UNIDAD 3

Bucles

Objetivos

- Conocer y utilizar las estructuras de repetición.
- Programar aplicaciones que repiten conjuntos de instrucciones mediante el uso de bucles.
- Distinguir entre un bucle controlado por contador, un bucle precondición y un bucle poscondición.
- Utilizar las estructuras adecuadas de control para conseguir que un programa funcione según las especificaciones funcionales.

Contenidos

Introducción

Un bucle es un tipo de estructura que contiene un bloque de instrucciones que se ejecuta repetidas veces; cada ejecución o repetición del bucle se llama *iteración.*

El uso de bucles simplifica la escritura de programas, minimizando el código duplicado. Cualquier fragmento de código que sea necesario ejecutar varias veces seguidas es susceptible de incluirse en un bucle. Java dispone de los bucles: `while`, `do-while` y `for`.

3.1. Bucles controlados por condición

El control del número de iteraciones se lleva a cabo mediante una condición. Si la evaluación de la condición es cierta, el bucle realizará una nueva iteración.

3.1.1. while

Al igual que la instrucción `if`, el comportamiento de `while` depende de la evaluación de una condición. El bucle `while` decide si realizar una nueva iteración basándose en el valor de la condición. Su sintaxis es:

```
while (condición) {
   bloque de instrucciones
   ...
}
```

El comportamiento de este bucle (véase Figura 3.1) es:

1. Se evalúa `condición`.
2. Si la evaluación resulta `true`, se ejecuta el bloque de instrucciones.
3. Tras ejecutarse el bloque de instrucciones, se vuelve al primer punto.
4. Si, por el contrario, la condición es `false`, terminamos la ejecución del bucle.

Por ejemplo, podemos mostrar los números del 1 al 3 mediante un bucle `while` controlado por la variable `i`, que empieza valiendo 1, con la condición `i <= 3`:

```
int i = 1; //valor inicial
while (i <= 3) { //el bucle iterará mientras i sea menor o igual que 3
   System.out.println(i); //mostramos i
   i++; //incrementamos i
}
```

Veamos una traza de la ejecución:

1. Se declara la variable `i` y se le asigna el valor 1.
2. La instrucción `while` evalúa la condición (`i <= 3`): ¿es $1 \leq 3$? Cierto.
3. Se ejecuta el bucle de instrucciones: `System.out.println` e `i++`. Ahora la `i` vale 2.

4. La instrucción `while` vuelve a evaluar la condición: ¿es 2 ≤ 3? Cierto.
5. Se ejecuta el bloque de instrucciones: `System.out.println` e `i++`. Ahora la `i` vale 3.
6. La instrucción `while` vuelve a evaluar la condición: ¿es 3 ≤ 3? Cierto.
7. Se ejecuta el bloque de instrucciones: `System.out.println` e `i++`. Ahora la `i` vale 4.
8. La instrucción `while` vuelve a evaluar la condición: ¿es 4 ≤ 3? Falso.
9. Se termina el bucle y se pasa a ejecutar la instrucción siguiente.

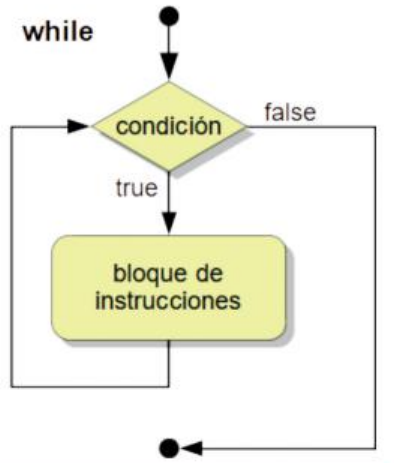


Figura 3.1. *Bucle* `while`.

Un bucle `while` puede realizar cualquier número de iteraciones, desde cero, cuando la primera evaluación de la condición resulta falsa, hasta infinitas, en el caso de que la condición sea siempre cierta. Esto es lo que se conoce como *bucle infinito.*

Veamos un ejemplo de un bucle `while` que nunca llega a ejecutarse:

```
int cuentaAtras = -8; //valor negativo
   while (cuentaAtras >= 0) {
   ...
}
```

En este caso, independientemente del bloque de instrucciones asociado a la estructura `while`, no se llega a entrar nunca en el bucle, debido a que la condición no se cumple ni siquiera la primera vez. Se realizan cero iteraciones. En cambio,

```
int cuentaAtras = 10;
while (cuentaAtras >= 0) {
   System.out.println(cuentaAtras);
}
```

Dentro del bloque de instrucciones no hay nada que modifique la variable `cuentaAtras`, lo que hace que la condición permanezca idéntica, evaluándose siempre `true` y haciendo que el bucle sea infinito.

Actividad propuesta 3.1

Diseña una aplicación que muestre la edad máxima y mínima de un grupo de alumnos. El usuario introducirá las edades y terminará escribiendo un −1.

Actividad resuelta 3.1

Diseñar un programa que muestre, para cada número introducido por teclado, si es par, si es positivo y su cuadrado. El proceso se repetirá hasta que el número introducido sea 0.

Solución

```
import java.util.Scanner;
/* No tenemos la certeza de cuántos números se introducirán por teclado, por eso,
 * el bucle while se ejecutará mientras que el número introducido no sea 0.
 * El bloque de instrucciones del bucle estará formado por las sentencias que muestran
 * si el número es par, positivo y su cuadrado. */
public class Main {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      boolean esPar, esPositivo; //indicadores para el informe
      System.out.print("Introduzca número: ");
      int num = sc.nextInt(); //leemos el número
      while (num != 0) { //repetimos mientras el número leído no sea 0
         //si divido un número entre 2 y obtengo como resto 0, significa que es par
         //el operador % (resto módulo) calcula el resto. Así sabremos la paridad
         esPar = num % 2 == 0 ? true : false; // si el resto es 0, será par
         esPositivo = num >= 0 ? true : false; //consideramos el 0 positivo
         System.out.println("Es par?: " + esPar + "\nEs positivo?: " + esPositivo);
         System.out.println("Cuadrado: " + num * num);
         System.out.print("Introduzca otro número: ");
         num = sc.nextInt(); // volvemos a leer num
      }
   }
}
```

Actividad resuelta 3.2

Implementar una aplicación para calcular datos estadísticos de las edades de los alumnos de un centro educativo. Se introducirán datos hasta que uno de ellos sea negativo, y se mostrará: la suma de todas las edades introducidas, la media, el número de alumnos y cuántos son mayores de edad.

Solución

```
import java.util.Scanner;
/* Desconocemos cuántas edades se van a utilizar como datos, el bucle while se
 * ejecutará mientras la edad introducida no sea negativa.
 * En cada iteración acumularemos la edad, incrementaremos un contador para llevar
 * la cuenta de las edades introducidas y, si el alumno es mayor de edad,
 * incrementaremos el contador de alumnos mayores de edad.
 * Cuando salgamos del bucle mostraremos los datos y calcularemos la media. */
public class Main {
 public static void main(String[] args) {
  Scanner sc = new Scanner(System.in);
  int sumaEdades = 0; //acumulará la suma de todas las edades
  int contadorAlumnos = 0, //contador de alumnos (o de edades introducidas)
```

```
        contadorMayorEdad = 0; //contador del número de alumnos mayores de edad
        double media; //media de las edades
        System.out.print("Introduzca edad: ");
        int edad = sc.nextInt(); //leemos la edad
        while (edad >= 0) { //repetimos mientras la edad no sea negativa
           sumaEdades += edad; //acumulamos la edad introducida
           contadorAlumnos++; //incrementamos, se ha introducido la edad de un
                              //alumno más
           if (edad >= 18) { //si la edad introducida corresponde a un mayor de edad
              contadorMayorEdad++; //incrementamos, ahora hay un mayor de edad más
           }
           System.out.print("Introduzca edad: ");
           edad = sc.nextInt(); // volvemos a leer
        }
        media = (double) sumaEdades/contadorAlumnos; //con el cast la división es real
                                                     //mostramos el informe estadístico
        System.out.println("Suma de todas las edades: " + sumaEdades);
        System.out.println("Media: " + media);
        System.out.println("Número total de alumnos: " + contadorAlumnos);
        System.out.println("Mayores de edad: " + contadorMayorEdad);
    }
}
```

Actividad resuelta 3.3

Codificar el juego «el número secreto», que consiste en acertar un número entre 1 y 100 (generado aleatoriamente). Para ello se introduce por teclado una serie de números, para los que se indica: «mayor» o «menor», según sea mayor o menor con respecto al número secreto. El proceso termina cuando el usuario acierta o cuando se rinde (introduciendo un –1).

Solución

```
import java.util.Scanner;
/* La aplicación generará un número aleatorio entre 1 y 100. A continuación el
 * jugador irá probando suerte con la ayuda de las indicaciones que la propia
 * aplicación le ofrece. El juego termina cuando acierta o cuando se rinde
 * (introduciendo un -1). */
public class Main {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int numSecreto = (int) (Math.random() * 100 + 1); //número aleatorio entre
           //1 y 100
      System.out.print("Introduzca un número entre 1 y 100: ");
      int num = sc.nextInt();
      while (numSecreto != num && //mientras no acertemos (son distintos)
             num != -1) { // y no introduzcamos un -1
         if (numSecreto < num) { //el número secreto es menor
           System.out.println("Menor");
         } else { //en otro caso, será mayor
           System.out.println("Mayor");
```

```
            }
            System.out.print("Introduzca otro número: ");
            num = sc.nextInt();
        }
        //salimos del bucle porque el jugador acierta el número o se rinde
        if (numSecreto == num) {
            System.out.println("Enhorabuena...");
        } else {
            System.out.println("Otra vez será...");
        }
    }
}
```

Actividad resuelta 3.4

Un centro de investigación de la flora urbana necesita una aplicación que muestre cuál es el árbol más alto. Para ello se introducirá por teclado la altura (en centímetros) de cada árbol (terminando la introducción de datos cuando se utilice −1 como altura). Los árboles se identifican mediante etiquetas con números únicos correlativos, comenzando en 0. Diseñar una aplicación que resuelva el problema planteado.

Solución

```
import java.util.Scanner;
/* Introducimos la altura de cada árbol dentro de un bucle y guardaremos la mayor y el
 * número de etiqueta del árbol al que corresponde.
 * En la búsqueda del máximo (o mínimo) se nos plantea un problema: con qué valor
 * inicializamos el máximo. Hemos de inicializar el máximo con un valor menor o
 * igual que todos los valores con los que trabajaremos.
 * máximo es -2. Si inicializamos arbitrariamente máximo=0, como 0 es mayor que
 * cualquier valor del conjunto, el algoritmo dirá que el máximo es 0 (error).
 * En este caso, al trabajar con alturas (positivas), podemos inicializar sin
 * problema a 0 (es menor que cualquier positivo). Sin embargo, en el caso
 * general, la mejor opción es inicializar el máximo al primer valor leído.*/
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int etiquetaArbolMasAlto; //número identificativo del árbol más alto
        int alturaArbolMasAlto; //altura del árbol más alto
        int etiqueta = 0; //número identificativo del árbol del que se piden los datos
        int altura; //altura del árbol del que se piden los datos
        System.out.print("Altura del árbol (" + etiqueta + "): ");
        altura = sc.nextInt();
        alturaArbolMasAlto = altura; //el primer árbol será, por ahora, el más alto
        etiquetaArbolMasAlto = 0; //el árbol con etiqueta 0 es, por ahora, el más alto
        while (altura != -1) {
            if (altura > alturaArbolMasAlto) { //hemos encontrado un árbol más alto
              alturaArbolMasAlto = altura;
              etiquetaArbolMasAlto = etiqueta;
            }
            etiqueta++; //incrementamos la etiqueta, para solicitar la altura del
                        //siguiente
            System.out.print("Altura del árbol (" + etiqueta + "): ");
            altura = sc.nextInt();
```

```
        }
        if (alturaArbolMasAlto == -1) {
           System.out.println("No hay ningún árbol");
        } else {
           System.out.println("El árbol más alto mide: " + alturaArbolMasAlto);
           System.out.println("Etiqueta del árbol: " + etiquetaArbolMasAlto);
        }
     }
}
```

3.1.2. do-while

Disponemos de un segundo bucle controlado por una condición: el bucle `do-while`, muy similar a `while`, con la diferencia de que primero se ejecuta el bloque de instrucciones y después se evalúa la condición para decidir si se realiza una nueva iteración. Su sintaxis es:

```
do {
   bloque de instrucciones
   ...
} while (condición);
```

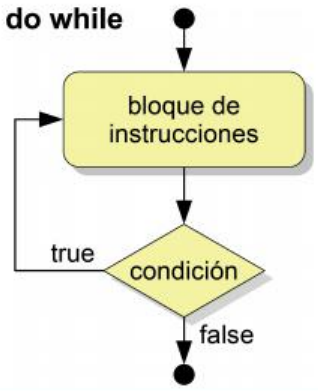


Figura 3.2. *Representación del flujo de ejecución de un bucle* `do-while`.

La Figura 3.2 describe su comportamiento. Se compone de los siguientes puntos:

1. Se ejecuta el bloque de instrucciones.
2. Se evalúa `condición`.
3. Según el valor obtenido, se termina el bucle o se vuelve al punto 1.

Como ejemplo, vamos a escribir el código que muestra los números del 1 al 10 utilizando un bucle `do-while`, en vez de un bucle `while`:

```
int i = 1;
do {
   System.out.println(i);
   i++;
} while (i <= 10);
```

Debemos recordar que es el único bucle que termina en punto y coma (`;`). Mientras el bucle `while` se puede ejecutar de 0 a infinitas veces, el `do-while` lo hace de 1 a infinitas veces. De hecho, la única diferencia con el bucle `while` es que `do-while` se ejecuta, al menos, una vez.

Actividad resuelta 3.5

Desarrollar un juego que ayude a mejorar el cálculo mental de la suma. El jugador tendrá que introducir la solución de la suma de dos números aleatorios comprendidos entre 1 y 100. Mientras la solución introducida sea correcta, el juego continuará. En caso contrario, el programa terminará y mostrará el número de operaciones realizadas correctamente.

Solución

```
import java.util.Scanner;
/* Al menos hay que calcular una suma, por este motivo vamos a utilizar un bucle
 * do-while. Los operandos estarán comprendidos entre 1 y 100*/
public class Main {
   public static void main(String[] args) {
      int resultado, operando1, operando2; //variables
      int numOperaciones = 0;
      do {
         operando1 = (int)(Math.random()*100+1);
         operando2 = (int)(Math.random()*100+1);
         System.out.print(operando1 +" + " + operando2 + " = ");
         resultado = new Scanner(System.in).nextInt();
         numOperaciones++;
      } while (resultado == operando1 + operando2);
      //numOperaciones contabiliza cuántas operaciones se han mostrado. De ellas
      //(numOperaciones -1) son correctas y la última es errónea (si no, no hubiera
      //terminado el do-while).
      System.out.println("A conseguido realizar: " + (numOperaciones -1) +
                         " sumas consecutivas");
   }
}
```

3.2. Bucles controlados por contador: for

El bucle `for` permite controlar el número de iteraciones mediante una variable (que suele recibir el nombre de *contador*). La sintaxis de la estructura `for` es:

```
for (inicialización; condición; incremento) {
   bloque de instrucciones
   ...
}
```

Donde,

- **Inicialización:** es una lista de instrucciones, separadas por comas, donde generalmente se inicializan las variables que van a controlar el bucle. Se ejecutan una sola vez antes de la primera iteración.

- **Condición:** es una expresión booleana que controla las iteraciones del bucle. Se evalúa antes de cada iteración; el bloque de instrucciones se ejecutará solo cuando el resultado sea `true`.
- **Incremento:** es una lista de instrucciones, separadas por comas, donde se suelen modificar las variables que controlan la condición. Se ejecuta al final de cada iteración.

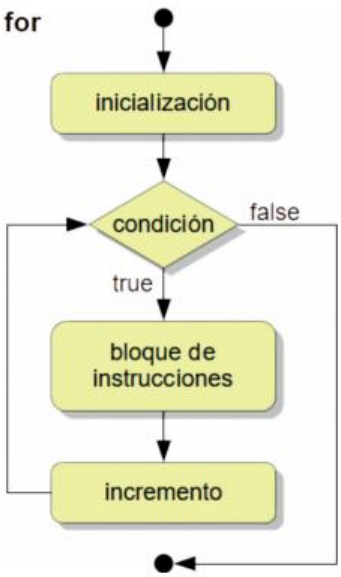


Figura 3.3. *Secuencia del flujo de control de un bucle* `for`.

El funcionamiento de `for` se describe en la Figura 3.3. Consiste en los siguientes puntos:

1. Se ejecuta la inicialización; esto se hace una sola vez al principio.
2. Se evalúa la condición: si resulta `false`, salimos del bucle y continuamos con el resto del programa; en caso de que la evaluación sea `true`, se ejecuta todo el bloque de instrucciones.
3. Cuando termina de ejecutarse el bloque de instrucciones, se ejecuta el incremento.
4. Se vuelve de nuevo al punto 2.

Aunque `for` está controlado por una condición que, en principio, puede ser cualquier expresión booleana, la posibilidad de configurar la inicialización y el incremento de las variables que controlan el bucle permite determinar de antemano el número de iteraciones.

Veamos un ejemplo donde solo se usa la variable `i` para controlar el bucle:

```
for (int i = 1; i <= 2; i++) {
   System.out.println("La i vale " + i);
}
```

Argot técnico

En este caso, la variable `i`, además de inicializarse, también se declara en la zona de inicialización. Esto significa que `i` solo puede usarse dentro de la estructura `for`. En la Unidad 4 profundizaremos en el ámbito de las variables.

A continuación, se muestra una traza de la ejecución del bucle anterior:

1. Primero se ejecuta la inicialización: `i=1`;
2. Evaluamos la condición: ¿es cierto que `i` ≤ 2? Es decir: ¿1 ≤ 2?
3. Cierto. Ejecutamos el bloque de instrucción: `System.out.println(...)`
4. Obtenemos el mensaje: «La i vale 1».
5. Terminado el bloque de instrucciones, ejecutamos el incremento: (`i++`) `i` vale 2.
6. Evaluamos la condición: ¿es cierto que `i` ≤ 2? Es decir: ¿2 ≤ 2?
7. Cierto. Ejecutamos `System.out.println(...)`
8. Obtenemos el mensaje:« La `i` vale 2».
9. Ejecutamos el incremento: (`i++`) la `i` vale 3.
10. Evaluamos la condición: ¿es cierto que `i` ≤ 2? Es decir: ¿3 ≤ 2?
11. Falso. El bucle termina y continúa la ejecución de las sentencias que siguen a la estructura `for`.

Actividad propuesta 3.2

Implementa la aplicación Eco, que pide al usuario un número y muestra en pantalla la salida:

Eco...

Eco...

Eco...

Se muestra «Eco...» tantas veces como indique el número introducido. La salida anterior sería para el número 3.

Actividad resuelta 3.6

Escribir una aplicación para aprender a contar, que pedirá un número *n* y mostrará todos los números del 1 a *n*.

Solución

```
import java.util.Scanner;
/* Sabemos con certeza el número de iteraciones del bucle: n, por lo que
 * utilizaremos un bucle for que recorrerá todos los números de 1 a n. */
public class Main {
  public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Introduzca un número: ");
      int n = sc.nextInt();
      for (int i = 1; i <= n; i++) { //i tomará los valores de 1 a n
        //la variable i es una variable del bloque de instrucciones del for, es
        //decir, solo se puede utilizar en dicho bloque (su ámbito es el bloque)
```

```
            //Utilizar la variable i fuera del bloque genera un error
            System.out.println(i); //mostramos i
        }
    }
}
```

Actividad resuelta 3.7

Escribir todos los múltiplos de 7 menores que 100.

Solución

```
/* Vamos a utilizar un bucle for, inicializando la i a 0, e iterando hasta que el valor
 * supere 100. Los múltiplos de 7, se caracterizan por que se diferencian en 7. */
public class Main {
    public static void main(String[] args) {
        for (int i = 0; i < 100; i += 7) {
            System.out.println(i);
        }
        // Cuando el bloque de instrucciones de for, while o do-while está formado
        // por una sola instrucción, no precisa de llaves { }.
        // Aunque, por claridad en el código, se aconseja ponerlas.
    }
}
```

Actividad resuelta 3.8

Pedir diez números enteros por teclado y mostrar la media.

Solución

```
import java.util.Scanner;
/* Como tenemos claro que vamos a solicitar 10 números al usuario, utilizaremos
 * un bucle for.
 * Sumaremos todos los números introducidos y al final dividiremos entre 10
 * para obtener la media. */
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n; //cada número introducido por el usuario
        int suma = 0; //acumulará la suma de todos los números introducidos
        double media; //la media puede contener decimales, por eso será double
        for (int i = 1; i <= 10; i++) {
            System.out.println("Escriba un número: ");
            n = sc.nextInt();
            suma += n; //es lo mismo que: suma = suma + n
        }
        media = suma / 10.0; //calculamos la media
        System.out.println("La media es: " + media); //mostramos
    }
}
```

Actividad resuelta 3.9

Implementar una aplicación que pida al usuario un número comprendido entre 1 y 10. Hay que mostrar la tabla de multiplicar de dicho número, asegurándose de que el número introducido se encuentra en el rango establecido.

Solución

```
import java.util.Scanner;
/* Las tablas de multiplicar nos traen recuerdos de nuestros tiempos de escolares,
 * cuando intentábamos aprenderlas (recitándolas una y otra vez). */
public class Main {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int num; //del que mostraremos la tabla de multiplicar
      //nos aseguramos de que el número está entre 1 y 10
      do {
         System.out.print("Introduzca un número (de 1 a 10): ");
         num = sc.nextInt();
      } while (!(1 <= num && num <= 10));
      System.out.println("\n\nTabla del " + num);
      for (int i = 1; i <= 10; i++) {
         System.out.println(num + " x " + i + " = " + num * i);
      }
   }
}
```

Actividad resuelta 3.10

Diseñar un programa que muestre la suma de los 10 primeros números impares.

Solución

```
/* El bucle for estará controlado por i (1..10).
 * El i-ésimo número impar se calcula: 2*i - 1 */
public class Main {
   public static void main(String[] args) {
      double suma = 0; // guardará la suma de los 10 primeros impares
      for (int i = 1; i <= 10; i++) {
         int impar = 2 * i - 1;
         suma += impar;
      }
      System.out.println("La suma de los 10 primeros impares es: " + suma);
   }
}
```

Actividad propuesta 3.3

Implementa un programa que pida al usuario un número positivo y lo muestre guarismo a guarismo. Por ejemplo, para el número 123, debe mostrar primero el 3, a continuación el 2 y por último el 1.

Actividad resuelta 3.11

Pedir un número y calcular su factorial. Por ejemplo, el factorial de 5 se denota 5! y es igual a $5 \times 4 \times 3 \times 2 \times 1 = 120$.

Solución

```
import java.util.Scanner;
/* El factorial de n se define como el producto de todos los enteros entre 1 y n.
 * Por ejemplo: el factorial de 10 es: 10*9*8*7*6*5*4*3*2*1 = 3628800 */
public class Main {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      //podríamos declarar factorial de tipo long, el tamaño de este tipo permite
      //calcular hasta el factorial de 25. Mejor utilizamos un double
      double factorial;
      int num;
      System.out.print("Introduzca un número: ");
      num = sc.nextInt();
      factorial = 1; // es importante inicializarlo a 1, ya que multiplicará
      for (int i = num; i > 0; i--) {
         factorial = factorial * i;
      }
      System.out.println("El factorial de " + num + " es: " + factorial);
   }
}
```

Actividad resuelta 3.12

Pedir 5 calificaciones de alumnos y decir al final si hay algún suspenso.

Solución

```
import java.util.Scanner;
/* Utilizamos una bandera para controlar si entre los alumnos existe al menos uno
 * con una asignatura suspensa (nota menor que 5). Una bandera es una variable,
 * normalmente booleana, que indica, mediante sus valores, alguna situación o
 * estado. En este caso:
 * -suspenso = false, significa que no existe ninguna nota suspensa
 * -suspenso = true, significa que existe, al menos, un alumno suspenso
 * Hay que tener cuidado cuando se activa una bandera, en no volver a desactivarla,
 * ya que entonces no refleja lo que intentamos evaluar, sino la última situación
 * ocurrida. */
public class Main {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      boolean suspensos = false; // suponemos que en principio no hay ningún suspenso
      for (int i = 0; i < 5; i++) {
         System.out.print("Introduzca nota (de 0 a 10): ");
         int notas = sc.nextInt();
         if (notas < 5) { //si la nota corresponde a un suspenso
            suspensos = true; //activamos la bandera a cierto
         }
      }
```

```
        if (suspensos) {
            System.out.println("Hay alumnos suspensos");
        } else {
            System.out.println("No hay suspensos");
        }
    }
}
```

Actividad resuelta 3.13

Dadas 6 notas, escribir la cantidad de alumnos aprobados, condicionados (nota igual a cuatro) y suspensos.

Solución

```
import java.util.Scanner;
/* Utilizaremos contadores que se incrementan cuando nos encontramos en una
 * situación concreta: la nota está aprobada, está condicionada o está suspensa. */
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int aprobados = 0, suspensos = 0, condicionados = 0; //contadores
        for (int i = 1; i <= 6; i++) {
            System.out.println("Nota del alumno número " + i + ": ");
            int nota = sc.nextInt();
            if (nota == 4) { //comprobaremos en que caso nos encontramos
                condicionados++;
            } else if (nota >= 5) {
                aprobados++;
            } else if (nota < 4) { //este if es redundante , al ser el único caso posible
                suspensos++; //y podríamos poner else {...}
            }
        }
        System.out.println("Aprobados: " + aprobados); //mostramos el informe
        System.out.println("Suspensos: " + suspensos);
        System.out.println("Condicionados: " + condicionados);
    }
}
```

Nota técnica

Las actividades que hemos realizado cada vez incluyen un mayor número de variables. Se recomienda utilizar identificadores lo más descriptivos posible, que permitan de un vistazo saber que representa el valor que contiene cada variable.

Por ejemplo, en la Actividad resuelta 3.13, se podrían haber usado los identificadores `n`, `a`, `s` y `c` para las notas, número de aprobados, suspensos e indicar si existe algún alumno condicionado. El problema es que estos identificadores son muy poco descriptivos, lo que dificulta la comprensión del código.

3.3. Salidas anticipadas

Dependiendo de la lógica que implementar en un programa, puede ser interesante terminar un bucle antes de tiempo y no esperar a que termine por su condición (realizando todas las iteraciones). Para poder hacer esto disponemos de:

- `break`: interrumpe completamente la ejecución del bucle.
- `continue`: detiene la iteración actual y continúa con la siguiente.

Cualquier programa puede escribirse sin utilizar `break` ni `continue`; se recomienda evitarlos, ya que rompen la secuencia natural de las instrucciones. Veamos un ejemplo:

```
i = 1;
while (i <= 10) {
   System.out.println("La i vale" + i);
   if (i == 2) {
      break;
   }
   i++;
}
```

En un primer vistazo da la impresión de que el bucle ejecutará 10 iteraciones, pero cuando está realizando la segunda (`i` vale 2), la condición de `if` se evalúa como cierta y entra en juego `break`, que interrumpe completamente el bucle, sin que se ejecuten las sentencias restantes de la iteración en curso ni el resto de las iteraciones. Tan solo se ejecutan dos iteraciones y se obtiene:

```
La i vale 1
La i vale 2
```

Veamos otro ejemplo:

```
i = 0;
while (i < 10) {
   i++;
   if (i % 2 == 0) { //si i es par
      continue;
   }
   System.out.println("La i vale " + i);
}
```

Cuando la condición `i % 2 == 0` sea cierta, es decir, cuando `i` es par, la sentencia `continue` detiene la iteración actual y continúa con la siguiente, saltándose el resto del bloque de instrucciones. `System.out.println` solo llegará a ejecutarse cuando `i` sea impar, o dicho de otro modo: en iteraciones alternas. Se obtiene la salida por consola:

```
La i vale 1
La i vale 3
La i vale 5
La i vale 7
La i vale 9
```

3.4. Bucles anidados

En el uso de los bucles es muy frecuente la anidación, que consiste en incluir un bucle dentro de otro, como describe la Figura 3.4.

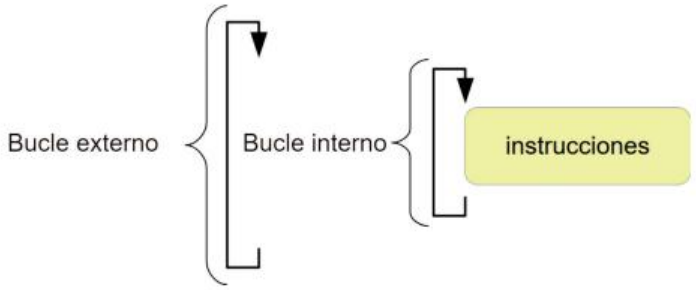


Figura 3.4. *Bucles anidados.*

Al hacer esto se multiplica el número de veces que se ejecuta el bloque de instrucciones de los bucles internos. Los bucles anidados pueden encontrarse relacionados cuando las variables de los bucles más externos intervienen en el control de la iteración de un bucle interno; o independientes, cuando no existe relación alguna entre ellos.

3.4.1. Bucles independientes

Cuando los bucles anidados no dependen, en absoluto, unos de otros para determinar el número de iteraciones, se denominan *bucles anidados independientes.* Veamos un ejemplo sencillo, la anidación de dos bucles for:

```
for (i = 1; i <= 4; i++) {
   for (j = 1; j <= 3; j++) {
      System.out.println("Ejecutando...");
   }
}
```

El bucle externo, controlado por la variable `i`, realizará cuatro iteraciones, donde `i` toma los valores 1, 2, 3 y 4. En cada una de ellas, el bucle interno, controlado por `j`, realizará tres iteraciones, tomando `j` los valores 1, 2 y 3. En total, el bloque de instrucciones se ejecutará doce veces.

Anidar bucles es una herramienta que facilita el procesado de tablas multidimensionales (Unidad 5). Se utiliza cada nivel de anidación para manejar el índice de cada dimensión. Sin embargo, el uso descuidado de bucles anidados puede convertir un algoritmo en algo ineficiente, disparando el número de instrucciones ejecutadas.

Actividad resuelta 3.14

Diseñar una aplicación que muestre las tablas de multiplicar del 1 al 10.

Solución

```
/* Ya tenemos un algoritmo (en un ejercicio anterior) para realizar la tabla de
 * multiplicar de un número dado. La idea es aprovecharlo, y ejecutar el código
 * repetidas veces para mostrar las tablas de multiplicar del 1 al 10. */
public class Main {
   public static void main(String[] args) {
      for (int tabla = 1; tabla <= 10; tabla++) {
         System.out.println("\n\nTabla del " + tabla);
         // por cada iteración del bucle exterior , el interior se ejecuta 10 veces
         for (int i = 1; i <= 10; i++) {
            System.out.println(tabla + " x " + i + " = " + tabla * i);
         }
      }
   }
}
```

3.4.2. Bucles dependientes

Puede darse el caso de que el número de iteraciones de un bucle interno no sea independiente de la ejecución de los bucles exteriores, y dependa de sus variables de control.

Decimos entonces que son *bucles anidados dependientes.* Veamos el siguiente fragmento de código, a modo de ejemplo, donde la variable utilizada en el bucle externo (`i`) se compara con la variable (`j`) que controla el bucle más interno. En algunas ocasiones, la dependencia de los bucles no se aprecia de forma tan clara como en el ejemplo:

```
for (i = 1; i <= 3; i++) {
   System.out.println("Bucle externo, i=" +i);
   j = 1;
   while (j <= i) {
      System.out.println("...Bucle interno, j=" + j);
      j++;
   }
}
```

que proporciona la salida:

Bucle externo, i=1

...Bucle interno, j=1

Bucle externo, i=2

...Bucle interno, j=1

...Bucle interno, j=2

Bucle externo, i=3

...Bucle interno, j=1

...Bucle interno, j=2

...Bucle interno, j=3

- Durante la primera iteración del bucle `i`, el bucle interno realiza una sola iteración.
- En la segunda iteración del bucle externo, con `i` igual a 2, el bucle interno realiza dos iteraciones.
- En la última vuelta, cuando `i` vale 3, el bucle interno se ejecuta tres veces.

La variable `i` controla el número de iteraciones del bucle interno y resulta un total de 1 + 2 + 3 = 6 iteraciones. Los posibles cambios en el número de iteraciones de estos bucles hacen que, *a priori,* no siempre sea tan fácil conocer el número total de iteraciones.

Actividad resuelta 3.15

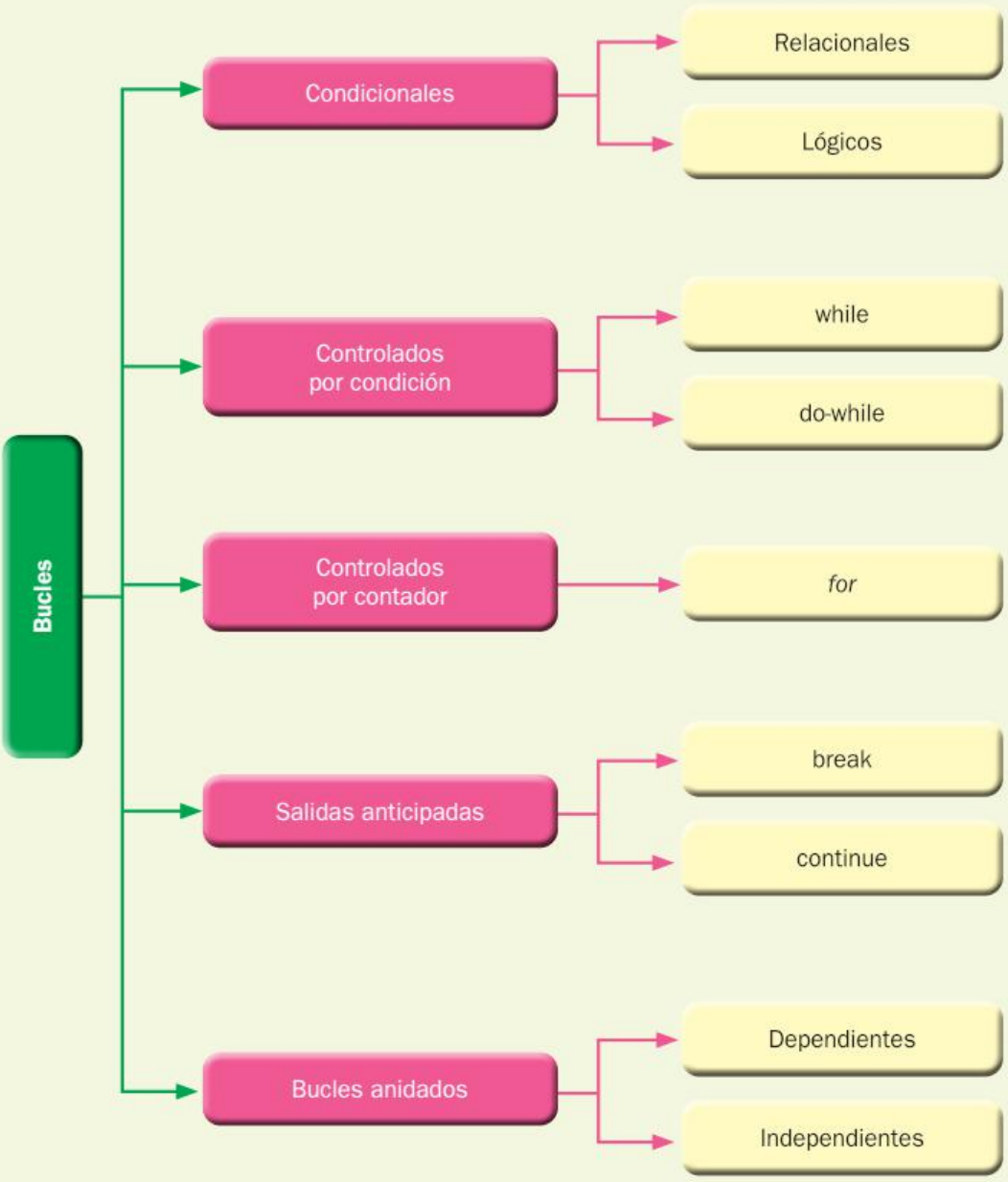
Pedir por consola un número *n* y dibujar un triángulo rectángulo de *n* elementos de lado, utilizando para ello asteriscos (*). Por ejemplo, para *n* = 4:

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

Solución

```
import java.util.Scanner;
/* Utilizaremos un bucle para mostrar cada fila, y dentro de este, otro para escribir
 * cada * (columna). El bucle que escribe cada columna (dentro de la fila) dependerá
 * de los valores de la fila, así conseguimos el efecto "triángulo". */
public class Main {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Escriba n: ");
      int n = sc.nextInt();
      for (int fila = 1; fila <= n; fila++) {
         for (int col = fila; col <= n; col++) {//el número de * coincide con: n-fila+1
            System.out.print("* "); //no println , para que no cambie de línea
         }
         System.out.println(""); //tras cada fila metemos una nueva línea
      }
   }
}
```

MAPA CONCEPTUAL



Actividades de comprobación

3.1. Un bucle `do-while` se ejecutará, como mínimo:

a) Cero veces.
b) Una vez.
c) Infinitas veces.
d) Ninguna de las opciones anteriores es correcta.

3.2. El uso de llaves para encerrar el bloque de instrucciones de un bucle:

a) Es siempre opcional.
b) Es opcional si el bloque está formado por una única instrucción.
c) En cualquier caso, su uso es obligatorio.
d) El programador decide su uso.

3.3. La instrucción que permite detener completamente las iteraciones de un bucle es:

a) `stop`.
b) `break`.
c) `continue`.
d) `finish`.

3.4. La instrucción que permite detener la iteración actual de un bucle, continuando con la siguiente, si procede, es:

a) `stop`.
b) `break`.
c) `continue`.
d) `finish`.

3.5. De un bucle `do-while`, cuya condición depende de una serie de variables que en el bloque de instrucciones no se modifican, se puede afirmar:

a) Que su número de iteraciones será siempre una.
b) Que el número de iteraciones será siempre par.
c) Que las variables cambiarán automáticamente en cualquier momento.
d) Ninguna de las opciones anteriores es correcta.

3.6. ¿Cuántas veces se ejecutará el bloque de instrucciones del bucle más interno en el siguiente fragmento de código?

```
for(i=1; i<=10; i++) {
   for(i=1; i<=5; i++) {
      System.out.println("Hola");
   }
}
```

a) 10 veces.
b) 5 veces.
c) 50 veces.
d) Infinitas veces.

3.7. **Analiza el siguiente código y busca qué valores de `a` y `b` implican un menor número de iteraciones:**

```
for (int i=a; i<=a+b; i++) {
   for(int j=a+b; j>=0; j--) {
      ...
   }
}
```

a) a=1 y b=3.
b) a=3 y b=1.
c) a=1 y b=1.
d) a=3 y b=3.

3.8. **En cada iteración, el incremento de un bucle `for` se ejecuta:**
a) En primer lugar.
b) Después de la inicialización.
c) Después de evaluar la condición.
d) Justo al finalizar cada iteración.

3.9. **Una variable que se declara dentro de su bloque de instrucciones solo se podrá utilizar:**
a) En cualquier parte del programa.
b) En todos los bucles.
c) Dentro del bloque de instrucciones donde se ha declarado.
d) Todas las opciones anteriores son correctas.

3.10. **En un bucle `for`, la inicialización, condición e incremento son:**
a) Todos obligatorios.
b) Todos opcionales.
c) La inicialización siempre es obligatoria.
d) La condición siempre es obligatoria.

Actividades de aplicación

3.11. Realiza un programa que convierta un número decimal en su representación binaria. Hay que tener en cuenta que desconocemos cuántas cifras tiene el número que introduce el usuario.

Por simplicidad, iremos mostrando el número binario con un dígito por línea.

3.12. Modifica la Actividad de aplicación 3.11 para que el usuario pueda introducir un número en binario y el programa muestre su conversión a decimal.

3.13. Escribe un programa que incremente la hora de un reloj. Se pedirán por teclado la hora, minutos y segundos, así como cuántos segundos se desea incrementar la hora introducida. La aplicación mostrará la nueva hora. Por ejemplo, si las 13:59:51 se incrementan en 10 segundos, resultan las 14:00:01.

3.14. Realiza un programa que nos pida un número *n,* y nos diga cuántos números hay entre 1 y *n* que sean primos. Un número primo es aquel que solo es divisible por 1 y por él mismo. Veamos un ejemplo para $n = 8$:

Comprobamos todos los números del 1 al 8 →
- 1 → primo
- 2 → primo
- 3 → primo
- 4 → no primo
- 5 → primo
- 6 → no primo
- 7 → primo
- 8 → no primo

Resultan un total de 5 números primos.

3.15. Diseña una aplicación que dibuje el triángulo de Pascal, para *n* filas. Numerando las filas y elementos desde 0, la fórmula para obtener el *m*-ésimo elemento de la *n*-ésima fila es:

$$E(n, m) = \frac{n!}{m!(n - m)!}$$

Donde *n*! es el factorial de *n.*
Un ejemplo de triángulo de Pascal con 5 filas ($n = 4$) es:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

3.16. Solicita al usuario un número *n* y dibuja un triángulo de base y altura *n,* de la forma (para *n* igual a 4):

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

3.17. Para dos números dados, *a* y *b*, es posible buscar el máximo común divisor (el número más grande que divide a ambos) mediante un algoritmo ineficiente pero sencillo: desde el menor de *a* y *b*, ir buscando, de forma decreciente, el primer número que divide a ambos simultáneamente. Realiza un programa que calcule el máximo común divisor de dos números.

3.18. De forma similar a la Actividad de aplicación 3.17, implementa un algoritmo que calcule el mínimo común múltiplo de dos números dados.

3.19. Calcula la raíz cuadrada de un número natural mediante aproximaciones. En el caso de que no sea exacta, muestra el resto. Por ejemplo, para calcular la raíz cuadrada de 23, probamos $1^2 = 1$, $2^2 = 4$, $3^2 = 9$, $4^2 = 16$, $5^2 = 25$ (nos pasamos), resultando 4 la raíz cuadrada de 23 con un resto (23 – 16) de 7.

3.20. Escribe un programa que solicite al usuario las distintas cantidades de dinero de las que dispone. Por ejemplo: la cantidad de dinero que tiene en el banco, en una hucha, en un cajón y en los bolsillos. La aplicación mostrará la suma total de dinero de la que dispone el usuario.

La manera de especificar que no se desea introducir más cantidades es mediante el cero.

Actividades de ampliación

3.21. Los algoritmos que se implementan mediante bucles suelen tener un mayor costo computacional, consumiendo un mayor tiempo de ejecución. Busca en internet información sobre el concepto de orden de un algoritmo y cómo afectan los bucles a dicho orden. ¿Qué orden de algoritmos prefieres? Justifica tu respuesta.

3.22. Investiga sobre los algoritmos de fuerza bruta, como por ejemplo los que calculan todos los posibles movimientos en una partida de ajedrez o buscan una clave mediante todas las combinaciones posibles. ¿Cuál es el inconveniente de este tipo de algoritmos?

3.23. Realiza una búsqueda y pregunta a docentes de otros módulos sobre procesos que ocurran en un ordenador y que se ejecutan de forma perpetua dentro de un bucle.

3.24. Calcula para un monitor, con una determinada resolución y frecuencia de refresco de cada píxel, cuántas operaciones harán falta para procesar todos los píxeles durante un segundo. Escribe un boceto de programa que pueda ejecutar esta acción.

3.25. Realiza una investigación en internet sobre algoritmos donde el uso de bucles sea imprescindible. Piensa si es posible convertir los algoritmos que has encontrado en otros que no usen bucles o que utilicen un número menor de iteraciones.

3.26. Escribe un programa que, mediante bucles, consiga que el tiempo de ejecución sea lo máximo posible. Realiza una competición con el resto de la clase para ver quién consigue el algoritmo más lento. Si los tiempos de ejecución son excesivos, pide ayuda al profesorado para que calcule el orden de tu algoritmo.