capítulo 6

Estructuras de control: bucles (lazos)

objetivos

En este capítulo aprenderá a:

- Distinguir entre las estructuras de selección y las de repetición.
- Entender el concepto de bucle.
- Construir bucles controlados por una condición de selección simple.
- Diseñar sumas y productos de una serie mediante bucles.
- Construir bucles anidados.
- Conocer el funcionamiento de la variante del bucle `for`, `for each`, introducido en Java 5 y 6.

introducción

- Una característica que aumenta considerablemente la potencia de las computadoras es su capacidad para resolución de algoritmos repetitivos con gran velocidad, precisión y fiabilidad, mientras que para las personas las tareas repetitivas son difíciles y tediosas de realizar; este capítulo cubre las estructuras de control iterativas o repetitivas de acciones; Java soporta tres tipos de ellas: los bucles `while`, `for` y `do-while`; todas éstas controlan el número de veces que una sentencia o listas de sentencias se ejecutan.

6.1 Sentencia `while`

Un bucle o lazo es cualquier construcción de programa que repite una sentencia o secuencia de sentencias determinado número de veces; cuando ésta se menciona varias veces en un bloque se denomina *cuerpo del bucle*; cada vez que éste se repite se denomina *iteración del bucle*. Las dos cuestiones principales de diseño en la construcción del bucle son: ¿cuál es el cuerpo del bucle? y ¿cuántas veces se iterará el cuerpo del bucle?

Un bucle `while` tiene una condición, una expresión lógica que controla la secuencia de repetición; su posición es delante del cuerpo del bucle y significa que `while` es un bucle *pretest*, de modo que cuando éste se ejecuta, se evalúa la condición antes de ejecutarse el cuerpo del bucle; la figura 6.1 representa el diagrama de `while`.

El diagrama indica que la ejecución de la sentencia o sentencias expresadas se repite mientras la condición del bucle permanece verdadera y termina al volverse falsa; también indica que la condición se examina antes de ejecutarse el cuerpo y, por consiguiente, si aquélla es inicialmente falsa, éste no se ejecutará; en otras palabras, el cuerpo de un bucle `while` se ejecutará cero o más veces.

sintaxis

```
1. while (condición_bucle)
     sentencia;                          → Cuerpo

2. while (condición_bucle)
   {
     sentencia-1;
     sentencia-2;
     .                                   → Cuerpo
     .
     .
     sentencia-n;
   }
```

`while`	es una palabra reservada de Java
`condición_bucle`	es una expresión lógica
`sentencia`	es una sentencia simple o compuesta

El comportamiento o funcionamiento de una sentencia o bucle `while` es:

1. Se evalúa *`condición_bucle`*.

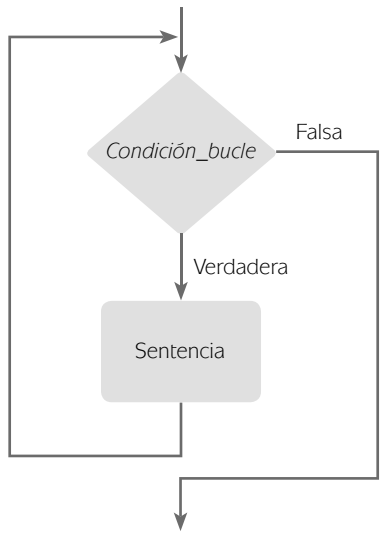


Figura 6.1 Diagrama de `while`.

2. Si es verdadera:
   a) La *sentencia* especificada, denominada *cuerpo del bucle*, se ejecuta.
   b) El control vuelve al paso 1.
3. En caso contrario: el control se transfiere a la sentencia posterior al bucle o sentencia while.

Por ejemplo:

```
// cuenta hasta 10
int x = 0;
while (x < 10)
  System.out.println("X: " + x++);
```

Otro ejemplo es:

```
// visualizar n asteriscos
contador = 0; ——————————→ inicialización
while (contador < n) ———→ prueba/condición
{
  System.out.print(" * ");
  contador++; ——————————→ actualización (incrementa en 1 contador)
} // fin de while
```

NOTA

Las sentencias del cuerpo del bucle se repiten mientras su condición o expresión lógica sea verdadera; cuando ésta se evalúa y resulta falsa, el bucle termina, se sale de él, y se ejecuta la siguiente sentencia.

La variable que representa la condición del bucle también se denomina *variable de control* debido a que su valor determina si el cuerpo se repite; ésta debe ser: 1) inicializada, 2) comprobada y 3) actualizada para que aquél se ejecute adecuadamente; cada etapa se resume así:

1. *Inicialización.* Se establece contador a un valor inicial antes de que se alcance la sentencia while, aunque podría ser cualquiera, generalmente es 0.
2. *Prueba/condición.* Se comprueba el valor de contador antes de la iteración; es decir, el comienzo de la repetición de cada bucle.
3. *Actualización.* Durante cada iteración, contador actualiza su valor incrementándose en uno mediante el operador ++.

Si la variable de control no se actualiza se ejecutará un bucle infinito; en otras palabras esto sucede cuando su condición permanece sin hacerse falsa en alguna iteración.

Por ejemplo:

```
// bucle infinito
contador = 1;
while (contador < 100)
{
  System.out.println(contador);
  contador-- ; ————→ decrementa en 1 contador
}
```

Se inicializa contador a 1 (menor que 100), como contador-- decrementa en 1 el valor de contador en cada iteración su valor nunca llegará a 100, número necesario para que la condición sea falsa; por consiguiente, contador < 100 siempre será verdadera, resultando un bucle infinito cuya salida será:

```
1
0
```

```
-1
-2
-3
-4
.
.
.
```

Por ejemplo:

```
// Bucle de muestra con while

class Bucle
{
  public static void main(String[] a)
  {
    int contador = 0;        // inicializa la condición
    while(contador < 5)      // condición de prueba
    {
      contador ++;           // cuerpo del bucle
      System.out.println("contador: " + contador);
    }
    System.out.println("Terminado.Contador: " + contador);
  }
}
```

Ejecución

```
contador:    1
contador:    2
contador:    3
contador:    4
contador:    5
Terminado.Contador: 5
```

EJEMPLO 6.1

Una de las aplicaciones más usuales del operador de incremento ++ es la de controlar la iteración de un bucle.

```
// programa cálculo de calorías
import java.util.Scanner;
class Calorias
{
  public static void main(String[] a)
  {
    int num_de_elementos, cuenta,
      calorias_por_alimento, calorias_total;
    Scanner entrada = new Scanner(System.in);
    System.out.print("¿Cuántos alimentos ha comido hoy? ");
    num_de_elementos = entrada.nextInt();
    System.out.println("Introducir el número de calorías de" +
    " cada uno de los " + num_elementos + " alimentos tomados:");
    calorias_total = 0;
    cuenta = 1;
    while (cuenta++ <= numero_de_elementos)
```

```
    {
      calorias_por_alimento = entrada.nextInt();
      calorias_total += calorias_por_alimento;
    }
    System.out.println(("Las calorías totales consumidas hoy son = " +
         calorias_total);
  }
```

Ejecución

```
¿Cuántos alimentos ha comido hoy? 4
Introducir el número de calorías de cada 1 de los 4
alimentos ingeridos:
500
350
1400
700
Las calorías totales consumidas hoy son = 2950
```

6.1.1 Terminaciones anormales de un bucle

Un error común en el diseño de una sentencia `while` se produce cuando el bucle sólo tiene una sentencia en lugar de varias como se planeó; el código siguiente

```
contador = 1;
while (contador < 25)
  System.out.println(contador);
contador++;
```

visualizará infinitas veces el valor 1 porque entra en un bucle infinito que no se actualiza al modificar la variable de control `contador`; la razón es que el punto y coma al final de la línea `System.out.println(contador);` hace que el bucle termine allí, aunque aparentemente el sangrado da la sensación de que el cuerpo de `while` contiene 2 sentencias, `System.out.println()` y `contador++`. El error se detecta rápidamente si el bucle se escribe correctamente:

```
contador = 1;
while (contador < 25)
  System.out.println(contador);
  contador++;
```

La solución más sencilla es utilizar las llaves de la sentencia compuesta como se muestra a continuación:

```
contador = 1;
while (contador < 25)
{
  System.out.println(contador);
  contador++;
}
```

6.1.2 Bucles controlados por centinelas

Por lo general, no se conoce con exactitud cuántos elementos de datos se procesarán antes de comenzar su ejecución debido a que hay muchos más por contar, o bien, porque el número de datos a procesar depende de la secuencia del proceso de cálculo.

Un medio para manejar esta situación es que el usuario introduzca, al final, un dato único, definido y específico en el llamado *valor centinela*; al hacerlo, la condición del bucle comprueba cada dato y termina cuando, al leer dicho valor, el valor centinela se selecciona con mucho cuidado pues no debe haber forma de que se produzca como dato. Como conclusión, el centinela sirve para terminar el proceso del bucle.

En el siguiente fragmento de código hay un bucle con centinela; se introducen notas mientras sean distintas a él.

```
/*
  entrada de datos numéricos,
  centinela -1
*/
final int centinela = -1;
System.out.print("Introduzca primera nota:");
nota = entrada.nextInt();
while (nota != centinela)
{
  cuenta++;
  suma += nota;
  System.out.print("Introduzca siguiente nota: ");
  nota = entrada.nextInt();
} // fin de while
System.out.println("Final");
```

Ejecución

Si se lee el primer valor de `nota`, por ejemplo 25, la ejecución puede ser:

```
Introduzca primera nota: 25
Introduzca siguiente nota: 30
Introduzca siguiente nota: 90
Introduzca siguiente nota: -1
Final
```

6.1.3 Bucles controlados por indicadores o banderas

Con frecuencia se utilizan variables de tipo `boolean` como indicadores o banderas de estado para controlar la ejecución de un bucle; su valor se inicializa normalmente como falso y, antes de la entrada al bucle, se redefine cuando un suceso específico ocurre dentro de éste. Un bucle controlado por un indicador o bandera se ejecuta hasta que se produce el suceso anticipado y se cambia el valor del indicador.

EJEMPLO 6.2

Se desea leer diversos datos de tipo `carácter` introducidos por teclado mediante un bucle `while`; el cual terminará cuando se lea un dato tipo `dígito` con rango entre 0 y 9.

La variable bandera `digito_leido` se utiliza para indicar cuando un dígito se introduce por medio del teclado; su valor es falso antes de entrar al bucle y mientras el dato leído sea un carácter, pero es verdadero si se trata de un dígito. Al comenzar, la ejecución del bucle deberá continuar mientras el dato leído sea un carácter y, en consecuencia, la variable `digito_leido` tenga un valor falso; el bucle se detendrá

cuando dicho dato sea un dígito y, en este caso, el valor de la variable digito_leido cambiará a verdadero. En consecuencia, la condición del bucle debe ser !digito_leido ya que es verdadera cuando digito_leido es falso. El bucle while será:

```
digito_leido = false; // no se ha leído ningún dato
while (!digito_leido)
{
  System.out.print("Introduzca un carácter: ");
  car = System.in.read(); // lee siguiente carácter
  System.in.skip(1);  // salta 1 carácter(fin de línea)
  digito_leido = (('0'<= car) && (car <= '9'));
  ...
} // fin de while
```

El bucle funciona de la siguiente forma:

1. Entrada del bucle: la variable digito_leido tiene un valor falso.
2. Como la condición del bucle !digito_leido es verdadera, se ejecutan las sentencias al interior del bucle.
3. Por medio del teclado se introduce un dato que se almacena en la variable car; si es un carácter, digito_leido se mantiene falso ya que es el resultado de la sentencia de asignación:

   ```
   digito_leido = (('0'<= car) && (car <= '9'));
   ```

   Si el dato ingresado es un dígito, entonces la variable digito_leido toma el valor verdadero resultante de la sentencia de asignación anterior.
4. El bucle termina cuando se lee un dígito entre 0 y 9 ya que la condición del bucle es falsa.

NOTA

El formato general del modelo de bucle controlado por un indicador es el siguiente:

1. Establecer el indicador de control a false o true para que while se ejecute correctamente; la primera vez, normalmente se inicializa a false.
2. Mientras la condición de control sea true:
   - 2.1 Realizar las sentencias del cuerpo del bucle.
   - 2.2 Cuando se produzca la condición de salida (en el ejemplo anterior que el dato carácter leído fuese un dígito), se deberá cambiar el valor de la variable indicador o bandera para que la condición de control cambie a false y el bucle termine.
3. Ejecutar las sentencias posteriores al bucle.

EJEMPLO 6.3

Se desea leer un dato numérico x con valor mayor que 0 para calcular la función f(x) = x*log(x).

La bandera xpositivo se utiliza para representar que el dato leído es mayor que 0, entonces la variable xpositivo se inicializa a false antes de que el bucle se ejecute y el dato de entrada se lea; cuando se ejecuta, el bucle debe continuar mientras el número leído sea negativo o 0, es decir, mientras la variable xpositivo sea false y se debe detener cuando el número leído sea mayor que 0, dando lugar a que xpositivo cambie a true. Considerando lo anterior, la condición del bucle debe ser !x_positivo ya que ésta es true cuando xpositivo es false; a su salida, el valor de la función se calcula y se escribe la codificación correspondiente:

```
import java.util.Scanner;
class FuncionLog
{
  public static void main(String[] a)
  {
    double f, x;
    boolean xpositivo;
```

```
        Scanner entrada = new Scanner(System.in);
        xpositivo = false; // inicializado a falso
        while (!xpositivo)
        {
          System.out.println("\n Valor de x: ");
          x = entrada.nextDouble();
          xpositivo = (x > 0.0); //asigna true si x>0.0
        }
        f = x*Math.log(x);
        System.out.println(" f(" + x + ") = " + f);

      }
    }
```

6.1.4 Sentencia `break` en bucles

La sentencia `break` a veces se utiliza para realizar una terminación anormal del bucle o antes de lo previsto; su sintaxis es:

```
break;
```

La sentencia `break` se utiliza para la salida de un bucle `while`, `do-while` o `for`, aunque su uso más frecuente es dentro de una sentencia `switch`.

```
while (condición1)
{
  if (condición2)
      break;
  // sentencias
}
```

EJEMPLO 6.4

El siguiente código lee y visualiza los valores de entrada hasta que se encuentra el valor `clave` especificado:

```
int clave = -9;
boolean activo = true;
while (activo)
{
  int dato;
  dato = entrada.nextInt();
  if (dato != clave)
    System.out.println(dato);
  else
    break;
}
```

¿Cómo funciona este bucle `while`? El método `nextInt()` lee un número entero desde el dispositivo de entrada; si su condición siempre fuera `true`, se ejecutaría indefinidamente; sin embargo, cuando hay un `dato==clave`, la ejecución sigue por `else` y, por su parte, `break` hace que la ejecución continúe en la sentencia siguiente a `while`.

PRECAUCIÓN

El uso de `break` en un bucle no es recomendable ya que puede dificultar la comprensión del comportamiento del programa; en particular, suele complicar la verificación de los invariantes. Además, la reescritura de los bucles sin `break` es fácil; por ejemplo, éste es el bucle anterior sin dicha sentencia:

```
final int clave = -9;
int dato = clave+1;

while (dato != clave)
{
  dato = entrada.nextInt();
  if (dato != clave)
  System.out.println(dato);
}
```

EJEMPLO 6.5

Calcular la media de 6 números

El proceso cotidiano al calcular una media de valores numéricos es: leerlos sucesivamente, sumarlos y dividir la suma total entre el número de valores leídos; el algoritmo más simple es:

```
Definir seis variables tipo float: num1,num2,num3,num4,num5,num6;
Definir variable tipo float: media;
Leer(num1,num2,num3,num4,num5,num6);
media = (num1+num2+num3+num4+num5+num6)/6;
```

Es evidente que si en lugar de 6 valores fueran 1 000, la modificación del código no sólo sería de longitud enorme, sino que la labor repetitiva de escritura sería tediosa; de esto deriva la necesidad de utilizar un bucle `while`; el algoritmo más simple es:

```
definir número de elementos como constante de valor 6
Inicializar contador de números
Inicializar acumulador (sumador) de números
Mensaje de petición de datos

mientras no estén leídos todos los datos hacer
  Leer número
  Acumular valor del número a variable acumulador
  Incrementar contador de números
fin_mientras

Calcular media (Acumulador/Total números)
Visualizar valor de la media
Fin
```

El programa Java sería el que se muestra a continuación:

```
// Calculo de la media de seis números

import java.util.Scanner;
class Media6
{
  public static void main(String []a)
  {
    Scanner entrada = new Scanner(System.in);
    final int TotalNum = 6;
    int contadorNum = 0;
    double sumaNum = 0.0;
    double media;
    System.out.println("/nIntroduzca %d números" + TotalNum);
    while (contadorNum < TotalNum)
    {
      // valores a procesar
      double numero;
      numero = entrada.nextDouble();
      sumaNum += numero; // añadir valor a Acumulador
      ++contadorNum;     // incrementar números leídos
    }
    media = sumaNum/contadorNum;
```

```
        System.out.println("Media: \n" + media);
      }
   }
```

6.1.5 La sentencia `break` con etiqueta

Para transferir el control a la siguiente sentencia de una estructura de bucles anidados, se utiliza la sentencia `break` con etiqueta; cuya sintaxis es:

```
break etiqueta
```

Por ejemplo, el siguiente fragmento escribe números enteros generados aleatoriamente y termina el bucle cuando el número es múltiplo de `7`.

Los números son generados por el método `random()` de la clase `Math` el cual devuelve un valor `double` mayor o igual a 0.0 y menor que 1.0; para obtener valores enteros se multiplica por una variable que toma valores desde `11` hasta 11^4 y se generan 10 números aleatorios para cada uno; después, se escriben 2 bucles anidados y la estructura se etiqueta con `mult7`. Cuando un dato entero generado es múltiplo de `7`, `break mult7`, hace que la ejecución pase a la siguiente sentencia.

```
class GeneraEnteros
{
  public static void main(String []a)
  {
    int numero;
    System.out.println("\nValores generados aleatoriamente");
    mult7:
    int tope = 11
    while (tope <= (int)Math.pow(11.,4))
    {
      int k = 1;
      while (k <= 10)
      {
        numero = (int)Math.random()*tope + 1;
        System.out.print(numero+" ");
        if (numero % 7 == 0)
            break mult7;
        k++
      }
      System.out.println();
      tope+=11
    }
    System.out.println("Bucles han terminado con número= " +numero);
  }
}
```

En el caso de no especificar etiqueta, el control de la ejecución se transfiere a la sentencia siguiente al bucle o sentencia `switch`, desde el que se ejecuta.

6.2 Repetición: bucle `for`

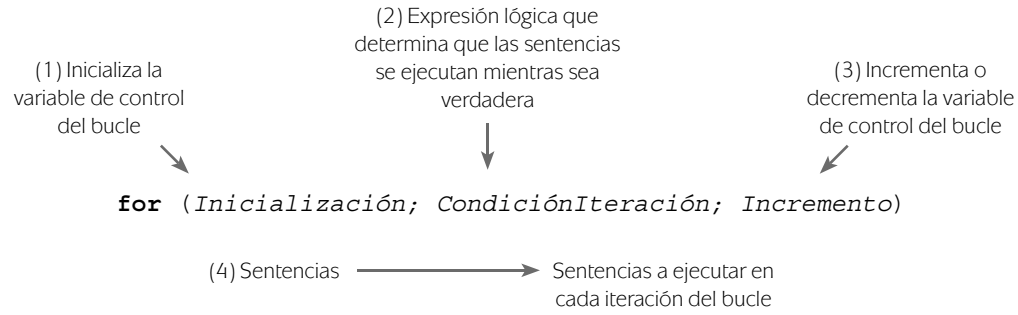
Además de `while`, Java proporciona otros dos tipos de bucles: `for` y `do`; el primero es el más adecuado para implementar conjuntos de sentencias que se ejecutan una vez por

cada valor de un rango especificado; a éstos se les llama *bucles controlados por contador*, y su algoritmo es:

```
por cada valor de una variable_contador de un rango específico:
ejecutar sentencias
```

La sentencia o bucle `for` es la mejor forma de programar la ejecución de un bloque de sentencias un número fijo de veces; éste sitúa las operaciones de control del bucle en la cabecera de la sentencia.

sintaxis



El bucle `for` se compone de:

- Una parte de inicialización que comienza la variable o variables de control; pueden definirse en esta parte y ser simples o múltiples.
- Una parte de condición que contiene una expresión lógica y que itera las sentencias mientras la expresión sea verdadera.
- Una parte que incrementa o decrementa la variable o variables de control del bucle.
- Sentencias o acciones que se ejecutarán por cada iteración del bucle.

La sentencia `for` equivale al siguiente código `while`:

```
inicialización;
while (condiciónIteración)
{
  sentencias del bucle for;
  incremento;
}
```

Por ejemplo:

```
// imprimir Hola 10 veces
for (int i = 0; i < 10; i++)
  System.out.println("Hola!");
```

O, como se indica en seguida:

```
int i;
for (i = 0; i < 10; i++)
```

```
{
  System.out.println("Hola!");
  System.out.println("El valor de i es: " + i);
}
```

EJEMPLO 6.6

Calcular la función $e^x - x$ y escribir los resultados.

```
import java.util.Scanner;
class ValoresFuncion
{
  public static void main(String []a)
  {
    final int VECES = 15;
    Scanner entrada = new Scanner(System.in);
    for (int i = 1; i <= VECES; i++)
    {
      double x, f;
      System.out.print("Valor de x: ");
      x = entrada.nextDouble();
      f = Math.exp(2*x) - x;
      System.out.println("f(" + x + ") = " + f);
    }
  }
}
```

El diagrama de sintaxis de la sentencia `for` se muestra en la figura 6.2.

Existen dos formas de realizar la sentencia `for` que se utilizan normalmente para implementar los bucles de conteo: formato ascendente, en el que la variable de control se incrementa y formato descendente, en el que se decrementa.

```
for (var_control=valor_inicial; var_control<=valor_límite;
                                          exp_incremento)
  sentencia;
```

formato ascendente

formato descendente

```
for (var_control=valor_inicial; var_control>=valor_límite;
                                          exp_decremento)
  sentencia;
```

En seguida se muestra un ejemplo del formato ascendente:

```
for (int n = 1; n <= 10; n++)
  System.out.println("\t" + n + "\t" + n * n );
```

La variable de control es `n` y su valor inicial es 1, mientras que el valor límite es 10 y la expresión de incremento es `n++`; esto significa que el bucle ejecuta la sentencia del cuerpo una vez por cada valor de `n` en orden ascendente de `1` a `10`; en la primera iteración `n` tomará el valor `1`; en la segunda, el valor `2` y así sucesivamente hasta llegar al valor `10`. La salida que se producirá al ejecutarse el bucle será:

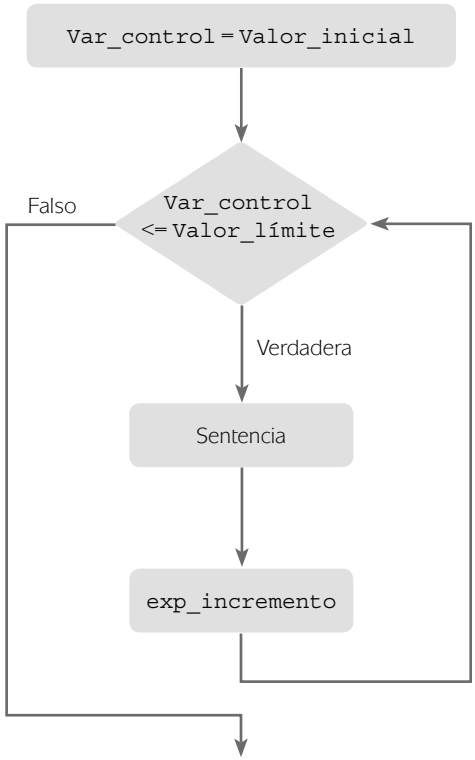


Figura 6.2 Diagrama de sintaxis de un bucle `for`.

```
1      1
2      4
3      9
4      16
5      25
6      36
7      49
8      64
9      81
10     100
```

A continuación se presenta un ejemplo del formato descendente:

```
for (int n = 10; n > 5; n--)
  System.out.println("\t" + n + "\t" + n * n );
```

Su salida es:

```
10     100
9      81
8      64
7      49
6      36
```

debido a que el valor inicial de la variable de control es `10`, y el límite que se ha puesto es `n > 5`; es decir, es verdadera cuando `n = 10, 9, 8, 7, 6`; la expresión de decremento es `n--` que disminuye en `1` el valor de `d` tras la ejecución de cada iteración.

A continuación se muestran otros intervalos de incremento/decremento:

Los rangos de incremento/decremento de la variable o expresión de control del bucle pueden tener cualquier valor y no siempre 1, es decir 5, 10, 20, 4, etcétera, dependiendo de los intervalos necesarios; así, el bucle:

```
for (int n = 0; n < 100; n += 20)
  System.out.println("\t" + n + "\t" + n * n );
```

utiliza la expresión de incremento

```
n += 20
```

que aumenta el valor de n en 20, puesto que equivale a `n = n + 20`; por tanto, la salida que producirá la ejecución del bucle es:

```
0     0
20    400
40    1600
60    3600
80    6400
```

Por ejemplo:

1. Inicializa la variable de control del bucle c al carácter 'A', lo cual equivale a inicializar al entero 65 porque éste es el código ASCII de A e itera mientras el valor de la variable c sea menor o igual que el ordinal del carácter 'Z'. La parte de incremento del bucle aumenta el valor de la variable c en 1; por consiguiente, el bucle se realiza tantas veces como letras mayúsculas se necesiten.

```
for (int c = 'A'; c <= 'Z'; c++)
  System.out.print(c + " ");
System.out.println();
```

2. Muestra un bucle descendente que inicializa la variable de control a 9, el cual se realiza mientras i no sea negativo; como la variable disminuye en 3, el bucle se ejecuta 4 veces con el valor de la variable de control i, 9, 6, 3 y 0.

```
for (int i = 9; i >= 0; i -= 3)
  System.out.println(" " + i*i);
```

3. La variable de control i se inicializa a 1 y se incrementa en múltiplos de 2; por consiguiente, i toma valores de 1, 2, 4, 8, 16, 32, 64 y como el siguiente, 128, no cumple la condición, termina el bucle.

```
for (i = 1; i < 100; i *= 2)
  System.out.println(" " + i);
```

4. Declara 2 variables de control i y j y las inicializa a 0 y la constante MAX; el bucle se ejecutará mientras i sea menor que j; además, i se incrementa en 1, mientras j se decrementa en 1.

```
final int MAX = 25;
int i,j;
for (i = 0, j = MAX; i < j; i++, j--)
  System.out.println("d = " + (i + 2 * j));
```

EJEMPLO 6.7

Suma de los M primeros números pares.

```
class SumaPares
{
  public static void main(String []a)
  {
    final int M = 12;
    int suma = 0;
    for (int n = 1; n <= M; n++)
      suma += 2*n;
    System.out.println("La suma de los " + M +
                       "primeros números pares: " + suma);
  }
}
```

El bucle puede diseñarse con un incremento de 2:

```
for (int n = 2; n <= 2*M; n += 2)
  suma += n;
```

6.2.1 Usos de bucles for

Java permite:

- Que el valor de la variable de control se modifique en valores diferentes de 1.
- Utilizar más de una variable de control.

Las variables de control se pueden incrementar o decrementar en valores de tipo int, pero también es posible hacerlo en valores de tipo float o double y, en consecuencia, se incrementaría o decrementaría en cantidad decimal; por ejemplo:

```
for (int n = 1; n <= 121; n = 2*n + 1)
  System.out.println("n es ahora igual a " + n);

int n, v=9;
for (int n = v; n >= -100; n -= 5)
  System.out.println("n es ahora igual a " + n);

for (double p= 0.75; p<= 5; p += 0.25)
  System.out.println("Perímetro es ahora igual a " + p);
```

Tampoco se requiere que la inicialización de una variable de control sea igual a una constante; ésta se puede inicializar y cambiar en cualquier cantidad que se desee, aunque es natural que cuando la variable de control no sea int, habrá menos garantías de precisión; por ejemplo: el siguiente código muestra un medio más para iniciar un bucle for.

```
double y = 20.0;
for (double x = Math.pow(y,3.0); x > 2.0; x = Math.sqrt(x))
  System.out.println("x es ahora igual a " + x);
```

6.2.2 Precauciones en el uso de `for`

Un bucle `for` se debe construir con precaución, asegurándose que la expresión de inicialización y la de incremento harán que la condición se convierta en `false` en algún momento; en particular "si el cuerpo de un bucle de conteo modifica los valores de cualquier variable implicada en la condición, entonces el número de repeticiones se puede modificar";[1] esta regla es importante porque su aplicación se considera una mala práctica de programación. Es decir, no es recomendable modificar el valor de cualquier variable de la condición del bucle dentro del cuerpo de un bucle `for`, ya que se pueden producir resultados imprevistos; por ejemplo, la ejecución de

```
int limite = 11;
for (int i = 0; i <= limite; i++)
{
  System.out.println(i);
  limite++;
}
```

produce una secuencia infinita de enteros, la cual puede terminar si el compilador tiene constantes `MAXINT` con máximos valores enteros; entonces la ejecución finalizará cuando `i` sea `MAXINT` y `limite` sea `MAXINT+1 = MININT` ya que, a cada iteración, la expresión `limite++` aumenta `limite` en 1, antes de que `i++` incremente `i`.

```
0
1
2
3
.
.
.
```

Como consecuencia, la condición del bucle `i <= limite` siempre es verdadera.

Otro ejemplo de un bucle mal programado es:

```
int limite = 1;

for (int i = 0; i <= limite; i++)
{
  System.out.println(i);
  i--;
}
```

el cual producirá ceros infinitos

```
0
0
0
.
.
```

porque en este caso la expresión `i--` del cuerpo decrementa `i` en `1` antes de que se incremente la expresión `i++` de la cabecera en `1`; como resultado `i` es siempre `0` cuando el bucle se comprueba.

[1] Joyanes, L., *Programación en Java 2*, Madrid, McGraw-Hill, 2002, p. 220.

Éste es otro ejemplo de bucle mal programado, la condición para terminarlo depende de un valor de la entrada:

```
final int LIM = 50
int iter,tope;
for (iter = tope = 0; tope <= LIM; iter++)
{
  System.out.println("Iteración: " + iter);
  tope = entrada.nextInt();
}
```

6.2.3 Bucles infinitos

El objetivo principal de un bucle `for` es implementar bucles de conteo en el que el número de repeticiones se conoce por anticipado; por ejemplo, la suma de enteros de `1` a `n`; sin embargo, existen muchos problemas en los que el número de repeticiones no se puede determinar por anticipado; para lo cual se puede implementar un bucle infinito.


```
for (;;)
  sentencia;
```

La sentencia se ejecuta indefinidamente a menos que se utilice `return` o `break`; aunque normalmente es una combinación `if-break` o `if-return`; la razón de la ejecución indefinida es que se eliminó la expresión de inicialización, la condición y la expresión de incremento, y al no existir una condición específica para terminar la repetición de sentencias, se asume que la condición es verdadera; por ejemplo:

```
for (;;)
  System.out.println("Siempre así, te llamamos siempre así...");
```

producirá la salida

```
Siempre así, te llamamos siempre así...
Siempre así, te llamamos siempre así...
...
```

un número ilimitado de veces, a menos que el usuario interrumpa la ejecución.

Para evitar esto, se requiere que el diseño del bucle `for` sea de la forma siguiente:

1. El cuerpo del bucle debe contener todas las sentencias que se desean ejecutar repetidamente.
2. Una sentencia terminará la ejecución del bucle cuando se cumpla determinada condición.

La sentencia de terminación suele ser `if-break` con la sintaxis:


```
if (condición) break;
```

o bien:

```
if (condición) break label;
```

condición	es una expresión lógica
`break`	termina la ejecución del bucle y transfiere el control a la sentencia siguiente al bucle.
`break label;`	termina la ejecución del bucle y transfiere el control a la sentencia siguiente al bucle con esa etiqueta.

La sintaxis completa:

```
for (;;)        // bucle
{
  lista_sentencias1
  if (condición_terminación) break;
  lista_sentencias2
}               // fin del bucle
```

lista_sentencias	puede ser vacía, simple o compuesta.

Aplicación de un bucle indefinido cuya ejecución se termina al teclear un valor equivalente a `CLAVE`:

```
final int CLAVE = -999;
for (;;)
{
  System.out.println("Introduzca un número " + CLAVE +
                     " para terminar");
  num = entrada.nextInt();
  if (num == CLAVE) break;
  ...
}
```

6.2.4 Los bucles `for` vacíos

Tenga cuidado de situar un punto y coma después del paréntesis inicial de `for`; es decir, el bucle

```
for (int i = 1; i <= 10; i++); ──────────────────────► problema
  System.out.println("Sierra Magina");
```

no se ejecuta correctamente, ni se visualiza la frase "Sierra Magina" 10 veces como se esperaría; tampoco se produce un mensaje de error por parte del compilador.

En realidad, lo que sucede es que se visualiza una vez la frase "Sierra Magina" ya que for es una sentencia vacía pues termina con un punto y coma (;); por tanto, no hace nada durante 10 iteraciones y al terminar se ejecuta la sentencia System.out.println(), y se escribe "Sierra Magina".

6.2.5 Expresiones nulas en bucles for

Cualquiera de las tres o todas las expresiones que controlan un bucle for pueden ser nulas o vacías; el punto y coma (;) es el que marca una expresión vacía. Si la intención es crear un bucle for que actúe exactamente como while, se deben dejar vacías la primera y tercera sentencias; por ejemplo, el siguiente for funciona como while, y tiene vacías las expresiones de inicialización y de incremento:

```
int contador = 0;
for (;contador < 5;)
{
  contador++;
  System.out.print("¡Bucle! ");
}
System.out.println("\n Contador: " + contador);
```

Ejecución

```
¡Bucle! ¡Bucle! ¡Bucle! ¡Bucle! ¡Bucle!
Contador: 5
```

La sentencia for no inicializa ningún valor, pero incluye una prueba de contador < 5 previamente inicializado; tampoco existe una sentencia de incremento, de modo que el bucle se comporta exactamente como la sentencia siguiente:

```
while(contador < 5)
{
  contador++;
  System.out.printl("¡Bucle! ");
}
```

6.2.6 Sentencia continue

Esta sentencia se utiliza en el contexto de un bucle y hace que la ejecución prosiga con la siguiente iteración saltando las sentencias que están a continuación; en bucles while/do-while, la ejecución prosigue con la condición de control del bucle, mientras que en bucles for, la ejecución prosigue con la expresión de incremento. Su sintaxis es la siguiente:

```
continue;
continue etiqueta;
```

EJEMPLO 6.9

Se generan n números aleatorios de forma que se escriban todos excepto los múltiplos de 3.

```
class Multipls
{
```

```
  public static void main(Strings [] a)
  {
    final int CLAVE = 3;
    final int RANGO = 999;
    int n = (int)Math.random()*RANGO +1;
    for (i = 0; i < n; i++)
    {
      int numero;
      numero = (int)Math.random()*RANGO +1;
      if (numero % CLAVE == 0)
      {
        System.out.println();
        continue;
      }
      System.out.print(" " + numero);
    }
  }
}
```

Al generarse un entero múltiplo de `3`, se realiza un `println` vacío para que encuentre un salto de línea y `continue` hace que la ejecución vuelva a la cabecera de `for`, por consiguiente no se escribe el número en la pantalla.

Cuando la sentencia `continue` se ejecuta con etiqueta en una estructura repetitiva, salta las sentencias que quedan hasta el final del cuerpo del bucle y la ejecución prosigue con la siguiente iteración que está en seguida de la etiqueta especificada; en bucles `while/do-while`, la ejecución prosigue con la evaluación de la condición de control que se encuentra después de la etiqueta; por último, en bucles `for` la ejecución prosigue con la expresión de incremento.

EJEMPLO 6.10

El programa `Asteriscos` escribe líneas con asteriscos en una cantidad igual al número de línea correspondiente del código fuente.

```
class Asteriscos
{
  public static void main(Strings [] a)
  {
    final int COLUMNA = 17;
    final int FILA =17

    siguiente:
    for (int f = 1; f <= FILA; f++)
    {
      System.out.println();
      for (int c = 1; c <= COLUMNA; c++)
      {
        if (c > f) continue siguiente;
        System.out.print'*');
      }
    }
  }
}
```

Si se cumple la condición `c > f`, se ejecuta la sentencia `continue siguiente`, de forma que el flujo continúa en la expresión f++ del bucle externo.

6.3 Bucle `for each` (Java 5.0 y Java 6)

Java 5.0 y Java 6.0 incorporan una construcción de bucles potente que permite al programador iterar a través de cada elemento de una colección o de un array sin tener que preocuparse por los valores de los índices; éste es el bucle `for each`, que establece una variable dada a cada elemento de la colección y a continuación ejecuta las sentencias del bloque; su sintaxis es:

sintaxis

```
for (variable : colección)
sentencia;
```

La expresión `colección` debe ser un arreglo o un objeto de una clase que implemente la interfaz `Iterable`, tal como lista de arreglos `ArrayList`, etcétera; por ejemplo, se visualiza cada elemento del arreglo `m` (ver capítulo 10).

```
for (int elemento : m)
  System.out.println(elemento);
```

El bucle se lee así: "para (`for`) cada (`each`) elemento de `m`"; los diseñadores de Java consideraron nombrar el bucle, en primer lugar `foreach` e `in` pero al final optaron por continuar con el código antiguo y la palabra reservada `for` para evitar conflictos con nombres de métodos o variables ya existentes como `System.in`; el cual ya se utilizó en este capítulo.

El efecto anterior del recorrido de colecciones se puede conseguir con un bucle `for` estándar, aunque `for each` es más conciso y menos propenso a errores; un bucle equivalente al anterior es:

```
for (int j = 0; j < m.length; j++)
  System.out.println(m[j]);
```

`for each` es una mejora sustancial sobre el bucle tradicional en caso de necesitar procesar todos los elementos de una colección; en el capítulo 10 "Arreglos" se amplía su concepto.

NOTA

- `for each` se puede interpretar como "por cada valor de... hacer las siguientes acciones".
- Su variable recorre los elementos de la colección o el arreglo y no los valores de los índices.

6.4 Repetición: bucle `do...while`

La sentencia `do-while` se utiliza para especificar un bucle condicional que se ejecuta al menos una vez; cuando se desea realizar una acción determinada al menos una o varias veces, se recomienda este bucle.

sintaxis

Acción (sentencia) a ejecutar al menos una vez

Expresión lógica que determina si la acción se repite

```
1. do sentencia while (expresión)

2. do
     sentencia
   while (expresión)
```

La construcción do comienza ejecutando *sentencia*; en seguida se evalúa *expresión*; si ésta es verdadera, entonces se repite la ejecución de sentencia; continuando hasta que expresión sea falsa.

EJEMPLO 6.11

Bucle para introducir un dígito.

```
do
{
  System.out.println(("Introduzca un dígito (0-9): ");
  digito = System.in.read(); // lee siguiente carácter
  System.in.skip(1);      // salta 1 carácter(fin de línea)

} while ((digito < '0') || (digito > '9'));
```

Este bucle se realiza mientras el carácter leído no sea un dígito.

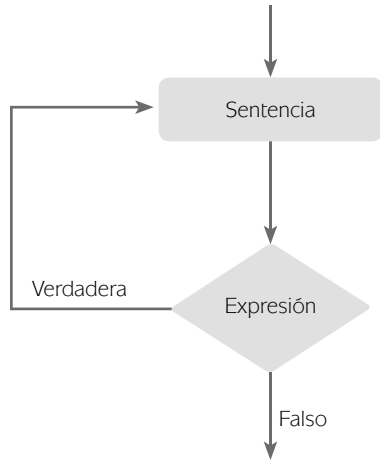


Figura 6.3 Diagrama de flujo de la sentencia do.

EJEMPLO 6.12

Aplicación simple de un bucle while: seleccionar una opción de saludo al usuario dentro de un programa.

```
class Saludo
{
  public static void main(String[] a)
  {
    char opcion;
    do
    {
      System.out.println("Hola");
      System.out.println("¿Desea otro tipo de saludo?");
      System.out.println("Pulse s para si y n para no,");
      System.out.print("y a continuación pulse intro: ");
      opcion = System.in.read();
    }while (opcion == 's'|| opcion == 'S');
    System.out.println("Adiós");
  }
}
```

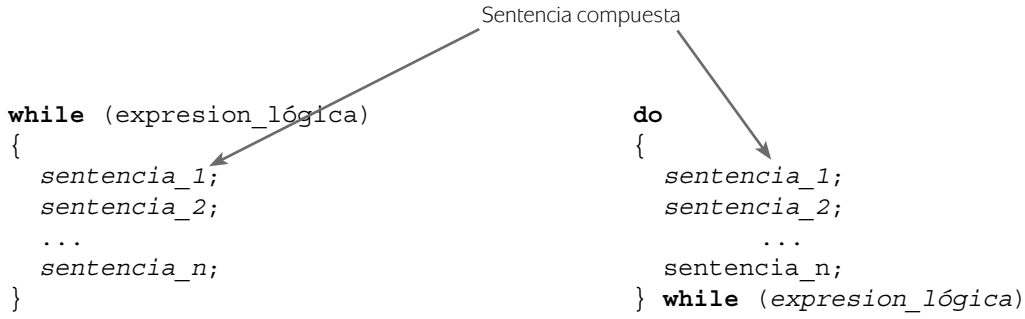
Salida de muestra:

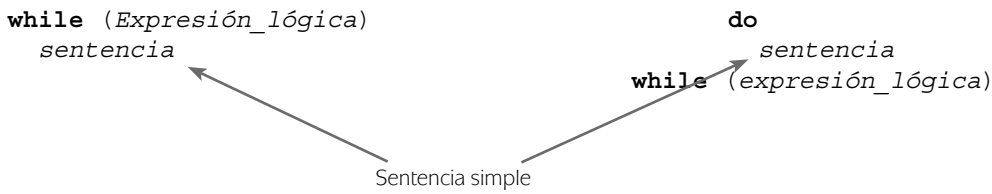
```
Hola
¿Desea otro tipo de saludo?
Pulse s para si y n para no,
y a continuación pulse intro: S
Hola
¿Desea otro tipo de saludo?
Pulse s para si y n para no,
y a continuación pulse intro: N
Adiós
```

6.4.1 Diferencias entre while y do-while

Una sentencia do-while es similar a while excepto que el cuerpo del bucle siempre se ejecuta al menos una vez.

sintaxis





EJEMPLO 6.13

Visualizar las letras minúsculas con los bucles `while` y `do-while`.

```
// bucle do-while
char car = 'a';
do
{
  System.out.print(car + " ");
  car++;
} while (car <= 'z');
```

```
  bucle while
char car = 'a';
while (car <= 'z')
{
  System.out.print(car + " ");
  car++;
}
```

EJEMPLO 6.14

Visualizar las potencias de dos enteros cuyos valores estén en el rango 1 a 1 000 con los bucles `while` y `do-while`.

```
// Realizado con while
pot = 1;
while (pot < 1000)
{
  System.out.println(pot);
  pot * = 2;
} // fin de while
```

```
// Realizado con do-while
pot = 1;
do
{
  System.out.println(pot);
  pot * = 2;
} while (pot < 1000);
```

6.5 Comparación de bucles `while`, `for` y `do-while`

El bucle `for` se utiliza normalmente cuando el conteo está implicado o el número de iteraciones requeridas se pueda determinar al principio de la ejecución del bucle o, simplemente, cuando es necesario seguir el número de veces que un suceso particular ocurre. El bucle `do-while` se ejecuta de un modo similar a `while`, excepto que las sentencias del cuerpo siempre se ejecutan al menos una vez.

La tabla 6.1 describe cuándo se usa cada uno de los tres bucles; `for` es el más utilizado de ellos; y es relativamente fácil reescribir un bucle `do-while` como uno `while`, insertando una asignación inicial de la variable condicional; sin embargo, no todos ellos se pueden expresar así de modo adecuado, ya que un `do-while` se ejecutará siempre al menos una vez mientras `while` puede no ejecutarse; por esta razón `while` suele preferirse a `do-while`, a menos que esté claro que se debe ejecutar una iteración como mínimo.

Tabla 6.1 Formatos de los bucles.

while	Su uso más frecuente es cuando la repetición no está controlada por contador; el test de condición precede a cada repetición del bucle; el cuerpo puede no ser ejecutado. Se debe utilizar cuando se desea saltar el bucle si la condición es falsa.
for	Bucle de conteo que se emplea cuando el número de repeticiones se conoce por anticipado y puede controlarlo un contador; también es adecuado para bucles que implican control no contable del bucle con simples etapas de inicialización y actualización; el test de la condición precede a la ejecución del cuerpo.
do-while	Es apropiado cuando se necesita que el bucle se ejecute al menos una vez.

Ejemplo de la construcción de los tres bucles:

```
cuenta = valor_inicial;
while (cuenta < valor_parada)
{
  ...
  cuenta++;
} // fin de while

for (cuenta = valor_inicial; cuenta < valor_parada; cuenta++)
{
  ...
}  // fin de for

cuenta = valor_inicial;
if (valor_inicial < valor_parada)
do
{
  ...
  cuenta++;
} while (cuenta < valor_parada);
```

6.6 Diseño de bucles

Hay tres puntos a considerar en el diseño de un bucle:

1. El cuerpo del bucle.
2. Las sentencias de inicialización.
3. Las condiciones para su terminación.

6.6.1 Bucles para diseño de sumas y productos

Muchas tareas frecuentes implican leer una lista de números y sumarlos; si se conoce cuántos números habrá, la tarea puede ejecutarse fácilmente con el siguiente algoritmo; donde el valor de la variable `total` es el número de cantidades que se suman; el resultado se acumula en `suma`.

```
suma = 0;
repetir lo siguiente total veces:
  leer(siguiente);
  suma = suma + siguiente;
fin_bucle
```

Este código se implementa fácilmente con un bucle `for`:

```
int suma = 0;
for (int cuenta = 1; cuenta <= total; cuenta++)
{
  siguiente = entrada.nextInt();
  suma = suma + siguiente;
}
```

Se espera que la variable `suma` tome un valor cuando se ejecute la sentencia

```
suma = suma + siguiente;
```

Puesto que `suma` debe tener un valor la primera vez que la sentencia se ejecuta, debe estar inicializada a algún valor antes de ejecutar el bucle; con el objeto de determinar el valor correcto de inicialización, se debe considerar qué sucede después de una iteración del bucle. Después de añadir el primer número, el valor de `suma` debe ser esa cantidad; esto es, el valor de `suma + siguiente` será igual a `siguiente` la primera vez que se ejecute el bucle. Para hacer esta operación, `suma` debe ser inicializado a 0; pero si en lugar de `suma`, se desea obtener productos de una lista de números, la técnica a utilizar es la que se indica en seguida:

```
int producto = 1;
for (int cuenta = 1; cuenta <= total; cuenta++)
{
  siguiente = entrada.nextInt();
  producto = producto * siguiente;
}
```

La variable `producto` debe tener un valor inicial, determinado a `1` en su definición; por eso no se puede suponer que todas las variables se deben inicializar a 0; si `producto` se inicializara a 0, seguiría siendo 0 después de que el bucle anterior terminara.

6.6.2 Fin de un bucle

Existen cuatro métodos para terminar un bucle de entrada:

1. Alcanzar el tamaño de la secuencia de entrada.
2. Preguntar antes de la iteración.
3. Terminar la secuencia de entrada con un valor centinela.
4. Agotar la entrada.

Tamaño de la secuencia de entrada

Si su programa puede determinar el tamaño de la secuencia de entrada por anticipado, ya sea preguntando al usuario o por algún otro método, puede utilizar un bucle *repetir* `n` *veces* para leer la entrada exactamente esa cantidad de veces, siendo `n` el tamaño de la secuencia.

Preguntar antes de la iteración

El segundo método para la terminación de un bucle de entrada es preguntar al usuario después de cada iteración del bucle si se iterará o no de nuevo; por ejemplo:

```
int numero, suma = 0;
char resp = ´S´;
while ((resp == ´s´|| (resp == ´S´))
{
  System.out.print("Introduzca un número: ");
  numero = entrada.nextInt();
  suma += numero;
  System.out.print("¿Existen más números?(S para Si, N para No): ");
  resp = System.in.read();
}
```

Este método es tedioso para listas grandes de números; en cuyo caso es preferible incluir una única señal de parada, como en el método siguiente.

Valor centinela

El método más práctico y eficiente para terminar un bucle que lee una lista de valores del teclado es con un valor centinela; al ser totalmente distinto a todos los valores posibles de la lista que se lee indica el final de la misma; un ejemplo común se presenta al leer una lista de números positivos y utilizar un número negativo como valor centinela que indique el final de la lista.

```
// Ejemplo de valor centinela (número negativo)
System.out.println("Introduzca una lista de enteros positivos");
System.out.println("Termine la lista con un número negativo");
suma = 0;
numero = entrada.nextInt();
while (numero >= 0)
{
  suma += numero;
  numero = entrada.nextInt();
}
System.out.println("La suma es: " + suma);
```

Si al ejecutar el segmento de programa anterior se introduce la lista

```
4
8
15
-99
```

el valor de la suma será `27` porque `-9`, el último número de la entrada de datos, no se añade a la suma al actuar como centinela; éste no forma parte de la lista de entrada de números.

Agotar la entrada

Cuando se leen las entradas de un archivo, el método más frecuente es comprobar simplemente si todas ellas fueron procesadas; el final del bucle se alcanza cuando éstas se agotan. Éste es el método usual en la lectura de archivos, utiliza la marca `eof` al final; en el capítulo de archivos se dedicará atención especial a esta acción.

EJEMPLO 6.15

Escribir un programa que visualice el factorial de un entero comprendido entre 2 y 20

El factorial de un entero `n` se calcula con un bucle `for` desde `2` hasta `n`, teniendo en cuenta que el de `1` es `1 (1! = 1)` y que `n! = n*(n-1)!`; por ejemplo:

```
4! = 4*3! = 4*3 2! = 4*3*2*1! = 4*3*2*1 = 24
```

En el programa se escribe un bucle `do-while` para validar la entrada de `n`, entre `2` y `20`, y otro bucle `for` para calcular el factorial; como ejemplo, éste se diseña con una sentencia vacía, en la expresión de incremento se calculan los `n` productos con el operador `*=` junto al de decremento `(--)`.

```
import java.util.Scanner;
class Factorial
{
  public static void main(String[] a)
  {
    long int n,m,fact;
    Scanner entrada = new Scanner(System.in);
    do
    {
      System.out.println("\nFactorial de número n, entre 2 y 20: ");
      n = entrada.nextLong();
    } while ((n < 2) || (n > 20));

    for (m = n, fact = 1; n > 1 ; fact *= n--);

    System.out.println(m + "! = " + fact);
  }
}
```

6.7 Bucles anidados

Es posible anidar bucles, los cuales tienen un bucle externo y uno o más internos; cada vez que se repite el externo, los internos también lo hacen; los componentes de control se vuelven a evaluar y se ejecutan todas las iteraciones requeridas.

EJEMPLO 6.16

La siguiente aplicación muestra una tabla de multiplicación que calcula y visualiza productos de la forma `x * y`, para cada `x` en el rango de `1` a `xultimo` y desde cada `y` en el rango `1` a `yultimo`; en este caso `xultimo` y `yultimo` son enteros prefijados. La tabla que se desea obtener es:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
        ...
```

```
for (x = 1; x <= xultimo; x++)

  for (y = 1; y <= yultimo; y++)
  {
    int producto;
    producto = x * y;
    System.out.println(x + "*" + y "=" producto);
  }
```

Bucle externo

Bucle interno

El bucle que tiene x como variable de control se denomina externo y al que tiene y como variable de control se le llama interno.

EJEMPLO 6.17

Escribir las variables de control de dos bucles anidados.

```
System.out.println("\n\t\t i \t j"); //cabecera de salida
for (int i = 0; i < 4; i++)
{
  System.out.println("Externo\t " + i);
  for (int j = 0; j < i; j++)
       System.out.println("Interno\t\t " + j);
} // fin del bucle externo
```

La salida del programa es:

```
          i    j
Externo   0
Externo   1
Interno        0
Externo   2
Interno        0
Interno        1
Externo   3
Interno        0
Interno        1
Interno        2
```

EJEMPLO 6.18

Escribir una aplicación que visualice el siguiente triángulo isósceles:

```
        *
       ***
      *****
     *******
    *********
```

Se construye mediante un bucle externo y dos internos; estos últimos se ejecutan cada vez que el primero se repite; el externo se repite cinco veces, una por cada fila; el número de repeticiones que realizan los internos se basa en el valor de la variable `fila`. El primer bucle interno visualiza los espacios en blanco no significativos; el segundo, uno o más asteriscos.

```
class Triangulo
{
  public static void main(String[] a)
  {
    final int NUMLINEAS = 5;
    final char BLANCO = '';
    final char ASTERISCO = '*';

    System.out.println(" ");

      // bucle externo: dibuja cada línea
    for (int fila = 1; fila <= NUMLINEAS; fila++)
    {
      System.out.print("\t");
        //primer bucle interno: escribe espacios
      for (int blancos = NUMLINEAS - fila; blancos > 0;
                                              blancos--)
        System.out.print(BLANCO);

      for (int cuenta_as = 1; cuenta_as < 2 * fila;
                                          cuenta_as++)
        System.out.print(ASTERISCO);

      System.out.println(" ");
    } // fin del bucle externo
  }
}
```

El bucle externo se repite 5 veces, uno por línea o fila; el número de repeticiones ejecutadas por los internos se basa en el valor de `fila`. La primera fila consta de un asterisco y 4 blancos, la segunda está conformada por 3 blancos y 3 asteriscos, y así sucesivamente; la quinta tendrá 9 asteriscos (2 * 5 – 1).

EJEMPLO 6.19

Escribir un programa que calcule y escriba en pantalla el factorial de `n`, entre los valores 1-10.

Con 2 bucles `for` se soluciona el problema; uno externo que determina el número `n` cuyo factorial se calcula en el interno.

```
class Factorial
{
  public static void main(String[] a)
  {
    final int N= 10;
    for (int n = 1, n <= N; n++)
    {
      long fact;
```

```
        int m;
        fact = 1;
        for (m = n ; m > 1; m--)
          fact *= m;
        System.out.println(n + "! = " + fact);
      }
    }
}
```

6.6 Transferencia de control: sentencias `break` y `continue`

Después de estudiar y comparar los tres tipos de bucles, y considerar sus elementos de diseño, vamos a reconsiderar la transferencia de control del flujo. Aunque, en apartados anteriores, se estudiaron las sentencias `break` y `continue` que realizan esta tarea en bucles,[2] ahora las sintetizaremos, complementando lo que antes explicamos.

Normalmente, como se ha visto, el uso de un tipo de bucle depende del número de iteraciones que se realizan dentro del mismo; así, en caso de conocer por adelantado el número de repeticiones necesarias, o que el propio programa pueda determinarlas, el bucle idóneo es `for`. Si no se conoce el número de iteraciones y el programa no puede determinar cuántas son necesarias, la elección adecuada es `while`; por último, si no se conocen las repeticiones, ni el programa las puede determinar por adelantado pero al menos el bucle debe ejecutar una iteración, entonces la decisión correcta es `do-while`.

6.6.1 Sentencia `break`

La sentencia **`break`**, ya estudiada antes, normalmente realiza dos acciones:

- La salida inmediata de un bucle.
- Saltar el resto de la sentencia `switch`.

En cualquiera de los casos, cuando se ejecuta la sentencia `break`, el flujo de control del programa continua en la siguiente sentencia después del bucle o de la estructura `switch`. Existen dos modalidades que ya se estudiaron: `break` y `break` con etiqueta.

EJEMPLO 6.20

Aplicación de `break`.

```
while (anyos <= 50)
{
  saldo += pago;
  double interes = saldo * tasaInteres / 100;
  saldo += interes;

  if (saldo >= limite) break;
  anyos++;
}
```

[2] Aunque Java mantiene la clásica sentencia `goto`, típica de programación estructurada, para transferencia y roturas de flujo de control, su uso no se recomienda y por ello no se menciona en el libro.

La salida del bucle se produce cuando `anyos` (años) es mayor que 50 o cuando `saldo >= limite` es condición de la sentencia `if`; sin el uso de `break`, el bucle anterior sería:

```
while (anyos <= 50 && saldo > limite)
{
  saldo += pago;
  double interes = saldo * tasaInteres / 100;
  saldo += interes;

  if (saldo < limite)
    anyos++;
}
```

Como se puede observar, la diferencia reside en el hecho de que `saldo > limite` se repite dos veces: una en la condición de salida del bucle `while` y otra en la expresión de la condición `if`.

EJEMPLO 6.21

A continuación se muestra otra aplicación de `break`:

```
for (int i = 1; i <= 25; i++)
{
  d = leerDistancia(i)
  if (d == 0) // salida de bucle
    break;
  System.out.println ("Distancia: ", d);
}
```

La segunda modalidad de `break` es la sentencia `break` con etiqueta:

```
explorarTablaCalificaciones:
for (int m = 0; m < longitud; m++)
{
  for (int n = 1; n <= 2; n++)
  {
    System.out.println ("M: " + m + " N: " + c);
    breakexplorarTablaCalificaciones; // salida bucle
  }
}
```

EJEMPLO 6.22

En seguida mostramos otra aplicación de `break`:

```
// declaraciones
static Scanner consola = new Scanner(System.in)

int suma;
int num;

boolean esNegativo;
```

```
suma = 0;
while (consola.hasNext( ))
{
  num = consola.nextInt( );
  if (num < 0) // valor negativo se termina el bucle
  {
    System.out.println("Número negativo encontrado");
    break;
  }
  suma = suma + num;
}
```

En el bucle `while`, cuando se encuentra un número negativo, la expresión de la sentencia `if` se evalúa a verdadero; después se imprime un mensaje específico y `break` termina el bucle.

NOTA

Las etiquetas se pueden aplicar a cualquier sentencia, incluso a `if` o un bloque de sentencias tales como:

```
etiqueta:
{
  ...
  if (condición) break
   etiqueta; // salida bucle
  ...
}
// se transfiere el control
   aquí al ejecutar la
   sentencia break
```

Comentario. Sin embargo, este sistema de transferencia de control incondicional debe utilizarse con mucha precaución ya que no es una buena práctica de programación.

6.6.2 Sentencia `continue`

La sentencia `continue` se utiliza en los tres tipos de bucles; cuando se ejecuta en un bucle, se saltan las sentencias restantes y se prosigue con la siguiente iteración. En una sentencia `while, do-while`, se evalúa la expresión lógica inmediatamente después de `continue` y después se ejecuta la expresión lógica; por ejemplo:

ADVERTENCIA

Las sentencias `break` y `continue` deben utilizarse con mucha precaución ya que su uso puede resultar confuso; además, la lógica de un programa se puede realizar sin necesidad de estas sentencias. En esta obra, normalmente, no se utilizan.

```
for (cuenta = 1; cuenta <= 100; cuenta++)
{
  System.out.print ("Introduzca un número, -1, Salir: ");
  n = en.nextIn( );
  if (n < 0) continue;
  suma += n: // no se ejecuta si n < 0
}
```

En este caso, la sentencia `continue` salta a la sentencia `cuenta++` de la expresión del bucle `for`.

En programación es habitual tener que repetir la ejecución de una sentencia, lo cual puede realizarse por medio de un bucle; este último es un grupo de instrucciones que se ejecutan reiteradamente hasta que se cumple una condición de terminación. Los bucles representan estructuras de control repetitivas y su número puede establecerse inicialmente o depender de la condición verdadera o falsa.

Un bucle se puede diseñar de diversas formas, las más importantes son: repetición controlada por contador y repetición controlada por condición:

- Una variable de control del bucle se utiliza para contar las repeticiones de un grupo de sentencias; se incrementa o decrementa normalmente en 1 cada vez que el grupo se ejecuta.
- La condición de finalización de bucle se utiliza para controlar la repetición cuando las iteraciones no se conocen por adelantado; un valor centinela se introduce para establecer el hecho que determinará si se cumple o no la condición.

- Los bucles `for` inicializan una variable de control a un valor y enseguida comprueban una expresión; si es verdadera, se ejecutan las sentencias del cuerpo del bucle. La expresión se comprueba cada vez que termina la iteración; cuando es falsa, se termina y la ejecución sigue en la siguiente sentencia. Es importante que en el cuerpo del bucle haya una sentencia que haga que alguna vez sea falsa la expresión que controla la iteración del bucle.
- Los bucles `while` comprueban una condición; si es verdadera, se ejecuta las sentencias del bucle. Después se vuelve a comprobar la condición; si sigue siendo verdadera, se ejecutan las sentencias del bucle. Esto finaliza cuando la condición es falsa. La condición está en la cabecera del bucle `while`, por eso el número de veces que se repite puede ser entre `0` y `n`.
- Los bucles `do while` también comprueban una condición; se diferencian de `while` en que comprueban la condición al final, en vez de la cabecera.
- La sentencia `break` produce la salida inmediata del bucle.
- Cuando la sentencia `continue` se ejecuta todas las sentencias siguientes se saltan y, si se cumple la condición del bucle, comienza una nueva iteración.

conceptos clave

- Bucle.
- Comparación entre `while`, `for` y `do`.
- Control de bucles.
- Iteración/repetición.
- Optimización de bucles.
- Sentencia `break`.
- Sentencia `do-while`.
- Sentencia `for`.
- Sentencia `while`.
- Terminación de un bucle.

ejercicios

6.1 ¿Cuál es la salida del siguiente segmento de programa?

```
for (cuenta = 1; cuenta < 5; cuenta++)
  System.out.println((2 * cuenta));
```

6.2 ¿Cuál es la salida de los siguientes bucles?

a)
```
for (n = 10; n > 0; n = n-2)
  {
    System.out.println("Hola");
    System.out.println(n);
    }
```

b)
```
double n = 2;
for (; n > 0; n = n-0.5)
   System.out.println(n);
```

6.3 Seleccionar y escribir el bucle adecuado para resolver las siguientes tareas:

a) Suma de la serie `1/2+1/3+1/4+1/5+...+1/50`.
b) Lectura de la lista de calificaciones de un examen de Historia.
c) Visualizar la suma de enteros en el intervalo `11...50`.

6.4 Considerar el siguiente código de programa:

```
int i = 1;
```

```
while (i <= n) {
  if ((i % n) == 0) {
    ++i;
  }
}
System.out.println(i);
```

a) ¿Cuál es la salida si n es 0?
b) ¿Cuál es la salida si n es 1?
c) ¿Cuál es la salida si n es 3?

6.5 A partir del siguiente código de programa:

```
for (i = 0; i < n; ++i) {
--n;
}
System.out.println(i);
```

a) ¿Cuál es la salida si n es 0?
b) ¿Cuál es la salida si n es 1?
c) ¿Cuál es la salida si n es 3?

6.6 ¿Cuál es la salida de los siguientes bucles?

```
for (int n = 1; n <= 10; n++)
for (int m = 10; m >= 1; m--)
  System.out.println("n= " + n + " " + m +" n*m = " + n * m );
```

6.7 Escribir un programa que calcule y visualice

```
1! + 2! + 3! + ... + (n-1)! + n!
```

donde n es un dato de entrada.

6.8 ¿Cuál es la salida del siguiente bucle?

```
suma = 0;
while (suma < 100)
  suma += 5;
System.out.println(suma);
```

6.9 Escribir un bucle `while` que visualice todas las potencias de un entero n, menores que un valor MAXLIMITE.

6.10 ¿Qué hace el siguiente bucle `while`? Reescribirlo con sentencias `for` y `do-while`.

```
num = 10;
while (num <= 100)
{
  System.out.println(num);
  num += 10;
}
```

6.11 Suponiendo que `m = 3` y `n = 5`, ¿cuál es la salida de los siguientes segmentos de programa?

a)
```
for (i = 0; i < n; i++)
  {
    for (j = 0; j < i; j++)
         System.out.print("*");
    System.out.println();
  }
```

b)
```
for (i = n; i > 0; i--)
{
 for (j = m; j > 0; j--)
   System.out.print("*");
 System.out.println();
}
```

6.12 ¿Cuál es la salida de los siguientes bucles?

a)
```
for (i = 0; i < 10; i++)
 System.out.println("2* " + i +" = " 2*i);
```

b)
```
for (i = 0; i <= 5; i++)
     System.out.println((2*i+1));
System.out.println();
```

c)
```
for (i = 1; i < 4; i++)
{
  System.out.println(i);
  for (j = i; j >= 1; j--)
    System.out.println(j);
}
```

6.13 Describir la salida de los siguientes bucles:

a)
```
for (i = 1; i <= 5; i++)
{
  System.out.println(i);
  for (j = i; j >= 1; j-=2)
    System.out.println(j);
}
```

b)
```
for (i = 3; i > 0; i--)
  for (j = 1; j <= i; j++)
    for (k = i; k >= j; k--)
     System.out.println(i+j+k));
```

c)
```
for (i = 1; i <= 3; i++)
  for (j = 1; j <= 3; j++)
  {
    for (k = i; k <= j; k++)
      System.out.println(i + " " + " " + j + " " + k);
  System.out.println();
}
```

6.14 ¿Cuál es la salida de este bucle?

```
i = 0;
while (i*i < 10)
{
  j = i;
  while (j*j < 100)
  {
    System.out.println((i+j);
    j *= 2;
  }
  i++;
}
System.out.println("\n*****");
```

problemas

6.1 Escribir un programa que visualice la siguiente salida:

```
1
1       2
1       2       3
1       2       3       4
1       2       3
1       2
```

6.2 Diseñar e implementar un programa que solicite a su usuario un valor no negativo `n` y visualice la siguiente salida:

```
1       2       3 ..... n-1     n
1       2       3 ..... n-1
...
1       2       3
1       2
1
```

6.3 Implementar el algoritmo de Euclides que encuentre el máximo común divisor de dos números enteros y positivos.

Algoritmo de Euclides de m y n:
Éste transforma un par de enteros positivos (`m`, `n`) en una par (`d`, `o`), dividiendo repetidamente el entero mayor entre el menor y reemplazando con el resto; cuando el resto es 0, el otro entero de la pareja será el máximo común divisor de la pareja original.

Ejemplo mcd (532 112)

		4	1	3 →	Cocientes
	532	112	84	28	
Restos	84	28	00	↓	
					mcd = 28

6.4 En una empresa de computadoras, los salarios de los empleados se aumentarán según su contrato actual:

Contrato	**Aumento %**
0 a 9 000 dólares	20
9 001 a 15 000 dólares	10
15 001 a 20 000 dólares	5
más de 20 000 dólares	0

Escribir un programa que solicite el salario actual de cada empleado y que, además, calcule y visualice el nuevo salario.

6.5 La constante *pi* (3.141592) se utiliza en matemáticas; un método sencillo de calcular su valor es:

$$Pi = 2 * \frac{2}{1} * \frac{2}{3} * \frac{4}{3} * \frac{4}{5} * \frac{6}{5} * \frac{6}{7} * \frac{8}{7} * \frac{8}{9} \quad \frac{2n}{2n-1} \; \frac{2n}{2n-1}$$

Escribir un programa que efectúe este cálculo con un número de términos especificados por el usuario.

6.6 Escribir un programa que determine y escriba la descomposición factorial de los números enteros comprendidos entre 1 900 y 2 000.

6.7 Escribir un programa que encuentre los tres primeros números perfectos pares y los tres primeros números perfectos impares.

Un número perfecto es un entero positivo que es igual a la suma de todos los enteros positivos (excluido él mismo) que son sus divisores. El primer número perfecto es 6, ya que sus divisores son 1, 2, 3 y 1 + 2 + 3 = 6.

6.8 El valor de e^x se puede aproximar por la suma siguiente:

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

Escribir un programa que tome un valor de *x* como entrada y visualice la suma para cada uno de los valores de `n` comprendidos entre 1 a 100.

6.9 Escribir un programa que encuentre el primer número primo introducido por medio del teclado.

6.10 Calcular la suma de la serie `1/1 + 1/2 + ... + 1/`*N* donde *N* es un número que se introduce por teclado.

6.11 Calcular la suma de los términos de la siguiente serie:

$$1/2 + 2/2^2 + 3/2^3 + \ldots + n/2^n$$

6.12 Encontrar un número natural *N* más pequeño de forma que la suma de los *N* primeros números exceda una cantidad introducida por el teclado.

6.13 Escribir un programa para mostrar mediante bucles los códigos ASCII de las letras mayúsculas y minúsculas.

6.14 Encontrar y mostrar todos los números de cuatro cifras que cumplan la condición de que la suma de las cifras de orden impar es igual a la suma de las cifras de orden par.

6.15 Calcular todos los números de tres cifras tales que la suma de los cubos de las cifras es igual al valor del número.