unidades. A continuación, el bucle cambia la posición en x, la posición en y, y la dirección del clon en unos valores iguales a "velocidad x", "velocidad y", y "rapidez de rotación", respectivamente. (Para cambiar la dirección del clon, giramos al clon hacia la derecha un número de grados igual a "rapidez de rotación"). Con ello conseguimos que el clon se mueva en una trayectoria parabólica a través del aire. (Para conseguir un movimiento parabólico, debemos hacer que la velocidad horizontal ("velocidad x") se mantenga constante, y que la velocidad vertical ("velocidad y") disminuya con el tiempo, y esto es precisamente lo que hace esta parte del programa).

Todavía dentro del bucle, el programa comprueba (con un bloque "si") si el clon está tocando el color azul del objeto "corte" (esto es, si el clon ha sido cortado por el jugador). Si ése es el caso, el programa hace una segunda comprobación (con un bloque "si / si no" anidado) para detectar si el disfraz del clon es el número 5 (esto es, para comprobar si el jugador ha seccionado la bomba):

- En caso afirmativo (grupo SI), la bomba seccionada reproduce el sonido "alien creak2", se va a la capa frontal, realiza una serie de 10 aumentos de tamaño en incrementos de 30, y envía el mensaje "terminar juego".
- En caso negativo (grupo SI NO), es decir, si hemos cortado una fruta, un nuevo bloque "si" comprueba si el número del disfraz del clon es menor que 5 (esto es, si es una fruta todavía no cortada), y en ese caso, el programa reproduce el sonido "pop", cambia el valor de "puntuación" en 1, y finalmente, cambia al disfraz que está 5 números más allá del número de disfraz actual (para ponerse el correspondiente disfraz de la fruta cortada).

Con esto terminan los condicionales anidados y el bucle "repetir hasta que ("posición en y"< -170)".

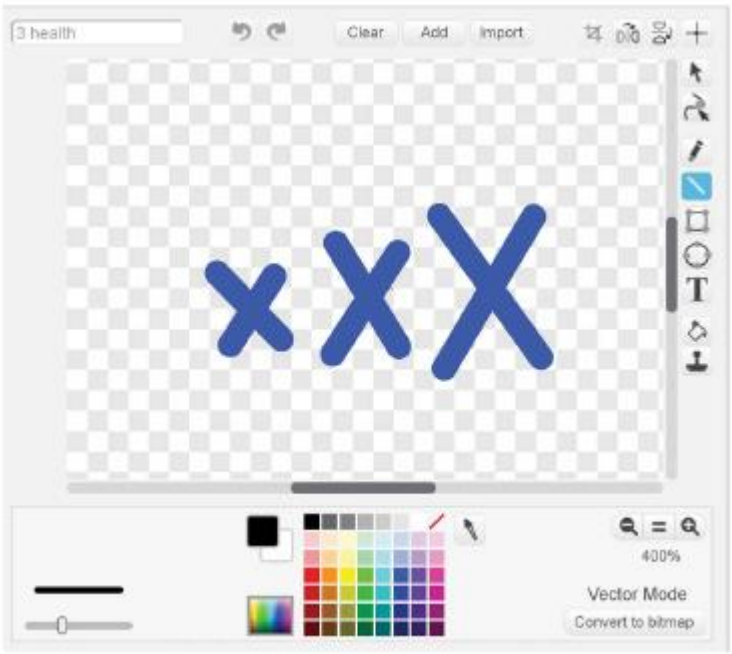
En este punto del programa ya hemos salido del bucle, lo que significa que el clon ya ha caído al suelo. Entonces, el programa comprueba si el disfraz del clon todavía es uno de los disfraces de las frutas sin cortar (disfraces 1, 2, 3, o 4, esto es, si el número del disfraz actual es menor que 5). En tal caso, el programa envía el mensaje "fruta fallada" (en el paso 12 veremos qué hace el programa que recibe este mensaje).

Por último, y ya fuera del condicional, el programa borra el clon, independientemente de si se trata de una bomba, una fruta sin cortar, o una fruta cortada, porque ese clon ya ha caído al suelo.

E) Crea el objeto "salud" y haz que funcione.
Nuestro juego es bastante difícil. El jugador no debe lanzar demasiados cortes porque podría seccionar una bomba y perder el juego. Sin embargo, tampoco puede relajarse porque también pierde la partida si se le escapan 3 frutas.

12) Crea el objeto "salud".
El objeto "salud" indicará cuántas frutas más puede fallar el jugador antes de perder la partida. Cada vez que el jugador falle el corte de una fruta, el objeto cambiará a su siguiente disfraz.

Clica en el botón "dibujar nuevo objeto". Abre el área de información de este nuevo objeto, y cambia su nombre a "salud". En el editor gráfico de Scratch, usa la herramienta "línea" para dibujar tres X azules, ver figura. (En nuestro caso, hemos dibujado las X con tamaño creciente de izquierda a derecha, pero diséñalas como a ti te parezca).



A continuación, duplica este disfraz tres veces más. Renombra al disfraz 1 como "3salud", al disfraz 2 como "2salud", al disfraz 3 como "1salud", y al disfraz 4 como "0salud".

Deja azules las tres X del disfraz "3salud", pero para el resto de disfraces, usa la herramienta "rellenar con color", el gradiente vertical, y dos colores rojos (claro y oscuro) para pintar algunas X con un gradiente rojo. El disfraz "2salud" solo tendrá una X roja (la de la izquierda), lo que significa que el jugador ya ha fallado una fruta. El disfraz "1salud" tendrá dos X rojas (las dos de la izquierda), esto es, el jugador ya ha

fallado dos frutas. Por último, el disfraz "0salud" tendrá las tres X rojas, lo que indica que el jugador ha fallado tres frutas. Asegúrate que los disfraces están en el orden que muestra la figura.

El objeto salud cambiará de disfraz basándose en diferentes mensajes que irá recibiendo. Añade el siguiente código al objeto "salud". (Los mensajes que usan estos programas ya deberían existir, porque se supone que ya los has creado antes).

Programa 1 (salud): Al pinchar en la bandera verde, el objeto simplemente se esconde (porque en la pantalla de inicio no debe aparecer el objeto "salud").

Programa 2 (salud): Al recibir el mensaje "comenzar juego" (esto es, cuando el usuario corta el botón de inicio), el objeto cambia al disfraz "3salud" (se pone el disfraz con las tres X azules, porque al empezar la partida todavía no ha fallado ninguna fruta), y a continuación, se muestra.

Programa 3 (salud): Al recibir el mensaje "fruta fallada" (esto es, cuando una fruta llega al suelo sin que el jugador la haya partido), el programa cambia al siguiente disfraz, y a continuación, comprueba si ese disfraz es el número 4 (esto es, el disfraz "0salud" con las tres X rojas). En ese caso, envía el mensaje "terminar juego" (porque el jugador ya ha fallado tres frutas).

Programa 4 (salud): Al recibir el mensaje "terminar juego" (esto es, cuando el jugador corta una bomba, o cuando falla tres frutas), el objeto se esconde.

F) Programa la finalización de la partida.
Como ya hemos mencionado, el jugador puede perder de dos formas distintas: (a) cuando el jugador falla tres frutas, o bien (b) cuando secciona una bomba. En cualquiera de estos dos casos, la aplicación envía el mensaje "terminar juego", y como resultado, ocurren varias cosas. Los clones del objeto "fruta" se borra a sí mismos, el objeto "salud" se esconde, y el escenario se desvanece a blanco. Ya hemos añadido el código necesario para los clones y para el objeto "salud", así que vamos a añadir el código que hace que el escenario se desvanece a blanco.

13) Crea el objeto "desvanecer a blanco".
Usaremos el efecto "desvanecer" para hacer que el escenario desaparezca. Pincha en el botón "dibujar nuevo objeto", y en el área de información, cambia el nombre de este nuevo objeto a "desvanecer a blanco". Ahora, en el editor gráfico, usa la herramienta "rellenar con color" para pintar *todo* el lienzo de blanco.

El objeto "desvanecer a blanco" bloquea la mayor parte de la pantalla. Cuando el programa comienza, el objeto "desvanecer a blanco" se ocultará, y se hará visible solo cuando reciba el mensaje "terminar juego". Añade el siguiente código al objeto "desvanecer a blanco":

Programa 1 (desvanecer a blanco): Al recibir el mensaje "pantalla inicio", el objeto se oculta.

Programa 2 (desvanecer a blanco): El programa comienza al recibir el mensaje "terminar juego". Como el objeto es del mismo tamaño que el escenario, el objeto acude a la posición $(0, 0)$ para cubrir completamente el escenario. A continuación, ajusta el efecto "desvanecer" a 100 para hacerse completamente invisible. Después, el objeto se va a la capa frontal, y se muestra. Posteriormente, y mediante un bucle "repetir ( )", hacemos que el objeto vaya apareciendo progresivamente reduciendo el efecto "desvanecer". (Ajusta adecuadamente el número de repeticiones, y el decremento del efecto "desvanecer"). Al aparecer el objeto, comprueba si el valor de "puntuación" es mayor que el valor de "puntuación más alta", y en ese caso, fija el valor de "puntuación más alta" al valor de "puntuación", para guardar el valor de este nuevo record de puntuación. Finalmente, y tras una breve espera de 1 segundo, el programa envía el mensaje "pantalla inicio" para volver a la pantalla de inicio.

¿Y AHORA QUÉ?

Scratch permite hacer juegos y animaciones muy divertidos, pero tiene sus limitaciones. Los juegos que desarrollamos en Scratch se parecen poco a los juegos "reales" con los que solemos jugar en nuestros ordenadores, consolas, o smartphones.

Por lo tanto, el siguiente paso lógico es aprender a programar con un lenguaje de programación profesional. Para ello hay múltiples opciones, pero las más recomendables tal vez sean Python o JavaScript. **Python** es el lenguaje de programación más sencillo de entre todos los lenguajes de uso profesional. Por su parte, **JavaScript** no es tan fácil, pero es el lenguaje usado por las aplicaciones web que se ejecutan en tu navegador.

Algunos textos recomendables son los siguientes:

1) "Python Crash Course" de Eric Matthes (Ed. No Starch Press).

2) "Invent Your Own Computer Games with Python" de Al Sweigart (Ed. No Starch Press).

3) "Automate the Boring Stuff with Python" de Al Sweigart (Ed. No Starch Press).

4) "A Smarter Way to Learn Python" de Mark Myers.

5) "Python Tricks" de Dan Bader (Ed. Dan Bader (dbader.org))

6) "JavaScript for Kids" de Nick Morgan (Ed. No Starch Press).

7) "A Smarter Way to Learn JavaScript" de Mark Myers.

8) "Head First JavaScript Programming" de Freeman y Robson (Ed. O'Reilly Media).

9) "Learn JavaScript VISUALLY with Interactive Exercises" de Ivelin Demirov.

REFERENCIAS.

Estos apuntes no son, en absoluto, originales. Las fuentes que hemos usado más frecuentemente son:

1) "Learn to Program with Scratch" de Majed Marji (Ed. No Starch Press).
Magnífico libro. Detallado y sistemático en el tratamiento del lenguaje Scratch, y accesible y riguroso en la presentación de los conceptos básicos de programación. Este libro es la principal referencia de estos apuntes.

2) "Scratch Programming Playground" de Al Sweigart (Ed. No Starch Press).
Un muy buen libro que muestra, paso a paso y de forma clara, cómo hacer los programas y videojuegos que cada capítulo propone.

3) "Scratch 2.0 Game Development" de Sergio van Pul y Jessica Chiang (Ed. PACKT Publishing).
Otro estupendo libro de programación de videojuegos con Scratch. Usar los proyectos del 3 al 8 (ambos inclusive).