¡Descubre cómo se hace un videojuego!

Gerardo Horvilleur

¡Descubre cómo se hace un videojuego!

Gerardo Horvilleur

Tabla de contenidos

Lista de tablas

Introducción

Para hacer un videojuego hay que aprender a programar. Una consola de videojuegos es una computadora especializada, y un videojuego es un programa para esa computadora. Por lo tanto, para hacer un videojuego es necesario saber programar.

Programar una computadora es como hacer magia. Los brujos tienen la idea de que por medio de conjuros se pueden invocar espíritus para que hagan lo que se les pide. Dentro de una computadora lo equivalente a un espíritu es un proceso computacional. Invocamos a los procesos computacionales por medio de nuestros programas, que son como los conjuros de los brujos.

Un proceso computacional es algo abstracto. No se puede ver ni tocar, pero existe. Se pueden ver sus efectos. El arte de programar consiste en saber qué "conjuros" escribir para que los procesos computacionales hagan exactamente lo que nostrosos deseamos. Y, al igual que el aprendiz de brujo, un programador principiante tiene que aprender a anticipar las consecuencias de sus conjuros. Aún pequeños errores (típicamente conocidos como *bugs*) en un programa pueden tener consecuencias complejas e inesperadas.

Por suerte aprender a programar es mucho menos peligroso que andar aprendiendo brujería. Lo peor que puede ocurrir es que alguno de nuestros programas no haga lo que esperamos.

Aprender a programar es como jugar un videojuego. Un programa de computadora es una lista de instrucciones que le dicen a un proceso computacional qué debe hacer en cada momento. Estas instrucciones se escriben en unos lenguajes especializados llamados lenguajes de programación. Existen muchos lenguajes de programación. El que nosotros vamos a emplear se llama simpleJ.

Un lenguaje de programación, al igual que los lenguajes que conocemos tales como el español o el inglés, tiene su propio vocabulario, sintaxis y semántica que hay que conocer para poder emplearlo. Por suerte, aprender un lenguaje de programación es mucho más sencillo que aprender un nuevo idioma. De hecho, aprender un lenguaje de programación ni siquiera se parece a aprender un idioma.

¿Entonces a que se parece aprender un lenguaje de programación? Aprender un lenguaje de programación se parece a aprender a jugar un videojuego. Es más ¡cada vez que usas un nuevo videojuego estas aprendiendo un nuevo lenguaje!

El lenguaje de un videojuego consiste en las ordenes que le das a tu personaje o vehículo para que haga lo que tu quieres. En ese lenguaje presionar cierto botón tal vez significa ¡Salta! y presionar otro botón significa ¡Agáchate!

Obviamente, el lenguaje de un videojuego es un lenguaje muy limitado. Sólo sirve para dar órdenes. ¡No se puede usar para expresar ideas abstractas o hacer un poema! Pasa exactamente lo mismo con un lenguaje de programación: sólo sirve para dar órdenes.

Cuidado, aprender un lenguaje de programación no es lo mismo que aprender a programar. Es como la diferencia entre saber qué botones hacen qué cosa en un videojuego y saber usar eso para pasar un nivel.

Típicamente, necesitas hacer varios intentos antes de poder pasar un nivel en un juego. Tal vez al principio hasta te cuesta trabajo controlar el personaje, pero a medida que vas avanzando te acostumbras y cada vez se te hace más fácil. Aún ya sabiendo controlar el personaje hay veces que para pasar un nivel es necesario hacer varios intentos hasta que uno descubre, por prueba y error, una manera de superar los obstáculos. Inclusive, diferentes personas encuentran diferentes maneras de pasar un nivel y todas las maneras son buenas, ninguna es mala, lo que importa es el resultado.

Exactamente lo mismo pasa con la programación. Primero tienes que entender el lenguaje que se usa para decirle a la computadora que hacer. Después tienes que ir aprendiendo cómo usar ese lenguaje para lograr que haga lo que tu quieres. Al principio va a ser un poco difícil, pero con la práctica cada vez te va a parecer más sencillo.

Escribir un programa es como pasar un nivel en un videojuego. Típicamente, la primera vez que lo intentas no lo logras. Pero fijándote bien en por qué el programa no hace lo que tu esperas y modificándolo para corregir los errores, después de unos cuantos intentos ya tendrás un programa que funcione correctamente. Y diferentes personas, para hacer lo mismo, escriben diferentes programas. Cada programa se puede escribir de muchas maneras diferentes y no se puede decir que una de ellas es la buena y las otras son malas (lo que a veces si se puede decir es que una manera es mejor que otra).

Este libro está dividido en doce niveles. Para cada nivel hay un *proyecto* del simpleJ devkit con todos los programas de ejemplo de ese nivel. Al leer las explicaciones de cada nivel es indispensable probar esos programas de ejemplo; son para que entiendas má fácilmente los conceptos que se explican ahí.

Al igual que cuando juegas un videojuego, cuando pruebas pasar un nivel por primera vez puede que te parezca difícil, pero ¡no te desesperes! Tómalo con calma. Si hay algo que no entiendes vuelve a leerlo con cuidado y prueba el programa de ejemplo cuantas veces sean necesarias, hasta que te quede claro cómo funciona. También es una buena idea ir siguiendo este tutorial junto con un amigo, así pueden comentar entre los dos lo que están descubriendo. Si hay algo que no logras entender, aunque lo leas varias veces, entonces puedes ir al sitio de Internet `http://www.simplej.com` e inscribirte en el foro para principiantes. Ahí siempre habrá alguien dispuesto a contestar tus preguntas.

La simpleJ virtual console. No es sencillo hacer un videojuego. Típicamente se requiere del trabajo de docenas de personas durante un par de años para hacer un videojuego. Pero esto no fue siempre así. Con las primeras generaciones de consolas era común que un videojuego fuera hecho por una sola persona en unos cuantos meses.

Para que sea más sencillo hacer videojuegos la simpleJ virtual console es una consola similar a las consolas de esa primera generación.

El simpleJ devkit. Para hacer un videojuego para un consola se necesita su *Developer's Kit*, un equipo especializado que cuesta miles de dólares y es bastante díficil de conseguir. El simpleJ devkit es lo que se emplea para desarrollar videojuegos para la simpleJ virtual console.

Nota

En el Apéndice D, *Instalación*, están las instrucciones para instalar el software de simpleJ que se encuentra en el CD que viene con este libro.

Nivel 1: Primeros pasos

El simpleJ devkit

Inicia el simpleJ devkit, selecciona el proyecto que se llama `Nivel_01` y haz click en el botón OK. En pocos segundos verás aparecer la ventana del devkit.

La ventana del devkit está dividida en 3 áreas:

Program	Aquí puedes ver los programas de ejemplo, escribir o modificar tus propios programas usando el *Editor*.
Video	Es la pantalla con la salida de video de la consola de videojuegos.
Log	Es la bitácora en la cual puedes ver los mensajes de error. Más adelante verás cómo puedes desplegar ahí información desde tus programas.

Arriba de estas 3 áreas hay unos botones que corresponden a las acciones que vas a usar más frecuentemente. También puedes tener acceso a estas acciones, y unas cuantas más, en los menúes que se encuentran arriba de esos botones.

En el *Editor* puedes ver ahora el siguiente programa que muestra todo lo que vas a saber hacer al terminar este nivel. Seguramente no entiendes casi nada de lo que dice ahí. No te preocupes. Dentro de poco te va a quedar claro qué significa.

```
setBackground(0, 0, 0);
showAt("Hola!", 12, 4);
pause(1.0);
showAt("Bienvenido a simpleJ", 5, 8);
pause(1.0);
showAt("Lo primero que vas a aprender", 1, 12);
showAt("es como desplegar mensajes en", 1, 13);
showAt("la pantalla.", 1, 14);
pause(4.0);
showAt("Tambien a cambiar el color", 1, 16);
pause(1.0);
```

```
showAt("del fondo", 1, 17);
pause(0.6);
setBackground(0, 0, 31);
pause(0.6);
setBackground(0, 31, 0);
pause(0.6);
setBackground(31, 0, 0);
pause(0.6);
setBackground(31, 31, 0);
pause(0.6);
setBackground(0, 31, 31);
pause(0.6);
setBackground(31, 0, 31);
pause(0.6);
setBackground(0, 0, 0);
pause(1.0);
showAt("y de las letras.", 11, 17);
pause(0.6);
setForeground(0, 0, 31);
pause(0.6);
setForeground(0, 31, 0);
pause(0.6);
setForeground(31, 0, 0);
pause(0.6);
setForeground(31, 31, 0);
pause(0.6);
setForeground(0, 31, 31);
pause(0.6);
setForeground(31, 0, 31);
pause(0.6);
setForeground(31, 31, 31);
pause(2.0);
showAt("Y como borrar la pantalla...", 1, 20);
pause(2.5);
clear();
pause(2.0);
setBackground(0, 0, 31);
```

Lo primero que vas a hacer es ejecutar el programa para ver qué hace. Haz click en el botón ▶ (Start) para ejecutar el programa.

Nota

En computación *ejecutar un programa* significa pedirle a la computadora que haga (ejecute) lo que dice el programa.

Al ejecutarlo despliega unos mensajes en pantalla (dentro del área de *Video*), haciendo unas breves pausas entre mensaje y mensaje, cambia el color del fondo y de las letras, para finalmente terminar borrando la pantalla.

Ahora que ya lo ejecutaste y viste lo que hace, vuelve a leer el programa. Probablemente ya empiezas a entender un poco de lo que dice ahí.

Para desplegar mensajes se emplea `showAt`

Ahora vas a escribir un programa muy sencillo. Lo único que debe hacer el programa es desplegar `Hola!` en la pantalla.

Para empezar a escribir un nuevo programa haz click en el botón  (New). Con eso le indicas al devkit que deseas editar un nuevo programa.

Nota

Si le hiciste cambios al programa que está en el editor y aún no los has grabado en el disco duro entonces, al intentar crear un nuevo programa, el devkit te indica que hay cambios que se perderían y te pregunta si quieres seguir adelante (y perder esos cambios) o cancelar la operación.

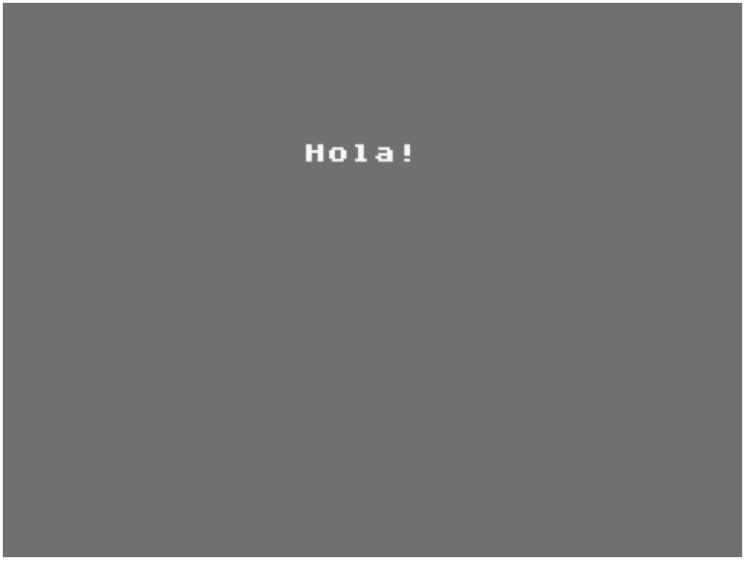
Para grabar un programa en el disco duro puedes usar el botón  (Save), o el botón  (Save as) si es que lo quieres grabar con otro nombre (o si todavía no tiene un nombre).

Escribe esto en el editor:

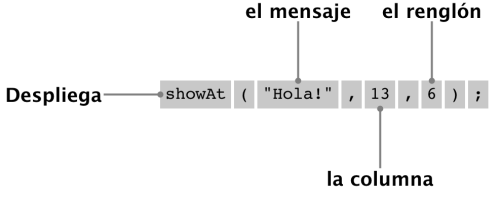
```
showAt("Hola!", 13, 6);
```

Ten cuidado de escribirlo *exactamente* como está escrito aquí arriba. Fíjate bien qué letras están escritas con mayúsculas y cuáles con minúsculas, es importante. Tampoco olvides escribir el punto y coma que está al final. Si haces algún error al escribirlo, entonces cuando trates de ejecutarlo el devkit te va a desplegar un mensaje de error (de color rojo) en el área de *Log*.

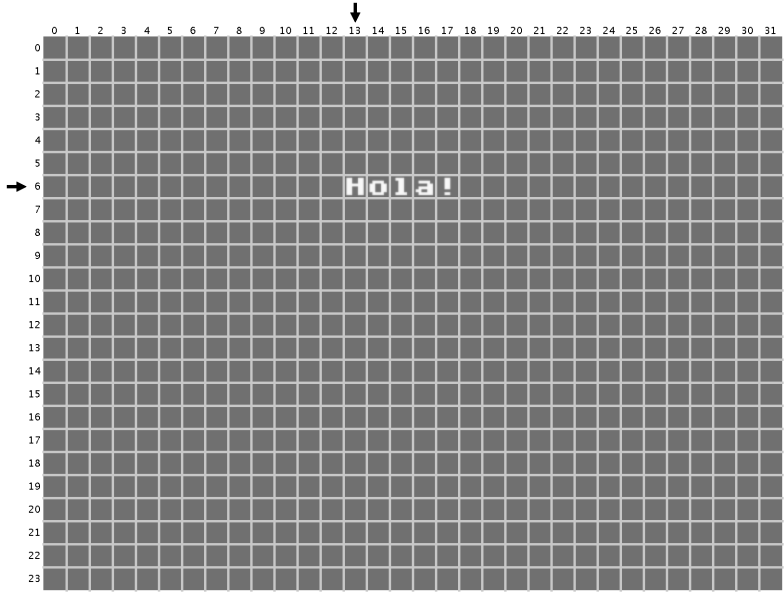
Ahora ejecútalo haciendo click en el botón ▶ (Start). Si te aparece un mensaje de error en el *Log* al intentar ejecutarlo no te preocupes. Nada más revisa lo que escribiste, compáralo con lo que dijimos que tenías que escribir y corrígelo en donde te hayas equivocado al copiarlo. Ya que lo puedas ejecutar sin que aparezca ningún mensaje de error debe aparecer esto en la pantalla:



Para ejecutar un programa la computadora lo examina para ver qué es lo que tiene que hacer. Así es como ve este programa:



Al ver el `showAt` la computadora sabe que lo que hay que hacer es desplegar un mensaje. El mensaje es `"Hola!"`, o más bien es `Hola!`, las comillas (`"`) son nada más para indicarle a la computadora en donde empieza y termina el mensaje. Y hay que desplegar el mensaje empezando en la columna `13` del renglón `6` (en simpleJ la pantalla puede desplegar 24 líneas de 32 caracteres cada una) tal como se ve aquí:



Nota

La numeración empieza en cero en vez de uno. Las columnas están numeradas del 0 al 31 y los renglones del 0 al 23.

Prueba hacerle unos cambios al programa. Cambia el mensaje que despliega y la posición en la que aparece en la pantalla. ¿Puedes hacer que despliegue el mensaje en la esquina superior izquierda? O algo un poco más difícil: ¿Puedes colocar el mensaje en la esquina inferior derecha?

Con pause se pueden hacer pausas

Ahora modifica tu programa para que quede así:

```
showAt("Hola!", 13, 6);
pause(1.5);
showAt("Como estas?", 10, 9);
```

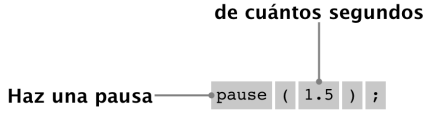
Nota

Pusimos aquí con letras negritas (más obscuras) las 2 líneas que le vas a agregar a tu programa, para que veas más fácilmente qué es lo que le tienes que cambiar. Cuando las escribas en el *Editor* no van a aparecer con letras negritas.

De ahora en adelante vamos a usar letras negritas para indicar los cambios que hay que hacer en un programa.

Ahora haz click en el botón ▶ (Start) para ejecutar el programa. Al ejecutarse primero despliega `Hola!`, después hace una pausa de segundo y medio antes de desplegar `Como estas?`.

Así es como la computadora ve la línea del `pause`:



El `1.5` quiere decir que se espere un segundo y medio. Si quieres que haga una pausa de una décima de segundo le pones `pause(0.1);`, y algo como `pause(4.0);` quiere decir que haga una pausa de 4 segundos.

Combinando el rojo, el verde y el azul obtienes todos los colores

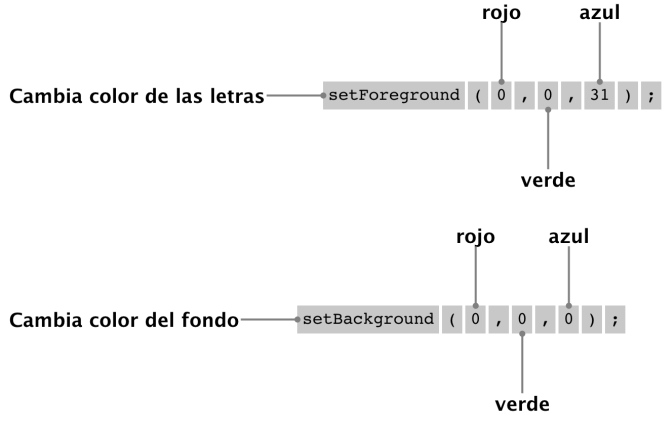
En las pantallas de los televisores y las computadoras se obtienen todos los colores por medio de combinaciones de rojo, verde y azul. Para que veas cómo funciona esto, modifica tu programa para que quede así:

```
showAt("Hola!", 13, 6);
pause(1.5);
showAt("Como estas?", 10, 9);
pause(1);
setForeground(0, 0, 0);
pause(2);
setForeground(31, 0, 0);
pause(2);
setForeground(0, 31, 0);
pause(2);
```

```
setForeground(0, 0, 31);
setBackground(0, 0, 0);
pause(2);
setForeground(31, 31, 31);
```

Y haz click en el botón ▶ (Start) para ejecutarlo y ver lo que hace. Primero pone las letras en negro, después en rojo, en verde, en azul (y pone el fondo en negro porque ¡letras azules sobre un fondo azul no serían visibles!), para finalmente dejar las letras de color blanco.

Así es como la computadora interpreta el `setForeground` y `setBackground`:



Tanto `setForeground` como `setBackground` necesitan tres números que indican la cantidad de rojo, verde y azul respectivamente. Esos números deben tener valores entre 0 y 31.

Un `setForeground(0, 0, 31);` indica que hay que mostrar las letras con el valor máximo de azul y nada de rojo ni de verde, por lo tanto las letras se ven azules. Mientras que un `setForeground(31, 0, 0);` pone las letras rojas y un `setForeground(0, 31, 0);` pone las letras verdes.

Si pones cantidades iguales de rojo, verde y azul te da tonos que van del negro hasta el blanco. Un `setBackground(0, 0, 0);` quiere decir que no hay nada de rojo, ni verde, ni azul, y por lo tanto pone el fondo en negro. Un `setBackground(31, 31, 31);` pone el rojo, verde y azul al máximo, con lo cual el fondo queda de color blanco, y un `setBackground(15, 15, 15);` hace que el fondo se vea gris.

Al emplear cantidades diferentes de rojo, verde y azul es como se obtienen los otros colores. Por ejemplo, un `setForeground(31, 31, 0);` pone las letras amarillas. Y un `setForeground(31, 0, 31);` las pone de color magenta.

Hazle cambios a tu programa y experimenta con diferentes combinaciones de rojo, verde y azul para que veas qué colores obtienes.

Nota

Tal vez te parezca extraño que al combinar rojo con verde te dé amarillo. Eso no es lo que pasa cuando usas lápices de colores sobre una hoja de papel. Esto se debe a que cuando usas lápices de colores estás usando un modelo de color *sustractivo*, y las pantallas de televisión o de computadora usan un modelo de color *aditivo*.

`clear` borra la pantalla

Modifica tu programa para que quede así:

```
showAt("Hola!", 13, 6);
pause(1.5);
showAt("Como estas?", 10, 9);
pause(1);
setForeground(0, 0, 0);
pause(2);
setForeground(31, 0, 0);
pause(2);
setForeground(0, 31, 0);
pause(2);
setForeground(0, 0, 31);
setBackground(0, 0, 0);
pause(2);
setForeground(31, 31, 31);
pause(2);
clear();
```

Y haz click en el botón ▶ (Start) para ejecutarlo. Al final borra todas las letras de la pantalla.

El `clear` es muy sencillo:

Borra las letras de la pantalla `clear();`

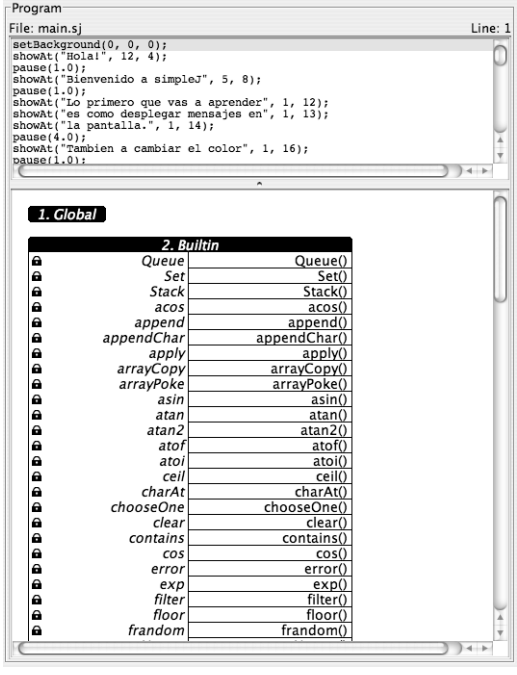
Ya puedes entender todo el programa del `Nivel_01`

Haz click en el botón (Save as) para grabar tu programa a disco, ponle un nombre como `mi_programa` (o algo así) y haz click en OK para que se grabe.

Ahora haz click en el botón (Open), selecciona el archivo `main.sj` y haz click en OK para verlo en el *Editor*.

Lee lo que dice el programa. ¡Ya puedes entender todo lo que hace!

También puedes ejecutar el programa paso por paso. Haz click en el botón (Step). El área *Program* se divide en dos: ahora está en el modo de ejecución paso por paso. En la parte superior está la vista del programa que estás ejecutando, con la línea que está a punto de ejecutarse resaltada con un fondo amarillo. En la parte inferior está la vista de la memoria de la computadora:



Ahorita no te preocupes por la vista de la memoria de la computadora, después veremos qué quiere decir lo que está ahí. Ahora puedes ir ejecutando el programa paso por paso. Haz click en ⏭ una vez más para ejecutar la primera línea del programa. En la salida de video el fondo cambia a negro. Cada vez que haces un click en el botón ⏭ se ejecuta una línea del programa. Si sigues ejecutando el programa línea por línea, después de ejecutarse la última línea, el área de *Program* se regresa automáticamente al modo de edición. En cualquier momento puedes hacer click en el botón ■ (Stop) para interrumpir la ejecución del programa y regresar al modo de edición.

Las computadoras no piensan

Seguramente ya tuviste oportunidad de equivocarte al escribir algo y de ver los mensajes de error que aparecen en el *Log*. Por ejemplo si te equivocas y escribes esto (con un espacio entre `show` y `At`):

```
show At("Hola!", 13, 6);
```

Te aparece este mensaje de error en el *Log*:

```
------------------------------
Error:
Compiler error
expecting ':', found 'at' at line 1 in file <no name>
unexpected token: ( at line 1 in file <no name>
unexpected token: Hola! at line 1 in file <no name>
unexpected token: 13 at line 1 in file <no name>
unexpected token: 6 at line 1 in file <no name>

------------------------------
```

Y si tu error es olvidarte de poner la `A` mayúscula en `showAt` y la escribes con una minúscula:

```
showat("Hola!", 13, 6);
```

Entonces te aparece este mensaje de error:

```
------------------------------
Error:
No such variable showat at line 1 in file <no name>
------------------------------
```

Uno se pregunta ¿No sería mejor si en vez de mostrar esos mensajes de error tan misteriosos, la computadora nos dijera algo como "Te equivocaste al escribir `showAt`"? Pues sí, sería mucho mejor. El problema está en que las computadoras no piensan y, por lo tanto, tampoco pueden entender lo que estás tratando de hacer.

Pero lo que nosotros podemos hacer es entender cómo la computadora procesa nuestros programas para ejecutarlos. Puedes encontrar mayor información respecto a esto en el Apéndice A, *Compilación*.

Se pueden poner comentarios

Un programa le dice a la computadora paso a paso lo que debe hacer. Para que un programa sea un buen programa no basta con que funcione correctamente, además debe estar escrito de manera que otras personas que lo lean puedan entender fácilmente como funciona. A veces ayuda poner dentro del programa explicaciones para los lectores, que la computadora simplemente debe ignorar. A estas explicaciones se les llama comentarios.

En simpleJ existen dos maneras de poner comentarios dentro de un programa. Para un comentario corto se puede emplear `//` (2 barras inclinadas una junto a la otra) que le indican a la computadora que ignore el resto de la línea. Por ejemplo, si una línea del programa contiene:

```
setBackground(31, 31, 0);     // Fondo amarillo
```

La computadora ejecuta el comando `setBackground(31, 31, 0);` e ignora el `//` y todo lo que viene después en esa línea.

Para un comentario más largo, que ocupa varias líneas, puedes emplear `/*` (barra inclinada seguida de asterisco) que le indica a la computadora que ignore todo lo que sigue hasta que encuentre `*/` (asterisco seguido de barra inclinada). Por ejemplo:

```
/*
  Empezamos por poner el fondo de color negro
  y desplegar unos mensajes de bienvenida.
  Con unas pausas para que se vea mas interesante
*/
setBackground(0, 0, 0);
showAt("Hola!", 12, 4);
pause(1.0);
```

```
showAt("Bienvenido a simpleJ", 5, 8);
pause(1.0);
showAt("Lo primero que vas a aprender", 1, 12);
showAt("es como desplegar mensajes en", 1, 13);
showAt("la pantalla.", 1, 14);
pause(4.0);
```

Dentro del proyecto `Nivel_01` está el mismo programa de ejemplo que ya viste pero con unos comentarios. Para verlo, haz click en el botón (Open) y selecciona `mejorado.sj`. Ejecútalo, con el botón , para que compruebes que hace lo mismo que la versión sin comentarios. Si quieres puedes ejecutarlo también paso a paso, con el botón , para que veas cómo la computadora se va saltando los comentarios.

Nivel 2: Variables, ciclos y procedimientos

Animaciones

Ahora vas a ver cómo se usan *variables*, *ciclos* y *procedimientos* para hacer unas animaciones sencillas.

En el menú de **Project** selecciona la opción **Switch to another project...** (**Project > Switch to another project...**) y selecciona el proyecto `Nivel_02`. Haz click en ▶ para ver las animaciones que hace el programa: texto que parpadea a diferentes velocidades, que desaparece y vuelve a aparecer gradualmente, y que entra a la pantalla por los lados.

Con lo que ya sabes podrías escribir un programa que haga todo esto. Para que el texto desaparezca, basta con cambiarle el color a las letras para que tengan el mismo color que el fondo y ya no se puedan ver (aunque siguen ahí). Para que vuelvan a aparecer simplemente las devuelves a su color original. Si quieres que desaparezcan gradualmente, les vas cambiando el color poco a poco hasta que tengan el mismo color que el fondo. Y para que el texto se mueva en la pantalla basta con irlo dibujando cada vez en una nueva posición en la pantalla.

Esto es justamente lo que hace este programa para lograr estos efectos. Pero, para que el programa no quede muy largo y sea más fácil hacerle cambios, usa variables, ciclos y procedimientos. Aunque se podría haber escrito este programa sin usarlos, es importante entender cómo funcionan porque es imposible hacer un videojuego sin ellos.

Variables

Las computadoras tienen una memoria en la cual almacenan el programa que están ejecutando y los datos sobre los cuales opera el programa. Los datos se almacenan dentro de *variables*.

Una variable es como un cajón dentro del cual puedes guardar un valor. Este valor puede ser un número, un texto o un procedimiento que le dice cómo hacer algo (también hay otros tipos de valores que veremos más tarde). En cualquier

momento el programa puede leer el contenido de una variable o sustituirlo por un nuevo valor.

Cada variable tiene un nombre. Dentro de un programa usas ese nombre para decirle a la computadora cuál es la variable que quieres emplear en un momento dado. Imagínate que ese nombre es como una etiqueta que le pones a cada cajón para saber qué es lo que tiene adentro. Tú escoges el nombre que le quieres poner a cada variable. Por ejemplo, puedes tener una variable que se llama `vidas` en la cual almacenas cuántas vidas le quedan al jugador y otra variable que se llama `puntos` que contiene la puntuación a la que ha llegado en el juego.

Las variables están agrupadas en *ambientes*. En cada momento la computadora tiene una lista de ambientes visibles. Cuando trata de accesar el valor almacenado en una variable recorre esta lista de ambientes hasta que encuentra un ambiente que contiene una variable con ese nombre. El nombre de una variable es único dentro del ambiente al que pertenece.

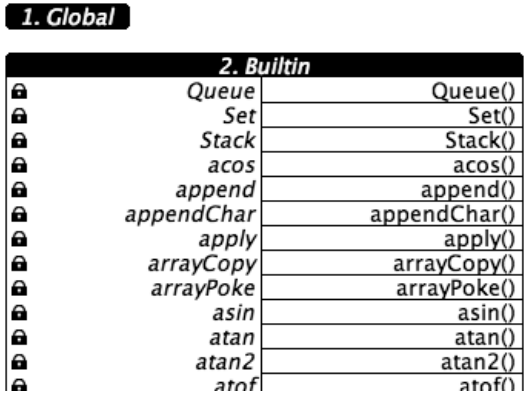
Para que esto te quede más claro vamos a ver un ejemplo. Haz click en  y selecciona el programa `variables.sj` para abrirlo en el editor:

```
var vidas;
vidas = 3;
print(vidas);
vidas = vidas - 1;
print(vidas);

var puntos = 0;
print(puntos);
puntos = puntos + 5;
print(puntos);
```

Si haces click en  para ejecutar el programa no parece que haga nada interesante. Simplemente te muestra estos números en el *Log*:

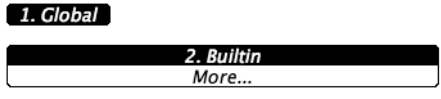
```
3
2
0
5
```

Lo interesante de este programa es ver lo que pasa dentro de la memoria de la computadora al ejecutarlo paso por paso. Haz click en ⏭ y fíjate en la vista de la memoria, te muestra esto:



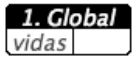
`Global` y `Builtin` son los dos ambientes que hay ahorita en la lista de ambientes visibles. Están numerados para que sepas en qué orden los va a recorrer la computadora cuando esté buscando una variable. `Global` es el ambiente en donde vas a crear muchas de las variables en tus programas. Al iniciar la ejecución de un programa está vacío, no contiene ninguna variable. `Builtin` es el ambiente en donde están las variables que contienen todos los procedimientos predefinidos. Tú ya conoces varios de estos procedimientos predefinidos: `showAt`, `pause`, `setBackground`, `setForeground` y `clear` (puedes usar la barra de scroll para recorrer la vista de la memoria y comprobar que están ahí, dentro de `Builtin`). En el Apéndice C, *Procedimientos predefinidos* están documentados todos los procedimientos predefinidos.

Como por ahora no nos interesa seguir viendo lo que está dentro de `Builtin`, haz un doble click sobre él para ocultar su contenido. La vista de memoria queda así:

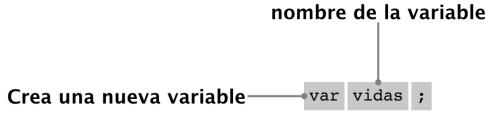


En cualquier momento le puedes volver a hacer un doble click para volver a ver lo que está adentro.

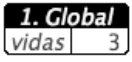
Ahora haz click en ⏭ para ejecutar la primera línea del programa. La vista de memoria ahora muestra esto:



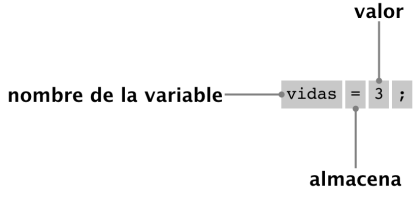
El programa creó una variable que se llama `vidas` dentro del ambiente `Global`. La vista de la memoria siempre muestra en rojo los cambios que ocurrieron al ejecutar una línea del programa para que sean más fáciles de ver. Así es como la computadora vio la línea que acaba de ejecutar:



Haz click una vez más en ⏭ para ejecutar la siguiente línea del programa. La vista de la memoria cambia a:

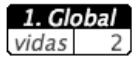


Esa línea le dijo a la computadora que almacenara el número 3 dentro de la variable `vidas`:

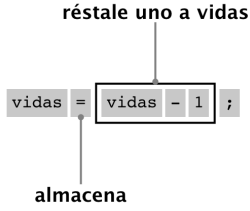


Un click más en ⏭ y la vista de la memoria se queda igual (simplemente ya no muestra `vidas` en rojo porque no cambió al ejecutar esa línea). Pero en el *Log* aparece un `3`. El procedimiento `print` se emplea para mostrar algún valor en el *Log*:

Otro click en ⏭ y la vista de la memoria cambia a:

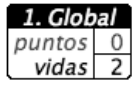


Esa línea del programa le indicó a la computadora que debía tomar el valor que estaba almacenado en la variable `vidas`, restarle uno y almacenar el resultado dentro de `vidas`:

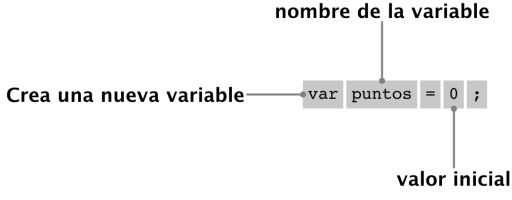


Un click más en ⏭ y muestra el nuevo valor de `vidas` en el *Log*.

Otro click en ⏭ y la siguiente línea del programa crea una nueva variable llamada `puntos`:



Aquí se puede ver que es posible asignarle un valor inicial a una variable al momento de crearla:



El resto del programa muestra el contenido de la variable `puntos`, le suma un `5` y vuelve a desplegar su contenido.

Nota

Existen reglas que indican cuales son los nombres válidos para una variable. En el Apéndice B, *Expresiones, variables, constantes, tipos, operadores y prioridades* están esas reglas, junto con más información acerca de todos los operadores aritméticos (suma, resta, multiplicación, etc.).

Ciclos

Un *ciclo* en un programa es cuando le pedimos a la computadora que ejecute varias veces las mismas instrucciones. Veamos unos ejemplos de ciclos, haz click en  y selecciona el programa `ciclos.sj` para abrirlo en el editor:

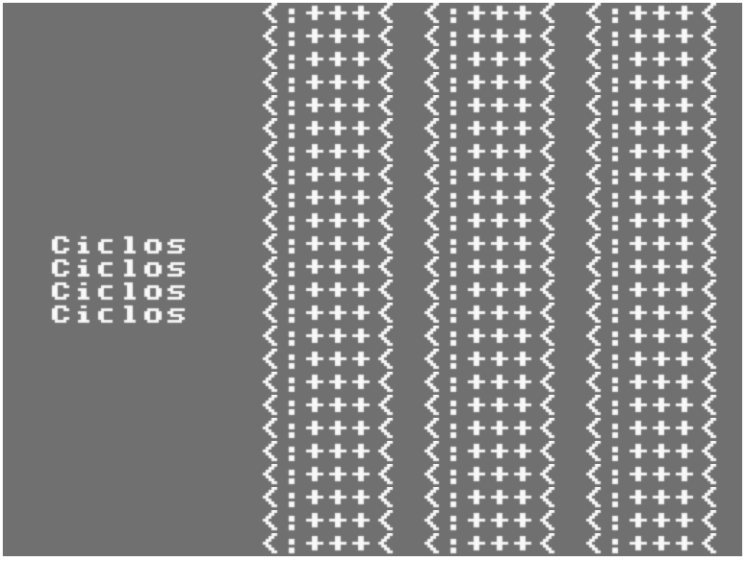
```
// Primera seccion
var contador = 0;
while (contador < 3)
  contador = contador + 1;

// Segunda seccion
contador = 0;
while (contador < 3) {
  print(contador);
  contador = contador + 1;
}

// Tercera seccion
var renglon = 10;
while (renglon < 14) {
  showAt("Ciclos", 2, renglon);
  renglon = renglon + 1;
}

// Cuarta seccion
while (renglon < 5) {
  showAt("Esto no se ejecuta", 2, renglon);
  renglon = renglon + 1;
}


// Quinta seccion
contador = 0;
while (contador < 4) {
```

```
  setForeground(0, 0, 31);
  pause(.2);
  setForeground(31, 31, 31);
  pause(.2);
  contador = contador + 1;
}

// Sexta seccion
var columna = 11;
while (columna < 32) {
  var renglon = 0;
  while (renglon < 24) {
    showAt("<:+++<", columna, renglon);
    renglon = renglon + 1;
  }
  columna = columna + 7;
}

// Fin
print("Fin");
```

Haz click en ▶ para ejecutarlo. El programa:

1. Escribe `0`, `1` y `2` en el *Log*.
2. Despliega cuatro veces la palabra `Ciclos` en la pantalla.
3. Hace que parpadeen las letras 4 veces.
4. Dibuja 3 bandas verticales de símbolos.
5. Escribe `Fin` en el *Log*.

Al terminar de ejecutarse el programa, la pantalla queda así:



El programa está dividido en seis secciones. Veamos qué hace cada una de ellas.

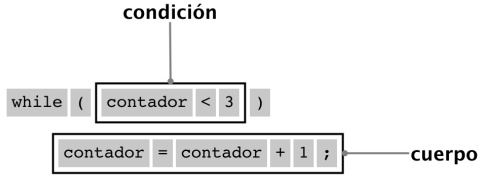
La primera sección:

```
// Primera seccion
var contador = 0;
while (contador < 3)
  contador = contador + 1;
```

Para empezar crea una variable `contador` y la inicializa en cero.

Después viene algo nuevo: `while`. El `while` se emplea dentro de un programa para decirle a la computadora que ejecute varias veces un grupo de instrucciones. A este grupo de instrucciones se le llama el *cuerpo* del `while`. ¿Cómo sabe la computadora cuántas veces debe ejecutar el cuerpo? El `while` también tiene una *condición*. Una condición es algo que puede ser cierto o falso. El cuerpo se sigue ejecutando mientras la condición sea cierta (en inglés la palabra *while* quiere decir *mientras*). Cuando un programa ejecuta varias veces el mismo grupo de instrucciones se dice que el programa *itera* sobre esas instrucciones, y que cada vez que las ejecuta es una *iteración*.

La condición se coloca, entre paréntesis, justo después de la palabra `while`. Depués de la condición viene el cuerpo. Revisemos la estructura del `while`:

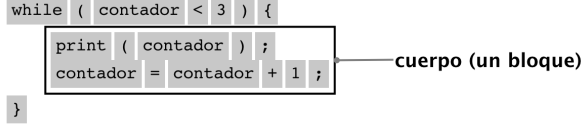


Se sigue ejecutando la instrucción del cuerpo mientras el contenido de la variable `contador` sea menor que tres. Aquí tenemos una sola instrucción dentro del cuerpo del `while` que incrementa en uno el valor almacenado en `contador`.

Usa el botón ⏭ para ejecutar la primera sección paso por paso. Fíjate cómo se regresa a checar si el valor de `contador` es inferior a tres antes de volverlo a incrementar. Cuando el valor de `contador` llega a 3 se salta el cuerpo del `while` para pasar a la siguiente sección:

```
// Segunda seccion
contador = 0;
while (contador < 3) {
  print(contador);
  contador = contador + 1;
}
```

Para tener varias instrucciones en el cuerpo de un `while` hay que ponerlas entre llaves (`{` y `}`). Así le indicas a la computadora cuáles son las instrucciones que tiene que ejecutar mientras que la condición sea cierta. A un grupo de instrucciones entre llaves se le llama un *bloque*. En esta segunda sección del programa usamos un bloque para ir desplegando en el *Log* los valores que toma la variable `contador`:



Ahora ejecuta la segunda sección paso por paso para que veas cómo funciona.

La tercera sección muestra cómo puedes usar el contenido de una variable, en este caso `renglon`, para que en cada iteración del `while` el programa despliegue la misma palabra en otra posición de la pantalla:

```
// Tercera seccion
var renglon = 10;
while (renglon < 14) {
  showAt("Ciclos", 2, renglon);
  renglon = renglon + 1;
}
```

Ejecuta esta sección paso por paso, con el botón ⏭. Fíjate que en cada iteración usa el valor de la variable `renglon` para ir desplegando la palabra `Ciclos` en los renglones 10, 11, 12 y 13.

Veamos ahora la cuarta sección:

```
// Cuarta seccion
while (renglon < 5) {
  showAt("Esto no se ejecuta", 2, renglon);
  renglon = renglon + 1;
}
```

Esta sección se parece mucho a la anterior, despliega un mensaje en la pantalla mientras que el valor de `renglon` sea menor que cinco. La primera vez que ejecutaste el programa completo (con ▶) ¡nunca desplegó el mensaje `"Esto no se ejecuta"`! Esto se debe a que, al llegar a esa parte del programa, `renglon` contiene un 14 y eso no es menor que cinco. Haz click en ⏭ y observa cómo se salta el cuerpo del `while` para ir inmediatamente a la siguiente sección del programa:

```
// Quinta seccion
contador = 0;
while (contador < 4) {
  setForeground(0, 0, 31);
  pause(.2);
  setForeground(31, 31, 31);
  pause(.2);
  contador = contador + 1;
}
```

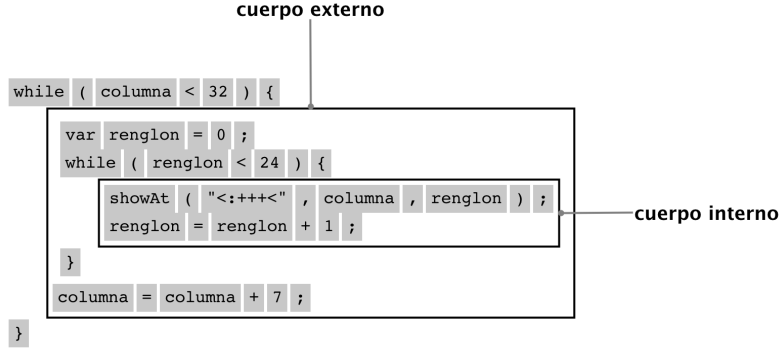
Aquí hay un bloque un poco más largo. Inicializa `contador` en cero y pasa al ciclo, en donde primero cambia el color de las letras al mismo color azul

que tiene el fondo. Las letras desaparecen, aunque siguen en la pantalla no las podemos ver. Hace una breve pausa y vuelve a poner el color de las letras en blanco. Las letras aparecen otra vez. Después de otra breve pausa incrementa en uno el valor de `contador` antes de volver a checar la condición del `while`. Ejecútala paso por paso para que veas cómo funciona.

La última sección está interesante:

```
// Sexta seccion
var columna = 11;
while (columna < 32) {
  var renglon = 0;
  while (renglon < 24) {
    showAt("<:+++<", columna, renglon);
    renglon = renglon + 1;
  }
  columna = columna + 7;
}
```

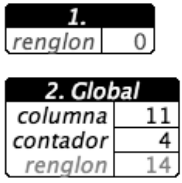
El segundo `while` está dentro del cuerpo del primero. A esto se le conoce como *ciclos anidados*:



Empieza a ejecutarlo paso por paso y observa lo que pasa cuando ejecutas la línea que dice:

```
var renglon = 0;
```

En la vista de la memoria apareció otro ambiente con una nueva variable `renglon`:



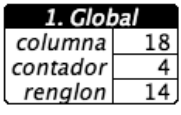
Cuando creas variables dentro de un bloque, la computadora crea un nuevo ambiente para poner ahí esas variables. Se dice que es un *ambiente local* y las variables que contiene son *variables locales*. En un programa grande las variables locales son muy útiles, te permiten definir variables sin tener que preocuparte si en alguna otra parte ya existe una variable con el mismo nombre que se esté usando para otras cosas.

Como el ambiente local es el que está al principio de la lista de ambientes visibles (es el número uno) cuando la computadora busca una variable que se llama `renglon` la que encuentra es la que está en este ambiente. Y, mientras que este ambiente sea el primero de la lista, la variable `renglon` que está en el ambiente `Global` no es visible para el programa. Por eso en la vista de la memoria aparece de color gris.

Sigue ejecutando el programa paso por paso y ve cómo va desplegando `<:+++<` en cada renglón de la pantalla. Cuando regresa a la línea que dice:

```
while (columna < 32) {
```

Vuelve a ocurrir algo interesante. La vista de la memoria muestra:



El ambiente local desapareció. Esto occurre porque el programa ya se salió del bloque que constituye el cuerpo del `while` interno. La variable `renglon` que está dentro de `Global` aparece ahora en negro, es decir, se hace visible otra vez.

Sigue ejecutando el programa paso por paso y verás que al volver a pasar por la línea:

```
var contador = 0;
```

Vuelve a crear un ambiente local para esa variable.

Sigue ejecutando el programa paso por paso y fíjate cómo ahora está haciendo una segunda columna de <:+++<, eso es porque ahora el valor de `columna` es `18`.

Ahorita, con un poco de paciencia, podrías seguir haciendo click en  hasta llegar al final del programa. Más adelante vamos a ver ciclos que se ejecutan cientos de veces. Para la computadora ese no es un problema. Ejecuta un ciclo el número de veces que sea necesario y además lo hace muy rápido. ¡Pero para una persona sería muy tedioso andar haciéndolo paso por paso! Por eso en el devkit, cuando está en el modo de ejecución paso por paso, hay una manera de pedirle que una parte del programa se ejecute más rápido. Si haces un doble click en una línea del programa, la computadora ejecuta todas las instrucciones hasta que llega a esa línea.

Pruébalo, haz un doble click sobre la línea que dice:

```
print("Fin");
```

La computadora termina de llenar las columnas con <:+++< y se detiene antes de ejecutar esa línea. Un click más en  , aparece `Fin` en el *Log* y termina de ejecutarse el programa.

Procedimientos

Un *procedimiento* es una serie de instrucciones con un nombre. Podemos usar ese nombre cada vez que queremos que la computadora ejecute esas instrucciones. Ya hemos usado algunos procedimientos predefinidos tales como `showAt` y `pause`. Ahora vamos a ver cómo definir nuestros propios procedimientos. Haz click en y selecciona el programa `procedimientos.sj` para abrirlo en el editor:

```
// Primera seccion
parpadea() {
  var i = 0;
  while (i < 4) {
    setForeground(0, 0, 31);
    pause(0.2);
    setForeground(31, 31, 31);
    pause(0.2);
    i = i + 1;
```

```
  }
}

showAt("simpleJ", 12, 4);
parpadea();
showAt("Procedimientos", 8, 6);
parpadea();

// Segunda seccion
multiShowAt(mensaje, x, y, cuantos) {
  var i = 0;
  while (i < cuantos) {
    showAt(mensaje, x, y + i);
    i = i + 1;
  }
}

multiShowAt("Hola", 2, 8, 3);
multiShowAt("Adios", 12, 8, 4);

// Tercera seccion
porLaDerecha(mensaje, x, y, tiempo) {
  var i = 31;
  while (i >= x) {
    showAt(mensaje, i, y);
    pause(tiempo);
    i = i - 1;
  }
}

porLaDerecha("Uno", 25, 8, .1);

// Cuarta seccion
nuevoPorLaDerecha(mensaje, x, y, tiempo) {
  var i = 31;
  mensaje = mensaje + " ";
  while (i >= x) {
    showAt(mensaje, i, y);
    pause(tiempo);
    i = i - 1;
  }
}

nuevoPorLaDerecha("Uno", 25, 9, .1);
```

```
// Quinta seccion
porLaIzquierda(mensaje, x, y, tiempo) {
  var i = length(mensaje);
  i = -i;
  while (i < x) {
    showAt(" " + mensaje, i, y);
    pause(tiempo);
    i = i + 1;
  }
}

porLaIzquierda("Dos", 10, 15, .1);

// Fin
print("Fin");
```

Haz click en ▶ para ejecutarlo. El programa:

1. Despliega `simpleJ` en la pantalla.
2. Hace parpadear 4 veces `simpleJ`.
3. Despliega `Procedimientos` en la pantalla.
4. Hace parpadear 4 veces `simpleJ` y `Procedimientos`.
5. Despliega tres veces `Hola` en la pantalla.
6. Despliega cuatro veces `Adios` en la pantalla.
7. Hace que entre la palabra `Uno` por la derecha de la pantalla (dejando varias copias de la letra `o` al final).
8. Hace que entre otra vez la palabra `Uno` por la derecha de la pantalla (esta vez sin dejar basura al final).
9. Hace que entre la palabra Dos por la izquierda de la pantalla.
10. Despliega `Fin` en el *Log*.

Al terminar de ejecutarse el programa la pantalla queda así:

Vamos a ver cómo funciona cada una de las secciones de este programa. La primera sección es:

```
// Primera seccion
parpadea() {
  var i = 0;
  while (i < 4) {
    setForeground(0, 0, 31);
    pause(0.2);
    setForeground(31, 31, 31);
    pause(0.2);
    i = i + 1;
  }
}

showAt("simpleJ", 12, 4);
parpadea();
showAt("Procedimientos", 8, 6);
parpadea();
```

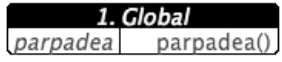
Haz click en ⏭ para pasar a modo de ejecución paso por paso. Haz click en ⏭ una vez más para ejecutar la línea:

```
parpadea() {
```

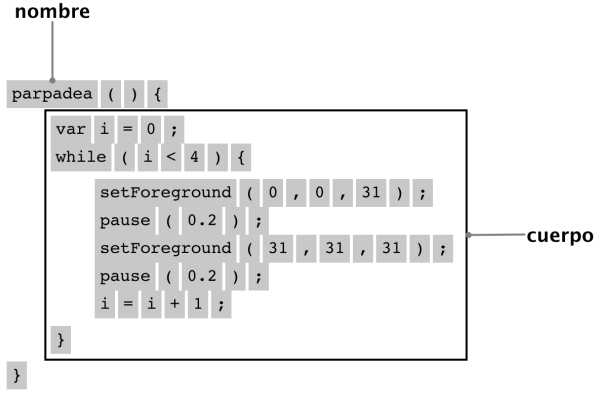
¡La computadora se salta varias líneas! Ahora está resaltada sobre fondo amarillo la línea que dice:

```
showAt("simpleJ, 12, 14);
```

Y la vista de la memoria muestra que hay una nueva variable llamada `parpadea`:



La línea que ejecutamos es el principio de una definición de procedimiento. Un procedimiento tiene un nombre y un cuerpo:



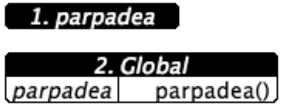
Cuando la computadora ejecuta la definición de un procedimiento almacena el cuerpo del procedimiento dentro de una variable con el nombre del procedimiento. En la vista de la memoria, no muestra como contenido de la variable el cuerpo del procedimiento con todas sus instrucciones porque ocuparía mucho espacio. En vez de eso muestra el nombre del procedimiento seguido de abrir y cerrar paréntesis: `parpadea()`.

Definir un procedimiento te permite almacenar una serie de instrucciones y ponerle un nombre. Depués puedes usar ese nombre para que se ejecuten esas instrucciones. Esto permite ejecutar esas mismas instrucciones en diferentes partes del programa sin tener que volver a escribirlas.

Haz click en ▶|, el `showAt` despliega `simpleJ` en la pantalla. Ahora estás a punto de ejecutar la línea:

```
parpadea();
```

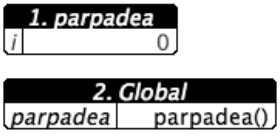
Esta línea le dice a la computadora que debe ejecutar las instrucciones almacenadas en `parpadea`. Esto se conoce como *llamar a un procedimiento*. Haz click otra vez en ▶|, el programa salta al principio del procedimiento `parpadea` y la vista de la memoria muestra:



Cuando llamas a un procedimiento se crea un ambiente local con el nombre de ese procedimiento. Usa el botón ▶| para ir ejecutando el procedimiento. Fíjate cómo al ejecutar la línea

```
var i = 0;
```

la variable `i` se crea dentro del ambiente local de `parpadea`:



Cada vez que creas una variable, ésta se almacena dentro del primer ambiente en la lista de ambientes visibles, el que se muestra con el número `1` en la vista de la memoria. Se dice que este es el *ambiente actual*.

Sigue ejecutando el procedimiento paso por paso. Usa la variable `i` como un contador para cambiar el color de las letras al color del fondo y de regreso a blanco cuatro veces. Al terminar de ejecutar el procedimiento se regresa a donde se había quedado cuando llamó al procedimiento. Ahora está en la línea:

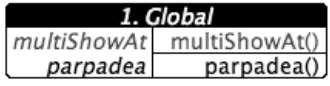
```
showAt("Procedimientos", 8, 6);
```

Sigue ejecutando paso por paso y observa cómo, después de desplegar `Procedimientos` en la pantalla, vuelve a llamar al procedimiento `parpadea`.

Cuando terminas de ejecutar paso por paso el procedimiento, la computadora se regresa a donde se quedó en el programa. Al principio de la segunda sección:

```
// Segunda seccion
multiShowAt(mensaje, x, y, cuantos) {
  var i = 0;
  while (i < cuantos) {
    showAt(mensaje, x, y + i);
    i = i + 1;
  }
}

multiShowAt("Hola", 2, 8, 3);
multiShowAt("Adios", 12, 8, 4);
```

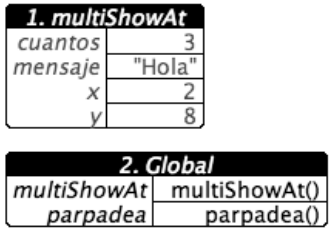
Aquí vamos a definir el procedimiento `multiShowAt`. Pero esta vez, entre los paréntesis después del nombre del procedimiento, tenemos una lista de nombres separados por comas: `mensaje`, `x`, `y` y `cuantos`. Son los *parámetros* del procedimiento. Haz click en ⏭ para ejecutar la definición del procedimiento. La vista de la memoria queda así:



Aunque `multiShowAt` es un procedimiento con parámetros, en la vista de la memoria no muestra los parámetros. Te has de estar preguntando ¿para qué sirven los parámetros? Eso es justamente lo que vamos a ver ahora. La computadora está a punto de ejecutar la línea que dice:

```
multiShowAt("Hola", 2, 8, 3);
```

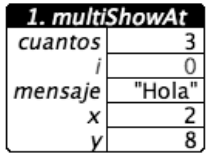
Esa instrucción es para decirle a la computadora que quieres llamar al procedimiento `multiShowAt`. Y como ese procedimiento fue definido con cuatro parámetros, al llamarlo hay que pasarle cuatro valores. En este caso lo estamos llamando con los valores: `"Hola"`, `2`, `8` y `3`. Haz click en ⏭ para que se ejecute el llamado. La vista de la memoria muestra:



Ahora, en vez de crear un ambiente vacío para el procedimiento, la computadora empleó los cuatro valores que le pasamos al llamar al procedimiento para inicializar cuatro variables con los nombres de sus parámetros. Ejecuta el procedimiento paso por paso. Al pasar por la línea que dice:

```
var i = 0;
```

La vista de la memoria queda así:



La variable local `i` se crea dentro del mismo ambiente que las variables usadas para los parámetros. Los parámetros son variables locales; que simplemente se crean e inicializan al llamar al procedimiento con unos valores. A estos valores se les conoce como *argumentos*. Sigue ejecutando el procedimiento paso por paso y observa cómo el parámetro `mensaje` contiene el mensaje a desplegar, `x` contiene la posición horizontal (columna), `y` contiene la posición vertical (renglón) inicial y `cuantos` el número de veces que se debe desplegar el mensaje.

Al terminar de ejecutar el procedimiento, la computadora se regresa a la línea:

```
multiShowAt("Adios", 12, 8, 4);
```

Ahí vuelve a llamar al procedimiento `multiShowAt` con otros argumentos. Ejecútalo paso por paso para que veas cómo ahora se ejecutan las mismas instrucciones con otros valores. Al terminar de ejecutar el procedimiento llegamos a la tercera sección del programa:

```
// Tercera seccion
porLaDerecha(mensaje, x, y, tiempo) {
  var i = 31;
  while (i >= x) {
    showAt(mensaje, i, y);
    pause(tiempo);
    i = i - 1;
  }
}

porLaDerecha("Uno", 25, 8, .1);
```

El procedimiento `porLaDerecha` es para que un texto entre moviéndose a la pantalla por la derecha. Sus argumentos son:

1. `mensaje`: el texto a desplegar.

2. `x`: la posición horizontal (columna) final en la pantalla.

3. `y`: la posición vertical (renglón) en la pantalla.

4. `tiempo`: duración de la pausa para la animación.

Este procedimiento emplea una variable `i` que va decrementado en cada iteración de un ciclo, mientras que sea mayor o igual a `x`, para primero desplegar `Uno` en la columna 31, depués en la 30 y así succesivamente hasta que lo despliega en la columna `x`. Ejecútalo paso por paso y fíjate cómo en cada iteración va desplegando `Uno` cada vez más a la izquierda. Al ejecutarlo también puedes ver que hay un error en el procedimiento: está dejando la última letra de la palabra `Uno` en la pantalla y por eso acaba quedando como `Unooooo`. A errores como éste, se les conoce como *bugs*.

Hay un bug cuando uno escribe un programa y al ejecutarlo resulta que no hace exactamente lo que esperábamos que hiciera. Es común que un programa que acabamos de escribir tenga bugs, eso nos pasa a todos, hasta a los programadores con mucha experiencia.

Cuando un programa tiene un bug hay que corregirlo. Para eso primero hay que entender por qué el programa no está haciendo lo que esperabamos que hiciera. Casi siempre es más o menos fácil darse cuenta de lo que está pasando. Pero a veces hay bugs que no son nada obvios. Uno puede llegar a pasarse horas tratando de entender algún comportamiento misterioso de un programa.

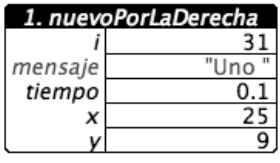
Ya que se sabe por qué ocurre el bug, hay que modificar el programa para corregirlo. En el caso del procedimiento `porLaDerecha` ya sabemos cuál es el problema: no estamos borrando la última letra del mensaje. La cuarta sección del programa tiene un procedimiento `nuevoPorLaDerecha` en donde ese bug ya está arreglado:

```
// Cuarta seccion
nuevoPorLaDerecha(mensaje, x, y, tiempo) {
  var i = 31;
  mensaje = mensaje + " ";
  while (i >= x) {
    showAt(mensaje, i, y);
    pause(tiempo);
    i = i - 1;
  }
}

nuevoPorLaDerecha("Uno", 25, 9, .1);
```

En este caso la corrección del problema está en la línea que dice:

```
mensaje = mensaje + " ";
```

Ejecuta el programa paso por paso. Después de ejecutar esa línea la vista de la memoria muestra:



La variable mensaje ahora contiene la palabra `Uno` con un espacio en blanco al final. A los textos entre comillas que se almacenan dentro de las variables se les llama *strings*. Cuando usamos el operador `+` con dos strings la computadora los une para formar un nuevo string con el texto del primero seguido inmediatamente por el texto del segundo. A esto se le llama *contatenación de strings*.

Sigue ejecutando el procedimiento paso por paso y fíjate cómo ahora ya no estamos dejando en la pantalla la última letra del mensaje como basura. Bueno, en realidad seguimos dejando en la pantalla la última letra del mensaje pero, como esa letra es un espacio en blanco, es como si borráramos la última letra.

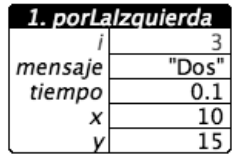
La quinta sección tiene un procedimiento que hace que el mensaje entre a la pantalla por el lado izquierdo:

```
// Quinta seccion
porLaIzquierda(mensaje, x, y, tiempo) {
  var i = length(mensaje);
  i = -i;
  while (i < x) {
    showAt(" " + mensaje, i, y);
    pause(tiempo);
    i = i + 1;
  }
}

porLaIzquierda("Dos", 10, 15, .1);
```

Ejecuta la sección paso por paso. Al ejecutar la línea que dice:

```
var i = length(mensaje);
```

La vista de la memoria muestra:

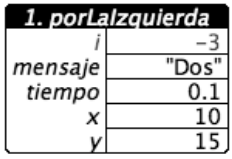


El procedimiento predefinido `length` espera como argumento un string y devuelve la longitud del string (cuántos caracteres tiene). Aquí `length` devuelve un `3` que almacenamos en la variable `i`. Este es el primer procedimiento que vemos que devuelve un valor. Más adelante veremos cómo puedes definir procedimientos que devuelven un valor. A los procedimientos que devuelven un valor se les llama *funciones*.

La línea que sigue contiene:

```
i = -i;
```

Ejecútala y la vista de la memoria queda así:



Ya habíamos visto que se podía emplear el operador `-` entre dos valores numéricos para restar el segundo del primero. También se puede usar delante de un valor numérico para obtener su complemento (si es positivo se vuelve negativo, si es negativo se vuelve positivo). Cuando ponemos el - entre dos valores numéricos para hacer una resta lo estamos usando como un operador *binario*. Cuando lo ponemos delante de un valor numérico para obtener su complemento lo estamos usando como un operador *unario*.

Para hacer una animación del texto entrando a la pantalla por el lado izquierdo tenemos que empezar por desplegar el mensaje en una columna que esté antes de la primera columna. En otras palabras, en una columna menor que cero, una columna negativa. Y la columna que seleccionamos depende de la longitud del mensaje, para un mensaje más largo hay que empezar más a la izquierda.

En la siguiente línea del procedimiento llamamos a `showAt` para desplegar el mensaje:

```
showAt(" " + mensaje, i, y);
```

Ya sabemos que hay que tener cuidado con el bug que va dejando una letra en la pantalla. En este caso, sería la primer letra en vez de la última (porque ahora el texto se mueve de izquierda a derecha). Ahora, en vez de modificar el contenido de la variable `mensaje`, estamos aprovechando para mostrar que también se le puede pasar como argumento a un procedimiento el resultado de una expresión. El primer argumento que le estamos pasando a `showAt` es el resultado de concatenar un espacio delante del mensaje.

Termina de ejecutar el programa paso por paso para que veas cómo va moviendo el texto de izquierda a derecha por la pantalla.

El ejemplo del nivel 02

Ahora ya puedes entender todo lo que está en el programa de ejemplo del nivel 02. Haz click en  y selecciona el programa `main.sj`. Ejecútalo paso por paso para asegurarte que entiendes cómo funciona.

Nivel 3: Tiles, constantes y decisiones

Interacción

Ahora vas a ver lo que son *tiles*, *constantes* y *decisiones*, y cómo usarlos para hacer un programa en el cual puedes usar las flechas del teclado para mover una bola por la pantalla.

En el menú de **Project** selecciona la opción **Switch to another project...** (**Project > Switch to another project...**) y selecciona el proyecto `Nivel_03`. Haz click en ▶ para probar el programa. Emplea las flechas del teclado para mover la bola por la pantalla. Cuando termines de probarlo haz click en ■ para detener la ejecución del programa.

Tiles

La pantalla contiene 768 posiciones organizadas en 32 columnas y 24 renglones. En cada posición de la pantalla se puede desplegar el dibujo de una letra, un dígito, un signo de puntuación o un símbolo. A estos dibujos se les llama *tiles*. La palabra *tile* (pronunciada "tayl") en inglés quiere decir *mosaico*. Existen 256 tiles diferentes, numerados del 0 al 255. Hasta ahora hemos visto que el procedimiento `showAt` coloca tiles en la pantalla para desplegar el mensaje que se le indicó.

Al hacer un videojuego es conveniente poder manipular directamente los tiles que están en la pantalla. Vamos a ver cómo se logra esto. Haz click en y selecciona el programa `tiles.sj` para abrirlo en el editor:

```
// Primera seccion
putAt(65, 20, 0);
putAt(66, 21, 0);
putAt(67, 22, 0);
pause(.5);

// Segunda seccion
putAt(97, 20, 1);
putAt(98, 21, 1);
```

```
putAt(99, 22, 1);
pause(.5);

// Tercera seccion
putAt(48, 20, 2);
putAt(49, 21, 2);
putAt(50, 22, 2);
pause(.5);

// Cuarta seccion
putAt(42, 20, 3);
putAt(43, 21, 3);
putAt(44, 22, 3);
pause(.5);

// Quinta seccion
putAt(0, 20, 4);
putAt(20, 21, 4);
putAt(127, 22, 4);
pause(.5);

// Sexta seccion
var tile = 0;
while (tile < 256) {
  var x = tile % 16;
  var y = tile / 16;
  putAt(tile, x, y);
  tile = tile + 1;
}
pause(.5);

// Septima seccion
putAt(8, 22, 9);
putAt(10, 23, 9);
putAt(25, 22, 10);
putAt(153, 23, 10);
putAt(8, 21, 11);
putAt(160, 22, 11);
putAt(160, 23, 11);
putAt(10, 24, 11);
putAt(8, 20, 12);
putAt(160, 21, 12);
putAt(160, 22, 12);
putAt(160, 23, 12);
```

```
putAt(160, 24, 12);
putAt(10, 25, 12);
pause(.5);

// Octava seccion
putAt(20, 28, 0);
var y = 0;
while (y < 23) {
  pause(.05);
  putAt(32, 28, y);
  y = y + 1;
  putAt(20, 28, y);
}
```

Haz click en ▶ para ejecutarlo. El programa:

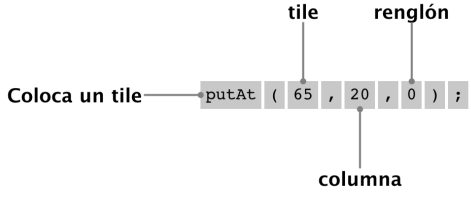
1. Despliega las letras "`A`", "`B`" y "`C`"
2. Despliega las letras "`a`", "`b`" y "`c`"
3. Despliega los números "`0`", "`1`" y "`2`"
4. Despliega los símbolos "`*`" (asterisco), "`+`" (más) y "`,`" (coma)
5. Despliega un corazón, una bola y un triángulo
6. Despliega letras, números y símbolos en un cuadro
7. Despliega un dibujo sencillo
8. Anima una bola cayendo por la pantalla

Al terminar de ejecutarse el programa, la pantalla queda así:

Ahora haz click en ⏭ para empezar a ejecutar el programa paso por paso. El programa tiene 8 secciones, la primera es:

```
// Primera seccion
putAt(65, 20, 0);
putAt(66, 21, 0);
putAt(67, 22, 0);
pause(.5);
```

Haz click una vez más en ⏭, en la pantalla aparece la letra A. El procedimiento `putAt` se emplea para colocar un tile en la pantalla:



El tile número 65 es el que contiene el dibujo para la letra "A". Los tiles del 65 al 90 contienen los dibujos para las letras de la "A" a la "Z". Usa el botón ⏭

para ejecutar el resto de la primera sección y fíjate cómo el 66 corresponde a la letra "B" y el 67 a la letra "C".

La segunda sección es:

```
// Segunda seccion
putAt(97, 20, 1);
putAt(98, 21, 1);
putAt(99, 22, 1);
pause(.5);
```

Los dibujos de las letras minúsculas, de la "a" a la "z", están en los tiles del 97 al 122. Con el botón ejecuta la segunda sección paso por paso y comprueba que el tile número 97 corresponde a la "a", el 98 a la "b" y el 99 a la "c".

La tercera sección es:

```
// Tercera seccion
putAt(48, 20, 2);
putAt(49, 21, 2);
putAt(50, 22, 2);
pause(.5);
```

Los dibujos de los dígitos del "0" al "9" están en los tiles del 48 al 57. Con el botón ejecuta la tercera sección paso por paso y fíjate cómo el tile número 48 corresponde al "0", el 49 al "1" y el 50 al "2".

La cuarta sección es:

```
// Cuarta seccion
putAt(42, 20, 3);
putAt(43, 21, 3);
putAt(44, 22, 3);
pause(.5);
```

Con el botón ejecútala paso por paso. Puedes ver que el tile 42 corresponde a un "*" (asterisco), el 43 a un "+" (más) y el 44 a una "," (coma). Aunque en este ejemplo empleamos tres números consecutivos, no todos los tiles para símbolos y signos de puntuación tienen números consecutivos.

La quinta sección es:

```
// Quinta seccion
putAt(0, 20, 4);
putAt(20, 21, 4);
putAt(127, 22, 4);
pause(.5);
```

Con el botón ⏭ ejecútala paso por paso. El tile 0 es un corazón, el 20 es una bola, y el 127 es un triángulo.

La sexta sección es:

```
// Sexta seccion
var tile = 0;
while (tile < 256) {
  var x = tile % 16;
  var y = tile / 16;
  putAt(tile, x, y);
  tile = tile + 1;
}
pause(.5);
```

Esta parte del programa coloca los 256 tiles (del 0 al 255) que existen, en un cuadrado de 16 columnas por 16 renglones. Emplea una variable llamada `tile` para ir contando, dentro de un ciclo, desde el 0 hasta el 255.

Dentro del ciclo usa el operador `%` para seleccionar la posición horizontal `x` (columna) en la cual va a colocar el tile. La operación realizada por el operador `%` se llama *módulo*, y consiste en obtener el residuo de la división del número que está a la izquierda del operador entre el número que tiene a la derecha. Por lo tanto, al dividir entre 16, el residuo tiene que ser un número del 0 al 15.

Para seleccionar la posición vertical `y` (renglón) emplea el operador `/` que es el operador para la división. Como lo que estamos dividiendo es un número entero (`tile`) entre otro número entero (`16`) entonces la computadora calcula también el resultado como un número entero.

Usa el botón ⏭ para ir ejecutando la sexta sección paso por paso. Observa cómo la variable `x` va tomando los valores del 0 al 15, mientras que la variable `y` se mantiene en cero. Esto ocurre porque cero entre 16 es cero y el residuo es cero, uno entre 16 es cero y el residuo es 1, dos entre 16 es 0 y el residuo es 2. Al llegar el valor de `tiles` a 16, tenemos que 16 entre 16 es 1 y el residuo es cero. Cuando `tiles` contiene 17, al dividir entre 16 da 1 y el residuo es 1.

Con el 18, la división sigue dando 1 y el residuo es 2. Al llegar al 32 la división da 2 y el residuo 0, con el 33 el resultado de la división es otra vez 2 pero ahora el residuo es 1.

Cuando ya te quede claro cómo funciona esta sección del programa puedes dar un doble click sobre la última línea de la sección (la que dice "`pause(0.5);`") para que la computadora ejecute el resto del ciclo más rápidamente.

Fíjate cómo además de las letras, dígitos, signos de puntuación y símbolos también existen tiles que se pueden emplear para hacer algunos dibujos sencillos. La séptima sección da un ejemplo de cómo hacerlo:

```
// Septima seccion
putAt(8, 22, 9);
putAt(10, 23, 9);
putAt(25, 22, 10);
putAt(153, 23, 10);
putAt(8, 21, 11);
putAt(160, 22, 11);
putAt(160, 23, 11);
putAt(10, 24, 11);
putAt(8, 20, 12);
putAt(160, 21, 12);
putAt(160, 22, 12);
putAt(160, 23, 12);
putAt(160, 24, 12);
putAt(10, 25, 12);
pause(.5);
```

Emplea el botón ⏭ para ejecutar esta sección paso por paso. Fíjate cómo emplea los tiles para crear el dibujo.

La última sección es:

```
// Octava seccion
putAt(20, 28, 0);
var y = 0;
while (y < 23) {
  pause(.05);
  putAt(32, 28, y);
  y = y + 1;
  putAt(20, 28, y);
}
```

Esta es la parte del programa que anima la bola cayendo. Para lograr esa animación lo que hace es: dibujar la bola en la pantalla, esperar cinco centésimas de segundo, borrar la bola de la pantalla y volverla a dibujar en su nueva posición.

Para borrar la bola emplea el tile número 32 que representa un espacio en blanco. Usa el botón  para ver paso por paso cómo funciona esta animación.

Constantes

Ya hemos estado usando muchas constantes en los programas que hemos visto. Cuando un programa contiene algo como `putAt(20, 15, 12)`, los números 20, 15 y 12 son constantes. En muchos casos es conveniente ponerle un nombre a las constantes, con eso es más sencillo leer el programa y hacerle modificaciones.

Haz click en  y selecciona el programa `sinNombres.sj` para abrirlo en el editor:

```
putAt(20, 20, 0);
var y = 0;
while (y < 23) {
  pause(.05);
  putAt(32, 20, y);
  y = y + 1;
  putAt(20, 20, y);
}
```

Haz click en  para ejecutarlo. Anima una bola cayendo por la pantalla. Es lo mismo que la octava sección del programa que acabas de ver, con la única diferencia de que ahora la bola está en la columna 20 en lugar de la 28.

Este programa, tal como está escrito, funciona, pero tiene un par de problemas.

Primero, para alguien que está leyendo el programa no es nada fácil ver qué quieren decir todos esos números que aparecen ahí. No hay ninguna explicación. Medio en tono de broma, a números misteriosos como estos en un programa se les conoce como *números mágicos* (porque no tienen ninguna explicación). Se podrían agregar comentarios al programa explicando qué significan cada uno de estos números, pero el programa acabaría lleno de comentarios. Un programa debe estar escrito de manera que se pueda entender con un mínimo de comentarios.

Segundo, si queremos que en vez de que anime una bola, se anime un cuadrado cayendo, tendríamos que cambiar un 20 por un 160 (el tile número 160 es un cuadrado) en varias partes del programa. En este programa el número 20 aparece cinco veces, pero ¡únicamente dos de ellas representan el número del tile de la bola! En los otros tres lugares donde hay un 20, éste representa el número de una columna.

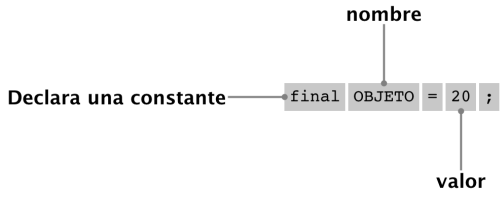
La solución correcta para estos problemas es emplear nombres para las constantes. Haz click en y selecciona el programa `conNombres.sj` para abrirlo en el editor:

```
final OBJETO = 20;
final ESPACIO = 32;

final X = 20;
final MIN_Y = 0;
final MAX_Y = 23;

putAt(OBJETO, X, MIN_Y);
var y = MIN_Y;
while (y < MAX_Y) {
  pause(.05);
  putAt(ESPACIO, X, y);
  y = y + 1;
  putAt(OBJETO, X, y);
}
```

Haz click en y comprueba que hace exactamente lo mismo que el programa anterior: anima una bola cayendo por la pantalla.

Haz click en y ejecútalo paso por paso. Al llegar a la línea que dice `putAt(OBJETO, x, MIN_Y);` la vista de la memoria muestra esto:

La palabra `final` se emplea para declarar una constante (darle un nombre):



Las constantes son muy similares a las variables. Tienen un nombre, pertenecen a un ambiente y contienen un valor. La diferencia reside en que no se puede modificar el valor que tienen almacenado. Por eso en la vista de la memoria aparecen con un pequeño candado a la izquierda del nombre.

El nombre de una constante se puede escribir con letras mayúsculas o minúsculas, pero se acostumbra siempre emplear mayúsculas, de esta manera es más fácil al leer un programa distinguir las constantes de las variables.

Termina de ejecutar el programa paso por paso, o haz click en ■, para regresar al editor. Ahora modifica las constantes `OBJETO`, `X`, `MIN_Y` y `MAX_Y` para que queden así:

```
final OBJETO = 160;
final ESPACIO = 32;

final X = 12;
final MIN_Y = 4;
final MAX_Y = 21;

putAt(OBJETO, X, MIN_Y);
var y = MIN_Y;
while (y < MAX_Y) {
  pause(.05);
  putAt(ESPACIO, X, y);
  y = y + 1;
  putAt(OBJETO, X, y);
}
```

Haz click en ▶, ahora el programa anima un cuadrado cayendo del renglón 4 hasta el 21, en la columna 12.

Controles

Para jugar un videojuego se necesitan controles y simpleJ tiene controles para dos jugadores. Los botones de los controles están en el teclado de la computadora.

Los botones para el primer jugador son las teclas: **flecha arriba**, **flecha abajo**, **flecha izquierda**, **flecha derecha**, **enter** (o **return**), **control**, **barra espaciadora** y **P**.

Los botones para el segundo jugador son las teclas: **R**, **F**, **D**, **G**, **shift**, **Z**, **X** y **Q**.

Haz click en  y selecciona el programa `controles.sj` para abrirlo en el editor:

```
showAt("Jugador 1:", 0, 0);
showAt("Jugador 2:", 0, 12);

while (true) {
  var boton = readCtrlOne();
  showAt("   ", 11, 0);  // borrar numero anterior
  showAt(boton, 11, 0);

  boton = readCtrlTwo();
  showAt("   ", 11, 12); // borrar numero anterior
  showAt(boton, 11, 12);
}
```

Haz click en  para ejecutar el programa. En la pantalla se ve:

Ahora presiona cualquiera de las teclas que corresponden a los botones de los controles. Al presionar una de esas teclas el cero que aparece a la izquierda de "`Jugador 1`", o a la izquierda de "`Jugador 2`" para los botones del segundo control, cambia a otro número. Al soltar la tecla, el número vuelve a ser cero.

El programa emplea las funciones predefinidas (una función es un procedimiento que devuelve un valor) `readCtrlOne` y `readCtrlTwo` para detectar qué botones están presionados. Cuando ningún botón está presionado estas funciones devuelven un cero. Cada botón tiene asignado un número, si un botón está presionado cuando llamas alguna de estas funciones entonces te devuelve el número correspondiente.

Estos son los números que corresponden a cada botón:

Tabla 3.1. Números para cada botón

Número	Jugador 1	Jugador 2
1	**flecha arriba**	**R**
2	**flecha abajo**	**F**
4	**flecha izquierda**	**D**
8	**flecha derecha**	**G**
16	**enter (o return)**	**shift**
32	**control**	**Z**
64	**barra espaciadora**	**X**
128	**P**	**Q**

Si presionas más de un botón obtienes la suma de sus números. Prueba presionar simultáneamente **flecha arriba** y **flecha izquierda**: te muestra un 5, que es 1 + 4 (número de **flecha arriba** + número de **flecha izquierda**).

Ahora haz click en ■ para detener la ejecución del programa. Y después ejecútalo paso por paso usando el botón ⏭.

Las primeras dos líneas simplemente despliegan los mensajes "`Jugador 1`:" y "`Jugador 2:`":

```
showAt("Jugador 1:", 0, 0);
showAt("Jugador 2:", 0, 12);
```

La línea siguiente inicia un ciclo:

```
while (true) {
```

Este es un ciclo que se ejecuta "para siempre", `true` es una constante cuyo valor siempre es verdadero (*true* en inglés quiere decir *verdadero*). Existe otra constante, `false` cuyo valor siempre es falso (*false* en inglés quiere decir *falso*). Estas dos constantes, `true` y `false`, son constantes *booleanas*. No existe ninguna otra constante booleana.

Al ejecutar la línea:

```
var boton = readCtrlOne();
```

Llama a la función `readCtrlOne` y almacena el valor que devuelve en una variable llamada `boton`. Al ejecutarlo paso por paso te va a devolver un cero porque no tienes ningún botón presionado.

Las dos líneas siguientes:

```
showAt("   ", 11, 0);  // borrar numero anterior
showAt(boton, 11, 0);
```

Primero usan espacios en blanco para borrar el número anterior de la pantalla (como esta es la primera vez todavía no hay nada que borrar), y después despliega el valor de la variable `boton`.

Las siguientes tres líneas:

```
boton = readCtrlTwo();
showAt("   ", 11, 12); // borrar numero anterior
showAt(boton, 11, 12);
```

Hacen lo mismo pero para el segundo control.

Sigue ejecutándolo paso por paso y, antes de volver a ejecutar la línea en donde llama a la función `readCtrlOne`, presiona **flecha izquierda** y *manténla apoyada mientras vuelves a hacer click en* ⏭. La variable `boton` ahora contiene un 4 que es el número que corresponde al botón **flecha izquierda**.

Cuando termines de probar el programa recuerda hacer un click en ■ para detener su ejecución.

Decisiones

Un programa para un videojuego debe poder hacer cosas diferentes de acuerdo a los botones del control que presione el jugador. Para lograr esto se necesitan *decisiones*. Una decisión es algo como "*Si* el botón flecha arriba está presionado, *entonces* mueve al personaje hacia arriba". Una decisión tiene dos partes: la condición (el botón flecha arriba está presionado) y el cuerpo (mueve el personaje hacia arriba). El cuerpo se ejecuta únicamente si la condición es cierta.

Haz click en 📂 y selecciona el programa `decisiones.sj` para abrirlo en el editor:

```
var i = 1;
while (i < 4) {
```

```
  if (i == 1)
    print("uno");
  if (i == 2)
    print("dos");
  if (i == 3)
    print("tres");
  i = i + 1;
}
```

Haz click en ▶ para ejecutar el programa. Despliega en el Log: `uno`, `dos` y `tres`.

Ahora haz click en ⏭ para ejecutarlo paso por paso. Las primeras dos líneas:

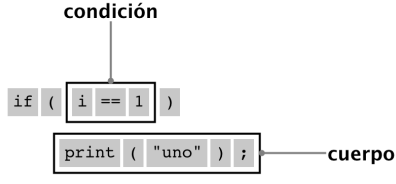
```
var i = 1;
while (i < 4) {
```

Simplemente declaran una variable `i,` que se inicializa en cero, e inician un ciclo que se va a ejecutar mientras que el valor de `i` sea inferior a cuatro.

El siguiente par de líneas, dentro del cuerpo del `while`, dicen:

```
if (i == 1)
  print("uno");
```

El `if` se emplea para decidir si una parte del programa se debe de ejecutar. La palabra *if* en ingles significa *si*. Al igual que el `while`, el `if` tiene una condición y un cuerpo:



En este caso, su significado es: si el valor de `i` es igual a uno entonces despliega "`uno`" en el Log. El doble signo de igual (==) se emplea para comparar si dos valores son iguales. De la misma manera que con el `while`, para poner varias instrucciones dentro del cuerpo es necesario agruparlas dentro de un bloque colocandolas entre llaves (`{` y `}`).

Ejecuta esta parte del programa paso por paso y observa cómo, después de checar que `i` es igual a uno, despliega "`uno`" en el Log.

El resto del programa contiene:

```
  if (i == 2)
    print("dos");
  if (i == 3)
    print("tres");
  i = i + 1;
}
```

Escribe en el Log "`dos`" si el valor de `i` es igual a dos o "`tres`" si su valor es igual a tres. Después incrementa en uno el valor de `i` y, como ya es el final del cuerpo del ciclo, se regresa al inicio del ciclo para volver a checar su condición.

Sigue ejecutando el programa paso por paso. Fíjate cómo, ya que el valor de `i` no es ni dos ni tres, se salta el cuerpo de las dos condiciones siguientes.

Al ejecutar por segunda vez el cuerpo del ciclo, el valor de `i` ya es dos. Continua ejecutando el programa paso por paso y observa cómo en esta segunda iteración únicamente ejecuta el cuerpo del segundo `if`. Y, en la tercera iteración, únicamente ejecuta el cuerpo del último `if`.

Mover una bola por la pantalla

Ahora vamos a hacer un programa con el cual puedes usar las flechas del teclado para mover una bola por la pantalla. Vamos a aprovechar para cambiar un poco la manera en la que vemos los programas. Hasta ahora lo que hemos hecho es mostrarte un programa y pedirte que lo ejecutes para que veas lo que hace; después lo ejecutabas paso por paso y te íbamos explicando cómo funcionaba. Ahora vamos a mostrarte el proceso que se sigue al escribir un programa.

Una manera de hacer que se puedan usar las flechas del teclado para mover una bola por la pantalla es haciendo un programa que siga estos pasos:

1. Dibujamos la bola en la pantalla

2. Hacemos una pausa

3. Borramos la bola de la pantalla

4. Vemos que boton esta apoyado y lo usamos para calcular la nueva posicion de la bola

5. Regresamos al paso (1)

Esto mismo lo podemos escribir en el editor como comentarios:

```
// 1. Dibujamos la bola en la pantalla
// 2. Hacemos una pausa
// 3. Borramos la bola de la pantalla
// 4. Vemos que boton esta apoyado
//    y lo usamos para calcular la nueva
//    posicion de la bola
// 5. Regresamos al paso (1)
```

Queremos que el programa se quede ejecutando "para siempre" los pasos del (1) al (4). Para hacer eso ya sabemos que podemos emplear un `while` con una condición que siempre es verdadera. Y ya no necesitamos numerar los pasos porque eso únicamente lo estabamos empleando para poder decir: "Regresamos al paso (1)". Es decir que lo podemos escribir así:

```
// Para "siempre"
while (true) {
  // Dibujamos la bola en la pantalla
  // Hacemos una pausa
  // Borramos la bola de la pantalla
  // Vemos que boton esta apoyado
  // y lo usamos para calcular la nueva
  // posicion de la bola
}
```

La computadora necesita, para poder dibujar y borrar la bola, saber cuál es su posición actual en la pantalla. Para eso podemos definir dos variables: una variable `xBola` para su posición horizontal y una variable `yBola` para su posición vertical. Teniendo estas dos variables, ya sabemos cómo dibujar y borrar la bola. También sabemos cómo hacer una pausa. Haciéndole estos cambios al programa, éste queda así:

```
var xBola = 16;
var yBola = 12;

// Para "siempre"
while (true) {
  // Dibujamos la bola en la pantalla
```

```
  putAt(20, xBola, yBola);

  // Hacemos una pausa
  pause(0.1);

  // Borramos la bola de la pantalla
  putAt(32, xBola, yBola);

  // Vemos que boton esta apoyado
  // y lo usamos para calcular la nueva
  // posicion de la bola
}
```

Inicializando `xBola` en 16 y `yBola` en 12 la bola queda inicialmente, más o menos, al centro de la pantalla. Al colocar el tile número 20 en la pantalla dibujamos una bola, y al reemplazarlo por el tile número 32, que es un espacio en blanco, la borramos.

Lo que sigue es ver qué botón está apoyado y usarlo para calcular la nueva posición de la bola en la pantalla. Ya sabemos que podemos emplear el procedimiento predefinido `readCtrlOne` para ver si un botón del control está presionado. Si ningún botón está presionado, entonces `readCtrlOne` devuelve un cero. Pero si está presionado alguno de **flecha arriba**, **flecha abajo**, **flecha izquierda** o **flecha derecha** entonces nos devuelve un uno, un dos, un cuatro o un ocho, respectivamente. Lo que podemos hacer es almacenar el valor devuelto por `readCtrlOne` en una variable, después comparar el valor almacenado en esta variable con los valores posibles para los botones de las flechas y, con base en eso, decidir la nueva posición de la bola. Al hacer esto, el programa queda así:

```
var xBola = 16;
var yBola = 12;

// Para "siempre"
while (true) {
  // Dibujamos la bola en la pantalla
  putAt(20, xBola, yBola);

  // Hacemos una pausa
  pause(0.1);

  // Borramos la bola de la pantalla
  putAt(32, xBola, yBola);
```

```
  // Vemos que boton esta apoyado
  var boton = readCtrlOne();

  // y lo usamos para calcular la nueva
  // posicion de la bola
  if (boton == 1)
    yBola = yBola - 1;
  if (boton == 2)
    yBola = yBola + 1;
  if (boton == 4)
    xBola = xBola - 1;
  if (boton == 8)
    xBola = xBola + 1;
}
```

Lo que hace para calcular la nueva posición de la bola es:

1. Si el botón es **flecha arriba** (uno) entonces decrementa en uno la posición vertical de la bola (recuerda que la posición vertical de la bola es el número del renglón en el cual la dibuja; entre más pequeño sea este número, más alto en la pantalla se dibuja la bola).

2. Si el botón es **flecha abajo** (dos) entonces incrementa en uno la posición vertical de la bola.

3. Si el botón es **flecha izquierda** (cuatro) entonces decrementa en uno la posición horizontal de la bola.

4. Si el botón es **flecha derecha** (ocho) entonces incrementa en uno la posición horizontal de la bola.

Este programa ya está dentro del `Nivel_03` con el nombre `mueveBola.sj`. Usa el botón para abrirlo en el editor y después haz click en para que veas cómo puedes emplear las flechas del teclado para mover la bola por la pantalla.

Pero este programa tiene un bug: ¿Qué pasa cuando llevas la bola hasta el borde de la pantalla? ¡Se sale de la pantalla! Esto ocurre porque en el programa nunca checamos, antes de cambiar la posición de la bola, si ya está en el borde de la pantalla.

Una manera de corregir este bug es empleando un `if` anidado dentro de otro. Por ejemplo:

```
if (boton == 1)
  if (yBola > 0)
    yBola = yBola - 1;
```

Esto quiere decir: si el botón es flecha arriba entonces si la posición vertical de la bola es mayor que cero, entonces decrementa en uno su posición vertical. Con un poco de trabajo se entiende, pero no es muy fácil de leer. Normalmente diríamos algo como: si el botón es flecha arriba y la posición vertical de la bola es mayor que cero, entonces decrementa en uno su posición vertical; que es más fácil de leer. Podemos escribir lo mismo en un programa, para eso empleamos el operador &&, este operador se llama *and* (la palabra *and* en inglés significa *y*). Este operador se emplea para crear una condición compuesta a partir de dos condiciones sencillas, de la manera siguiente:

```
condicion1 && condicion2
```

Esta es una condición compuesta, que es verdadera únicamente cuando *condición1 y condicion2* son verdaderas simultáneamente.

Emplea el botón  para abrir el programa `mueveBolaCorregido.sj` en donde ya se emplea el operador *and* para checar que la bola no se salga de la pantalla:

```
var xBola = 16;
var yBola = 12;

// Para "siempre"
while (true) {
  // Dibujamos la bola en la pantalla
  putAt(20, xBola, yBola);

  // Hacemos una pausa
  pause(0.1);

  // Borramos la bola de la pantalla
  putAt(32, xBola, yBola);

  // Vemos que boton esta apoyado
  var boton = readCtrlOne();
```

```
  // y lo usamos para calcular la nueva
  // posicion de la bola
  if (boton == 1 && yBola > 0)
    yBola = yBola - 1;
  if (boton == 2 && yBola < 23)
    yBola = yBola + 1;
  if (boton == 4 && xBola > 0)
    xBola = xBola - 1;
  if (boton == 8 && xBola < 31)
    xBola = xBola + 1;
}
```

Haz click en  para ejecutarlo, y comprueba que la bola ya no se sale de la pantalla.

Este programa es un poco difícil de leer porque está lleno de números mágicos por todos lados. El programa `main.sj` es este mismo programa, pero ya empleando nombres para las constantes. Usa el botón  para abrir `main.sj` en el editor.

Esta versión del programa además despliega unos mensajes para explicar que se deben emplear las flechas del teclado para mover la bola por la pantalla. Al mover la bola sobre los mensajes los va borrando ¿Puedes modificar el programa para que la bola ya no borre los mensajes? Una pista: hay que volver a dibujar los mensajes después de borrar la bola, pero antes de volverla a dibujar.

Nivel 4: Principio de un juego

Se puede gradualmente mejorar un programa

Se pueden ir haciendo cambios al programa que viste en el nivel 03 hasta convertirlo en un juego completo. En este nivel le vamos a agregar comida. Cada vez que te comes la comida te da un punto y aparece más comida en algún otro lugar de la pantalla. En el siguiente nivel (el 04), para que ya sea un juego completo, le agregaremos vidas y un enemigo que intenta comerse a tu personaje.

En el menú de **Project** selecciona la opción **Switch to another project...** (**Project > Switch to another project...**) y selecciona el proyecto `Nivel_04`. Haz click en ▶ para probar el programa. Al igual que en el nivel anterior usas las flechas para mover tu personaje, que es una bola, por la pantalla. Pero ahora hay comida, un signo de más, que puedes comer para ir ganando puntos.

Cada vez que te comes la comida se escucha un sonido (necesitas tener bocinas conectadas a tu computadora para escucharlo). En ciertas computadoras puede que el sonido no se escuche al instante de comer la comida, se escucha un poco después. Si esto ocurre en tu computadora entonces en el menú de **Audio** selecciona la opción **Enable audio delay workaround** (**Audio > Enable audio delay workaround**) para corregir este problema. Baja la calidad del audio, pero al menos ya se escucha en el momento indicado.

Conviene partir un programa en procedimientos

Al final del nivel 03 el ciclo principal del programa quedó así:

```
// Para "siempre"
while (true) {
  // Dibujamos la bola
  putAt(BOLA, xBola, yBola);

  // Esperamos una decima de segundo
  pause(0.1);

  // Borramos la bola
```

```
  putAt(ESPACIO, xBola, yBola);

  // Vemos que boton esta apoyado
  var boton = readCtrlOne();

  // y lo usamos para calcular la nueva
  // posicion de la bola
  if (boton == BOTON_ARRIBA && yBola > MIN_Y)
    yBola = yBola - 1;
  if (boton == BOTON_ABAJO && yBola < MAX_Y)
    yBola = yBola + 1;
  if (boton == BOTON_IZQUIERDA && xBola > MIN_X)
    xBola = xBola - 1;
  if (boton == BOTON_DERECHA && xBola < MAX_X)
    xBola = xBola + 1;
}
```

Está más o menos sencillo y tiene unos comentarios que indican qué hace cada parte. Como ya mencionamos antes, poner comentarios es una buena idea pero hay que mantenerlos a un mínimo. Conviene más escribir un programa que se pueda entender sin necesidad de tantos comentarios. Vamos a ver cómo se pueden emplear procedimientos para lograr esto. De una vez vamos a aprovechar para cambiar el programa y a la bola llamarle *personaje*, que describe mejor su función dentro del juego. El primer cambio es modificar el nombre de la constante que estamos empleando. En vez de:

```
final BOLA = 20;
```

Ahora vamos a poner:

```
final PERSONAJE = 20;
```

Y también cambiamos los nombres de las variables que empleamos para almacenar su posición. En vez de:

```
var xBola = 16;
var yBola = 12;
```

Ahora tenemos:

```
var xPersonaje = 16;
var yPersonaje = 12;
```

El siguiente cambio es modificar el ciclo principal para que en vez de hacer todo el trabajo ahí mismo, le delegue parte a otros procedimientos. El ciclo principal queda así:

```
// Para "siempre"
while (true) {
  dibujaPersonaje();
  pause(0.1);
  borraPersonaje();
  var boton = readCtrlOne();
  muevePersonaje(boton);
}
```

Ahora es más fácil de leer. Ya no necesitamos los comentarios porque los nombres de los procedimientos nos dan una idea de lo que hace cada uno de ellos. Además, en un momento que empecemos a hacerle mejoras, va a ser más sencillo entender los cambios que iremos haciendo. Lo que falta ahora es escribir esos procedimientos.

Nota

Aunque primero hicimos este cambio en el ciclo principal y después vamos a escribir los procedimientos, en el programa ya terminado tenemos que asegurarnos de colocar los procedimientos antes del programa principal. Porque, al ejecutarse un programa, cada procedimiento tiene que definirse antes de poder llamarlo desde otra parte del programa.

Los procedimientos `dibujaPersonaje` y `borraPersonaje` son muy sencillos:

```
dibujaPersonaje() {
  putAt(PERSONAJE, xPersonaje, yPersonaje);
}

borraPersonaje() {
  putAt(ESPACIO, xPersonaje, yPersonaje);
}
```

El procedimiento `muevePersonaje` tampoco tiene nada de complicado:

```
muevePersonaje(boton) {
  if (boton == BOTON_ARRIBA && yPersonaje > MIN_Y)
```

```
    yPersonaje = yPersonaje - 1;
  if (boton == BOTON_ABAJO && yPersonaje < MAX_Y)
    yPersonaje = yPersonaje + 1;
  if (boton == BOTON_IZQUIERDA && xPersonaje > MIN_X)
    xPersonaje = xPersonaje - 1;
  if (boton == BOTON_DERECHA && xPersonaje < MAX_X)
    xPersonaje = xPersonaje + 1;
}
```

El archivo `conProcedimientos.sj` contiene el programa ya con estos cambios. Usa el botón  para abrirlo en el editor para que veas cómo quedó después de partir el ciclo principal en varios procedimientos.

Números al azar

A veces en un programa es necesario poder generar números al azar. Para las mejoras que le vamos a hacer al programa necesitamos poder colocar la comida al azar en la pantalla. Vamos a ver cómo se emplea el procedimiento predefinido `random` para generar números al azar.

Haz click en el botón  y abre el archivo `random.sj`. Ahora haz click en  para ejecutarlo. La pantalla se llena de puntos que andan apareciendo y despareciendo al azar.

Este es el programa:

```
final PUNTO = 46;
final ESPACIO = 32;

setBackground(0, 0, 0);

while (true) {
  var x = random(32);
  var y = random(24);
  putAt(PUNTO, x, y);

  x = random(32);
  y = random(24);
  putAt(ESPACIO, x, y);
}
```

Empieza definiendo dos constantes, una para el tile de un punto y otra para el tile de un espacio. Pone la pantalla en negro y entra a un ciclo infinito. Dentro

del ciclo emplea el procedimiento predefinido `random` para colocar puntos y espacios al azar en la pantalla (a veces al colocar un espacio borra un punto que estaba ahí).

Cada vez que llamas al procedimiento `random` te devuelve un número generado al azar. El número que le pasas como argumento a `random` sirve para especificar el rango dentro del cual debe estar el número que genera al azar. El valor que devuelve `random` siempre está entre cero y el valor que le pasaste como argumento menos uno:

Por lo tanto, las dos líneas:

```
var x = random(32);
var y = random(24);
```

Crean una variable `x` y la inicializan con un número al azar entre 0 y 31 (para la posición horizontal del punto en la pantalla), y después crean una variable `y` que inicializan con un valor entre 0 y 23 (para la posición vertical del punto en la pantalla).

Emplea el botón para ejecutar el programa paso por paso parq que veas cómo cada vez que llamas al procedimiento `random` te devuelve números diferentes.

Sonidos

La manera más sencilla de generar sonidos en simpleJ es con el procedimiento predefinido `note`. Usa el botón para abrir el programa `sonidos.sj` en el editor:

```
note("C3");
pause(.25);
note("D3");
pause(.25);
note("E3");
pause(.25);
note("F3");
pause(.25);
note("G3");
```

```
pause(.25);
note("A4");
pause(.25);
note("B4");
pause(.25);
note("C4");
pause(1);

note("C4");
pause(.25);
note("B4");
pause(.25);
note("A4");
pause(.25);
note("G3");
pause(.25);
note("F3");
pause(.25);
note("E3");
pause(.25);
note("D3");
pause(.25);
note("C3");
pause(.25);
pause(2);

note("A#5");
pause(.3);
note("A5");
pause(.3);
note("G#4");
pause(.3);
note("G4");
pause(.3);
note("F#4");
pause(.3);
note("F4");
pause(.3);
note("A#5");
pause(.3);
note("A5");
pause(.3);
note("G#4");
pause(.3);
```

```
note("G4");
pause(.3);
note("F#4");
pause(.3);
note("F4");
pause(1);

note("A1");
pause(.4);
note("G#6");
pause(.4);
```

Ahora haz click en ▶ para ejecutarlo y escucha las notas de música que genera.

El procedimiento predefinido `note` espera un argumento de tipo string que le especifica qué nota generar:

El string debe contener una letra (mayúscula) y un número. La letra le indica cuál es la nota y el número le indica cuál es la octava. Entre más pequeño sea el número, más baja es la octava (sonido más grave); entre más grande, más alta es la octava (sonido más agudo). La octava debe ser un número entero entre 1 y 6. La siguiente tabla indica la correspondencia entre las letras y las notas musicales:

Tabla 4.1. Letras y Notas

Letra	Nota
C	Do
D	Re
E	Mi
F	Fa
G	Sol
A	La
B	Si

Se puede emplear un "#" o una "b" entre la letra de la nota y el número de la octava para indicar *sostenido* o *bemol* respectivamente. Es decir que "C#4" representa un *do sostenido* y "Eb4" es un *mi bemol*.

La nota más grave es la "A1" y la más aguda es la "G#6".

Comida y puntos

Ahora ya estamos listos para agregarle al programa la comida y los puntos. En el programa `conProcedimientos.sj` el ciclo principal había quedado así:

```
// Para "siempre"
while (true) {
  dibujaPersonaje();
  pause(0.1);
  borraPersonaje();
  var boton = readCtrlOne();
  muevePersonaje(boton);
}
```

En el programa `main.sj` el ciclo principal es básicamente el mismo con unos cuantos cambios que aparecen en negritas:

```
// Para "siempre"
while (true) {
  dibujaPersonaje();
  pause(0.1);
  borraPersonaje();
  var boton = readCtrlOne();
  muevePersonaje(boton);
  if (personajeComioComida()) {
    nuevaPosicionComida();
    dibujaComida();
    puntos = puntos + 1;
    muestraPuntos();
    note("C6");
  }
}
```

Básicamente lo que hicimos fue agregarle un `if` en el cual preguntamos si el personaje se comió la comida y en caso de que esto haya ocurrido: decidimos una nueva posición para la comida, dibujamos la comida en su nueva posición,

incrementamos en uno el valor de `puntos`, mostramos la nueva puntuación en la pantalla y hacemos un sonido.

Aquí estamos empleando algunos nuevos procedimientos. Vamos a verlos uno por uno, empezando por `muestraPuntos`:

```
muestraPuntos() {
  showAt("Puntos: " + puntos, 20, 0);
}
```

Muestra en la primera línea de la pantalla el string "`Puntos: `" concatenado con el valor almacenado en la variable `puntos`. Por el principio del programa definimos la variable `puntos` de esta manera:

```
/* La puntuacion del jugador */
var puntos = 0;
```

Y, como estamos empleando la primera línea de la pantalla para mostrar los puntos, modificamos el valor de la constante `MIN_Y` para que el personaje no pueda subir hasta esa línea:

```
final MIN_Y = 1;
```

Veamos ahora los procedimientos para la comida. El procedimiento `dibuja-Comida` es muy sencillo:

```
dibujaComida() {
  putAt(COMIDA, xComida, yComida);
}
```

Al inicio del programa definimos que la constante COMIDA tenga el número correspondiente al tile con un signo de más (+):

```
final COMIDA = 43;
```

Y también declaramos unas variables para la posición horizontal y vertical de la comida en la pantalla:

```
/* La posicion de la comida en la pantalla */
var xComida;   // posicion horizontal
var yComida;   // posicion vertical
```

Esta es la definición del procedimiento `personajeComioComida`:

```
personajeComioComida() {
  return xPersonaje == xComida && yPersonaje == yComida;
}
```

Este procedimiento simplemente devuelve `true` si la posición horizontal del personaje es igual a la posición horizontal de la comida y sus posiciones verticales también son iguales, de otra manera devuelve `false`. Es decir que el valor devuelto por `personajeComioComida` es verdadero si el personaje y la comida están en la misma posición en la pantalla.

El procedimiento `nuevaPosicionComida` es un poco más interesante. Tenemos que generar una posición al azar. Para la posición horizontal es muy sencillo, basta con generar un número al azar entre 0 y 31. Pero para la posición vertical tenemos que generar un número entre 1 y 23. No puede ser un cero porque no queremos colocar la comida en el primer renglón de la pantalla; el personaje no puede subir hasta ahí. La manera más sencilla de generar un número al azar entre 1 y 23 es generar un número al azar entre 0 y 22 y después incrementarlo en uno. Esto prodría escribirse así:

```
yComida = random(23) + 1;
```

Pero aquí decidimos aprovechar las constantes `MIN_Y` y `MAX_Y` para que, si en un futuro decidiéramos cambiar el límite inferior o superior verticales, no sea necesario hacerle cambios al procedimiento `nuevaPosicionComida`:

```
nuevaPosicionComida() {
  xComida = random(32);
  yComida = random(MAX_Y - MIN_Y) + MIN_Y;
}
```

Con esto ya casi está listo el programa. Antes del ciclo principal mostramos en la pantalla las instrucciones explicando cómo emplear al programa:

```
// Mostramos las instrucciones
showAt("Usa las flechas del teclado", 1, 4);
showAt("para mover a tu personaje", 1, 5);
showAt("por la pantalla.", 1, 6);
showAt("Cada vez que te comes la", 1, 8);
showAt("comida te da un punto.", 1, 9);
showAt("Personaje:", 5, 13);
putAt(PERSONAJE, 17, 13);
showAt("Comida:", 8, 15);
putAt(COMIDA, 17, 15);
```

```
showAt("Presiona la barra de", 10, 20);
showAt("espacio para empezar", 10, 21);
```

Las instrucciones indican que hay que presionar la barra de espacio para empezar. Aquí necesitamos que el programa se espere hasta que se presione la barra de espacio. Para lograr esto, por el principio del programa definimos una constante para el "botón espacio":

```
final BOTON_ESPACIO = 64;
```

Y empleamos un `while` para esperar que se presione la barra de espacio:

```
while (readCtrlOne() != BOTON_ESPACIO)
  ;
```

Este ciclo se ejecuta mientras que el valor devuelto por el procedimiento predefinido `readCtrlOne` sea diferente de `BOTON_ESPACIO`. El operador `!=` significa *diferente de* y únicamente es cierto cuando los valores que está comparando son diferentes uno del otro. Como no nos interesa estar ejecutando ninguna otra cosa dentro de este ciclo el cuerpo del `while` sólo contiene un punto y coma, es un `while` con un *cuerpo vacío*.

Al presionar la barra de espacio, y antes de entrar al ciclo principal, se ejecutan estas instrucciones:

```
clear();
nuevaPosicionComida();
dibujaComida();
muestraPuntos();
```

Borramos la pantalla para quitar de ahí las instrucciones de cómo se emplea el programa. Le damos una posición inicial a la comida y la dibujamos en la pantalla. Finalmente, mostramos la puntuación inicial (cero), antes de entrar al ciclo principal.

Nivel 5: Un juego completo

Falta poco para tener un juego completo

En el nivel anterior (04) ya hicimos que el personaje gane puntos al comer la comida que aparece al azar en algún lugar de la pantalla. En este nivel le vamos a agregar un enemigo que intenta comerse al personaje del jugador. El personaje empieza el juego con tres vidas y cada vez que el enemigo se lo come pierde una vida; al perder su última vida se acaba el juego. Pero también obtiene una vida extra por cada diez puntos que gana.

En el menú de **Project** selecciona la opción **Switch to another project...** (**Project** > **Switch to another project...**) y selecciona el proyecto `Nivel_05`. Haz click en ▶ para probar el programa. Cuando el enemigo te come, se escucha un sonido diferente al que se escucha cuando tú te comes la comida. Además, cada vez que ganas diez puntos, al ganar otra vida, también se escucha otro sonido diferente.

Definición de funciones

Antes de empezar a hacerle las mejoras al programa vamos a ver cómo se define una función. Recuerda que una función es un procedimiento que devuelve un valor. Haz click en y abre el programa `funciones.sj` en el editor:

```
doble(x) {
  return x * 2;
}

print(doble(21));
print(doble(13));

parONon(n) {
  if (n % 2 == 0)
    return "par";
  return "non";
}

print(parONon(1));
print(parONon(6));
```

```
multiploDeTres(n) {
  if (n % 3 != 0)
    return;
  print(n + " es multiplo de 3");
}

var i = 1;
while (i < 10) {
  multiploDeTres(i);
  i = i + 1;
}
```

Haz click en ▶ para ejecutarlo. En el Log aparecen estos mensajes:

```
42
26
non
par
3 es multiplo de 3
6 es multiplo de 3
9 es multiplo de 3
```

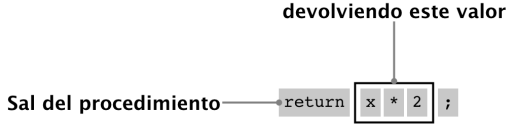
Ahora emplea el botón ▶| para ir ejecutando el programa paso por paso. Primero define un procedimiento `doble`, el cual en realidad es una función que devuelve el doble del número que se le pasó como argumento. Al ejecutar la línea que dice:

```
print(doble(21));
```

Llama a la función `doble` pasándole un 21 como argumento. Dentro del cuerpo de esta función hay una sola línea:

```
return x * 2;
```

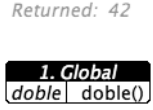
Así es como la computadora entiende esa línea:



El `return` le indica a la computadora que debe terminar de ejecutar en ese instante el procedimiento, ignorando las instrucciones que vienen después de

ella, y regresar al lugar donde fue llamado este procedimiento. En este caso devolviendo el valor de `x` multiplicado por 2.

Justo después de ejecutar esa línea, la vista de la memoria muestra:



Indicando, en color verde, que la función `doble` devolvió un `42` (el doble de 21).

Es importante notar que `return` está haciendo dos cosas: interrumpir la ejecución del procedimiento para regresar al punto de donde fue llamado y devolver un valor. Sigue ejecutando el programa paso por paso hasta llegar a la línea en donde se llama a la función `parONon`:

```
print(parONon(1));
print(parONon(6));
```

En donde la definición de `parONon` es:

```
parONon(n) {
  if (n % 2 == 0)
    return "par";
  return "non";
}
```

Con el botón ⏭ ejecuta paso por paso las 2 líneas en las cuales se llama a esta función. La función obtiene el residuo de dividir el valor que se le pasó entre dos, y checa si el resultado es igual a cero (es decir, si es un número par). Si fue igual a cero, entonces devuelve como resultado el string "`par`". Si el residuo fue diferente de cero, entonces sigue con la ejecución de la función y en la línea siguiente se topa con una instrucción indicándole que termine de ejecutarse devolviendo como valor el string "`non`".

Existe también la posibilidad de emplear `return` dentro de un procedimiento que no es una función. Es decir dentro de un procedimiento que no devuelve ningún valor. En este caso simplemente se emplea `return` seguido de un punto y coma, sin ningún valor para devolver. Un ejemplo de esto es el procedimiento `multiploDeTres`:

```
multiploDeTres(n) {
  if (n % 3 != 0)
    return;
  print(n + " es multiplo de 3");
}
```

Si el residuo al dividir el valor que se le pasó como argumento entre tres es diferente de cero (en otras palabras: no es un múltiplo de tres), entonces termina la ejecución del procedimiento y regresa. De lo contrario sigue con su ejecución y despliega en el Log el valor seguido de `" es multiplo de 3"`.

Ejecuta paso por paso las líneas en donde se llama al procedimiento `multiploDeTres`:

```
var i = 1;
while (i < 10) {
  multiploDeTres(i);
  i = i + 1;
}
```

Esta parte del programa va llamando al procedimiento `multiploDeTres` con todos los números del uno al nueve. Observa cómo, cuando el valor no es un múltiplo de tres, termina la ejecución del procedimiento antes de llegar a la línea en donde muestra el mensaje en el Log.

Un juego completo: primer intento

Vamos a tomar el programa tal como quedó al final del nivel 04 y le vamos a hacer unos cambios para convertirlo en un juego completo. No necesitas ir escribiendo los cambios que te vamos a describir, dentro del proyecto `Nivel_05` hay un archivo llamado `primerIntento.sj` que ya tiene todos esos cambios. Si quieres ábrelo en el editor, con el botón  , para que lo puedas ir viendo mientras te describimos los cambios.

Al final del nivel 04 nos quedamos con un programa que tenía esta inicialización y ciclo principal:

```
clear();
nuevaPosicionComida();
dibujaComida();
muestraPuntos();

// Para "siempre"
```

```
while (true) {
  dibujaPersonaje();
  pause(0.1);
  borraPersonaje();
  var boton = readCtrlOne();
  muevePersonaje(boton);
  if (personajeComioComida()) {
    nuevaPosicionComida();
    dibujaComida();
    puntos = puntos + 1;
    muestraPuntos();
    note("C6");
  }
}
```

Ahora queremos tener un juego que te muestre las instrucciones de cómo usarlo, te deje jugar mientras te queden vidas y, al terminar, se regrese a mostrarte las instrucciones. Para eso ahora tenemos este ciclo principal:

```
while (true) {
  muestraInstrucciones();
  juega();
}
```

El procedimiento `muestraInstrucciones` es muy sencillo:

```
muestraInstrucciones() {
  clear();
  showAt("Usa las flechas del teclado", 1, 2);
  showAt("para mover a tu personaje", 1, 3);
  showAt("por la pantalla.", 1, 4);
  showAt("Cada vez que te comes la", 1, 6);
  showAt("comida te da un punto.", 1, 7);
  showAt("Evita que te coman!", 1, 9);

  showAt("Personaje:", 5, 13);
  putAt(PERSONAJE, 17, 13);

  showAt("Comida:", 8, 15);
  putAt(COMIDA, 17, 15);

  showAt("Enemigo:", 7, 17);
  putAt(ENEMIGO, 17, 17);
```

```
  showAt("Presiona la barra de", 10, 20);
  showAt("espacio para empezar", 10, 21);

  while (readCtrlOne() != BOTON_ESPACIO)
    ;
}
```

Muestra las instrucciones y se espera a que el jugador presione la barra de espacio.

La parte realmente interesante, el juego en sí, está dentro del procedimiento `juega`:

```
juega() {
  init();
  while (vidas > 0) {
    dibujaPersonaje();
    dibujaEnemigo();
    pause(0.1);
    borraPersonaje();
    borraEnemigo();
    var boton = readCtrlOne();
    muevePersonaje(boton);
    if (personajeComioComida()) {
      nuevaPosicionComida();
      dibujaComida();
      puntos = puntos + 1;
      muestraPuntos();
      note("C6");
    }
    mueveEnemigo();
    if (enemigoComioPersonaje()) {
      nuevaPosicionPersonaje();
      vidas = vidas - 1;
      muestraVidas();
      note("E2");
    }
  }
  showAt("GAME OVER", 12, 12);
  pause(3);
}
```

Este procedimiento se parece mucho al ciclo principal del programa que teníamos al final del nivel 03. Aquí marcamos con letras negritas las diferencias para que sean más fáciles de ver.

Una de las principales diferencias es que el ciclo de juego ya no se ejecuta "para siempre", ahora se ejecuta mientras que le queden vidas al jugador (`vidas > 0`). Al terminar despliega un mensaje de "`GAME OVER`" y hace una pausa antes de regresar al ciclo principal para que se vuelvan a mostrar las instrucciones del juego.

Cada vez que ejecutamos el procedimiento `juega` empezamos por llamar a un procedimiento `init` para que regrese todas las variables a su estado inicial, borre la pantalla y muestre los puntos y las vidas:

```
init() {
  xPersonaje = 16;
  yPersonaje = 12;
  nuevaPosicionComida();
  xEnemigo = 31;
  yEnemigo = 23;
  puntos = 0;
  vidas = 3;
  clear();
  muestraPuntos();
  muestraVidas();
  dibujaComida();
}
```

En la versión anterior del programa declarábamos e inicializábamos las variables para la posición del personaje en el mismo lugar del programa:

```
/* La posicion del personaje en la pantalla */
var xPersonaje = 16;   // posicion horizontal
var yPersonaje = 12;   // posicion vertical
```

Ahora, como debemos devolver el personaje a su posición inicial cada vez que empieza el juego, simplemente declaramos las variables:

```
/* La posicion del personaje en la pantalla */
var xPersonaje;
var yPersonaje;
```

Y hacemos su inicialización dentro del procedimiento `init`.

También agregamos unas variables para la posición del enemigo en la pantalla y cuántas vidas le quedan al jugador:

```
/* La posicion del enemigo en la pantalla */
var xEnemigo;
var yEnemigo;

/* Cuantas vidas le quedan al jugador */
var vidas;
```

Estas variables también se inicializan dentro del procedimiento `init`.

El procedimiento `init` llama a un procedimiento `muestraVidas` que despliega cuántas vidas le quedan al jugador. Este procedimiento es muy similar al procedimiento `muestraPuntos` que ya teníamos, nada más que, en vez de mostrar los puntos en la parte superior derecha de la pantalla, muestra las vidas en la parte superior izquierda de la pantalla:

```
muestraVidas() {
  showAt("Vidas: " + vidas, 0, 0);
}
```

Dentro del ciclo del juego, dentro de `init`, agregamos unos llamados a los procedimientos `dibujaEnemigo` y `borraEnemigo` para poder animar el enemigo por la pantalla:

```
dibujaEnemigo() {
  putAt(ENEMIGO, xEnemigo, yEnemigo);
}

borraEnemigo() {
  putAt(ESPACIO, xEnemigo, yEnemigo);
}
```

Ahora llegamos a la parte más interesante del programa: cómo funciona el enemigo. Dentro del ciclo del juego, esta es la parte que se encarga del enemigo:

```
mueveEnemigo();
if (enemigoComioPersonaje()) {
  nuevaPosicionPersonaje();
  vidas = vidas - 1;
  muestraVidas();
  note("E2");
 }
```

Aquí llamamos a `mueveEnemigo` para decidir la nueva posición del enemigo. Después, si `enemigoComioPersonaje` devuelve verdadero entonces, con `nuevaPosicionPersonaje`, escogemos una nueva posición al azar para el personaje, decrementamos en uno las vidas que le quedan, mostramos las vidas en la pantalla y hacemos un sonido (diferente al sonido de cuando el personaje se come la comida).

La función `enemigoComioPersonaje` es muy sencilla:

```
enemigoComioPersonaje() {
  return xEnemigo == xPersonaje &&
         yEnemigo == yPersonaje;
}
```

simplemente checa si el personaje y el enemigo están en la misma posición de la pantalla.

El procedimiento `nuevaPosicionPersonaje`, para colocar al personaje en una nueva posición al azar en la pantalla, es prácticamente idéntico al procedimiento `nuevaPosicionComida` que ya teníamos:

```
nuevaPosicionPersonaje() {
  xPersonaje = random(32);
  yPersonaje = random(MAX_Y - MIN_Y) + MIN_Y;
}
```

Lo único que nos falta ver es cómo funciona el procedimiento `mueveEnemigo`. Este procedimiento es donde está la *inteligencia artificial* del enemigo; es la parte del programa que controla el comportamiento del enemigo. El comportamiento que necesitamos es sencillo: el enemigo siempre debe intentar ir hacia donde se encuentra el personaje. Si el personaje está más arriba que el enemigo entonces este último debe subir, si el personaje está más abajo entonces el enemigo debe bajar. Lo mismo ocurre horizontalmente: si el personaje está a la izquierda del enemigo entonces el enemigo debe ir hacia la izquierda, si está a su derecha entonces debe ir hacia la derecha.

Primero vamos a ver la función `sign`, que es una función auxiliar empleada por `mueveEnemigo`:

```
sign(n) {
  if (n < 0)
    return -1;
  if (n > 0)
```

```
    return 1;
  return 0;
}
```

La función `sign` es muy sencilla. Al llamarla hay que pasarle como argumento un número. Si este número es negativo entonces `sign` devuelve como resultado un `-1`, si el número es positivo entonces devuelve un `1`, y si es cero entonces devuelve un `0`. Entender lo que hace `sign`, y como lo hace, no es difícil. Seguramente lo que te has de estar preguntando es ¿y eso para qué me sirve? Justamente es lo que vamos a ver ahora en el procedimiento `mueveEnemigo`.

El procedimiento `mueveEnemigo` es bastante corto:

```
mueveEnemigo() {
  var dx = xPersonaje - xEnemigo;
  var dy = yPersonaje - yEnemigo;
  xEnemigo = xEnemigo + sign(dx);
  yEnemigo = yEnemigo + sign(dy);
}
```

A primera vista no es obvio cómo funciona, pero es bastante sencillo. Veamos primero lo que hace para calcular el nuevo valor de `xEnemigo`, la posición horizontal del enemigo:

```
var dx = xPersonaje - xEnemigo;
```

Hace la resta de `xPersonaje` (la posición horizontal del personaje) menos `xEnemigo` (la posición horizontal del enemigo) y almacena el resultado en una variable `dx`.

Aquí tenemos tres casos: si el personaje está a la izquierda del enemigo entonces el valor de `dx` es negativo, si el personaje está a la derecha del enemigo entonces el valor de dx es positivo, y si los dos están en la misma columna entonces `dx` contiene un cero. Ahora fíjate en la línea en donde modificamos el valor de `xEnemigo`:

```
xEnemigo = xEnemigo + sign(dx);
```

Empleamos la función `sign` para que si `dx` es negativo, el personaje está a la izquierda del enemigo, le restamos uno a `xEnemigo` (sumar -1 es lo mismo que restar 1). Si `dx` es positivo, el personaje está a la derecha del enemigo, le sumamos uno a `xEnemigo`. Y si `dx` es cero, el personaje está en la misma

columna que el enemigo, no cambiamos el valor de `xEnemigo` (al sumarle un cero su valor se queda igual que antes).

Estamos haciendo exactamente lo mismo con la variable `dy` para calcular la nueva posición vertical del enemigo.

Si todavía no lo has hecho, usa el botón para abrir en el editor el programa `primerIntento.sj` y haz click en para probarlo.

El programa ya empieza a funcionar un poco más como un juego, pero hay un problema: ¡Está demasiado difícil! Es imposible escaparse del enemigo. Además, ¿por qué el enemigo se puede mover en diagonal?

Un juego completo: segundo intento

Es fácil entender por qué el enemigo se está moviendo en diagonal. ¿Qué pasa cuando el personaje está más a la derecha y más abajo que el enemigo? Pues como el personaje está más abajo entonces hace que el enemigo baje, y como el personaje también está más a la derecha pues hace que el enemigo se mueva a la derecha. El problema está en que hace estos dos movimientos al mismo tiempo y el resultado es un movimiento en diagonal.

Para corregir este problema lo que tenemos que hacer es que el enemigo se mueva horizontalmente o verticalemente, pero no los dos al mismo tiempo. Lo que hay que hacer es que el enemigo, en un momento dado, decida si se va a mover verticalmente u horizontalmente y depués haga únicamente ese movimiento. Una posible estrategia para decidir si el movimiento del enemigo va a ser vertical u horizontal es ver en qué dirección se encuentra más lejos el personaje. Si el personaje está casi en el mismo renglón que el enemigo pero está en una columna lejana, entonces hace un movimiento horizontal. Si el personaje está en un renglón distante pero casi en la misma columna, entonces hace un moviento vertical.

Para programar esto conviene usar un `if` junto con un `else`. El `else` permite indicar qué queremos ejecutar cuando la condición del `if` es falsa. Veamos un ejemplo. Emplea el botón para abrir el programa `else.sj` en el editor y haz click en para ejecutarlo. El programa despliega en el Log:

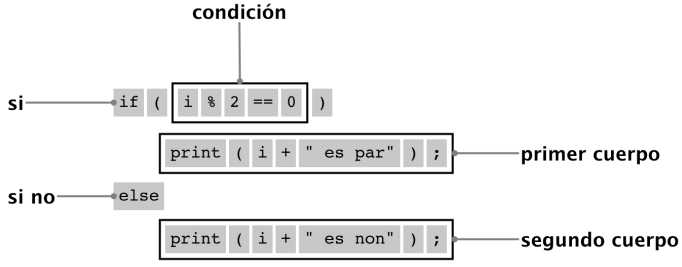
```
0 es par
1 es non
```

```
2 es par
3 es non
```

Este es el programa:

```
var i = 0;
while (i < 4) {
  if (i % 2 == 0)
    print(i + " es par");
  else
    print(i + " es non");
  i = i + 1;
}
```

El `else` permite agregarle un segundo cuerpo al `if`. Si la condición del `if` es verdadera, entonces se ejecuta el contenido del primer cuerpo; si no, se ejecuta el contenido del segundo (*else* en inglés significa *si no*):



Emplea el botón ⏭ para ejecutar el programa `else.sj` paso por paso y fíjate cómo se ejecuta únicamente alguno de los cuerpos del `if` dependiendo de si la condición es verdadera o falsa.

Ya con el `else` es sencillo corregir el procedimiento `mueveEnemigo`. Podemos escribir algo como esto:

```
mueveEnemigo() {
  var dx = xPersonaje - xEnemigo;
  var dy = yPersonaje - yEnemigo;
  if (dx > dy)
    xEnemigo = xEnemigo + sign(dx);
  else
    yEnemigo = yEnemigo + sign(dy);
}
```

Si la distancia horizontal (`dx`) es mayor que la distancia vertical (`dy`) entonces el enemigo se mueve horizontalmente, de lo contrario se mueve verticalmente.

El programa `segundoIntento.sj` ya tiene esta corrección. Ábrelo con el botón y haz click en para probarlo.

¡No funciona! El enemigo sólo se mueve verticalmente y se queda en la última columna. ¿Por qué hace eso?

Un juego completo: tercer intento

Lo que está ocurriendo es que, al iniciar el juego, el personaje está a la izquierda del enemigo y por lo tanto el valor de `dx` es negativo. Y un número negativo siempre es menor que un número positivo. Lo que deberíamos estar comparando son las magnitudes de `dx` y `dy`, independientemente de su signo.

El programa `tercerIntento.sj` tiene una corrección para ese problema. Contiene una nueva función llamada `abs`:

```
abs(n) {
  if (n < 0)
    return -n;
  return n;
}
```

Al llamar a la función `abs` le pasas un número y te devuelve el *valor absoluto* de ese número. Es decir, si el número que le pasas es positivo te devuelve ese mismo número, pero si el número es negativo entonces te devuelve un número positivo con la misma magnitud. En otras palabras, la función `abs` te devuelve la magnitud de un número, independientemente de cuál sea su signo.

Empleando la función `abs` el procedimiento `mueveEnemigo` queda así:

```
mueveEnemigo() {
  var dx = xPersonaje - xEnemigo;
  var dy = yPersonaje - yEnemigo;
  if (abs(dx) > abs(dy))
    xEnemigo = xEnemigo + sign(dx);
  else
    yEnemigo = yEnemigo + sign(dy);
}
```

Usa el botón para abrir el programa `tercerIntento.sj` y haz click en ▶ para probarlo.

El enemigo ya se mueve correctamente. Pero es demasiado rápido, no hay manera de sacarle la vuelta.

Un juego completo: cuarto intento

Hay que hacer que el enemigo se mueva un poco más lentamente para que exista la posibilidad de evitarlo. Porque si es imposible evitarlo entonces el juego está demasiado difícil y no es divertido jugarlo. Conviene encontrar una manera de hacer más lento al enemigo que permita fácilmente ajustar su velocidad y de esta manera poder regular la dificultad del juego. Un juego que es demasiado fácil es aburrido, y un juego que es demasiado difícil es frustrante.

Usa el botón para abrir el programa `cuartoIntento.sj`, ahí hay una mejora al procedimiento mueveEnemigo que permite ajustar fácilmente la velocidad del enemigo:

```
mueveEnemigo() {
  if (random(100) > 60)
    return;
  var dx = xPersonaje - xEnemigo;
  var dy = yPersonaje - yEnemigo;
  if (abs(dx) > abs(dy))
    xEnemigo = xEnemigo + sign(dx);
  else
    yEnemigo = yEnemigo + sign(dy);
}
```

Aquí mostramos en negritas la mejora que hicimos para poder ajustar la velocidad del enemigo. Con esta mejora estamos decrementando la velocidad del enemigo en, aproximadamente, un 40%. Y es fácil hacer que vaya un poco más rápido o más lento. Su funcionamiento es sencillo: generamos un número al azar entre 0 y 99, cuando ese número es mayor que 60, lo cual ocurre aproximadamente un 40% de las veces, ya no ejecutamos el resto del procedimiento `mueveEnemigo` y, por lo tanto, el enemigo no se mueve en ese momento.

Si queremos que el enemigo se mueva un poco más rápido basta cambiar el 60 por un número un poco más grande, eso hace menos probable que el número generado al azar sea mayor y, por lo tanto, disminuye la probabilidad de que no se mueva el enemigo. Eso hace que el enemigo se mueva más seguido, es

decir, más rápido. Y si en vez del 60 ponemos un número más pequeño, entonces el enemigo se mueve más lentamente porque es más probable que el número generado al azar sea mayor y, por lo tanto, no se ejecute el resto del procedimiento para mover al enemigo.

Haz click en ▶ para probar el programa. La velocidad del enemigo ya es más razonable y se puede disfrutar jugando este juego. Pero, si te fijas con cuidado, puedes ver que todavía hay un bug en el programa: cuando el enemigo pasa por encima de la comida, esta última desaparece. ¿Por qué está pasando eso?

Al fin: un juego completo

Lo que está ocurriendo es que al pasar el enemigo por encima de la comida, al borrar al enemigo estamos colocando un tile con un espacio en blanco y ya no volvemos a dibujar la comida. La manera más sencilla de corregir esto es haciendo que dentro del ciclo del juego, en el momento en donde se vuelven a dibujar al personaje y al enemigo, también volvamos a dibujar siempre la comida.

De una vez podemos aprovechar para hacerle otra mejora al juego: por cada 10 puntos que ganas obtienes una nueva vida.

Estos cambios están en el programa `main.sj`, usa el botón 📂 para abrirlo en el editor.

Por el principio del programa definimos una constante para indicar por cada cuántos puntos ganas una vida:

```
final PUNTOS_NUEVA_VIDA = 10;
```

También modificamos el procedimiento muestraInstrucciones para avisar que así se pueden ganar vidas:

```
muestraInstrucciones() {
  clear();
  showAt("Usa las flechas del teclado", 1, 2);
  showAt("para mover a tu personaje", 1, 3);
  showAt("por la pantalla.", 1, 4);
  showAt("Cada vez que te comes la", 1, 6);
  showAt("comida te da un punto.", 1, 7);
  showAt("Evita que te coman!", 1, 9);
  showAt("Nueva vida cada " + PUNTOS_NUEVA_VIDA +
```

```
          " puntos.",
          1, 10);

  showAt("Personaje:", 5, 13);
  putAt(PERSONAJE, 17, 13);

  showAt("Comida:", 8, 15);
  putAt(COMIDA, 17, 15);

  showAt("Enemigo:", 7, 17);
  putAt(ENEMIGO, 17, 17);

  showAt("Presiona la barra de", 10, 20);
  showAt("espacio para empezar", 10, 21);

  while (readCtrlOne() != BOTON_ESPACIO)
    ;
}
```

Fíjate cómo empleamos la contante `PUNTOS_NUEVA_VIDA` dentro del procedimiento `muestraInstrucciones` en vez de poner directamente ahí un `10`. De esta manera si deseas cambiar el número de puntos que se necesitan obtener para ganar una vida entonces basta con cambiar eso únicamente en donde se define la constante `PUNTOS_NUEVA_VIDA`.

Y también modificamos el procedimiento `juega` para que siempre vuelva a dibujar la comida (y ya no desaparezca cuando el enemigo pasa por encima de ella). Aprovechamos también para hacer que cada vez que llegas a un múltiplo de `PUNTOS_NUEVA_COMIDA` te dé una vida extra:

```
juega() {
  init();
  while (vidas > 0) {
    dibujaPersonaje();
    dibujaComida();
    dibujaEnemigo();
    pause(0.1);
    borraPersonaje();
    borraEnemigo();
    var boton = readCtrlOne();
    muevePersonaje(boton);
    if (personajeComioComida()) {
      nuevaPosicionComida();
```

```
      puntos = puntos + 1;
      muestraPuntos();
      if (puntos % PUNTOS_NUEVA_VIDA == 0) {
         vidas = vidas + 1;
         muestraVidas();
         note("C5");
      } else
        note("C6");
    }
    mueveEnemigo();
    if (enemigoComioPersonaje()) {
      nuevaPosicionPersonaje();
      vidas = vidas - 1;
      muestraVidas();
      note("E2");
    }
  }
  showAt("GAME OVER", 12, 12);
  pause(3);
}
```

Ya puedes usar el juego en la simpleJ virtual console

Ahora que ya tenemos un primer juego completo seguramente lo quieres probar en la simpleJ virtual console. Es muy sencillo hacerlo, simplemente en el menú de **Project** selecciona la opción **Package game for simpleJ virtual console** (**Project > Package game for simpleJ virtual console**) ¡Listo! Si ahora ejecutas la simpleJ virtual console puedes ver que ya está ahí un nuevo juego que se llama `Nivel_06`.

Nota

El programa que queda como juego es el que se llama main.sj dentro de tu proyecto. Aunque al hacer **Project > Package game for simpleJ virtual console** esté otro programa en el editor.

Nivel 6: Múltiples enemigos

Con múltiples enemigos el juego es más interesante

Vamos a mejorar el juego para que ahora varios enemigos traten de comerse al personaje.

En el menú de **Project** selecciona la opción **Switch to another project...** (**Project > Switch to another project...**) y selecciona el proyecto `Nivel_06`. Haz click en ▶ para probar el programa. Hay cuatro enemigos que intentan comerse al personaje. Al inicio del juego, cada uno de los cuatro está en una esquina de la pantalla.

Arreglos

En el nivel anterior teníamos dos variables para guardar la posición horizontal y vertical del enemigo:

```
var xEnemigo;
var yEnemigo;
```

Para cuatro enemigos podríamos emplear ocho variables:

```
var xEnemigo1;
var xEnemigo2;
var xEnemigo3;
var xEnemigo4;
var yEnemigo1;
var yEnemigo2;
var yEnemigo3;
var yEnemigo4;
```

Y emplearlas así:

```
dibujaEnemigos() {
  putAt(ENEMIGO, xEnemigo1, yEnemigo1);
  putAt(ENEMIGO, xEnemigo2, yEnemigo2);
  putAt(ENEMIGO, xEnemigo3, yEnemigo3);
```

```
  putAt(ENEMIGO, xEnemigo4, yEnemigo4);
}
```

Aunque podríamos hacer esto, no es la mejor manera de manejar a los cuatro enemigos dentro del juego. Es bastante engorroso andar escribiendo cada cosa cuatro veces; además, ¿qué pasaría si queremos poner, en vez de cuatro, unos cuarenta enemigos?

La manera más sencilla de programar esto es empleando *arreglos*. Un arreglo nos permite emplear una sola variable para almacenar varios valores.

Veamos cómo funcionan los arreglos. Haz click en  y abre el programa `arreglos.sj`:

```
var datos = new array[5];

datos[0] = 10;
datos[1] = 3;
datos[2] = -1;
datos[3] = 45;
datos[4] = 28;

print(datos[0]);
print(datos[1]);
print(datos[2]);
print(datos[3]);
print(datos[4]);

var i = 0;
while (i < 5) {
  print(datos[i]);
  i = i + 1;
}

print(length(datos));

i = 0;
while (i < length(datos)) {
  print(datos[i]);
  i = i + 1;
}

for (var i = 0; i < length(datos); i = i + 1)
```

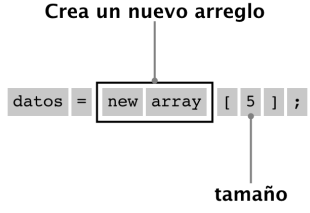
```
  print(datos[i]);

i++;

for (var i = 0; i < length(datos); i++)
  print(datos[i]);

var a = [3, 7, 25, 46, 31, 29];

for (var i = 0; i < length(a); i++)
  print(a[i]);
```

Haz click en ⏭ para empezar a ejecutar el programa paso por paso.

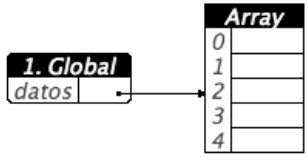
La primera línea:

```
var datos = new array[5];
```

Indica que queremos crear un arreglo que puede almacenar cinco valores y almacenarlo en la variable `datos`:



Haz click en ⏭ para ejecutar la primera línea y la vista de la memoria queda así:

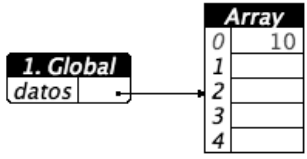


La computadora creó un arreglo con capacidad para cinco valores y almacenó una *referencia* a ese arreglo dentro de la variables `datos`. Se dice que el arreglo puede almacenar cinco *elementos*; las localidades para estos cinco elementos están numeradas de cero a cuatro, esos números son los *subíndices* de los elementos.

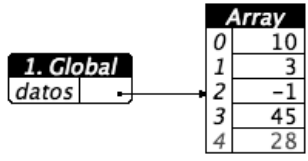
Para emplear un elemento de un arreglo se emplea el nombre de la variable, que contiene la referencia al arreglo, seguida del subíndice entre corchetes. La línea siguiente del programa es:

```
datos[0] = 10;
```

Haz click en ⏭ para ejecutarla. En la vista de la memoria puedes ver que almacenó un `10` dentro del primer elemento del arreglo (que tiene como subíndice `0`):



Sigue ejecutando el programa paso por paso hasta que termine de llenar los elementos del arreglo:



Las cinco líneas siguientes del programa despliegan en el Log los valores almacenados dentro del arreglo:

```
print(datos[0]);
print(datos[1]);
print(datos[2]);
print(datos[3]);
print(datos[4]);
```

Ejecútalas paso por paso.

Una de las ventajas de los arreglos es que podemos usar una variable como subíndice para accesar sus elementos. Esto nos permite emplear un ciclo para recorrer todos los elementos del arreglo en vez de tener que escribir una instrucción por cada uno de ellos. Las siguientes líneas del programa vuelven a desplegar en el Log los elementos del arreglo, pero esta vez empleando una variable, dentro de un ciclo, como subíndice:

```
var i = 0;
while (i < 5) {
  print(datos[i]);
  i = i + 1;
}
```

Ejecuta este ciclo paso por paso y comprueba que despliega en el Log los valores de cada elemento del arreglo.

Aquí sabemos que el arreglo contiene cinco elementos. Por eso hicimos que en el ciclo la variable `i` tomara los valores sucesivos desde el `0` hasta el `4`. Pero, a veces, necesitamos escribir un ciclo que funcione con un arreglo de cualquier longitud. En estos casos podemos emplear la función predefinida `length` para averiguar el tamaño del arreglo. Ya habíamos visto anteriormente que podíamos emplear `length` para saber la longitud de un string (cuántos caracteres contiene); también la podemos llamar con un arreglo como argumento y nos devuelve el tamaño del arreglo. La línea siguiente del programa emplea `length` para desplegar en el Log el tamaño del arreglo:

```
print(length(datos));
```

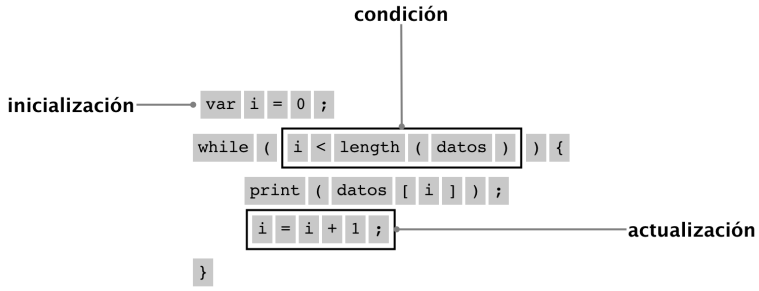
Haz click en ⏭ para ejecutarla, y comprueba que despliega un `5` en el Log.

La parte que sigue del programa es un ciclo similar al anterior pero usando la función `length`:

```
i = 0;
while (i < length(datos)) {
  print(datos[i]);
  i = i + 1;
}
```

Emplea el botón ⏭ para ejecutarla paso por paso.

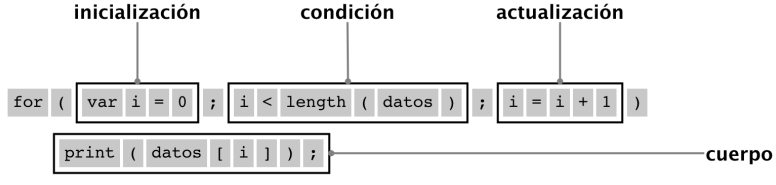
Ya hemos visto varias veces un patrón que ocurre muy seguido al emplear un ciclo. Antes del ciclo inicializamos algún contador, típicamente en cero; ejecutamos el ciclo mientras que se cumpla una condición, por lo general, que el valor de la variable sea inferior a un límite; al final del cuerpo actualizamos, casi siempre incrementamos en uno, el valor del contador:



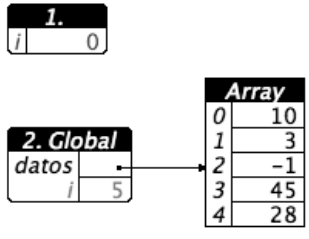
Este patrón se emplea tan frecuentemente que existe una manera más práctica de escribirlo: empleando el `for`. Con el `for` es más sencillo darse cuenta de lo que se quiere lograr con un ciclo porque la inicialización, la condición y la actualización están una junto a la otra en el programa. Las tres se colocan, entre paréntesis y separadas por puntos y comas, después de la palabra `for`:

```
for (var i = 0; i < length(datos); i = i + 1)
  print(datos[i]);
```

Que la computadora interpreta así:



Ejecuta este ciclo paso por paso. En la vista de la memoria puedes ver que se crea una variable `i` que es local dentro del cuerpo del ciclo:



Incrementar en uno el valor de una variable también es una operación que se ejecuta con frecuencia. Por eso existe un operador especializado para realizarla.

Es el operador ++, con el cual podemos escribir `i++` en vez de `i = i + 1`. La siguiente línea del programa emplea este operador para incrementar en uno el valor de la variable global `i`:

```
i++;
```

Emplea el botón ⏭ para ejecutar esa línea; observa cómo el valor de `i` pasa de `5` a `6`.

Las líneas siguientes muestran un ciclo empleando el `for` y el operador ++:

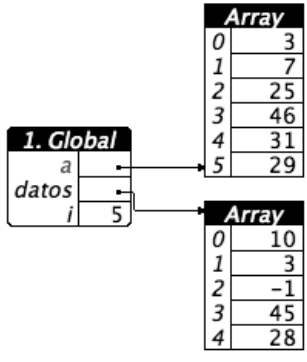
```
for (var i = 0; i < length(datos); i++)
  print(datos[i]);
```

Ejecútalo paso por paso; comprueba que hace exactamente lo mismo que la versión anterior del ciclo.

Ya vimos que para crear un arreglo se emplea `new array` seguido del tamaño del arreglo entre corchetes. Existe otra manera de crear un arreglo: poniendo una lista de valores entre corchetes (sin el `new array`) puedes crear un arreglo que contiene esos valores. La línea siguiente del programa crea un arreglo con seis valores y almacena una referencia a ese arreglo dentro de la variable `a`:

```
var a = [3, 7, 25, 46, 31, 29];
```

Haz click en ⏭ para ejecutarla; la vista de la memoria queda así:



Múltiples enemigos: primer intento

Vamos a tomar el programa tal como quedó al final del nivel 05 y le vamos a hacer unos cambios para convertirlo en un juego completo. No necesitas ir escribiendo los cambios que te vamos a describir, dentro del proyecto `Nivel_06` hay un archivo llamado `primerIntento.sj` que ya tiene todos esos cambios. Si quieres ábrelo en el editor, con el botón  , para que lo puedas ir viendo mientras te describimos los cambios.

Esta es la nueva versión del procedimiento `juega`, con los cambios en negritas:

```
juega() {
  init();
  while (vidas > 0) {
    dibujaPersonaje();
    dibujaComida();
    dibujaEnemigos();
    pause(0.1);
    borraPersonaje();
    borraEnemigos();
    var boton = readCtrlOne();
    muevePersonaje(boton);
    if (personajeComioComida()) {
      nuevaPosicionComida();
      puntos = puntos + 1;
      muestraPuntos();
      if (puntos % PUNTOS_NUEVA_VIDA == 0) {
        vidas = vidas + 1;
        muestraVidas();
        note("C5");
      } else
        note("C6");
    }
    mueveEnemigos();
    if (enemigoComioPersonaje()) {
      nuevaPosicionPersonaje();
      vidas = vidas - 1;
      muestraVidas();
      note("E2");
    }
  }
  showAt("GAME OVER", 12, 12);
```

```
  pause(3);
}
```

El único cambio fue renombrar algunos procedimientos para que ahora estén en plural. Podríamos haber dejado los nombres que teníamos, ya que para la computadora lo único que importa es lo que contiene la definición del procedimiento y no cómo se llama, pero siempre es conveniente asegurarse que los nombres de los procedimientos reflejen lo mejor posible lo que hacen: así el programa es más fácil de entender. Por cierto, dejamos el nombre del procedimiento `enemigoComioPersonaje` en singular porque, en un momento dado, es un sólo enemigo el que se come al personaje.

También pusimos los nombres de las variables en plural:

```
/* Las posiciones de los enemigos en la pantalla */
var xEnemigos;
var yEnemigos;
```

Y modificamos el procedimiento `init` para que inicialice estas dos variables con unos arreglos:

```
init() {
  xPersonaje = 16;
  yPersonaje = 12;
  nuevaPosicionComida();
  xEnemigos = [MAX_X, MAX_X, MIN_X, MIN_X];
  yEnemigos = [MAX_Y, MIN_Y, MAX_Y, MIN_Y];
  puntos = 0;
  vidas = 3;
  clear();
  muestraPuntos();
  muestraVidas();
}
```

En `xEnemigos[0]` y `yEnemigos[0]` se almacenan las posiciones horizontal y vertical del primer enemigo, en `xEnemigos[1]` y `yEnemigos[1]` las del segundo, en `xEnemigos[2]` y `yEnemigos[2]` las del tercero, y en `xEnemigos[3]` y `yEnemigos[3]` las del cuarto. Estamos usando las constantes que definimos con las posiciones mínimas y máximas en la pantalla, tanto horizontales como verticales, para colocar a un enemigo en cada esquina de la pantalla.

Veamos ahora las nuevas definiciones de los procedimientos específicos para los enemigos. Los procedimientos `dibujaEnemigos` y `borraEnemigos` son bastante sencillos:

```
dibujaEnemigos() {
  for (var i = 0; i < 4; i++)
    putAt(ENEMIGO, xEnemigos[i], yEnemigos[i]);
}

borraEnemigos() {
  for (var i = 0; i < 4; i++)
    putAt(ESPACIO, xEnemigos[i], yEnemigos[i]);
}
```

Simplemente usamos un ciclo (con un `for`) para dibujar o borrar a cada uno de los cuatro enemigos.

El procedimiento `enemigoComioPersonaje` está un poco más interesante:

```
enemigoComioPersonaje() {
  for (var i = 0; i < 4; i++)
    if (xEnemigos[i] == xPersonaje &&
        yEnemigos[i] == yPersonaje)
      return true;
  return false;
}
```

También usamos un ciclo, para ir comparando la posición del personaje con la posición de cada uno de los cuatro enemigos. Si las posiciones, tanto horizontal como vertical, de un enemigo son iguales a las del personaje entonces en ese mismo instante nos regresamos del procedimiento devolviendo `true` como resultado. No tiene caso seguir comparando con las posiciones de los demás enemigos porque ya sabemos que un enemigo se comió al personaje. Pero, si ninguno de los enemigos está en la misma posición de la pantalla que el personaje entonces, al terminar de ejecutarse el ciclo, devolvemos como resultado `false`.

El procedimiento `mueveEnemigos` también es bastante sencillo:

```
mueveEnemigos() {
  if (random(100) > 40)
    return;
  for (var i = 0; i < 4; i++) {
    var dx = xPersonaje - xEnemigos[i];
```

```
    var dy = yPersonaje - yEnemigos[i];
    if (abs(dx) > abs(dy))
      xEnemigos[i] = xEnemigos[i] + sign(dx);
    else
      yEnemigos[i] = yEnemigos[i] + sign(dy);
  }
}
```

Empezamos el procedimiento con la técnica que ya teníamos de generar un número al azar y compararlo con una constante para no siempre mover a los enemigos. Nada más que ahora cambiamos la constante, en vez de un `60` tenemos un `40`: con más enemigos hay que hacer que se muevan más lentamente para que el juego no esté demasiado difícil. Aproximadamente el 40% de las veces que ejecutemos este procedimiento el número generado al azar será menor que `40` y se ejecuta el resto del procedimiento, el cual contiene un ciclo para mover a cada uno de los cuatro enemigos.

Si todavía no lo has hecho, usa el botón para abrir en el editor el programa `primerIntento.sj` y haz click en para probarlo.

Ya hay cuatro enemigos persiguiendo al personaje. Pero, después de un momento de andar jugando, ¡los enemigos empiezan a desaparecer! En vez de cuatro enemigos sólo se pueden ver en pantalla dos o tres. ¿Qué es lo que está pasando?

Múltiples enemigos: segundo intento

Lo que está ocurriendo es que, en algún momento, dos enemigos se están moviendo a exactamente la misma posición de la pantalla. Y una vez que esto ocurre, como los dos toman decisiones idénticas respecto a cómo moverse, ya se quedan para siempre uno encima del otro y parecen ser un sólo enemigo.

En el programa `segundoIntento.sj` ya está corregido este problema. Usa el botón para abrirlo en el editor.

Para evitar que los enemigos se encimen necesitamos, antes de mover a un enemigo, checar que no haya otro enemigo en la nueva posición que le queremos asignar. Por lo tanto necesitamos tener alguna manera de saber si ya hay un enemigo en cierta posición de la pantalla. Para esto escribimos un procedimiento `hayEnemigoEn`:

```
hayEnemigoEn(x, y) {
  for (var i = 0; i < 4; i++)
    if (xEnemigos[i] == x && yEnemigos[i] == y)
      return true;
  return false;
}
```

Este procedimiento es muy similar al procedimiento `enemigoComioPersonaje` que ya teníamos. La única diferencia es que en vez de comparar las posiciones horizontal y vertical de cada enemigo con `xPersonaje` y `yPersonaje` ahora las estamos comparando con los valores `x` y `y` que se pasan como argumentos al llamar a este procedimiento. De una vez lo aprovechamos para reescribir el procedimiento `enemigoComioPersonaje` de una manera más sencilla:

```
enemigoComioPersonaje() {
  return hayEnemigoEn(xPersonaje, yPersonaje);
}
```

Aunque no fue para esto que escribimos el procedimiento `hayEnemigoEn` nunca hay que dejar de aprovechar una oportunidad de simplificar un programa y hacerlo más fácil de entender o modificar.

Ahora el procedimiento `mueveEnemigos` queda así:

```
mueveEnemigos() {
  if (random(100) > 40)
    return;
  for (var i = 0; i < 4; i++) {
    var dx = xPersonaje - xEnemigos[i];
    var dy = yPersonaje - yEnemigos[i];
    if (abs(dx) > abs(dy)) {
      var xNueva = xEnemigos[i] + sign(dx);
      if (!hayEnemigoEn(xNueva, yEnemigos[i]))
        xEnemigos[i] = xNueva;
    } else {
      var yNueva = yEnemigos[i] + sign(dy);
      if (!hayEnemigoEn(xEnemigos[i], yNueva))
        yEnemigos[i] = yNueva;
    }
  }
}
```

Antes de mover a un enemigo horizontalmente calculamos su nueva posición horizontal y la almacenamos en una variable `xNueva`. Después empleamos esa variable junto con la posición vertical del enemigo (que no ha cambiado) para ver si ya hay algún enemigo ahí. Si te fijas bien puedes ver que aquí estamos usando algo nuevo: en el `if` hay un signo de admiración (`!`) justo después de abrir el paréntesis de la condición. Este signo de admiración es el operador *not*, el cual se emplea para negar un valor booleano. La negación de `true` es `false`; la negación de `false` es `true`. Por lo tanto, lo que estamos diciendo aquí es: "si **no** hay un enemigo en la posición con coordenadas `xNueva` y `yEnemigos[i]` entonces almacena el valor de `xNueva` en `xEnemigos[i]`". Hacemos lo mismo para el cambio de posición vertical con la variable `yNueva`.

Haz click en ▶ para probar el programa. Los enemigos ya no se enciman uno sobre el otro, pero es un poco molesta la manera en que todos avanzan y se detienen al mismo tiempo. Se vería más natural si cada uno hiciera pausas en diferentes momentos.

Múltiples enemigos: tercer intento

Usa el botón para abrir el programa `tercerIntento.sj`. Ahí modificamos el procedimiento `mueveEnemigos` para que ahora use números al azar para graduar la velocidad de los enemigos dentro del ciclo en vez de hacerlo antes del ciclo:

```
mueveEnemigos() {
  for (var i = 0; i < 4; i++) {
    if (random(100) > 40)
      return;
    var dx = xPersonaje - xEnemigos[i];
    var dy = yPersonaje - yEnemigos[i];
    if (abs(dx) > abs(dy)) {
      var xNueva = xEnemigos[i] + sign(dx);
      if (!hayEnemigoEn(xNueva, yEnemigos[i]))
        xEnemigos[i] = xNueva;
    } else {
      var yNueva = yEnemigos[i] + sign(dy);
      if (!hayEnemigoEn(xEnemigos[i], yNueva))
        yEnemigos[i] = yNueva;
    }
  }
}
```

Haz click en ▶ para probarlo. Ya no se mueven todos los enemigos al mismo tiempo, pero ¡se están moviendo con velocidades diferentes! Hasta hay uno de ellos que casi ni se mueve. ¿Qué es lo que está pasando?

La explicación es sencilla. Al entrar al ciclo genera un número al azar entre 0 y 99; si este número es mayor que 40 (lo cual ocurre aproximadamente el 60% de las veces) termina la ejecución del procedimiento y ningún enemigo se mueve. Si fue menor o igual a 40 entonces mueve al primer enemigo y se regresa al principio del ciclo. Ahí vuelve a generar un número al azar y si éste fue mayor que 40 entonces termina la ejecución del procedimiento y ¡ya no movió a ninguno de los otros tres enemigos! Si fue menor que 40 entonces mueve al segundo enemigo y se vuelve a regresar al principio del ciclo donde otra vez existe la probabilidad que ya no mueva a los otros dos enemigos. Por lo tanto, para cada enemigo se vuelve cada vez menos probable que se mueva en un momento dado. De hecho las probabilidades son aproximadamente de 40%, 16%, 6.4% y 2.56%; eso no es lo que deseamos; queremos que para cada uno de ellos la probabilidad de moverse en un momento dado sea la misma, para que se muevan a la misma velocidad.

Una manera de lograrlo sería reescribiendo el procedimiento de esta manera:

```
mueveEnemigos() {
  for (var i = 0; i < 4; i++) {
    if (random(100) <= 40) {
      var dx = xPersonaje - xEnemigos[i];
      var dy = yPersonaje - yEnemigos[i];
      if (abs(dx) > abs(dy)) {
        var xNueva = xEnemigos[i] + sign(dx);
        if (!hayEnemigoEn(xNueva, yEnemigos[i]))
          xEnemigos[i] = xNueva;
      } else {
        var yNueva = yEnemigos[i] + sign(dy);
        if (!hayEnemigoEn(xEnemigos[i], yNueva))
          yEnemigos[i] = yNueva;
      }
    }
  }
}
```

Invertimos la comparación dentro del `if` y únicamente movemos a ese enemigo si el número generado al azar es menor o igual a 40. Si no lo es entonces no

movemos a ese enemigo, pero tampoco nos salimos del procedimiento, simplemente nos regresamos al inicio del ciclo para ahora, tal vez, mover al siguiente enemigo. Esta manera de hacerlo es correcta y funciona. Pero existe una manera más elegante de evitar que se ejecute el resto del cuerpo de un ciclo en una de sus iteraciones: empleando un `continue`.

Haz click en  y abre el programa `continue.sj`:

```
for (var i = 0; i < 10; i++) {
  if (i % 3 != 0)
    continue;
  print(i);
}
```

Este programa consiste en un ciclo donde la variable `i` va tomando los valores desde 0 hasta el 9. En cada iteración, si el valor de `i` no es un múltiplo de tres (si el residuo al dividirlo entre tres es diferente de cero) entonces ejecuta un `continue` para saltarse el resto del cuerpo del ciclo e iniciar inmediatamente la siguiente iteración.

Emplea  para ejecutarlo paso por paso y observa cómo con el `continue` ya no se ejecuta el resto del cuerpo del `for` para esa iteración.

Nota

El `continue` también se puede emplear con un `while`, pero hay que tener cuidado porque es probable que al final del cuerpo se encuentre una instrucción para incrementar un contador que sirva para controlar el número de iteraciones; al ejecutar el `continue` dentro de un `while` nunca llegaría a esa instrucción y es posible que el programa se quede *ciclado* (ejecutando "para siempre" esa misma parte del programa).

Con el `for` esto no ocurre porque la instrucción que actualiza la variable se encuentra en la cabeza y siempre se ejecuta, aún después de un `continue`.

Al fin: múltiples enemigos

El programa `main.sj` ya tiene la versión de `mueveEnemigos` con el `continue`:

```
mueveEnemigos() {
  for (var i = 0; i < 4; i++) {
    if (random(100) > 40)
      continue;
    var dx = xPersonaje - xEnemigos[i];
    var dy = yPersonaje - yEnemigos[i];
    if (abs(dx) > abs(dy)) {
      var xNueva = xEnemigos[i] + sign(dx);
      if (!hayEnemigoEn(xNueva, yEnemigos[i]))
        xEnemigos[i] = xNueva;
    } else {
      var yNueva = yEnemigos[i] + sign(dy);
      if (!hayEnemigoEn(xEnemigos[i], yNueva))
        yEnemigos[i] = yNueva;
    }
  }
}
```

Haz click en  para abrirlo, y después en  para volverlo a probar.

Nivel 7: Switch y case

Un juego un poco más interesante

Para que el juego esté un poco más interesante vamos a cambiar la manera en la que controlas al personaje. En vez de irlo moviendo con las flechas ahora el personaje se va a ir moviendo, por sí solo, y vas a usar las flechas para cambiar la dirección en la cual se mueve. Además, en vez de que el personaje sea una bola, ahora va a ser una flecha que apunta en la dirección hacia la cual se está moviendo.

En el menú de **Project** selecciona la opción **Switch to another project...** (**Project > Switch to another project...**) y selecciona el proyecto `Nivel_07`. Haz click en ▶ para probar el programa y ver cómo se controla ahora el personaje.

Switch y case

Esta nueva manera de controlar al personaje es bastante sencilla de programar empleando `switch` y `case`, con los cuales puedes indicarle a la computadora que ejecute una opción de entre varias. Para ver cómo funcionan haz click en y abre el programa `switch.sj`, que contiene lo siguiente:

```
digito(d) {
  var mensaje;
  switch (d) {
    case 0:
      mensaje = "cero";
      break;

    case 1:
      mensaje = "uno";
      break;

    case 2:
      mensaje = "dos";
      break;

    case 3:
      mensaje = "tres";
```

```
      break;

    case 4:
      mensaje = "cuatro";
      break;

    case 5:
      mensaje = "cinco";
      break;

    case 6:
      mensaje = "seis";
      break;

    case 7:
      mensaje = "siete";
      break;

    case 8:
      mensaje = "ocho";
      break;

    case 9:
      mensaje = "nueve";
      break;

    default:
      mensaje = "otro";
  }
  print(mensaje);
}

digito(3);
digito(8);
digito(27);

numeros(n) {
  switch (n) {
    case 9:
      print("nueve");
    case 8:
      print("ocho");
    case 7:
      print("siete");
```

```
    case 6:
      print("seis");
    case 5:
      print("cinco");
    case 4:
      print("cuatro");
    case 3:
    case 2:
    case 1:
      print("tres, dos y uno");
  }
}

numeros(6);
numeros(2);
numeros(11);
```

Ahora ve haciendo click en ▶| para ejecutarlo paso por paso. Lo primero que hace el programa es definir un procedimiento `digito`, y justo después lo llama pasándole como argumento un `3`.

Dentro del procedimiento `digito` empieza por crear una variable `mensaje`, sin asignarle por ahora ningún valor. Enseguida está la línea que dice:

```
switch (d) {
```

Aquí estamos empleando un `switch` para decirle a la computadora que debe seleccionar una opción con base en el valor que contiene la variable `d`, que en este instante es un `3`. Las opciones vienen después, entre llaves, empleando un `case` por cada opción posible.

Para cada una de las opciones está la palabra `case` seguida de un valor que indica en qué caso debe seleccionarse esta opción. Después del valor hay un signo de dos puntos (`:`) seguido por las instrucciones a ejecutar cuando se selecciona esta opción.

Ve haciendo click en ▶| y observa cómo va checando cada una de las opciones hasta que llega a la parte donde dice:

```
case 3:
  mensaje = "tres";
  break;
```

Como el valor de la variable `d` es igual a `3` entonces ejecuta las instrucciones asociadas con esta opción. Almacena el string `"tres"` en la variable mensaje, y llega al `break` que le indica que aquí se terminan las instrucciones para esta opción. Al ejecutar el `break` sabe que ya no hay nada más que hacer para esta opción y se salta a la instrucción que viene después del cuerpo del `switch`:

```
print(mensaje);
```

Al ejecutarla despliega el mensaje (`"tres"`) en el Log y termina de ejecutar el procedimiento `digito`.

El programa vuelve a llamar al procedimiento `digito` ahora con un `8` como argumento. Ejecútalo paso por paso, con ⏭, y fíjate cómo ahora ejecuta este código dentro del cuerpo del `switch`:

```
case 8:
  mensaje = "ocho";
  break;
```

Ahora el programa llama al procedimiento `digito` con un `27` como argumento. Ejecútalo paso por paso y observa que como no hay ningún `case` que corresponda a ese valor entonces llega hasta la parte marcada `default`:

```
default:
  mensaje = "otro";
```

El `default` se emplea cuando quieres que se ejecuten unas instrucciones si el valor que se empleó en el `switch` no corresponde a ninguna de las opciones indicadas con los `case`.

A continuación el programa define un procedimiento `numeros` para que veas lo que pasa cuando no se emplean el `break` y el `default`. Sigue ejecutando el programa paso por paso y observa lo que ocurre.

Primero llamamos a `numeros` pasándole un `6` como argumento. Al igual que como vimos anteriormente con el procedimiento `digito` el `switch` va checando los `case` hasta que llega al que corresponde al valor `6`. Ahí ejecuta la instrucción que viene después de los dos puntos; pero como esta vez no hay un `break` se sigue ejecutando las instrucciones de todos los `case` siguientes hasta que se sale del cuerpo del `switch`.

Después llamamos a `numeros` pasándole un `2` como argumento. En el `case` que le corresponde no hay ninguna instrucción justo después de los dos puntos,

pero como tampoco hay un `break` entonces sigue adelante y ejecuta la instrucción que está después del `case` que corresponde al `1`.

Y para terminar volvemos a llamar a `numeros` pasándole un `11` como argumento. Ahora no hay ningún `case` que corresponda a ese valor; como tampoco hay un `default` entonces simplemente no ejecuta nada dentro del cuerpo del `switch`.

Nueva manera de controlar al personaje

Haz click en  y abre el programa `main.sj` para ver cómo está programada la nueva manera de controlar y dibujar al personaje.

Esta vez no hay ningún cambio en el procedimiento `juega`, se queda tal como estaba en el nivel anterior. Ahora los cambios están en los procedimientos para dibujar y mover al personaje. También definimos una nueva variable y unas cuantas constantes.

En vez de una constante para dibujar al personaje como una bola, ahora tenemos cuatro constantes para los tiles de las flechas en cada una de las cuatro direcciones posibles:

```
/* Constantes para los tiles del personaje, la comida,
   los enemigos y el espacio en blanco */
final PERSONAJE_ARRIBA = 28;
final PERSONAJE_ABAJO = 29;
final PERSONAJE_IZQUIERDA = 30;
final PERSONAJE_DERECHA = 31;
final COMIDA = 43;
final ENEMIGO = 42;
final ESPACIO = 32;
```

También agregamos una variable `dirPersonaje` para almacenar en qué dirección se está moviendo el personaje:

```
/* Posicion y direccion del personaje en la pantalla */
var xPersonaje;
var yPersonaje;
var dirPersonaje;
```

Dentro de la variable `dirPersonaje` almacenamos un número que indica en qué dirección se está moviendo el personaje. Definimos unas constantes con los números que vamos a emplear para indicar la dirección:

```
/* Constantes para las direcciones */
final ARRIBA = 1;
final ABAJO = 2;
final IZQUIERDA = 3;
final DERECHA = 4;
```

Estas constantes son totalmente arbitrarias, podríamos haber empleado otros valores. Lo único importante es que sea un número diferente para cada dirección.

Ahora dentro del procedimiento `init` también tenemos que indicar una dirección inicial para el personaje:

```
init() {
  xPersonaje = 16;
  yPersonaje = 12;
  dirPersonaje = DERECHA;
  nuevaPosicionComida();
  xEnemigos = [MAX_X, MAX_X, MIN_X, MIN_X];
  yEnemigos = [MAX_Y, MIN_Y, MAX_Y, MIN_Y];
  puntos = 0;
  vidas = 3;
  clear();
  muestraPuntos();
  muestraVidas();
}
```

El procedimiento `dibujaPersonaje` emplea un `switch` para seleccionar con qué tile dibujar al personaje dependiendo de la dirección hacia la cual se está moviendo:

```
dibujaPersonaje() {
  switch (dirPersonaje) {
    case ARRIBA:
      putAt(PERSONAJE_ARRIBA, xPersonaje, yPersonaje);
      break;

    case ABAJO:
      putAt(PERSONAJE_ABAJO, xPersonaje, yPersonaje);
      break;

    case IZQUIERDA:
      putAt(PERSONAJE_IZQUIERDA, xPersonaje, yPersonaje);
      break;
```

```
    case DERECHA:
      putAt(PERSONAJE_DERECHA, xPersonaje, yPersonaje);
      break;
  }
}
```

Y el procedimiento `muevePersonaje` ahora empieza por llamar a un procedimiento auxiliar `actualizaDireccion` para cambiar la dirección en la que se mueve el personaje si el jugador presionó alguna de las flechas en el teclado:

```
actualizaDireccion(boton) {
  if (boton == BOTON_ARRIBA)
    dirPersonaje = ARRIBA;
  if (boton == BOTON_ABAJO)
    dirPersonaje = ABAJO;
  if (boton == BOTON_IZQUIERDA)
    dirPersonaje = IZQUIERDA;
  if (boton == BOTON_DERECHA)
    dirPersonaje = DERECHA;
}

muevePersonaje(boton) {
  actualizaDireccion(boton);
  switch (dirPersonaje) {
    case ARRIBA:
      if (yPersonaje > MIN_Y)
        yPersonaje--;
      break;

    case ABAJO:
      if (yPersonaje < MAX_Y)
        yPersonaje++;
      break;

    case IZQUIERDA:
      if (xPersonaje > MIN_X)
        xPersonaje--;
      break;

    case DERECHA:
      if (xPersonaje < MAX_X)
        xPersonaje++;
      break;
```

```
  }
}
```

El procedimiento `actualizaDireccion` simplemente checa si está presionada alguna de las flechas en el teclado y, de ser así, actualiza el contenido de la variable `dirPersonaje` para que ahora se mueva en esa dirección.

Después de llamar a `actualizaDireccion`, el procedimiento `muevePersonaje` emplea un `switch` para seleccionar cómo cambiar la posición del personaje en la pantalla, cuidando que no se salga de la pantalla.

Estos fueron los únicos cambios que se le tuvieron que hacer al programa para modificar la manera con la cual se dibuja y controla al personaje. Estos cambios fueron sencillos porque el programa ya estaba bien organizado con el trabajo dividido entre procedimientos más o menos independientes, en el que cada uno se encargaba de una parte de la funcionalidad del juego.

Nivel 8: Ambientes

Enemigos con comportamientos diferentes

Para que el juego esté un poco más interesante vamos a hacer que cada uno de los enemigos tenga un comportamiento diferente.

En el menú de **Project** selecciona la opción **Switch to another project...** (**Project > Switch to another project...**) y selecciona el proyecto `Nivel_08`. Haz click en ▶ para probar el programa. No todos los enemigos se mueven a la misma velocidad; dos de ellos persiguen al personaje por toda la pantalla, mientras que los otros dos nunca se alejan mucho de la comida para "cuidarla".

Ambientes

Hasta ahora hemos estado empleando dos variables para almacenar las posiciones de los enemigos en la pantalla:

```
/* Las posiciones de los enemigos en la pantalla */
var xEnemigos;
var yEnemigos;
```

Cada una de estas variables contiene una referencia a un arreglo de cuatro elementos, uno por cada enemigo.

En esta nueva versión del juego para cada enemigo, además de su posición horizontal y vertical en la pantalla, queremos tener también su velocidad y su tipo de comportamiento. En donde su velocidad es un número entre 0 y 100, y su comportamiento es una de dos constantes: `ATACA` o `DEFIENDE`. Los enemigos de tipo `ATACA` son los que persiguen al personaje del jugador por toda la pantalla, mientras que los de tipo `DEFIENDE` son los que "cuidan" la comida y, por lo tanto, nunca se alejan mucho de ella.

Una manera de lograr esto sería empleando cuatro variables:

```
var xEnemigos;
var yEnemigos;
var velocidadEnemigos;
var tipoEnemigos;
```

Y en cada una de ellas almacenar una referencia a un arreglo con cuatro elementos, uno por cada enemigo.

Pero existe una mejor manera de almacenar los datos de cada enemigo: empleando *ambientes*. Ya hemos estado empleando ambientes. Tenemos un ambiente `Global` que almacena todas las variables globales y los procedimientos que definimos. Y al llamar un procedimiento o al definir una variable dentro de un bloque también hemos estado creando ambientes locales.

Ya vimos que los arreglos nos permiten almacenar múltiples valores empleando una sola variable. Y accesamos cada uno de estos valores, los elementos del arreglo, por medio de un número llamado subíndice. Con los ambientes también podemos emplear una sola variable para almacenar varios valores, nada más que en esto caso se accesan cada uno de estos valores por medio de un nombre.

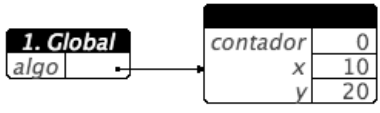
Esto va a quedar mucho más claro viendo un ejemplo. Haz click en y abre el programa `ambientes.sj`:

```
var algo = {x: 10, y: 20, contador: 0};

print(algo.x);
print(algo.y);

algo.x = 15;
algo.contador++;

print(algo);
```

Haz click en para empezar a ejecutar el programa paso por paso. La primera línea del programa dice:

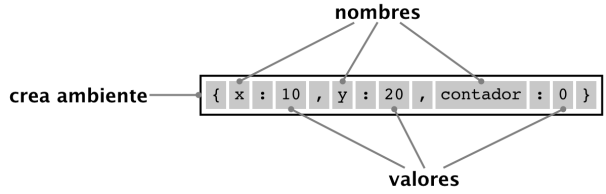
```
var algo = {x: 10, y: 20, contador: 0};
```

Haz click una vez más en para ejecutar esa línea; la vista de la memoria muestra:



La computadora creó un nuevo ambiente y almacenó en la variable `algo` una referencia a ese ambiente. El ambiente contiene tres variables `contador`, `x`

y `y` con los valores `0`, `10` y `20` respectivamente. Así fue como la computadora interpretó la creación del ambiente:



En cualquier parte de un programa puedes crear un ambiente colocando una lista de pares de nombres y valores entre llaves. Cada par consiste de un nombre para una variable separado de su valor por dos puntos (`:`). Y los pares van separados por comas.

Las dos líneas siguientes del programa muestran cómo accesar las variables que están dentro de ese ambiente:

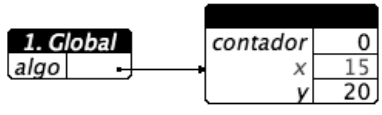
```
print(algo.x);
print(algo.y);
```

Simplemente pones el nombre de la variable que contiene la referencia al ambiente (en este caso es `algo`) seguido de un punto y el nombre de la variable dentro de ese ambiente. Usa el botón ⏭ para ejecutar paso por paso esas dos líneas y comprueba que te despliega `10` y `20`, los valores de `x` y `y`, en el Log.

La siguiente línea muestra que también puedes modificar alguna de las variables dentro de ese ambiente:

```
algo.x = 15;
```

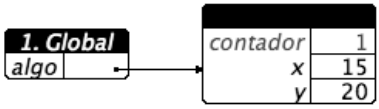
Haz click en ⏭ para ejecutarla; la vista de la memoria muestra:



La línea que sigue también muestra que se pueden emplear las variables de ese ambiente como cualquier otra variable. En este caso empleando el operador `++` para incrementar en uno el valor de `algo.contador`:

```
algo.contador++;
```

Al hacer click en ▶| la vista de la memoria muestra:



La última línea del programa muestra que le puedes pasar una referencia a un ambiente como argumento a un procedimiento:

```
print(algo);
```

En este caso estamos pasándole al procedimiento predefinido `print` la referencia al ambiente que está almacenada dentro de `algo`; en el Log aparece:

```
{
  contador: 1,
  x: 15,
  y: 20
}
```

Arreglos de ambientes

Es posible combinar de muchas maneras diferentes los arreglos con los ambientes. Vamos a ver un ejemplo de un arreglo cuyos elementos son referencias a ambientes. Haz click en 📂 y abre el programa `arreglosDeAmbientes.sj`:

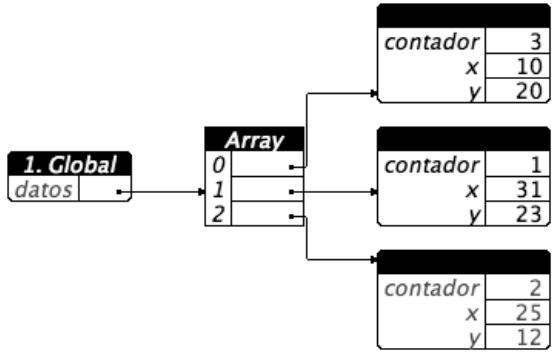
```
var datos = [
  {x: 10, y: 20, contador: 3},
  {x: 31, y: 23, contador: 1},
  {x: 25, y: 12, contador: 2}
];

print(datos[0].x);
print(datos[1].y);
print(datos[2].contador);

datos[1].y = 19;

muestraYModifica(amb) {
  print(amb.x);
  print(amb.y);
  print(amb.contador);
  amb.contador++;
```

```
}

for (var i = 0; i < length(datos); i++)
  muestraYModifica(datos[i]);
```

El programa empieza con:

```
var datos = [
  {x: 10, y: 20, contador: 3},
  {x: 31, y: 23, contador: 1},
  {x: 25, y: 12, contador: 2}
];
```

Aquí estamos creando un arreglo con tres elementos, cada uno de los cuales es una referencia a un ambiente. Usa el botón ⏭ para ejecutar estas líneas paso por paso; la vista de la memoria queda así:



La variable `datos` contiene una referencia a un arreglo de tres elementos, cada uno de los cuales es una referencia a un ambiente.

Nota

Aunque lo que contiene la variable `datos` es una referencia a un arreglo, muchas veces se dice que "`datos` contiene un arreglo" o inclusive se habla de "el arreglo `datos`". Aunque esto no es estrictamente cierto, permite hablar más fácilmente acerca de lo que hace un programa y no causa ningún problema, siempre y cuando recordemos que en realidad lo que contiene es una referencia a un arreglo.

Las líneas siguientes muestran cómo se puede accesar el contenido de esos ambientes:

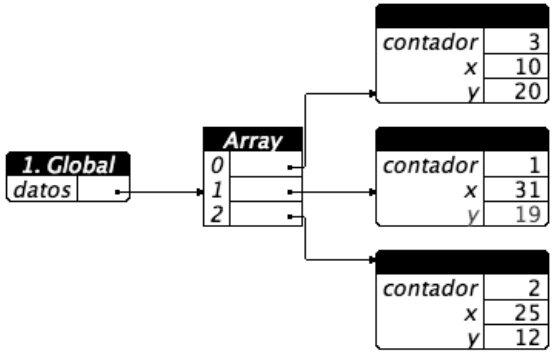
```
print(datos[0].x);
print(datos[1].y);
print(datos[2].contador);
```

Simplemente usamos un subíndice entre corchetes después del nombre de la variable `datos`, para accesar el ambiente deseado del arreglo, y después ponemos un punto seguido del nombre de la variable dentro de ese ambiente. Emplea el botón ⏭ para ejecutar esas líneas paso por paso y comprueba que se despliegan en el Log el valor de `x` del primer ambiente, el valor de `y` del segundo ambiente y el valor de `contador` del tercer ambiente.

La línea siguiente muestra que de la misma manera se puede modificar el contenido de alguno de esos ambientes:

```
datos[1].y = 19;
```

Al hacer click en ⏭ para ejecutar esa línea, la vista de la memoria queda así:



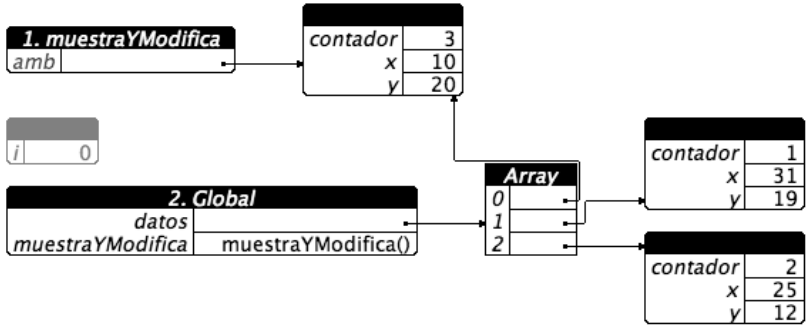
A continuación el programa define un procedimiento `muestraYModifica` que espera una referencia a un ambiente como argumento y, posteriormente, emplea un `for` para llamar a ese procedimiento con cada uno de los ambientes:

```
muestraYModifica(amb) {
  print(amb.x);
  print(amb.y);
  print(amb.contador);
  amb.contador++;
}

for (var i = 0; i < length(datos); i++)
  muestraYModifica(datos[i]);
```

Al ejecutar la línea que dice:

```
muestraYModifica(datos[i]);
```

Como tenemos `datos[i]`, sin un punto ni nombre de variable después, lo que le estamos pasando como argumento al procedimiento `muestraYModifica` es ese elemento del arreglo datos, el cual es una referencia a un ambiente. Sigue ejecutando el programa paso por paso; al llamar al procedimiento `muestraYModifica` la vista de la memoria queda así:

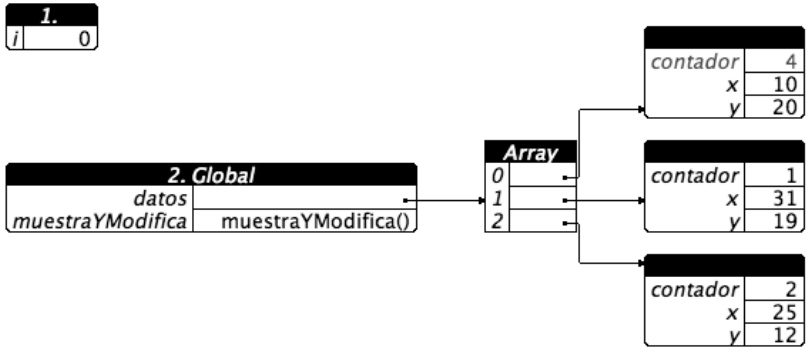


Dentro del ambiente local del procedimiento `muestraYModifica` la variable `amb` contiene una referencia al primer ambiente del arreglo `datos`. Y el arreglo `datos` sigue conservando su referencia a este mismo ambiente como primer elemento. Como `amb` contiene una referencia a un ambiente, podemos accesar las variables de este ambiente simplemente usando un punto seguido del nombre de la variable.

Ejecuta el procedimiento paso por paso y observa cómo va desplegando el contenido del ambiente en el Log. Al ejecutar la última línea del procedimiento:

```
amb.contador++;
```

Se termina de ejecutar el procedimiento, pero podemos ver en la vista de la memoria que efectivamente incrementó en uno la variable `contador` dentro del primer ambiente del arreglo `datos`:



Sigue empleando el botón para terminar de ejecutar el programa y ver cómo va llamando al procedimiento `muestraYModifica` con cada uno de los ambientes del arreglo `datos`.

Un arreglo de ambientes dentro del juego

Haz click en y abre el programa `main.sj` para que veas cómo empleamos un arreglo de ambientes para tener enemigos con diferentes comportamientos dentro del juego.

Primero definimos dos contantes para representar los dos tipos de enemigos, los que persiguen al personaje del jugador y los que defienden la comida:

```
/* Constantes para los tipos de enemigos */
final ATACA = 1;
final DEFIENDE = 2;
```

También definimos una constante con la distancia máxima a la cual pueden alejarse de la comida los enemigos que la están defendiendo:

```
/* Constante para la distancia maxima al
   defender la comida */
final MAX_DIST_COMIDA = 12;
```

Ahora usamos una sola variable para almacenar ahí el arreglo de ambientes con toda la información necesaria para los enemigos:

```
/* Los enemigos */
var enemigos;
```

Y dentro del procedimiento `init` la inicializamos con un arreglo de cuatro ambientes, uno por cada enemigo:

```
init() {
  xPersonaje = 16;
  yPersonaje = 12;
  dirPersonaje = DERECHA;
  nuevaPosicionComida();
  enemigos = [
    {tipo: ATACA, velocidad: 60, x: MAX_X, y: MAX_Y},
    {tipo: ATACA, velocidad: 40, x: MIN_X, y: MAX_Y},
    {tipo: DEFIENDE, velocidad: 60, x: MAX_X, y: MIN_Y},
    {tipo: DEFIENDE, velocidad: 40, x: MIN_X, y: MIN_Y}
  ];
  puntos = 0;
  vidas = 3;
  clear();
  muestraPuntos();
  muestraVidas();
}
```

Para cada uno de los enemigos tenemos su tipo, velocidad, posición horizontal y posición vertical.

Modificamos los procedimientos `dibujaEnemigos`, `borraEnemigos` y `hayEnemigoEn` para que ahora empleen el arreglo de ambientes `enemigos`:

```
dibujaEnemigos() {
  for (var i = 0; i < length(enemigos); i++)
    putAt(ENEMIGO, enemigos[i].x, enemigos[i].y);
}

borraEnemigos() {
  for (var i = 0; i < length(enemigos); i++)
    putAt(ESPACIO, enemigos[i].x, enemigos[i].y);
}

hayEnemigoEn(x, y) {
  for (var i = 0; i < length(enemigos); i++)
    if (enemigos[i].x == x && enemigos[i].y == y)
      return true;
  return false;
}
```

Observa que ahora para los ciclos en vez de emplear un 4 como número mágico, ya estamos aprovechando la longitud del arreglo `enemigos`: `i < length(enemigos)`.

La parte más interesante del programa es cómo se maneja el comportamiento diferente para cada uno de los enemigos. El procedimiento mueveEnemigos ahora está así:

```
mueveEnemigos() {
  for (var i = 0; i < length(enemigos); i++) {
    if (random(100) > enemigos[i].velocidad)
      continue;
    switch (enemigos[i].tipo) {
      case ATACA:
        ataca(enemigos[i]);
        break;

      case DEFIENDE:
        defiende(enemigos[i]);
        break;
    }
  }
}
```

En vez de comparar el número generado al azar con una constante ahora lo comparamos con `enemigos[i].velocidad`, lo cual nos permite controlar de forma independiente la velocidad de cada uno de los enemigos. Para mover al enemigo, si el número generado al azar no fue mayor que su velocidad, entonces empleamos un `switch` sobre `enemigos[i].tipo` para dedicir, con base en el tipo del enemigo, si empleamos el procedimiento `ataca` o el procedimiento `defiende` para calcular su nueva posición. Al llamar a alguno de estos dos procedimientos le pasamos como argumento el ambiente del enemigo que debe mover.

El procedimiento `ataca` es similar al procedimiento que habíamos empleado originalmente para calcular la nueva posición de un enemigo:

```
ataca(enemigo) {
  var dx = xPersonaje - enemigo.x;
  var dy = yPersonaje - enemigo.y;
  if (abs(dx) > abs(dy)) {
    var xNueva = enemigo.x + sign(dx);
    if (!hayEnemigoEn(xNueva, enemigo.y))
```

```
      enemigo.x = xNueva;
  } else {
    var yNueva = enemigo.y + sign(dy);
    if (!hayEnemigoEn(enemigo.x, yNueva))
      enemigo.y = yNueva;
  }
}
```

Al llamar a este procedimiento le pasamos como argumento una referencia al ambiente del enemigo para el cual debe calcular su nueva posición en la pantalla. Dentro de `ataca` la referencia queda dentro del parámetro (variable local) `enemigo` con lo cual puede accesar su posición horizontal como `enemigo.x` y la vertical como `enemigo.y`.

El procedimiento `defiende` empieza por ver qué tan lejos se encuentra el enemigo de la comida:

```
defiende(enemigo) {
  var dx = xComida - enemigo.x;
  var dy = yComida - enemigo.y;
  if (abs(dx) + abs(dy) > MAX_DIST_COMIDA) {
    if (abs(dx) > abs(dy)) {
      var xNueva = enemigo.x + sign(dx);
      if (!hayEnemigoEn(xNueva, enemigo.y))
        enemigo.x = xNueva;
    } else {
      var yNueva = enemigo.y + sign(dy);
      if (!hayEnemigoEn(enemigo.x, yNueva))
        enemigo.y = yNueva;
    }
  } else
    ataca(enemigo);
}
```

En la variables `dx` y `dy` almacena la diferencia en su posición con respecto a la posición de la comida. Después calcula la distancia a la comida como la suma de los valores absolutos de esas diferencias: `abs(dx) + abs(dy)`; si esa distancia es mayor que `MAX_DIST_COMIDA` entonces se va hacia la comida, de lo contrario (si no está demasiado lejos de la comida) entonces le delega el trabajo al procedimiento `ataca` para irse hacia el personaje del jugador.

Nivel 9: Tiles modificables y colores

Redefiniendo tiles y colores se ve más profesional el juego

Para que se vea mejor el juego vamos a redefinir las imágenes y colores de algunos de los tiles.

En el menú de **Project** selecciona la opción **Switch to another project...** (**Project > Switch to another project...**) y selecciona el proyecto `Nivel_09`. Haz click en ▶ para probar el programa. Es exactamente el mismo juego que en el nivel anterior, pero ahora los tiles empleados para el personaje, los enemigos y la comida tienen forma y colores más apropiados para el juego.

Tiles modificables

Ya vimos que la pantalla está dividida en 768 posiciones organizadas en 32 columnas por 24 renglones; en cada una de esas posiciones podemos colocar cualquiera de los 256 tiles disponibles. Cada tile tiene una imagen cuadrada que está formada por 64 puntos, organizados en 8 columnas por 8 renglones. Por ejemplo, así es como está formada la imagen para la letra "`A`":

Estos puntos que se emplean para desplegar una imagen en la pantalla se llaman *pixeles*. Cuando inicia la ejecución de un programa cada uno de los tiles ya tiene una imagen predefinida empleando dos colores que podemos modificar con los procedimientos `setBackground` y `setForeground`. Vamos a ver que existe una manera de redefinir las imágenes de los tiles para adaptarlos

a las necesidades gráficas de nuestro juego; también vamos a ver que no necesitamos limitarnos únicamente a dos colores, se pueden emplear hasta 16 colores para las imágenes de los tiles.

Esto va a quedar mucho más claro viendo un ejemplo. Haz click en y abre el programa `tilesModificables.sj`:

```
putAt(65, 16, 12);
showAt("ABCD", 4, 20);
pause(.5);

setTilePixels(65, [1, 1, 1, 1, 1, 1, 1, 1,
                   1, 0, 0, 0, 0, 0, 0, 1,
                   1, 0, 1, 1, 1, 1, 0, 1,
                   1, 0, 1, 0, 0, 1, 0, 1,
                   1, 0, 1, 0, 0, 1, 0, 1,
                   1, 0, 1, 1, 1, 1, 0, 1,
                   1, 0, 0, 0, 0, 0, 0, 1,
                   1, 1, 1, 1, 1, 1, 1, 1]);
pause(.5);

putAt(65, 0, 0);
showAt("ABABABA", 10, 20);
pause(.5);

setTilePixels(65, [0, 0, 1, 1, 1, 1, 0, 0,
                   0, 1, 1, 1, 1, 1, 1, 0,
                   1, 0, 0, 1, 1, 0, 0, 1,
                   1, 0, 0, 1, 1, 0, 0, 1,
                   1, 0, 0, 1, 1, 0, 0, 1,
                   1, 0, 1, 0, 0, 1, 0, 1,
                   0, 0, 1, 0, 0, 1, 0, 0,
                   0, 1, 1, 0, 1, 1, 0, 0]);
pause(.5);

setTileColor(0, 0, 0, 0);
setTileColor(1, 31, 31, 0);
pause(.5);

setTilePixels(65, [ 0,  1,  2,  3,  4,  5,  6,  7,
                    0,  1,  2,  3,  4,  5,  6,  7,
                    0,  1,  2,  3,  4,  5,  6,  7,
                    0,  1,  2,  3,  4,  5,  6,  7,
```

```
                    8,  9, 10, 11, 12, 13, 14, 15,
                    8,  9, 10, 11, 12, 13, 14, 15,
                    8,  9, 10, 11, 12, 13, 14, 15,
                    8,  9, 10, 11, 12, 13, 14, 15]);

setTileColor(2, 31, 24, 0);
setTileColor(3, 31, 16, 0);
setTileColor(4, 31, 8, 0);
setTileColor(5, 31, 0, 0);
setTileColor(6, 31, 0, 8);
setTileColor(7, 31, 0, 16);
setTileColor(8, 31, 0, 24);
setTileColor(9, 31, 0, 31);
setTileColor(10, 24, 0, 31);
setTileColor(11, 16, 0, 31);
setTileColor(12, 8, 0, 31);
setTileColor(13, 0, 0, 31);
setTileColor(14, 0, 16, 31);
setTileColor(15, 0, 31, 31);
pause(.5);

for (var y = 0; y < 24; y++)
  for (var x = 0; x < 32; x++)
    putAt(65, x, y);
pause(1);
```

Haz click en ▶ para ejecutar el programa. Aparecen unas letras en la pantalla, la imagen de la letra "A" se transforma en imágenes diferentes y, al final, la pantalla se llena con un patrón multicolor que se repite.

Ahora haz click en ⏭ para ejecutar el programa paso por paso y que puedas ir siguiendo la descripción de lo que hace cada parte del programa.

Las primeras líneas:

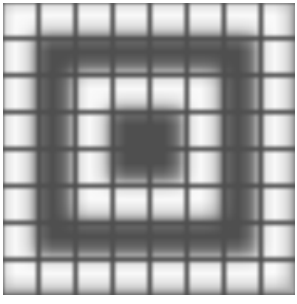
```
putAt(65, 16, 12);
showAt("ABCD", 4, 20);
pause(.5);
```

simplemente emplean los procedimientos `putAt` y `showAt` para desplegar la letra "A" (tile número 65) y el mensaje "ABCD" en la pantalla. Y hacen una breve pausa para que nos dé tiempo de verlas.

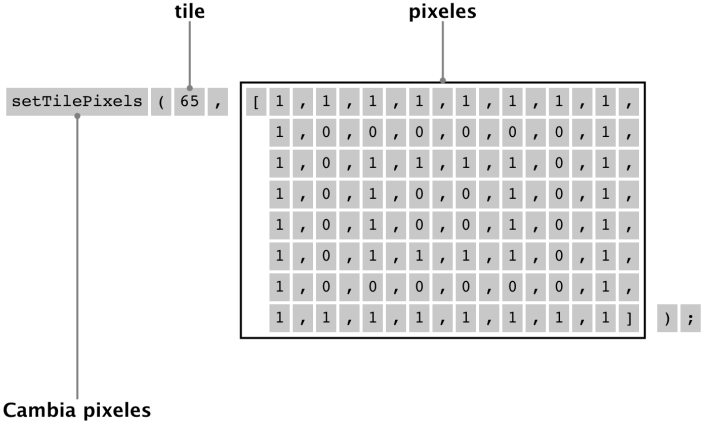
La siguiente parte del programa:

```
setTilePixels(65, [1, 1, 1, 1, 1, 1, 1, 1,
                   1, 0, 0, 0, 0, 0, 0, 1,
                   1, 0, 1, 1, 1, 1, 0, 1,
                   1, 0, 1, 0, 0, 1, 0, 1,
                   1, 0, 1, 0, 0, 1, 0, 1,
                   1, 0, 1, 1, 1, 1, 0, 1,
                   1, 0, 0, 0, 0, 0, 0, 1,
                   1, 1, 1, 1, 1, 1, 1, 1]);
pause(.5);
```

emplea el procedimiento predefinido `setTilePixels` para modificar la imagen del tile número 65. Emplea el botón ⏭ para ejecutarlo paso por paso y observa cómo en los dos lugares de la pantalla donde teníamos una letra "A" su imagen se transforma en dos cuadrados, uno dentro del otro:



Este es el significado del procedimiento `setTilePixels`:



El procedimiento `setTilePixels` espera dos argumentos. El primero es el número del tile para el cual se desea modificar la imagen. El segundo es un arreglo con 64 números que indican el color a emplear para cada uno de los pixeles; un `0` indica el color que se modifica con `setBackground` y un `1` el color que se modifica con `setForeground`. Los primeros ocho números son los colores para el primer renglón de pixeles, los ocho siguientes son para el segundo renglón y así sucesivamente hasta llegar al octavo renglón.

Las líneas siguientes en el programa:

```
putAt(65, 0, 0);
showAt("ABABABA", 10, 20);
pause(.5);
```

emplean `putAt` y `showAt` para colocar el tile número 65, que normalmente tiene como imagen la letra "A", en la pantalla. Emplea el botón ⏭ para ejecutarlas paso por paso y observa que en vez de una "A" ahora aparece un cuadrado dentro de otro, porque esa es la nueva imagen para el tile número 65.

Las líneas siguientes:

```
setTilePixels(65, [0, 0, 1, 1, 1, 1, 0, 0,
                   0, 1, 1, 1, 1, 1, 1, 0,
                   1, 0, 0, 1, 1, 0, 0, 1,
                   1, 0, 0, 1, 1, 0, 0, 1,
                   1, 0, 0, 1, 1, 0, 0, 1,
                   1, 0, 1, 0, 0, 1, 0, 1,
                   0, 0, 1, 0, 0, 1, 0, 0,
                   0, 1, 1, 0, 1, 1, 0, 0]);
pause(.5);
```

Vuelven a modificar la imagen que corresponde al tile número 65. Ejecútalas con el botón ⏭ y observa cómo en todos los lugares de la pantalla donde había un cuadrado dentro de otro ahora tenemos esta imagen:

Las líneas siguientes:

```
setTileColor(0, 0, 0, 0);
setTileColor(1, 31, 31, 0);
pause(.5);
```

muestran que, en vez de emplear `setBackground` y `setForeground`, también se puede emplear el procedimiento predefinido `setTileColor` para cambiar los colores. Usa el botón ▶| para ejecutar estas líneas paso por paso y observa cómo el primer llamado a `setTileColor` cambia el color del fondo y el segundo cambia el color de las letras.

Esta es la manera en que se emplea el procedimiento `setTileColor`:

La computadora tiene 16 *registros* para almacenar los colores de los tiles. Un registro es una memoria especializada. Estos 16 registros están numerados del 0 al 15; para almacenar un color dentro de alguno de estos registros se llama al procedimiento `setTileColor` con cuatro argumentos. El primer argumento es el número del registro en el cual se desea almacenar el color, los otros tres argumentos son números entre 0 y 31 que especifican los valores para los componentes rojo, verde y azul del color deseado. Los 64 números que se le pasan dentro de un arreglo como segundo argumento a `setTilePixels` son en realidad los números de los registros de donde la computadora toma los colores para pintar cada uno de esos pixeles en la pantalla. El procedimiento `setBackground` que hemos estado usando hasta ahora lo único que hace

es almacenar los valores de rojo, verde y azul dentro del registro número cero; el procedimiento `setForeground` almacena los valores en el registro número uno.

Las líneas siguientes muestran un ejemplo donde ya se emplean los 16 registros de color disponibles para las imágenes de los tiles:

```
setTilePixels(65, [ 0,  1,  2,  3,  4,  5,  6,  7,
                    0,  1,  2,  3,  4,  5,  6,  7,
                    0,  1,  2,  3,  4,  5,  6,  7,
                    0,  1,  2,  3,  4,  5,  6,  7,
                    8,  9, 10, 11, 12, 13, 14, 15,
                    8,  9, 10, 11, 12, 13, 14, 15,
                    8,  9, 10, 11, 12, 13, 14, 15,
                    8,  9, 10, 11, 12, 13, 14, 15]);

setTileColor(2, 31, 24, 0);
setTileColor(3, 31, 16, 0);
setTileColor(4, 31, 8, 0);
setTileColor(5, 31, 0, 0);
setTileColor(6, 31, 0, 8);
setTileColor(7, 31, 0, 16);
setTileColor(8, 31, 0, 24);
setTileColor(9, 31, 0, 31);
setTileColor(10, 24, 0, 31);
setTileColor(11, 16, 0, 31);
setTileColor(12, 8, 0, 31);
setTileColor(13, 0, 0, 31);
setTileColor(14, 0, 16, 31);
setTileColor(15, 0, 31, 31);
pause(.5);
```

Emplea el botón ⏭ para ejecutarlas paso por paso y observa cómo inicialmente partes del tile que emplean los registros de color del dos al quince se ven negras (porque al iniciar la ejecución del programa esos registros contienen ceros para el rojo, verde y azul). Pero al ir ejecutando los llamados a `setTileColor` van cambiando a los colores seleccionados.

El final del programa:

```
for (var y = 0; y < 24; y++)
  for (var x = 0; x < 32; x++)
```

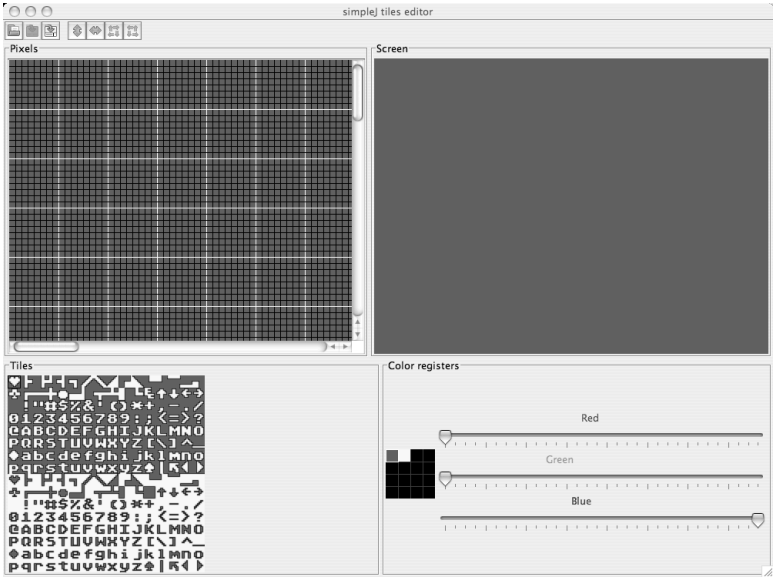
```
    putAt(65, x, y);
pause(1);
```

Usa un ciclo anidado dentro de otro para llenar la pantalla con el tile número 65. Usa el botón ⏭ para ejecutarlo paso por paso y que veas cómo se va cubriendo la pantalla con el mismo patrón multicolor. Recuerda que en cualquier momento puedes dar un doble click sobre la última línea del programa para que ejecute rápidamente las instrucciones hasta llegar ahí.

El simpleJ tiles editor

Los tiles modificables permiten hacer juegos empleando diferentes imágenes y más de dos colores. Pero es un poco tedioso andar definiendo cada una de esas imágenes con un arreglo de 64 números. Para facilitar la creación de estas imágenes existe el simpleJ tiles editor que te permite crear las imágenes empleando el mouse.

Ejecuta el tiles editor; aparece una ventana como esta:



La ventana está dividida en cuatro áreas:

Pixels	Aquí es donde puedes modificar los pixeles de la imagen de cada tile.

Screen	Sirve para colocar los tiles que quieres editar y que puedas ver cómo se ven uno junto al otro.
Tiles	Aquí puedes seleccionar el tile que quieres colocar en el área de *Screen* para poder editarlo. También te permite copiar la imagen de un tile a otro.
Color registers	Sirve para seleccionar el color que quieres almacenar en cada uno de los 16 registros de color.

Arriba de estas cuatro áreas hay unos botones que corresponden a las acciones que vas a usar más frecuentemente. También puedes tener acceso a estas acciones desde los menúes que se encuentran arriba de esos botones.

Nota

En el CD que viene con este libro hay un video que se llama "`tiles editor.avi`". Si ves ese video va a ser más sencillo que entiendas como emplear el simpleJ tiles editor.

Haz click sobre el tile que tiene la imagen de la letra "A" en el área de *Tiles*. Ese tile queda seleccionado y aparece rodeado de un borde cuadrado de color negro. Puedes hacer click sobre cualquiera de los tiles para seleccionarlo.

Ahora, con el tile que tiene la imagen de la letra "A" seleccionado, haz click en la esquina superior izquierda del área *Screen*. Aparece ahí el tile con la imagen de la letra "A" y también aparece, en más grande, en la esquina superior izquierda del área *Pixels*. El área *Pixels* muestra exactamente lo mismo que se encuentra en el área *Screen*, pero más grande para que sea más sencillo seleccionar los pixeles que quieres modificar.

Dentro del área *Pixels*, haz click sobre cualquiera de los pixeles que están de color blanco. Al hacer click sobre el pixel su color cambia de blanco a azul; la imagen del tile también aparece modificada en las áreas *Screen* y *Tiles*. Ahora vuelve a presionar el botón del mouse sobre el área *Screen* y mantenlo apoyado mientras mueves el mouse sobre los pixeles del tile con la letra "A": puedes ir "pintando" con el color del fondo. También puedes presionar el botón del mouse sobre el área *Screen* y mantenerlo presionado mientras mueves el mouse para ir pintando con un tile dentro de esa área.

Dentro del área *Color registers* hay 16 cuadrados organizados en 4 renglones por 4 columnas. Cada uno de esos cuadrados representa uno de los 16 registros de color que se emplean para los colores de los tiles. El primer cuadrado está de color azul, el segundo de color blanco y todos los demás están de color negro. Haz click sobre el cuadrado de color blanco para seleccionarlo. Ahora vuelve a pintar sobre el tile que está en la esquina superior izquierda en el área *Screen*: los pixeles se pintan de blanco.

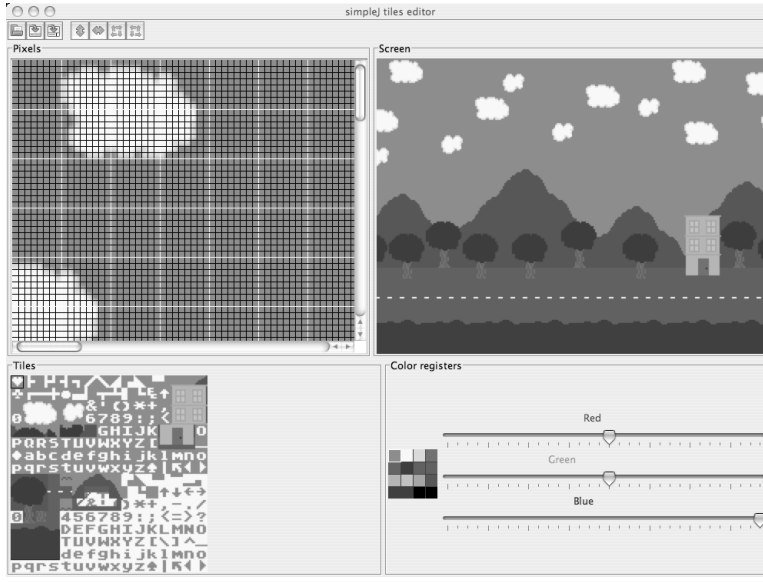
Haz click sobre el cuadrado negro que está justo a la derecha del cuadrado blanco en el área *Color registers* para seleccionar ese registro de color. Ahora pinta con ese color en el área *Screen*: los pixeles quedan de color negro.

A la derecha de los 16 cuadrados que representan los registros de color, hay tres sliders con las etiquetas `Red`, `Green` y `Blue` (que en inglés son los nombres para los colores rojo, verde y azul). Puedes desplazar cualquiera de esos tres sliders empleando el mouse para cambiar el valor de los componentes rojo, verde y azul del registro de color que tengas seleccionado. Pruébalo y observa cómo cambia el color del cuadrado que representa ese registro de color; también cambia el color de los pixeles que emplean ese registro de color en las áreas *Pixels*, *Screen* y *Tiles*.

Ya vimos que en el área *Tiles* puedes hacer un click para seleccionar el tile que deseas colocar en el área *Screen*. A veces es útil crear tiles que son similares a un tile que ya existe, vamos a ver cómo puedes copiar la imagen de un tile a otro. Estando seleccionado el tile que corresponde a la letra "A", haz click *con el botón derecho del mouse en vez del izquierdo* sobre el tile que tiene una imagen de un corazón (hasta arriba y a la izquierda dentro del área Tiles). El tile que estaba seleccionado se copia a esa posición y queda seleccionado. Ahora haz click sobre el botón  y observa cómo la imagen de ese tile se pone de cabeza. Si vuelves a hacer click en ese botón, la imagen del tile regresa a como estaba. También puedes emplear el botón  para voltear la imagen de izquiera a derecha o los botones  y para rotar la imagen 90 grados en el sentido de las manecillas de un reloj o en el sentido contrario.

El tiles editor permite almacenar los registros de color, las imágenes de los tiles y el contenido del área *Screen* en un archivo que después se puede leer desde un programa para emplearlos dentro de un juego. Haz click en el botón  para abrir un archivo con un ejemplo. Como has estado haciendo cambios en el tiles editor primero te va a aparecer una ventana preguntando si quieres descartar esos cambios. Haz click sobre el botón **Yes** para confirmar que si deseas descartar los cambios. Ahora te aparece una nueva ventana donde puedes ver las

carpetas con los proyectos de simpleJ, hay una carpeta por cada proyecto. Entra a la carpeta `Nivel_09` y ahí selecciona abrir el archivo `tilesEditorDemo.tmap`. Aparece esto en el tiles editor:



Observa cómo al modificar las imágenes y colores de unos cuantos tiles se puede dibujar un paisaje con un edificio, una carretera, un río, árboles, montañas y nubes.

Ya puedes cerrar el tiles editor. En un momento vamos a ver cómo leer el contenido de un archivo creado con el tiles editor para poder emplearlo dentro de un juego, pero primero vamos a ver un poco acerca de estructuras anidadas.

Estructuras anidadas

En simpleJ existen dos tipos de estructuras: los arreglos y los ambientes. Cuando hay una estructura dentro de otra estructura se dice que tenemos estructuras anidadas. En el nivel anterior ya vimos ejemplos de arreglos que contienen ambientes, pero también es posible tener otros tipos de anidamiento. Por cierto, aunque decimos que una estructura contiene a otra, en realidad lo que ocurre es que una estructura contiene una referencia a otra estructura, no están realmente una dentro de la otra.

Nota

Al leer las explicaciones de está sección por primera vez puede que te parezcan un poco complejas. No te desesperes. Son más sencillas de lo que parecen. Lee con cuidado, siguiendo la explicación en los diagramas que aparecen en la vista de memoria y verás que en realidad describen algo bastante sencillo.

Haz click en el botón y abre el programa `estructurasAnidadas.sj`:

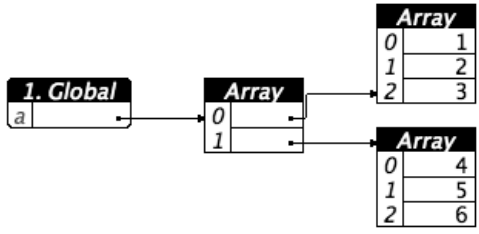
```
/* Arreglo de arreglos */
var a = [[1, 2, 3],
         [4, 5, 6]];

print(a[0]);
print(a[1]);
print(a[0][0]);
print(a[0][1]);
a[0][2] = 10;
a[1][0] = 20;


/* Ambiente con arreglo */
var b = {p: 1, q: [10, 20, 30]};

print(b.p);
print(b.q);
print(b.q[0]);
b.q[1] = 100;


/* Mayor anidamiento */
var c = {a: [[100, 200, 300],
             [400, 500, 600],
             [700, 800, 900]]};

print(c.a);
print(c.a[0]);
print(c.a[1][2]);
c.a[2][0] = -1;
pause(1);
```

Ahora haz click en para empezar a ejecutarlo paso por paso.

Las primeras líneas del programa son:

```
/* Arreglo de arreglos */
var a = [[1, 2, 3],
         [4, 5, 6]];
```

Haz click dos veces en ⏭ para ejecutarlas. La vista de la memoria queda así:



La variable `a` contiene una referencia a un arreglo, el cual a su vez contiene referencias a otros dos arreglos.

Después vienen dos líneas que dicen:

```
print(a[0]);
print(a[1]);
```

Esas líneas despliegan en el Log el contenido de los dos elementos del arreglo al cual hace referencia la variable `a`. Haz click en ⏭ y en Log aparece `[1, 2, 3]` que es el arreglo al que se hace referencia en `a[0]`. Haz un vez más click en ⏭ y ahora aparece en el Log `[4, 5, 6]` que es el arreglo al que se hace referencia en `a[1]`.

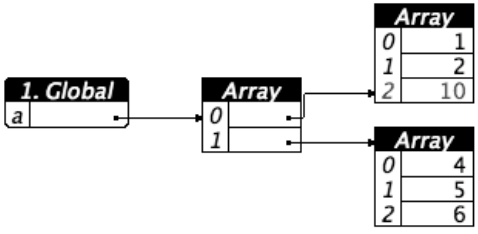
También es posible accesar directamente los elementos de los areglos que se encuentran más anidados. Las líneas siguientes muestran cómo hacerlo:

```
print(a[0][0]);
print(a[0][1]);
a[0][2] = 10;
a[1][0] = 20;
```

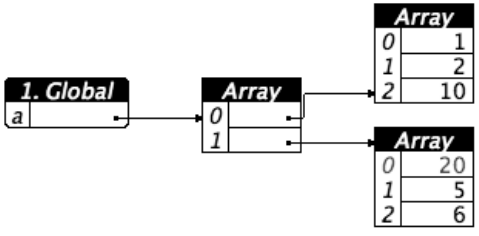
La expresión `a[0][0]` indica que queremos accesar el primer elemento del arreglo al que hace referencia la variable a (la parte que dice `a[0]`) y después, como ese también es un arreglo, queremos accesar su primer elemento (el segundo `[0]`). Haz click en ⏭ y fíjate cómo en el Log despliega un `1`, que es el contenido del primer elemento del primer arreglo. La expresión `a[0][1]` in-

dica que queremos desplegar el segundo elemento (el `[1]`) del primer arreglo al que hace referencia la variable a (el `a[0]`). Haz click una vez más en ⏭ y en el Log aparece ahora un `2`, que es el contenido del segundo elemento del primer arreglo.

La expresión `a[0][2] = 10` indica que hay que almacenar un `10` en el tercer elemento (el `[2]`) del primer arreglo al que hace referencia la variable `a` (el `a[0]`). Haz click en ⏭ y la vista de la memoria queda así:



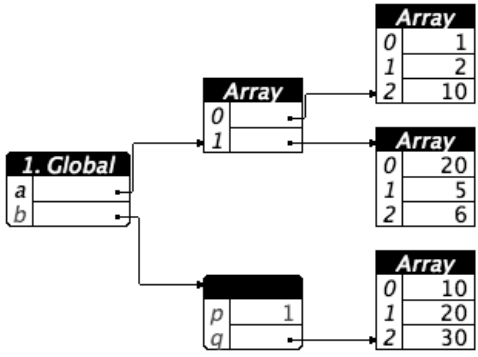
La expresión `a[1][0] = 20` indica que hay que almacenar un `20` en el primer elemento (el `[0]`) del segundo arreglo al que hace referencia la variable `a` (el `a[1]`). Haz click en ⏭ y la vista de la memoria queda así:



La linea siguiente:

```
/* Ambiente con arreglo */
var b = {p: 1, q: [10, 20, 30]};
```

Crea un ambiente que contiene un arreglo. Haz click en para ejecutarla y la vista de la memoria queda así:



Las dos líneas siguientes despliegan en el Log el contenido de las variables `p` y `q` que están dentro del ambiente al que hace referencia la variable `b`:

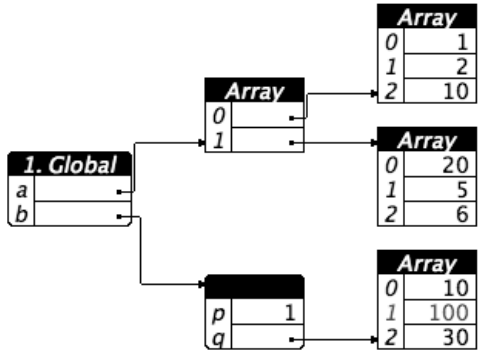
```
print(b.p);
print(b.q);
```

Haz click dos veces en ⏭ y en el Log se despliega `1` y `[10, 20, 30]` que son el contenido de esas variables.

También es posible accesar directamente los elementos del arreglo que se encuentra dentro del ambiente. Las líneas siguientes muestran cómo hacerlo:

```
print(b.q[0]);
b.q[1] = 100;
```

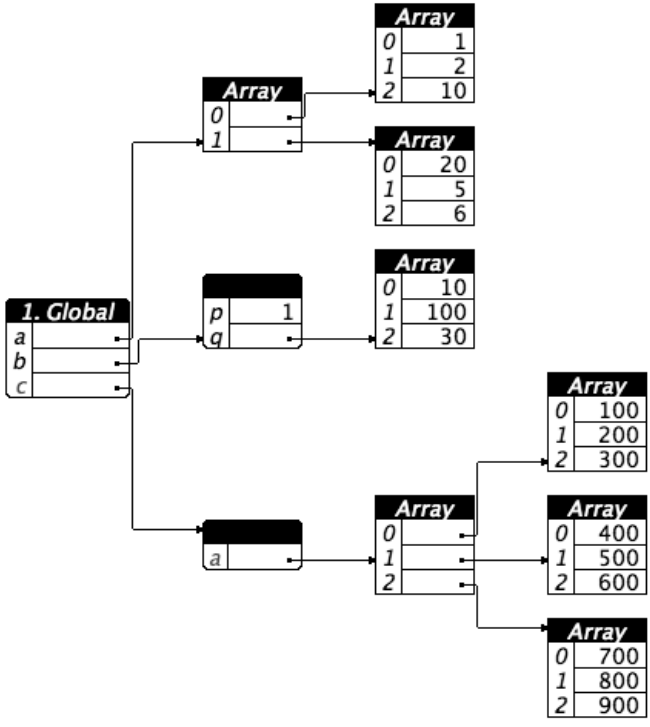
La expresión `b.q[0]` permite accesar el primer elemento del arreglo al que hace referencia la variable `q` que se encuentra dentro del ambiente al que hace referencia la variable `b`. Haz click en ⏭ y en el Log se despliega `10`. La expresión `b.q[1] = 100` indica que hay que almacenar un `100` dentro del segundo elemento del arreglo al que hace referencia la variable `q` dentro del ambiente al que hace referencia la variable `b`. Haz click en ⏭ y la vista de la memoria queda así:



Las estructuras se pueden anidar hasta cualquier nivel de profundidad. La siguiente parte del programa crea un ambiente que contiene un arreglo de arreglos:

```
/* Mayor anidamiento */
var c = {a: [[100, 200, 300],
             [400, 500, 600],
             [700, 800, 900]]};
```

Haz click tres veces en ⏭ para ejecutar esas líneas y la vista de la memoria queda así:



La variable `c` contiene una referencia a un ambiente, el cuál a su vez contiene una variable `a` con una referencia a un arreglo de tres elementos, cada uno de los cuales es a su vez un arreglo con tres elementos.

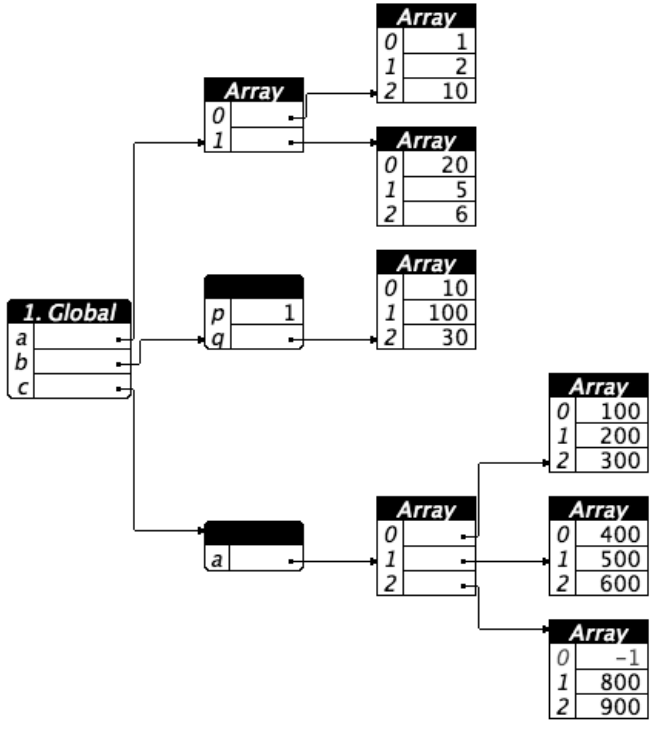
Las siguientes líneas muestran cómo accesar los diferentes niveles de esa estructura anidada:

```
print(c.a);
print(c.a[0]);
print(c.a[1][2]);
c.a[2][0] = -1;
```

La expresión `c.a` hace referencia al arreglo de arreglos al que hace referencia la variable `a` dentro del ambiente al que hace referencia la variable `c`. Haz click en ⏭ y en el Log se despliega `[[100, 200, 200], [400, 500, 600], [700, 800, 900]]`. La expresión `c.a[0]` hace referencia al primero de los tres arreglos. Haz click en ⏭ y en el Log se despliega `[100, 200, 300]`.

La expresión `c.a[1][2]` hace referencia al tercer elemento (el `[2]`) del arreglo, que es el segundo elemento (el `[1]`) del arreglo al que hace referencia la variable `a` dentro del ambiente al que hace referencia la variable `c`. Haz click en ⏭ y en el Log se despliega un `600`.

Con la expresión `c.a[2][0] = -1` se almacena un `-1` en el primer elemento (el `[0]`) del arreglo, que es el tercer elemento (el `[2]`) del arreglo al que hace referencia la variable `a` dentro del ambiente al que hace referencia la variable `c`. Haz click en ⏭ y la vista de la memoria muestra:



Cómo leer un archivo creado con el simpleJ tiles editor

Haz click en 📂 y abre el programa `tilesEditor.sj`. Ahora haz click en ▶ para ejecutarlo. Aparece en el área de Video el paisaje que está en el archivo `tilesEditorDemo.tmap` que fue creado con el simpleJ tiles editor.

El programa es bastante corto:

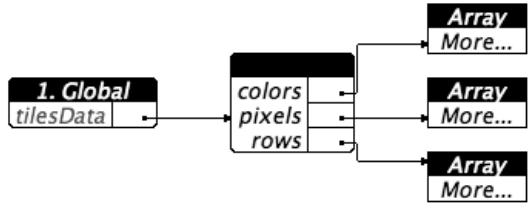
```
/* Leer definiciones de tiles creadas con
   el tiles editor */
var tilesData = readTilesFile("tileseditorDemo.tmap");

/* Poner colores en el mapa de colores */
for (var i = 0; i < 16; i++)
  setTileColor(i, tilesData.colors[i].red,
                  tilesData.colors[i].green,
                  tilesData.colors[i].blue);

/* Grabar nuevas definiciones de tiles */
for (var i = 0; i < 256; i++)
  setTilePixels(i, tilesData.pixels[i]);

/* Dibujar la pantalla */
for (var r = 0; r < 24; r++)
  for (var c = 0; c < 32; c++)
    putAt(tilesData.rows[r][c], c, r);

pause(1);
```

Ahora haz click en ▶| para empezar a ejecutarlo paso por paso.

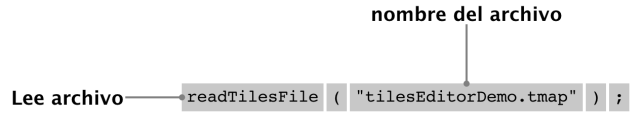
Las primeras líneas son las que se emplean para leer los datos almacenados en el archivo `tilesEditorDemo.tmap`:

```
/* Leer definiciones de tiles creadas con
   el tiles editor */
var tilesData = readTilesFile("tileseditorDemo.tmap");
```

Haz click una vez más en ▶| y la vista de la memoria queda así:



Este es el significado del procedimiento `readTilesFile`:



En realidad, `readTilesFile` es una función porque devuelve un resultado (que estamos almacenando dentro de la variable `tilesData`). Este resultado es un ambiente que contiene tres variables: `colors`, `pixels` y `rows`. Cada una de estas tres variables contiene una referencia a un arreglo.

Nota

En la vista de memoria los arreglos aparecen colapsados porque son bastante grandes y le tomaría mucho tiempo al devkit dibujarlos. Puedes expander cada uno de ellos haciendo un doble click sobre su rectángulo correspondiente, pero no te lo recomendamos a menos que tengas una computadora muy rápida.

La variable `colors` tiene una referencia a un arreglo con 16 elementos, uno para cada registro de color. Cada uno de estos 16 elementos contiene una referencia a un ambiente; cada uno de estos ambientes contiene tres variables: `red`, `green`, y `blue`; estas variables contienen un número entre 0 y 31 para indicar el valor de los componentes rojo, verde y azul para ese registro de color.

La parte que sigue del programa es la que recorre ese arreglo de 16 elementos para asignarle los colores leídos a los registros de color:

```
/* Poner colores en el mapa de colores */
for (var i = 0; i < 16; i++)
  setTileColor(i, tilesData.colors[i].red,
                  tilesData.colors[i].green,
                  tilesData.colors[i].blue);
```

Es simplemente un ciclo en donde la variable `i` va tomando sucesivamente los valores desde el 0 hasta el 15. En cada iteración empleamos la expresión `tilesData.colors[i]` para accesar el ambiente con los datos que corresponden al registro de color con ese número, seguido de `.red`, `.green` y `.blue` para accesar sus componentes rojo, verde y azul; éstos se emplean como argumentos para llamar al procedimiento `setTileColor`, para que modifique el contenido del registro de color correspondiente.

Emplea el botón ⏭ para ejecutar este ciclo paso por paso. En la primera iteración puedes ver que el color de fondo en el área de Video cambia a un azul más claro. Los otros cambios a los registros de color por ahora son invisibles porque no tenemos nada en la pantalla que emplee esos colores.

La variable `pixels` (dentro del ambiente devuelto por la función `readTilesFile`) contiene una referencia a un arreglo con 256 elementos, un elemento para cada tile modificable. Cada uno de esos elementos es a su vez una referencia a un arreglo con 64 números que indican el color a emplear para cada uno de los 8 por 8 pixeles que tiene la imagen de cada tile.

La parte que sigue del programa es la que recorre este arreglo de 256 elementos para asignarle su imagen a cada tile:

```
/* Grabar nuevas definiciones de tiles */
for (var i = 0; i < 256; i++)
  setTilePixels(i, tilesData.pixels[i]);
```

Aquí también tenemos un simple ciclo que recorre los 256 arreglos que hay dentro del arreglo `pixels` para redefinir la imagen de cada uno de los tiles, empleando el procedimiento `setTilePixels`. Puedes ejecutar este procedimiento paso por paso con el botón ⏭, pero es un poco tedioso ya que no se ve nada que cambie en el área de Video. Te recomendamos que mejor hagas un doble click sobre la línea que dice "`for (var r = 0; r < 24; r++)`" para ejecutar rápidamente todas las instrucciones hasta llegar a esa línea.

La variable `rows` contiene un arreglo con 24 elementos, uno por cada renglón de la pantalla, y cada uno de esos elementos es a su vez un arreglo con 32 números, uno por cada columna de la pantalla.

La siguiente parte del programa emplea dos ciclos anidados para colocar los tiles correspondientes a esos números en la pantalla:

```
/* Dibujar la pantalla */
for (var r = 0; r < 24; r++)
  for (var c = 0; c < 32; c++)
    putAt(tilesData.rows[r][c], c, r);
```

En el ciclo externo, la variable `r` toma sucesivamente los valores desde el 0 hasta el 23 y representa el renglón actual. En el ciclo interno, la variable `c` toma sucesivamente los valores desde el 0 hasta el 31 y representa una columna dentro de ese renglón. Dentro de esos ciclos se emplea la expresión `tilesDa-`

`ta.rows[r][c]` para accesar el arreglo que corresponde al renglón número `r` y, dentro de ese renglón, el valor que corresponde a la columna número `c`. Este valor se pasa como primer argumento al procedimiento `putAt` para indicarle cuál es el tile que debe colocar en la pantalla y empleando como segundo y tercer argumentos las variables `c` y `r` para indicarle en qué posición debe ponerlo. Emplea el botón ⏭ para ir ejecutando estos ciclos anidados paso por paso y observa cómo se va dibujando el paisaje en la pantalla. Si quieres puedes ir un poco más rápido haciendo un doble click en la línea que dice "`for (var r = 0; r < 24; r++)`" para ir llenando la pantalla de renglón en renglón.

Tiles redefinidos en el juego

Ejecuta el simpleJ tiles editor y emplea el botón 📂 para abrir el archivo `tiles.tmap` (que está dentro de la carpeta `Nivel_09`). Ahí puedes ver cómo se redefinieron las imágenes de 6 tiles y los registros de color para poder dibujar la comida, los enemigos y las flechas que representan al personaje.

Ahora cierra el tiles editor y regresa al devkit. Usa el botón 📂 del devkit para abrir el programa `main.sj`. La única diferencia con el programa del juego tal como estaba en el Nivel_08 consiste que en vez de tener estas líneas:

```
// Ponemos el fondo de color negro
setBackground(0, 0, 0);
```

que se empleaban para poner de color negro el fondo, ahora tenemos esto:

```
/* Leer definiciones de tiles creadas con
   el tiles editor */
var tilesData = readTilesFile("tiles.tmap");

/* Poner colores en el mapa de colores */
for (var i = 0; i < 16; i++)
  setTileColor(i, tilesData.colors[i].red,
                  tilesData.colors[i].green,
                  tilesData.colors[i].blue);

/* Grabar nuevas definiciones de tiles */
setTilePixels(PERSONAJE_ARRIBA,
              tilesData.pixels[PERSONAJE_ARRIBA]);
setTilePixels(PERSONAJE_ABAJO,
              tilesData.pixels[PERSONAJE_ABAJO]);
setTilePixels(PERSONAJE_DERECHA,
```

```
            tilesData.pixels[PERSONAJE_DERECHA]);
setTilePixels(PERSONAJE_IZQUIERDA,
            tilesData.pixels[PERSONAJE_IZQUIERDA]);
setTilePixels(COMIDA,
            tilesData.pixels[COMIDA]);
setTilePixels(ENEMIGO,
            tilesData.pixels[ENEMIGO]);
```

Estas líneas son muy similares al ejemplo que acabamos de ver en el programa `tilesEditor.sj`. Usamos la función `readTilesFile` para leer el archivo `tiles.map` y almacenamos el resultado dentro de la variable `tilesData`. Después empleamos un ciclo para almacenar los valores leídos del archivo dentro de los 16 registros de color. Finalmente, en vez de emplear un ciclo para redefinir los 256 tiles, únicamente modificamos los tiles cuyos números corresponden a las constantes `PERSONAJE_ARRIBA`, `PERSONAJE_ABAJO`, `PERSONAJE_DERECHA`, `PERSONAJE_IZQUIERDA`, `COMIDA` y `ENEMIGO`.

Nivel 10: Sprites

Los sprites sirven para animar personajes o vehículos

Hasta ahora hemos estado usando tiles para representar la comida, el personaje y sus enemigos. Aunque no funciona tan mal, en realidad los tiles se prestan más a ser empleados para dibujar el mundo (como el paisaje con tiles redefinidos que vimos en el nivel anterior) en el cual se mueven los personajes o vehículos. Para dibujar estos personajes o vehículos lo ideal es usar *sprites*. Los sprites son dibujos que se pintan delante del fondo y se pueden colocar en cualquier posición de la pantalla. Los tiles están limitados a colocarse en alguna posición dentro de las 32 columnas y 24 renglones de la pantalla, por lo cual al emplear tiles para representar a los personajes de un juego éstos no pueden hacer movimientos, ni horizontales ni verticales, de menos de 8 pixeles. Mientras que los sprites no tienen esta limitación, se pueden colocar en cualquier lugar de la pantalla y desplazarse, tanto horizontalmente como verticalmente, hasta en un solo pixel.

Veamos un ejemplo. En el menú de **Project** selecciona la opción **Switch to another project...** (**Project > Switch to another project...**) y selecciona el proyecto `Nivel_10`. Haz click en ▶ para probar el programa. Aparece el mismo paisaje que teníamos en el nivel anterior pero ahora tiene aviones, coches y barcos que se mueven sobre el fondo. En esta animación el fondo está dibujado con tiles redefinidos y los vehículos que se mueven están dibujados con sprites.

Características de los sprites

En simpleJ se pueden emplear hasta 32 sprites. A cada uno se le puede asignar, independientemente de los demás, su posición tanto vertical como horizontal con precisión de un pixel además de asignarle una de las 256 imágenes disponibles. Estas 256 imágenes están divididas en dos grupos de 128 imágenes cada uno. Hay 128 imágenes de 8 por 8 pixeles y 128 imágenes de 16 por 16 pixeles. Estas imágenes se pueden modificar de una manera similar a la que se emplea para modificar las imágenes de los tiles. Existen 15 registros de color para seleccionar los colores de los sprites, estos registros son completamente independientes de los 16 registros de color empleados para los tiles.

Esto va a quedar mucho más claro viendo un ejemplo. Haz click en  y abre el programa `sprites.sj`:

```
showAt("Sprites!", 2, 2);
pause(.5);

setSmallSpriteImage(0, 65);
for (var i = 0; i < 40; i++) {
  putSpriteAt(0, i * 2, i);
  pause(.02);
}
pause(.5);

setSmallSpriteImage(0, 20);
pause(.5);
for (var i = 39; i >= 0; i--) {
  putSpriteAt(0, i * 2, i);
  pause(.02);
}
pause(.5);

setSpriteColor(0, 31, 31, 0);
pause(.5);
for (var i = 0; i < 40; i++) {
  putSpriteAt(0, i * 2, i);
  pause(.02);
}
pause(.5);

setSmallSpritePixels(20,
  [15, 15,  1,  1,  1,  1, 15, 15,
   15,  1,  1,  1,  1,  1,  1, 15,
    1, 15, 15,  1,  1, 15, 15,  1,
    1, 15, 15,  1,  1, 15, 15,  1,
    1, 15, 15,  1,  1, 15, 15,  1,
    1, 15,  1, 15, 15,  1, 15,  1,
   15, 15,  1, 15, 15,  1, 15, 15,
   15,  1,  1, 15,  1,  1, 15, 15]);
setSpriteColor(1, 31, 0, 0);
for (var i = 39; i >= -8; i--) {
  putSpriteAt(0, i * 2, i);
  pause(.02);
}
for (var i = -8; i < 257; i++) {
```

```
  putSpriteAt(0, i, 20);
  pause(.02);
}
pause(.5);

putSpriteAt(0, 124, 92);
setLargeSpriteImage(0, 10);
setLargeSpritePixels(10,
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
);
pause(.5);

for (var i = 124; i >= -16; i--) {
  putSpriteAt(0, i, 92);
  pause(.02);
}
pause(.5);

for (var i = -16; i < 257; i++) {
  putSpriteAt(0, i, 92);
  pause(.02);
}
pause(1);
```

Haz click en ▶ para ejecutar el programa. Aparece el mensaje "`Sprites!`" en la pantalla; aparece una "A" que se mueve delante del mensaje; la "A" se convierte en una bola y se mueve de regreso; la bola cambia de color a amarillo y se vuelve a mover; la bola se convierte en un robot rojo que se mueve hasta

salir de la pantalla por la esquina superior izquierda; el robot entra a la pantalla por el lado izquierdo y se mueve hasta salir por el lado derecho; aparece en el centro de la pantalla un cuadrado relleno de amarillo con un cuadrado rojo adentro; el cuadrado se mueve hacia la izquierda hasta que sale de la pantalla; el cuadrado vuelve a entrar a la pantalla por el lado izquierdo y se mueve hasta que sale por el lado derecho.

Ahora haz click en ⏭ para ejecutar el programa paso por paso y que puedas ir siguiendo la descripción de lo que hace cada parte del programa.

Las primeras líneas:

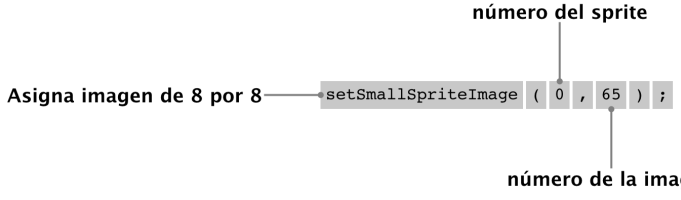
```
showAt("Sprites!", 2, 2);
pause(.5);
```

simplemente despliegan en la pantalla el mensaje "`Sprites!`" y hacen una pausa de medio segundo.

La siguiente parte del programa:

```
setSmallSpriteImage(0, 65);
for (var i = 0; i < 40; i++) {
  putSpriteAt(0, i * 2, i);
  pause(.02);
}
pause(.5);
```

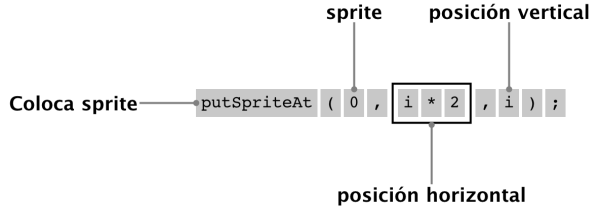
Empieza por emplear el procedimiento `setSmallSpriteImage` para asignarle la imagen de una letra "`A`" al sprite número cero. Así es como la computadora interpreta esa línea:



Al iniciar la ejecución de un programa las 128 imágenes pequeñas para los sprites (de 8 por 8 pixeles) son copias de las primeras 128 imágenes que se emplean para los tiles. Por lo tanto el `65` corresponde a la imagen de una "`A`". Haz click en el botón ⏭ para ejecutar esa línea. Aunque ya cambiamos la

imagen asignada al sprite número cero, no vemos ningún cambio porque no hemos colocado el sprite dentro de la pantalla.

Sigue haciendo click en el botón ⏭ para que entre al ciclo donde la variable `i` va tomando los valores desde el 0 hasta el 39 (menor que 40). Dentro de ese ciclo se llama al procedimiento `putSpriteAt` que se emplea para colocar un sprite en la pantalla. Este procedimiento se emplea así:



El primer argumento es el número del sprite que se desea colocar en la pantalla (un número entre 0 y 31). El segundo argumento es su posición horizontal, medida en pixeles desde el borde izquierdo de la pantalla. El tercer argumento es su posición vertical, medida en pixeles desde el borde superior de la pantalla. Al pasar por primera vez por esa línea ya vemos una "A" que aparece en la esquina superior izquierda de la pantalla. Sigue haciendo click en el botón ⏭ y observa cómo con los valores sucesivos de `i,` el sprite se mueve en cada iteración dos pixeles a la derecha y uno hacia abajo. Después de ejecutar unas cuantas iteraciones paso por paso, haz un doble click sobre la línea que está después del ciclo (la que dice `pause(.5);`) para que el programa se ejecute rápidamente hasta llegar a esa línea.

La parte que sigue dentro del programa:

```
setSmallSpriteImage(0, 20);
pause(.5);
for (var i = 39; i >= 0; i--) {
  putSpriteAt(0, i * 2, i);
  pause(.02);
}
pause(.5);
```

Vuelve a emplear el procedimiento `setSmallSpriteImage` ahora para cambiar la imagen del sprite por una bola (la imagen número 20). Haz click en el botón ⏭ y observa cómo cambia la imagen empleada para desplegar el sprite número cero. Después hay un ciclo donde la variable i toma los valores desde el 39 hasta el cero para que el sprite se mueva de regreso a la esquina de

la pantalla. Emplea el botón ⏭ para ejecutar unas cuantas iteraciones del ciclo paso por paso. Después haz un doble click sobre la línea después del ciclo (la que dice `pause(.5);`) para que el programa se ejecute rápidamente hasta llegar a esa línea.

Lo que sigue en el programa:

```
setSpriteColor(0, 31, 31, 0);
pause(.5);
for (var i = 0; i < 40; i++) {
  putSpriteAt(0, i * 2, i);
  pause(.02);
}
pause(.5);
```

Empieza por emplear el procedimiento `setSpriteColor` para cambiar el registro de color número cero a amarillo. Así es como se emplea este procedimiento:

El número del registro tiene que ser un valor entre 0 y 14 (únicamente hay 15 registros de color para los sprites); los valores para el rojo, el verde y el azul deben ser números entre el 0 y el 31.

Nota

Los 15 registros de color para los sprites son independientes de los 16 registros de color empleados para los tiles. Es decir que en total hay 31 registros de color (16 + 15). Los 16 registros de color para los tiles se modifican con el procedimiento `setTileColor`, mientras que los 15 registros de color para los sprites se modifican con el procedimiento `setSpriteColor`.

Haz click en el botón ⏭ y observa cómo la bola cambia de color de blanco a amarillo. Después hay un ciclo que vuelve a mover el sprite en la pantalla. Haz

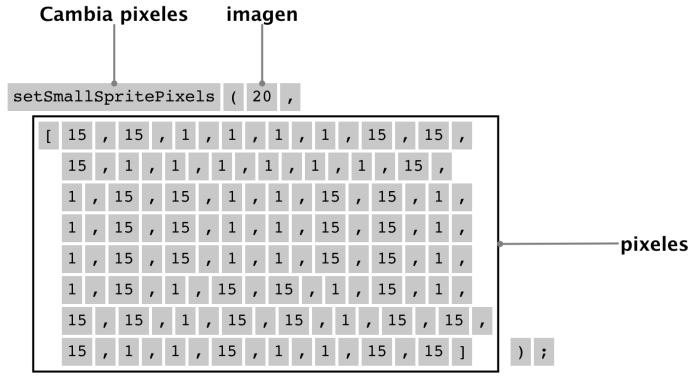
un doble click en la línea que viene después del ciclo (la que dice `pause(.5);`) para ejecutar rápidamente esa parte del programa.

Lo que sigue en el programa:

```
setSmallSpritePixels(20,
  [15, 15,  1,  1,  1,  1, 15, 15,
   15,  1,  1,  1,  1,  1,  1, 15,
    1, 15, 15,  1,  1, 15, 15,  1,
    1, 15, 15,  1,  1, 15, 15,  1,
    1, 15, 15,  1,  1, 15, 15,  1,
    1, 15,  1, 15, 15,  1, 15,  1,
   15, 15,  1, 15, 15,  1, 15, 15,
   15,  1,  1, 15,  1,  1, 15, 15]);
setSpriteColor(1, 31, 0, 0);
for (var i = 39; i >= -8; i--) {
  putSpriteAt(0, i * 2, i);
  pause(.02);
}
for (var i = -8; i < 257; i++) {
  putSpriteAt(0, i, 20);
  pause(.02);
}
pause(.5);
```

Empieza por llamar al procedimiento `setSmallSpritePixels` para modificar la imagen pequeña (de 8 por 8) número 20 de los sprites. Emplea el botón ⏭ para ejecutar ese llamado y observa cómo la imagen de la bola se convierte en un robot de color negro.

Este es el significado del llamado al procedimiento `setSmallSpritePixels`:



El procedimiento `setSmallSpritePixels` espera dos argumentos. El primero es el número de la imagen de sprite que se desea modificar, debe ser un número entre 0 y 127. El segundo es un arreglo con 64 números que indican el registro de color a emplear para cada uno de los pixeles; cada uno de estos números debe ser un valor entre 0 y 15; los valores del 0 a 14 se emplean para seleccionar uno de los 15 registros de color, un valor de 15 indica que ese pixel debe ser transparente (no se dibuja) . Los primeros ocho números son los colores para el primer renglón de pixeles, los ocho siguientes para el segundo renglón y así sucesivamente hasta llegar al octavo renglón.

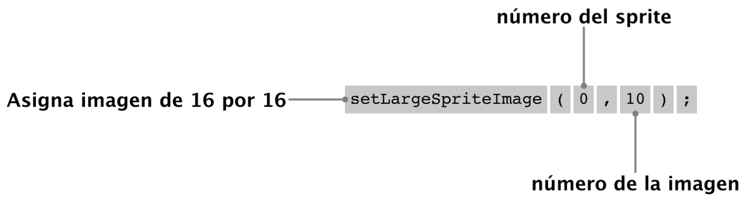
La línea siguiente llama al procedimiento `setSpriteColor` para cambiar el contenido del registro de color número 1 de negro a rojo. Haz click en ⏭ y observa cómo cambia el color del robot.

Después vienen dos ciclos. El primero mueve el sprite hacia la esquina superior izquierda de la pantalla, pero ahora en vez de detenerse cuando la variable `i` llega a cero se sigue hasta que llega a -8. Los sprites se pueden colocar, tanto horizontalmente como verticalmente, hasta el pixel -16 para que sea posible que un sprite salga de la pantalla por el lado superior o el lado izquierdo. El segundo ciclo mueve el sprite horizontalmente desde la posición -8 hasta la 256 (menor a 257) manteniéndolo siempre en 20 verticalmente. Al llegar a la posición horizontal 256, el sprite sale de la pantalla por el lado derecho. También es posible mover un sprite hasta la posición vertical 192 para que salga de la pantalla por la parte inferior. Haz un doble click en la línea que viene después de los dos ciclos (la que dice "`pause(.5);`") para que veas cómo el sprite puede entrar y salir de la pantalla.

Lo que sigue en el programa es:

```
putSpriteAt(0, 124, 92);
setLargeSpriteImage(0, 10);
setLargeSpritePixels(10,
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
);
pause(.5);
```

Primero emplea el procedimiento `putSpriteAt` para colocar el sprite por el centro de la pantalla. Ejecuta esta línea con el botón ⏭ y observa cómo aparece el robot directamente cerca del centro de la pantalla. La línea siguiente emplea el procedimiento `setLargeSpriteImage` para asignarle la imagen grande de sprites número 10 al sprite cero (recuerda que hay 128 imágenes pequeñas de 8 por 8 pixeles y 128 imágenes grandes de 16 por 16 pixeles disponibles para los sprites). El significado de este llamado es el siguiente:



Haz click en el botón ⏭ para ejecutar el llamado a `setLargeSpriteImage`. El sprite desaparece de la pantalla. En realidad el sprite sigue ahí pero las 128 imágenes grandes de sprites, al iniciar la ejecución de un programa, están definidas como completamente transparentes. Para que se vea algo es necesario redefinir la imagen empleando los registros de color del 0 al 14. Para eso se

emplea el procedimiento `setLargeSpritePixels`, cuyo significado es el siguiente:

Cambia pixeles — **imagen** — **pixeles**

```
setLargeSpritePixels ( 10 ,
  [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
    0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
    0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
    0 , 0 , 0 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 0 , 0 , 0 ,
    0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 ,
    0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 ,
    0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 ,
    0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 ,
    0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 ,
    0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 ,
    0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 ,
    0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 ,
    0 , 0 , 0 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 0 , 0 , 0 ,
    0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
    0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
    0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ]
) ;
```

El procedimiento `setLargeSpritePixels` espera dos argumentos. El primero es el número de la imagen de sprite que se desea modificar, debe ser un número entre 0 y 127. El segundo es un arreglo con 256 números que indican el registro de color a emplear para cada uno de los pixeles; cada uno de estos números debe ser un valor entre 0 y 15; los valores del 0 a 14 se emplean para seleccionar uno de los 15 registros de color, un valor de 15 indica que ese pixel debe ser transparente (no se dibuja) . Los primeros dieciseis números son los colores para el primer renglón de pixeles, los dieciseis siguientes para el segundo renglón y así sucesivamente hasta llegar al último renglón.

Emplea el botón ⏭ para ejecutar el llamado a `setLargeSpritePixels` y observa cómo aparece ahora el sprite como un cuadrado relleno de amarillo (el color almacenado en el registro 0) con un cuadrado rojo adentro (el color almacenado en el registro 1).

El resto del programa:

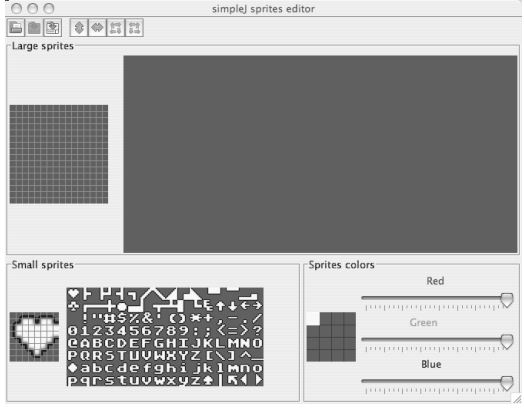
```
for (var i = 124; i >= -16; i--) {
  putSpriteAt(0, i, 92);
  pause(.02);
}
pause(.5);

for (var i = -16; i < 257; i++) {
  putSpriteAt(0, i, 92);
  pause(.02);
}
pause(1);
```

simplemente emplea dos ciclos para mover el sprite. Primero hacia la izquierda hasta que se sale de la pantalla; después hacia la derecha hasta que se sale de la pantalla por el otro lado.

El simpleJ sprites editor

Es un poco tedioso andar definiendo las imágenes de los sprites con arreglos de 64 ó 256 números. Para facilitar la creación de estas imágenes existe el simpleJ sprites editor que te permite crear las imágenes empleando el mouse.

Ejecuta el sprites editor; aparece una ventana como esta:



La ventana está dividida en tres áreas:

Large sprites	Aquí es donde puedes editar las 128 imágenes de 16 por 16 pixeles.

Small sprites	Aquí es donde puedes editar las 128 imágenes de 8 por 8 pixeles.
Sprites colors	Sirve para seleccionar el color que quieres almacenar en cada uno de los 15 registros de color.

Arriba de estas tres áreas hay unos botones que corresponden a las acciones que vas a usar más frecuentemente. También puedes tener acceso a estas acciones desde los menúes que se encuentran arriba de esos botones.

Nota

En el CD que viene con este libro hay un video que se llama "`sprites editor.avi`". Si ves ese video va a ser más sencillo que entiendas como emplear el simpleJ sprites editor.

El área *Small sprites* tiene dos partes. A la derecha tiene las 128 imágenes de 8 por 8 pixeles y a la izquierda te muestra en grande la imagen seleccionada para que puedas editar sus pixeles con el mouse. Haz click sobre cualquiera de esas imágenes y observa cómo automáticamente aparece en grande a la izquierda para que la puedas editar.

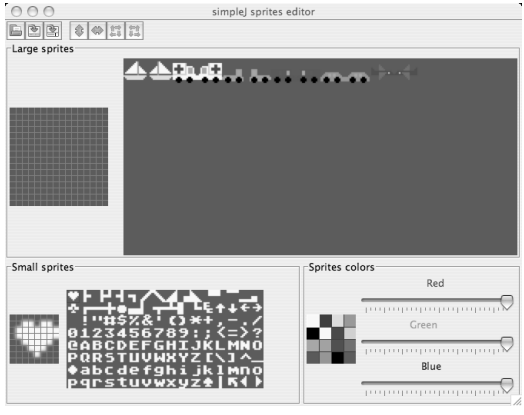
El área *Large sprites* funciona de la misma manera. Al iniciar el sprites editor no hay ninguna imagen definida de 16 por 16 pixeles. Haz click dentro del rectángulo negro que está a la derecha del área *Large sprites* y observa cómo aparece ahí un cuadrado rojo que marca el borde de esa imagen.

Fíjate también cómo al seleccionar una de las imágenes grandes, desapareció el cuadrado rojo alrededor de la imagen de 8 por 8 pixeles que tenías seleccionada en el área *Small sprites*. Selecciona ahora una de las imágenes del área *Small sprites* y observa cómo desaparece el cuadrado rojo que indicaba cuál de las imágenes del área *Large sprites* estaba seleccionada. En un momento dado sólo puede estar seleccionada una sola imagen, ya sea grande (de 16 por 16) o pequeña (de 8 por 8). Los botones  ,  ,  y  operan sobre la imagen que tengas seleccionada en ese momento. Aquí también puedes emplear el botón derecho del mouse, como en el tiles editor, para copiar una imagen.

El área *Sprites colors* funciona exactamente igual que el área *Color registers* en el tiles editor. La única diferencia es que lo que dibujes con el último color, el que está en la esquina inferior derecha dentro del cuadrado de colores, al

dibujar el sprite en la pantalla va a quedar como transparente. Dentro del sprites editor puedes modificar ese color para poder probar cómo se ven los sprites sobre fondos de diferentes colores, pero recuerda que lo que dibujes con ese color va a quedar como transparente al emplear esas imágenes para los sprites dentro de un juego.

El sprites editor permite almacenar los registros de color y las imágenes de los sprites en un archivo que después se puede leer desde un programa para emplearlos dentro de un juego. Haz click en el botón para abrir un archivo con un ejemplo. Como has estado haciendo cambios en el tiles editor primero te va a aparecer una ventana preguntando si quieres descartar esos cambios. Haz click sobre el botón **Yes** para confirmar que sí deseas descartar los cambios. Ahora te aparece una nueva ventana donde puedes ver las carpetas con los proyectos de simpleJ, hay una carpeta por cada proyecto. Entra a la carpeta `Nivel_10` y ahí selecciona abrir el archivo `sprites.smap`. Aparece esto en el sprites editor:



Ahí puedes ver las imágenes empleadas para representar diferentes vehículos como sprites.

Ya puedes cerrar el sprites editor. Ahora vamos a ver cómo leer el contenido de un archivo creado con el sprites editor para poder emplear esas imágenes en un programa.

Cómo leer un archivo creado con el simpleJ sprites editor

Haz click en  y abre el programa `spritesEditor.sj`. Ahora haz click en  para ejecutarlo. Aparecen en la pantalla las doce imágenes de los vehículos que están definidas en el archivo `sprites.smap`.

El programa es bastante corto:

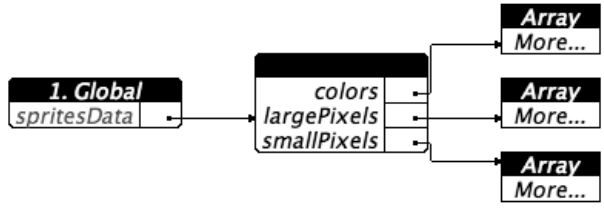
```
/* Leer definiciones de sprites creadas con
   el sprites editor */
var spritesData = readSpritesFile("sprites.smap");

/* Poner colores en el mapa de colores */
for (var i = 0; i < 15; i++)
  setSpriteColor(i, spritesData.colors[i].red,
                    spritesData.colors[i].green,
                    spritesData.colors[i].blue);

/* Grabar nuevas definiciones de sprites de 16 por 16 */
for (var i = 0; i < 128; i++)
  setLargeSpritePixels(i, spritesData.largePixels[i]);

/* Grabar nuevas definiciones de sprites de 8 por 8 */
for (var i = 0; i < 128; i++)
  setSmallSpritePixels(i, spritesData.smallPixels[i]);

/* Muestra las imagenes */
for (var i = 0; i < 12; i++) {
  setLargeSpriteImage(i, i);
  putSpriteAt(i, i * 16, i * 8);
}
```

Ahora haz click en para empezar a ejecutarlo paso por paso.

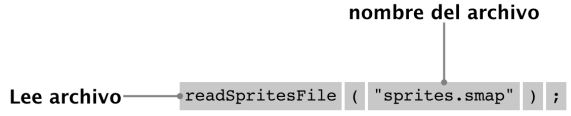
Las primeras líneas son las que se emplean para leer los datos almacenados en el archivo `sprites.smap`:

```
/* Leer definiciones de sprites creadas con
   el sprites editor */
var spritesData = readSpritesFile("sprites.smap");
```

Haz click una vez más en ⏭ y la vista de la memoria queda así:



Este es el significado del procedimiento `readSpritesFile`:



En realidad `readSpritesFile` es una función porque devuelve un resultado (que estamos almacenando dentro de la variable `spritesData`). Este resultado es un ambiente que contiene tres variables: `colors`, `largePixels` y `smallPixels`. Cada una de estas tres variables contiene una referencia a un arreglo.

Nota

En la vista de memoria los arreglos aparecen colapsados porque son bastante grandes y le tomaría mucho tiempo al devkit dibujarlos. Puedes expander cada uno de ellos haciendo un doble click sobre su rectángulo correspondiente, pero no te lo recomendamos a menos que tengas una computadora muy rápida.

La variable `colors` tiene una referencia a un arreglo con 16 elementos, uno para cada uno de los 15 registros de color más un color que se emplea en el sprites editor para representar los pixeles transparentes (ese último color normalmente no se emplea dentro de un programa pero de todas maneras lo devuelve la función `readSpritesFile` porque es parte del archivo). Cada uno de estos 16 elementos contiene una referencia a un ambiente; cada uno de estos ambientes contiene tres variables: `red`, `green`, y `blue`; estas variables contienen un número entre 0 y 31 para indicar el valor de los componentes rojo, verde y azul para ese registro de color.

La parte que sigue del programa es la que recorre los primeros 15 elementos de ese arreglo para asignarle los colores leídos a los registros de color:

```
/* Poner colores en el mapa de colores */
for (var i = 0; i < 15; i++)
  setSpriteColor(i, spritesData.colors[i].red,
                    spritesData.colors[i].green,
                    spritesData.colors[i].blue);
```

Es simplemente un ciclo en donde la variable `i` va tomando sucesivamente los valores desde el 0 hasta el 14. En cada iteración empleamos la expresión `spritesData.colors[i]` para accesar el ambiente con los datos que corresponden al registro de color con ese número, seguido de `.red`, `.green` y `.blue` para accesar sus componentes rojo, verde y azul y emplearlos como argumentos para llamar al procedimiento `setSpriteColor` para que modifique el contenido del registro de color correspondiente.

Puedes emplear el botón ⏭ para ejecutar este ciclo paso por paso, pero no se va a ver nada que cambie en la pantalla porque todavía ninguno de los sprites está ahí.

La variable `largePixels` (dentro del ambiente devuelto por la función `readSpritesFile`) contiene una referencia a un arreglo con 128 elementos, un elemento para cada una de las imágenes de 16 por 16 pixeles. Cada uno de esos elementos es a su vez una referencia a un arreglo con 256 números que indican el color a emplear para cada uno de los 16 por 16 pixeles que tiene cada imagen grande para los sprites.

La parte que sigue del programa es la que recorre este arreglo de 128 elementos para asignar los pixeles correspondientes a las imágenes grandes de los sprites:

```
/* Grabar nuevas definiciones de sprites de 16 por 16 */
for (var i = 0; i < 128; i++)
  setLargeSpritePixels(i, spritesData.largePixels[i]);
```

Aquí también tenemos un ciclo que recorre los 128 arreglos que hay dentro del arreglo `largePixels` para redefinir las imágenes grandes (16 por 16 pixeles) de los sprites empleando el procedimiento `setLargeSpritePixels`. Nuestro archivo con las imágenes para los sprites únicamente contiene doce de estas imágenes, por lo tanto bastaría con emplear un ciclo que vaya desde 0 hasta 11, pero es mejor dejar así el programa por si después se agregan nuevas imágenes dentro del archivo. Puedes ejecutar este procedimiento paso por paso con el botón ⏭, pero es un poco tedioso ya que no se ve nada que cambie en el área de Video. Te recomendamos que mejor hagas un doble click sobre la línea que dice "`for (var i = 0; i < 128; i++)`" (no la que

está al principio de este ciclo, la que viene un poco después) para ejecutar rápidamente todas las instrucciones hasta llegar a esa línea.

La variable `smallPixels` (dentro del ambiente devuelto por la función `readSpritesFile`) contiene una referencia a un arreglo con 128 elementos, un elemento para cada una de las imágenes de 8 por 8 pixeles. Cada uno de esos elementos, es a su vez una referencia a un arreglo con 64 números que indican el color a emplear para cada uno de los 8 por 8 pixeles que tiene cada imagen pequeña para los sprites.

La parte que sigue del programa es la que recorre este arreglo de 128 elementos para asignar los pixeles correspondientes a las imágenes grandes de los sprites:

```
/* Grabar nuevas definiciones de sprites de 8 por 8 */
for (var i = 0; i < 128; i++)
  setSmallSpritePixels(i, spritesData.smallPixels[i]);
```

Aquí también tenemos un ciclo que recorre los 128 arreglos que hay dentro del arreglo `smallPixels` para redefinir las imágenes pequeñas (8 por 8 pixeles) de los sprites empleando el procedimiento `setSmallSpritePixels`. Nuestro archivo con las imágenes para los sprites no contiene ninguna de estas imágenes, por lo tanto no es necesario este ciclo, pero es mejor dejar así el programa por si después se agregan nuevas imágenes dentro del archivo. Puedes ejecutar este procedimiento paso por paso con el botón ⏭, pero es un poco tedioso ya que no se ve nada que cambie en el área de Video. Te recomendamos que mejor hagas un doble click sobre la línea que dice "`for (var i = 0; i < 12; i++)`" para ejecutar rápidamente todas las instrucciones hasta llegar a esa línea.

El final del programa:

```
/* Muestra las imagenes */
for (var i = 0; i < 12; i++) {
  setLargeSpriteImage(i, i);
  putSpriteAt(i, i * 16, i * 8);
}
```

Emplea un ciclo para asignarle a cada uno de los doce primeros sprites (del 0 hasta el 11) cada una de las doce imágenes grandes (de 16 por 16) que tenemos en el archivo `sprites.smap`, y colocar los sprites en la pantalla. Emplea el botón ⏭ para ejecutarlo paso por paso y observa cómo van apareciendo los

sprites en la pantalla, cada uno 16 pixeles más a la derecha y 8 pixeles más abajo que el anterior.

Animación con el procedimiento `vbi`

Hasta ahora hemos estados haciendo animaciones empleando un ciclo con una pausa adentro para controlar la velocidad con la cual se mueven los objetos en la pantalla. Funciona, pero no es la mejor manera de hacer animaciones. En las pantallas de televisión y de computadora la imágen se redibuja varias veces por segundo. En el caso de las televisiones la imagen se redibuja 60 veces por segundo. En el simpleJ devkit y la simpleJ virtual console la imagen se redibuja 25 veces por segundo. Con los sprites existe la posibilidad de que el programa intente cambiar la posición o la imagen de un sprite mientras que este está siendo dibujado en la pantalla y eso puede causar errores como que, por una fracción de segundo, se dibuje basura en la pantalla o que el sprite aparezca en otra parte de la pantalla. Esos errores no son muy graves, el juego sigue funcionando, pero son molestos para el jugador.

Para evitar estos problemas hay que modificar lo que aparece en la pantalla únicamente cuando se terminó de dibujar y antes de que se vuelve a dibujar. En las consolas de videojuegos los circuitos que se encargan de dibujar la imagen le mandan una señal al procesador cada vez que terminan de dibujarla para avisarle que en ese momento puede hacer cambios antes de que se dibuje otra vez la pantalla. A esta señal se le conoce como *Vertical Blank Interrupt* (*VBI*). En simpleJ también existe la señal VBI, para ejecutar algo al recibir esa señal se emplea un procedimiento que se llama `vbi`. Si en tu programa defines un procedimiento llamado `vbi`, que no espera ningún argumento, entonces simpleJ se encarga de ejecutarlo cada vez que termina de dibujar la pantalla y antes de volverla a dibujar.

Veamos un ejemplo. Haz click en  y abre el programa `vbi.sj`:

```
showAt("Demo vbi()", 12, 12);
pause(.5);

setSmallSpriteImage(0, 20);
var x = -8;

vbi() {
  putSpriteAt(0, x, 12 * 8);
  x++;
  if (x == 256)
```

```
    x = -8;
}

while (true) {
  setBackground(31, 0, 0);
  pause(.5);
  setBackground(0, 0, 31);
  pause(.5);
}
```

Haz click en ▶ para ejecutar el programa. Despliega el mensaje "Demo vbi()"; una bola se mueve por la pantalla de la izquierda hacia la derecha, al salir por el lado derecho vuelve a entrar por el lado derecho; mientras que la bola se mueve el color del fondo anda cambiando entre azul a rojo.

Haz click en ■ para detener la ejecución del programa. Ahora haz click en ▶| para empezar a ejecutarlo paso por paso.

Las primeras lineas:

```
showAt("Demo vbi()", 12, 12);
pause(.5);
```

simplemente despliegan en la pantalla el mensaje "`Demo vbi()`" y hacen una pausa de medio segundo.

La siguiente parte del programa:

```
setSmallSpriteImage(0, 20);
var x = -8;
```

Le asignan al sprite cero la imagen pequeña número `20` (una bola) y declaran una variable `x` con el valor inicial `-8`. La variable `x` se emplea para almacenar la posición horizontal del sprite en la pantalla, con un `-8` el sprite va a empezar fuera de la pantalla (por el lado izquierdo).

Después viene la definición del procedimiento `vbi`:

```
vbi() {
  putSpriteAt(0, x, 12 * 8);
  x++;
  if (x == 256)
    x = -8;
}
```

Observa como al hacer click en para ejecutar la definición de este procedimiento en la vista de la memoria aparece en el ambiente *Global* una variable `vbi` y la bola se mueve por la pantalla aunque nuestro programa esté detenido. Al existir en el ambiente *Global* un procedimiento llamado `vbi`, que no recibe ningún argumento, automáticamente se ejecuta este procedimiento cada vez que se termina de redibujar la pantalla, es decir 25 veces por segundo, independientemente de lo que esté haciendo el resto del programa.

El procedimiento `vbi` de este ejemplo es bastante sencillo. Coloca el sprite cero en la posición horizontal indicada por la variable `x` y la posición vertical `96` (12 veces 8); incrementa en uno el valor de `x`; si el valor de `x` ya llegó a `256` (el sprite salió por el lado derecho de la pantalla) entonces le asigna un `-8` (para devolver el sprite al lado izquierdo).

La parte que sigue del programa:

```
while (true) {
  setBackground(31, 0, 0);
  pause(.5);
  setBackground(0, 0, 31);
  pause(.5);
}
```

Es un ciclo infinito que se encarga de ir cambiando el color del fondo entre rojo y azul. Emplea el botón para irlo ejecutando paso por paso y observa cómo se ejecuta independientemente del movimiento de la bola, que es controlado por el procedimiento `vbi`. Fíjate también, en la vista de la memoria, cómo el valor de la variable `x` está cambiando porque está siendo modificado 25 veces por segundo (aunque sólo se actualiza en la vista de la memoria cada vez que haces un click en el botón).

Recuerda hacer click en el botón cuando termines de probar este ejemplo para detener la ejecución del programa (y del procedimiento `vbi`).

Una animación con sprites

Usa el botón para abrir el programa `main.sj`. Es un programa relativamente corto:

```
final VELERO_DERECHA = 0;
final VELERO_IZQUIERDA = 1;
```

```
final AMBULANCIA_DERECHA = 2;
final AMBULANCIA_IZQUIERDA = 3;
final CAMION_DERECHA = 4;
final CAMION_IZQUIERDA = 5;
final MILITAR_DERECHA = 6;
final MILITAR_IZQUIERDA = 7;
final COCHE_DERECHA = 8;
final COCHE_IZQUIERDA = 9;
final AVION_DERECHA = 10;
final AVION_IZQUIERDA = 11;

/* Leer definiciones de tiles creadas con
   el tiles editor */
var tilesData = readTilesFile("tilesEditorDemo.tmap");

/* Poner colores en el mapa de colores */
for (var i = 0; i < 16; i++)
  setTileColor(i, tilesData.colors[i].red,
                  tilesData.colors[i].green,
                  tilesData.colors[i].blue);

/* Grabar nuevas definiciones de tiles */
for (var i = 0; i < 256; i++)
  setTilePixels(i, tilesData.pixels[i]);

/* Dibujar la pantalla */
for (var r = 0; r < 24; r++)
  for (var c = 0; c < 32; c++)
    putAt(tilesData.rows[r][c], c, r);

/* Leer definiciones de sprites creadas con
   el sprites editor */
var spritesData = readSpritesFile("sprites.smap");

/* Poner colores en el mapa de colores */
for (var i = 0; i < 15; i++)
  setSpriteColor(i, spritesData.colors[i].red,
                    spritesData.colors[i].green,
                    spritesData.colors[i].blue);

/* Grabar nuevas definiciones de sprites */
for (var i = 0; i < 12; i++)
  setLargeSpritePixels(i, spritesData.largePixels[i]);
```

```
/* Datos para los sprites de los vehiculos */
var vehiculos = [
  {pixeles: VELERO_DERECHA, x: 20, vx: 1, y: 175},
  {pixeles: VELERO_DERECHA, x: 100, vx: 1, y: 172},
  {pixeles: VELERO_IZQUIERDA, x: 250, vx: -1, y: 165},
  {pixeles: AMBULANCIA_DERECHA, x: 40, vx: 2, y: 150},
  {pixeles: CAMION_DERECHA, x: 100, vx: 2, y: 150},
  {pixeles: CAMION_DERECHA, x: 170, vx: 2, y: 150},
  {pixeles: COCHE_DERECHA, x: 10, vx: 3, y: 145},
  {pixeles: COCHE_DERECHA, x: 100, vx: 3, y: 145},
  {pixeles: COCHE_DERECHA, x: 200, vx: 3, y: 145},
  {pixeles: COCHE_IZQUIERDA, x: 200, vx: -3, y: 137},
  {pixeles: COCHE_IZQUIERDA, x: 150, vx: -3, y: 137},
  {pixeles: COCHE_IZQUIERDA, x: 100, vx: -3, y: 137},
  {pixeles: COCHE_IZQUIERDA, x: 50, vx: -3, y: 137},
  {pixeles: CAMION_IZQUIERDA, x: 10, vx: -2, y:132},
  {pixeles: MILITAR_IZQUIERDA, x: 100, vx: -2, y:132},
  {pixeles: MILITAR_IZQUIERDA, x: 120, vx: -2, y:132},
  {pixeles: MILITAR_IZQUIERDA, x: 140, vx: -2, y:132},
  {pixeles: MILITAR_IZQUIERDA, x: 160, vx: -2, y:132},
  {pixeles: MILITAR_IZQUIERDA, x: 180, vx: -2, y:132},
  {pixeles: AVION_IZQUIERDA, x: 100, vx: -4, y: 70},
  {pixeles: AVION_DERECHA, x: 0, vx: 4, y: 30}
];

/* Seleccionar imagenes para los sprites */
for (var i = 0; i < length(vehiculos); i++)
  setLargeSpriteImage(i, vehiculos[i].pixeles);

vbi() {
  for (var i = 0; i < length(vehiculos); i++) {
    putSpriteAt(i, vehiculos[i].x, vehiculos[i].y);
    vehiculos[i].x = vehiculos[i].x + vehiculos[i].vx;
    if (vehiculos[i].vx > 0 && vehiculos[i].x >= 256)
      vehiculos[i].x = -16;
    if (vehiculos[i].vx < 0 && vehiculos[i].x <= -16)
      vehiculos[i].x = 256;
  }
}
```

Las primeras líneas:

```
final VELERO_DERECHA = 0;
final VELERO_IZQUIERDA = 1;
```

```
final AMBULANCIA_DERECHA = 2;
final AMBULANCIA_IZQUIERDA = 3;
final CAMION_DERECHA = 4;
final CAMION_IZQUIERDA = 5;
final MILITAR_DERECHA = 6;
final MILITAR_IZQUIERDA = 7;
final COCHE_DERECHA = 8;
final COCHE_IZQUIERDA = 9;
final AVION_DERECHA = 10;
final AVION_IZQUIERDA = 11;
```

Definen unas constantes para poder referirse a cada una de las imágenes de los vehículos por un nombre, en vez de un número.

La parte que sigue:

```
/* Leer definiciones de tiles creadas con
   el tiles editor */
var tilesData = readTilesFile("tilesEditorDemo.tmap");

/* Poner colores en el mapa de colores */
for (var i = 0; i < 16; i++)
  setTileColor(i, tilesData.colors[i].red,
                  tilesData.colors[i].green,
                  tilesData.colors[i].blue);

/* Grabar nuevas definiciones de tiles */
for (var i = 0; i < 256; i++)
  setTilePixels(i, tilesData.pixels[i]);

/* Dibujar la pantalla */
for (var r = 0; r < 24; r++)
  for (var c = 0; c < 32; c++)
    putAt(tilesData.rows[r][c], c, r);
```

Es para leer las definiciones y colores de los tiles dibujados con el simpleJ tiles editor para dibujar el paisaje de fondo. Lo hace exactamente de la misma manera que empleamos en el nivel anterior.

Después vienen estas líneas:

```
/* Leer definiciones de sprites creadas con
   el sprites editor */
var spritesData = readSpritesFile("sprites.smap");
```

```
/* Poner colores en el mapa de colores */
for (var i = 0; i < 15; i++)
  setSpriteColor(i, spritesData.colors[i].red,
                    spritesData.colors[i].green,
                    spritesData.colors[i].blue);

/* Grabar nuevas definiciones de sprites */
for (var i = 0; i < 12; i++)
  setLargeSpritePixels(i, spritesData.largePixels[i]);
```

Para leer las imágenes y colores para los sprites, tal como lo vimos en el ejemplo de cómo leer los archivos creados con el simpleJ sprites editor.

La parte que sigue ya es un poco más interesante:

```
/* Datos para los sprites de los vehiculos */
var vehiculos = [
  {pixeles: VELERO_DERECHA, x: 20, vx: 1, y: 175},
  {pixeles: VELERO_DERECHA, x: 100, vx: 1, y: 172},
  {pixeles: VELERO_IZQUIERDA, x: 250, vx: -1, y: 165},
  {pixeles: AMBULANCIA_DERECHA, x: 40, vx: 2, y: 150},
  {pixeles: CAMION_DERECHA, x: 100, vx: 2, y: 150},
  {pixeles: CAMION_DERECHA, x: 170, vx: 2, y: 150},
  {pixeles: COCHE_DERECHA, x: 10, vx: 3, y: 145},
  {pixeles: COCHE_DERECHA, x: 100, vx: 3, y: 145},
  {pixeles: COCHE_DERECHA, x: 200, vx: 3, y: 145},
  {pixeles: COCHE_IZQUIERDA, x: 200, vx: -3, y: 137},
  {pixeles: COCHE_IZQUIERDA, x: 150, vx: -3, y: 137},
  {pixeles: COCHE_IZQUIERDA, x: 100, vx: -3, y: 137},
  {pixeles: COCHE_IZQUIERDA, x: 50, vx: -3, y: 137},
  {pixeles: CAMION_IZQUIERDA, x: 10, vx: -2, y:132},
  {pixeles: MILITAR_IZQUIERDA, x: 100, vx: -2, y:132},
  {pixeles: MILITAR_IZQUIERDA, x: 120, vx: -2, y:132},
  {pixeles: MILITAR_IZQUIERDA, x: 140, vx: -2, y:132},
  {pixeles: MILITAR_IZQUIERDA, x: 160, vx: -2, y:132},
  {pixeles: MILITAR_IZQUIERDA, x: 180, vx: -2, y:132},
  {pixeles: AVION_IZQUIERDA, x: 100, vx: -4, y: 70},
  {pixeles: AVION_DERECHA, x: 0, vx: 4, y: 30}
];
```

Crea un arreglo de ambientes y almacena una referencia a este arreglo dentro de la variable `vehiculos`. Cada uno de estos ambientes almacena la informa-

ción para uno de los vehículos que vamos a animar. Para cada vehículo tiene estas variables:

1. `pixeles`: el número de la imagen a emplear para ese vehículo.

2. `x`: la posición inicial horizontal para ese vehículo en la pantalla.

3. `vx`: la velocidad del vehículo al moverse por la pantalla. Un número positivo indica que se mueve hacia la derecha, mientras que un número negativo indica que se mueve hacia la izquierda. Entre mayor sea el número, mayor será la velocidad del vehículo.

4. `y`: la posición vertical del vehículo en la pantalla.

La parte siguiente del programa:

```
/* Seleccionar imagenes para los sprites */
for (var i = 0; i < length(vehiculos); i++)
  setLargeSpriteImage(i, vehiculos[i].pixeles);
```

Simplemente recorre el arreglo `vehiculos` y, para cada elemento, le asigna al sprite correspondiente (el sprite con el mismo número que el subíndice del elemento dentro del arreglo) la imagen indicada para ese vehículo dentro de su variable `pixeles` (`vehiculos[i].pixeles`).

Lo último del programa:

```
vbi() {
  for (var i = 0; i < length(vehiculos); i++) {
    putSpriteAt(i, vehiculos[i].x, vehiculos[i].y);
    vehiculos[i].x = vehiculos[i].x + vehiculos[i].vx;
    if (vehiculos[i].vx > 0 && vehiculos[i].x >= 256)
      vehiculos[i].x = -16;
    if (vehiculos[i].vx < 0 && vehiculos[i].x <= -16)
      vehiculos[i].x = 256;
  }
}
```

Es la definición del procedimiento `vbi` que mueve los sprites después de cada vez en que se dibuja la pantalla, para que aparezca en su nueva posición la siguiente vez que se dibuja. Recorre el arreglo `vehiculos` y para cada elemento:

1. Coloca el sprite del vehículo en su nueva posición en la pantalla:

```
putSpriteAt(i, vehiculos[i].x, vehiculos[i].y);
```

2. Calcula su nueva posición sumándole su velocidad a su posición horizontal:

```
vehiculos[i].x = vehiculos[i].x + vehiculos[i].vx;
```

3. Si su velocidad es positiva (se mueve hacia la derecha) y ya salió de la pantalla por el lado derecho entonces lo devuelve hasta justo antes del borde izquierdo:

```
if (vehiculos[i].vx > 0 && vehiculos[i].x >= 256)
  vehiculos[i].x = -16;
```

4. Si su velocidad es negativa (se mueve hacia la izquierda) y ya salió de la pantalla por el lado izquierdo entonces lo devuelve hasta justo antes del borde derecho:

```
if (vehiculos[i].vx < 0 && vehiculos[i].x <= -16)
  vehiculos[i].x = 256;
```

Nivel 11: Audio

Se pueden hacer sonidos más interesantes

Hasta ahora hemos estado usando el procedimiento `note` para hacer sonidos. simpleJ tiene un generador de audio con cuatro canales independientes que se pueden emplear para hacer sonidos más interesantes.

Veamos un ejemplo. En el menú de **Project** selecciona la opción **Switch to another project...** (**Project** > **Switch to another project...**) y selecciona el proyecto `Nivel_11`. Haz click en  para probar el programa. Aparece en la pantalla un mensaje que te indica que apoyes la barra de espacio para hacer un sonido; cada vez que presionas sobre la barra de espacio se escucha un sonido, similar a los sonidos empleados en las primeras consolas que hubo de videojuegos.

Manejo básico de los canales de audio

Cada uno de los cuatro canales de audio tiene varios parámetros que se pueden ajustar independientemente para cada canal. Estos parámetros son: *frequency*, *attack*, *decay*, *sustain*, *release* y *volume*. Empezaremos viendo cómo se emplean *frequency*, *attack* y *decay*.

Haz click en  y abre el programa `frequencyAttackDecay.sj`:

```
frecuencias() {
  setSoundFrequency(0, 1000);
  soundOn(0);
  pause(1);

  setSoundFrequency(0, 5000);
  soundOn(0);
  pause(1);

  setSoundFrequency(0, 10000);
  soundOn(0);
  pause(1);

  setSoundFrequency(0, 40000);
  soundOn(0);
  pause(1);
```

```
}

frecuencias();

setSoundDecay(0, 600);
frecuencias();

setSoundAttack(0, 300);
frecuencias();

setSoundDecay(0, 10);
frecuencias();
```

Haz click en ▶ para ejecutar el programa. Se escuchan varias notas: algunas agudas, otras graves. Esas mismas notas se escuchan varias veces, pero cada vez con un *timbre* diferente. En música, el timbre de un instrumento es lo que le da su sonido característico. No suena igual la misma nota en un piano, que en una guitarra o en una trompeta.

Ahora haz click en ⏭ para ejecutar el programa paso por paso y que puedas ir siguiendo la descripción de lo que hace cada parte del programa.

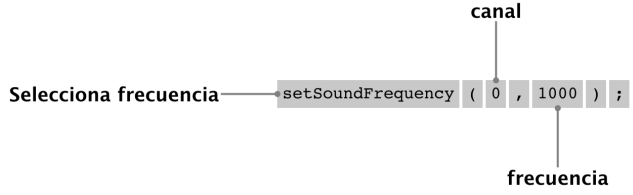
Al principio del programa:

```
frecuencias() {
  setSoundFrequency(0, 1000);
  soundOn(0);
  pause(1);

  setSoundFrequency(0, 5000);
  soundOn(0);
  pause(1);

  setSoundFrequency(0, 10000);
  soundOn(0);
  pause(1);

  setSoundFrequency(0, 40000);
  soundOn(0);
  pause(1);
}

frecuencias();
```

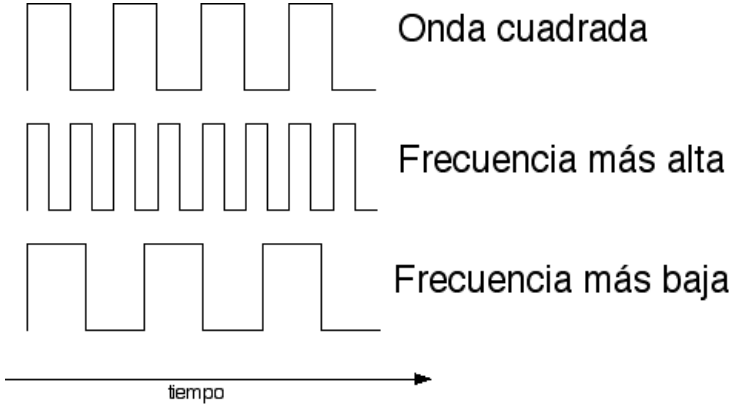
Se define un procedimiento que se llama `frequencias` y después se llama a ese procedimiento. Haz click en ⏭ para que se ejecute la definición de este procedimiento; vuelve a hacer click en ⏭ para ejecutar la línea donde se llama al procedimiento y vuelve a hacer otro click en ese mismo botón para llegar a la linea del procedimiento que dice:

```
setSoundFrequency(0, 1000);
```

Es para seleccionar la frecuencia del sonido que se desea generar. Este es su significado:



El primer argumento es para indicar el canal de sonido (un número entre 0 y 3) y el segundo argumento para seleccionar la frecuencia que se le desea asignar (un número entre 0 y 65535). Cada canal de audio en simpleJ genera sonidos mandándole a las bocinas una onda cuadrada para que vibren y produzcan un sonido. Entre más rápido vibren (mayor frecuencia), más agudo se escucha el sonido; si vibran más lentamente, entonces el sonido suena más grave.



Nota

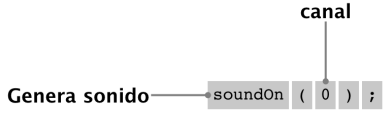
Normalmente la frecuencia de un sonido se mide en *Hertz*. Los Hertz indican el número de veces que sube y baja por segundo una onda. El

número que se pasa como segundo parámetro al procedimiento `setSoundFrequency` no está en Hertz; para pasar de Hertz al valor que hay que pasar como parámetro, necesitas multiplicar la frecuencia en Hertz por 23.77723356 y redondear el resultado. Es decir que si quieres generar un tono de 1000 Hertz entonces el valor de este parámetro debe ser 23777.

Haz click en ▶| para ejecutar esa línea. No se observa que pase nada. La línea siguiente dice:

```
soundOn(0);
```

El significado de esa línea es:



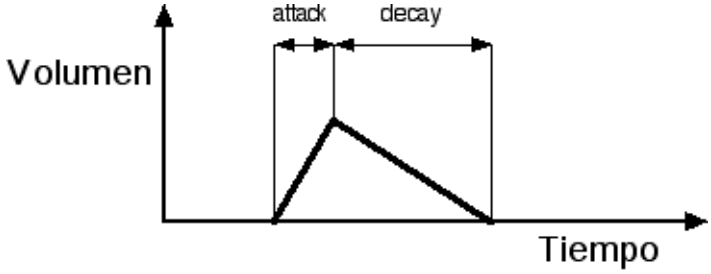
Es decir que sirve para que se genere en ese momento el sonido en el canal indicado (en este caso es el canal cero) de acuerdo a los parámetros que tiene configurados. Haz click una vez más en ▶| y se escucha el sonido.

El resto del procedimiento `frecuencias` emplea el procedimiento `setSoundFrequency` para asignarle otras frecuencias al canal cero y el procedimiento `soundOn` para generar el sonido con esa nueva frecuencia. Emplea el botón ▶| para seguir ejecutando el resto del procedimiento y escucha cómo cambia el tono con cada frecuencia.

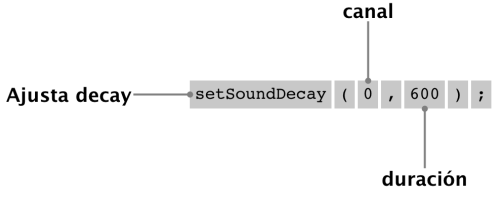
Al terminar de ejecutar el procedimiento `frecuencias` llega a la parte que dice:

```
setSoundDecay(0, 600);
frecuencias();
```

Para cambiar el timbre de un sonido hay que cambiar su *envolvente*. La envolvente de un sonido es cómo cambia su volumen en el tiempo. Al generar un sonido su volumen aumenta gradualmente hasta que llega a un máximo y, a partir de ahí, baja gradualmente hasta que ya no se escucha. El tiempo que tarda en llegar a su volumen máximo es el tiempo de *attack* y el tiempo que tarda en volver a bajar es su tiempo de *decay*.



Este es el siginificado del procedimiento `setSoundDecay`:



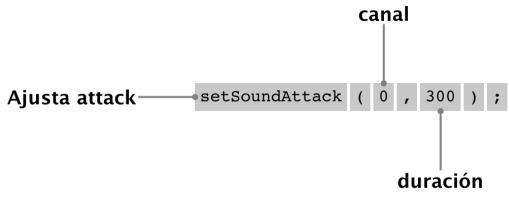
El primer argumento es el número del canal al cual se le desea modificar su decay, el segundo es el tiempo en milisegundos (un número entre 0 y 65535) que debe durar el decay. El `600` que estamos especificando es un decay que dura 0.6 segundos.

Haz click en ⏭ para ejecutar esta línea y sigue haciendo click en ese mismo botón para ejecutar ahora el procedimiento `frecuencias` y escucha cómo se vuelven a oír los mismos tonos pero ahora suenan como si fueran producidos por un instrumento diferente.

Al terminar de ejecutar el procedimiento `frecuencias` llega a la parte que dice:

```
setSoundAttack(0, 300);
frecuencias();
```

El procedimiento `setSoundAttack` es muy similar al procedimiento `setSoundDecay`, la única diferencia es que éste cambia el tiempo de attack para el canal de audio seleccionado:



Aquí también el tiempo se especifica en milisegundos y puede ser un número entre 0 y 65535. El `300` en este caso indica una duración de 0.3 segundos.

Sigue ejecutando el programa paso por paso con el botón ▶|. Escucha cómo suena ahora el sonido producido por el canal de audio número cero.

La parte final del programa vuelve a cambiar el decay para el canal cero:

```
setSoundDecay(0, 10);
frecuencias();
```

Emplea el botón ▶| para seguir ejecutando el programa y escucha cómo se oye ahora el sonido.

Se puede tener una envolvente más compleja

Los parámetros *sustain* y *release* se emplean para crear sonidos con una envolvente más compleja, y el parámetro *volume* se emplea para ajustar el volumen máximo del sonido (el volumen al que llega al terminar el attack).

Haz click en 📂 y abre el programa `sustainRelease.sj`:

```
pruebaSonido() {
  soundOn(0);
  pause(2);
  soundOff(0);
  pause(1);
}

setSoundFrequency(0, 3000);
setSoundAttack(0, 100);

setSoundSustain(0, 3);
setSoundRelease(0, 700);
pruebaSonido();
```

```
setSoundSustain(0, 1);
setSoundDecay(0, 300);
pruebaSonido();

setSoundVolume(0, 3);
pruebaSonido();
```

Haz click en ▶ para ejecutar el programa. Primero se escucha una nota que se mantiene en su volumen máximo durante un par de segundos y después disminuye el volumen. Después se vuelve a escuchar la misma nota pero esta vez baja un poco el volumen antes de mantenerse durante un par de segundos. Finalmente se vuelve a escuchar la misma nota con un volumen mucho más bajo.

Ahora haz click en ▶| para ejecutar el programa paso por paso y que puedas seguir la descripción de lo que hace cada parte del programa.

Al principio del programa

```
pruebaSonido() {
  soundOn(0);
  pause(2);
  soundOff(0);
  pause(1);
}

setSoundFrequency(0, 3000);
setSoundAttack(0, 100);
```

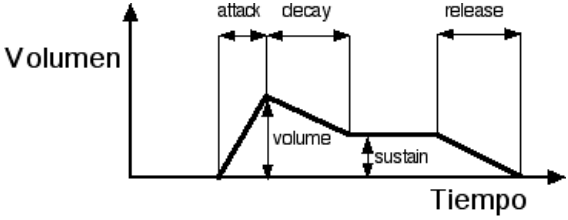
Se define un procedimiento `pruebaSonido` y se selecciona la frecuencia y tiempo de attack para el canal cero de audio. Haz click tres veces en ▶| para ejecutar esta parte del programa.

Lo que sigue es:

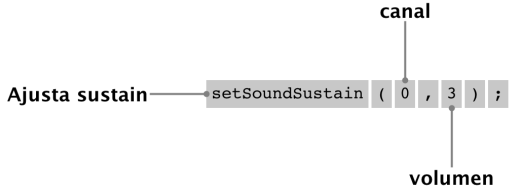
```
setSoundSustain(0, 3);
setSoundRelease(0, 700);
pruebaSonido();
```

Ya vimos anteriormente que el attack es el tiempo que tarda la envolvente en llegar a su nivel máximo y que el decay es el tiempo que tarda en volver a bajar. Con el parámetro *sustain* se puede indicar que el volumen al final del decay sea diferente de cero y por lo tanto que se siga escuchando el sonido. El sonido

se sigue escuchando hasta que se le indica al generador de audio que apague ese canal y en eso el volumen baja hasta cero en el tiempo indicado por el parámetro *release*.



El significado del procedimiento `setSoundSustain` es el siguiente:



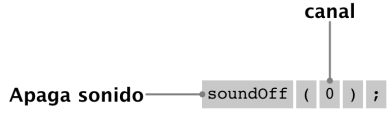
En donde el primer argumento es el canal (un número entre 0 y 3) y el segundo argumento indica el volumen de sustain como un porcentaje del volumen máximo. Este segundo argumento tiene que ser un número entre 0 y 3 con el siguiente significado:

0: 0% del volumen máximo
1: 33% del volumen máximo
2: 67% del volumen máximo
3: 100% del volumen máximo
En este caso le estamos pasando como argumento un `3`, es decir que el nivel de sustain es el mismo que el volumen máximo y por lo tanto no cambia el volumen en el decay.

El siginificado del procedimiento `setSoundRelease` es:

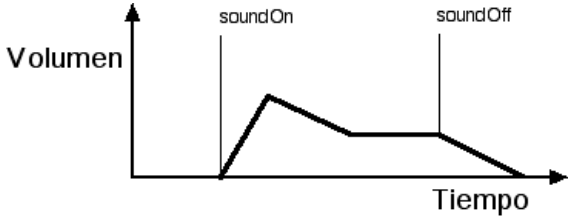
El primer argumento es el canal (un número entre 0 y 3) y el segundo argumento es el tiempo de release en milisegundos (un número entre 0 y 65535). En este caso es un `700`, lo que indica un tiempo de release de 0.7 segundos.

Haz click tres veces en el botón ▶| para ejecutar el `setSoundSustain`, el `setSoundRelease` y llamar al procedimiento `pruebaSonido`. Ahora haz click en ▶| un par de veces más para ejecutar el llamado al procedimiento `soundOn`. Después del attack, el sonido se queda en su volumen máximo. Ahora vuelve a hacer click en ▶| un par de veces más para ejecutar el llamado al procedimiento `soundOff`. El volumen del sonido baja gradualmente hasta apagarse (en 0.7 segundos). El significado del procedimiento `soundOff` es:



Tiene un solo argumento que es el número del canal (un número entre 0 y 3) para el cual se desea apagar la generación del sonido.

El diagrama siguiente muestra cómo al ejecutar un llamado a `soundOn` el volumen sube hasta su máximo en el tiempo de attack, baja hasta el nivel de sustain en el tiempo de decay y se mantiene ahí hasta que se ejecuta un llamado a `soundOff` y en eso baja hasta cero en el tiempo de release:



La parte siguiente del programa:

```
setSoundSustain(0, 1);
setSoundDecay(0, 300);
pruebaSonido();
```

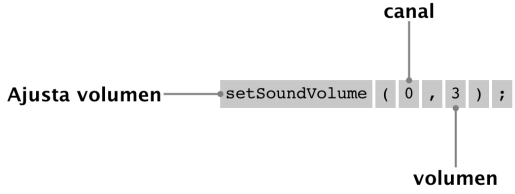
Define que el volumen de sustain debe ser un 33% del volumen máximo (el `1` como segundo argumento), selecciona un tiempo de decay de 0.3 segundos (300 milisegundos) y llama otra vez al procedimiento `pruebaSonido`. Emplea el botón para ejecutar todo esto paso por paso y escucha cómo al ejecutar el llamado a `soundOn`, dentro del procedimiento `pruebaSonido`, el volumen

sube hasta un máximo y después vuelve a bajar antes de mantenerse constante. Al ejecutar el `soundOff` el volumen baja hasta cero.

La parte final del programa:

```
setSoundVolume(0, 3);
pruebaSonido();
```

Emplea el procedimiento `setSoundVolume` para seleccionar un volumen máximo más bajo. Este es el significado de ese procedimiento:



El primer argumento es el canal (un número entre 0 y 3) y el segundo argumento es el volumen máximo que debe ser un número entre 0 y 15.

Emplea el botón ⏭ para ejecutar el resto del programa paso por paso y escucha cómo ahora el sonido se escucha con un volumen mucho más bajo.

Múltiples sonidos simultáneos

Con simpleJ es posible tener hasta los cuatro canales de audio generando cada uno un sonido diferente de forma simultánea.

Haz click en 📂 y abre el programa `channels.sj`:

```
setSoundFrequency(0, 4000);
setSoundVolume(0, 8);
setSoundSustain(0, 3);
setSoundAttack(0, 0);
setSoundRelease(0, 0);

setSoundFrequency(1, 10000);
setSoundVolume(1, 8);
setSoundSustain(1, 3);
setSoundAttack(1, 0);
setSoundRelease(1, 0);

soundOn(0);
```

```
pause(1);

soundOn(1);
pause(1);

soundOff(0);
pause(1);

soundOn(0);
pause(1);

soundOff(1);
pause(1);
```

Haz click en ▶ para ejecutar el programa. Se escucha un sonido, sin que se interrumpa ese sonido se escucha una segundo sonido con otra frecuencia, se apaga el primer sonido, vuelve a empezar el primer sonido y poco después se apaga el segundo sonido.

Ahora haz click en ⏭ para ejecutar el programa paso por paso.

El programa empieza con:

```
setSoundFrequency(0, 4000);
setSoundVolume(0, 8);
setSoundSustain(0, 3);
setSoundAttack(0, 0);
setSoundRelease(0, 0);

setSoundFrequency(1, 10000);
setSoundVolume(1, 8);
setSoundSustain(1, 3);
setSoundAttack(1, 0);
setSoundRelease(1, 0);
```

Aquí se configuran los parámetros para los dos primeros canales (el 0 y el 1) de manera que tengan su volumen máximo en 8, su sustain en 100% (un 3) y que tanto el attack como el release sean instantáneos (un 0). La única diferencia es que en el primero su parámetro de frecuencia es 4000, mientras que para el segundo es 10000. Emplea el botón ⏭ para ejecutar todas estas instrucciones.

La parte que sigue del programa es muy sencilla:

```
soundOn(0);
pause(1);

soundOn(1);
pause(1);

soundOff(0);
pause(1);

soundOn(0);
pause(1);

soundOff(1);
pause(1);
```

Enciende el primer canal de audio; hace una pausa de un segundo; enciende el segundo canal de audio; hace otra pausa de un segundo; apaga el primer canal de audio; vuelve a hacer una pausa; vuelve a encender el primer canal de audio; hace una pausa; apaga el segundo canal de audio.

Emplea el botón  para ejecutar estas instrucciones y escuchar cómo enciende y apaga cada canal de audio.

Se pueden cambiar dinámicamente los parámetros de un canal

Es posible generar sonidos más interesantes cambiando dinámicamente los parámetros de un canal de audio mientras se está generando un sonido.

Haz click en y abre el programa `dinamico.sj`:

```
setSoundFrequency(0, 4000);
setSoundVolume(0, 15);
setSoundSustain(0, 3);
setSoundAttack(0, 0);
setSoundRelease(0, 0);

soundOn(0);
pause(1);

for (var f = 4000; f < 20000; f = f + 100) {
  setSoundFrequency(0, f);
  pause(.04);
```

```
}
for (var f = 20000; f >= 4000; f = f - 100) {
  setSoundFrequency(0, f);
  pause(.04);
}
pause(1);

for (var i = 0; i < 10; i++) {
  for (var f = 4000; f < 9000; f = f + 100) {
    setSoundFrequency(0, f);
    pause(0.01);
  }
  for (var f = 9000; f >= 4000; f = f - 100) {
    setSoundFrequency(0, f);
    pause(0.01);
  }
}
```

Haz click en ▶ para ejecutar el programa. Se escucha un sonido que va cambiando de tono. Se hace más agudo y se regresa hacia un tono más grave. Después hace lo mismo varias veces más rápidamente.

Si quieres puedes emplear el botón para ejecutar este programa paso por paso, aunque puede que sea un poco tedioso porque contiene varios ciclos.

El principio del programa:

```
setSoundFrequency(0, 4000);
setSoundVolume(0, 15);
setSoundSustain(0, 3);
setSoundAttack(0, 0);
setSoundRelease(0, 0);

soundOn(0);
pause(1);
```

Configura los parámetros para el canal cero de manera que tenga una frecuencia de 4000, tenga el volúmen máximo posible (el 15 en `setSoundVolume`), se mantenga en ese máximo (el 3 en `setSoundSustain`) y tanto el attack como el release sean instantáneos (un cero). Después de eso inicia la generación del sonido con el llamado a `soundOn` y hace una pausa de un segundo.

La parte que sigue del programa:

```
for (var f = 4000; f < 20000; f = f + 100) {
  setSoundFrequency(0, f);
  pause(.04);
}
for (var f = 20000; f >= 4000; f = f - 100) {
  setSoundFrequency(0, f);
  pause(.04);
}
pause(1);
```

Contiene dos ciclos. En el primero emplea una variable `f` que varía desde 4000 hasta 19900 (menor que 20000) en incrementos de 100 en 100. Dentro de ese ciclo emplea el contenido de la variable `f` para modificar la frecuencia del sonido que se está escuchando y hace una pausa de cuatro centésimas de segundo antes de pasar a la siguiente iteración del ciclo. El segundo ciclo hace lo mismo pero esta vez la variable `f` toma los valores desde 20000 hasta 4000 en decrementos de 100 en 100. Al terminar estos dos ciclos hace una pausa de un segundo.

La última parte del programa:

```
for (var i = 0; i < 10; i++) {
  for (var f = 4000; f < 9000; f = f + 100) {
    setSoundFrequency(0, f);
    pause(0.01);
  }
  for (var f = 9000; f >= 4000; f = f - 100) {
    setSoundFrequency(0, f);
    pause(0.01);
  }
}
```

Contiene dos ciclos muy similares a los anteriores (las únicas diferencias son que ahora la frecuencia sube sólo hasta 9000 en vez de llegar hasta 20000 y que la pausa es de una centésima de segundo) que a su vez están dentro de otro ciclo para que se ejecuten 10 veces. Con esto se logra que la frecuencia del sonido suba y baje diez veces rápidamente.

Sonido dinámico con el procedimiento `sfi`

En el nivel anterior vimos que el lugar ideal para cambiar la posición de los sprites en la pantalla era dentro de un procedimiento `vbi`, de la misma manera también existe un procedimiento dentro del cual es conveniente hacer los

cambios dinámicos a los parámetros de los canales de audio. Este procedimiento se llama `sfi`, su nombre es una abreviación de *Sound Frame Interrupt*.

Si definimos un procedimiento sin argumentos llamado `sfi` este va a ser llamado 25 veces por segundo. El sintetizador de audio de simpleJ genera el audio en unos paquetes llamados `frames`, genera 25 frames por segundo, entre la generación de un frame y la siguiente llama al procedimiento `sfi` para darle la oportunidad de cambiar los parámetros de los canales de audio sin que esto interfiera con la generación de sonido (estas interferencias tienden a ser mínimas pero es mejor evitarlas).

Veamos un ejemplo. Haz click en  y abre el programa `sfi.sj`:

```
for (var i = 0; i < 4; i++) {
  setSoundVolume(i, 5);
  setSoundSustain(i, 3);
  soundOn(i);
}

sfi() {
  for (var i = 0; i < 4; i++)
    setSoundFrequency(i, random(10000) + 5000);
}
```

Ahora haz click en  para ejecutarlo. Se escuchan múltiples sonidos saltando rápidamente de una frecuencia a otra. Para terminar de ejecutar el programa haz click en el botón .

Haz click en el botón  para empezar a ejecutar el programa paso por paso.

El principio del programa:

```
for (var i = 0; i < 4; i++) {
  setSoundVolume(i, 5);
  setSoundSustain(i, 3);
  soundOn(i);
}
```

Emplea un ciclo para inicializar algunos parámetros de los cuatro canales de sonido. La variable `i` va tomando los valores desde el 0 hasta el 3 y para cada uno de esos canales:

1. Pone su volumen en 5

2. Pone su sustain en 100% (el 3)

3. Inicia la generación de audio en ese canal

Emplea el botón ▶| para ejecutar paso por paso las cuatro iteraciones del ciclo. No se escucha ningún sonido porque no le hemos asignado ninguna frecuencia a esos canales (tal vez logres escuchar simplemente un click al ejecutar el procedimiento `soundOn` para cada canal).

La parte siguiente del programa:

```
sfi() {
  for (var i = 0; i < 4; i++)
    setSoundFrequency(i, random(10000) + 5000);
}
```

Es donde se define el procedimiento `sfi`. Lo único que hace este procedimiento es iterar empleando una variable local `i` para modificar la frecuencia de los cuatro canales. La frecuencia que le asigna a cada uno de ellos es un número al azar entre 5000 y 14999 (genera un número al azar entre 0 y 9999 al que le suma 5000).

Haz click una vez más en el botón ▶| para ejecutar la definición del procedimiento `sfi`. Como este procedimiento se ejecuta automáticamente 25 veces por segundo, se escuchan los sonidos cambiando de una frecuencia a otra sin que sea necesario volver a hacer click en el botón ▶|. Para terminar de ejecutar el programa haz click en el botón ■.

Control de sonidos generados con el procedimiento `sfi`

Ya vimos que la manera correcta de generar un sonido dinámico es por medio del procedimiento `sfi`. Ahora vamos a ver cómo controlar la generación de ese sonido desde un programa.

Usa el botón 📂 para abrir el programa `main.sj`. Es un programa muy corto:

```
final BOTON_ESPACIO = 64;

var sonido = false;
var haciendoSonido = false;
```

```
var contadorSonido;

setSoundAttack(0, 150);
setSoundDecay(0, 850);

sfi() {
  if (haciendoSonido) {
    setSoundFrequency(0, contadorSonido * 200 + 10000);
    contadorSonido++;
    if (contadorSonido == 25) {
      haciendoSonido = false;
      sonido = false;
      soundOff(0);
    }
  } else {
    if (sonido) {
      haciendoSonido = true;
      contadorSonido = 0;
      setSoundFrequency(0, 10000);
      soundOn(0);
    }
  }
}

showAt("Apoya la barra de espacio", 3, 10);
showAt("para escuchar un sonido", 3, 11);

while (true) {
  if (readCtrlOne() == BOTON_ESPACIO && sonido == false)
    sonido = true;
}
```

La primera parte:

```
final BOTON_ESPACIO = 64;

var sonido = false;
var haciendoSonido = false;
var contadorSonido;

setSoundAttack(0, 150);
setSoundDecay(0, 850);
```

Declara una constante `BOTON_ESPACIO` para poder detectar cuando se apoya la barra de espacio; crea las tres variables `sonido`, `haciendoSonido` y `contadorSonido`; configura los tiempos de attack (0.15 segundos) y de decay (0.85 segundos) para el canal cero de audio.

Esto es para lo que se emplean esas tres variables:

1. `sonido`: inicialmente tiene el valor `false`; al almacenar un `true` en esta variable se inicia la generación del sonido; al terminar de generar el sonido su valor regresa a `false`.

2. `haciendoSonido`: inicialmente tiene el valor `false`; mientras se está generando el sonido tiene el valor `true`; al terminar de generar el sonido su valor regresa a `false`.

3. `contadorSonido`: se emplea para ir contando cuántas veces se ha ejecutado el procedimiento `sfi` durante la generación del sonido; de una vez se aprovecha para controlar la frecuencia del sonido.

Nota

Aunque las descripciones de las variables `sonido` y `haciendoSonido` parecen casi idénticas, en realidad cada una sirve para algo un poco diferente. La variable `sonido` sirve para que el programa le indique al procedimiento `sfi` que debe empezar a generar un sonido (almacenando un `true` en esa variable) y para que el procedimiento `sfi` le indique al programa que ya terminó de generar el sonido (almacenando un `false` en esa variable). Mientras que la variable `haciendoSonido` se emplea para que el procedimiento `sfi` sepa que está en el proceso de generar un sonido (cuando la variable contiene `true`).

La parte siguiente del programa es la declaración del procedimiento `sfi`:

```
sfi() {
  if (haciendoSonido) {
    setSoundFrequency(0, contadorSonido * 200 + 10000);
    contadorSonido++;
    if (contadorSonido == 25) {
      haciendoSonido = false;
      sonido = false;
      soundOff(0);
```

```
    }
  } else {
    if (sonido) {
      haciendoSonido = true;
      contadorSonido = 0;
      setSoundFrequency(0, 10000);
      soundOn(0);
    }
  }
}
```

Para entender cómo funciona este procedimiento hay que entender que puede ser llamado en dos situaciones: se está generando un sonido o no se está generando un sonido. Si se está generando un sonido, entonces tiene que modificar la frecuencia del sonido y checar si ya terminó de generarlo. Si no se está generando un sonido, entonces tiene que checar si ya hay que empezar a generarlo.

Para saber si se está generando un sonido basta con checar el valor almacenado en la variable `haciendoSonido`. Justamente para eso está el `if (haciendoSonido)` al principio del procedimiento. Si el valor de `haciendoSonido` es `true` entonces se está generando un sonido y ejecuta el primer cuerpo del `if`, de lo contrario ejecuta el segundo cuerpo del `if` (lo que viene después del `else`).

Veamos primero lo que hace si no se está generando ningún sonido en ese momento. Dentro del cuerpo que viene después del `else` sólo hay un `if` que checa si el valor de la variable sonido es `true`, en cuyo caso (es decir, sí hay que empezar a generar un sonido) hace lo siguiente:

1. Almacena `true` en de la variable `haciendoSonido` para indicar que se está generando un sonido.

2. Almacena un cero en la variable `contadorSonido`. Esta variable se emplea para controlar el cambio dinámico de frecuencia del sonido y también para saber cuándo se termina de generar el sonido.

3. Pone en `10000` la frecuencia para el canal cero de audio (llamando al procedimiento `setSoundFrecuency`).

4. Inicia la generación del sonido (llamando al procedimiento `soundOn`).

En el otro caso, si se está generando un sonido, hace esto:

1. Modifica la frecuencia del sonido empleando el valor de la variable `contadorSonido` (lo multiplica por `200` y le suma `10000`).

2. Incrementa en uno el valor de la variable `contadorSonido`.

3. Checa si el valor de la variable `contadorSonido` ya llegó a `25`. Si ya llegó entonces:

    a. Almacena `false` en la variable `haciendoSonido` para indicar que ya no está generando un sonido.

    b. Almacena `false` en la variable `sonido` para indicar que ya está listo para volver a generar un sonido.

    c. Apaga la generación del sonido (llamando al procedimiento `soundOff`).

La parte final del programa:

```
showAt("Apoya la barra de espacio", 3, 10);
showAt("para escuchar un sonido", 3, 11);

while (true) {
  if (readCtrlOne() == BOTON_ESPACIO && sonido == false)
    sonido = true;
}
```

Despliega en la pantalla el mensaje indicando que hay que apoyar la barra de espacio para generar un sonido; después entra a un ciclo infinito donde checa si se presionó la barra de espacio (si `readCtrlOne() == BOTON_ESPACIO`) y si se puede generar un sonido en ese momento (si `sonido == false`), en caso de que ambas sean ciertas ordena generar un sonido almacenando `true` en la variable `sonido` (con lo cual el procedimiento `sfi` inicia la generación del sonido).

Nivel 12: ¿Puedes leer un programa?

Un juego con tiles redefinidos y sprites

Ahora vamos a ver un juego que emplea tiles redefinidos y sprites. En el menú de **Project** selecciona la opción **Switch to another project...** (**Project > Switch to another project...**) y selecciona el proyecto `Nivel_12`. Haz click en ▶ para probar el programa. Es un juego en el que usas una raqueta para pegarle a una pelota que destruye los bloques que están en la pantalla.

Un programa se debe poder leer

Un programa no es únicamente para decirle a una computadora qué es lo que debe hacer. Un programa también debe estar escrito de manera que otra persona pueda leerlo y entenderlo. De hecho, cuando un programador escribe un programa su principal objetivo debe ser que alguien que lo lea, pueda entender claramente qué es lo que se le está ordenando a la computadora que haga.

Una parte importante de cualquier programa es su documentación. Además de los comentarios incluidos dentro del texto del programa se acostumbra también escribir algún documento con explicaciones y diagramas que ayudan a entender cómo funciona, pero esta documentación es simplemente auxiliar. **La documentación más importante de cualquier software es el propio programa.** Junto con un programa pueden venir cientos (a veces son miles) de páginas de documentación adicional, pero la documentación definitiva son las instrucciones dentro del programa porque ¡eso es lo único que la computadora toma en cuenta al ejecutarlo! (Hasta los comentarios dentro del programa son invisibles para la computadora).

Claro que no cualquiera puede leer un programa. Para eso hay que entender en qué consiste programar una computadora y también hay que conocer el lenguaje de programación en el que está escrito.

Tú ya puedes leer un programa

Si seguiste con cuidado los once niveles anteriores entonces tú ya puedes leer un programa. No es sencillo, pero tampoco es tan difícil. Basta con poner

atención y tener un poco de paciencia. Para entender cómo está escrito un programa hay que leerlo varias veces. La primera vez lo lees desde el principio hasta el final, no vas a entender todavía cómo funciona, pero con eso te enteras un poco de cómo está estructurado y qué partes tiene. En una segunda leída tratas de entender en qué orden la computadora va a ejecutar cada uno de los procedimientos y qué es lo que hace cada uno de ellos. Algunos de los procedimientos son muy sencillos y basta con leerlos una vez para entenderlos, otros son más complejos y pueden requerir que les dediques un poco de tiempo a analizarlos con cuidado. También hay que ponerle mucha atención a las variables globales (las que están definidas fuera de los procedimientos) porque estas variables son las más importantes para entender cómo funciona el programa.

Para que pruebes qué tan bien puedes leer un programa toda la explicación de este nivel está dentro del texto del programa. Nada más te vamos a dar unas cuantas indicaciones para que lo puedas entender más fácilmente:

1. El programa está estructurado con base en *estados*. Lo que hace la computadora depende del estado en el que se encuentre en ese momento el juego. Hay cinco estados:

   a. CARGANDO

   b. PRESENTACION

   c. LANZA PELOTA

   d. JUEGA

   e. GAME OVER

   Cuando se dan ciertas condiciones el juego pasa de un estado a otro. Por ejemplo, cuando está en el estado JUEGA y el jugador pierde su última pelota, el juego pasa al estado GAME OVER.

   Cuando el juego pasa de un estado a otro se dice que ocurre una *transición*. Al ocurrir una transición de un estado a otro, la computadora puede ejecutar ciertas accciones. Por ejemplo, cada vez que ocurre una transición al estado PRESENTACION la computadora vuelve a desplegar la imagen de presentación del videojuego.

   El estado actual del juego se almacena dentro de la variable `estado`.

2. Antes de leer los archivos con los datos para los tiles y los sprites se define el procedimiento `vbi`. Esto es para que mientras que se leen estos archivos ya se esté ejecutando el código que corresponde al estado CARGANDO (que muestra una animación muy sencilla en la pantalla).

3. Este programa emplea un operador lógico que todavía no habíamos visto, es el operador `||` (dos rayas verticales), este operador se llama *or* (la palabra *or* en inglés significa *o*). Este operador se emplea para crear una condición compuesta a partir de dos condiciones sencillas, de la manera siguiente:

   `condicion1 || condicion2`

   Esta es una condición compuesta, que es verdadera cuando *condición1 o condicion2* es verdadera (o si las dos son verdaderas).

   Para entender cómo funciona el *or*, ejecuta paso por paso el programa `or.sj` que se encuentra dentro del proyecto `Nivel_12`.

4. Empleamos aquí el `break` de una nueva manera. Anteriormente habíamos visto que se podía emplear `break` para indicar dónde se debe terminar de ejecutar un case dentro de un `switch`. También se puede emplear un `break` dentro de un ciclo para terminar anticipademente la ejecución de un ciclo. Para entender cómo funciona, ejecuta paso por paso el programa `break.sj` que se encuentra dentro del proyecto `Nivel_12`.

5. Este juego combina tiles con sprites. Recuerda que al colocar un sprite su posición se especifica en pixeles, mientras que cuando colocas un tile su posición se especifica en tiles (que son múltiplos de 8 pixeles).

6. Una *bandera* es una variable que se emplea para indicar si ya ocurrió algo.

El programa

```
/* Constantes para los botones*/
final BOTON_IZQUIERDA = 4;
final BOTON_DERECHA = 8;
final BOTON_RETURN = 16;
final BOTON_ESPACIO = 64;

/* El tile con un espacio en blanco,
   para borrar bloques */
final ESPACIO = 32;
```

```
/* Los cinco estados en los que puede estar el juego */
final ESTADO_CARGANDO = 0;
final ESTADO_PRESENTACION = 1;
final ESTADO_LANZA_PELOTA = 2;
final ESTADO_JUEGA = 3;
final ESTADO_GAME_OVER = 4;

/* Posicion vertical, en tiles, del primer y el ultimo
   renglon con bloques */
final PRIMER_RENGLON = 3;
final ULTIMO_RENGLON = 8;

/* Numero de renglones con bloques (en tiles) */
final RENGLONES = ULTIMO_RENGLON - PRIMER_RENGLON + 1;

/* Ancho de un bloque (en tiles) */
final ANCHO_BLOQUE = 4;

/* Cuantos bloques caben en un renglon */
final BLOQUES_POR_RENGLON = 32 / ANCHO_BLOQUE;

/* Cuantos bloques hay en total */
final TOTAL_BLOQUES = BLOQUES_POR_RENGLON * RENGLONES;

/* Posicion vertical, en pixeles, de la parte
   superior de la raqueta */
final Y_RAQUETA = 192 - 6;

/* Ancho de la raqueta (en pixeles) */
final ANCHO_RAQUETA = 32;

/* Velocidad de la raqueta.
   En pixeles por cada vez que se ejecuta vbi */
final VELOCIDAD_RAQUETA = 10;

/* Las tres velocidades para la pelota.
   En pixeles por cada vez que se ejecuta vbi */
final VELOCIDAD1_PELOTA = 4;
final VELOCIDAD2_PELOTA = 5;
final VELOCIDAD3_PELOTA = 6;

/* Posicion vertical inicial de la esquina
   superior de la pelota al lanzarla (en pixeles) */
```

```
final Y_INICIAL_PELOTA = 80;

/* El estado actual del juego */
var estado = ESTADO_CARGANDO;

/* Contador empleado en la animacion mientras que
   se cargan los archivos de tiles y de sprites.
   Tambien se emplea para la pause en GAME OVER */
var contador = 0;

/* Bandera para indicar cuando se terminaron de
   leer los archivos de tiles y de sprites */
var terminoDeCargar = false;

/* Secuencia de caracteres para la animacion
   mientras que se cargan los datos */
var animacionCargando = ["|", "/", "-", "\\"];

/* Cuantos bloques quedan sin destruir */
var cuantosBloques;

/* Arreglo bidimensional (en realidad es un arreglo
   de arreglos) de booleanos para saber si todavia
   hay un bloque en cierta posicion de la pantalla.
   Contiene un elemento por cada posicion en la que
   puede haber un bloque.
   Si hay un bloque en esa posicion entonces el
   elemento contiene true, de lo contario contiene
   false */
var bloques =
       new array[RENGLONES][BLOQUES_POR_RENGLON];

/* Cuantas pelotas le quedan al jugador (incluyendo
   la pelota con la que esta jugando */
var pelotas;

/* Cuantos puntos ha ganado hasta ahora el jugador */
var puntos;

/* La posicion horizontal del borde izquierdo de
   la raqueta (en pixeles) */
var xRaqueta;

/* La posicion horizontal del borde izquierdo de
```

```
   la pelota (en pixeles) */
var xPelota;

/* La posicion vertical del borde superior de
   la pelota (en pixeles) */
var yPelota;

/* La direccion horizontal en la que se esta
   moviendo la pelota.
   Un -1 indica hacia la izquierda.
   Un 1 indica hacia la derecha. */
var dxPelota;

/* La direccion vertical en la que se esta
   moviendo la pelota.
   Un -1 indica hacia arriba.
   Un 1 indica hacia abajo. */
var dyPelota;

/* La velocidad con la cual se esta moviendo
   la pelota (en pixeles por cada vez que se
   ejecuta vbi) */
var velocidadPelota;

/* Coloca todos los bloques en la pantalla */
colocaBloques() {
  /* Dibuja los seis renglones de bloques */
  for (var y = PRIMER_RENGLON;
       y <= ULTIMO_RENGLON;
       y++) {
    // Los tiles a emplear para este renglon. En este
    // ciclo la variable 'base' va teniendo los valores
    // 0, 0, 3, 3, 6 y 6
    var base = (y - PRIMER_RENGLON) / 2 * 3;
    for (var x = 0; x < 32; x++) {
      // El tile que corresponde a esta posicion de la
      // pantalla. La variable 'x' va tomando los
      // valores 0, 1, 1 y 2 por cada uno de los bloques
      var tile = x % 4;
      if (tile > 1)
        tile--;
      putAt(base + tile, x, y);
    }
  }
```

```
  // Falta destruir todos los bloques
  cuantosBloques = TOTAL_BLOQUES;

  // Indica que hay un bloque en cada posicion
  // almacenando 'true' en todos los elementos
  // del arreglo 'bloques'
  for (var r = 0; r < RENGLONES; r++)
    for (var c = 0; c < BLOQUES_POR_RENGLON; c++)
      bloques[r][c] = true;
}


/* Despliega en la esquina superior izquierda
   de la pantalla cuantas pelotas le quedan al
   jugador */
muestraPelotas() {
  showAt("PELOTAS: " + pelotas, 0, 0);
}


/* Convierte un numero a un string con ceros
   a la izquierda (para desplegar los puntos).
   n: cuantos digitos debe contener el string
   numero: el numero */
cerosAlPrincipio(n, numero) {
  // Convierte 'numero' a un string
  numero = "" + numero;

  // Agrega el numero necesario de ceros a la
  // izquierda
  while (length(numero) < n)
    numero = "0" + numero;

  return numero;
}

/* Depliega en la esquina superior derecha
   de la pantalla cuantos puntos ha ganado
   el jugador */
muestraPuntos() {
  showAt("PUNTOS:", 18, 0);
  showAt(cerosAlPrincipio(5, puntos), 26, 0);
}
```

```
/* Modifica la posicion horizontal de la
   raqueta para que quede centrada en la
   pantalla */
centraRaqueta() {
  xRaqueta = (256 - ANCHO_RAQUETA) / 2;
}


/* Posiciona los dos sprites de la raqueta
   en la pantalla. Calculando antes su nueva
   posicion en caso de que el jugador presione
   la flecha izquierda o derecha */
mueveRaqueta() {
  // Lee control del jugador
  var btn = readCtrlOne();

  // Si esta presionado el boton IZQUIERDA
  // mueve la raqueta a la izquierda sin
  // salirse de la pantalla
  if (btn == BOTON_IZQUIERDA) {
    xRaqueta = xRaqueta - VELOCIDAD_RAQUETA;
    if (xRaqueta < 0)
      xRaqueta = 0;
  }

  // Si esta presionado el boton DERECHA
  // mueve la raqueta a la derecha sin
  // salirse de la pantalla
  if (btn == BOTON_DERECHA) {
    xRaqueta = xRaqueta + VELOCIDAD_RAQUETA;
    if (xRaqueta > 256 - ANCHO_RAQUETA)
      xRaqueta = 256 - ANCHO_RAQUETA;
  }

  // Coloca los dos sprites de la raqueta
  // en su posicion en la pantalla
  putSpriteAt(1, xRaqueta, Y_RAQUETA);
  putSpriteAt(2, xRaqueta + 16, Y_RAQUETA);
}


/* Checa si el punto con coordenadas (x, y),
```

```
   en pixeles, toca alguno de los bloques.
   En caso de ser cierto decrementa de uno
   el numero de bloques que quedan por
   destruir, almacena false en el elemento
   correspondiente del arreglo 'bloque', y
   devuelve 'true'. De lo contrario simplemente
   devuelve 'false' */
tocaEn(x, y) {
  // Convierte la coordenada horizontal 'x'
  // (en pixeles) a una columna 'c' (en bloques)
  var c = x / 32;

  // Si esta mas alla del borde derecho de
  // la pantalla entonces no toco ninguno
  // de los bloques (en este programa nunca
  // ocurre que x sea negativo y por lo
  // tanto no es necesario checar del lado
  // izquierdo
  if (c >= BLOQUES_POR_RENGLON)
    return false;

  // Convierte la coordenada vertical 'y'
  // (en pixeles) a un renglon 'r' (en bloques)
  var r = y / 8 - PRIMER_RENGLON;

  // Si esta antes del primer renglon o
  // despues del ultimo renglon entonces
  // no puede haber tocado alguno de los
  // bloques
  if (r < 0 || r >= RENGLONES)
    return false;

  // Si no hay un bloque ahi entonces no
  // es necesario hacer nada
  if (!bloques[r][c])
    return false;

  // Si llegamos hasta aqui quiere decir
  // que si toco un bloque.
  // Marca en el arreglo 'bloques' que ya
  // no esta ese bloque
  bloques[r][c] = false;

  // Decrementa de uno el numero de bloques
```

```
  // que quedan por destruir
  cuantosBloques--;

  // Borra el bloque de la pantalla
  for (var i = 0; i < ANCHO_BLOQUE; i++)
    putAt(ESPACIO,
          c * ANCHO_BLOQUE + i,
          PRIMER_RENGLON + r);

  // Agrega un punto al jugador y actualiza
  // la informacion de los puntos en la pantalla
  puntos++;
  muestraPuntos();

  // Si toco un bloque rojo (renglon es 0 o 1)
  // asegura que la velocidad de la pelota sea
  // por lo menos la velocidad que corresponde
  // a destruir un bloque rojo
  if (r < 2 && velocidadPelota < VELOCIDAD3_PELOTA)
    velocidadPelota = VELOCIDAD3_PELOTA;

  // Si toco un bloque amarillo (renglon es 2 o 3)
  // asegura que la velocidad de la pelota sea
  // por lo menos la velocidad que corresponde
  // a destruir un bloque amarillo
  if (r < 4 && velocidadPelota < VELOCIDAD2_PELOTA)
    velocidadPelota = VELOCIDAD2_PELOTA;

  // Devuelve 'true' porque si toco un bloque
  return true;
}


/* Checa si el borde izquierdo de la pelota
   le pego a un bloque */
tocaBordeIzquierdo() {
  // Checa en las coordenadas que corresponden
  // a la esquina superior izquierda y a la
  // esquina inferior izquierda de la pelota
  return tocaEn(xPelota, yPelota) ||
         tocaEn(xPelota, yPelota + 8);
}
```

```
/* Checa si el borde derecho de la pelota
   le pego a un bloque */
tocaBordeDerecho() {
  // Checa en las coordenadas que corresponden
  // a la esquina superior derecha y a la
  // esquina inferior derecha de la pelota
  return tocaEn(xPelota+8, yPelota) ||
         tocaEn(xPelota+8, yPelota + 8);
}


/* Checa si el borde superior de la pelota
   le pego a un borde */
tocaBordeSuperior() {
  // Checa en las coordenadas que corresponden
  // a la esquina superior izquiera y a la
  // esquina superior derecha de la pelota
  return tocaEn(xPelota, yPelota) ||
         tocaEn(xPelota + 8, yPelota);
}


/* Checa si el borde inferior de la pelota
   le pego a un borde */
tocaBordeInferior() {
  // Checa en las coordenadas que corresponden
  // a la esquina inferior izquierda y a la
  // esquina inferior derecha de la pelota
  return tocaEn(xPelota, yPelota + 8) ||
         tocaEn(xPelota + 8, yPelota + 8);
}


/* Mueve la pelota de un pixel (tanto en la
   direccion horizontal como vertical).
   Checa si le pego a un bloque, a un borde
   de la pantalla o a la raqueta para modificar
   la direccion en la que se mueve. Si destruyo
   el ultimo bloque que quedaba entonces vuelve
   a colocar todos los bloques. Si la pelota se
   salio de la pantalla entonces decrementa el
   numero de pelotas, genera la transicion al
   estado LANZA_PELOTA o al estado GAME_OVER
   y delvuelve 'false' para indicar que la
```

```
   pelota ya no se esta moviendo, de lo contrario
   devuelve 'true' para que se siga moviendo la
   pelota */
muevePelota() {
  // Calcula la nueva posicion de la pelota
  xPelota = xPelota + dxPelota;
  yPelota = yPelota + dyPelota;

  // Si la pelota pega con el borde izquierdo de
  // la pantalla entonces ahora se mueve hacia
  // la derecha
  if (xPelota <= 0) {
    xPelota = 0;
    dxPelota = 1;
    soundOn(0);
  }

  // Si la pelota pega con el borde derecho de la
  // pantalla entonces ahora se mueve hacia la
  // izquierda
  if (xPelota >= 256 - 8) {
    xPelota = 256 - 8;
    dxPelota = -1;
    soundOn(0);
  }

  // Si la pelota pega con el borde superior de la
  // pantalla entonces ahora se mueve hacia abajo
  if (yPelota <= 0) {
    yPelota = 0;
    dyPelota = 1;
    soundOn(0);
  }

  // Si el borde inferior de la raqueta esta a la
  // altura de la raqueta, o mas abajo.
  if (yPelota + 8 >= Y_RAQUETA) {
    // Si el borde derecho de la pelota esta a la
    // izquierda de la raqueta o si el borde
    // izquierdo de la pelota esta a la derecha de
    // la raqueta (es decir, si no toca la raqueta)
    if (xPelota + 8 < xRaqueta ||
        xPelota > xRaqueta + ANCHO_RAQUETA) {
      // Si la pelota ya esta abajo del borde superior
```

```
      // de la raqueta
      if (yPelota >= Y_RAQUETA) {
        soundOn(1);

        // Una pelota menos
        pelotas--;
        muestraPelotas();

        // Si todavia le quedan pelotas se pasa al
        // estado para lanzar otra pelota, de lo
        // contrario ya se acabo el juego
        if (pelotas > 0)
          transicionALanzaPelota();
        else
          transicionAGameOver();

        // Quita la pelota de la pantalla
        putSpriteAt(0, -10, 0);

        // Ya no se debe de seguir moviendo la pelota
        return false;
      }
    } else {
      // Rebota la pelota contra la raqueta
      yPelota = Y_RAQUETA - 8;
      dyPelota = -1;
      soundOn(0);
    }
  }

  // Checa si la pelota va hacia abajo
  if (dyPelota == 1) {
    // Checa si el borde inferior de la
    // pelota toco un bloque
    if (tocaBordeInferior()) {
      // Rebota hacia arriba
      dyPelota = -1;
      soundOn(0);
    }
  } else {
    // La pelota va hacia arriba, entonces
    // checa si su borde superior toco un
    // bloque
    if (tocaBordeSuperior()) {
```

```
      // Rebota hacia abajo
      dyPelota = 1;
      soundOn(0);
    }
  }

  // Checa si la pelota va hacia la derecha
  if (dxPelota == 1) {
    // Checa si el borde derecho de la
    // pelota toco un bloque
    if (tocaBordeDerecho()) {
      // Rebota hacia la izquierda
      dxPelota = -1;
      soundOn(0);
    }
  } else {
    // La pelota va hacia la izquierda,
    // entonces checa si su borde
    // izquierdo toco un bloque
    if (tocaBordeIzquierdo()) {
      // Rebota hacia la derecha
      dxPelota = 1;
      soundOn(0);
    }
  }

  // Si ya no hay bloques por destruir
  // entonces volver a colocar todos los
  // bloques
  if (cuantosBloques == 0)
    colocaBloques();

  // Coloca el sprite de la pelota en su
  // nueva posicion en la pantalla
  putSpriteAt(0, xPelota, yPelota);

  // Indica que la pelota sigue en movimiento
  return true;
}


/* Dibuja la pantalla de presentacion y
   pasa al estado PRESENTACION */
transicionAPresentacion() {
```

```
  for (var r = 0; r < 24; r++)
    for (var c = 0; c < 32; c++)
      putAt(tilesData.rows[r][c], c, r);
  estado = ESTADO_PRESENTACION;
}


/* Muestra un mensaje indicando como
   lanzar la pelota y pasa al estado
   LANZA_PELOTA */
transicionALanzaPelota() {
  // Despliega el mensaje
  showAt("Presiona ESPACIO para", 4, 12);
  showAt("lanzar la pelota", 4, 13);

  // Pasa al estado LANZA_PELOTA
  estado = ESTADO_LANZA_PELOTA;
}


/* Borra el mensaje que indica como
   lanzar la pelota, lanza la pelota y
   pasa al estado JUEGA */
transicionAJuega() {
  // Borra el mensaje
  showAt("                     ", 4, 12);
  showAt("               ", 4, 13);

  // Lanza la pelota asignandole una posicion
  // inicial cerca del centro de la pantalla,
  // un movimiento vertical hacia abajo y
  // eligiendo al azar si se mueve hacia la
  // izquiera o hacia la derecha
  yPelota = Y_INICIAL_PELOTA;
  xPelota = random(48) + 100;
  if (random(2) == 0)
    dxPelota = 1;
  else
    dxPelota = -1;
  dyPelota = 1;
  velocidadPelota = VELOCIDAD1_PELOTA;

  // Pasa al estado JUEGA
  estado = ESTADO_JUEGA;
```

```
}


/* Despliega el mensaje de "GAME OVER",
   modifica 'contador' para la pausa,
   esconde los dos sprites de la raqueta
   y pasa al estado GAME_OVER */
transicionAGameOver() {
  // Despliega el mensaje
  showAt("GAME OVER", 12, 13);

  // Duracion de la pausa antes de pasar
  // al estado de PRESENTACION
  contador = 60;

  // Esconde (poniendo fuera de la pantalla)
  // los dos sprites de la raqueta
  putSpriteAt(1, -16, 0);
  putSpriteAt(2, -16, 0);

  // Pasa al estado GAME_OVER
  estado = ESTADO_GAME_OVER;
}


/* Se ejecuta en cada llamado al procedimiento vbi
   mientras que el juego esta en el estado CARGANDO.
   Hace una animacion sencilla mientras que se estan
   leyendo los datos para los tiles y los sprites.
   Al terminar de leerlos hace la transicion al
   estado PRESENTACION */
cargandoVBI() {
  // Despliega mensaje cambiando a cada vez el
  // caracter empleado para la animacion
  showAt("Cargando " + animacionCargando[contador],
         10, 12);
  contador++;
  if (contador == length(animacionCargando))
    contador = 0;

  // Si ya termino de cargar los datos entonces
  // pasa al estado PRESENTACION
  if (terminoDeCargar)
    transicionAPresentacion();
```

```
}


/* Se ejecuta en cada llamado al procedimiento
   vbi mientras que el juego esta en el estado
   PRESENTACION. Cuando el jugador presiona el
   boton RETURN inicializa las variables del
   juego, dibuja la pantalla, y hace la
   transicion al estado LANZA_PELOTA */
presentacionVBI() {
  // El jugador presiono RETURN?
  if (readCtrlOne() == BOTON_RETURN) {
    // entonces inicializa el juego
    puntos = 0;
    pelotas = 5;
    clear();
    muestraPelotas();
    muestraPuntos();
    colocaBloques();
    centraRaqueta();

    // y pasa al estado LANZA_PELOTA
    transicionALanzaPelota();
  }
}


/* Se ejecuta en cada llamado al procedimiento
   vbi mientras que el juego esta en el estado
   LANZA_PELOTA. Permite que el jugador mueva
   la raqueta y cuando este presiona ESPACIO
   hace la transicion al estado JUEGA */
lanzaPelotaVBI() {
  // Permite al jugador mover la raqueta
  mueveRaqueta();

  // El jugador presiono ESPACIO?
  if (readCtrlOne() == BOTON_ESPACIO)
    // entonces pasa al estado JUEGA
    transicionAJuega();
}


/* Se ejecuta en cada llamado al procedimiento
```

```
   vbi mientras que el juego esta en el estado
   JUEGA. Mueve la raqueta y la pelota.
   El cambio de estado a LANZA_PELOTA o a
   GAME_OVER esta incorporado dentro del
   procedimiento muevePelota. */
juegaVBI() {
  // Permite al jugador mover la raqueta
  mueveRaqueta();

  // Mueve la pelota. El control de la velocidad
  // se obtiene llamando al procedimiento
  // muevePelota el numero de veces indicado por
  // el valor de la variable 'velocidadPelota'.
  // La razon de hacerlo asi es que permite detectar
  // mas facilmente las colisiones con un bloque, la
  // raqueta o un borde de la pantalla.
  for (var i = 0; i < velocidadPelota; i++)
    // Ya no se esta moviendo la pelota?
    // (salio de la pantalla)
    if (!muevePelota())
      // entonces ya no hay que seguirla moviendo
      break;
}


/* Se ejecuta en cada llamado al procedimiento
   vbi mientras que el juego esta en el estado
   GAME_OVER. Hace una pausa decrementando un
   contador para controlar la duracion de la
   pausa. Cuando el valor del contador llega a
   cero hace la transicion al estado PRESENTACION */
gameOverVBI() {
  contador--;
  if (contador == 0)
    transicionAPresentacion();
}


/* El procedimiento vbi, que se ejecuta
  automaticamente cada vez que se termina de
  redibujar la pantalla (25 veces por segundo),
  simplemente le delega el trabajo al
  procedimiento que corresponde al estado
  actual del juego. Definimos este procedimiento
```

```
  antes de leer los datos de los tiles y los
  sprites para que se ejecute la animacion del
  estado CARGANDO mientras tanto */
vbi() {
  switch (estado) {
    case ESTADO_CARGANDO:
      cargandoVBI();
      break;

    case ESTADO_PRESENTACION:
      presentacionVBI();
      break;

    case ESTADO_LANZA_PELOTA:
      lanzaPelotaVBI();
      break;

    case ESTADO_JUEGA:
      juegaVBI();
      break;

    case ESTADO_GAME_OVER:
      gameOverVBI();
      break;
  }
}

/* Lee definiciones de tiles creadas con
   el tiles editor */
var tilesData = readTilesFile("tiles.tmap");

/* Pon colores en el mapa de colores */
for (var i = 0; i < 16; i++)
  setTileColor(i, tilesData.colors[i].red,
                  tilesData.colors[i].green,
                  tilesData.colors[i].blue);

/* Graba nuevas definiciones de tiles */
for (var i = 0; i < 256; i++)
  setTilePixels(i, tilesData.pixels[i]);


/* Lee definiciones de sprites creadas con
   el sprites editor */
```

```
var spritesData = readSpritesFile("sprites.smap");

/* Pon colores en el mapa de colores */
for (var i = 0; i < 15; i++)
  setSpriteColor(i, spritesData.colors[i].red,
                    spritesData.colors[i].green,
                    spritesData.colors[i].blue);

/* Graba nuevas definiciones de sprites de 16 por 16 */
for (var i = 0; i < 128; i++)
  setLargeSpritePixels(i, spritesData.largePixels[i]);

/* Graba nuevas definiciones de sprites de 8 por 8 */
for (var i = 0; i < 128; i++)
  setSmallSpritePixels(i, spritesData.smallPixels[i]);

/* Imagen de la pelota */
setSmallSpriteImage(0, 0);

/* Imagen de la raqueta */
setLargeSpriteImage(1, 0);
setLargeSpriteImage(2, 1);

/* Sonido para pelota pegandole a un bloque
   o al borde */
setSoundFrequency(0, 10000);
setSoundAttack(0, 10);
setSoundDecay(0, 100);
setSoundSustain(0, 0);
setSoundRelease(0, 1);
setSoundVolume(0, 15);

/* Sonido para pelota saliendo de la pantalla */
setSoundFrequency(1, 2000);
setSoundAttack(1, 100);
setSoundDecay(1, 300);
setSoundSustain(1, 0);
setSoundRelease(1, 1);
setSoundVolume(1, 15);

/* Modifica la bandera para que se ejecute
   la transicion a ESTADO_PRESENTACION  */
terminoDeCargar = true;
```

Conclusión

En estos doce niveles aprendiste los conceptos básicos de cómo se programa una computadora para hacer un videojuego. Pero todavía hay mucho más que puedes aprender, la mejor manera de hacerlo es escribiendo tus propios programas, estudiando cómo están escritos programas hechos por otras personas y leyendo libros acerca de temas más avanzados de programación. Una buena fuente de información es el sitio `http://www.simplej.com` en Internet. Ese sitio contiene artículos explicando algunas características más avanzadas de simpleJ y foros en los cuales puedes participar para intercambiar ideas con otras personas.

Apéndice A. Compilación

Los programas se procesan en varias fases

Para ejecutar un programa la computadora lo procesa en cuatro fases sucesivas. Las cuatro fases son las siguientes:

1. *Análisis Léxico.* Parte el texto del programa en piezas que se llaman *Tokens*.

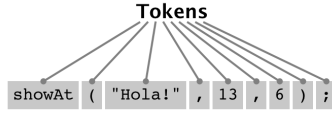
2. *Análisis Sintáctico.* Agrupa los tokens para formar *Sentencias*.

3. *Generación de Código.* Traduce las sentencias a una representación interna que se llama *Lenguaje de Máquina*.

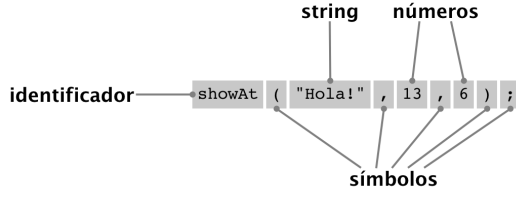
4. *Ejecución.* Ejecuta el programa en lenguaje de máquina.

En cada una de estas cuatro fases existe la posibilidad de que se encuentre con algo que no sepa cómo procesar y mande un mensaje de error.

Análisis Léxico. Cuando nuestro programa contiene algo como `showAt("Hola!", 13, 6);`, lo primero que hace la computadora es partirlo en piezas que se llaman *Tokens*:



Existen diferentes tipos de tokens. En este caso particular, éstos son los tipos de los tokens:



`showAt` es un *identificador*, `"Hola!"` es un *string*, `13` y `6` son *números*, y los demás son *símbolos*.

Cuando tienes un programa como éste:

```
showAt("Hola!", 13, 6);
pause(1.5);
```

La computadora lo convierte en esta lista de tokens:

```
showAt ( "Hola!" , 13 , 6 ) ; pause ( 1.5 ) ;
```

Al armar la lista de tokens la computadora ignora los espacios en blanco y los cambios de línea entre los tokens. Al escribir un programa puedes poner uno o más espacios en blanco entre dos tokens, la computadora simplemente los ignora. Un cambio de línea lo interpreta como un espacio en blanco.

Es decir que puedes escribir tu programa así:

```
showAt    (      "Hola!"    , 13    , 6
) ;  pause (   1.5  )    ;
```

Y de todas maneras va a funcionar. Porque la computadora lo convierte en exactamente la misma lista de tokens que en la versión anterior:

```
showAt ( "Hola!" , 13 , 6 ) ; pause ( 1.5 ) ;
```

Nota

Aunque para la computadora las dos versiones del programa son exactamente iguales, para una persona la primera versión es mucho más fácil de leer.

Lo que no debes hacer es poner espacios en blanco dentro de un token, porque la computadora entonces lo va a tomar como si fueran dos tokens diferentes. Por ejemplo, si escribes:

```
show At("Hola!", 1 3, 6);
```

Entonces la computadora, al analizarlo, lo va a convertir en esta lista de tokens:

```
show At ( "Hola!" , 1 3 , 6 ) ;
```

Y al pasar a la fase de análisis sintáctico te va a desplegar un error porque esa secuencia de tokens no corresponde a ninguna sentencia válida.

Nota

Dentro de un string puedes poner cuantos espacios en blanco quieras y de todas maneras la computadora lo sigue interpretando como un solo token. Eso es porque los strings están delimitados por unas comillas (") que le indican a la computadora dónde empieza y termina el string. Además, esos espacios en blanco no son ignorados por la computadora, se conservan como parte del token. Típicamente, estos espacios forman parte del algún mensaje que quieres desplegar.

Por otra parte, dentro de un string no está permitido tener cambios de línea. Todo el string tiene que estar en una sola línea. Es decir que si tratas de escribir un programa así:

```
showAt("Hola!
Como estas?", 13, 6);
```

La computadora te va a decir que tienes un error.

Análisis Sintáctico. La computadora tiene unas reglas de "gramática" que le permiten agrupar los tokens en "frases" bien formadas del lenguaje de programación. A estas frases se les llama *Sentencias*.

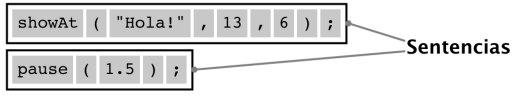
Una de esas reglas dice algo parecido a esto:

> Un identificador, seguido de *Argumentos* entre paréntesis, con un punto y coma al final, es una *Sentencia*.
>
> *Argumentos* es una lista de números o strings, posiblemente vacía, separados por comas.

Al ver una lista de tokens como ésta:

```
showAt ( "Hola!" , 13 , 6 ) ; pause ( 1.5 ) ;
```

Agrupa los tokens en estas dos sentencias:



Apéndice B. Expresiones, variables, constantes, tipos, operadores y prioridades

Expresiones

Uno de los conceptos básicos en el lenguaje de programación de simpleJ es el concepto de expresiones. Una expresión es algo que la computadora evalúa para obtener su valor. Las expresiones están formadas por constantes, variables, operadores y paréntesis. Estos son algunos ejemplos de expresiones:

```
45
-2
"hola"
2 + 3
(2 + 3) * (4 + 5)
a + b - 1
sqrt(16)
a[i + 1] = a[i - 1] * b.x
a <= b
a < b && !(b <= c)
```

Al evaluar una expresión la computadora emplea las *prioridades* de los operadores y su *asociatividad* para saber en qué orden debe aplicarlos. Cuando en una expresión se emplean varios operadores con diferentes prioridades, primero se evalúan los operadores de más alta prioridad. Cuando hay varios operadores con la misma prioridad, entonces se emplea la asociatividad de estos operadores para determinar en qué orden se evalúan. Se le puede indicar a la computadora que evalúe la expresión en otro orden por medio de paréntesis.

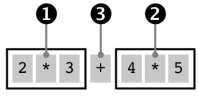
Veamos unos cuantos ejemplos empleando los operadores aritméticos básicos que son: `+` (suma), `-` (resta), `*` (multiplicación) y `/` (división).

El operador + tiene la misma prioridad que el operador -; el operador * tiene la misma prioridad que el operador /; la prioridad de los operadores * y / es mayor que la prioridad de los operadores + y -.

Con estas prioridades una expresión como esta:

```
2 * 3 + 4 * 5
```

se evalúa de la manera siguiente:

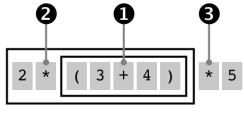


Como el operador * tiene mayor prioridad que el operador + primero se evalúan las multiplicaciones y después se evalúa la suma.

Podemos emplear paréntesis para indicar que la suma debe evaluarse antes que las multiplicaciones. Por ejemplo, si escribimos así la expresión:

```
2 * (3 + 4) * 5
```

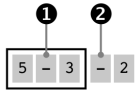
entonces la expresión se evalúa así:



Primero se evalúa la suma y, como el operador * tiene asociatividad de izquierda a derecha, primero se evalúa la multiplicación de la izquierda y después la de la derecha. En este caso, como son multiplicaciones el orden de los factores no afecta el producto y, por lo tanto, la asociatividad no afecta el resultado. Pero en otros casos, por ejemplo con las restas, la asociatividad del operador sí es importante. Por ejemplo, esta expresión:

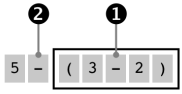
```
5 - 3 - 2
```

se evalúa así:



porque el operador - también tiene asociatividad de izquierda a derecha (casi todos los operadores tienen asociatividad de izquierda a derecha) y el resultado de evaluar esta expresión es cero. Aquí también se pueden emplear paréntesis para modificar el orden en el cual se evalúa la expresión, si escribimos:

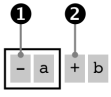
```
5 - (3 - 2)
```
entonces la expresión se evalúa así:



y el resutado de la evaluación es cuatro.

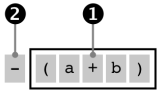
El operador `-` se puede emplear de dos maneras diferentes. Una de ellas es para hacer una resta, por ejemplo: `a - b`. Al emplearlo así se dice que es un operador *binario*. La otra manera es para cambiar el signo de algún valor, por ejemplo `-a`. Al emplearlo de esta segunda manera, se dice que es un operador *unario*. Los operadores unarios siempre tienen más prioridad que los operadores binarios. Es decir que una expresión como esta:

```
-a + b
```
se evalúa así:



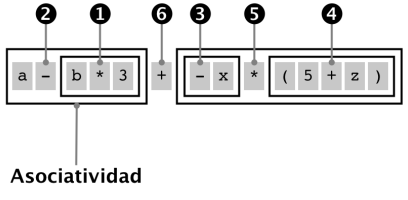
Aquí también podemos emplear paréntesis para cambiar el orden de evaluación. La expresión:

```
-(a + b)
```
se evalúa de esta manera:



Aplicando estas reglas acerca de las prioridades y la asociatividad podemos saber cómo se evalúa cualquier expresión. Por ejemplo, la expresión:

```
a - b * 3 + -x * (5 + z)
```
se evalúa así:



Nota

Los paréntesis además de emplearse para indicar el orden de evaluación de una expresión también se emplean al definir un procedimiento y al llamar un procedimiento, por ejemplo:

```
suma(a, b) {
  return a + b;
}

print(suma(2, 3));
```

De hecho, los paréntesis en `print(suma(2, 3))` son un operador. Es el operador empleado para llamar a un procedimiento.

La tabla siguiente muestra todos los operadores disponibles en simpleJ, ordenados por prioridad de menor a mayor.

Tabla B.1. Operadores

Operador	Asociatividad
=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=	derecha a izquierda
?:	derecha a izquierda
||	izquierda a derecha
&&	izquierda a derecha
|	izquierda a derecha
^	izquierda a derecha
&	izquierda a derecha
==, !=	izquierda a derecha
<, <=, >, >=	izquierda a derecha
<<, >>, >>>	izquierda a derecha
+, - (binario)	izquierda a derecha
*, /, %	izquierda a derecha
- (unario), ~, !, ++, --	
(), [], .	izquierda a derecha

Variables

Una variable representa una localidad de memoria donde se puede almacenar una referencia a un valor. Cuando se ejecuta un programa paso por paso, en la vista de memoria del simpleJ devkit se muestran los valores de tipo nulo, booleano, entero, flotante, string y procedimiento como si estuvieran almacenados dentro de la variable, cuando en realidad lo que se almacena dentro de la variable es una referencia a este valor; esto se debe a que estos valores son inmutables, no se pueden modificar, y desde el punto de vista del funcionamiento del programa el resultado es como si se almacenara el valor dentro de la variable (y hace que el diagrama de la memoria sea más sencillo).

Cada variable tiene un nombre que es único dentro del ambiente al que pertenece. No pueden existir dentro del mismo ambiente dos variables con el mismo nombre. No hay ningún problema con tener dos, o más, variables con el mismo nombre siempre y cuando pertenezcan a ambientes diferentes.

Los nombres de las variables únicamente pueden contener letras, dígitos o "_" (guión bajo). El nombre de la variable debe empezar con una letra y, para los nombres de las variables, el guión bajo se considera una letra. Las letras minúsculas y mayúsculas se consideran como letras diferentes: `vidas`, `Vidas` y `VIDAS` son tres nombres diferentes. No existe ningún límite para la longitud del nombre de una variable.

En simpleJ existen unas palabras *reservadas*. Una palabra reservada no se puede emplear como el nombre de una variable. Las palabras reservadas son:

Tabla B.2. Palabras reservadas

`array`	`else`	`new`	`this`
`break`	`false`	`null`	`true`
`case`	`final`	`return`	`var`
`continue`	`for`	`super`	`while`
`default`	`if`	`switch`	
`do`	`lambda`	`synchronized`	

Ejemplo de algunos nombres válidos para variables:

```
a
x1
```

```
r2d2
c3po
vidas
enemigo_x
jugadorY
cuantosPuntos
_un_nombre_muy_largo_que_empieza_con_guion_bajo
otraManeraDeEscribirUnNombreLargoParaUnaVariable
```

Las variables siempre pertenecen a un ambiente. En cada momento, mientras se ejecuta un programa, hay un *ambiente actual*, que es el ambiente donde se crean las nuevas variables por medio de la palabra reservada `var`. Se pueden crear varias variables simultáneamente separando sus nombres por comas. Opcionalmente, se les puede también asignar un valor inicial al crearlas empleando `=`. Ejemplos:

```
var i;
var x, y, z;
var vidas = 3;
var nombre = "Pedro", edad = 23, puntos = 0;
```

Se puede crear una variable cuyo contenido no es modificable empleando la palabra `final` en vez de `var` al crearla. Al emplear `final` es indispensable emplear `=` para asignarle un valor inicial, es un error de compilación el no hacerlo. También es posible poner varios nombres, con sus valores iniciales, separados por comas. Ejemplos:

```
final ENEMIGOS = 4;
final MAX_X = 31, MAX_Y = 23;
```

Al emplear `final` las reglas para darle un nombre a una variable son los mismos que al emplear `var`, pero se recomienda emplear únicamente mayúsculas para que sea más fácil al leer un programa detectar qué variables no son modificables. Frecuentemente, a las variables declaradas con `final` se les llama *constantes* aunque, estrictamente hablando, son simplemente variables cuyo contenido no se puede modificar (tratar de modificar el contenido de una variable creada con `final` genera un error a la hora de ejecución). Más adelante vamos a ver otro tipo de constantes que también se conocen como *literales*.

Otra manera de crear una variable es definiendo un procedimiento. Por ejemplo, si definimos:

```
suma(a, b) {
  return a + b;
}
```

a la hora de ejecución se crea una variable `suma` que contiene una referencia a un procedimiento. También es posible emplear la palabra `final` para que esa variable no sea modificable:

```
final suma(a, b) {
  return a + b;
}
```

Tipos y constantes

Una variable contiene una referencia a un valor y ese valor tiene un *tipo*. En simpleJ hay tipos básicos y tipos compuestos. Los tipos básicos son los tipos cuyos valores son *inmutables*, no se pueden modificar. Hay cinco tipo básicos que son entero, flotante, string, booleano, procedimiento y nulo. Los tipos compuestos son los tipos cuyos valores son *mutables*, es decir que que se puede modificar su contenido. Hay dos tipos mutables que son arreglos y ambientes.

Nota

Los tipos cola, conjunto y pila (queue, set y stack) son en realidad ambientes que contienen datos y procedimientos.

Cada uno de estos tipos tiene representación en memoria (como la computadora almacena un valor de ese tipo dentro de su memoria) y una, o a veces más, representación en el texto de un programa (como una secuencia de caracteres). A la representación dentro del texto de un programa como una secuencia de caracteres se le llama *literal*.

Enteros

Los enteros se almacenan en memoria como un número de 32 bits empleando complementos a dos para los números negativos.

Un literal entero es una secuencia de dígitos, opcionalmente con un signo `-` (menos) al principio. Ejemplos de literales enteros:

```
2
428
-3
```

Estos literales enteros se interpretan como un número en notación decimal (base diez). También es posible escribir literales enteros en octal (base ocho) y en hexadecimal (base 16). Para escribir un literal en octal se emplea un `0` al principio. Para escribir un literal en hexadecimal se emplea un `0x` ó `0X` al principio. Por lo tanto, el decimal `42` se puede escribir como `052` en octal y como `0x2a` ó `0X2A` en hexadecimal.

También es posible escribir un caracter entre comillas sencillas para representar su valor ASCII. Por ejemplo, escribir `'a'` es lo mismo que escribir `97`. Se puede emplear `'\''` para obtener el valor ASCII de una comilla sencilla y `'\\'` para obtener el valor ASCII de un backslash.

Flotantes

Los flotantes se almacenan en memoria como un número de 64 bits empleando la norma IEEE 754 para números flotantes de doble precisión.

Un literal flotante es una secuencia de dígitos con un solo punto dentro de la secuencia o con un exponente, positivo o negativo, al final. Los flotantes negativos tienen un signo `-` (menos) al principio. Todos estos son literales flotantes:

```
2.3
-12.65
1e3
.25E-10
3.2e+5
```

Strings

Los strings se almacenan en memoria como un `String` de Java.

Un literal string es una secuencia de caracteres entre comillas dobles. Para colocar comillas dobles dentro de un literal string hay que precederlas con un backslash; para colocar un backslash se pone un doble backslash. Ejemplos de literales string:

```
"Hola"
"Este string contiene \"comillas\"."
"Este es un backslash: \\."
```

Booleanos

Los booleanos se almacenan en memoria como un `Boolean` de Java.

Sólo existen dos literales booleanos que son `true` y `false`.

Nulo

Existe un solo literal de tipo nulo que es `null`. Este único valor nulo no ocupa espacio en memoria; al almacenar `null` en una variable, ésta no hace referencia a ningún valor.

Nota

Aunque al almacenar `null` en una variable, ésta no hace referencia a ningún valor esto no es lo mismo que una variable a la cual todavía no se le asigna un valor. Por ejemplo, si tenemos esto en un programa:

```
var a = null;
if (a == null)
  print("a es nulo");
```

al ejecutarlo despliega en el Log el mensaje "**a es nulo**". Pero si el programa está así:

```
var a;
if (a == null)
  print("a es nulo");
```

al ejecutarlo se produce un error de ejecución al intentar ejecutar la línea que dice "`if (a == null)`" porque se intentó accesar el valor de una variable que no ha sido inicializada.

Procedimientos

Un procedimiento se almacena en memoria como una estructura que contiene código del procedimiento traducido a instrucciones para la máquina virtual de simpleJ y una referencia al ambiente donde se definió ese procedimiento.

Un literal de tipo procedimiento se escribe por medio de una *expresión lambda*, aunque existe una notación simplificada para definir procedimientos de una manera más "tradicional". Por ejemplo, esta definición de un procedimiento:

```
suma(a, b) {
  return a + b;
}
```

Es equivalente a esta definición:

```
var suma = lambda(a, b) { return a + b; };
```

donde se ve explícitamente el procedimiento como un literal expresado como expresión lambda que se almacena como el valor de la variable `suma`.

Arreglos

Un arreglo emplea varias direcciones consecutivas de memoria para almacenar referencias a valores. Estas direcciones consecutivas se accesan por medio de subíndices enteros empezando en cero. No es posible modificar el tamaño de un arreglo.

Un literal de tipo arreglo se escribe como una secuencia de valores, separados por comas, entre corchetes (`[` y `]`). Ejemplos:

```
[1, 2, 3]
[2.45, -10, "abc", true, null]
[[2, 4, 8],
 [3, 9, 27]]
```

Un literal de tipo arreglo se compila a instrucciones para la máquina virtual de simpleJ que crean un nuevo arreglo cada vez que se ejecuta. Es decir que si se ejecuta este programa:

```
creaArreglo() {
  return [1, 10, 100, 1000];
}

var a = creaArreglo();
var b = creaArreglo();
```

la variable `a` y la variable `b` van a contener referencias a arreglos con los mismos valores, pero son arreglos diferentes. Cada vez que se ejecuta la línea `return [1, 10, 100, 1000];` se crea un nuevo arreglo.

También se puede crear un arreglo sin darle valores iniciales a sus elementos empleando `new array`. Estos pueden ser arreglos sencillos (arreglos unidi-

mensionales) o arreglos de arreglos (arreglos "multidimensionales"). Por ejemplo, al ejecutar:

```
var a = new array[10];
var b = new array[3][4];
```

La variable `a` queda con una referencia a un arreglo de 10 elementos, mientras que `b` queda con una referencia a un arreglo de tres elementos, cada uno de los cuales contiene a su vez una referencia a un arreglo de cuatro elementos (son tres arreglos de cuatro elementos cada uno).

Ambientes

Un ambiente emplea varias direcciones de memoria para almacenar referencias a valores. Estas direcciones de memoria se accesan por medio de nombres. El tamaño de un ambiente puede modificarse dinámicamente.

Un literal de tipo ambiente se escribe como una secuencia de pares, separados por comas, entre llaves (`{` y `}`). Cada par consiste de un nombre y un valor separados por el caracter dos puntos (`:`). Ejemplos:

```
{nombre: "Juan", edad: 17}
{x: 10, y: 25, vx: -1, vy: 1}
{tipo: "enemigo",
 estado: "atacando",
 ataques: ["bola de fuego", "dardos venenosos"]}
```

Un literal de ambiente se compila a instrucciones para la máquina virtual de simpleJ que crean un nuevo ambiente cada vez que se ejecuta. Es decir que si se ejecuta este programa:

```
creaAmbiente() {
  return {x: 10, y: 20};
}

var a = creaAmbiente();
var b = creaAmbiente();
```

la variable `a` y la variable `b` van a contener referencias a ambientes con los mismos valores, pero son ambientes diferentes. Cada vez que se ejecuta la línea `return {x: 10, y: 2);` se crea un nuevo ambiente.

Operadores

Los operadores se emplean para indicar las operaciones a realizar para evaluar el valor de una expresión. En simpleJ existen siete tipos de operadores que son: los operadores aritméticos, los operadores relacionales, los operadores de igualdad, los operadores lógicos, los operadores de manipulación de bits, los operadores de asignación y el operador condicional.

Operadores aritméticos

Los operadores aritméticos son: `+` (suma), `-` (como operador binario es resta; como operador unario es cambio de signo), `*` (multiplicación), `/` (división), `%` (módulo, también conocido como residuo), `++` (incremento de uno), `--` (decremento de uno).

Una operación aritmética con dos enteros da un resultado entero. Si uno de los dos es flotante entonces el otro se convierte a flotante y se hace la operación con flotantes. Por lo tanto, el resultado de evaluar `7 / 2` es `3`, pero al evaluar `7.0 / 2` el resultado es `3.5`. Se genera un error de ejecución si alguno de los operandos no es un número.

Nota

El operador `+` también se emplea para la concatenación de strings. Al aplicar este operador primero checa si alguno de los dos operandos es un string, en caso de serlo entonces convierte el otro operando a un string y los concatena. Si ninguno de los dos operandos es un string entonces aplica las mismas reglas que para los demás operadores aritméticos.

La regla es diferente para el operador `%` (módulo) que siempre trunca sus operandos a valores enteros, por lo tanto el resultado de `12.7 % 5.3` es `2`.

Los operadores unarios `++` (incremento) y `--` (decremento) se pueden emplear como prefijo (antes del operando) o como postfijo (después del operando). El operando debe ser una variable o un elemento de un arreglo.

Usarlos como prefijo indica que la variable debe ser incrementada (o decrementada) antes de emplear su valor en la expresión; mientras que usarlos como postfijo indica que la variable debe ser incrementada (o decrementada) después

de usar su valor. El contenido de la variable debe ser un número (entero o flotante).

Operadores relacionales

Los operadores relacionales son: < (menor), <= (menor o igual), > (mayor) y >= (mayor o igual).

Se pueden emplear para comparar dos números o dos strings. El resultado es un booleano (`true` o `false`). En el caso de que sean strings el elemento menor es el primero al ordenarlos en orden "alfabético".

Nota

Para la computadora todas las letras mayúsculas vienen antes que todas la minúsculas. Es decir que la `Z` viene antes que la `a`.

Comparar un número con un string o emplear un operando cuyo valor no sea un número o string resulta en un error de ejecución.

Operadores de igualdad

Los operadores de igualdad son == (igual) y != (diferente).

Se pueden emplear para comparar valores de cualquier tipo. El resultado es un booleano (`true` o `false`). Un entero es igual a un flotante si representan el mismo valor. Dos strings son iguales si contienen exactamente la misma secuencia de caracteres. Dos valores booleanos son iguales únicamente si los dos son `true` o los dos son `false`. Valores de otros tipos (nulo, procedimiento, arreglo y ambiente) únicamente son iguales a sí mismos. Objetos de diferentes tipos, excepto enteros y flotantes, son siempre diferentes.

Operadores lógicos

Los operadores lógicos son && (y), || (o) y ! (no); todos ellos esperan que sus operandos sean booleanos y su resultado también es un valor booleano.

Expresiones conectadas con el operador lógico && tienen como resultado `true` únicamente si todas las expresiones se evalúan como `true`. La evaluación procede de izquierda a derecha y se detiene tan pronto se encuentra un `false`.

Expresiones conectadas con el operador lógico `||` tienen como resultado `true` si cualquiera de las expresiones se evalúa como `true`. La evaluación procede de izquierda a derecha y se detiene tan pronto se encuentra un `true`.

El operador unario de negación `!` convierte un `true` en un `false` y un `false` en un `true`.

Operadores de manipulación de bits

Los operadores de manipulación de bits son & (y), | (o), ^ (o exclusivo), << (recorrer a la izquierda), >> (recorrer a la derecha), >>> (recorrer a la derecha ignorando el signo) y ~ (negación).

Todos los operadores de manipulación de bits esperan que sus operandos sean números. Los flotantes se convierten a enteros antes de aplicar el operador. El resultado siempre es un entero. Aplicar uno de estos operadores a algo que no sea un número causa un error de ejecución.

El operador & hace un *and* (y) entre los bits correspondientes de sus operandos.

El operador | hace un *or* (o) entre los bits correspondientes de sus operandos.

El operador ^ hace un *xor* (o exclusivo) entre los bits correspondientes de sus operandos.

El operador << recorre hacia la izquierda los bits de su primer operando del número de posiciones indicado por su segundo operando.

El operador >> recorre hacia la derecha los bits de su primer operando del número de posiciones indicado por su segundo operando. El bit más significativo, el bit de signo, se propaga hacia la derecha.

El operador >>> recorre hacia la derecha los bits de su primer operando del número de posiciones indicado por su segundo operando. El bit más significativo es reemplazado por un cero.

El operador unario ~ hace un *not* (no) de cada uno de los bits de su operando (obtiene su complemento a unos).

Operadores de asignación

Los operadores de asignación son `=`, `+=`, `-=`, `*-`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=` y `>>>=`.

En simpleJ una asignación es una expresión; su valor es el valor almacenado en la variable que tiene del lado izquierdo del operador. Los operadores de asignación tienen asociatividad de derecha a izquierda; esto se puede emplear para asignar el mismo valor a varias variables. Por ejemplo, la expresión:

```
a = b = c = 0;
```

tiene el mismo significado que:

```
a = (b = (c = 0));
```

con lo cual se asigna un cero a las variables `a`, `b` y `c`.

Además del operador estandar de asignación (el `=`) en simpleJ también hay operadores que combinan la asignación con otra operación binaria. Estos operadores son de la forma *op*`=`, donde *op* es uno de: `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>` o `>>>`. El significado de algo como:

```
a += 2;
```

es equivalente a:

```
a = a + 2;
```

La única diferencia es que la expresión del lado izquierdo únicamente se evalúa una vez. Es decir que esto:

```
botones[readCtrlOne()] += 1;
```

no es lo mismo que:

```
botones[readCtrlOne()] = botones[readCtrlOne()] + 1;
```

porque en el segundo caso se llama dos veces a la función `readCtrlOne` en vez de una sola.

Operador condicional

En simpleJ hay un operador condicional ternario (con tres operandos) que es `?`.

Se emplea de la manera siguiente:

```
pregunta ? valor1 : valor2;
```

Y su significado es: primero evalúa `pregunta`, si su resultado es `true`, entonces el resultado de la expresión es `valor1`, de lo contrario es `valor2`. Si el valor de `pregunta` no es un booleano, entonces se produce un error de ejecución.

Apéndice C. Procedimientos predefinidos

Aquí hay una breve descripción de cada uno de los procedimientos predefinidos. Para cada uno de ellos se explica qué hace, qué argumentos espera y, en caso de ser una función, qué resultado devuelve. En el caso de procedimientos que esperen como argumento un número entero es posible pasarles un flotante y el procedimiento automáticamente toma únicamente la parte entera de ese número. Para los procedimientos en donde se espera como argumento un número flotante y se le pasa un entero el procedimiento simplemente lo interpreta como el valor flotante correspondiente. En cualquier otro caso es un error pasar un argumento de un tipo que no corresponde al esperado por el procedimiento.

Matemáticos

Los procedimientos matemáticos proporcionan las funciones matemáticas básicas y la generación de números aleatorios.

`acos(a)`

Función que devuelve el arco coseno de un ángulo, en radianes, en el rango de 0.0 a PI.

Argumentos:

1. `a` - flotante: El valor del cual se quiere obtener el arco coseno.

Devuelve:

Un flotante con arco coseno del argumento.

`asin(a)`

Función que devuelve el arco seno de un ángulo, en radianes, en el rango de -PI/2 a PI/2.

Argumentos:

1. a - flotante: El valor del cual se quiere obtener el arco seno.

Devuelve:

Un flotante con el arco seno del argumento.

atan(a)

Función que devuelve el arco tangente de un ángulo, en radianes, en el rango de -PI/2 a PI/2.

Argumentos:

1. a - flotante: El valor del cual se quiere obtener al arco tangente.

Devuelve:

Un flotante con el arco tangente del argumento.

atan2(y, x)

Función que convierte de coordenadas rectangulares (x, y) a polares (r, theta). Esta función calcula el ángulo theta evaluando el arco tangente de y/x, en radianes, en el rango de -PI a PI.

Argumentos:

1. y - flotante: Ordenada de las coordenadas.

2. x - flotante: Abscisa de las coordenadas.

Devuelve:

Un flotante con el componente theta del punto (r, theta) en coordenadas polares que corresponde al punto (x, y) en coordenadas polares.

ceil(a)

Función que devuelve el flotante más pequeño (más cercano al infinito negativo) que no es menor que el argumento y es igual a un entero matemático.

Argumentos:

1. a - flotante: Un valor.

Devuelve:

El más pequeño flotante (más cercano al infinito negativo) que no es menor que el valor y es igual a un entero matemático.

cos(a)

Función que devuelve el coseno de un ángulo.

Argumentos:

1. a - flotante: El ángulo en radianes.

Devuelve:

Un flotante con el coseno del ángulo.

exp(a)

Función que devuelve *e* (el número de Euler) elevado a una potencia.

Argumentos:

1. a - flotante: El exponente al que hay que elevar *e*.

Devuelve:

Un flotante con el valor de e^a, donde *e* es la base de los logaritmos naturales.

floor(a)

Función que devuelve el flotante más grande (más cercano al infinito positivo) que no es mayor que el argumento y es igual a un entero matemático.

Argumentos:

1. a - flotante: Un valor.

Devuelve:

El más grande flotante (más cercano al infinito positivo) que no es mayor que el argumento y es igual a un entero matemático.

`frandom()`

Función que devuelve un número aleatorio con distribución uniforme entre 0.0 y 1.0.

Devuelve:

Un flotante aleatorio entre 0.0 y 1.0.

`log(a)`

Función que devuelve el logaritmo (base *e*) de un valor.

Argumentos:

1. `a` - flotante: Un número mayor que 0.0.

Devuelve:

Un flotante con el logaritmo natural de `a`.

`pow(a, b)`

Función que devuelve el valor del primer argumento elevado a la potencia del segundo argumento.

Argumentos:

1. `a` - flotante: La base.
2. `b` - flotante: El exponente.

Devuelve:

Un flotante con el valor a^b.

`random(n)`

Función que devuelve un número entero aleatorio, con distribución uniforme, entre 0 y n-1.

Argumentos:

1. `n` - entero: Un número entero mayor que 0.

Devuelve:

Un entero aleatorio, con distribución uniforme, entre 0 y n-1.

`round(a)`

Función que devuelve el entero más cercano al argumento. Se calcula sumándole 1/2 al argumento y obteniendo el piso (`floor`) del resultado.

Argumentos:

1. `a` - flotante: El valor a redondear.

Devuelve:

El flotante con el valor del argumento redondeado al entero más cercano.

`sin(a)`

Función que devuelve el seno de un ángulo.

Argumentos:

1. `a` - flotante: El ángulo en radianes.

Devuelve:

Un flotante con el seno del ángulo.

`sqrt(a)`

Función que devuelve la raíz cuadrada positiva de un valor.

Argumentos:

1. `a` - flotante: Un valor positivo.

Devuelve:

Un flotante con la raíz cuadrada positiva del valor.

`tan(a)`

Función que devuelve la tangente de un ángulo.

Argumentos:

1. `a` - flotante: El ángulo en radianes.

Devuelve:

Un flotante con la tangente del ángulo.

Manejo de strings

Procedimientos y funciones para manipular strings.

`atof(s)`

Función que traduce la representación de un número como un string en un valor flotante. Es un error si el string no contiene la representación de un número flotante.

Argumentos:

1. `s` - string: El string que representa un número flotante.

Devuelve:

El número flotante representado por el string.

`atoi(s)`

Función que traduce la representación de un número como un string en un valor entero. Es un error si el string no contiene la representación de un número entero.

Argumentos:

1. `s` - string: El string que representa un número entero.

Devuelve:

El número entero representado por el string.

`appendChar(s, n)`

Función que concatena un caracter unicode al final de un string.

Argumentos:

1. `s` - string: Un string.
2. `n` - entero: El valor Unicode del caracter.

Devuelve:

Un nuevo string que corresponde al string pasado como primer argumento con el caracter Unicode que corresponde al segundo argumento concatenado al final.

`charAt(s, n)`

Función que devuelve el código Unicode del caracter que se encuentra en una posición de un string.

Argumentos:

1. `s` - string: El string del cual se desea extraer un caracter.
2. `n` - entero: La posición del caracter dentro del string. Si el string es de longitud L entonces debe ser un número entre 0 y L-1.

Devuelve:

Un entero con el código Unicode del caracter en esa posición del string.

`length(s)`

Función que devuelve la longitud (número de caracteres) de un string.

Argumentos:

1. `s` - string: Un string.

Devuelve:

Un entero con la longitud del string.

Control del IAVC (Integrated Audio and Video Controller)

El IAVC es el chip (simulado en software) que se encarga de manejar el audio, video, controles y tarjeta de memoria en simpleJ. Estos son los procedimientos que permiten controlar el IAVC.

Los programas controlan el IAVC por medio de 32768 direcciones de memoria de 8 bits (un byte) cada una. Algunas de estas direcciones corresponden a las memorias en donde el IAVC almacena la imágenes de los tiles, los sprites, los tiles de la pantalla y las formas de onda para los canales de audio. Las otras direcciones son para accesar los registros de control del IAVC. Los procedimientos `poke`, `pokew`, `arrayPoke` y la función `peek` son los que permiten accesar directamente estas direcciones de memoria. Los demás procedimientos y funciones ofrecen una manera más sencilla de manipular el IAVC, sin que sea necesario conocer todos los detalles de estas direcciones de memoria.

`arrayPoke(addr, data, offset, count)`

Procedimiento que copia información de un arreglo de enteros a grupo de direcciones consecutivas de memoria del IAVC.

Argumentos:

1. `addr` - entero: La dirección inicial de memoria a donde se deben copiar los datos.

2. `data` - arreglo de enteros: De donde se deben copiar los datos. Debe contener números entre 0 y 255, para números fuera de este rango únicamente se toma el byte menos significativo y se descarta el resto.

3. `offset` - entero: El subíndice del arreglo donde se encuentra el primer dato que se debe copiar al IAVC.

4. `count` - entero: Cuántos elementos del arreglo se deben copiar al IAVC.

`clear()`

Procedimiento que borra la pantalla. En realidad rellena los 24 renglones de 32 tiles cada uno que se encuentran en la parte superior izquierda del área de

memoria de video del IAVC con el tile número 32 (cuya imagen, si no se ha redefinido, es un espacio en blanco). Para mayor información, ver la documentación del procedimiento `setScreenOffset`.

isButtonDown(buttons, mask)

Función que permite detectar si un botón se encuentra presionado aún cuando el jugador presione varios botones del control simultáneamente.

Ejemplo de uso:

```
final BOTON_ARRIBA 1
var botones = readCtrlOne();
if (isButtonDown(botones, BOTON_ARRIBA)) {
```

Argumentos:

1. `buttons` - entero: Un número con un bit prendido por cada botón que se encuentre apoyado en el control. Típicamente es el resultado de llamar la función `readCtrlOne` o la función `readCtrlTwo`.

2. `mask` - entero: La máscara que indica cuáles botones se desean checar.

Devuelve:

`true` si el botón (o alguno de los botones) indicado se encuentra presionado, de lo contrario devuelve `false`.

memCardLoad()

Función que devuelve los 512 bytes almacenados en la tarjeta de memoria para este programa.

Devuelve:

Un arreglo de 512 enteros. Cada uno de estos enteros está dentro del rango de 0 a 255. Si no hay información almacenada para este programa en la tarjeta de memoria entonces devuelve un arreglo con puros ceros.

memCardSave(data)

Procedimiento que almacena datos en el área de la tarjeta de memoria asignada a este programa.

Argumentos:

1. `data` - arreglo de enteros: Un arreglo con 512 enteros. Debe contener números entre 0 y 255, para números fuera de este rango únicamente se toma el byte menos significativo y se descarta el resto.

`note(nt)`

Procedimiento que genera un sonido correspondiente a una nota de música. Emplea el canal cero de audio para generar este sonido. Se puede modificar el timbre de este sonido empleando los procedimientos `setSoundAttack`, `setSoundDecay`. Se puede modificar su volumen con el procedimiento `setSoundVolume`. También es posible emplear los procedimientos `setSoundSustain` y `setSoundRelease` para modificar su timbre, pero en este caso es necesario emplear el procedimiento `soundOff` para terminar la generación de la nota.

Argumentos:

1. `note` - string: La nota de música que se debe generar.

   El string debe contener una letra (mayúscula) y un número. La letra le indica cuál es la nota y el número le indica cuál es la octava. Entre más pequeño sea el número, más baja es la octava (sonido más grave); entre más grande, más alta es la octava (sonido más agudo). La octava debe ser un número entero entre 1 y 6. La siguiente tabla indica la correspondencia entre las letras y las notas musicales:

Tabla C.1. Letras y Notas

Letra	Nota
C	Do
D	Re
E	Mi
F	Fa
G	Sol
A	La
B	Si

Se puede emplear un "#" o una "b" entre la letra de la nota y el número de la octava para indicar *sostenido* o *bemol* respectivamente. Es decir que "C#4" representa un *do sostenido* y "Eb4" es un *mi bemol*.

La nota más grave es la "A1" y la más aguda es la "G#6".

peek(addr)

Función para leer una de las direcciones de memoria del IAVC.

Argumentos:

1. addr - entero: La dirección de memoria que se debe accesar, es un número entre 0 y 32767. Si se encuentra fuera de este rango, entonces únicamente se emplean los 15 bits menos significativos y se descarta el resto.

Devuelve:

Un número entero entre 0 y 255 que corresponde al dato devuelvedo por el IAVC al accesar esa dirección de memoria.

poke(addr, b)

Procedimiento que almacena un byte (8 bits) en una de las direcciones de memoria del IAVC.

Argumentos:

1. addr - entero: La dirección de memoria en la cual se debe almacenar el byte, es un número entre 0 y 32767. Si se encuentra fuera de este rango, entonces únicamente se emplean los 15 bits menos significativos y se descarta el resto.

2. b - entero: Un número entre 0 y 255 que se debe almacenar en la dirección de memoria del IAVC indicada por addr.

pokew(addr, w)

Procedimiento que almacena un valor de 2 bytes (16 bits) en dos direcciones consecutivas de la memoria del IAVC.

Argumentos:

1. `addr` - entero: La dirección de memoria en la cual se debe almacenar el byte más significativo de `w`, el byte menos significativo se almacena en la dirección siguiente (`addr + 1`), es un número entre 0 y 32767. Si se encuentra fuera de este rango, entonces únicamente se emplean los 15 bits menos significativos y se descarta el resto.

2. `w` - entero: Un número entre 0 y 65535. Dos bytes que se deben almacenar en las direcciones de memoria del IAVC `addr` y `addr + 1`.

putAt(tileIndex, x, y)

Procedimiento que coloca un tile en el área de memoria de video del IAVC. Esta área de memoria está organizada en 32 renglones de 64 tiles cada uno, de los cuales únicamente es visible en la pantalla un área de 24 renglones de 32 tiles cada uno. Los parámetros x y y de este procedimiento son independientes del área que esté visible en ese momento en la pantalla. Para mayor información, ver la documentación del procedimiento `setScreenOffset`.

Argumentos:

1. `tileIndex` - entero: El tile que se debe colocar. Es un número entre 0 y 255.

2. `x` - entero: La posición horizontal del tile. Es un número entre 0 y 63.

3. `y` - entero: La posición vertical del tile. Es un número entre 0 y 31.

putSpriteAt(spriteIndex, x, y)

Procedimiento que mueve la posición de un sprite en la pantalla de manera que su esquina superior izquierda quede en la posición (`x`, `y`).

Argumentos:

1. `spriteIndex` - entero: El sprite que se debe mover, es un número entre 0 y 31.

2. `x` - entero: La coordenada horizontal, en pixeles, donde debe quedar la esquina superior izquierda del sprite. Es un número entre -16 y 256.

3. `y` - entero: La coordenada vertical, en pixeles, donde debe quedar la esquina superior derecha del sprite. Es un número entre -16 y 192.

`readCtrlOne()`

Función que devuelve un entero entre 0 y 255 indicando los botones que están presionados en ese instante en el control número uno.

A cada bit (botón apoyado) corresponde un valor que es una potencia de dos. Si dos o más botones están apoyados, entonces el valor es la suma de los valores que corresponden a cada botón.

Tabla C.2. Números para cada botón

Valor	Botón
1	**flecha arriba**
2	**flecha abajo**
4	**flecha izquierda**
8	**flecha derecha**
16	**enter (o return)**
32	**control**
64	**barra espaciadora**
128	**P**

Devuelve:

Un entero entre 0 y 255. Este entero corresponde a un byte en donde cada uno de sus ocho bits está prendido si el botón correspondiente está presionado en ese instante.

`readCtrlTwo()`

Función que devuelve un entero entre 0 y 255 indicando los botones que están presionados en ese instante en el control número dos.

A cada bit (botón apoyado) corresponde un valor que es una potencia de dos. Si dos o más botones están apoyados, entonces el valor es la suma de los valores que corresponden a cada botón.

Tabla C.3. Números para cada botón

Valor	Botón
1	**R**
2	**F**
4	**D**
8	**G**
16	**shift**
32	**Z**
64	**X**
128	**Q**

Devuelve:

Un entero entre 0 y 255. Este entero corresponde a un byte en donde cada uno de sus ocho bits está prendido si el botón correspondiente está presionado en ese instante.

`setBackground(red, green, blue)`

Procedimiento para cambiar el color del fondo. En realidad cambia el registro de color número cero para los tiles y, por lo tanto, es equivalente a `setTileColor(0, red, green, blue)`. Para mayor información, ver la documentación del procedimiento `setTileColor`.

Argumentos:

1. `red` - entero: El componente rojo del color. Es un número entre 0 y 31.
2. `green` - entero: El componente verde del color. Es un número entre 0 y 31.
3. `blue` - entero: El componente azul del color. Es un número entre 0 y 31.

`setForeground(red, green, blue)`

Procedimiento para cambiar el color de las letras. En realidad cambiar el registro de color número uno para los tiles y, por lo tanto, es equivalente a `setTile-`

`Color(1, red, green, blue)`. Para mayor información, ver la documentación del procedimiento `setTileColor`.

Argumentos:

1. `red` - entero: El componente rojo del color. Es un número entre 0 y 31.
2. `green` - entero: El componente verde del color. Es un número entre 0 y 31.
3. `blue` - entero: El componente azul del color. Es un número entre 0 y 31.

`setLargeSpriteImage(spriteIndex, imageIndex)`

Procedimiento que asigna una de las 128 imágenes grandes (de 16 por 16 pixeles) a un sprite.

Argumentos:

1. `spriteIndex` - entero: Número entre 0 y 31 para indicar a cuál sprite se le debe asignar la imagen.
2. `imageIndex` - entero: Número entre 0 y 127 para indicar cuál de las 128 imágenes grandes se le debe asignar al sprite.

`setLargeSpritePixels(imageIndex, pixels)`

Procedimiento para modificar los pixeles de una imagen grande (16 por 16 pixeles) de sprite en la memoria del IAVC.

Argumentos:

1. `imageIndex` - entero: Número entre 0 y 127 para indicar a cuál de las 128 imágenes grandes se le deben redefinir sus pixeles.
2. `pixels` - arreglo de enteros: Arreglo con 256 enteros, cada uno debe ser un valor entre 0 y 15. Los valores entre 0 y 14 se emplean para seleccionar el registro de color a emplear para ese pixel. Un 15 indica un pixel transparente. Cada grupo consecutivo de 16 valores corresponde a un renglón de pixeles de la imagen, empezando por el renglón superior.

setScreenOffset(x, y)

Procedimiento para indicar el área de la memoria de video que se debe mostrar en la pantalla. La memoria de video está organizada en 32 renglones de 64 tiles cada uno, en la pantalla se muestra una ventana que cubre sólo una parte de la memoria de video. Esta área es de 24 renglones de 32 tiles cada uno. Con este procedimiento se puede mover esta ventana para ir mostrando diferentes partes de la memoria de video, un uso posible es para ir moviendo el fondo cuando el personaje recorre un mundo más grande que la pantalla.

Al emplear este procedimiento la imagen del fondo se puede desplazar únicamente en incrementos de 8 pixeles. Para lograr movimientos más continuos (de 1 en 1 pixel) es necesario combinar este procedimiento con el procedimiento `setSmoothScroll`.

Argumentos:

1. `x` - entero: Un número entre 0 y 32. Indica cuánto hay que desplazar horizontalmente la ventana sobre el área de video. La unidad de desplazamiento es un tile.

2. `y` - entero: Un número entre 0 y 8. Indica cuánto hay que desplazar verticalmente la ventana sobre el área de video. La unidad de desplazamiento es un tile.

setSmallSpriteImage(spriteIndex, imageIndex)

Procedimiento que asigna una de las 128 imágenes pequeñas (de 8 por 8 pixeles) a un sprite.

Argumentos:

1. `spriteIndex` - entero: Número entre 0 y 31 para indicar a cuál sprite se le debe asignar la imagen.

2. `imageIndex` - entero: Número entre 0 y 127 para indicar cuál de las 128 imágenes pequeñas se le debe asignar al sprite.

setSmallSpritePixels(imageIndex, pixels)

Procedimiento para modificar los pixeles de una imagen pequeña (8 por 8 pixeles) de sprite en la memoria del IAVC.

Argumentos:

1. `imageIndex` - entero: Número entre 0 y 127 para indicar a cuál de las 128 imágenes pequeñas se le deben redefinir sus pixeles.

2. `pixels` - arreglo de enteros: Arreglo con 64 enteros, cada uno debe ser un valor entre 0 y 15. Los valores entre 0 y 14 se emplean para seleccionar el registro de color a emplear para ese pixel. Un 15 indica un pixel transparente. Cada grupo consecutivo de 8 valores corresponde a un renglón de pixeles de la imagen, empezando por el renglón superior.

`setSmoothScroll(x, y)`

Procedimiento para desplazar la ventana sobre el área de video en incrementos de un solo pixel. Únicamente puede emplearse para hacer desplazamientos hasta de 16 pixeles. Para lograr desplazamientos más grandes hay que combinarlo con el procedimiento `setScreenOffset`.

Argumentos:

1. `x` - entero: Un número entre 0 y 15. Indica cuánto hay que desplazar horizontalmente la ventana sobre el área de video. La unidad de desplazamiento es un pixel.

2. `y` - entero: Un número entre 0 y 15. Indica cuánto hay que desplazar verticalmente la ventana sobre el área de video. La unidad de desplazamiento es un pixel.

`setSoundAttack(channel, time)`

Procedimiento para fijar el tiempo de attack en un canal de audio. El tiempo de attack es el tiempo que toma un sonido desde que se empieza a generar hasta que llega a su volumen máximo.

Argumentos:

1. `channel` - entero: El canal de audio. Un número entre 0 y 3.

2. `time` - entero: El tiempo de attack en milisegundos. Un número entre 0 y 65535.

`setSoundDecay(channel, time)`

Procedimiento para fijar el tiempo de decay en un canal de audio. El tiempo de decay es el tiempo que toma un sonido desde que llega a su volumen máximo hasta que baja a su volumen de sustain.

Argumentos:

1. `channel` - entero: El canal de audio. Un número entre 0 y 3.
2. `time` - entero: El tiempo de decay en milisegundos. Un número entre 0 y 65535.

`setSoundFrequency(channel, frequency)`

Procedimiento para fijar la frecuencia de un canal de audio.

Argumentos:

1. `channel` - entero: El canal de audio. Un número entre 0 y 3.
2. `frequency` - entero: La frecuencia. Un número entre 0 y 65535. Este número no está en Hertz; para pasar de Hertz al valor que hay que pasar como parámetro, hay que multiplicar la frecuencia en Hertz por 23.77723356 y redondear el resultado.

`setSoundRelease(channel, time)`

Procedimiento para fijar el tiempo de release de un canal de audio. El tiempo de release es el tiempo que toma un sonido desde que se ordena detener la generación del sonido por medio del procedimiento `soundOff` hasta que su volumen llega a cero.

Argumentos:

1. `channel` - entero: El canal de audio. Un número entre 0 y 3.
2. `time` - entero: El tiempo de release en milisegundos. Un número entre 0 y 65535.

`setSoundSustain(channel, sustain)`

Procedimiento para fijar el volumen de sustain de un canal de audio como un porcentaje de su volumen máximo.

Argumentos:

1. `channel` - entero: El canal de audio. Un número entre 0 y 3.

2. `sustain` - entero: El volumen de sustain como un porcentaje del volumen máximo. Un número entre 0 y 3 con el siguiente significado:

   0: 0% del volumen máximo
   1: 33% del volumen máximo
   2: 67% del volumen máximo
   3: 100% del volumen máximo

`setSoundVolume(channel, volume)`

Procedimiento para fijar el volumen máximo de un canal de audio.

Argumentos:

1. `channel` - entero: El canal de audio. Un número entre 0 y 3.

2. `volume` - entero: El volumen máximo. Un número entre 0 y 15.

`setSoundWave(channel, waveform)`

Procedimiento para redefinir la forma de onda para un canal de audio. Al iniciar la ejecución de un programa, los cuatro canales de audio tienen una onda cuadrada.

Argumentos:

1. `channel` - entero: El canal de audio. Un número entre 0 y 3.

2. `waveform` - arreglo de enteros: Un arreglo de 256 enteros con valores -128 y 127.

`setSpriteColor(index, red, green, blue)`

Procedimiento para almacenar los componentes rojo, verde y azul en uno de los 15 registros de color para sprites.

Argumentos:

1. `index` - entero: El registro de color. Es un número entre 0 y 14.
2. `red` - entero: El componente rojo del color. Es un número entre 0 y 31.
3. `green` - entero: El componente verde del color. Es un número entre 0 y 31.
4. `blue` - entero: El componente azul del color. Es un número entre 0 y 31.

`setTileColor(index, red, green, blue)`

Procedimiento para almacenar los componentes rojo, verde y azul en uno de los 16 registros de color para tiles.

Argumentos:

1. `index` - entero: El registro de color. Un número entre 0 y 15.
2. `red` - entero: El componente rojo del color. Es un número entre 0 y 31.
3. `green` - entero: El componente verde del color. Es un número entre 0 y 31.
4. `blue` - entero: El componente azul del color. Es un número entre 0 y 31.

`setTilePixels(index, pixels)`

Procedimiento para modificar los pixeles de una imagen de tile en la memoria del IAVC.

Argumentos:

1. `imageIndex` - entero: Número entre 0 y 255 para indicar a cuál de las 256 imágenes se le deben redefinir sus pixeles.

2. `pixels` - arreglo de enteros: Arreglo con 64 enteros, cada uno debe ser un valor entre 0 y 15. Estos valores se emplean para seleccionar el registro de color a emplear para ese pixel. Cada grupo consecutivo de 8 valores corresponde a un renglón de pixeles de la imagen, empezando por el renglón superior.

`showAt(msg, x, y)`

Procedimiento que coloca una secuencia de tiles en el área de memoria de video del IAVC para desplegar un mensaje. Esta área de memoria está organizada en 32 renglones de 64 tiles cada uno, de los cuales únicamente es visible en la pantalla un área de 24 renglones de 32 tiles cada uno. Los parámetros x y y de este procedimiento son independientes del área que esté visible en ese momento en la pantalla. Para mayor información ver la documentación del procedimiento `setScreenOffset`.

Argumentos:

1. `msg` - número o string: El mensaje a desplegar

2. `x` - entero: La posición horizontal donde debe desplegarse el mensaje. Es un número entre 0 y 63.

3. `y` - entero: La posición vertical del tile. Es un número entre 0 y 31.

`soundOff(channel)`

Procedimiento para terminar la generación de un sonido en uno de los canales de audio. Al ejecutarlo se inicia el release.

Argumentos:

1. `channel` - entero: El canal de audio. Un número entre 0 y 3.

`soundOn(channel)`

Procedimiento para iniciar la generación de un sonido en uno de los canales de audio. Al ejecutarlo se inicia el attack.

Argumentos:

1. `channel` - entero: El canal de audio. Un número entre 0 y 3.

Archivos

Estos procedimientos permiten leer archivos de datos. También hay aquí procedimientos para ejecutar instrucciones contenidas en un archivo.

`readFile(filename)`

Función para leer los datos de un archivo.

Argumentos:

1. `filename` - string: El nombre del archivo.

Devuelve:

Un arreglo de enteros. La longitud del arreglo es igual al tamaño del archivo. Cada elemento del arreglo contiene un valor entre 0 y 255 que representa el byte correspondiente dentro del archivo.

`readSpritesFile(filename)`

Función para leer los datos de un archivo creado con el simpleJ sprites editor.

Devuelve un ambiente con tres variables:

1. `colors`: Contiene una referencia a un arreglo con 16 elementos, uno para cada uno de los 15 registros de color más un color que se emplea en el sprites editor para representar los pixeles transparentes (ese último color normalmente no se emplea dentro de un programa pero de todas maneras lo devuelve la función `readSpritesFile` porque es parte del archivo). Cada uno de estos 16 elementos contiene una referencia a un ambiente; cada uno de estos ambientes contiene tres variables: `red`, `green`, y `blue`; estas variables contienen un número entre 0 y 31 para indicar el valor de los componentes rojo, verde y azul para ese registro de color.

2. `largePixels`: Contiene una referencia a un arreglo con 128 elementos, un elemento para cada una de las imágenes de 16 por 16 pixeles. Cada uno de esos elementos, es a su vez una referencia a un arreglo con 256 números que indican el color a emplear para cada uno de los 16 por 16 pixeles que tiene cada imagen grande para los sprites.

3. `smallPixels`: Contiene una referencia a un arreglo con 128 elementos, un elemento para cada una de las imágenes de 8 por 8 pixeles. Cada uno de esos elementos, es a su vez una referencia a un arreglo con 64 números que indican el color a emplear para cada uno de los 8 por 8 pixeles que tiene cada imagen pequeña para los sprites.

`readTilesFile(filename)`

Función para leer los datos de un archivo creado con el simpleJ tiles editor.

Devuelve un ambiente con tres variables:

1. `colors`: Contiene una referencia a un arreglo con 16 elementos, uno para cada registro de color. Cada uno de estos 16 elementos contiene una referencia a un ambiente; cada uno de estos ambientes contiene tres variables: `red`, `green`, y `blue`; estas variables contienen un número entre 0 y 31 para indicar el valor de los componentes rojo, verde y azul para ese registro de color.

2. `pixels`: Contiene una referencia a un arreglo con 256 elementos, un elemento para cada tile modificable. Cada uno de esos elementos es a su vez una referencia a un arreglo con 64 números que indican el color a emplear para cada uno de los 8 por 8 pixeles que tiene la imagen de cada tile.

3. `rows`: Contiene un arreglo con 24 elementos, uno por cada renglón de la pantalla, y cada uno de esos elementos es a su vez un arreglo con 32 números, uno por cada columna de la pantalla.

`source(filename)`

Procedimiento para leer, compilar y ejecutar las instrucciones contenidas en un archivo. Las instrucciones se ejecutan dentro del contexto del ambiente actual al ejecutar el llamado al procedimiento `source`. Las variables y procedimientos que se definan dentro de ese archivo quedan dentro del ambiente actual, en el que fue llamado el procedimiento `source`.

Argumentos:

1. `filename` - string: El nombre del archivo.

`source(filename, env)`

Procedimiento para leer, compilar y ejecutar las instrucciones contenidas en un archivo. Las instrucciones se ejecutan dentro del contexto del ambiente `env`. Las variables y procedimientos que se definan dentro de ese archivo quedan dentro del ambiente indicado por el argumento `env`.

Argumentos:

1. `filename` - string: El nombre del archivo.
2. `env` - ambiente: El ambiente donde deben quedar los procedimientos y variables definidos dentro del archivo.

Arreglos

Procedimientos y funciones para manipular arreglos.

`arrayCopy(src, srcOffset, dst, dstOffset, count)`

Procedimiento para copiar elementos de un arreglo a otro. Copia del arreglo `src` los elementos desde `srcOffset` hasta `srcOffset + count - 1` al arreglo `dst` en las posiciones que van de `dstOffset` hasta `dstOffset + count - 1`.

Argumentos:

1. `src` - arreglo: Arreglo de donde se van a copiar unos elementos.
2. `srcOffset` - entero: Subíndice dentro del arreglo src a partir del cual se van a tomar los elementos para copiarlos.
3. `dst` - arreglo: Arreglo a donde se van a copiar unos elementos.
4. `dstOffset` - entero: Subíndice dentro del arreglo `dst` a partir del cual se van a colocar los elementos.
5. `count` - entero: Cuántos elementos se van a copiar.

`length(arr)`

Función que devuelve la longitud de un arreglo.

Argumentos:

1. `arr` - arreglo: El arreglo.

Devuelve:

La longitud del arreglo `arr`.

`range(lower, upper)`

Función que devuelve un nuevo arreglo llenado con enteros consecutivos desde `lower` hasta `upper`.

Argumentos:

1. `lower` - entero: El valor para el primer elemento del arreglo.

2. `upper` - entero: El valor para el último elemento del arreglo.

Devuelve:

Un arreglo con `upper - lower + 1` elementos. Conteniendo los enteros consecutivos desde `lower` hasta `upper`.

Ambientes

Procedimientos y funciones para manipular ambientes.

`getNames(env)`

Función que devuelve los nombres de todos las variables dentro de un ambiente.

Argumentos:

1. `env` - ambiente: El ambiente del cual hay que obtener la lista de nombres.

Devuelve:

Un arreglo de strings donde cada elemento es el nombre de una variable en el ambiente `env`.

`getValues(env)`

Función que devuelve los valores de todas las variables dentro de un ambiente.

Argumentos:

1. `env` - ambiente: El ambiente del cual hay que obtener la lista de valores.

Devuelve:

Un arreglo de valores, con un valor por cada variable en el ambiente `env`.

`hasName(name)`

Función para averiguar si existe una variable con cierto nombre en el ambiente actual.

Argumentos:

1. `name` - string: El nombre de la variable.

Devuelve:

`true` si existe una variable con ese nombre dentro del ambiente actual, de lo contrario devuelve `false`.

`hasName(name, env)`

Función para averiguar si existe una variable con cierto nombre en un ambiente.

Argumentos:

1. `name` - string: El nombre de la variable.
2. `env` - ambiente: El ambiente.

Devuelve:

`true` si existe una variable con ese nombre en el ambiente `env`, de lo contrario devuelve `false`.

`removeName(name)`

Procedimiento que elimina una variable del ambiente actual.

Argumentos:

1. `name` - string: El nombre de la variable a eliminar. Si no existe en el ambiente actual una variable con ese nombre, entonces no hace nada.

`removeName(name, env)`

Procedimiento que elimina una variable de un ambiente.

Argumentos:

1. `name` - string: El nombre de la variable a eliminar. Si no existe en el ambiente una variable con ese nombre, entonces no hace nada.

2. `env` - ambiente: El ambiente del cual hay que eliminar la variable.

Tipos

Funciones para preguntar acerca del tipo de algún valor.

`isArray(obj)`

Función para determinar si un valor es de tipo arreglo.

Argumentos:

1. `obj` - cualquier tipo: El valor.

Devuelve:

`true` si `obj` es un arreglo, de lo contrario devuelve `false`.

`isBoolean(obj)`

Función para determinar si un valor de tipo booleano. Los únicos valores booleanos son `true` y `false`.

Argumentos:

1. `obj` - cualquier tipo: El valor.

Devuelve:

`true` si `obj` es un booleano, de lo contrario devuelve `false`.

`isCollection(obj)`

Función para determinar si un valor es de tipo colección. Las colecciones son:

1. Queue (cola)
2. Set (conjunto)
3. Stack (pila)

Argumentos:

1. `obj` - cualquier tipo: El valor.

Devuelve:

`true` si `obj` es una cola, un conjunto o una pila, de lo contrario devuelve `false`.

`isNumber(obj)`

Función para determinar si un valor de tipo numérico. Los enteros y los flotantes son numéricos.

Argumentos:

1. `obj` - cualquier tipo: El valor.

Devuelve:

`true` si `obj` es un entero o un flotante, de lo contrario devuelve `false`.

`isProcedure(obj)`

Función para determinar si un valor es de tipo procedure (una función es un caso especial de un procedimiento).

Argumentos:

1. `obj` - cualquier tipo: El valor.

Devuelve:

`true` si `obj` es un procedure, de lo contrario devuelve `false`.

`isQueue(obj)`

Función para determinar si un valor es una cola (queue).

Argumentos:

1. `obj` - cualquier tipo: El valor.

Devuelve:

`true` si `obj` es una cola, de lo contrario devuelve `false`.

`isSet(obj)`

Función para determinar si un valor es un conjunto (set).

Argumentos:

1. `obj` - cualquier tipo: El valor.

Devuelve:

`true` si `obj` es un conjunto, de lo contrario devuelve `false`.

`isStack(obj)`

Función para determinar si un valor es una pila (stack).

Argumentos:

1. `obj` - cualquier tipo: El valor.

Devuelve:

`true` si `obj` es una pila, de lo contrario devuelve `false`.

`isString(obj)`

Función para determinar si un valor es un string.

Argumentos:

1. `obj` - cualquier tipo: El valor.

Devuelve:

`true` si `obj` es un string, de lo contrario devuelve `false`.

Estructuras de datos

Además de los arreglos y los ambientes, simpleJ también tiene soporte para tres tipos de contenedores: colas (queues), conjuntos (sets) y pilas (stacks). Un contenedor es una estructura que contiene cero o más datos.

Estos contenedores están implementados como ambientes que contienen los datos y procedimientos para operar sobre estos datos. Estos contenedores se crean por medio de unas funciones especiales que se llaman *constructores*.

`Queue()`

Constructor que crea una nueva cola. Una cola es una estructura que puede contener varios valores. Los valores se agregan al final de la cola y se extraen del principio de la cola.

Ejemplo:

```
var q = Queue();

q.put(1);
q.put(2);
q.put("abc");
q.put([10, 20, 30]);

while (!q.isEmpty())
  print(q.get());
```

`q.contains(element)`

Función para averiguar si la cola contiene un dato.

Argumentos:

1. `element` - cualquier tipo: El dato.

Devuelve:

`true` si la cola contiene el dato `element`, de lo contrario devuelve `false`.

`q.createWith(data)`

Función que crea una nueva cola, la cual contiene los datos de otra estructura.

Argumentos:

1. `data` - arreglo, queue, set o stack: La estructura que contiene los datos con los cuales hay que crear la cola.

Devuelve:

Una cola conteniendo los datos que se encuentran dentro de `data`.

`q.get()`

Función que devuelve el primer elemento de la cola (y lo quita de la cola).

Devuelve:

El primer elemento de la cola.

`q.isEmpty()`

Función para averiguar si la cola está vacía (si no contiene ningún elemento).

Devuelve:

`true` si la cola está vacía, de lo contrario devuelve `false`.

`q.put(element)`

Procedimiento que agrega un elemento al final de la cola.

Argumentos:

1. `element` - cualquier tipo: El elemento que se debe agregar al final de la cola.

`q.putAll(data)`

Procedimiento que agrega a la cola todos los datos de otra estructura.

Argumentos:

1. `data` - arreglo, queue, set o stack: La estructura que contiene los datos que hay que agregar al final de la cola.

`q.removeAll()`

Procedimiento que elimina todos los datos de la cola. Justo después de llamar a este procedimiento, un llamado a `q.isEmpty()` devuelve `true`.

`q.size()`

Procedimiento para averiguar cuántos elementos contiene la cola.

Devuelve:

Un entero con el número de elementos que hay en la cola.

`q.toArray()`

Función para crear un arreglo con los elementos que están en la cola.

Devuelve:

Un arreglo que contiene los elementos que están en la cola.

`Set()`

Constructor que crea un nuevo conjunto. Un conjunto es una estructura que no contiene duplicados de ninguno de sus elementos.

Ejemplo:

```
var s = Set();

s.put(1);
```

```
s.put(2);
s.put(1);
s.put("abc");

print(s.contains(1));
s.remove(1);
print(s.contains(1));
```

`s.contains(element)`

Función para averiguar si el conjunto contiene un dato.

Argumentos:

1. `element` - cualquier tipo: El dato.

Devuelve:

`true` si el conjunto contiene el dato, de lo contrario devuelve `false`.

`s.createWith(data)`

Función que crea un nuevo conjunto, el cual contiene los datos de otra estructura. Si un elemento está dos o más veces en la estructura de datos, sólo aparece una sola vez dentro del conjunto.

Argumentos:

1. `data` - arreglo, queue, set o stack: La estructura que contiene los datos con los cuales hay que crear el conjunto.

Devuelve:

Un conjunto conteniendo los datos que se encuentran dentro de `data`.

`s.isEmpty()`

Función para averiguar si el conjunto está vacío (si no contiene ningún elemento).

Devuelve:

`true` si el conjunto está vacío, de lo contrario devuelve `false`.

`s.put(element)`

Procedimiento que agrega un elemento a un conjunto. Si ya estaba ese elemento dentro del conjunto, entonces no hace nada.

Argumentos:

1. `element` - cualquier tipo: El elemento que se debe agregar al conjunto.

`s.putAll(data)`

Procedimiento que agrega al conjunto todos los datos de otra estructura. Si un elemento está dos o más veces en la estructura de datos, sólo aparece una sola vez dentro del conjunto.

Argumentos:

1. `data` - arreglo, queue, set o stack: La estructura que contiene los datos que hay que agregar al conjunto.

`s.remove(element)`

Procedimiento que elimina un dato del conjunto. Si ese dato no pertenecía al conjunto, entonces no hace nada.

Argumentos:

1. `element` - cualquier tipo: El elemento que hay que eliminar del conjunto.

`s.removeAll()`

Procedimiento que elimina todos los datos del conjunto. Justo después de llamar a este procedimiento, un llamado a `s.isEmpty()` devuelve `true`.

`s.size()`

Procedimiento para averiguar cuántos elementos contiene el conjunto.

Devuelve:

Un entero con el número de elementos que hay en el conjunto.

`s.toArray()`

Función para crear un arreglo con los elementos que hay en el conjunto.

Devuelve:

Un arreglo que contiene los elementos que están en el conjunto.

`Stack()`

Constructor que crea una nueva pila. Una pila es una estructura que puede contener varios valores. Los valores se agregan en la parte superior de la pila y se extraen de la parte superior de la pila.

Ejemplo:

```
var s = Stack();

s.put(1);
s.put(2);
s.put("abc");
s.put([10, 20, 30]);

while (!s.isEmpty())
  print(s.get());
```

`s.contains(element)`

Función para averiguar si la pila contiene un dato.

Argumentos:

1. `element` - cualquier tipo: El dato.

Devuelve:

`true` si la pila contiene el dato `element`, de lo contrario devuelve `false`.

`s.createWith(data)`

Función que crea una nueva pila, la cual contiene los datos de otra estructura.

Argumentos:

1. `data` - arreglo, queue, set o stack: La estructura que contiene los datos con los cuales hay que crear la pila.

Devuelve:

Una pila conteniendo los datos que se encuentran dentro de `data`.

`s.isEmpty()`

Función para averiguar si la pila está vacía (si no contiene ningún elemento).

Devuelve:

`true` si la pila está vacía, de lo contrario devuelve `false`.

`s.pop()`

Función que devuelve el elemento que está en la parte superior de la pila (y lo quita de la pila).

Devuelve:

El elemento que está en la parte superior de la pila.

`s.push(element)`

Procedimiento que agrega un elemento en la parte superior de la pila.

Argumentos:

1. `element` - cualquier tipo: El elemento que se debe agregar en la parte superior de la pila.

`s.putAll(data)`

Procedimiento que agrega a la pila todos los datos de otra estructura.

Argumentos:

1. `data` - arreglo, queue, set o stack: La estructura que contiene los datos que hay que agregar en la parte superior de la pila.

`s.removeAll()`

Procedimiento que elimina todos los datos de la pila. Justo después de llamar a este procedimiento, un llamado a `s.isEmpty()` devuelve `true`.

`s.size()`

Procedimiento para averiguar cuántos elementos contiene la pila.

Devuelve:

Un entero con el número de elementos que hay en la pila.

`s.toArray()`

Función para crear un arreglo con los elementos que hay en la pila.

Devuelve:

Un arreglo que contiene los elementos que están en la pila.

`append(a, b)`

Función para crear una nueva estructura que contiene todos los elementos de otras dos estructuras. Cada una de las estructuras puede ser un arreglo, una cola, un conjunto o una pila. No es necesario que las dos estructuras sean del mismo tipo. La estructura creada es del mismo tipo que el primer argumento.

Argumentos:

1. `a` - arreglo, queue, set o stack: La primera estructura.
2. `b` - arreglo, queue, set o stack: La segunda estructura.

Devuelve:

Una estructura, del mismo tipo que la estructura `a`, que contiene todos los elementos de las estructuras `a` y `b`.

`chooseOne(data)`

Función que devuelve al azar uno de los elementos de una estructura. Todos los elementos tienen la misma probabilidad de ser seleccionados.

Argumentos:

1. data - arreglo, queue, set o stack: La estructura.

Devuelve:

Uno de los elementos de la estructura escogido aleatoriamente.

`contains(data, element)`

Función para averiguar si una estructura contiene un dato.

Argumentos:

1. `data` - arreglo, queue, set o stack: La estructura.
2. `element` - cualquier tipo: El dato.

Devuelve:

`true` si la estructura contiene el dato `element`, de lo contrario devuelve `false`.

`max(data)`

Función que devuelve el elemento más grande de una estructura de datos. Todos los datos dentro de la estructura deben ser números o strings (no se pueden mezclar). En el caso de que sean números, el elemento más grande es el más cercano al infinito positivo. En el caso de que sean strings, el elemento mayor es el último al ordenarlos en orden "alfabético". Es un error llamar esta función con una estructura vacía.

Nota

Para la computadora todas las letras mayúsculas vienen antes que todas la minúsculas. Es decir que la `Z` viene antes que la `a`.

Argumentos:

1. `data` - arreglo, queue, set o stack: La estructura.

Devuelve:

El elemento más grande de la estructura.

`min(data)`

Función que devuelve el elemento más pequeño de una estructura de datos. Todos los datos dentro de la estructura deben ser números o strings (no se pueden mezclar). En el caso de que sean números, el elemento más pequeño es el más cercano al infinito negativo. En el caso de que sean strings, el elemento menor es el primero al ordenarlos en orden "alfabético". Es un error llamar esta función con una estructura vacía.

Nota

Para la computadora todas las letras mayúsculas vienen antes que todas la minúsculas. Es decir que la `Z` viene antes que la `a`.

Argumentos:

1. `data` - arreglo, queue, set o stack: La estructura.

Devuelve:

El elemento más pequeño de la estructura.

`prod(data)`

Función que devuelve el producto (multiplicación) de todos los elementos de una estructura. Todos estos elementos tienen que ser números.

Argumentos:

1. `data` - arreglo, queue, set o stack: La estructura.

Devuelve:

El producto de todos los elementos en la estructura. Si la estructura está vacía, entonces devuelve un uno.

`size(data)`

Función para averiguar cuántos elementos contiene una estructura.

Argumentos:

1. `data` - arreglo, queue, set o stack: La estructura.

Devuelve:

Un entero con el número de elementos en la estructura.

`sort(data)`

Función que devuelve una nueva estructura con los datos de otra estructura ordenados de menor a mayor. Todos los datos dentro de la estructura deben ser números o strings (no se pueden mezclar).

Nota

Para la computadora todas las letras mayúsculas vienen antes que todas la minúsculas. Es decir que la `Z` viene antes que la `a`.

Argumentos:

1. `data` - arreglo, queue, set o stack: La estructura.

Devuelve:

Una nueva estructura con los elementos de `data` ordenados de menor a mayor. Esta nueva estructura es del mismo tipo que `data`.

`sum(data)`

Función que devuelve la sumatoria (suma) de todos los elementos de una estructura. Todos estos elementos tienen que ser números.

Argumentos:

1. `data` - arreglo, queue, set o stack: La estructura.

Devuelve:

La sumatoria de todos los elementos en la estructura. Si la estructura está vacía, entonces devuelve un cero.

`toArray(data)`

Función que crea un nuevo arreglo, el cual contiene los datos de otra estructura.

Argumentos:

1. `data` - arreglo, queue, set o stack: La estructura que contiene los datos con los cuales hay que crear el arreglo.

Devuelve:

Un arreglo conteniendo los datos que se encuentran dentro de `data`.

`toQueue(data)`

Función que crea una nueva cola, la cual contiene los datos de otra estructura.

Argumentos:

1. `data` - arreglo, queue, set o stack: La estructura que contiene los datos con los cuales hay que crear la cola.

Devuelve:

Una cola conteniendo los datos que se encuentran dentro de `data`.

`toSet(data)`

Función que crea un nuevo conjunto, el cual contiene los datos de otra estructura. Si un elemento está dos o más veces en la estructura de datos, sólo aparece una sola vez dentro del conjunto.

Argumentos:

1. `data` - arreglo, queue, set o stack: La estructura que contiene los datos con los cuales hay que crear el conjunto.

Devuelve:

Un conjunto conteniendo los datos que se encuentran dentro de `data`.

`toStack(data)`

Función que crea una nueva pila, la cual contiene los datos de otra estructura.

Argumentos:

1. `data` - arreglo, queue, set o stack: La estructura que contiene los datos con los cuales hay que crear la pila.

Devuelve:

Una pila conteniendo los datos que se encuentran dentro de `data`.

Con procedimientos como argumentos

En simpleJ los procedimientos (y por lo tanto también las funciones) son un dato de tipo procedure. Al igual que cualquier otro tipo de dato se pueden asignar a una variable, pasar como argumento a otro procedimiento o ser devuelto por una función.

`apply(proc, arr)`

Procedimiento que aplica un procedimiento a los datos de un arreglo. El número de datos que contiene el arreglo debe ser igual al número de argumentos que espera el procedimiento. Si `proc` es una función entonces `apply` también se comporta como una función (devuelve un resultado).

Ejemplo:

```
var p = showAt;
apply(p, ["Mensaje", 10, 12]);
print(apply(sqrt, [16]));
```

Argumentos:

1. `proc` - procedure: El procedimiento a aplicar.

2. `arr` - arreglo: El arreglo con los argumentos para el procedimiento `proc`.

Devuelve:

Si el valor de `proc` es una función entonces devuelve el resultado de llamar esa función con los datos del arreglo `arr` como argumentos. Si no es una función, entonces no devuelve nada.

`filter(pred, data)`

Función que aplica un predicado a una estructura de datos y devuelve una nueva estructura de datos que únicamente contiene los elementos para los cuales el predicado fue cierto. Un *predicado* es una función que espera un solo argumento y devuelve `true` o `false`. La nueva estructura es del mismo tipo que la estructura que se le pasó como segundo argumento.

Ejemplo:

```
print(filter(isNumber, [1, "a", 10.0, true]));
```

Argumentos:

1. `pred` - procedure: Función que espera un solo argumento y devuelve `true` o `false`.
2. `data` - arreglo, queue, set o stack: Estructura con los datos que hay que filtrar.

Devuelve:

Una nueva estructura que contiene los datos de `data` para los cuales `pred` devolvió `true`. Es del mismo tipo que `data`.

`map(func, data)`

Función que devuelve una estructura que contiene el resultado de aplicar una función de un solo argumento a cada uno de los elementos de una estructura.

Ejemplo:

```
print(map(sqrt, [1, 4, 9, 16, 25]));
```

Argumentos:

1. `func` - procedure: Función que espera un solo argumento.
2. `data` - arreglo, queue, set o stack: Estructura con los datos.

Devuelve:

Una nueva estructura que contiene los resultados de aplicarle la función `func` a cada uno de los datos en `data`. Es del mismo tipo que `data`.

`mappend(func, data)`

Función que devuelve un arreglo con todos los valores que se obtienen al aplicar una función func que espera un arreglo y devuelve una estructura a cada uno de los elementos de una estructura.

Ejemplo:

```
numeros(n) {
  return range(0, n);
}

print(mappend(numeros, [1, 5, 10]));
```

Argumentos:

1. `func` - procedure: Función que espera un solo argumento y devuelve una estructura.

2. `data` - arreglo, queue, set o stack: Estructura con los datos.

Devuelve:

Un arreglo con todos los valores que se obtienen al aplicar la función `func` a cada uno de los elementos de `data`.

`reduce(binOp, data, firstValue)`

Función que acumula todos los valores de una estructura en un solo valor empleando un valor inicial y una función que espera dos argumentos.

Ejemplo:

```
concat(s1, s2) {
  return s1 + s2;
}
```

```
print(reduce(concat, ["hola", "adios", "algo"], "xxx"));
print(reduce(concat, ["123", "456", "789"], "0"));
```

Argumentos:

1. `binOp` - procedure: Función que espera dos argumentos.

2. `data` - arreglo, queue, set o stack: Estructura con los datos.

3. `firstValue` - cualquier tipo: El valor inicial.

Devuelve:

El resultado de acumular todos los valores de `data` con `firstValue` empleando la función `binOp`.

`reducef(binOp, data)`

Función que acumula todos los valores de una estructura en un solo valor empleando una función que espera dos argumentos. Es un error llamarla con una estructura vacía.

Ejemplo:

```
concat(s1, s2) {
  return s1 + s2;
}

print(reducef(concat, ["hola", "adios", "algo"]));
print(reducef(concat, ["123", "456", "789"]));
```

Argumentos:

1. `binOp` - procedure: Función que espera dos argumentos.

2. `data` - arreglo, queue, set o stack: Estructura con los datos.

Devuelve:

El resultado de acumular todos los valores de `data` empleando la función `binOp`.

sortQueue(q, comp)

Función que devuelve una nueva cola con los elementos de otra cola ordenados de acuerdo al criterio especificado por una función de dos argumentos.

Ejemplo:

```
mayorQue(x, valor) {
  return x > valor;
}

var q = Queue();
q.put(2);
q.put(1);
q.put(3);
q.put(5);

print(toArray(sortQueue(q, mayorQue)));
```

Argumentos:

1. `q` - queue: Los datos que hay que ordenar.

2. `comp` - procedure: La función de comparación. Esta función devuelve `true` si su primer argumento debe ir antes en la cola que su segundo argumento, de lo contrario devuelve `false`.

Devuelve:

Una nueva cola con los datos de `q` ordenados de acuerdo al criterio indicado por la función `comp`.

sortc(data, comp)

Función que devuelve una nueva estructura con los elementos de otra estructura ordenados de acuerdo al criterio especificado por una función de dos argumentos. No tiene mucho sentido emplear este procedimiento para ordenar un conjunto (el orden de los elementos dentro de un conjunto debería ser irrelevante).

Ejemplo:

```
mayorQue(x, valor) {
  return x > valor;
```

```
}

print(sortc([4, 2, 10, 5, 20], mayorQue));
```

Argumentos:

1. `data` - arreglo, queue, set o stack: Estructura con los datos que hay que ordenar.

2. `comp` - procedure: La función de comparación. Esta función devuelve `true` si su primer argumento debe ir antes en la estructura que su segundo argumento, de lo contrario devuelve `false`.

Devuelve:

Una nueva estructura con los datos de `data` ordenados de acuerdo al criterio indicado por la función `comp`. Es del mismo tipo que `data`.

Otros

Procedimientos que no se pueden clasificar dentro de alguna de las secciones anteriores.

`error(msg)`

Procedimiento que detiene la ejecución del programa con un mensaje de error.

Argumentos:

1. `msg` - string: El mensaje de error.

`pause(time)`

Procedimiento que suspende la ejecución de un programa durante un tiempo determinado.

Argumentos:

1. `time` - flotante: El tiempo, en segundos, durante el cual debe suspenderse la ejecución del programa.

`print(obj)`

Procedimiento para desplegar un valor en el Log.

Argumentos:

1. `obj` - cualquier tipo: El valor a desplegar.

Apéndice D. Instalación

simpleJ funciona en Windows, Mac OS X, Linux y Solaris. Puede que también funcione con otros sistemas operativos pero todavía no lo hemos probado.

Nota

Si no logras instalar simpleJ, entonces manda un correo electrónico a <ayuda-instalacion@simplej.com> para que alguien te ayude.

Cómo instalar simpleJ en Windows

Para instalar simpleJ en Windows basta con abrir el programa `simpleJ-1_0-setup.exe` que se encuentra en el CD que viene con este libro.

simpleJ está escrito en Java y requiere el Java Runtime Environment 1.4.2 o más nuevo (recomendamos emplear la versión 1.5). El programa `simpleJ-1_0-setup.exe` se encarga de eso automáticamente. Si sabes que ya tienes la versión correcta de Java en tu computadora entonces puedes instalar simpleJ abriendo directamente el programa `simpleJ-1_0-install.jar` que está en el CD .

Nota

Si al tratar de instalar simpleJ empleando el programa `simpleJ-1_0-setup.exe` te topas con algún problema entonces puede que se deba a que ya tengas instalada en tu computadora una versión demasiado vieja de Java. En caso de que esto ocurra, sigue los pasos siguientes:

1. Emplea el administrador de programas (que se encuentra en el panel de control de Windows) para desinstalar la vieja versión de Java.

2. Abre el programa `jre-1_5_0_07-windows-i586-p.exe` (también incluido en el CD) para instalar una versión más reciente del Java Runtime Environment.

3. Al terminar de instalar Java ya puedes instalar simpleJ abriendo el programa `simpleJ-1_0-install.jar`.

Si aún así no logras instalar simpleJ, entonces manda un correo electrónico a `<ayuda-instalacion@simplej.com>` para que alguien te ayude.

Cómo instalar simpleJ en Mac OS X

Para instalar simpleJ en Windows basta con abrir el programa `simpleJ-1_0-install.jar` que se encuentra en el CD que viene con este libro. Al terminar de instalarse puedes encontrar todos los programas de simpleJ dentro de la carpeta `simpleJ` que está dentro de `Aplicaciones`.

Cómo instalar simpleJ en Linux

simpleJ está escrito en Java y necesita el Java Runtime Environment 1.4.2 o más nuevo (recomendamos emplear la versión 1.5). Si no tienes Java en tu computadora, entonces primero lo debes descargar de `http://java.sun.com/javase/downloads` e instalarlo.

Ya que tengas Java instalado, entonces puedes instalar simpleJ ejecutando este comando:

```
java -jar simpleJ-1_0-install.jar
```

(el programa `simpleJ-1_0-install.jar` se encuentra en el CD que viene con este libro).

Una vez que esté instalado simpleJ, ve al directorio donde lo instalaste y ejecuta alguno de estos comandos:

1. Para ejecutar el simpleJ devkit

   ```
   java -jar devkit.jar
   ```

2. Para ejecutar la simpleJ virtual console

   ```
   java -jar console.jar
   ```

3. Para ejecutar el simpleJ tiles editor

```
java -jar tileseditor.jar
```

4. Para ejecutar el simpleJ sprites editor

```
java -jar sprites.jar
```

Cómo instalar simpleJ en Solaris

Instala simpleJ ejecutando este comando:

```
java -jar simpleJ-1_0-install.jar
```

(el programa `simpleJ-1_0-install.jar` se encuentra en el CD que viene con este libro).

Una vez que esté instalado simpleJ, ve al directorio donde lo instalaste y ejecuta alguno de estos comandos:

1. Para ejecutar el simpleJ devkit

```
java -jar devkit.jar
```

2. Para ejecutar la simpleJ virtual console

```
java -jar console.jar
```

3. Para ejecutar el simpleJ tiles editor

```
java -jar tileseditor.jar
```

4. Para ejecutar el simpleJ sprites editor

```
java -jar sprites.jar
```

Cómo instalar simpleJ en otros sistemas operativos

simpleJ está escrito en Java y necesita el Java Runtime Environment 1.4.2 o más nuevo (recomendamos emplear la versión 1.5). Si no tienes Java en tu computadora, entonces primero conseguirlo e instalarlo.

Ya que tengas Java instalado, entonces puedes instalar simpleJ ejecutando este comando:

```
java -jar simpleJ-1_0-install.jar
```

(el programa `simpleJ-1_0-install.jar` se encuentra en el CD que viene con este libro).

Una vez que esté instalado simpleJ, ve al directorio donde lo instalaste y ejecuta alguno de estos comandos:

1. Para ejecutar el simpleJ devkit

   ```
   java -jar devkit.jar
   ```

2. Para ejecutar la simpleJ virtual console

   ```
   java -jar console.jar
   ```

3. Para ejecutar el simpleJ tiles editor

   ```
   java -jar tileseditor.jar
   ```

4. Para ejecutar el simpleJ sprites editor

   ```
   java -jar sprites.jar
   ```

Nota

No hemos probado simpleJ con otros sistemas operativos, pero si Java está disponible entonces lo más probable es que simpleJ funcione sin ningún problema.

Colofón

Este libro fue escrito empleando un editor de XML con el DTD de DocBook (el formato estándar para libros técnicos). Muchos de los diagramas también fueron creados a partir de descripciones en XML. Para administrar estos archivos se empleó el sistema de control de versiones CVS. Toda la conversión de estos archivos XML para producir el PDF que se mandó a la imprenta se llevó a cabo por medio de programas en Java controlados por Ant.