Sumario

UT 04: Estructuras de control....2
1 Introducción....2
2 Sentencias y bloques....3
3 Estructuras de selección (condicionales)....5
3.1 Estructura if / if-else....5
3.2 Estructura switch....8
4 Estructuras de repetición....11
4.1 Estructura for....11
4.2 Estructura for-each....12
4.3 Estructura while....14
4.4 Estructura do-while....15
5 Estructuras de salto (break, continue)....17
5.1 Sentencias break / continue....17
5.2 Etiquetas....19
5.3 Sentencia return....20
6 Subrutinas. Paso de parámetros y ámbito de variables....21
7 Funciones....29

UT 04: Estructuras de control

1 Introducción

Existen múltiples situaciones que nuestros programas deben representar y que requieren tomar ciertas decisiones, ofrecer diferentes alternativas o llevar a cabo determinadas operaciones repetitivamente para conseguir sus objetivos. La gran mayoría de lenguajes poseen estructuras que permiten a los programadores controlar el flujo de la información de sus programas. Esto realmente es una ventaja para la persona que está aprendiendo un nuevo lenguaje, o tiene previsto aprender más de uno, ya que estas estructuras suelen ser comunes a todos (con algunos cambios de sintaxis o conjunto de reglas que definen las secuencias correctas de los elementos de un lenguaje de programación.).

Para alguien que no ha programado nunca, un ejemplo sencillo le va a permitir entender qué es eso de las sentencias de control de flujo. Piensa en un fontanero (programador), principalmente trabaja con agua (datos) y se encarga de hacer que ésta fluya por donde él quiere (programa) a través de un conjunto de tuberías, codos, latiguillos, llaves de paso, etc. (sentencias de control de flujo). Esas estructuras de control de flujo son las que estudiaremos, conoceremos su estructura, funcionamiento, cómo utilizarlas y dónde. A través de ellas, al construir nuestros programas podremos hacer que los datos (agua) fluyan por los caminos adecuados para representar la realidad del problema y obtener un resultado adecuado.

Los tipos de estructuras de programación que se emplean para el control del flujo de los datos son los siguientes:

- Secuencia: compuestas por 0, 1 o N sentencias que se ejecutan en el orden en que han sido escritas. Es la estructura más sencilla y sobre la que se construirán el resto de estructuras.
- Selección: es un tipo de sentencia especial de decisión y de un conjunto de secuencias de instrucciones asociadas a ella. Según la evaluación de la sentencia de decisión se generará un resultado (que suele ser verdadero o falso) y en función de éste, se ejecutarán una secuencia de instrucciones u otra. Las estructuras de selección podrán ser simples, compuestas y múltiples.
- Iteración: es un tipo de sentencia especial de decisión y una secuencia de instrucciones que pueden ser repetidas según el resultado de la evaluación de la sentencia de decisión. Es decir, la secuencia de instrucciones se ejecutará repetidamente si la sentencia de decisión arroja un valor correcto, en otro la estructura de repetición se detendrá.

Además de las sentencias típicas de control de flujo, en esta unidad haremos una revisión de las sentencias de salto, que aunque no son demasiado recomendables, es necesario conocerlas.

2 Sentencias y bloques

¿Qué es un bloque de sentencias? Es un conjunto de sentencias que se encierra entre llaves y que se ejecutaría como si fuera una única orden. Sirve para agrupar sentencias y para clarificar el código. Los bloques de sentencias son utilizados en Java en la práctica totalidad de estructuras de control de flujo, clases, métodos, etc. Podemos entender dos formas de construir un bloque de sentencias.

En este primer archivo, las sentencias están colocadas en orden secuencial.

```
package organizacion_sentencias;
/**
*
* Organización de sentencias secuencial
*/
public class Organizacion_sentencias_1 {
public static void main(String[] args) {
System.out.println ("Organización secuencial de sentencias");
int dia=12;
System.out.println ("El día es: " + dia);
int mes=11;
System.out.println ("El mes es: " + mes);
int anio=2011;
System.out.println ("El anio es: " + anio);
}
}
```

En este segundo archivo, se declaran al principio las variables necesarias.

```
package organizacion_sentencias;
/**
*
* Organización de sentencias con declaración previa
* de variables
*/
public class Organizacion_sentencias_2 {
public static void main(String[] args) {
// Zona de declaración de variables
int dia=10;
int mes=11;
int anio=2011;
System.out.println ("Organización con declaración previa de variables");
System.out.println ("El día es: " + dia);
System.out.println ("El mes es: " + mes);
System.out.println ("El anio es: " + anio);
}
}
```

En Java no es imprescindible hacerlo así, pero antes de utilizar cualquier variable ésta debe estar previamente declarada. Aunque la declaración de dicha variable puede hacerse en cualquier lugar de nuestro programa.

```
package organizacion_sentencias;
/**
*
* Organización de sentencias en zonas diferenciadas
* según las operaciones que se realicen en el código
*/
public class Organizacion_sentencias_3 {
public static void main(String[] args) {
// Zona de declaración de variables
int dia;
int mes;
int anio;
String fecha;
//Zona de inicialización o entrada de datos
dia=10;
mes=11;
anio=2011;
fecha="";
//Zona de procesamiento
fecha=dia+"/"+mes+"/"+anio;
//Zona de salida
System.out.println ("Organización con zonas diferenciadas en el código");
System.out.println ("La fecha es: " + fecha);
}
}
```

En este tercer archivo se ha organizado el código en las siguientes partes:

- declaración de variables
- petición de datos de entrada
- procesamiento de dichos datos y obtención de la salida.

Este tipo de organización está más estandarizada y hace que nuestros programas ganen en legibilidad. En todo caso se deben tener en cuenta siempre en Java las siguientes premisas:

- Declara cada variable antes de utilizarla.
- Inicializa con un valor cada variable la primera vez que la utilices.
- No es recomendable usar variables no inicializadas en nuestros programas,

pueden provocar errores o resultados imprevistos.

3 Estructuras de selección (condicionales)

Estas estructuras constan de una sentencia especial de decisión y de un conjunto de secuencias de instrucciones. La sentencia de decisión será evaluada y ésta devolverá un valor (verdadero o falso). En función del valor devuelto se ejecutará una secuencia de instrucciones u otra.

Por ejemplo, si el valor de una variable es mayor o igual que 5 se imprime por pantalla la palabra APROBADO y si es menor, se imprime SUSPENSO. Para este ejemplo, la comprobación del valor de la variable será la sentencia especial de decisión. La impresión de la palabra APROBADO será una secuencia de instrucciones y la impresión de la palabra SUSPENSO será otra. Cada secuencia estará asociada a cada uno de los resultados que puede arrojar la evaluación de la sentencia especial de decisión.

En el lenguaje de programación C, verdadero o falso se representan mediante un literal entero. 0 representará Falso y 1 o cualquier otro valor, representará Verdadero. En Java las variables de tipo booleano sólo podrán tomar los valores true (verdadero) o false (falso).

La evaluación de las sentencias de decisión o expresiones que controlan las estructuras de selección devolverán siempre un valor verdadero o falso.

Las estructuras de selección se dividen en:

- Estructuras de selección simples o estructura if.
- Estructuras de selección compuesta o estructura if – else.
- Estructuras de selección basadas en el operador condicional.
- Estructuras de selección múltiples o estructura switch.

3.1 Estructura if / if-else

La estructura if es una estructura de selección o estructura condicional, en la que se evalúa una expresión lógica o sentencia de decisión y en función del resultado, se ejecuta una sentencia o un bloque de éstas.

La estructura if puede presentarse de las siguientes formas:

Estructura if simple.

```
if (expresión-lógica)
sentencia1;
if (expresión-lógica)
{
sentencia1;
sentencia2;
...;
sentenciaN;
```

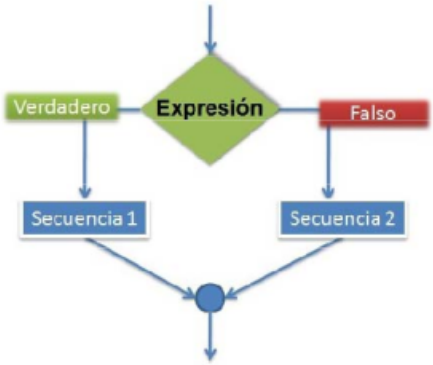
```
}
```

Si la evaluación de la expresión lógica ofrece un resultado verdadero, se ejecuta la sentencia1 o bien el bloque de sentencias asociado. Si el resultado de dicha evaluación es falso, no se ejecutará ninguna instrucción asociada a la estructura condicional.

Estructura if de doble alternativa.

```
if (expresión-lógica)
{
sentencia1;
if (expresión-lógica)
...;
sentencia1;
sentenciaN;
else
} else {
sentencia1;
sentencia2;
...;
sentenciaN;
}
```

Si la evaluación de la expresión lógica ofrece un resultado verdadero, se ejecutará la primera sentencia o el primer bloque de sentencias. Si, por el contrario, la evaluación de la expresión lógica ofrece un resultado falso, no se ejecutará la primera sentencia o el primer bloque y sí se ejecutará la segunda sentencia o el segundo bloque.



Haciendo una interpretación cercana al pseudocódigo tendríamos que si se cumple la condición (expresión lógica) se ejecutará un conjunto de instrucciones, y si no se cumple se ejecutará otro conjunto de instrucciones.

Hay que tener en cuenta que la cláusula else de la sentencia if no es obligatoria. En algunos casos no necesitaremos utilizarla, pero sí se recomienda cuando es necesario llevar a cabo alguna acción en el caso de que la expresión lógica no se cumpla.

En aquellos casos en los que no existe cláusula else, si la expresión lógica es falsa, simplemente se continuarán ejecutando las siguientes sentencias que aparezcan bajo la estructura condicional if.

Los condicionales if e if-else pueden anidarse, de tal forma que dentro de un bloque de sentencias puede incluirse otro if o if-else. El nivel de anidamiento queda a criterio del

programador, pero si éste es demasiado profundo podría provocar problemas de eficiencia y legibilidad en el código. En otras ocasiones, un nivel de anidamiento excesivo puede denotar la necesidad de utilización de otras estructuras de selección más adecuadas. Cuando se utiliza anidamiento de este tipo de estructuras, es necesario poner especial atención en saber a qué if está asociada una cláusula else. Normalmente, un else estará asociado con el if inmediatamente superior o más cercano que exista dentro del mismo bloque y que no se encuentre ya asociado a otro else.

```
package sentencias_condicionales;
/**
*
* Ejemplos de utilización de diferentes estructuras
* condicionales simples, completas y anidamiento de éstas.
*/
public class Sentencias_condicionales {
/*Vamos a realizar el cálculo de la nota de un examen
* de tipo test. Para ello, tendremos en cuenta el número
* total de pregunta, los aciertos y los errores. Dos errores
* anulan una respuesta correcta.
*
* Finalmente, se muestra por pantalla la nota obtenida, así
* como su calificación no numérica.
*
* La obtención de la calificación no numérica se ha realizado
* combinando varias estructuras condicionales, mostrando expresiones
* lógicas compuestas, así como anidamiento.
*
*/
public static void main(String[] args) {
// Declaración e inicialización de variables
int num_aciertos = 12;
int num_errores = 3;
int num_preguntas = 20;
float nota = 0;
String calificacion="";
//Procesamiento de datos
nota = ((num_aciertos - (num_errores/2))*10)/num_preguntas;
if (nota < 5)
{
calificacion="INSUFICIENTE";
}
else
{
/* Cada expresión lógica de estos if está compuesta por dos
* expresiones lógicas combinadas a través del operador Y o AND
* que se representa con el símbolo &&. De tal manera, que para
* que la expresión lógica se cumpla (sea verdadera) la variable
* nota debe satisfacer ambas condiciones simultáneamente
```

```
*/
if (nota >= 5 && nota <6)
calificacion="SUFICIENTE";
if (nota >= 6 && nota <7)
calificacion="BIEN";
if (nota >= 7 && nota <9)
calificacion="NOTABLE";
if (nota >= 9 && nota <=10)
calificacion="SOBRESALIENTE";
}
//Salida de información
System.out.println ("La nota obtenida es: " + nota);
System.out.println ("y la calificación obtenida es: " + calificacion);
}
}
```

3.2 Estructura switch

¿Qué podemos hacer cuando nuestro programa debe elegir entre más de dos alternativas?, una posible solución podría ser emplear estructuras if anidadas, aunque no siempre esta solución es la más eficiente. Cuando estamos ante estas situaciones podemos utilizar la estructura de selección múltiple switch. En la siguiente tabla se muestra tanto la sintaxis, como el funcionamiento de esta estructura.

Sintaxis:

```
switch (expresion) {
case valor1:
sentencia1_1;
sentencia1_2;
....
break;
....
....
case valorN:
sentenciaN_1;
sentenciaN_2;
....
break;
default:
sentencias-default;
}
```

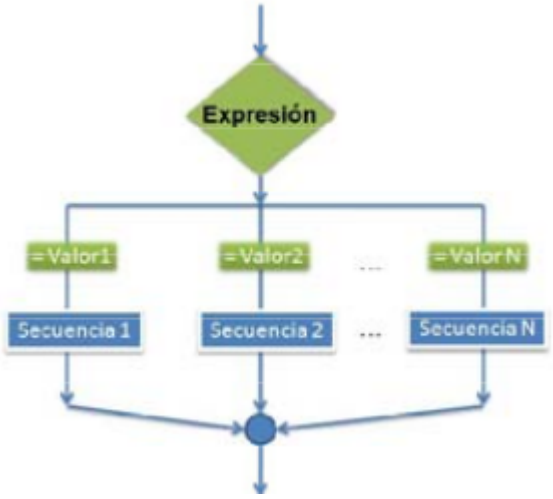
Condiciones:

- Donde expresión debe ser del tipo char, byte, short o int, y las constantes de cada case deben ser de este tipo o de un tipo compatible.

- La expresión debe ir entre paréntesis.
- Cada case llevará asociado un valor y se finalizará con dos puntos.
- El bloque de sentencias asociado a la cláusula default puede finalizar con una sentencia de ruptura break o no.

Funcionamiento:

- Las diferentes alternativas de esta estructura estarán precedidas de la cláusula case que se ejecutará cuando el valor asociado al case coincida con el valor obtenido al evaluar la expresión del switch.
- En las cláusulas case, no pueden indicarse expresiones condicionales, rangos de valores o listas de valores. (otros lenguajes de programación sí lo permiten). Habrá que asociar una cláusula case a cada uno de los valores que deban ser tenidos en cuenta.
- La cláusula default será utilizada para indicar un caso por defecto, las sentencias asociadas a la cláusula default se ejecutarán si ninguno de los valores indicados en las cláusulas case coincide con el resultado de la evaluación de la expresión de la estructura switch.
- La cláusula default puede no existir, y por tanto, si ningún case ha sido activado finalizaría el switch.
- Cada cláusula case puede llevar asociadas una o varias sentencias, sin necesidad de delimitar dichos bloques por medio de llaves.
- En el momento en el que el resultado de la evaluación de la expresión coincide con alguno de los valores asociados a las cláusulas case, se ejecutarán todas las instrucciones asociadas hasta la aparición de una sentencia break de ruptura. (la sentencia break se analizará en epígrafes posteriores)



En resumen, se ha de comparar el valor de una expresión con un conjunto de constantes, si el valor de la expresión coincide con algún valor de dichas constantes, se ejecutarán los bloques de instrucciones asociados a cada una de ellas. Si no existiese coincidencia, se ejecutarían una serie de instrucciones por defecto.

```
package sentencias_condicionales;
/**
*
* @author Pc
*/
public class condicional_switch {
/*Vamos a realizar el cálculo de la nota de un examen
* de tipo test. Para ello, tendremos en cuenta el número
* total de preguntas, los aciertos y los errores. Dos errores
* anulan una respuesta correcta.
*
* La nota que vamos a obtener será un número entero.
*
* Finalmente, se muestra por pantalla la nota obtenida, así
* como su calificación no numérica.
*
* La obtención de la calificación no numérica se ha realizado
* utilizando la estructura condicional múltiple o switch.
*
*/
public static void main(String[] args) {
// Declaración e inicialización de variables
int num_aciertos = 17;
int num_errores = 3;
int num_preguntas = 20;
int nota = 0;
String calificacion="";
//Procesamiento de datos
nota = ((num_aciertos - (num_errores/2))*10)/num_preguntas;
switch (nota) {
case 5: calificacion="SUFICIENTE";
break;
case 6: calificacion="BIEN";
break;
case 7: calificacion="NOTABLE";
break;
case 8: calificacion="NOTABLE";
break;
case 9: calificacion="SOBRESALIENTE";
break;
case 10: calificacion="SOBRESALIENTE";
break;
```

```
default: calificacion="INSUFICIENTE";
}
//Salida de información
System.out.println ("La nota obtenida es: " + nota);
System.out.println ("y la calificación obtenida es: " + calificacion);
}
}
```

4 Estructuras de repetición

En Java existen cuatro clases de bucles:

- Bucle for (repite para)
- Bucle for/in (repite para cada)
- Bucle While (repite mientras)
- Bucle Do While (repite hasta)

Los bucles for y for/in se consideran bucles controlados por contador. Por el contrario, los bucles while y do...while se consideran bucles controlados por sucesos.

4.1 Estructura for

El bucle for es un bucle controlado por contador. Este tipo de bucle tiene las siguientes características:

- Se ejecuta un número determinado de veces.
- Utiliza una variable contadora que controla las iteraciones del bucle.

En general, existen tres operaciones que se llevan a cabo en este tipo de bucles:

- Se inicializa la variable contadora.
- Se evalúa el valor de la variable contador, por medio de una comparación de su valor con el número de iteraciones especificado.
- Se modifica o actualiza el valor del contador a través de incrementos o decrementos de éste, en cada una de las iteraciones.

Sintaxis:

```
for (inicialización; condición; iteración)
```

- condición es una expresión que evaluará la variable de control. Mientras la condición sea falsa, se repetirá el cuerpo del bucle. Cuando la condición se cumpla, terminará la ejecución del bucle.
- iteración indica la manera en la que la variable de control va cambiando en cada iteración del bucle. Podrá ser mediante incremento o decremento, y no solo de uno en uno.

```
public class repetitiva_for {
/* En este ejemplo se utiliza la estructura repetitiva for
* para representar en pantalla la tabla de multiplicar del siete
*/
public static void main(String[] args) {
// Declaración e inicialización de variables
int numero = 7;
int contador;
int resultado=0;
//Salida de información
System.out.println ("Tabla de multiplicar del " + numero);
System.out.println (".............................. ");
//Utilizamos ahora el bucle for
for (contador=1; contador<=10;contador++)
/* La cabecera del bucle incorpora la inicialización de la variable
* de control, la condición de multiplicación hasta el 10 y el
* incremento de dicha variable de uno en uno en cada iteración del
* bucle.
* En este caso contador++ incrementará en una unidad el valor de
* dicha variable.
*/
{
resultado = contador * numero;
System.out.println(numero + " x " + contador + " = " + resultado);
/* A través del operador + aplicado a cadenas de caracteres,
* concatenamos los valores de las variables con las cadenas de
* caracteres que necesitamos para representar correctamente la
* salida de cada multiplicación.
*/
}
}
}
```

4.2 Estructura for-each

Junto a la estructura for, for-each también se considera un bucle controlado por contador. Este bucle es una mejora incorporada en la versión 5.0. de Java. Este tipo de bucles

permite realizar recorridos sobre arrays y colecciones de objetos. Los arrays son colecciones de variables que tienen el mismo tipo y se referencian por un nombre común. Así mismo, las colecciones de objetos son objetos que se dice son iterables, o que se puede iterar sobre ellos.

Este bucle es nombrado también como bucle for mejorado, o bucle foreach. En otros lenguajes de programación existen bucles muy parecidos a este.

La sintaxis es:

```
for (declaración: expresión) {
sentencia1;
...
sentenciaN;
}
```

- expresión es un array o una colección de objetos.
- declaración es la declaración de una variable cuyo tipo sea compatible con expresión. Normalmente, será el tipo y el nombre de la variable a declarar.

Para cada elemento de la expresión, guarda el elemento en la variable declarada y ejecuta las instrucciones contenidas en el bucle. Después, en cada una de las iteraciones del bucle tendremos en la variable declarada el elemento actual de la expresión. Por tanto, para el caso de los arrays y de las colecciones de objetos, se recorrerá desde el primer elemento que los forma hasta el último.

A continuación vemos un ejemplo de bucle de este tipo y su utilización sobre un array.

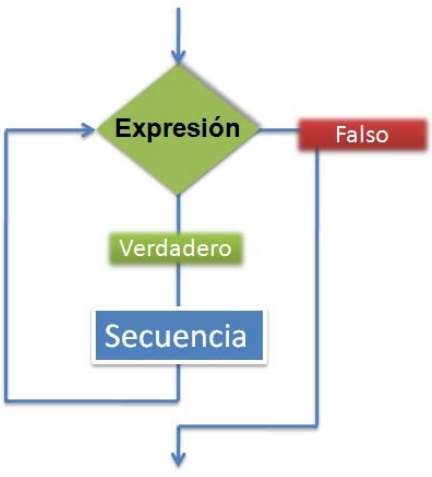
```
1
2   public class repetitiva_for_in {
3       public static void main(String[] args) {
4           // Declaración e inicialización de variables
5           String[] semana = {"Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"};
6
7           //Salida de información
8
9           //Utilizamos ahora el bucle for/in
10          for (String dia: semana){
11          /* La cabecera del bucle incorpora la declaración de la variable dia
12           * a modo de contenedor temporal de cada uno de los elementos que forman
13           * el array semana.
14           * En cada una de las iteraciones del bucle, se irá cargando en la variable
15           * dia el valor de cada uno de los elementos que forman el array semana,
16           * desde el primero al último.
17           */
18              System.out.println(dia);
19
20          }
21      }
22
23  }
24
```

Los bucles for-each permitirán al programador despreocuparse del número de veces que se ha de iterar, pero no sabremos en qué iteración nos encontramos salvo que se añada artificialmente alguna variable contadora que nos pueda ofrecer esta información.

4.3 Estructura while

El bucle while es la primera de las estructuras de repetición controladas por sucesos que vamos a estudiar. La utilización de este bucle responde al planteamiento de la siguiente pregunta: ¿Qué podemos hacer si lo único que sabemos es que se han de repetir un conjunto de instrucciones mientras se cumpla una determinada condición? La característica fundamental de este tipo de estructura repetitiva estriba en ser útil en aquellos casos en los que las instrucciones que forman el cuerpo del bucle podría ser necesario ejecutarlas o no. Es decir, en el bucle while siempre se evaluará la condición que lo controla, y si dicha condición es cierta, el cuerpo del bucle se ejecutará una vez, y se seguirá ejecutando mientras la condición sea cierta. Pero si en la evaluación inicial de la condición ésta no es verdadera, el cuerpo del bucle no se ejecutará.

Es imprescindible que en el interior del bucle while se realice alguna acción que modifique la condición que controla la ejecución del mismo, ya que en caso contrario estaríamos ante un bucle infinito.



Sintaxis:

```
while (condición)
sentencia;

while (condición) {
sentencia1;
...
sentenciaN;
}
```

En el momento en el que la condición no se cumpla, el control del flujo del programa pasará a la siguiente instrucción que exista justo detrás del bucle while. La condición se evaluará siempre al principio, y podrá darse el caso de que las instruccionescontenidas en él no lleguen a ejecutarse nunca si no se satisface la condición de partida.

```
public class repetitiva_while {
public static void main(String[] args) {
// Declaración e inicialización de variables
int numero = 7;
int contador;
int resultado=0;
//Salida de información
System.out.println ("Tabla de multiplicar del " + numero);
System.out.println (".............................. ");

//Utilizamos ahora el bucle while
contador = 1; //inicializamos la variable contadora
while (contador <= 10) //Establecemos la condición del bucle
{
resultado = contador * numero;
System.out.println(numero + " x " + contador + " = " + resultado);
//Modificamos el valor de la variable contadora, para hacer que el
//bucle pueda seguir iterando hasta llegar a finalizar
contador++;
}
}
}
```

4.4 Estructura do-while

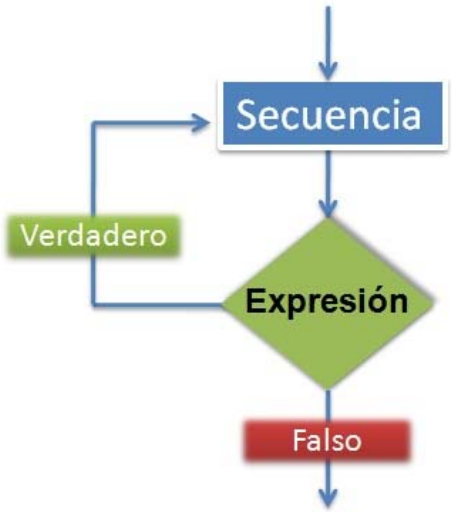
La segunda de las estructuras repetitivas controladas por sucesos es do - while. En este

caso, la pregunta que nos planteamos es la siguiente: ¿Qué podemos hacer si lo único que sabemos es que se han de ejecutar, al menos una vez, un conjunto de instrucciones y seguir repitiéndose hasta que se cumpla una determinada condición?.

La característica fundamental de este tipo de estructura repetitiva estriba en ser útil en aquellos casos en los que las instrucciones que forman el cuerpo del bucle necesitan ser ejecutadas, al menos, una vez y repetir su ejecución hasta que la condición sea verdadera. Por tanto, en esta estructura repetitiva siempre se ejecuta el cuerpo del bucle una primera vez. Es imprescindible que en el interior del bucle se realice alguna acción que modifique la condición que controla la ejecución del mismo, ya que en caso contrario estaríamos ante un bucle infinito.

```
do
sentencia;
while (condición);
```

```
do {
     sentencia1;
     ...
     sentenciaN;
}
while (condición);
```



Veamos un ejemplo:

```
public class repetitiva_dowhile {
public static void main(String[] args) {
// Declaración e inicialización de variables
int numero = 7;
int contador;
int resultado=0;
//Salida de información
System.out.println ("Tabla de multiplicar del " + numero);
System.out.println (".............................. ");
//Utilizamos ahora el bucle do-while
contador = 1; //inicializamos la variable contadora
do
{
resultado = contador * numero;
System.out.println(numero + " x " + contador + " = " + resultado);
//Modificamos el valor de la variable contadora, para hacer que el
//bucle pueda seguir iterando hasta llegar a finalizar
```

```
contador++;
}while (contador <= 10); //Establecemos la condición del bucle
}
}
```

5 Estructuras de salto (break, continue)

En la gran mayoría de libros de programación y publicaciones de Internet se nos recomienda que prescindamos de sentencias de salto incondicional, es más, se desaconseja su uso por provocar una mala estructuración del código y un incremento en la dificultad para el mantenimiento de los mismos. Pero Java incorpora ciertas sentencias o estructuras de salto que podrían ser útiles en algunos casos.

Estas estructuras de salto corresponden a las sentencias break, continue, las etiquetas de salto y la sentencia return. Pasamos ahora a analizar su sintaxis y funcionamiento.

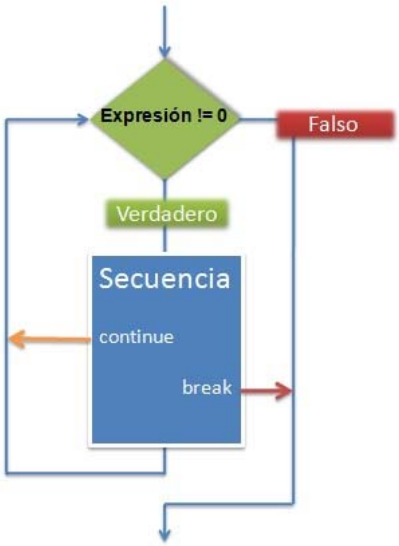
5.1 Sentencias break / continue

Se trata de dos instrucciones que permiten modificar el comportamiento de otras estructuras o sentencias de control, simplemente por el hecho de estar incluidas en algún punto de su secuencia de instrucciones.

La sentencia break incidirá sobre las estructuras de control **switch, while, for y do / while** del siguiente modo:

- Si aparece una sentencia break dentro de la secuencia de instrucciones de cualquiera de las estructuras mencionadas anteriormente, dicha estructura terminará inmediatamente.
- Si aparece una sentencia break dentro de un bucle anidado sólo finalizará la sentencia de iteración más interna, el resto se ejecuta de forma normal.

Es decir, **break** sirve para romper el flujo de control de un bucle, aunque no se haya cumplido la condición del bucle. Si colocamos un break dentro del código de un bucle, cuando se alcance el break, automáticamente se saldrá del bucle pasando a ejecutarse la siguiente instrucción inmediatamente después de él.



La sentencia **continue** incidirá sobre las sentencias o estructuras de control while, for y do / while del siguiente modo:

- Si aparece una sentencia continue dentro de la secuencia de instrucciones de cualquiera de las sentencias anteriormente indicadas, dicha sentencia dará por terminada la iteración actual y se ejecuta una nueva iteración, evaluando de nuevo la expresión condicional del bucle.

- Si aparece en el interior de un bucle anidado solo afectará a la sentencia de iteración más interna, el resto se ejecutaría de forma normal. Es decir, la sentencia continue forzará a que se ejecute la siguiente iteración del bucle, sin tener en cuenta las instrucciones que pudiera haber después del continue, y hasta el final del código del bucle.

```
6   public class sentencia_break {
7       public static void main(String[] args) {
8           // Declaración de variables
9           int contador;
10
11
12          //Procesamiento y salida de información
13
14          for (contador=1;contador<=10;contador++)
15          {
16              if (contador==7)
17                  break;
18              System.out.println ("Valor: " + contador);
19          }
20           System.out.println ("Fin del programa");
21          /* El bucle sólo se ejecutará en 6 ocasiones, ya que cuando
22           * la variable contador sea igual a 7 encontraremos un break que
23           * romperá el flujo del bucle, transfiriéndonos a la sentencia que
24           * imprime el mensaje de Fin del programa.
25           */
26      }
```

Veamos cómo utilizar la sentencia **continue** en un bucle "for" para imprimir por pantalla solo los números pares:

```
 * Uso de la sentencia continue
 */
public class sentencia_continue {
    public static void main(String[] args) {
        // Declaración de variables
        int contador;

        System.out.println ("Imprimiendo los números pares que hay del 1 al 10... ");
        //Procesamiento y salida de información

        for (contador=1;contador<=10;contador++)
        {
            if (contador % 2 != 0) continue;
            System.out.print(contador + " ");
        }
         System.out.println ("\nFin del programa");
        /* Las iteraciones del bucle que generarán la impresión de cada uno
         * de los números pares, serán aquellas en las que el resultado de
         * calcular el resto de la división entre 2 de cada valor de la variable
         * contador, sea igual a 0.
         */
    }

}

```

5.2 Etiquetas

Los saltos incondicionales y, en especial los saltos a una etiqueta, son totalmente desaconsejables. No obstante, Java permite asociar etiquetas cuando se va a realizar un salto. De este modo puede conseguirse algo más de legibilidad en el código.

Las estructuras de salto break y continue, pueden tener asociadas etiquetas. Es a lo que se llama un break etiquetado o un continue etiquetado. Pero sólo se recomienda su uso cuando se hace necesario salir de bucles anidados hacia diferentes niveles.

¿Y cómo se crea un salto a una etiqueta? En primer lugar, crearemos la etiqueta mediante un identificador seguido de dos puntos (:). A continuación, se escriben las sentencias Java asociadas a dicha etiqueta encerradas entre llaves. Por así decirlo, la creación de una etiqueta es como fijar un punto de salto en el programa para poder saltar a él desde otro lugar de dicho programa.

¿Cómo se lleva a cabo el salto? Es sencillo, en el lugar donde vayamos a colocar la sentencia break o continue, añadiremos detrás el identificador de la etiqueta. Con ello, conseguiremos que el salto se realice a un lugar determinado.

La sintaxis será:

```
break <etiqueta>
```

Esta herramienta tiene cierta similitud con las anclas que pueden crearse en el interior de una página web con HTML, a las que nos llevará un hiperenlace. También es similar a la

etiqueta usada en archivos batch bajo MSDOS, combinada con la sentencia GOTO. A continuación vemos un ejemplo de declaración y uso de etiquetas en un bucle. Como se puede apreciar, las sentencias asociadas a cada etiqueta están encerradas entre llaves para delimitar así su ámbito de acción.

```
1  /**
2   *
3   * Uso de etiquetas en bucle
4   */
5  public class etiquetas {
6      public static void main(String[] args) {
7
8          for (int i=1; i<3; i++) //Creamos cabecera del bucle
9          {
10             bloque_uno: {   //Creamos primera etiqueta
11                 bloque_dos:{    //Creamos segunda etiqueta
12                     System.out.println("Iteración: "+i);
13                     if (i==1) break bloque_uno;   //Llevamos a cabo el primer salto
14                     if (i==2) break bloque_dos;   //Llevamos a cabo el segundo salto
15                 }
16                 System.out.println("después del bloque dos");
17             }
18             System.out.println("después del bloque uno");
19         }
20         System.out.println("Fin del bucle");
21     }
22 }
```

5.3 Sentencia return

Se usa para modificar la ejecución de un método. La sentencia return puede utilizarse de dos formas:

- Para terminar la ejecución del método donde esté escrita, con lo que transferirá el control al punto desde el que se hizo la llamada al método, continuando el programa por la sentencia inmediatamente posterior.
- Para devolver un valor, siempre que junto a return se incluya una expresión de un tipo determinado. En el lugar donde se invocó del método se obtendrá el valor resultante de la evaluación de la expresión que acompañaba al método.

En general, una sentencia return suele aparecer al final de un método, de este modo el método tendrá una entrada y una salida. También es posible utilizar una sentencia return en cualquier punto de un método, con lo que éste finalizará en el lugar donde se encuentre dicho return. No será recomendable incluir más de un return en un método y por regla general, deberá ir al final del método.

El valor de retorno es opcional. Si lo hubiera, debería de ser del mismo tipo o de un tipo compatible al tipo del valor de retorno definido en la cabecera del método, pudiendo ser desde un entero a un objeto creado por nosotros. Si no lo tuviera, el tipo de retorno sería void, y return serviría para salir del método sin necesidad de llegar a ejecutar todas las

instrucciones que se encuentran después del return.

```
import java.io.*;
/**
*
* Uso de return en un método
*/
public class sentencia_return {
private static BufferedReader stdin=new BufferedReader(new
InputStreamReader(System.in));
public static int suma(int numero1, int numero2)
{
int resultado;
resultado = numero1 + numero2;
return resultado; //Devolvemos el resultado de la suma
}
public static void main(String[] args) throws IOException {
//Declaración de variables
String input; //Esta variable recibirá la entrada de teclado
int primer_numero, segundo_numero; //Almacenarán los operandos
// Solicitamos que el usuario introduzca dos números por consola
System.out.print ("Introduce el primer operando:");
input = stdin.readLine(); //Leemos cadena de caracteres
primer_numero = Integer.parseInt(input); //Transformamos a entero
System.out.print ("Introduce el segundo operando: ");
input = stdin.readLine(); //Leemos cadena de caracteres
segundo_numero = Integer.parseInt(input); //Transformamos a entero
//Imprimimos los números introducidos
System.out.println ("Los operandos son: " + primer_numero + " y " +
segundo_numero);
System.out.println ("obteniendo su suma... ");
//Invocamos al método que realiza la suma, pasándole los parámetros
adecuados
System.out.println("La suma de ambos operandos es:" +
suma(primer_numero,segundo_numero));
}
}
```

6 Subrutinas. Paso de parámetros y ámbito de variables.

Una de las estrategias de programación consiste en dividir programas complejos en trozos más pequeños y manejables, mediante el uso de subrutinas. Una **subrutina** consiste en un grupo de instrucciones juntas que realizan una determinada tarea. En alguna otra parte del programa se "llama" a la subrutina, es decir, se usa su nombre como

si se tratara del conjunto de instrucciones completo. Las subrutinas pueden usarse una y otra vez, en diferentes puntos del programa. Se puede emplear una subrutina incluso dentro de otra subrutina. Las subrutinas pueden, a su vez, utilizarse en otras subrutinas más complejas.

En Java, cualquier subrutina debe formar parte de una clase (class) u objeto. Estas subrutinas son llamados **métodos**.

Un método **static** es parte de una clase, más que parte del objeto definido por esa clase. Los métodos static corresponden directamente a las subrutinas tradicionales de los lenguajes de programación no orientados a objetos.

Los métodos pertenecientes a objetos son también llamados "métodos instancia".

A veces a la subrutina se la considera una “caja negra” porque no se sabe lo que contiene (o, para ser mas precisos, normalmente no se quiere saber lo que contiene, para evitar entrar en la complejidad que la subrutina esta ocultando). Naturalmente, una “caja negra” que no pueda interactuar con el resto del mundo, no sirve para nada. La “caja negra” necesita algún tipo de interface con el resto del mundo, que permita interactuar con lo que hay dentro de la caja y aprovechar la salida.

Una caja negra física, puede tener botones en el exterior, que se puedan pulsar, diales que se puedan mover y conectores que permitan pasar información hacia dentro y hacia fuera. Puesto que la idea es esconder la complejidad, y no crearla, tenemos la primera regla de las cajas negras:

"La interface con una caja negra, debe ser directa, bien definida, y fácil de entender".

¿Existe algún ejemplo de caja negra en la vida real?. Si; estamos rodeados de ellas. Un televisor, un coche, una nevera,… Un usuario puede encender y apagar el televisor, cambiar canales y ajustar el volumen usando los elementos de su interface — botones, mandos remotos— y todo eso lo puede hacer sin entender absolutamente nada acerca de como trabaja el dispositivo. La caja negra tiene cosas en su interior — en la subrutina, el código para realizar una tarea, en el televisor, toda su electrónica–. Lo que hay dentro de una caja negra, se llama **implementacion**. Podemos considerar una segunda regla:

"Para utilizar una caja negra, no es necesario conocer absolutamente nada de su implementacion; solo es necesario conocer su interface."

De hecho, es posible cambiar la implementacion sin cambiar el comportamiento de la caja, de tal forma que desde fuera todo parezca igual. Por ejemplo, cuando se cambió el interior del televisor, que estaba construido con válvulas de vacío, y se utilizaron transistores, el usuario no necesitó conocer nada mas, nada cambió para él. Igualmente es posible el reescribir el interior de una rutina, para usar mas eficientemente el código, por ejemplo, sin que afecte en nada al programa que usa la subrutina. La idea de trabajar con cajas negras aporta ventajas tanto al constructor de ellas, como a su usuario. La caja negra puede usarse en un numero ilimitado de situaciones distintas.

El implementador (programador que la escribe), no necesita conocer nada sobre los sitios en donde se usaran. El implementador sólo necesita asegurarse que la caja realizará correctamente el trabajo que se le asigne y que la interface se relaciona correctamente

con el resto del mundo. Esta es la tercera regla de las cajas negras:

"El implementador de la caja negra no necesita conocer nada acerca del sistema completo en donde se utilizará la caja."

De esta forma, la caja negra divide al mundo en dos partes: la interior (implementacion) y la exterior. La interface es la frontera que conecta ambas partes. Además de ser una conexión con el exterior, la interface también incluye las especificaciones sobre qué hace la caja y cómo se controla usando los elementos de la interface física.

No basta decir que el televisor tiene un interruptor para ponerla en marcha, sino que se necesita especificar también cuál es el interruptor que sirve para encenderla y apagarla.

Para poner todo esto en términos informáticos, la interface de una subrutina tiene un componente **semántico** y otro **sintáctico**. La parte sintáctica de la interface le dice qué tiene que escribir para poder llamar a la subrutina. El componente semántico especifica exactamente que tarea realiza la subrutina.

Para escribir correctamente un programa hay que conocer las especificaciones sintácticas de la subrutina. Para entender el propósito de la subrutina y que su uso sea efectivo, hay que conocer las especificaciones semánticas de la subrutina.

En Java, todos los métodos se definen dentro de una clase. Esto hace a Java tan especial en medio de los lenguajes de programación, ya que algunos permiten que los métodos existan libremente. Uno de los propósitos de la clase es poder agrupar juntos los métodos relacionados y las variables. El hecho de agrupar estos métodos en clases (y las clases en paquetes) ayuda a controlar la confusión resultante por tanto nombre distinto.

La definición de un método en Java tiene la siguiente forma:

```
 modificadores tipo-retorno nombre-método ( lista-parámetros ) {
       instrucciones
}
```

Los **modificadores** son palabras que definen ciertas características del método (por ejemplo, si es static o no). Las instrucciones se escribirse entre llaves, { y }:

```
static void playGame() {
// “static” es el modificador; “void” es tipo-retorno;
          // “playGame” es el nombre; lista de parámetros vacía
          . . .
// las instrucciones para definir el método van aquí
}
```

En este ejemplo, getNextN es un método non-static dado que en su definición no incluye el modificador “static”:

```
int getNextN(int N) {
// no hay modificadores; “int” es el tipo retorno
// “getnextN” es el nombre del método;
```

```
         // la lista de parámetros incluye uno
        // llamado “N” y con tipo “int”
        . . .
// instrucciones para definir el método
}
```

Con el modificador "public" se indica que el método puede ser llamado desde cualquier lugar del programa e incluso desde fuera de la clase en donde se ha definido.

```
public static boolean lessThan(double x, double y) {
// “public” y “static” son modificadores;
// “boolean” es el tipo a retornar;
// “lessThan” es el nombre del método;
// la lista de parámetros incluye dos con
// nombres “x” e “y”, el tipo de cada parámetro
// es “double”
. . .
// instrucciones para implementar el método
}
```

Un modificador “private” indicaría que el método podría ser llamado únicamente desde dentro de la clase. Los modificadores public y private se llaman **limitadores de acceso** (access specifiers). Si en un método no se definen limitadores de acceso, por defecto puede ser llamado por cualquiera dentro del “package”. Para entender la necesidad de definir limitadores de acceso, pensemos que los métodos públicos son parte de la interface de la caja negra, mientras que los métodos privados son parte de la implementacion oculta de la caja.

Las reglas de las cajas negras implican que solo se debe declarar un método como publico si es realmente una parte importante de la interface que se ve desde fuera.

Las subrutinas diseñadas para calcular y devolver valores se llaman **funciones**. En ellas hay que especificar el tipo de retorno del método, que será el tipo de valor que se calcula con la función. Por ejemplo el tipo de retorno boleano en

```
public static boolean lessThan(double x, double y)
```

indica que la función lessThan calcula un valor boleano.

Un método también puede ser usado como una subrutina normal, que no devuelve ningún valor, especificando como tipo de retorno “void”. Este termino se emplea para indicar que no existe valor de retorno o que este está vacío.

Finalmente viene la **lista de parámetros** del método. Los parámetros forman parte de la interface del método, representan la información que se pasa al metodo desde el exterior, para ser usada en sus cálculos internos. Por ejemplo, una clase llamada Televisión incluye el método changeChanel(). Pero... ¿A que canal queremos cambiar?.

Dado que el numero de canal es un entero, el tipo del parámetro que se dará a la función será de tipo int y la declaración del método changeChanel() podría ser:

```
public void changeChannel(int channelNum) {…}
```

Cuando se llama al método se le debe proporcionar un numero de canal. Por ejemplo:

```
changeChanel(127);
```

Los métodos no se ejecutan hasta que son llamados, excepto el método "main", que se ejecuta directamente al arrancar el programa. Por ejemplo el método playGame() definido anteriormente, puede ser llamado utilizando la siguiente instrucción desde dentro de la clase:

```
playGame();
```

Esta instrucción puede aparecer en cualquier sitio de la misma clase que incluye la definición de playGame(), desde el método main() o desde cualquier otro método. Dado que playGame() en un método público, también puede ser llamado desde otras clases, pero en ese caso, hay que especificar qué clase lo contiene. Supongamos por ejemplo que playGame() esta definido en una clase llamada Poker. Para efectuar la llamada desde fuera de la clase, debería escribir:

```
Poker.playGame();
```

El mismo nombre de método podría estar definido en otra clase, por ejemplo:

```
Blackjack.playGame().
```

De forma general, la instrucción de llamada a un método tiene la forma:

- `nombre-método(parámetros);` => Método definido en la misma clase.
- `nombre-clase.nombre-método(parámetros);` => Método static definido en cualquier parte, en diferente clase

Aquí vemos un ejemplo de boceto de un programa "casi" completo:

```
public class GuessingGame {
            public static void main(String[] args) {
               Console console = new Console();
               console.putln("Quiero jugar.  Pensaré un número entre");
               console.putln("1 y 100, y pruebe a ver si lo adivina.");
               boolean playAgain;
               do {
                  playGame();  // llama a la subrutina para jugar
                  console.put("Quiere jugar otra vez? ");
                  playAgain = console.getlnBoolean();
               } while (playAgain);
               console.close();
            } // end of main()
            static void playGame() {
               . . .  // instrucciones de implementacion del juego
            } // end of playGame()
```

```
        } // end of class GuessingGame
```

Esto es el boceto del posible programa para jugar, pero tenemos un pequeño problema. Necesitamos hacer preguntas y leer sus respuestas (usar la consola), pero "console" es una variable local definida dentro del método main(). Una variable definida localmente, dentro de un método, es parte de la caja negra, por tanto inaccesible desde fuera. No hay forma de usar la consola desde dentro de otro modo como playGame(). Si definimos en playGame() una variable Console será completamente distinta a la console definida en main() — lo que generará dos ventanas distintas en la pantalla. Como solución sencilla, en lugar de inicializar en el método una variable local, console puede ser una variable static perteneciente a la clase GuessingGame. La variable puede ser declarada en la clase fuera de cualquier método. Una variable local solo existe mientras el bloque se ejecuta. Sin embargo, una variable static existe mientras el programa este funcionando.

Para mover la declaración de console fuera del método main() la transformamos en variable static, que puede ser accedida por cualquiera de las dos rutinas, main(), y playGame(). Aquí esta el programa ejemplo completo:

NOTA: Será necesario descargar las clases “Console.java” y “ConsoleCanvas.java” y dejarlo en la misma carpeta donde va a escribir el código fuente.

```
package juegos;
import ConsolaIO.Console;
public class GuessingGame {
    static Console console;  // declara console como static variable
    public static void main(String[] args) {
       console = new Console();
       console.putln("Quiero jugar.  Pensare un número entre");
       console.putln("1 y 100, y pruebe a ver si lo adivina.");
       boolean playAgain;
       do {
          playGame();  // call subroutine to play one game
          console.put("Quiere jugar otra vez? ");
          playAgain = console.getlnBoolean();
       } while (playAgain);
       console.close();
    } // end of main()
    static void playGame() {
    int computersNumber = (int)(100 * Math.random()) + 1;
      // El valor asignado a computerNumber es aleatorio
      // y escogido entre 1 y 100, inclusive
        int usersGuess; // guardamos el numero pensado por el usuario
    console.putln();
    console.put("Cual es su primera jugada? ");
    do {
        usersGuess = console.getInt();
        if (usersGuess == computersNumber)
         console.putln("Lo consiguió!  Mi numero era " + computersNumber);
        else if (usersGuess  computersNumber)
```

```
          console.put("Es muy alto, vuelva a probar: ");
    } while (usersGuess != computersNumber);
    console.putln();
        } // end of playGame()
 } // end of class GuessingGame
```

Si el método es una caja negra, los parámetros proporcionan el medio para traspasar información desde fuera al mundo interior de la caja. Los parámetros forman parte de la interface del método y permiten personalizar el comportamiento del método para adaptarlo a situaciones particulares. Veamos otro ejemplo:

```
static void Print3NSequence(int startingValue) {
      // imprimir en consola la secuencia 3N+1 usando
      // startingValue como valor inicial de N
      int N = startingValue;  //para el elemento de la secuencia
      int count = 1;   // cuenta el numero de elementos
      console.putln("La secuencia 3N+1 empieza en " + N);
      console.putln();
      console.putln(N);  // imprimir el primer termino
      while (N > 1) {
            if (N % 2 == 1)     // N es par?
                  N = 3 * N + 1;
            else
                  N = N / 2;
            count++;   // contar el elemento
            console.putln(N);  // imprimir elemento
      }
      console.putln();
      console.putln("Hay " + count + " elementos en la secuencia.");
}  // fin de  Print3NSequence()
```

Este método puede llamarse utilizando una instrucción como

```
Print3NSequence(17);
```

Podemos usar el método en un programa que permita al usuario el imprimir la secuencia empezando en diferentes valores:

```
public static void main(String[] args) {
      console = new Console();
      console.putln("Este programa imprime la secuencia 3N+1");
      console.putln("empezando en el valor que le indique.");
      console.putln();
      do {
            console.putln("Indique valor inicial;")
            console.put("Para finalizar el programa entre 0: ");
            int K = console.getInt();  // lee valor inicial
```

```
            if (K > 0)   // imprime secuencia se  K es > 0
                  Print3NSequence(K);
      } while (K > 0);   // continua solo si K > 0
} // end main()
```

En la definición del método los parámetros se llaman parámetros formales o falsos parámetros. Los parámetros que se pasan al método cuando se llama se denominan parámetros reales. Los parámetros formales pueden ser identificadores, es decir, nombres, para los que debemos indicar de qué tipo son (int, boolean, String...). Un parámetro real es un valor, que puede estar representado por cualquier expresión que proporcione un valor del tipo adecuado. Cuando se llama a un método, hay que proporcionar un parámetro real por cada parámetro formal de la definición del método.

El sistema evalúa cada parámetro real e inicializa con ese valor el correspondiente parámetro formal. Por ejemplo:

```
         static void doTask(int N, double x, boolean test) {
             // aquí, las instrucciones
         }
```

Este método puede ser llamado así:

```
doTask(17,Math.sqrt(z+1),z>=10);
```

Ejecutar esta instrucción tiene básicamente el mismo efecto que el siguiente bloque:

```
{
      int N = 17; // declara un int llamado N con valor inicial 17
      double x = Math.sqrt(z+1);  // calcula Math.sqrt(z+1), y
      // lo emplea para inicializar x de tipo double
      boolean test = (z >= 10); // evalúa "z >= 10"
      // y usa el resultado true/false para inicializar
      // una nueva variable llamada test
      // aquí las instrucciones
}
```

Como hemos visto, para poder llamar correctamente a un método, es necesario conocer su nombre, cuántos parámetros tiene y de qué tipo es cada parámetro. Esta información se llama **firma (signature)** del método. La firma del método `doTask` se puede escribir como: `doTask(int,double,boolean)`. No se incluyen los nombres de lo parámetros. De hecho, no se necesita conocer los nombres, ya que estos no son parte de la interface. Los nombres solo se han empleado para programar el método.

Java es algo especial en el hecho de que permita que dos subrutinas dentro de una misma clase puedan tener el mismo nombre, mientras sus firmas sean diferentes. Esta característica se llama **sobrecarga** (overload). Decimos que el nombre del método puede ser sobrecargado (overloaded) porque tiene diferentes propósitos. El ordenador consigue diferenciar las subrutinas. Puede saber a cuál llamar en función del numero de parámetros y los tipos que se proporcionan en la instrucción de llamada.

En la clase Console hemos visto cómo se empleaban las sobrecargas. En esa clase, por ejemplo, se incluyen muchos métodos diferentes llamados putln. Esos métodos tienen todos firmas distintas, como:

```
putln(int)       putln(int,int)      putln(double)
putln(String)    putln(String,int)   putln(char)
putln(boolean)   putln(boolean,int)  putln()
```

La razón para emplear el mismo nombre para todas ellas es que estos métodos están relacionados semánticamente, pero para el sistema es muy diferente imprimir una String que un boolean, y así sucesivamente. Cada operación requiere un método diferente.

Dentro de la firma no esta incluido el tipo de retorno del método. Es ilegal el que haya en una misma clase dos métodos con la misma firma pero distinto tipo de retorno. Por ejemplo esto sería un error:

```
int    get() { ... }
double get() { ... }
```

Por eso en la clase Console, los métodos que leen diferentes tipos tienen nombres diferentes como getInt()y getDouble().

Los parámetros se usan cuando se llama al método, para “introducir” valores, pero una vez se inicia la ejecución del método, los parámetros se parecen mucho a las variables locales. Los cambios que se realizan dentro del método a los parámetros formales, no tienen efecto en el resto del programa. Es distinto cuando el método emplea variables definidas fuera, ya que estas existen de forma independiente del método y son accesibles desde otras partes del programa de la misma forma que desde el método.

Estas variables se conocen como globales, por oposición a las variables locales del método. El **ámbito** de una variable global es toda la clase en la que esta definida. Los cambios realizados en una variable global se extienden mas allá del método que los realiza. Se puede considerar la variable global como una parte de la interface del método.

El método puede emplear las variables globales para comunicarse con el resto del programa. Es una especie de falsa comunicación realizada por la puerta trasera, que es mucho menos visible que la hecha a través de los parámetros y corre el riesgo de violar la regla de la interface de caja negra. Se recomienda evitar las variables globales cuando los parámetros puedan ser mas apropiados.

7 Funciones

La subrutina que devuelve valores se llama **función**. Una función solo puede devolver valores de un tipo concreto, o **tipo de retorno** (return type) de la función. La llamada a una función aparece en los lugares en que el ordenador espera encontrar un valor, como en la parte derecha de una instrucción de asignación, como parámetro en una llamada a subrutina o en medio de alguna expresión. Las funciones de valor boleano también pueden ser usadas como condiciones en las instrucciones if, while o do.

También es correcto emplear una llamada a función en una instrucción como si fuera una subrutina normal. En esta caso, el ordenador ignorará el valor calculado por la subrutina. Por ejemplo, la función `console.getln()`, que devuelve un tipo String, lee y devuelve la línea tecleada por el usuario. Normalmente la línea devuelta, se asigna a una variable en una instrucción:

```
String name=console.getln();
```

También es útil emplearla como subrutina con la llamada

```
console.getln();
```

para que lea y descargue todo lo que se haya tecleado hasta el siguiente retorno de carro.

La función tiene la misma forma que una subrutina, excepto que tiene que especificar el valor que se va a devolver. Esto se puede realizar con la instrucción return, que tiene esta forma:

```
return expresión;
```

Esta instrucción solo pueden aparecer en el interior de la definición de la función, y el tipo de la expresión debe ser el mismo que el que se ha especificado como tipo de retorno para esta función. Cuando el ordenador ejecuta la instrucción return, evalúa la expresión. finaliza la ejecución de la función y usa el valor de la expresión como valor de retorno de la función.

Dentro de un método ordinario, que declara como tipo de retorno “void” se puede utilizar la instrucción return para finalizar la ejecución del método inmediatamente y devolver el control al punto del programa que lo invocó. Esta forma de finalizar la ejecución de la subrutina solo es obligatoria en funciones.

Veamos un ejemplo de función:

```
        static int nextN(int currentN) {
           if (currentN % 2 == 1)     // prueba si N es impar
              return 3*currentN + 1;  // si lo es, devuelve esto
           else
              return currentN / 2;    // si no, devuelve esto
        }
```

Muchos programadores prefieren utilizar una instrucción return única como final real de la función. Esto permite encontrar la instrucción return mas fácilmente. Podemos escoger el escribir la función de esa forma, en el siguiente ejemplo:

```
        static int nextN(int currentN) {
           int answer;  // answer sera el valor de retorno
           if (currentN % 2 == 1)    // prueba si N es impar
              answer = 3*currentN+1; // si lo es, prepara esto
           else
              answer = currentN / 2; // si no, prepara esto
           return answer;   // (Devuelve el valor !No lo olvide!)
```

```
        }
```

En este caso, la mejora frente a la versión original no es muy grande, pero si nextN() fuera una función larga que realizara cálculos complejos, tendría sentido ocultar la complejidad dentro de la función:

```
   static void Print3NSequence(int startingValue) {

           // imprimir la secuencia 3N+1 en la consola usando
           // startingValue como valor inicial de N
           int N = startingValue;  // N es el termino de la secuencia
           int count = 1;   // cuenta el numero de términos
           console.putln("La secuencia 3N+1 empieza en " + N);
           console.putln();
           console.putln(N);  // imprimir el primer termino
           while (N > 1) {
               N = nextN(N);  // calcula siguiente termino
               count++;   // lo cuenta
               console.putln(N);  // lo imprime
           }
           console.putln();
           console.putln("Aquí hay " + count + " términos en la secuencia.");
        }  // end of Print3NSequence()
```

Aquí tenemos algunos ejemplos mas sobre funciones. El primero calcula la letra que se ha de colocar según una escala de valores numéricos:

```
 static char letterGrade(int numGrade) {
        // devuelve la letra correspondiente
        // a una escala numérica
        if (numGrade >= 90)
           return 'A';   // 90 o superior devuelve A
        else if (numGrade >= 80)
           return 'B';   // 80 a 89 devuelve B
        else if (numGrade >= 65)
           return 'C';   // 65 a 79 devuelve C
        else if (numGrade >= 50)
           return 'D';   // 50 a 64 devuelve D
        else
           return 'F';   // cualquier otra cosa devuelve F
     }  // end of function letterGrade()
```

El tipo del valor de retorno de letterGrade() es char. Las funciones pueden devolver cualquier tipo de valor. Aquí tenemos una que devuelve un valor de tipo boolean:

```
      static boolean isPrime(int N) {
         // devuelve true si N es numero primo, esto es,
         // si N es positivo y no es divisible por ningún
         // numero positivo excepto por 1 y él mismo
```

```
          int maxToTry = (int)Math.sqrt(N);
               // probaremos de dividir N por números entre
               // 2 y maxToTry; si N no es divisible por
               // ninguno de esos números, entonces N es primo.
               // (observe que  Math.sqrt(N) devuelve un valor
               // de tipo double, el valor debe cambiarse a
               // tipo int antes de poder asignar
               // a maxToTry.)
           for (int divisor = 2; divisor <= maxToTry; divisor++) {
               if ( N % divisor == 0 )   // prueba si N es divisible
                  return false;  // si lo es, no es primo
           }
           // al llegar a este punto, N debe ser primo. De otra forma
           // la función ya habría terminado al ejecutar
           // la instrucción return en el bucle for.
           return true;  // si, N es primo
        }  // end of function isPrime()
```

También tendremos la posibilidad de mejorar el programa definiendo la impresión de los términos de la secuencia en columna, con varios términos en cada línea. Así haremos la salida mas presentable. Solo habrá que controlar cuantos términos se ha escrito en la línea actual, y cuando el valor alcance un determinado numero iniciaremos nueva línea. Para conseguir que los términos aparezcan alineados en columna, usaremos la versión de console.put() con firma put(int,int). El segundo parámetro int indica la anchura de la columna. Para enseñar otra nueva idea, el programa especifica el numero de columnas de salida y la anchura de cada columna como constante.

Una constante es como una variable excepto que su valor no puede cambiar nunca mientras el programa este funcionando. (para cambiar el valor, deberá editar el programa y recompilarlo). En Java, una constante se define utilizando en la declaración de la variable el modificador “final“.

La palabra final, indica que la constante esta en su forma final– que el valor asignado es el final y no puede ser cambiado.

```
public class ThreeN {
       /*
      Este programa calcula y presenta la secuencia 3N+1
      Empieza la secuencia en donde el usuario indique
      Los términos de la secuencia se imprimen en columnas
      con varios términos en cada línea. Después de finalizar
      la secuencia, se informa del numero de términos
       */
       final static int numberOfColumns = 5;  // constant indica el
                                    // numero de términos en columna
       final static int columnWidth = 8;  // constant indica anchura
                                          // en numero de caracteres en
                                          // cada columna
       Console console;  // ventana para entrada/salida
       public static void main(String[] args) {
```

```
            console = new Console();
            console.putln("Este programa imprime la secuencia 3N+1 ");
            console.putln("empezando en el valor que especifique.");
            console.putln();
            int K = 0;
            do {
               console.putln("Indique valor inicial;");
               console.put("para finalizar programa entre 0: ");
               K = console.getInt();  // lee valor inicial del usuario
               if (K > 0)   // imprime la secuencia si  K is > 0
                  Print3NSequence(K);
            } while (K > 0);   // continua solo si K > 0
            console.close();
         } // end main()
         static void Print3NSequence(int startingValue) {
            // imprime la secuencia 3N+1 en la consola usando
            // startingValue como valor inicial de N
            int N = startingValue;  // N representa el termino.
            int count = 1;   // cuenta el numero de términos.
            int onLine = 1;  // onLine cuenta términos en la línea actual.
            console.putln("La secuencia 3N+1 empieza en " + N);
            console.putln();
            console.put(N, columnWidth);  // imprime el primer termino
            while (N > 1) {
                N = nextN(N);  // calcula siguiente
                count++;   // cuenta el termino
                if (onLine == numberOfColumns) { // si la línea esta llena
                   console.putln();  //imprime retorno de carro
                   onLine = 0;   // y borra contador de términos en línea
                }
                console.put(N, columnWidth);  // imprime el termino
                onLine++;   // incrementa contador de términos en línea
            }
            console.putln();  // final de la línea actual
            console.putln();  // añade línea en blanco
            console.putln("Aquí hay " + count + " términos en la secuencia.");
         }  // end of Print3NSequence()
         static int nextN(int currentN) {
             // calcula el siguiente termino de la secuencia 3N+1
             if (currentN % 2 == 1)
                return 3 * currentN + 1;
             else
                return currentN / 2;
         }  // end of nextN()
      } // end of class ThreeN
```