TEMA 5: ESTRUCTURAS DE CONTROL

1.- ENTRADA/SALIDA DE INFORMACIÓN

Antes de comenzar a estudiar las estructuras de control, tenemos que abordar un aspecto importantísimo en cualquier lenguaje de programación: la entrada/salida de datos por consola. Esto en Java, sobre todo la entrada, no es tan simple como en otros lenguajes de programación, como por ejemplo C.

En el caso de la **salida de información** por consola utilizaremos los métodos **`println`** y **`print`**, la diferencia entre ellos, es que el primero añade al final un salto de línea y el segundo no. Estos métodos pertenecen a la clase ***System***, y utilizan el flujo estándar de salida (***out***). Su uso es muy simple, mostrarán en pantalla aquello que introduzcamos entre los paréntesis de los métodos, pudiendo combinar texto (cadenas) que siempre irán entre comillas dobles, con variables, llamadas a otros métodos, expresiones, ... Tan sólo tendremos que unir las cadenas con lo demás mediante el operador de concatenación ("+"). Dentro de las cadenas podemos utilizar modificadores como \t (tabulador), \n (salto de línea), etc.

Ejemplos:

```
int a=5, b=2;

System.out.println("Hola mundo");   // Mostrará: Hola mundo

System.out.println("La multiplicación de " + a + " por " + b + " es: " + (a*b));
      // Mostrará La multiplicación de 5 por 2 es: 10
```

Por su parte, para **recoger datos de teclado** correctamente, tenemos que manejar excepciones, manejar flujos de datos, etc, conceptos que se irán aprendiendo a lo largo del módulo, pero que ahora son demasiado avanzados para abordar su estudio. Así que en las primeras prácticas que hagamos proporcionaremos información a nuestros programas de dos formas posibles:

- Mediante paso de parámetros al programa.
- Utilizando una clase hecha por mí, que os permitirá leer datos de teclado, utilizando los conceptos antes comentados, aunque de momento no comprendáis su funcionamiento, simplemente utilizaréis esos métodos, para la entrada de datos por consola.

También hablaremos de lo que es una ***sentencia,*** es una ***expresión*** que acaba en ***punto y coma*** (;). Se permite incluir varias sentencias en una línea, aunque lo habitual es utilizar una línea para cada sentencia. Por ejemplo:

```
i = 0; j = 5; x = i + j;// Línea compuesta de tres sentencias
```

2.- ESTRUCTURAS DE CONTROL

Cuando una máquina empieza la ejecución de un programa, empieza leyendo la primera línea de código y la ejecuta, pasa a la siguiente y la ejecuta y así sucesivamente hasta llegar al final del programa. La trayectoria que sigue al ir realizando todos los pasos presentes en el algoritmo constituye lo que se denomina **flujo de control del programa**.

Cualquier herramienta que controle el flujo de control de un algoritmo constituye lo que denominaremos una **estructura de control**. Existen distintos tipos de estructuras de control que pueden clasificarse en tres tipos básicos:

1. *Estructura secuencial*. Una sentencia se ejecuta a continuación de la otra, sin alterar el flujo.
2. *Estructura selectiva*. Dirigen el flujo de ejecución del programa a unas instrucciones u otras dependiendo del cumplimiento de una condición.
3. *Estructura repetitiva*. Permiten la ejecución repetida de un conjunto de instrucciones, un número determinado o indeterminado de veces.

3.- ESTRUCTURA SECUENCIAL

Es una estructura con una entrada y una salida en la cual figuran una serie de acciones cuya ejecución es lineal y en el orden en que aparecen. A su vez todas las acciones tienen una única entrada y una única salida. Dicho de otra forma una estructura secuencial es aquella en la que una acción sigue a otra sin romper la secuencia.

Ejemplo:
```
importe = cantidad * precio;
System.out.println(importe);
```

Veamos otro ejemplo completo que resuelve una ecuación de segundo grado:

```
public static void main(String[] args) {

        // Variables locales
        float a,b,c;
        double x1,x2;

        a = Integer.parseInt(args[0]); // Coeficiente grado 2
        b = Integer.parseInt(args[1]); // Coeficiente grado 1
        c = Integer.parseInt(args[2]); // Término independiente

        x1 = (-b + Math.sqrt(Math.pow(b,2) - (4*a*c))) /( 2*a);
        x2 = (-b - Math.sqrt(Math.pow(b,2) - (4*a*c))) /( 2*a);

        System.out.println("Las raíces valen: " + x1 + " y " + x2);
}
```

Esta estructura por sí sola es insuficiente, ya que no permite resolver problemas que exigen una toma de decisión, ni la ejecución de un conjunto de acciones un número determinado de veces.

4.- ESTRUCTURAS DE SELECCIÓN O CONDICIONALES

En determinados problemas necesitamos, en función de una condición tomar una alternativa u otra. Para ello utilizamos la herramienta condicional, también conocida como bifurcación. En una estructura condicional se evalúa una expresión lógica y, en función el resultado de la misma, se realiza una opción u otra.

Dicho en otras palabras, es la herramienta que va a posibilitar alterar el flujo de control de un programa dependiendo de la evaluación de una condición.

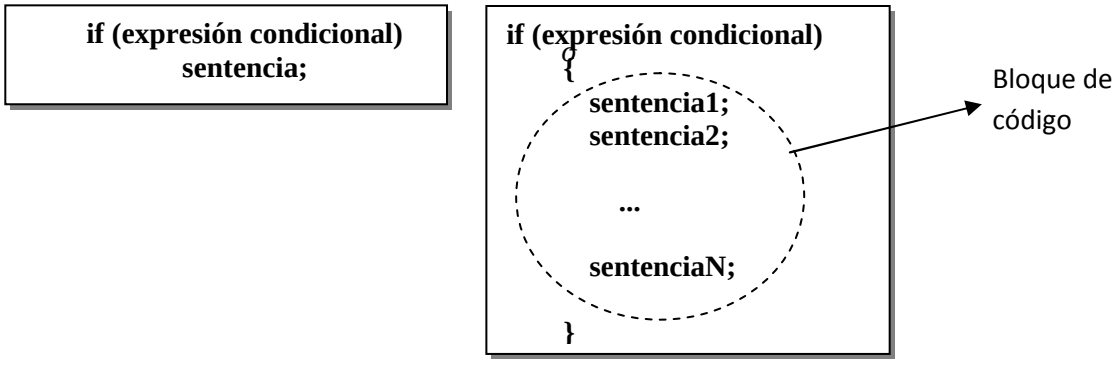
Existen tres tipos de sentencias alternativas:
- ✓ Sentencias de selección simple.
- ✓ Sentencias de selección binaria.
- ✓ Sentencias de selección múltiple.

4.1. Sentencias de selección simple

Según el cumplimiento o no de una determinación condición se procesará la ejecución de un conjunto de acciones o no. El orden de evaluación de una estructura condicional simple en Java es:

- ✓ Evaluar la *expresión condicional*.
- ✓ Si la *expresión condicional* es cierta
  - Ejecutar sentencia o bloque de sentencias (*bloque de código*).
- ✓ En caso contrario
  - Continuar con la ejecución del programa de forma secuencial.

La sintaxis de las sentencias de selección simple en Java es:



donde *expresión condicional* puede ser cualquier expresión válida de Java. En el caso de que sólo haya que ejecutar una sentencia se puede utilizar la sintaxis de los dos recuadros (aunque es más simple la del primero), cuando haya que ejecutar más de una sentencia (**bloque de código**) obligatoriamente habrá que usar la sintaxis del segundo recuadro.

En un **bloque de código**, se rodean las sentencias con llaves de apertura y cierre. Una vez hecho esto, las sentencias forman una unidad lógica, que se puede utilizar en cualquier lugar donde pueda hacerlo una sentencia sencilla.

***Ejemplo*:**

```
if (dato > 0)
    System.our.println(“Incorrecto”);

System.our.println(dato);
```

En este ejemplo si la variable dato contiene un valor positivo (mayor de cero) el algoritmo mostrará el mensaje “Incorrecto”, ha continuación, independientemente del valor de la variable dato, mostrará el contenido de la misma.

Ejemplo: Programa que almacena en dos variables de nombre ‘x y ‘y’ dos valores enteros, mostrando en aquellos casos en los que ‘x’ es mayor que ‘y’ un mensaje que diga “Verdadero”.

```
public static void main(String[] args) {
      // TODO Auto-generated method stub
      float x, y;

      x = Integer.parseInt(args[0]);
      y = Integer.parseInt(args[1]);

      if (x > y)
         System.our.println(“Verdadero”);
 }
```

***Ejemplo*:**
```
if ((dato > 0) && (dato<50)) {
    System.our.println("El dato es mayor de 0 y menor de 50");
    System.our.println("El valor de dato es: " + dato);
}
```

En este ejemplo si la variable dato contiene un valor comprendido entre 1 y 49 el algoritmo mostrará el mensaje "El dato es mayor de 0 y menor de 50", y el valor de la variable. Si el valor de dato no está en el rango indicado el programa no hace nada.

A continuación vamos a volver a ver el ejemplo para resolver una ecuación de segundo grado pero esta vez mejorado:

```
public static void main(String[] args) {
        float a,b,c;
        double x1,x2;

        a = Integer.parseInt(args[0]); // Coeficiente grado 2
        b = Integer.parseInt(args[1]); // Coeficiente grado 1
        c = Integer.parseInt(args[2]); // Término independiente

        if ((Math.pow(b,2) - 4*a*c) > 0) {
              x1 = (-b + Math.sqrt(Math.pow(b,2) - (4*a*c))) /( 2*a);
              x2 = (-b - Math.sqrt(Math.pow(b,2) - (4*a*c))) /( 2*a);
              System.out.println("Las raíces valen: " + x1 + " y " +x2);
        } // end if
 }
```

4.2. Sentencias de selección binaria

El orden de evaluación de una estructura condicional doble es:

- ✓ Evaluar la *expresión condicional*.
- ✓ Si la *expresión condicional* es cierta
  - ▪ Ejecutar primera sentencia o primer bloque de sentencias.
- ✓ En caso contrario
  - ▪ Ejecutar segunda sentencia o segundo de sentencias.
- ✓ Continuar con la ejecución del programa de forma secuencial.

Sintaxis:

```
if (expresión condicional)
        sentencia1;
else
      sentencia2;
```

o

```
if (expresión condicional)
      {
        bloque de código 1;
      }
else
      {
        bloque de código 2;
      }
```

***Ejemplo*:**
```
if (dato > 0) entonces
    System.out.println("Incorrecto");
else
   System.out.println("Correcto");

System.out.println(dato);
```

En el ejemplo anterior si la variable dato contiene un valor positivo (mayor de cero) el programa mostrará el mensaje "Incorrecto", y si contiene un dato negativo o cero mostrará el mensaje "Correcto". Ha continuación, independientemente del valor de la variable dato, mostrará el contenido de la misma.

Ejemplo: Programa que nos dice si un número pasado al programa es par o impar.

```
public static void main(String[] args) {

        int n = Integer.parseInt(args[0]);

        if ((n%2)!=0)
            System.out.println("El número " + n + " es impar");
        else
            System.out.println("El número " + n + " es par");
}
```

Anidamientos

- No hay restricciones sobre las sentencias que pueden aparecer en las distintas ramas de estas sentencias de selección.
- Puede, por tanto, existir una sentencia de selección dentro de otra (anidamiento).
- Debe evitarse si el código resultante es complicado de leer.
- Se pueden anidar cuantas sentencias if se deseen, pero hay que tener en cuenta lo siguiente:

  *Una sentencia **else** siempre quedará asociada con el **if** precedente más próximo, siempre y cuando no tenga ya asociada otra sentencia **else**.*

Ejemplo: Programa que a partir de dos valores numéricos que almacena en las variables 'x' e 'y', determina si son iguales, y en caso no serlo, indica cual de ellos es el mayor.

```
public static void main(String[] args) {

        int x = Integer.parseInt(args[0]);
        int y = Integer.parseInt(args[0]);

        if (x > y)
            System.out.println("El mayor es: " + x);
        else
            if (x == y)
                System.out.println("Son iguales");
            else
                System.out.println("El mayor es:" + y);
}
```

Ejemplo de sentencia de selección anidada:

```
if (coste < 100)
    coste = coste + 1
else
    if (coste < 200)
        coste = coste + 2
    else
        coste = coste + 3
```

En esta sentencia anidada:
- Cuando coste < 100 ⇒ *coste = coste + 1*
- Cuando coste >=100 y coste < 200 ⇒ *coste = coste + 2*
- Cuando coste >=200 ⇒ *coste = coste + 3*

De una forma no anidada, aunque menos eficiente, el código anterior se puede escribirse como:

```
if (coste < 100)
    coste = coste + 1
if ((coste >= 100) && (coste < 200))
    coste = coste + 2
if (coste >=200)
    coste = coste + 3
```

En general, la sentencia de selección anidada es útil cuando uno o más alternativas son dependientes de la rama que toma otra alternativa.

Siempre que se encuentre un problema en el que alguna condición deba ser evaluada antes de comprobar otra condición, será apropiado el uso de la estructura anidada.

Ejemplo:

```
if ((dia>=1) Y (dia <=7))
    if (dia == 1)
     ....
    // end if
    if (dia == 2)
     ....
    // end if
    ....
else
    escribe("Numero de día incorrecto")
// end if
```

Volvamos a ver el ejemplo de resolución de una ecuación de segundo grado, nuevamente mejorado.

```
public static void main(String[] args) {
	// TODO Auto-generated method stub
	float a,b,c;
	double x1,x2;

	a = Integer.parseInt(args[0]); // Coeficiente grado 2
	b = Integer.parseInt(args[1]); // Coeficiente grado 1
	c = Integer.parseInt(args[2]); // Término independiente

	if ((Math.pow(b,2) - 4*a*c) == 0) { // Raiz doble
		x1 = -b / (2*a);
		System.out.println("El resultado es una raíz real doble: " + x1);
	}
	else {
	    if ((Math.pow(b,2) - 4*a*c) > 0) { // Dos raices
			x1 = (-b + Math.sqrt(Math.pow(b,2) - (4*a*c))) /( 2*a);
			x2 = (-b - Math.sqrt(Math.pow(b,2) - (4*a*c))) /( 2*a);
			System.out.println("Las raíces valen: " + x1 + " y " +x2);
	    }
	    else {
			System.out.println("Raíces complejas");
	    } // end if
	} // end if
}
```

4.3. Sentecias if elseif else

Permite introducir más de una expresión de comparación. Si la primera condición no se cumple, se compara la segunda y así sucesivamente. En el caso de que no se cumpla ninguna de las comparaciones se ejecutan las sentencias correspondientes al ***else***.

```
if (booleanExpression1) {
      sentecias1;
      }
else if (booleanExpression2) {
      sentencias2;
      }
else if (booleanExpression3){
      sentencias3;
      }
else {
      sentencias4;
      }
```

Véase a continuación el siguiente ejemplo:

```
int numero = 61;            // La variable "numero" tiene dos dígitos
if (Math.abs(numero) < 10) // Math.abs() calcula el valor absoluto. (false)
     System.out.println("Numero tiene 1 digito ");
else if (Math.abs(numero) < 100) // Si numero es 61, estamos en este caso (true)
     System.out.println("Numero tiene 2 digitos ");
else { // Resto de los casos
     System.out.println("Numero tiene mas de 3 digitos ");
     System.out.println("Se ha ejecutado la opcion por defecto ");
     }
```

4.4. Sentencias de selección múltiple

La sentencia **switch** es la sentencia de selección múltiple de Java. Esta sentencia compara el valor resultante de una expresión o variable con una lista de constantes de tipo entero o carácter, de manera que cuando se establece una asociación o correspondencia entre ambos se ejecuta una o más sentencias. Se trata de una alternativa a la bifurcación ***if elseif else.***

El formato general de la sentencia **switch** es el siguiente:

```
switch (Expresión) {
   case constante1:    Bloque de sentencias 1;
                       break;
   case constante2:    Bloque de sentencias 2;
                       break;
            .
            .
            .
   case constanteN:    Bloque de sentencias N;
                       break;
   [default:           Sentencias por defecto;
                       break;]
   }
```

donde:

- **Expresión:** Cualquier expresión que devuelva un valor de tipo entero (***int***) o de tipo carácter (***char***).
- **constantel, constante2..., constanteN**: Sólo puede ser una constante de tipo numérico entero o de tipo carácter.
- **Bloque de sentencias:** Puede estar constituido por una o `*n*' sentencias que serán ejecutadas cuando el resultado de la expresión coincida con el valor de la constante especificada.
- **Break:** Es un mandato imperativo que genera la salida del bloque de código. Su uso es opcional, aunque su omisión genera una ejecución continuada que parte del bloque de sentencias asociado a la constante 'case' evaluada como cierta, pasando por el resto de los bloques de sentencias asociados a otras constantes 'case' hasta la localización de una sentencia 'break' o la finalización de la sentencia 'switch'.

En aquellos casos en los no se establece ninguna correspondencia entre el valor obtenido como resultado en la expresión y las constantes 'case' especificadas se ejecuta siempre por defecto la sentencia `***default***'. El uso de esta sentencia no es estrictamente obligatorio siendo perfectamente omitible, aunque siempre es recomendable mantenerla en un buen estilo de programación

A la hora de construir una alternativa múltiple, es conveniente tener presente:

a) En una sentencia `switch', no puede haber dos constantes 'case' con el mismo valor, salvo que una sentencia `switch' se encuentre anidada dentro de otra sentencia `switch', en cuyo caso sería posible que existieran constantes 'case' iguales.

b) A diferencia de la sentencia `if', la sentencia 'switch' no puede evaluar expresiones relacionales o lógicas, únicamente puede realizar operaciones de comparación entre el resultado de la expresión y las constantes especificadas en cada sentencia `case'.

c) Cualquier constante de tipo carácter utilizada en una sentencia ‘switch' es convertida automáticamente a su equivalente valor entero.

Secuencia de evaluación o ejecución:

1 .Obtener el resultado de la Expresión.

2. Comparar el resultado obtenido con *Constante1*.
   Si son iguales
   Ejecutar *Bloque de sentencias1*
   En caso contrario
      Saltar al siguiente `case'.

2. Comparar el resultado obtenido con *Constante2*.
   Si son iguales
   Ejecutar *Bloque de sentencias2*
   En caso contrario
      Saltar al siguiente `case'.

2. Comparar el resultado obtenido con *ConstanteN*.
   Si son iguales
   Ejecutar *Bloque de sentenciasN*
   En caso contrario
      Saltar a la opción por defecto.

5. Ejecutar *Bloque de sentencias por defecto* (default).

6. Continuar con la ejecución del programa secuencialmente

Ejemplo: Programa que determina si el carácter introducido a través del dispositivo estándar de entrada, es una vocal o no.

```
public static void main(String[] args) {

      char vocal = (char)Integer.parseInt(args[0]);

      switch (vocal)
      {
            case 'a': System.out.println("La vocal es a."); break;
            case 'e': System.out.println("La vocal es e."); break;
            case 'i': System.out.println("La vocal es i."); break;
            case 'o': System.out.println("La vocal es o."); break;
            case 'u': System.out.println("La vocal es u."); break;
             default:System.out.println("No es una vocal.");
      }
}
```

5. ESTRUCTURAS DE ITERACIÓN O REPETITIVAS

Una estructura repetitiva, ***ciclo, bucle, lazo*** *o* ***loop*** es una estructura de control que indica la repetición de un conjunto de acciones cero o más veces. Estas acciones del ciclo se pueden repetir un número fijo de veces o bien el número que imponga una determinada condición. Al conjunto de dichas acciones lo llamaremos *cuerpo del bucle* y cada vez que se ejecutan las acciones presentes en el cuerpo del bucle, se dice, que se produce una *iteración*.

Las estructuras iterativas se pueden clasificar, en función del número de repeticiones, en los siguientes tipos:

- ✓ Controlados por contador. Son bucles en los que el número de iteraciones está fijo o predeterminado antes de empezar a iterar. En este grupo destaca el bucle "*for*".
- ✓ Controlados por centinela. En este tipo de bucles el número de iteraciones no está predeterminado, sino que depende de alguna condición que se va modificando con las distintas iteraciones. Existen tres tipos diferentes de bucles controlados por centinela: bucle "*while*", bucle "*do while*" y el "*repeat until*", este último no existe en Java.

Los bucles controlados por centinela son imprescindibles, ya que su funcionamiento no puede simularse con un bucle contador. Por su parte los bucles contador no son imprescindibles, ya que su funcionamiento si puede simularse con un bucle centinela.

Al igual que las sentencias condicionales, las iterativas también se pueden *anidar*, para ello se empieza diseñando la más externa y se termina con las más internas. El comienzo y final de un bucle anidado deben estar dentro del mismo bucle que los anida.

5.1. Bucle "For"

En Java, esta estructura o instrucción repetitiva ofrece una gran potencia y enorme flexibilidad, pues no presenta un formato fijo, dando posibilidad a un amplio número de variaciones en su utilización o tratamiento. Su formato más general es:

a) Formato para la ejecución de una sola sentencia.

```
for (Inicialización; Expresión Condicional; Incremento/Decremento)
    Sentencia1;
```

b) Formato para la ejecución de un bloque de sentencias.

```
for (Inicialización; Expresión Condicional; Incremento/Decremento)
  {
     Bloque de Sentencias;
  }
```

Toda instrucción `for' consta de tres partes:

1.- *Inicialización.* Normalmente suele ser una sentencia de asignación donde la variable de control del bucle toma un valor inicial, se ejecuta al comienzo del for.

2.- *Expresión Condicional.* Esta segunda parte está constituida por una expresión condicional que determina el momento en el que debe finalizar la ejecución del bucle. Se evalúa al comienzo de cada iteración. El bucle termina cuando la expresión de comparación toma el valor ***false***.

3.- *Incremento/Decremento.* En esta tercera parte, normalmente se actualiza la variable de control del bucle mediante el incremento o decremento de la misma y siempre partiendo del valor que ésta haya tomado inicialmente en la primera parte del bucle, así como en sucesivas vueltas. Esta actualización siempre se realiza tras la ejecución de la sentencia o bloque de sentencias que tenga el bucle.

Estas tres partes o áreas en las que se subdivide una sentencia `for' se caracterizan por ser optativas, siendo posible la ausencia de alguna o todas ellas, debiendo estar siempre separadas por punto y coma (;). La ***inicialización*** y el ***incremento/decremento*** pueden tener varias expresiones separadas por comas.

La secuencia de ejecución es la siguiente:

1. Inicializar variable de control del bucle (primera parte).
2. Evaluar *Expresión Condicional* (segunda parte).
   Si *Expresión Condicional* es verdadera,
      Ejecutar bloque de sentencias
   En caso contrario
      Finalizar y continuar con la ejecución del programa secuencialmente.
3. Actualizar variable de control del bucle (tercera parte).
4. Retornar al punto dos.

Ejemplo: Veamos un programa que nos sume los números naturales de 1 al 100, y nos visualice el resultado de esa suma.

```
public static void main(String[] args)
{
        int suma=0;

        for (int cont = 1; cont<=100; cont++)
              suma = suma + cont;

        System.out.println(suma);
}
```

***Ejemplo*:** Obtener la suma de los múltiplos de 3, comprendidos entre 7 y 100, e ir mostrándolos por pantalla.

```
public static void main(String[] args)
{
        int suma=0;
        for (int cont = 9; cont<=100; cont=cont+3) {
              suma = suma + cont;
              System.out.print(cont+" ");
        }

        System.out.println("\nLa suma es: "+suma);
 }
```

En este ejemplo estamos almacenando en la variable suma, la suma de los múltiplos de 3 comprendidos entre 7 y 100, la variable contadora ha comenzado con el valor de 9, que es el primer múltiplo de 3 comprendido dentro del rango indicado, ya que sabemos que los valores 7 y 8 no son múltiplos de 3. En cada iteración la variable cont se irá incrementando en tres unidades.

Ejemplo: En este ejemplo más complejo, el código situado a la izquierda produce la salida que aparece a la derecha:

Código:

```
for(int i = 1, j = i + 10; i < 5; i++, j = 2*i)
   {
     System.out.println(" i = " + i + " j = " + j);
   }
```

Salida:

```
i = 1  j = 11
i = 2  j = 4
i = 3  j = 6
i = 4  j = 8
```

5.2. Bucle *"while"*

Esta estructura de control nos permite programar un conjunto de operaciones que se estará ejecutando mientras se cumpla una determinada condición (*expresión condicional*). La comprobación de la condición se realiza al principio del bucle. Las sentencias que forman el cuerpo del bucle se ejecutan 0 o más veces.

Su sintaxis en Java es:

a) Formato para la ejecución de una sola sentencia.

```
while  (Expresión condicional)
        Sentencia1;
```

b) Formato para la ejecución de un bloque de sentencias.

```
while (Expresión condicional)
  {
        Bloque de sentencias;
  }
```

La secuencia de ejecución es la siguiente:

1. Evaluar *Expresión Condicional*.

   Si *Expresión Condicional* es verdadera,

   Ejecutar bloque de sentencias

   En caso contrario

   Finalizar y continuar con la ejecución del programa secuencialmente.

2. Volver al punto 1.

Ejemplo: Programa que suma todos aquellos números leídos del dispositivo estándar de entrada (teclado) mientras no sean negativos.

```
public static void main(String[] args) {

        int suma=0, num;

        System.out.print("Introduzca un número: ");
        num = Leer.datoInt(); // Lectura del primer número

        while (num >= 0) // Mientras el número sea positivo
        {
            suma = suma + num;

            System.out.print("Introduzca un número: ");
            num = Leer.datoInt(); // Lectura de los restantes números
        }

        System.out.println("Suma = " + suma);
}
```

***Ejemplo*:** En este ejemplo vamos a ir leyendo caracteres de teclado hasta que el usuario introduzca el carácter '**.**', iremos mostrando los caracteres introducidos por pantalla y además contaremos cuantos se han introducido.

```
public static void main(String[] args)
{
    int cont=0;
    char ch;

    System.out.print("Introduzca caracteres ('.' para salir): ");
    ch = Leer.datoChar(); // Lectura del primer carácter

    while (ch != '.') // Mientras el número sea positivo
    {
        System.out.print(ch);
        cont++;

        ch = Leer.datoChar(); // Lectura de los restantes caracteres
    }

    System.out.println("\nSe han introducido " + cont + " caracteres");
}
```

Destacar de este ejemplo la forma de leer los caracteres, una a principio del bucle y otra al final del bucle, esta es la forma habitual de operar con este tipo de bucles, es por ello por lo que puede que el bucle no se ejecute ninguna vez.

5.3. Bucle "*do while*"

Se especifica la condición que ha de cumplirse para que se ejecute el bloque de operaciones que forman el cuerpo del bucle. Permite la programación de un conjunto de acciones que se ejecutarán ***mientras*** que la condición se cumpla. En este caso, la verificación de la comprobación se realiza al finalizar el bloque de operaciones. Las sentencias que forman el cuerpo del bucle se ejecutan 1 o más veces.

Su sintaxis es la siguiente:

a) Formato para la ejecución de una sola sentencia.

```
do
    Sentencia1;
while (Expresión condicional);
```

b) Formato para la ejecución de un bloque de sentencias.

```
do
  {
        Bloque de sentencias;
  }
while (Expresión condicional);
```

La secuencia de ejecución es la siguiente:

1. Ejecutar bloque de sentencias
2. Evaluar *Expresión Condicional*.

   Si *Expresión Condicional* es verdadera,

   Volver al punto 1.

   En caso contrario

   Finalizar y continuar con la ejecución del programa secuencialmente.

Ejemplo: Programa que vaya sumando números hasta que nos den el número cero.

```
public static void main(String[] args)
{
	int suma=0, num;

	do {
	    System.out.print("Introduzca un número (0 para salir): ");
	    num = Leer.datoInt();

	    suma = suma + num;
	}
	while (num!=0);

	System.out.println("\nLa suma de los números introducidos es: " + suma);
}
```

Ejemplo: Programa que lee una serie de números y escribe cuántos números son positivos y negativos se han leído. La finalización de la entrada de datos se realiza por la respuesta al siguiente mensaje "¿Terminar (S/N)?, escrito después de introducir cada número.

```
public static void main(String[] args) {
	int pos=0, neg=0, num;
	char respuesta;

	do
	{
		System.out.print("Introduzca un número: ");
		num = Leer.datoInt();

		if (num>=0) pos++; else neg++;

		System.out.print("Terminar (S/N): ");
		respuesta=Leer.datoChar();
	}
	while (respuesta=='S');

	System.out.println("Hay "+pos+" positivos y "+neg+" negativos");
   }
```

6. SENTENCIAS BREAK, CONTINUE Y RETURN

La sentencia ***break*** se utiliza con una doble finalidad. Para marcar el final de un '*case*' en una sentencia '*switch*' y para forzar de manera inmediata la terminación de un bucle, rompiendo su secuencia interna de ejecución. Su uso permite establecer ciertas salidas desde dentro del bucle ignorando la evaluación de la expresión condicional del ciclo.

El formato de esta sentencia es el siguiente:

```
break;
```

En aquellos casos en los que existan estructuras de control repetitivas anidadas, un '*break*' produce la salida inmediata de aquel bucle en el que se encuentre comprendida.

Por otro lado la sentencia ***continue*** es utilizada en estructuras de control repetitivas, siendo su modo de funcionamiento similar a la sentencia *break*, salvo que en lugar de forzar una terminación fuerza una nueva vuelta o iteración de bucle, saltando cualquier porción de código existente entre el lugar donde se haya ubicada dicha sentencia y el final del bucle.

Su formato:

```
continue;
```

Ejemplo:

```
for (int j=1; j<=10; j++)
   {
       if (j==5) continue;
       System.out.print(j);
   }
```

La salida que dará el anterior algoritmo será: `1 2 3 4 6 7 8 9 10`

6.1. Sentencias break y continue con etiquetas.

Las ***etiquetas*** permiten indicar un lugar donde continuar la ejecución de un programa después de un ***break*** o ***continue***. El único lugar donde se pueden incluir etiquetas es ***justo delante de un bloque*** de código entre llaves {} (*if*, *switch*, *do...while*, *while*, *for*) y sólo se deben utilizar cuando se tiene uno o más bucles (o bloques) dentro de otro bucle y se desea salir (*break*) o continuar con la siguiente iteración (*continue*) de un bucle que no es el actual.

La sentencia ***break labelName*** por lo tanto finaliza el bloque que se encuentre a continuación de ***labelName***. Por ejemplo, en las sentencias,

```
bucleI: // etiqueta o label
for( int i = 0, j = 0; i < 100 ; i++){
    while ( true ) {
       if( (++j) > 5) { break bucleI; }   // Finaliza ambos bucles
       else { break; }                    // Finaliza el bucle interior ( while)
       }
}
```

la expresión `break bucleI;` finaliza los dos bucles simultáneamente, mientras que la expresión break; sale del bucle ***while*** interior y seguiría con el bucle ***for*** en ***i***. Con los valores presentados ambos bucles finalizarán con `i = 5` y `j = 6`.

La sentencia ***continue*** (siempre dentro de al menos un bucle) permite transferir el control a un bucle con nombre o etiqueta. Por ejemplo, la sentencia,

```
continue bucle1;
```

transfiere el control al bucle ***for*** que comienza después de la etiqueta ***bucle1:*** para que realice una nueva iteración:

```
bucle1:
for (int i=0; i<n; i++) {
    bucle2:
    for (int j=0; j<m; j++) {
       ...
       if (expression) continue bucle1; then continue bucle2;
       ...
       }
}
```

6.2. Sentencia return.

Otra forma de salir de un bucle (y de un método) es utilizar la sentencia ***return***. A diferencia de ***continue*** o ***break***, la sentencia ***return*** sale también del método o función. En el caso de que la función devuelva alguna variable, este valor se deberá poner a continuación del return (return value;).

NOTA: HASTA QUE NO SE TENGA UN DOMINIO CORRECTO DE LA PROGRAMACIÓN NO SE DEBEN USAR ESTAS TRES SENTENCIAS DENTRO DE UN BUCLE. LA FORMA CORRECTA DE SALIR DE UN BUCLE ES A TRAVES DE LAS EXPRESIONES CONDICIONALES. HAY LENGUAJES DE PROGRAMACIÓN QUE DAN MUCHOS PROBLEMAS POR SALIR "MAL" DE UN BUCLE.