Referencia JAVA

Víctor Chico Rodríguez

January 21, 2020

Contents

1 Introducción

Java es un lenguage de programación orientado a objetos, fue desarrollado en 1995 por Sun MicroSystems y está basado en C++, es un lenguaje que se ejecuta en una máquina virtual que interpreta (la JVM, siglas de *Java Virtual Machine*) las instrucciones compiladas a bytecode (el lenguaje de la máquina virtual). Java fue adquirido por Oracle en el año 2010.

1.1 Estructura de un programa

```
//Este fichero pertecene al paquete (carpeta) curso.java.manual (curso/java/manual)
package curso.java.manual;

//Esto es una importación de la clase Scanner del paquete java.io
import java.io.Scanner;

//Esto es una clase pública que se llama HolaMundo
public class HolaMundo {

    /*Este es el método main (principal),
es un método especial que servirá comom punto de entrada a la aplicación.
Este método es:
- público (puede ser accedido desde cualquier clase)
- estático (puede ser accedido sin necesidad de crear un objeto de la clase
-  no devuelve nada (tipo void)
- recibe como argumentos un array (matriz) de objetos de tipo String (cadena de texto)
*/
    public static void main (String [] args) {
//Este es un objeto de la clase Scanner que se llama scan
//Este objeto se inicializa con la palabra reservada new y
//recibe como argumento System.in (Entrada del sistema)
Scanner scan = new Scanner(System.in);

/*
  Esto es una llamada a un método, concretamente al método print(String)
  del atributo out de la clase System, este método imprime en la pantalla
  (consola de texto) el texto que se le pase como parámetro y continúa
  en la misma línea.
  Como parámetro se le pasa la cadena de texto (String) "¿Cómo te llamas?"
  Los valores de tipo String van siempre entre comillas dobles "
```

```
 */
System.out.print("¿Cómo te llamas?");

//Este es un objeto de la clase String (cadena de texto) que se llama nombre.
//Este objeto se inicializa automáticamente con el valor que devuleve
//el método readLine() (Método sin argumentos) del objeto scan.
String nombre = scan.nextString();

/*
  Esto es otra llamada a un método, en este caso al println del atributo
  out de la clase System, nótese la diferencia con la llamada anterior
  (print -- println), ese ln añadido lo que hace es saltar de línea una
  vez haya impreso lo que le pasemos como parámetro.

  En este caso, como parámetro se le pasa una cadena de texto (igual que antes)
  con el valor "Hola, " a lo que le concatenamos (sumamos) el valor de
  la variable nombre
 */
System.out.println("Hola, "+nombre);
    }
}
```

Figure 1: Hola Mundo

```
¿Cómo te llamas? Víctor
Hola, Víctor
```

2 Tipos de datos

Los tipos de datos primitivos en java son los siguientes:

tipo	descripción	clase asociada
byte	número entero de 8 bits (-128 a 127)	Byte
short	número entero de 16 bits (-32768 a 32767	Short
int	número entero de 32 bits (-2^{32} a 2^{32})	Integer
long	número entero de 64 bits (-2^{64} a 2^{64})	Long
float	número decimal de 32 bits	Float
double	número decimal de 64 bits	Double
boolean	valor booleano o lógico (verdadero o falso)	Boolean
char	caracter de texto (único)	Character

Los tipos de datos normalmente se usan en su forma primitiva (columna tipo) y se pueden asignar directamente, pero a veces es util usar métodos de su clase asociada.

3 Operadores

3.1 Asignación

El operador = se usa para asignar valores a variables:

```
int a = 0;
```

3.2 Aritméticos

En java se pueden realizar multitud de operaciones matemáticas con la misma precedencia que en la vida real, si se necesita modificar se pueden utilizar paréntesis, los operadores aritméticos son los siguientes:

Operador	Descripción
+	Operador de suma
-	Operador de resta
*	Operador de multiplicación
/	Operador de división
%	Operador de resto de la división

El siguiente código es una pequeña demostración de los operadores mencionados:

```
public class Aritmeticos {

     public static void main (String[] args) {

 // Variable de tipo int que tendrá como valor el resultado de 1 + 2
 int resultado = 1 + 2;
 // El valor de resultado es 3
 System.out.println("1 + 2 = " + resultado);
 int resultado_original = resultado;

 // Los operadores se pueden usar entre variables (numéricas) y números
 // en este caso se resta 1 al valor de resultado primero y se asigna a
 // la variable resultado después
```

```
resultado = resultado - 1;
// El valor de resultado es 2
System.out.println(resultado_original + " - 1 = " + resultado);
resultado_original = resultado;

// Multiplicamos el resultado por 2 y lo volvemos a asignar a la variable
//resultado
resultado = resultado * 2;
// El valor de resultado es 4
System.out.println(resultado_original + " * 2 = " + resultado);
resultado_original = resultado;

// Dividimos el resultado entre 2 y lo asignamos
resultado = resultado / 2;
// El valor de resultado es 2
System.out.println(resultado_original + " / 2 = " + resultado);
resultado_original = resultado;

resultado = resultado + 8;
// El valor de resultado es 10
System.out.println(resultado_original + " + 8 = " + resultado);
resultado_original = resultado;

// Dividimos el resultado entre 7 y nos quedamos con el resto, luego lo
// asignamos
resultado = resultado % 7;
// El valor de resultado es 3
System.out.println(resultado_original + " % 7 = " + resultado);
    }
}
```

Figure 2: Aritmeticos

```
1 + 2 = 3
3 - 1 = 2
2 * 2 = 4
4 / 2 = 2
2 + 8 = 10
10 % 7 = 3
```

Como vimos anteriormente, el operador suma + se puede utilizar también para concatenar texto:

```
class Concatenacion {
    public static void main(String[] args){
String firstString = "Esto es";
String secondString = " una cadena de texto concatenada.";
String thirdString = firstString+secondString;
System.out.println(thirdString);
    }
}
```

Figure 3: Concatenación

```
Esto es una cadena de texto concatenada.
```

3.2.1 Operadores unarios

En java hay un tipo de operadores aritméticos que sólo se utilizan en un operando, son los operadores unarios:

Operador	Descripción
+	Indica un valor positivo
-	Indica un valor negativo
++	Incrementa en 1 el valor
–	Decrementa en 1 el valor
!	Invierte el valor de un booleano

```
class Unarios {
    public static void main(String[] args) {
int resultado = +1;
// El resultado es 1
System.out.println(resultado);

resultado--;
// El resultado es 0
System.out.println(resultado);

resultado++;
// El resultado es 1
System.out.println(resultado);
```

```
resultado = -resultado;
// El resultado es -1
System.out.println(resultado);

boolean exito = false;
// false
System.out.println(exito);
// true
System.out.println(!exito);
    }
}
```

Figure 4: Unarios

```
1
0
1
-1
false
true
```

Los operadores de incremento y decremento ($++$ y $--$) actuan de manera diferente dependiendo de si se ponen delante o detrás del valor a modificar, si se usan de manera prefija `++variable` el valor se incrementa primero y la variable se usa después (ya incrementada), si se usa de manera postfija `variable++` se utilizará el valor de la variable sin incrementar y luego se incrementará:

```
class PrePost {
    public static void main(String[] args){
int i = 3;
i++;
// imprime 4
System.out.println(i);
++i;
// imprime 5
System.out.println(i);
// imprime 6
System.out.println(++i);
```

```
// imprime 6
System.out.println(i++);
// imprime 7
System.out.println(i);
    }
}
```

Figure 5: Prefijos y Postfijos

```
4
5
6
6
7
```

3.3 Lógicos

Son operadores que devuelven valores lógicos (verdadero o falso)

Operador	Descripción
==	igual que
!=	distinto que
>	mayor que
>=	mayor o igual que
<	menor que
<=	menor o igual que
&&	Y lógico
||	Ó lógico
instanceof	Objeto pertenece a clase

Normalmente estos operadores se utilizarán en sentencias que requieran un valor lógico, como los condicionales o los bucles, de los que hablaremos más adelante, en este ejemplo vemos como, en base a los valores 1 y 2, que operaciones se ejecutan y cuales no:

```
class Comparacion {

    public static void main(String[] args){
int valor1 = 1;
int valor2 = 2;
System.out.println ("valor1="+valor1+", valor2="+valor2);
```

```
if(valor1 == valor2) {
    System.out.println("valor1 == valor2 --> " + (valor1 == valor2));
}
if (valor1 != valor2) {
    System.out.println("valor1 != valor2 --> " + (valor1 != valor2));
}
if (valor1 > valor2) {
    System.out.println("valor1 > valor2 --> " + (valor1 > valor2));
}
if (valor1 < valor2) {
    System.out.println("valor1 < valor2 --> " + (valor1 < valor2));
}
if (valor1 <= valor2) {
    System.out.println("valor1 <= valor2 --> " + (valor1 <= valor2));
}
    }
}
```

Figure 6: Comparación

```
valor1=1, valor2=2
valor1 != valor2 --> true
valor1 < valor2 --> true
valor1 <= valor2 --> true
```

A veces es interesante comprobar si una comprobación cumple mas de una condición o si una sentencia se ejecutará si se cumple alguna de las condiciones posibles, es en este caso que utilizaremos los operadores lógicos `&&` y `||`.

```
class Condicionales {

    public static void main(String[] args){
int valor1 = 1;
int valor2 = 2;
if((valor1 == 1) && (valor2 == 2))
    System.out.println("valor1 es 1 AND (Y) valor2 es 2");
if((valor1 == 1) || (valor2 == 1))
    System.out.println("valor1 es 1 OR (O) valor2 es 1");
    }
}
```

Figure 7: Operadores Condicionales

```
valor1 es 1 AND (Y) valor2 es 2
valor1 es 1 OR (O) valor2 es 1
```

4 Condicionales

En java tenemos principalmente dos estructuras condicionales, la primera es la que se compone con las sentencias `if` y `else`, y la segunda es la sentencia `switch`.

4.1 if y else

La sentencia `if` se escribe de la siguiente manera:

```
if (condicion) {
    proceso;
}
```

Donde `condicion` es un valor booleano (lógico), que puede ser una variable de tipo boolean, un valor `true` o `false` directamente, aunque no tuviera mucho sentido en este caso, o el resultado de una comparación como las que acabamos de ver.

Si la condición se cumple el `proceso` (que puede ser un número indeterminado de sentencias) se ejecuta, si no se cumple, no se ejecuta, decimos que se produce un salto condicional.

Hay veces que queremos que si se cumple una condición se ejecute un determinado código y, si no se cumple, otro, esto lo conseguimos con la sentencia `else` que tiene una forma parecida al `if`, pero en este caso no se especifica condición, sino que la condición es que no se cumpla el `if`.

```
if (condicion) {
    proceso;
} else {
    otroProceso;
}
```

Puede suceder que queramos comprobar una cosa y luego, independientemente otra, en ese caso solo tendríamos que tener un `if` primero y, una vez cerrado, otro con otra condición, en ese caso serían sentencias independientes y no habría ningún problema, pero podemos querer comprobar algo y, si se cumple, otra cosa después, esto lo hacemos *anidando* sentencias `if` o `else`:

```
if (condicion1) {
    proceso1;
    if (condicion2) {
proceso2;
    }
    proceso3;
} else {
    if (condicion3) {
proceso4;
    }
}
```

Si nos fijamos en el `else` (aunque esto puede ocurrir en cualquier otra parte, incluido el bloque del `if`), podemos observar que, en caso de no cumplirse la `condicion1`, podemos tener dentro otra estructura completa de sentencias `if` y cada una puede tener sus respectivos `else` y así indefinidamente, una manera de organizar mejor esté código es utilizando la sentencia compuesta `else if` que nos permite hacer varias comprobaciones sin aumentar el nivel de anidación, por ejemplo:

```
class Elseif {
  public static void main (String [] args) {
    int val = 10;
    if (val == 0) {
      System.out.println("val = 0");
    } else if (val == 1) {
      System.out.println("val = 1");
    } else if (val == 2) {
      System.out.println("val = 2");
    } else if (val == 3) {
      System.out.println("val = 3");
    } else if (val == 4) {
      System.out.println("val = 4");
    } else if (val == 5) {
      System.out.println("val = 5");
    } else {
      System.out.println("val > 5");
    }
  }
}
```

Figure 8: Else-If

```
val > 5
```

En este caso como el valor de la variable `val` es 10, pasaría por cada una de las condicione y, al no cumplirse, entraría por la sentencia `else` si hiciéramos esto anidando sentencias `if` y `else` el código se *iría* muy a la derecha y sería más dificil de leer, pero aún tenemos otra sentencia que nos permite resolver estos problemas de una manera más elegante, la sentencia `switch`.

4.2 switch

El ejemplo anterior, escrito con una sentencia switch sería el siguiente:

```
class Switch {
    public static void main (String [] args) {
      int val = 10;
      switch(val) {
      case 0:
System.out.println("val = 0");
break;
      case 1:
System.out.println("val = 1");
break;
      case 2:
System.out.println("val = 2");
break;
      case 3:
System.out.println("val = 3");
break;
      case 4:
System.out.println("val = 4");
break;
      case 5:
System.out.println("val = 5");
break;
      default:
System.out.println("val > 5");
      }
    }
```

```
  }
```

Figure 9: Switch

```
val > 5
```

Como se puede observar, el código es mucho más claro, tenemos una sola sentencia condicional, `switch`, y esta, en base al valor que tenga la variable, entrará por un `case` o por otro y, en caso de que no coincida con ninguno, entrará por el `default`. Si, por ejemplo, cambiásemos el valor de `val` a 3, la salida que nos mostraría el programa sería la siguiente:

```
val = 3
```

Podemos observar también una sentencia que no habíamos visto antes, la sentencia `break`, esta sentencia *rompe* la ejecución del bloque en el que se encuentra, sería como ir a la llave de cierre, normalmente está desaconsejado su uso, pero en la sentencia `switch` es necesaria para cortar la ejecución donde nos interese, ya que, a diferencia de con las estructuras `if-else`, que están englobadas con llaves que nos hacen de corte, los `case` y `default` son etiquetas, y no delimitan código, lo marcan. Veamos que pasa si no ponemos la sentencia `break` en un `switch`.

```
class Switch2 {
    public static void main (String [] args) {
      int val = 2;
      //Inicializamos un contador para saber por cuantos cases pasamos;
      int contador = 0;
      switch(val) {
      case 0:
contador++;
      case 1:
contador++;
      case 2:
contador++;
      case 3:
contador++;
      case 4:
contador++;
      case 5:
contador++;
```

```
      case 6:
contador++;
      case 7:
contador++;
      case 8:
contador++;
      case 9:
contador++;
      case 10:
contador++;
System.out.println("He pasado por "+contador+" cases. El número es menor o igual que 1
      }
    }
  }
```

Figure 10: Switch2

```
He pasado por 9 cases. El número es menor o igual que 10
```

¿Qué ha pasado? El programa ha ejecutado todos los cases uno detrás de otro, ya que ninguno tenía una sentencia `break` para parar la ejecución y ha llegado hasta el último, donde ha imprimido el mensaje. Este ejemplo nos sirve también para ver que la etiqueta `default` no es imprescindible, como en la instrucción `if` no es imprescindible el `else`, simplemente, si no se cumple ninguna de las condiciones contempladas, no se hará nada.

4.3 Condicional ternario

Por último nos queda un último tipo de condicional, llamado ternario o de asignación, esta estructura nos permite asignar un valor a una variable en base al valor de otra y se escribe de la siguiente forma:

```
String miString = (condicion)?"condicion es verdadera":"condicion es falsa";
```

Analizando por partes tenemos, a la izquierda del igual, una declaración de variable de tipo `String` como las que hemos visto hasta ahora, a la derecha tenemos, primero una condición lógica (del mismo tipo que las que se usan en las sentencias `if`, luego un signo de interrogación `?` que es el que nos indica que ese valor lógico no es para asignar a la variable, como hemos visto cuando asignábamos variables de tipo `boolean`, sino que es la condición

para asignar la variable, el siguiente valor `"condicion es verdadera"` es el valor que tomará la variable `miString` si `(condicion)` es verdadera. Luego encontramos un signo de dos puntos `:` que separa las condiciones verdadera y falsa y, por último `"condicion el falsa"` que, como se puede intuir, es el valor que tomará `miString` si `(condicion)` es falsa.

Este condicional puede ser escrito con sentencias `if-else` de la siguiente manera (el resultado del código será el mismo):

```
String miString;
if (condicion) {
  miString = "condicion es verdadera";
} else {
  miString = "condicion es falsa";
}
```

La decisión de usar una u otra dependerá de si se prefiere legibilidad del código (ternaria) o comprensión más visual (if-else).

5 Bucles

La ejecució normal de un programa en java (y en casi cualquier lenguaje de programación) se hace *de arriba a abajo* desde que empieza hasta que termina, los bucles son estructuras de control que permiten que una parte del código se ejecute más de una vez en base a una condición.

5.1 El bucle while

El tipo de bucle más simple que nos encontramos es el bucle `while`, este bucle se va a ejecutar *mientras* (while) la condición se cumpla y, una vez esta deje de cumplirse, seguirá desde el final del mismo.

Es importante que la condición deje de cumplirse en algún momento, y esto es válido para cualquier tipo de bucle, si la condición siempre se cumple decimos que tenemos un bucle infinito, el cual hará que nuestro programa se bloquee.

La estructura de un bucle `while` es la siguiente:

Por ejemplo, si queremos un programa que muestre por pantalla los números del 1 al 10, podemos hacer lo siguiente:

```
class BucleWhile {
```

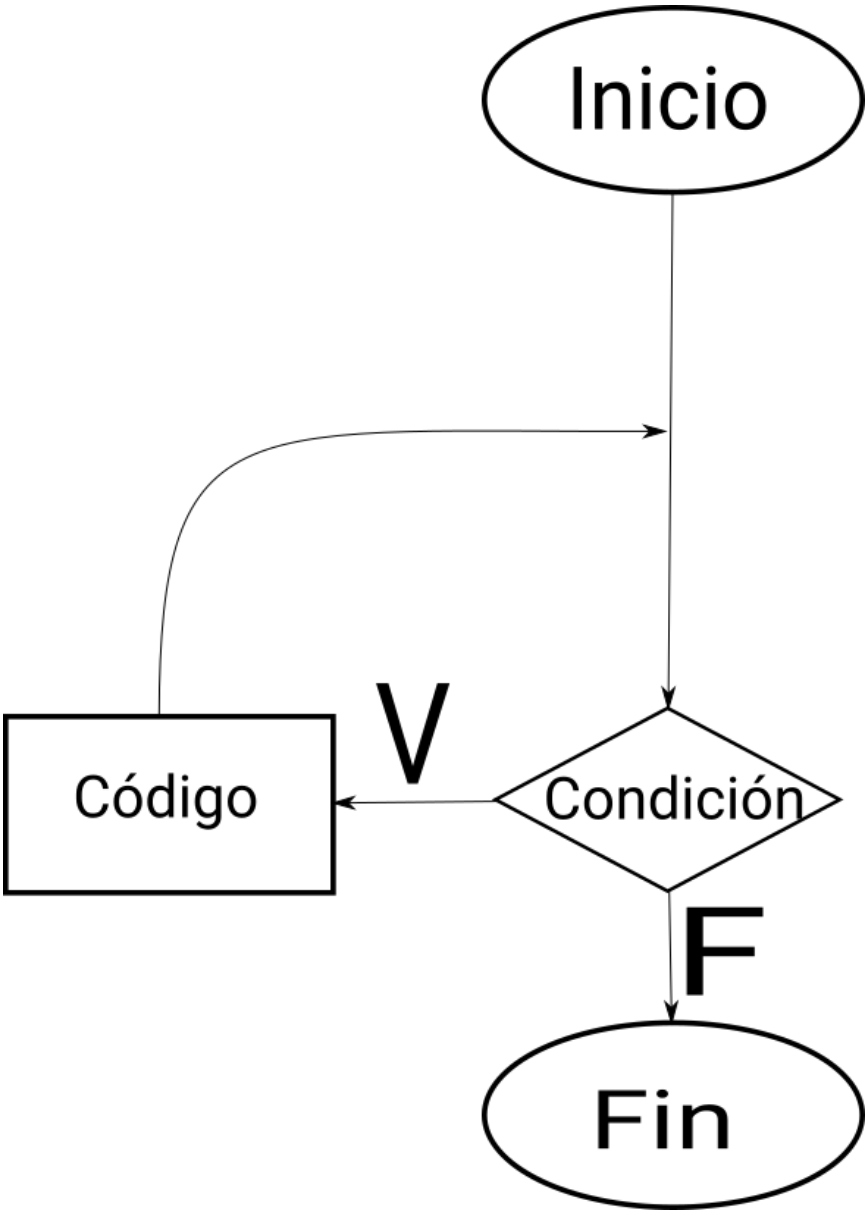

Figure 11: Diagrama de un bucle while

```
    public static void main(String[] args) {
//Ponemos el número con el valor que queremos al principio
int numeroActual=1;

//Bucle while
//Condición: que numeroActual sea menor o igual que 10
while (numeroActual<=10) {

    //Imprimimos por pantalla el número con su valor en este momento
    System.out.println(numeroActual);

    //Aumentamos el valor del número
    //Si no lo hacemos, el valor de númeroActual siempre será menor o igual a 10 y teno
    numeroActual++;
}
    }

}
```

Figure 12: Bucle While

5.2 El bucle for

El bucle `for` es un caso especial del bucle `while`, este bucle se va a ejecutar igualmente mientras se cumpla la condición dada, por lo que su diagrama es el mismo, pero nos permite simplificar la programación metiendo en la cabecera tanto la inicialización de la variable como su modificación, por ejemplo, si como en el caso anterior queremos escribir los números del 1 al 10 con un bucle `for` lo haríamos así:

```
class BucleFor {

    public static void main (String[] args) {
for (int numeroActual=1; numeroActual<=10; numeroActual++) {
    System.out.println(numeroActual);
}
    }

}
```

Figure 13: Bucle For

Como podemos ver, el resultado de este programa será exáctamente el mismo que el anterior:

```
1
2
3
4
5
6
7
8
9
10
```

La decisión de utiliar un tipo de bucle u otro depende del programador, pero se suele utiliar el bucle `for` para situaciones en las que haya que *contar*, como en el caso que hemos puesto porque nos permite crear y deshechar la variable en la propia cabecera sin tener que llevar datos innecesarios, aunque por supuesto podemos usar una variable que tengamos de antes como en el bucle `while` e, incluso, no modificar la variable en la cabecera y hacerlo en el cuerpo.

```
class BucleForSinInicializacion {

    public static void main (String[] args) {
    int numeroActual=1;
    for (; numeroActual<=10; numeroActual++) {
    System.out.println(numeroActual);
}
    }

}
```

Figure 14: Bucle For sin inicialización en la cabecera

```
class BucleForSinModificacion {

    public static void main (String[] args) {
for (int numeroActual=1; numeroActual<=10;) {
```

```
    System.out.println(numeroActual);
    numeroActual++;
}
    }

}
```

Figure 15: Bucle For Sin Modificacion en la cabecera

Y, por supuesto, si sacamos de la cabecera tanto la inicialización como la modificación de la variable, lo que tenemos es un bucle `while` con otro nombre:

```
class BucleForSinInicializacionNiModificacion {

    public static void main (String[] args) {
int numeroActual=1;
for (; numeroActual<=10;) {
    System.out.println(numeroActual);
    numeroActual++;
}
    }

}
```

Figure 16: Bucle For Sin inicialización ni Modificacion en la cabecera

5.3 El bucle do-while

Hasta ahora hemos visto bucles que se ejecutan sólo si se cumple una determinada condición, pero ¿y si queremos que un fragmento de código se ejecute como mínimo una vez pero si se cumple la condición se ejecute unas cuantas mas? Podríamos duplicar el mismo código, una vez fuera del bucle y otra vez dentro, pero para ahorrarnos la reduncancia tenemos el bucle `do-while`.

Este bucle se trata de un bucle `while` en el que la condición para volverlo a ejecutar se encuentra al final y no al principio, fijémonos en el siguiente diagrama:

Podemos continuar con nuestro ejemplo de contar de 1 a 10, veamos como se haría con un bucle do-while:

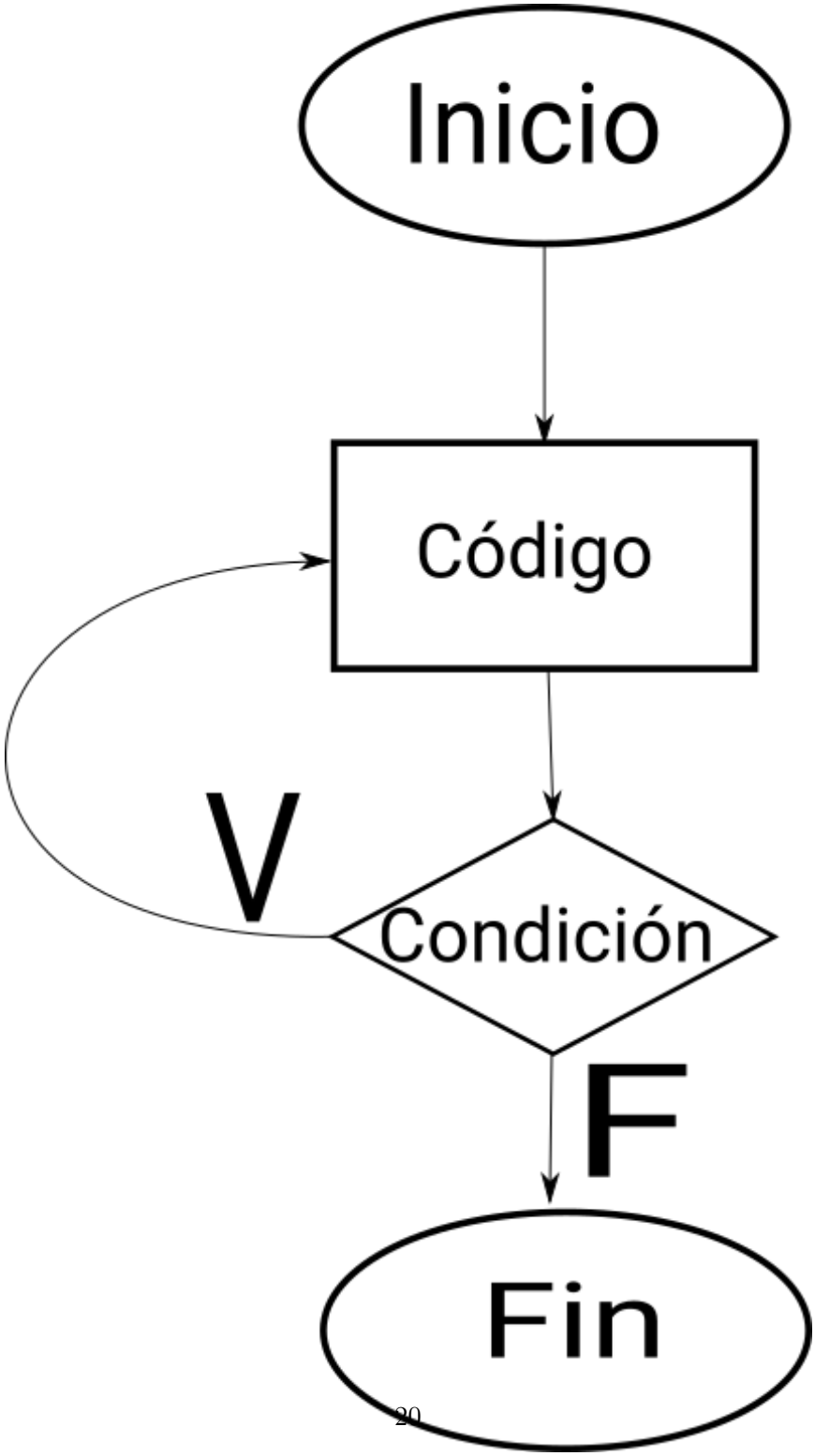

Figure 17: Diagrama de un bucle do-while

```
class DoWhile {

  public static void main(String[] args) {
    //Inicializamos la variable fuera del bucle
    int numeroActual=1;
    do {
      //Imprimimos su valor
      System.out.println(numeroActual);
      //Aumentamos la variable
      numeroActual++;
      //Comprobamos la condición, como ya se ejecuta una vez como mínimo
      //es necesario poner menor (<) y no menor o igual (<=), si lo hiciéramos
      //cuando el valor de numeroActual fuera 10 cumpliría y volvería a ejecutar
      //el código, por lo tanto contaría hasta 11
    } while (numeroActual < 10);
  }

}
```

Figure 18: Bucle do-while

```
1
2
3
4
5
6
7
8
9
10
```

6 Métodos y funciones

Con lo que ya sabemos podemos crear programas muy potentes, podemos controlar si un bloque de código se ejecutará o no y cuantas veces lo hará, pero la ejecución sigue siendo *de arriba a abajo*, estamos en lo que se conoce como **programación estructurada**, pero vayamos más allá, hasta ahora, si queríamos ejecutar un bloque de código más de una vez podíamos hacer bucles, pero estos siempre se ejecutarán con los mismos datos, con los métodos

podemos ejecutar el mismo código pero con **parámetros** diferentes, veamos un ejemplo sencillo:

```
class MiPrimerMetodo {

  //Aquí tenemos el método main, no devuelve nada (void)
  //y recibe como parámetro un array de Strings (varias cadenas de texto)
  public static void main(String [] args) {
    //Creamos una variable llamada nombre y la inicializamos
    String nombre = "Bimo";
    //Llamamos a nuestro método y le pasamos como parámetro la variable
    saludar(nombre);
    //Cambiamos el valor de la variable
    nombre = "Kirby";
    //Y volvemos a llamar al método
    saludar(kirby);
  }

  //Aquí tenemos nuestro método, se llama saludar y no devuelve nada,
  //recibe como parámetro una cadena de texto llamada nombre,
  //cada vez que se ejecute saludará a quien venga escrito en la variable nombre
  static void saludar(String nombre) {
    System.out.println("Hola, "+nombre);
  }

}
```

Figure 19: Mi primer método

```
Hola, Bimo
Hola, Kirby
```

En Java, se dice que los parámetros que se pasan a los métodos se hacen *por valor*, esto quiere decir que cada método hace una copia del parámetro cuando se invoca y, se asignamos un nuevo valor al parámetro dentro del método, este no cambiará en el método que lo invocó, veamos un ejemplo:

```
class PasoDeValor {

  //Declaramos un método al que llamaremos,
```

```
  //Como podemos ver, podemos declararlo antes
  //del método main, aunque se le llamará después
  static void cambiarValor(int numero) {
    numero = 2;
  }

  public static void main (String [] args) {
    //Declaramos una variable y le asignamos un valor
    int numero = 1;

    //Llamamos al métod de cambio de valor
    cambiarValor(numero);

    //Mostramos el resultado por pantalla
    System.out.println(numero);
  }

}
```

Figure 20: Paso de valor

```
1
```

Se puede pensar que el que los valores no pasen de los métodos a quien los llamó es poco util, pero si puede hacerse, para ello usaremos la sentencia **return**, que literalmente *devuelve* el valor que le digamos, si modificamos un poco el programa anterior lo veremos:

```
class RetornoDeValor {

  //Declaramos un método al que llamaremos,
  //Como podemos ver, podemos declararlo antes
  //del método main, aunque se le llamará después.
  //En este caso hemos cambiado el tipo de retorno del método
  // de void (no devuleve nada) a int, y hemos añadido la sentencia
  // return con nuestra variable.
  static int cambiarValor(int numero) {
    numero = 2;
    return numero;
  }
```

```
  public static void main (String [] args) {
    //Declaramos una variable y le asignamos un valor
    int numero = 1;

    //Llamamos al métod de cambio de valor
    //y le asignamos el valor del retorno
    numero = cambiarValor(numero);

    //Mostramos el resultado por pantalla
    System.out.println(numero);
  }

}
```

Figure 21: Retorno de valor

```
2
```

Ahora, para ver otra característica del paso de parámetros a los métodos, tenemos que hacerlo mediante objetos, de momento no nos preocupemos mucho de ello, básicamente lo que tenemos que tener en cuenta es que si pasamos un objeto como parámetro y cambiamos el valor de una de sus propiedades dentro de un método, esta permanecerá cambiada incluso fuera del método, esto puede parecer lo opuesto a lo que acabamos de ver, pero no es así, si en lugar de cambiar el valor de una de las propiedades del objeto lo que hiciéramos fuera instanciar un nuevo objeto en la variable (como asignar un nuevo valor en las variables que ya conocemos), el valor de la variable original permanecería intacto, veamoslo de nuevo con dos ejemplos:

```
import java.util.ArrayList;

class CambioDeObjeto {
  public static void main (String [] args) {
    /*
      Creamos un objeto de tipo ArrayList, este objeto consiste un una lista del tipo d
      que le digamos entre los acentos, en este caso Integer (número entero), se declar
      sabemos TipoDeObjeto nombreDeLaVariable, y luego se le asigna un valor, en este c
      nuevo de la clase ArrayList, los paréntesis que aparecen al final son porque esta
      al método constructor del objeto, sin parámetros en este caso, aunque puede haber
```

```
     */
    ArrayList<Integer> lista = new ArrayList<Integer>();

    /*
      Añadimos un objeto de la clase Integer (un 1), en el objeto lista, para ello llam
      add de la clase ArrayList, con un parámetro de tipo entero (el 1), este método aí
     */
    lista.add(1);

    //Imprimimos el valor del primer elemento de la lista (posición 0), llamando al mét
    System.out.println(lista.get(0));
  }

  /*
    Se asigna un nuevo valor a la variable lista
   */
  static void nuevoObjeto(ArrayList<Integer> lista) {
    lista = new ArrayList<Integer>();
    lista.add(2);
  }
}
```

Figure 22: Cambio de objeto

```
1
```

Ahora vamos a ver como podemos añadir un valor al objeto que ya existe

```
import java.util.ArrayList;

class CambioDePropiedad {
  public static void main (String [] args) {
    /*
      Creamos un objeto de tipo ArrayList, este objeto consiste un una lista del tipo d
      que le digamos entre los acentos, en este caso Integer (número entero), se declar
      sabemos TipoDeObjeto nombreDeLaVariable, y luego se le asigna un valor, en este c
      nuevo de la clase ArrayList, los paréntesis que aparecen al final son porque esta
      al método constructor del objeto, sin parámetros en este caso, aunque puede haber
     */
    ArrayList<Integer> lista = new ArrayList<Integer>();
```

```
    //Llamamos al método para asignar un nuevo valor a la propiedad
    anadirValor(lista);

    /*
      Añadimos un objeto de la clase Integer (un 1), en el objeto lista, para ello llam
      add de la clase ArrayList, con un parámetro de tipo entero (el 1), este método aí
      En este caso, se añadirá en la posición 1 (la segunda).
     */
    lista.add(1);

    //Imprimimos el valor del primer elemento de la lista (posición 0), llamando al mét
    System.out.println(lista.get(0));
  }

  /*
    Se asigna un nuevo valor a la variable lista
   */
  static void anadirValor(ArrayList<Integer> lista) {
    lista.add(2);
  }
}
```

Figure 23: Cambio de propiedad

2

7 Clases y objetos