JavaScript de Andy Harris

TEMA 3 – Adivinar números: bucles for y while

Estamos aprendiendo las tareas más importantes de la programación. Hasta ahora, hemos aprendido a almacenar la información en variables, enviar mensajes al usuario, obtener información del usuario y a realizar tomas de decisiones.

Ahora, vamos a ver un nuevo elemento crítico en programación, la facultad de un programa de repetirse las veces que queramos. Concretamente, aprenderemos a:

- **Usar los bucles for para repetir código un determinado número de veces.**
- **Modificar los bucles for para saltar algunos valores.**
- **Hacer que los bucles for vayan hacia atrás.**
- **Aprender a realizar la traza del código para comprobar que entendemos su funcionamiento.**
- **Crear bucles while.**
- **Evitar los bucles sin fin.**
- **Planificar programas complejos con pseudocódigo.**

Proyecto: “Adivinar un número”

Cómo siempre, vamos a empezar con un ejemplo. Al final del Tema seremos capaces de escribir el programa. Es un programa clásico de adivinar un número. El ordenador “pensará” un número entre 1 y 100 (ver Figura 3.1). El usuario intentará adivinar el número (ver Figura 3.2). Después de cada intento, el ordenador le dice al usuario si el número a encontrar es mayor, menor o correcto (ver Figuras 3.3 y 3.4). El ordenador guarda el número de intentos que tarda el usuario en adivinar el número. Cómo vemos, este programa preguntará reiteradamente al usuario por un número hasta que este lo acierte. Se ve que este programa va a utilizar tomas de decisión que hemos visto en el Tema 2. Ya que se informa al usuario si el número que hay que adivinar es mayor o menor que el intentado. Además, el programa se repite tantas veces cómo intentos necesite el usuario. A este comportamiento repetitivo se le llama **bucle** en programación. La mayoría de lenguajes de programación ofrecen al menos dos tipos de bucles. El primero se usará cuando necesitamos que algo se repita un número determinado de veces. Se llaman bucles for, y lo vemos en el siguiente programa:

Figura 3.1
Figura 3.2
Figura 3.3
Figura 3.4

Creando el “Corredor loco”

El “Corredor loco” es una simulación muy sencilla de una carrera de 10 vueltas. El programa únicamente muestra en un cuadro de diálogo en qué vuelta está el corredor. La Figura 3.5 muestra el programa. Aunque se puede escribir este programa como una serie de sentencias alert, el programa utiliza una sóla. Conseguimos que el programa repita la sentencia colocándola dentro de un blucle for. Este es el código:

```
<html>
<head><title>Corredor loco</title>
<script>
// Corredor loco
// Demuestra el bucle for básico

var vuelta = 0;
for (vuelta = 1; vuelta <= 10; vuelta++){
alert("El Corredor loco está en la vuelta: " + vuelta);
} // end for
</script>
</head>
<body>
<h1>El Corredor loco<br /></h1>
<hr>
</body>
</html>
```

El programa tiene una única variable, llamada `vuelta`, y una sentencia `alert`. Hemos encerrado la sentencia `alert` entre llaves. La sentencia **for** hace que la sentencia `alert` se ejecute 10 veces. La ventaja de usar un bucle en lugar de 10 sentencias `alert` es la flexibilidad. Si queremos que la carrera dure 100 vueltas, sólo tenemos que cambiar una línea de código.

Figure 3.5

Usando la sentencia for

La sentencia **for** pone en marcha un bucle. Es especialmente útil cuando sabemos exactamente cuántas veces queremos repetir algo. Echemos un vistazo a esta línea del programa “Corredor loco”:

```
for (vuelta = 1; vuelta <= 10; vuelta++){
```

Los bucles **for** se componen de tres partes dentro de los paréntesis. El primer segmento incializa la variable de control del bucle. La segunda parte es la condición que establece el final del bucle. La tercera parte cambia la variable de control y se ejecuta cada vez que se acaba el bucle. El segmento de inicialización, `vuelta = 1;`, especifica el nombre de una variable que se va a utilizar para controlar el bucle y su valor inicial. Para hacer un bucle for, necesitamos una variable numérica. En este caso, hemos utilizado la variable `vuelta`, que especificará en qué vuelta está el bucle. El punto y coma separa esta parte del bucle for de la siguiente. En el segundo segmento especificamos la condición de salida del bucle de la siguiente manera:

El código que se encuentra dentro de las llaves del bucle for se repetirá mientras la condición sea verdadera. En nuestro ejemplo, la variable `vuelta` empieza con el valor 1, y el código se repetirá mientras la variable sea menor o igual a 10. La tercera parte del for es:

```
vuelta++
```

El operador ++ es un atajo para decir “suma 1”. O sea, `vuelta++` es lo mismo que escirbir `vuelta = vuelta + 1`. La misión de esta parte del bucle for es cambiar el valor de la variable de forma que la variable `vuelta` alcance en algún momento un valor que haga que la condición sea falsa. Básicamente, lo que estamos haciendo es: “Usa la variable `vuelta`. Empenzando con el valor 1, seguir mientras sea menor o igual a 10. Cada vez que se complete el bucle, añadir 1 a la variable `vuelta`”.

La traza del código

Para explicar lo que sucede en el bucle, vamos a utilizar una técnica que usan los programadores habitualmente que se llama la traza del código. Esta técnica es de gran ayuda cuando estamos tratando de averiguar cómo funciona un trozo de código, o cuando estamos intentando averiguar porqué no hace lo que esperamos. Para hacer la traza del código, vamos a hacer una tabla en una hoja de papel. Vamos a escribir los nombres de las variables y las condiciones en las cabeceras de las columnas, así:

`vuelta`	`vuelta` <= 10
____	_____
____	_____
____	_____

Ahora estudiemos el código línea a línea. Cada vez que el valor de la variable cambia, escribimos el nuevo valor en la tabla. Si se utiliza el valor en una condición, escribimos si esta se hace verdadera o falsa. Empecemos con la primera línea:

```
var vuelta = 0;
```

Después de que el ordenador ha ejecutado esta línea, nuestra tabla quedará:

`vuelta`	`vuelta` <= 10
0	true
____	_____
____	_____

El valor de `vuelta` vale 0, y 0 es menor que 10 luego la condición es verdadera, por tanto siguiente línea, y seguimos con el proceso:

```
for (vuelta = 1; vuelta <= 10; vuelta++){
```

Una vez ejecutada esta línea por primera vez, esto es lo que tenemos:

`vuelta`	`vuelta` <= 10
0	true
1	true
____	_____

La primera vez que el programa pasa por la sentencia for, se inicializa la variable `vuelta` a 1. La condición se evalúa a verdadera, y el programa sigue con el código entre llaves. La sentencia `alert` le muestra al usuario el valor de `vuelta`. Cuando el ordenador encuentra la llave de cierre de final del bucle, el control vuelve automáticamente a la línea for. Esta vez, se ejecuta la tercera parte de la sentencia for, y se incrementa el valor de `vuelta` en 1, luego en el segundo paso por la línea del for, nuestra tabla tendrá estos valores:

`vuelta`	`vuelta` <= 10
0	true
1	true
2	true

El valor de `vuelta` es ahora 2, y la condición sigue siendo válida, por tanto, el bucle sigue ejecutándose. Las cosas se ponene interesantes cuando `vuelta` vale 10. Veámos la traza del código hasta aquí:

`vuelta`	`vuelta` <= 10
0	true
1	true
2	true
3	true
4	true
5	true
6	true
7	true
8	true
9	true
10	true

La próxima vez que pasamos por la sentencia for, el valor de `vuelta` vale 11. Esto es interesante, porqué por fin la condición ha cambiado, ahora se evalúa a **falso**:

`vuelta`	`vuelta` <= 10
0	true
1	true
2	true
3	true
4	true
5	true
6	true
7	true
8	true
9	true
10	true
11	**false**

La condición `vuelta` <= 10 se evalúa a falso, y el bucle no se vuelve a ejecutar, el control del programa sigue en la línea siguiente a la llave de cierre (}). Tal y cómo dijimos, el código se ha ejecutado exactamente diez veces, mostrando sucesivamente los valores entre 1 y 10.

Saltando valores

La forma del bucle for del programa “Corredor loco” demuestra el uso más habitual del bucle for. Podemos usar el bucle for de otras formas. El programa “Contar de cinco en cinco” muestra una de esas formas.

Creando el programa “Contar de cinco en cinco”

La Figura 3.6 muestra el programa “Contar de cinco en cinco”, que es un programa sencillo que incluye un bucle que cuenta de cinco en cinco en cada pasada. El programa es muy parecido al del “Corredor loco”. Echemos un vistazo al código, para ver las diferencias:

Figure 3.6

```
<html>
<head> <title>Contar de cinco en cinco</title>
<script>
// Contar de cinco en cinco
// demuestra cómo modificar el bucle for

var i = 0;
for (i=5; i <= 100; i += 5){
   alert (i);
} // end for loop
</script>
</head>
<body>
<h1>Contar de cinco en cinco<br /></h1>
<hr>
</body>
</html>
```

Cómo vemos la única diferencia es:

```
for (i=5; i <= 100; i += 5){
```

Ahora la variable de control se llama `i`. Hemos elegido este nombre porque no es necesario que la variable `i` tenga un significado especial. Es tradicional utilizar este nombre para controlar los bucles for.

CONSEJO – Puede parecer extraño que una disciplina tan joven cómo la programación de ordenadores pueda tener tradiciones, pero este particular (usar i cómo variable de control del bucle for) es un ejemplo. La razón de esta tradición se remonta al primer lenguaje de alto nivel: FORTRAN. En las primeras versiones de FORTRAN, las variables de tipo entero debían empezar por i, j, k.

Puede intentar predeterminar lo que hace la línea:

```
for (i=5; i <= 100; i += 5){
```

`i` empieza con el valor 5 y llega hasta el valor 100. Cada vez que se ejecuta el bucle, el valor de `i` se incrementa en cinco.

Usando el operador +=

Hemos visto el operador += en acción concatenando cadenas de caracteres. También se puede usar con variables numéricas. En este contexto, significa sumar 5 al contenido de la variable `i`. Podíamos también haber escrito `i = i + 5` que es exactamente lo mismo. La mayoría de programadores prefieren el operador +=, porque es más rápido.

Contando hacia atrás

Podemos diseñar también los bucles **for** para que cuenten hacia atrás. Todo lo que hay que hacer es modificar la línea del **for**.

Creando el programa “Contando hacia atrás”

La Figure 3.7 muestra otra versión del programa “Contar de cinco en cinco”, sólo que ahora hacia atrás.

El código es muy similar a los programas anteriores:

```
<html>
<head>
<title>Contar hacia atrás</title>
<script>
// Contar hacia atrás
// demuestra el contar hacia atrás con un bucle for
var i = 0;
for (i = 100; i > 0; i-=5){
alert(i);
} // end for loop
</script>
</head>
<body>
<h1> Contar hacia atrás<br /></h1>
<hr>
</body>
</html>
```

Figure 3.7

Cómo podíamos esperar, la única diferencia sucede en la línea del for. Esta vez, hemos inicializado la variable `i` a un valor mayor, que el programa decrementa en cinco, en cada ejecución del bucle. La condición que se comprueba es que `i` sea mayor que 0.

Usando el bucle while

El bucle for es una estructura fácil de usar y con una potencia tremenda cuando se trata de repetir código. Pero siempre que sepamos el número de veces que queremos que se repita. Cuando no sabemos, a priori, cuántas veces se tiene que ejecutar un bucle hay que utilizar otra estructura: el bucle **while**.

Creando el programa “El acertijo”

Las Figuras 3.8 and 3.9 muestran el programa “El acertijo”, que pregunta al usuario por un viejo acertijo, y le pide al usuario que lo resuelva. El programa necesita repetirse un cierto número de veces, que dependerá exclusivamente de si la respuesta del usuario es correcta o no. Si el usuario no consigue acerta el programa no acaba nunca.

Figure 3.8
Figure 3.9
Figure 3.10

Repitiendo un bucle un número indeterminado de veces

Veamos el código del programa “El acertijo”. Usa un bucle, pero no un bucle for.

```
<html>
<head>
<title>El acertijo</title>
<script>
// El acertijo
// Demuestra el bucle while
var solucion = "para llegar al otro lado";
var intento = "";
while (intento != solucion){
respuesta = prompt ("Porqué cruzó el pavo la carretera?", "");
if (intento == solucion){
alert ("Muy gracioso, verdad?");
} else {
alert ("No, no es eso...");
} // end if
} // end while
</script>
</head>
<body>
<h1>El acertijo <br /></h1>
<hr>
</body>
</html>
```

El programa empieza con dos variables. Una para guardar la solución y otra para guardar la respuesta del usuario.

TRUCO – La elección del nombre de las variables es deliberado. Con el nombre de las mismas ya sabemos qué vamos a guardar en cada una. Podíamos haber escogido cómo nombre de variable respuesta, pero esto podría dar lugar a confusión pues no sabríamos si se trata de la respuesta correcta o de la respuesta del usuario.

Usando la sentencia while

Una estructura **while** es la base del programa. Esta estructura empieza con la línea:

```
while (intento != solucion){
```

La estructura termina con su llave de cierre correspondiente:

```
} // end while
```

El código entre las dos llaves es lo que se repite. La sintáxis de una sentencia **while** es más sencilla que la del bucle for. Consiste solamente en la condición. En este caso, el bucle se seguirá ejecutando hasta que el usuario dé una respuesta correcta. Los bucles **while** siempre tienen una condición, y el código se sigue ejecutando mientras esta condición se evalúe a verdadera. En cuanto la condición sea falsa, se ejecutará la línea siguiente a la llave de cierre.

Detectando bucles que no se ejecutan nunca

Los bucles while son muy sencillos de construir, pero pueden ser muy peligrosos. Veamos si puede adivinar lo que le sucede al siguiente código:

```
var i = 10;
while (i < 10){
alert(i);
i++;
} // end while
```

La variable `i` se inicializa con el valor 10. A continuación se comprueba la condición, si `i` es menor que 10. La condición se evalúa a falsa, y el ordenador jamás ejecuta el código que hay dentro del bucle while. Este programa terminará inmediatamente, sin hacer nada. Cuando diseñemos un bucle while es importante comprobar que se ejecuta al menos una vez.

Detectando bucles sin fin

Este es un problema más difícil de detectar:

```
var i = 0;
while (i < 10){
alert (i);
} // end while
```

Ahora el código inicializa la variable correctamente, y la condición del bucle while se evalúa a verdadera. Pero hay un problema todavía más grave. En ningún sitio se modifica la variable de la condición que siempre valdrá 0. He aquí una traza parcial del programa:

i	i < 10
0	true
0	true
0	true

El valor de i nunca será mayor de 0, y la condición permanecerá verdadera para siempre. El código ha entrado en un **bucle infinito**. Es relativamente fácil hacer un bucle infinito cuando estamos aprendiendo a programar. La única forma de parar un bucle infinito es entrando en el administrador de tareas y matar la tarea del navegador.

Construyendo bucles bien formados

Los programadores experimentados han aprendido unos cuantos trucos que les ayudan a evitar los problemas de los bucles que hemos detectado:

- Diseñe la condición con cuidado. Piense detenidamente sobre las variables que va a utilizar en la condición, asegúrese de que están escritas correctamente. Compruebe que la condición está haciendo lo que se supone que hace.

- Inicialice las variables. Cualquier variable usada en la condición debe ser inicializada fuera del bucle, de forma que se pueda predecir lo que va a suceder la primera vez que se ejecute el bucle. Si tiene un bucle que no se ejecuta nunca, compruebe que las variables de la condición guardan el valor que usted supone.

- Asegúrese de que la condición llega as dispararse. El bucle debe contener algún código que pueda cambiar el valor de la variable de forma que en algún momento la condición pueda evaluarse a falso y el ordenador pueda salir del bucle.

Volviendo al programa “Adivinar un número”

El programa “Adivinar un número” es un programa interesante, porque incorpora muchos de las conceptos vistos hasta ahora e incorpora una estructura de bucle. En lugar de mostrar simplemente el código, vamos a ver cómo se ha diseñado el programa. El diseño es a menudo la fase más difícil de la programación.

Diseñando el programa

Lo primero que tenemos que hacer es pensar qué es lo que tiene que hacer el programa. A continuación transcribir cómo se desarrolla una ejecución del programa:

Ordenador	Usuario
Estoy pensando un número.	50
Demasiado alto.	25
Demasiado bajo.	30

Ese es! Lo has conseguido en tres intentos.

Aunque es una forma muy básica de lo que hace el programa, incorpora los elementos que hemos de tener en cuenta para construir el programa. El usuario debe ser capaz de introducir algunos valores, y el ordenador debe evaluar esas entradas. Concretamente, el ordenador debe comparar las entradas con un número generado de forma aleatoria. El código debe ser capaz de ejecutarse un número indeterminado de veces (hasta que el usuario acierte), además el ordenador debe registrar cuántos intentos lleva el usuario.

CONSEJO – Es una gran idea escribir un resumen cómo este de la ejecución del programa. Será más fácil juntar todas las parte del programa si tenemos claro cuáles son.

Escribiendo pseudocódigo

El proceso descrito anteriormente no nos da información suficiente para empezar a programar. Muchos programadores escriben su plan de trabajo en un lenguaje que se llama pseudocódigo. El pseudocódigo expresa las ideas principales del programa, pero es más fácil trabajar con él porque no necesita el rigor de un lenguaje de programación. He aquí el pseudocódigo para representar el programa “Adivinar un número”:

```
Crear las variables
intento = el intento del usuario
objetivo = número aleatorio generado por el ordenador
turnos = número de intentos que lleva el usuario
obtener un número aleatorio entre 1 y 100
explicar el juego al usuario
pedir al usuario que haga un intento hasta que acierte,
incrementar el contador de turnos
si el intento del usuario es demasiado alto,
decirlo y obtener un nuevo intento,
si el intento del usuario es demasiado bajo,
decirlo y obtener un nuevo intento,
decirle al usuario que ha acertado y en cuántos intentos
```

Hemos escrito el pseudocódigo en español, expresando una idea en cada línea. Si leyendo cada línea de pseudocódigo somos capaces de saber cómo hemos de implementarlo en JavaScript, nuestro pseudocódigo estará suficientemente detallado.

CONSEJO – El principal problema de los programadores principiantes es que quieren escribir el código demasiado pronto. Si se encuentra delante de la pantalla sin saber que tiene que escribir a continuación, apague el ordenador. Coja papel y lápiz y escriba un buen pseudocódigo. Cuando lo tenga, escribirá mejores programas.

No suele ser demasiado complicado, una vez que conocemos un lenguaje de programación, pasar del pseudocódigo al código. He aquí mi versión:

```
<html>
<head>
<title>Adivinar un número</title>
<script>
// Adivinar un número
// El clásico programa de adivinar números
var intento = 50;
var objetivo = 0;
var turnos = 0;
var mensaje = "";
objetivo = Math.floor(Math.random() * 100) + 1;
//alert(objetivo);
mensaje = "Estoy pensando un número entre 1 y 100. \n";
mensaje += "Intente adivinarlo, y le diré si su intento es demasiado
alto, \n";
mensaje += "demasiado bajo, o correcto. Intente hacerlo con el menor
número de intentos.";
alert (mensaje);
intento = eval(prompt("Cuál es su intento?", intento));
while (intento != objetivo){
turnos++;
```

```
if (intento > objetivo){
intento = eval(prompt (turnos + ". Demasiado alto!! Siguiente
intento?", intento));
} // end if
if (intento < objetivo){
intento = prompt (turnos + ". Demasiado bajo!! Siguiente intento?",
intento);
} // end if
if (intento == objetivo){
mensaje = "LO ACERTASTE!!! \n";
mensaje += "En "+ turnos + " intentos! ";
alert (mensaje);
} // end if
} // end while
</script>
</head>
```

Generando el objetivo

Visto que necesitamos generar un número aleatorio entre 1 y 100, recordamos el algoritmo para generar números aleatorios que vimos en el Tema 2 , “Números aleatorios y sentencia if”. Escribir el código es sencillo, cuando sabemos lo que queremos hacer.

TRUCO – Fijaros en esta línea de código:
//alert (objetivo);
Es una herramienta muy útil para depuración. Cuando escribimos el código que genera el valor objetivo, queremos saber que se está haciendo dentro del intervalo que hemos definido. La sentencia alert muestra el valor del objetivo, lo que me garantiza que el código que genera el objetivo está funcionando adecuadamente. Además, una vez sabemos el valor del objetivo, es más fácil comprobar que el programa sigue haciendo lo correcto cada vez que se repite el bucle. Cómo esta línea es tan útil, hemos decidido dejarla, una vez acabado el proceso de depuración. Simplemente la hemos comentado. De esta manera, si alguna vez tenemos necesidad de volver a depurar el programa la tendremos disponible. Cada vez que no esté seguro del valor que toma una variable utilice una sentencia alert y muestre su valor. Por supuesto no se olvide de comentarla una vez acabada la depuración.

Preparando el bucle

El pseudocódigo dice, “hasta que el usuario acierte”. El truco de convertir esto en código JavaScript es construir una condición con la que pueda trabajar el ordenador. Esta es mi solución:

```
while (intento != objetivo){
```

La condición utiliza dos variables, luego hemos tomado la precaución de inicializarlas antes de que empiece el bucle. El ordenador ha generado un valor para objetivo, luego está inicializada. Nos aseguramos que intento tiene un valor válido antes de empezar el bucle.

Obteniendo la entrada del usuario

La entrada del usuario es muy sencilla. La evaluamos inmediatamente (con la sentencia `eval`) para no tener que preocuparnos de valores no numéricos. Notar también que la entrada del usuario se repite varias veces. Primero preguntamos por un intento del usuario fuera del bucle (para inicializar intento), y luego se piden valores dentro del bucle. El cuadro de diálogo informa al usuario si su intento actual es mayor o menor y pide un nuevo intento. Esto es necesario, porque el valor de intento debe cambiar para que el bucle tenga alguna opción de terminar. El cuadro de diálogo también incluye el último intento del usuario para que este pueda verlo. Además, esto le está indicando al usuario que se espera un valor numérico.

Evaluando la entrada

Cuando el usuario introduce un intento, el programa la convierte en un valor numérico. Luego una serie de sentencias if comparan el intento con el objetivo. Si el valor es demasiado alto o demasiado bajo, el siguiente cuadro de diálogo informa al usuario de la situación y pide una nueva entrada. Si el valor es correcto, se presenta un mensaje acorde al usuario y además se le informa del número de intentos que ha necesitado para acertar. Usamos una variable especial mensaje cuando las salidas implican varias concatenaciones. La experiencia dice que usar una variable en estas situaciones disminuye el riesgo de errores. Cómo ejemplo, veamos el siguiente código:

```
var mensaje = num1 + " + " + num2 + " = " + eval(num1 + num2)
```

Puede no resulta obvio lo que hace la línea, y además es muy fácil equivocarse con las comillas y los signos más. Compare that line to the following series of statements:

```
var mensaje;
mensaje = num1;
mensaje += num2;
mensaje += " = ";
mensaje += eval(num1 + num2);
```

El segundo trozo de código es más largo, pero describe mejor lo que se está haciendo (el programa concatena una variable que muestra la suma de otras dos variables). Si obtiene un error en la primera versión, no sabrá muy bien a qué altura de la línea sucede. En la segunda versión sabrá exactamente en qué línea está el error.

Resumen

En este tema, hemos ampliado el uso de las condiciones para añadir bucles al comportamiento de los programas. Hemos aprendido a usar el bucle **for** para los casos en que sabemos a priori el número de repeticiones. Hemos visto algunas variaciones sobre el funcionamiento básico del bucle for, incluyendo bucles que se incrementan en un valor arbitrario e incluso bucles que cuentan hacia atrás. En este tema, también se introduce el potente bucle **while**. Hemos aprendido cómo usar los bucles en los programas y cómo evitar los peligros más habituales de estos. Por último, hemos visto cómo utilizar el pseudocódigo para ayudarnos a escribir programas más elaborados. En el tema siguiente, aprenderemos el modelo de objetos de JavaScript y veremos nuevas formas de obtener información del usuario.

EJERCICIOS – Tema 3

3.1 Escribir un programa que sume los 100 primeros números y muestre el resultado.
3.2. Mejorar el programa “Adivinar un número” de forma que una vez acertado, pregunte al usuario si quiere jugar de nuevo. Repetir el juego hasta que el usuario diga que no.
3.3. Escribir una versión de “Adivinar un número” con los roles cambiados. Ahora, el usuario elige un número al azar, y el ordenador trata de adivinarlo. El usuario contesta “a” (para alto), “b” (para bajo), o “c” (para correcto). Una vez acertado el ordenador mostrará un mensaje: “Qué bien lo he acertado en XX intentos!!”. Escribir el pseudocódigo antes de programar este ejercicio.