Módulo 1: Programación en Java

Roger F. González Camacho

Agosto 2023

Gobierno de Canarias
Servicio Canario de Empleo

Centro GASTRONÓMICO del Atlántico

GOBIERNO DE ESPAÑA
MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL
MINISTERIO DE TRABAJO Y ECONOMÍA SOCIAL
SERVICIO PÚBLICO DE EMPLEO ESTATAL
SEPE

1.1 Almacenamiento

- Variables.
- Constantes.
- Operadores.

Variables

Un programa maneja datos para hacer cálculos, presentarlos en informes por pantalla o impresora, solicitarlos al usuario, guardarlos en disco, etc. Para poder manejar esos datos, el programa los guarda en variables.

Sirven para almacenar datos durante la ejecución del programa; el valor asociado puede cambiar varias veces durante la ejecución del programa.

Variables

Una variable es una zona en la memoria del computador con un valor que puede ser almacenado para ser usado más tarde en el programa.

Las variables vienen determinadas por:

- **Un nombre**, que permite al programa acceder al valor que contiene en memoria. Debe ser un identificador válido.
- **Un tipo de dato**, que especifica qué clase de información guarda la variable en esa zona de memoria.
- **Un rango de valores** que puede admitir dicha variable, que vendrá determinado por el tipo de dato.
- Al nombre que le damos a la variable se le llama identificador. Los identificadores permiten nombrar los elementos que se están manejando en un programa. Vamos a ver con más detalle ciertos aspectos sobre los identificadores que debemos tener en cuenta.

Variables

Las variables se declaran de la siguiente forma:

```
tipoVariable identificadorVariable;
```

Por ejemplo:

```
int aux;
```

Identificadores

Un identificador en Java es una secuencia ilimitada sin espacios de letras y dígitos Unicode, de forma que el primer símbolo de la secuencia debe ser una letra, un símbolo de subrayado (_) o el símbolo dólar ($). Por ejemplo, son válidos los siguientes identificadores:

```
x5      αtη     NUM_MAX     numCuenta
```

Convenios y reglas para nombrar variables.

A la hora de nombrar un identificador existen una serie de normas de estilo de uso generalizado que, no siendo obligatorias, se usan en la mayor parte del código Java. Estas reglas para la nomenclatura de variables son las siguientes:

- **Java distingue las mayúsculas de las minúsculas**. Por ejemplo, Alumno y alumno son variables diferentes.
- **No se suelen utilizar identificadores que comiencen con «$» o «_»**, además el símbolo del dólar, por convenio, no se utiliza nunca.
- No se puede utilizar el valor booleano (true o false) ni el valor nulo (null).
- Los **identificadores deben ser lo más descriptivos posibles**. Es mejor usar palabras completas en vez de abreviaturas crípticas. Así nuestro código será más fácil de leer y comprender. En muchos casos también hará que nuestro código se autodocumente. Por ejemplo, si tenemos que darle el nombre a una variable que almacena los datos de un cliente sería recomendable que la misma se llamara algo así como FicheroClientes o ManejadorCliente, y no algo poco descriptivo como Cl33.

Convenios y reglas para nombrar variables.

Identificador	Convención	Ejemplo
Nombre de variable.	Comienza por letra minúscula, y si tienen más de una palabra se colocan juntas y el resto comenzando por mayúsculas.	numAlumnos, suma
Nombre de constante.	En letras mayúsculas, separando las palabras con el guion bajo, por convenio el guion bajo no se utiliza en ningún otro sitio.	TAM_MAX, PI
Nombre de una clase.	Comienza por letra mayúscula.	String, MiTipo
Nombre de función.	Comienza por letra mayúscula.	modifica_Valor, obtiene_Valor

Palabras reservadas.

Las palabras reservadas, a veces también llamadas palabras clave o keywords , son secuencias de caracteres formadas con letras ASCII cuyo uso se reserva al lenguaje y, por tanto, no pueden utilizarse para crear identificadores.

A-D	D-I	I-P	P-T	T-Z
abstract	do	implements	protected	throw
boolean	double	import	public	throws
break	else	instanceof	rest	transient
byte	extends	int	return	true
case	false	interface	short	try
catch	final	long	static	void
char	finally	native	strictfp	volatile
class	float	new	super	while
const*	for	null	switch	
continue	goto*	package	synchronized	
default	if	privative	this	

Tipos de variables.

1. **Variables de tipos primitivos y variables referencia**, según el tipo de información que contengan. En función de a qué grupo pertenezca la variable, tipos primitivos o tipos referenciados, podrá tomar unos valores u otros, y se podrán definir sobre ella unas operaciones u otras.

2. **Variables y constantes**, dependiendo de si su valor cambia o no durante la ejecución del programa. La definición de cada tipo sería:
   - **Variables**. Sirven para almacenar los datos durante la ejecución del programa, pueden estar formadas por cualquier tipo de dato primitivo o referencia. Su valor puede cambiar a lo largo de la ejecución del programa. Realmente una variable representa una zona de memoria del ordenador que contiene un determinado valor (del tipo de datos de la variable) y al que se accede a través del identificador.
   - **Constantes o variables finales:** Son aquellas variables cuyo valor no cambia a lo largo de todo el programa.

Ejemplos – tipos de variables

- **Variable constante llamada PI**: esta variable por haberla declarado como constante no podrá cambiar su valor a lo largo de todo el programa.

- **Variable miembro llamada x**: Esta variable pertenece a la clase ejemplovariables. La variable x puede almacenar valores del tipo primitivo int. El valor de esta variable podrá ser modificado en el programa, normalmente por medio de algún otro método que se cree en la clase.

- **Variable local llamada valorantiguo**: Esta variable es local porque está creada dentro del método obtenerX(). Sólo se podrá acceder a ella dentro del método donde está creada, ya que fuera de él no existe.

Ejemplos – tipos de variables

```java
/**
 * Aplicación ejemplo de tipos de variables

 *
 * @author FMA
 */
public class ejemplovariables {
    final double PI =3.1415926536;  // PI es una constante
    int x;                          // x es una variable miembro
                                    // de clase ejemplovariables

    int obtenerX(int x) {               // x es un parámetro
            int valorantiguo = this.x;  // valorantiguo es una variabe local
            return valorantiguo;
    }

    // el método main comienza la ejecución de la aplicación
    public static void main(String[] args) {
        // aquí iría el código de nuestra aplicación


    } // fin del método main

} // fin de la clase ejemplovariables
```

Tipos de datos

En los lenguajes fuertemente tipados, a todo dato (constante, variable o expresión) le corresponde un tipo que es conocido antes de que se ejecute el programa.

El tipo limita el valor de la variable o expresión, las operaciones que se pueden hacer sobre esos valores, y el significado de esas operaciones. Esto es así porque un tipo de dato no es más que una especificación de los valores que son válidos para esa variable , y de las operaciones que se pueden realizar con ellos.

- **Tipos de datos sencillos o primitivos**. Representan valores simples que vienen predefinidos en el lenguaje; contienen valores únicos, como por ejemplo un carácter o un número.
- **Tipos de datos referencia**. Se definen con un nombre o referencia (puntero) que contiene la dirección en memoria de un valor o grupo de valores. Dentro de este tipo tenemos por ejemplo los vectores o arrays, que son una serie de elementos del mismo tipo, o las clases, que son los modelos o plantillas a partir de los cuales se crean los objetos.

Tipos de datos primitivos

Los tipos primitivos son aquéllos datos sencillos que constituyen los tipos de información más habituales: números, caracteres y valores lógicos o booleanos.

Al contrario que en otros lenguajes de programación orientados a objetos, estas variables en Java no son objetos.

Declaración e inicialización.

Llegados a este punto cabe preguntarnos ¿Cómo se crean las variables en un programa? ¿Qué debo hacer antes de usar una variable en mi programa? Pues bien, como podrás imaginar, debemos crear las variables antes de poder utilizarlas en nuestros programas, indicando qué identificador (nombre) va a tener y qué tipo de información va a almacenar, en definitiva, debemos **declarar la variable.**

```
int nmAlumnos = 15;
double radio = 3.14, importe = 102.95;
```

Declaración e inicialización.

Si la variable va a permanecer inalterable a lo largo del programa, la declararemos como constante, utilizando la palabra reservada ***final*** de la siguiente forma:

```
final double PI = 3.1415926536;
```

Declaración e inicialización.

En ocasiones puede que al declarar una variable no le demos valor, ¿qué crees que ocurre en estos casos? Pues que el compilador le asigna un valor por defecto, aunque depende del tipo de variable que se trate:

- **Las variables miembro** sí se inicializan automáticamente, si no les damos un valor. Cuando son de tipo numérico, se inicializan por defecto a 0, si don de tipo carácter, se inicializan al carácter null (\0), si son de tipo boolean se les asigna el valor por defecto false, y si son tipo referenciado se inicializan a null.
- **Las variables locales** no se inicializan automáticamente. Debemos asignarles nosotros un valor antes de ser usadas, ya que si el compilador detecta que la variable se usa antes de que se le asigne un valor, produce un error.

```
int p;

int q = p; // error
```

```
int p;

if (. . . )

     p = 5 ;

int q = p; // error
```

Tipos referenciados

Estos tipos de datos se llaman tipos referenciados o referencias, porque se utilizan para almacenar la dirección de los datos en la memoria del ordenador.

```
int[] arrayDeEnteros;

Cuenta cuentaCliente;
```

Declaramos una lista de números del mismo tipo, en este caso, enteros. En la segunda instrucción estamos declarando la variable u objeto ***cuentaCliente*** como una referencia de tipo **Cuenta**.

Tipos referenciados

Estos tipos de datos se llaman tipos referenciados o referencias, porque se utilizan para almacenar la dirección de los datos en la memoria del ordenador.

```
String mensaje;

mensaje= "El primer programa";
```

Ejemplos - tipos referenciados


```
public class ejemplotipos {

    // el método main inicia la ejecución de la aplicación
    public static void main(String[] args) {
        // Código de la aplicación
        int i = 10;
        double d = 3.14;
        char c1 = 'a';
        char c2 = 65;
        boolean encontrado = true;
        String msj = "Bienvenido a Java";

        System.out.println("La variable i es de tipo entero y su valor es: " + i);
        System.out.println("La variable f es de tipo double y su valor es: "+d);
        System.out.println("La variable c1 es de tipo carácter y su valor es: "+c1);
        System.out.println("La variable c2 es de tipo carácter y su valor es: "+c2);
        System.out.println("La variable encontrado es de tipo booleano y su valor es: "+encontrado);
        System.out.println("La variable msj es de tipo String y su valor es: " + msj);
    } // fin del método main

} // fin de la clase ejemplotipos
```

Tipos enumerados

Son una forma de declarar una variable con un conjunto restringido de valores. Por ejemplo, los días de la semana, las estaciones del año, los meses, etc. Es como si definiéramos nuestro propio tipo de datos.

```
10  public class tiposenumerados {
11      public enum Dias {Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo};
12
13      public static void main(String[] args) {
14          // codigo de la aplicacion
15          Dias diaactual = Dias.Martes;
16          Dias diasiguiente = Dias.Miercoles;
17
18          System.out.print("Hoy es: ");
19          System.out.println(diaactual);
20          System.out.println("Mañana\nes\n"+diasiguiente);
21
22      } // fin main
23
24  } // fin tiposenumerados
```

Literales de los tipos primitivos

- **Un literal, valor literal o constante literal** es un valor concreto para los tipos de datos primitivos del lenguaje, el tipo String o el tipo null.
- **Los literales booleanos** tienen dos únicos valores que puede aceptar el tipo: true y false. Por ejemplo, con la instrucción boolean encontrado = true; estamos declarando una variable de tipo booleana a la cual le asignamos el valor literal true.
- **Los literales enteros** se pueden representar en tres notaciones:
  - **Decimal:** por ejemplo 20. Es la forma más común.
  - **Octal:** por ejemplo 024. Un número en octal siempre empieza por cero, seguido de dígitos octales (del 0 al 7).
  - **Hexadecimal:** por ejemplo 0x14. Un número en hexadecimal siempre empieza por 0x seguido de dígitos hexadecimales (del 0 al 9, de la 'a' a la 'f' o de la 'A' a la 'F').

Literales de los tipos primitivos

Secuencias de escape en Java

Secuencia de escape	Significado	Secuencia de escape	Significado
\b	Retroceso	\r	Retorno de carro
\t	Tabulador	\‘’	Carácter comillas dobles
\n	Salto de línea	\’	Carácter comillas simples
\f	Salto de página	\\	Barra diagonal

```
String texto = "Juan dijo: \"Hoy hace un día fantástico…\"";
```

Literales de los tipos primitivos

Tipo	Descripción	Tamaño (en bits)	Valor mínimo	Valor máximo	Ejemplos
boolean	Valor booleano	1	N.A.	N.A.	false, true
char	Carácter Unicode	16	\u0000	\uFFFF	a, A, n, N, ñ, Ñ
byte	Entero con signo	8	-128	128	-20, 127
short	Entero con signo	16	-332768	32767	-250, 2017
int	Entero con signo	32	-2147483648	2147483647	5, 512, 12345
long	Entero con signo	64	-9223372036854775808	9223372036854775807	36854775808
float	Coma flotante de precisión simple	32	±3.40282347E+38	±1.40239846E-45	0.0F
double	Coma flotante de precisión doble	64	±1.79769313486231570E+308	±4.94065645841246544E-324	0.0, 0.0D

Operadores y expresiones

Llevan a cabo operaciones sobre un conjunto de datos u operandos, representados por literales y/o identificadores. Los operadores pueden ser unarios, binarios o terciarios, según el número de operandos que utilicen sean uno, dos o tres, respectivamente. Los operadores actúan sobre los tipos de datos primitivos y devuelven también un tipo de dato primitivo.

Los operadores se combinan con los literales y/o identificadores para formar expresiones. Una expresión es una combinación de operadores y operandos que se evalúa produciendo un único resultado de un tipo determinado.

El resultado de una expresión puede ser usado como parte de otra expresión o en una sentencia o instrucción. Las expresiones, combinadas con algunas palabras reservadas o por sí mismas, forman las llamadas sentencias o instrucciones.

Operadores - Ejemplo

Por ejemplo, pensemos en la siguiente expresión Java:

```
i + 1;
```

Con esta expresión estamos utilizando un operador aritmético para sumarle una cantidad a la variable i, pero es necesario indicar al programa qué hacer con el resultado de dicha expresión:

```
suma = i + 1;
```

Operadores aritméticos

Los operadores aritméticos son aquellos operados que combinados con los operandos forman expresiones matemáticas o aritméticas.

Operadores aritméticos básicos

Operador	Operación Java	Expresión Java	Resultado
-	Operador unario de cambio de signo	-10	-10
+	Adición	1.2 + 9.3	10.5
-	Sustracción	312.5 – 12.3	300.2
*	Multiplicación	1.7 * 1.2	1.02
/	División (entera o real)	0.5 / 0.2	2.5
%	Resto de la división entera	25 % 3	1

Operadores aritméticos

Resultados de las operaciones aritméticas en Java

Tipo de los operandos	Resultado
Un operando de tipo `long` y ninguno real (`float` o `double`)	`long`
Ningún operando de tipo `long` ni real (`float` o `double`)	`int`
Al menos un operando de tipo `double`	`double`
Al menos un operando de tipo `float` y ninguno `double`	`float`

Operadores aritméticos

Operadores incrementales en Java

Tipo operador	Expresión Java	
	Prefija:	Postfija:
++ (incremental)	x=3;	x=3;
	y=++x;	y=x++;
	// x vale 4 e y vale 4	// x vale 4 e y vale 3
--(decremental)	5-- // el resultado es 4	

Operadores aritméticos - Ejemplos

```
10  public class operadoresaritmeticos {
11      public static void main(String[] args) {
12          short x = 7;
13          int y = 5;
14          float f1 = 13.5f;
15          float f2 = 8f;
16          System.out.println("El valor de x es " + x +" y el valor de y es " +y);
17          System.out.println("El resultado de x + y es " + (x + y));
18          System.out.println("El resultado de x - y es " + (x - y));
19          System.out.printf("%s\n%s%s\n","División entera:","x / y = ",(x/y));
20          System.out.println("Resto de la división entera: x % y = " + (x % y));
21          System.out.printf("El valor de f1 es %f y el de f2 es %f\n",f1,f2);
22          System.out.println("El resultado de f1 / f2 es " + (f1 / f2));
23      } // fin de main
24  } // fin de la clase operadoresaritmeticos
```

Operadores de asignación

El principal operador de esta categoría es el operador asignación "=", que permite al programa darle un valor a una variable, y ya hemos utilizado varias ocasiones en esta unidad. Además de este operador, Java proporciona otros operadores de asignación combinados con los operadores aritméticos, que permiten abreviar o reducir ciertas expresiones.

Por ejemplo, el operador "+=" suma el valor de la expresión a la derecha del operador con el valor de la variable que hay a la izquierda del operador, y almacena el resultado en dicha variable. En la siguiente tabla se muestran todos los operadores de asignación compuestos que podemos utilizar en Java

Operadores de asignación

Ejemplo

Operadores de asignación combinados en Java

Operador	Ejemplo en Java	Expresión equivalente
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2

Operadores de asignación

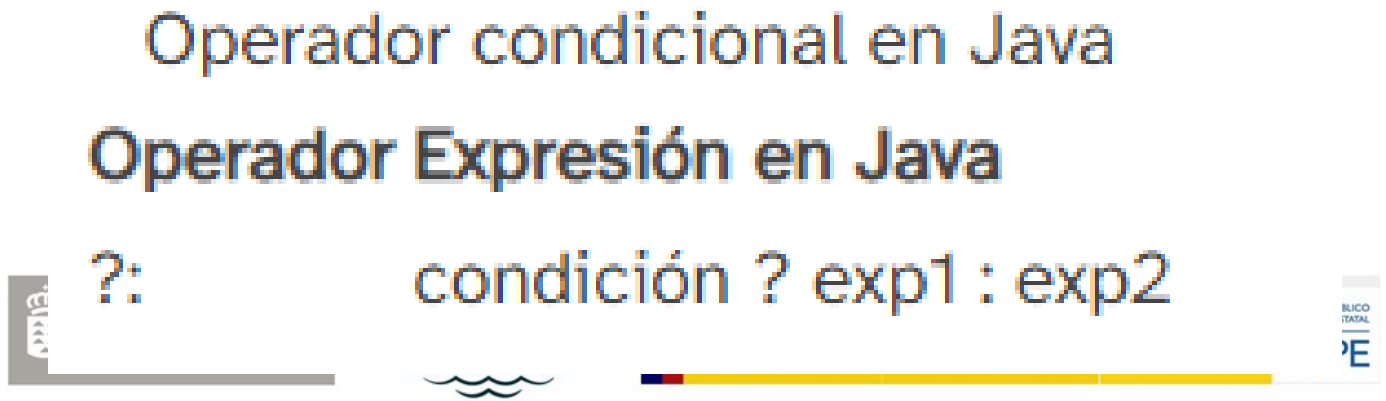
Ejemplo

```
public class operadoresasignacion {

    // clase principal main que inicia la aplicación
    public static void main(String[] args) {
        int x;
        int y;
        x = 5;  // operador asignación
        y = 3;  // operador asignación

        //operadores de asignación combinados
        System.out.printf("El valor de x es %d y el valor de y es %d\n", x,y);
        x += y;
        // podemos utilizar indistintamente printf o println
        System.out.println(" Suma combinada: x += y " + " ..... x vale " + x);
        x = 5;
        x -= y;
        System.out.println(" Resta combinada: x -= y " + " ..... x vale " + x);
        x = 5;
        x *= y;
        System.out.println(" Producto combinado: x *= y " + " ..... x vale " + x);
        x = 5;
        x /= y;
        System.out.println(" Division combinada: x /= y " + " ..... x vale " + x);
        x = 5;
        x %= y;
        System.out.println(" Resto combinada: x %= y " + " ..... x vale " + x);
    } // fin main
} // fin operadoresasignacion
```

Operador condicional

El operador condicional “?:” sirve para evaluar una condición y devolver un resultado en función de si es verdadera o falsa dicha condición. Es el único operador ternario de Java, y como tal, necesita tres operandos para formar una expresión.

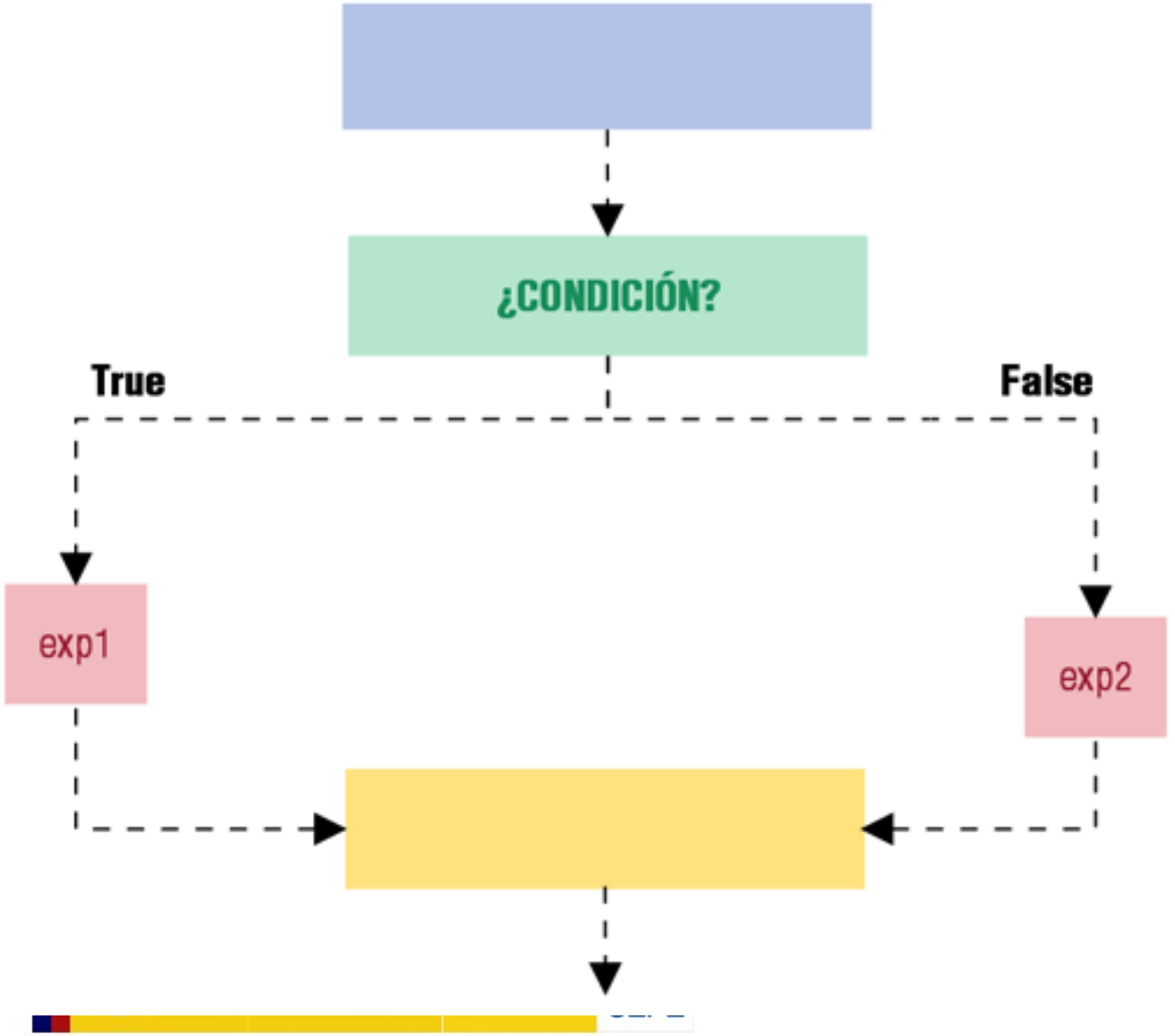
El primer operando se sitúa a la izquierda del símbolo de interrogación, y siempre será una expresión booleana, también llamada condición. El siguiente operando se sitúa a la derecha del símbolo de interrogación y antes de los dos puntos, y es el valor que devolverá el operador condicional si la condición es verdadera. El último operando, que aparece después de los dos puntos, es la expresión cuyo resultado se devolverá si la condición evaluada es falsa.



Operador condicional

Ejemplo

```
(x>y)?x:y;
```



Operadores de relación

Los operadores relacionales se utilizan para comparar datos de tipo primitivo (numérico, carácter y booleano). El resultado se utilizará en otras expresiones o sentencias, que ejecutarán una acción u otra en función de si se cumple o no la relación.

Estas expresiones en Java dan siempre como resultado un valor booleano true o false. En la tabla siguiente aparecen los operadores relacionales en Java.

Operadores relacionales en Java

Operador	Ejemplo en Java	Significado
==	op1 == op2	op1 igual a op2
!=	op1 != op2	op1 distinto de op2
>	op1 > op2	op1 mayor que op2
<	op1 < op2	op1 menor que op2
>=	op1 >= op2	op1 mayor o igual que op2
<=	op1 <= op2	op1 menor o igual que op2

Operadores de relación

Ejemplo

```
public class ejemplorelacionales {
   // método principal que inicia la aplicación
   public static void main( String args[] )
   {
      // clase Scanner para petición de datos
      Scanner teclado = new Scanner( System.in );
      int x, y;
      String cadena;
      boolean resultado;
      System.out.print( "Introducir primer número: " );
      x = teclado.nextInt(); // pedimos el primer número al usuario
      System.out.print( "Introducir segundo número: " );
      y = teclado.nextInt(); // pedimos el segundo número al usuario
      // realizamos las comparaciones
      cadena=(x==y)?"iguales":"distintos";
      System.out.printf("Los números %d y %d son %s\n",x,y,cadena);
      resultado=(x!=y);
      System.out.println("x != y  // es " + resultado);
      resultado=(x < y );
      System.out.println("x < y  // es " + resultado);
      resultado=(x > y );
      System.out.println("x > y  // es " + resultado);
      resultado=(x <= y );
      System.out.println("x <= y  // es " + resultado);
      resultado=(x >= y );
      System.out.println("x >= y  // es " + resultado);
    } // fin método main
} // fin clase ejemplorelacionales
```

Operadores lógicos

Los operadores lógicos realizan operaciones sobre valores booleanos, o resultados de expresiones relacionales, dando como resultado un valor booleano.

Los operadores lógicos los podemos ver en la tabla que se muestra a continuación. Existen ciertos casos en los que el segundo operando de una expresión lógica no se evalúa para ahorrar tiempo de ejecución, porque con el primero ya es suficiente para saber cuál va a ser el resultado de la expresión.

Por ejemplo, en la expresión ***a && b*** si a es falso, no se sigue comprobando la expresión, puesto que ya se sabe que la condición de que ambos sean verdadero no se va a cumplir. En estos casos es más conveniente colocar el operando más propenso a ser falso en el lado de la izquierda. Igual ocurre con el operador **||**, en cuyo caso es más favorable colocar el operando más propenso a ser verdadero en el lado izquierdo.

Operadores lógicos

Ejemplo

Operadores lógicos en Java

Operador	Ejemplo en Java	Significado
!	!op	Devuelve true si el operando es false y viceversa.
&	op1 & op2	Devuelve true si op1 y op2 son true
|	op1 | op2	Devuelve true si op1 u op2 son true
^	op1 ^ op2	Devuelve true si sólo uno de los operandos es true
&&	op1 && op2	Igual que &, pero si op1 es false ya no se evalúa op2
||	op1 || op2	Igual que |, pero si op1 es true ya no se evalúa op2

Operadores lógicos

Ejemplo

```
public class operadoreslogicos {
  public static void main(String[] args) {
    System.out.println("OPERADORES LÓGICOS");

    System.out.println("Negacion:\n ! false es : " + (! false));
    System.out.println(" ! true es : " + (! true));

    System.out.println("Operador AND (&):\n false & false es : " + (false & false));
    System.out.println(" false & true es : " + (false & true));
    System.out.println(" true & false es : " + (true & false));
    System.out.println(" true & true es : " + (true & true));

    System.out.println("Operador OR (|):\n false | false es : " + (false | false));
    System.out.println(" false | true es : " + (false | true));
    System.out.println(" true | false es : " + (true | false));
    System.out.println(" true | true es : " + (true | true));

    System.out.println("Operador OR Exclusivo (^):\n false ^ false es : " + (false ^ false));
    System.out.println(" false ^ true es : " + (false ^ true));
    System.out.println(" true ^ false es : " + (true ^ false));
    System.out.println(" true ^ true es : " + (true ^ true));

    System.out.println("Operador &&:\n false && false es : " + (false && false));
    System.out.println(" false && true es : " + (false && true));
    System.out.println(" true && false es : " + (true && false));
    System.out.println(" true && true es : " + (true && true));

    System.out.println("Operador ||:\n false || false es : " + (false || false));
    System.out.println(" false || true es : " + (false || true));
    System.out.println(" true || false es : " + (true || false));
    System.out.println(" true || true es : " + (true || true));
  } // fin main
} // fin operadoreslogicos
```

Trabajo con cadenas.

Ya hemos visto en el apartado de literales que el objeto **String** se corresponde con una secuencia de caracteres entrecomillados, como por ejemplo “hola”. Este literal se puede utilizar en Java como si de un tipo de datos primitivo se tratase, y, como caso especial, no necesita la orden new para ser creado.

Para aplicar una operación a una variable de tipo String, escribiremos su nombre seguido de la operación, separados por un punto. Entre las principales operaciones que podemos utilizar para trabajar con cadenas de caracteres están las siguientes:

- **Creación.** Como hemos visto en el apartado de literales, podemos crear una variable de tipo String simplemente asignándole una cadena de caracteres encerrada entre comillas dobles.

Trabajo con cadenas.

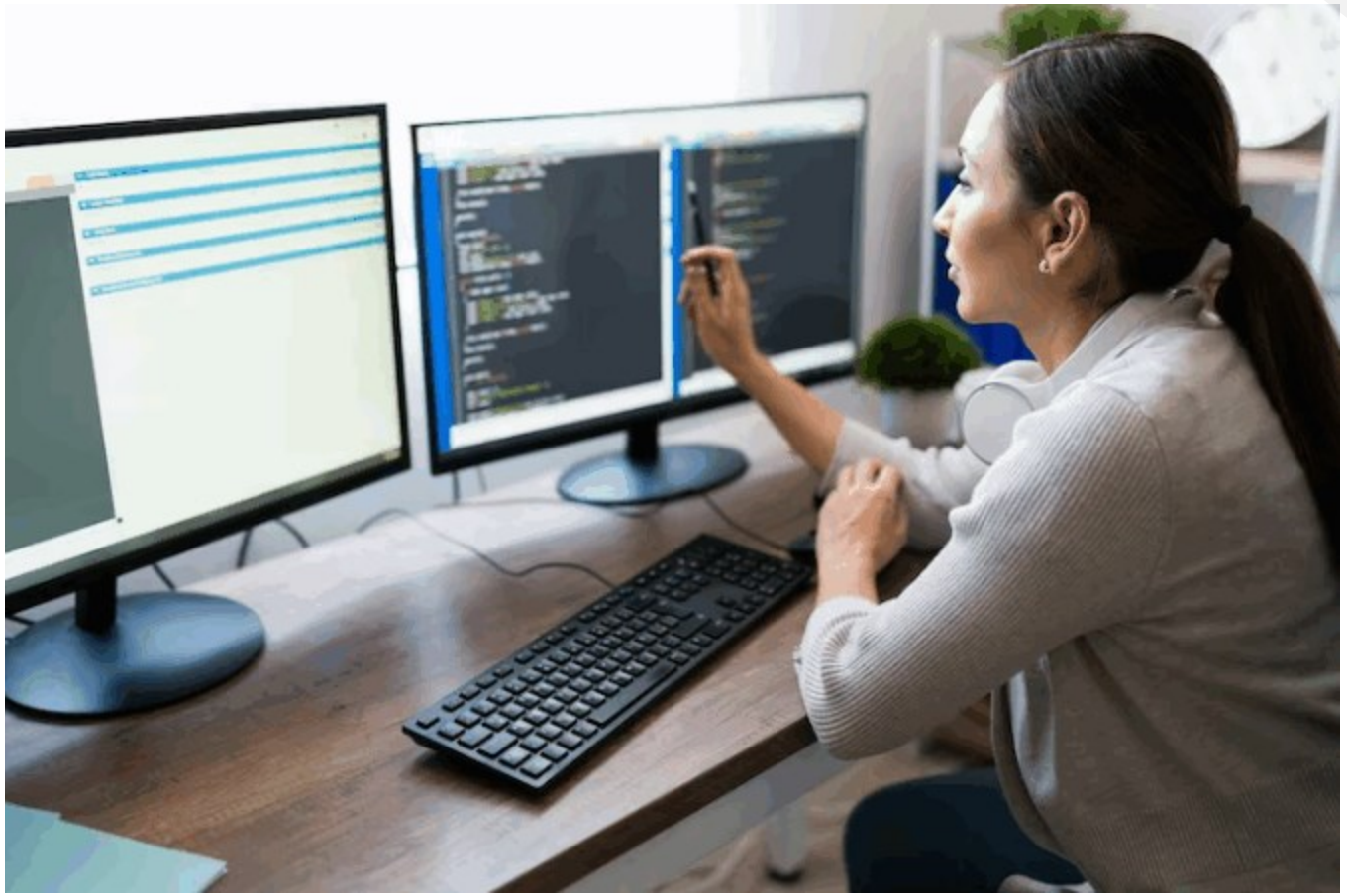
- **Obtención de longitud**. Si necesitamos saber la longitud de un String, utilizaremos el método length().
- **Concatenación**. Se utiliza el operador + o el método concat() para concatenar cadenas de caracteres.
- **Comparación.** El método equals() nos devuelve un valor booleano que indica si las cadenas comparadas son o no iguales. El método equalsIgnoreCase() hace lo propio, ignorando las mayúsculas de las cadenas a considerar.
- **Obtención de subcadenas.** Podemos obtener cadenas derivadas de una cadena original con el método substring(), al cual le debemos indicar el inicio y el fin de la subcadena a obtener.
- **Cambio a mayúsculas/minúsculas**. Los métodos toUpperCase() y toLowerCase() devuelven una nueva variable que transforma en mayúsculas o minúsculas, respectivamente, la variable inicial.
- **Valueof.** Utilizaremos este método para convertir un tipo de dato primitivo (int, long, float, etc.) a una variable de tipo String.

Trabajo con cadenas.

Ejemplo

```
public class ejemplocadenas {
    public static void main(String[] args)
    {
            String cad1 = "CICLO DAM";
            String cad2 = "ciclo dam";

            System.out.printf( "La cadena cad1 es: %s y cad2 es: %s", cad1,cad2 );

            System.out.printf( "\nLongitud de cad1: %d", cad1.length() );

            // concatenación de cadenas (concat o bien operador +)
            System.out.printf( "\nConcatenación: %s", cad1.concat(cad2) );

            //comparación de cadenas
            System.out.printf("\ncad1.equals(cad2) es %b", cad1.equals(cad2) );
            System.out.printf("\ncad1.equalsIgnoreCase(cad2) es %b", cad1.equalsIgnoreCase(cad2) );
            System.out.printf("\ncad1.compareTo(cad2) es %d", cad1.compareTo(cad2) );

            //obtención de subcadenas
            System.out.printf("\ncad1.substring(0,5) es %s", cad1.substring(0,5) );

            //pasar a minúsculas
            System.out.printf("\ncad1.toLowerCase() es %s", cad1.toLowerCase() );

            System.out.println();
    } // fin main

} // fin ejemplocadenas
```

Ejercicios



Cadenas de caracteres.

Probablemente, una de las cosas que mas utilizarás cuando estés programando en cualquier lenguaje de programación son las cadenas de caracteres. Las cadenas de caracteres son estructuras de almacenamiento que permiten almacenar una secuencia de caracteres de casi cualquier longitud. Y la pregunta ahora es, ¿qué es un carácter?

En Java y en todo lenguaje de programación, y por ende, en todo sistema informático, los caracteres se codifican como secuencias de bits que representan a los símbolos usados en la comunicación humana. Estos símbolos pueden ser letras, números, símbolos matemáticos e incluso ideogramas y pictogramas.

Cadenas de caracteres.

```
String cad="Ejemplo de cadena";
```

```
String cad=new String ("Ejemplo de cadena");
```

Operaciones avanzadas con cadenas de caracteres (I).

¿Qué operaciones puedes hacer con una cadena? Muchas más de las que te imaginas. Empezaremos con la operación mas sencilla: la concatenación. La concatenación es la unión de dos cadenas, para formar una sola. En Java es muy sencillo, pues sólo tienes que utilizar el operador de concatenación (signo de suma):

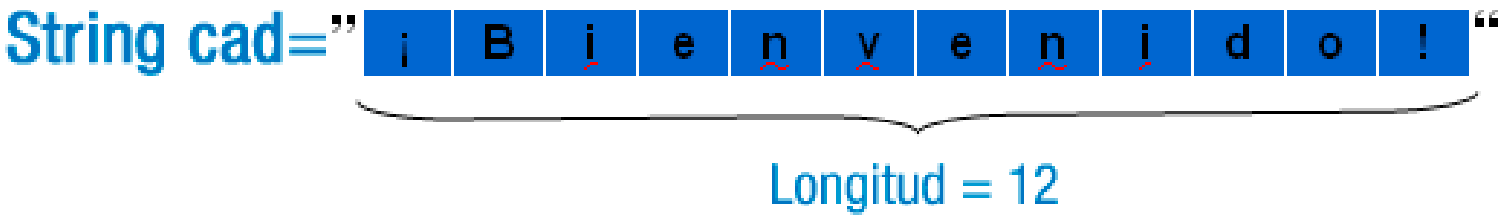
```
String cad = "¡Bien"+"venido!";

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

```
String cad="¡Bien".concat("venido!");

System.out.printf(cad);
```

Operaciones avanzadas con cadenas de caracteres (II).

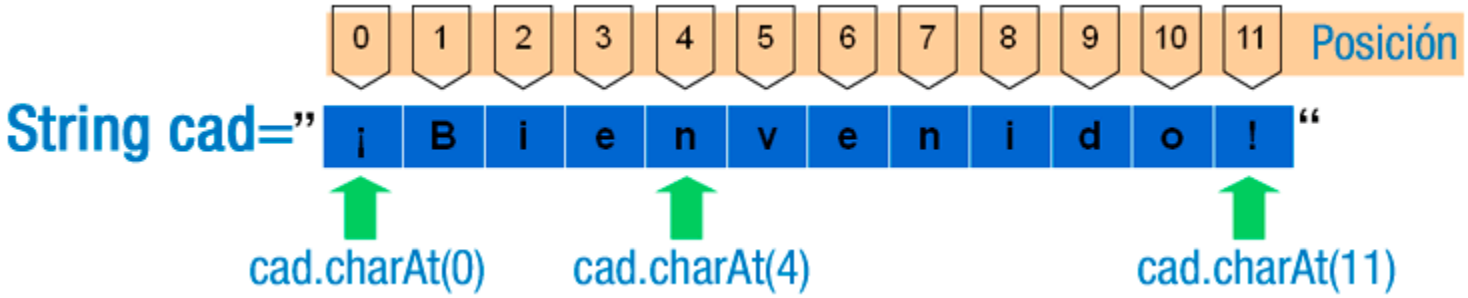
Vamos a continuar revisando las operaciones que se pueden realizar con cadenas. Como verás las operaciones a realizar se complican un poco a partir de ahora. En todos los ejemplos la variable cad contiene la cadena "¡Bienvenido!", como se muestra en las imágenes.

- **int length( ).** Retorna un número entero que contiene la longitud de una cadena, resultado de contar el número de caracteres por la que esta compuesta. Recuerda que un espacio es también un carácter.



Operaciones avanzadas con cadenas de caracteres (II).

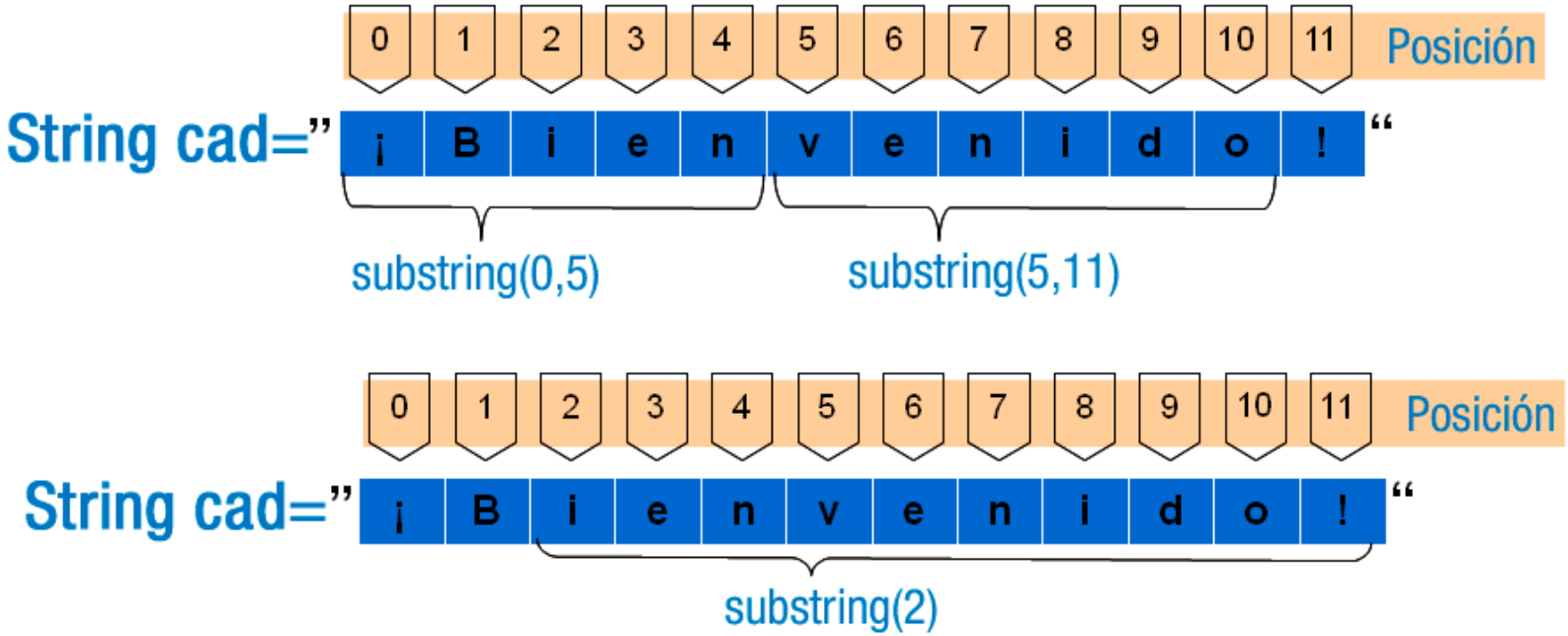
- **char charAt(int pos).** Retorna el carácter ubicado en la posición pasada por parámetro. El carácter obtenido de dicha posición será almacenado en un tipo de dato char. Las posiciones se empiezan a contar desde el 0 (y no desde el 1), y van desde 0 hasta longitud - 1.Por ejemplo, el código siguiente mostraría por pantalla el carácter "v":




```
char t = cad.charAt(5);

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

Operaciones avanzadas con cadenas de caracteres (II).

- **String substring(int beginIndex, int endIndex).** Este método permite extraer una subcadena de otra de mayor tamaño. Una cadena compuesta por todos los caracteres existentes entre la posición beginIndex y la posición endIndex - 1. Por ejemplo, si pusiéramos cad.substring(0,5) en nuestro programa, sobre la variable cad anterior, dicho método devolvería la subcadena "¡Bien" tal y como se muestra en la imagen.



Operaciones avanzadas con cadenas de caracteres (II).

- **String substring (int beginIndex).** Cuando al método substring solo le proporcionamos un parámetro, extraerá una cadena que comenzará en el carácter con posición beginIndex e irá hasta el final de la cadena. En el siguiente ejemplo se mostraría por pantalla la cadena "ienvenido!":

```
String subcad = cad.substring(2);

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

```
String cad2="Número cinco: " + 5;

System.out.println(cad2);
```

Operaciones avanzadas con cadenas de caracteres (III).

¿Cómo comprobarías que la cadena "3" es mayor que 0? No puedes comparar directamente una cadena con un número, así que necesitarás aprender cómo convertir cadenas que contienen números a tipos de datos numéricos (int, short, long, float o double). Esta es una operación habitual en todos los lenguajes de programación, y Java, para este propósito, ofrece los métodos valueOf, existentes en todas las clases envoltorio descendientes de la clase Number: Integer, Long, Short, Float y Double.

```
String c="1234.5678";

double n;
try {

     n=Double.valueOf(c).doubleValue();

} catch (NumberFormatException e)

{  /* Código a ejecutar si no se puede convertir */  }
```

Operaciones avanzadas con cadenas de caracteres (IV).

¿Cómo puedo comprobar si dos cadenas son iguales? ¿Qué más operaciones ofrece Java sobre las cadenas? Java ofrece un montón de operaciones más sobre cadenas de caracteres. En la siguiente tabla puedes ver las operaciones más importantes. En todos los ejemplos expuestos, las variables cad1, cad2 y cad3 son cadenas ya existentes, y la variable num es un número entero mayor o igual a cero.

Métodos importantes de la clase String.

Método.	Descripción
`cad1.compareTo(cad2)`	Permite comparar dos cadenas entre sí lexicográficamente. Retornará 0 si son iguales, un número menor que cero si la cadena (`cad1`) es anterior en orden alfabético a la que se pasa por argumento (`cad2`), y un número mayor que cero si la cadena es posterior en orden alfabético.
`cad1.equals(cad2)`	Cuando se comparan si dos cadenas son iguales, no se debe usar el operador de comparación "`==`", sino el método `equals`. Retornará `true` si son iguales, y `false` si no lo son.
`cad1.compareToIgnoreCase(cad2)` `cad1.equalsIgnoreCase(cad2)`	El método `compareToIgnoreCase` funciona igual que el método compareTo, pero ignora las mayúsculas y las minúsculas a la hora de hacer la comparación. Las mayúsculas van antes en orden alfabético que las minúsculas, por lo que hay que tenerlo en cuenta. El método `equalsIgnoresCase` es igual que el método `equals` pero sin tener en cuenta las minúsculas.
`cad1.trim()`	Genera una copia de la cadena eliminando los espacios en blanco anteriores y posteriores de la cadena.
`cad1.toLowerCase()`	Genera una copia de la cadena con todos los caracteres a minúscula.
`cad1.toUpperCase()`	Genera una copia de la cadena con todos los caracteres a mayúsculas.
`cad1.indexOf(cad2)` `cad1.indexOf(cad2,num)`	Si la cadena o carácter pasado por argumento está contenida en la cadena invocante, retorna su posición, en caso contrario retornará -1. Opcionalmente se le puede indicar la posición a partir de la cual buscar, lo cual es útil para buscar varias apariciones de una cadena dentro de otra.
`cad1.contains(cad2)`	Retornará `true` si la cadena pasada por argumento está contenida dentro de la cadena. En caso contrario retornará `false`.
`cad1.startsWith(cad2)`	Retornará `true` si la cadena comienza por la cadena pasada como argumento. En caso contrario retornará `false`.
`cad1.endsWith(cad2)`	Retornará `true` si la cadena acaba por la cadena pasada como argumento. En caso contrario retornará `false`.
`cad1.replace(cad2,cad3)`	Generará una copia de la cadena `cad1`, en la que se reemplazarán todas las apariciones de `cad2` por `cad3`. El reemplazo se hará de izquierda a derecha, por ejemplo: reemplazar "`zzz`" por "`xx`" en la cadena "`zzzzz`" generará "`xxzz`" y no "`zzxx`".

Ejercicios

1.2 Estructuras de control

- Condicionales (IF).
- Iterativas.
  - For
  - While
- Tratamiento de errores.

Sentencias de control de flujo.

¿Como habrás deducido, con lo que sabemos hasta ahora no es suficiente. 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.

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

Sentencias de control de flujo.

- **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 caso la estructura de repetición se detendrá.

Sentencias y bloques.

- **¿Cómo se escribe un programa sencillo?** Si queremos que un programa sencillo realice instrucciones o sentencias para obtener un determinado resultado, es necesario colocar éstas una detrás de la otra, exactamente en el orden en que deben ejecutarse.

- **¿Podrían colocarse todas las sentencias una detrás de otra, separadas por puntos y comas en una misma línea?** Claro que sí, pero no es muy recomendable. Cada sentencia debe estar escrita en una línea, de esta manera tu código será mucho más legible y la localización de errores en tus programas será más sencilla y rápida. De hecho, cuando se utilizan herramientas de programación, los errores suelen asociarse a un número o números de línea.

- **¿Puede una misma sentencia ocupar varias líneas en el programa?** Sí. Existen sentencias que, por su tamaño, pueden generar varias líneas. Pero siempre finalizarán con un punto y coma.

Sentencias y bloques.

- **¿En Java todas las sentencias se terminan con punto y coma?** Efectivamente. Si detrás de una sentencia ha de venir otra, pondremos un punto y coma. Escribiendo la siguiente sentencia en una nueva línea. Pero en algunas ocasiones, sobre todo cuando utilizamos estructuras de control de flujo, detrás de la cabecera de una estructura de este tipo no debe colocarse punto y coma. No te preocupes, lo entenderás cuando analicemos cada una de ellas.

- **¿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. La siguiente tabla muestra dos formas de construir un bloque de sentencias.

Estructuras de selección.

¿Cómo conseguimos que nuestros programas puedan tomar decisiones? Para comenzar, lo haremos a través de las estructuras de selección. Estas estructuras constan de una sentencia especial de decisión y de un conjunto de secuencias de instrucciones.

- Las estructuras de selección se dividen en:
  - Estructuras de selección simples o estructura if.
  - Estructuras de selección compuestas o estructura if-else.
  - Estructuras de selección basadas en el operador condicional.
  - Estructuras de selección múltiples o estructura switch.

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:

```
if (expresión-lógica) //al ser una sola instrucción, no son necesarias las llaves

sentencia1;
```

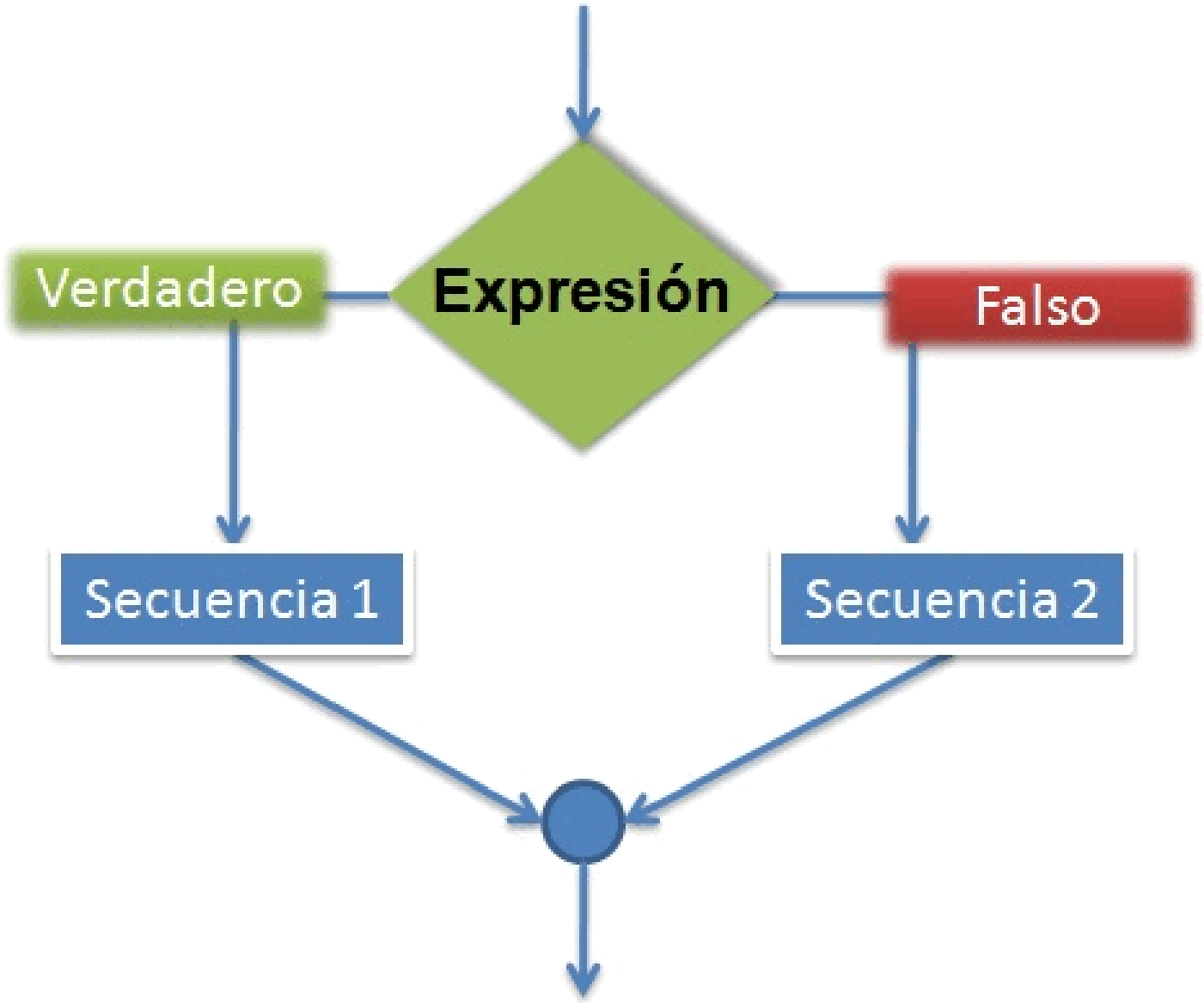
Estructura if / if-else.

```
if (expresión-lógica)
{

    sentencia1;

    sentencia2;

    …;

    sentenciaN;

}
```

```
if (expresión-lógica) //al ser una sola instrucción, no son necesarias las llaves

    sentencia1;

else

    sentencia2;
```

```
if (expresión-lógica)
{

    sentencia1;

    …;

    sentenciaN;

}

else

{

    sentencia1;

    …;

    sentenciaN;

}
```

Estructura if / if-else.



Estructura if / if-else. Ejemplo

```
//Java Program to demonstate the use of if statement.
public class IfExample {
public static void main(String[ ] args) {
   //defining an 'age' variable
   int age=20;
   //checking the age
   if(age>18){
      System.out.print("Age is greater than 18");
   }
}
}
```

Estructura if / if-else. Ejemplo

```
//A Java Program to demonstrate the use of if-else statement.
//It is a program of odd and even number.
public class IfElseExample {
public static void main(String[ ] args) {
   //defining a variable
   int number=13;
   //Check if the number is divisible by 2 or not
   if(number%2==0){
      System.out.println("even number");
   }else{
      System.out.println("odd number");
   }
}
}
```

Estructura if / if-else. Ejemplo

```
public class LeapYearExample {
public static void main(String[ ] args) {
   int year=2020;
   if(((year % 4 ==0) && (year % 100 !=0)) || (year % 400==0)){
      System.out.println("LEAP YEAR");
   }
   else{
      System.out.println("COMMON YEAR");
   }
}
}
```

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.

Estructura switch.

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

Estructura switch.

Condiciones:

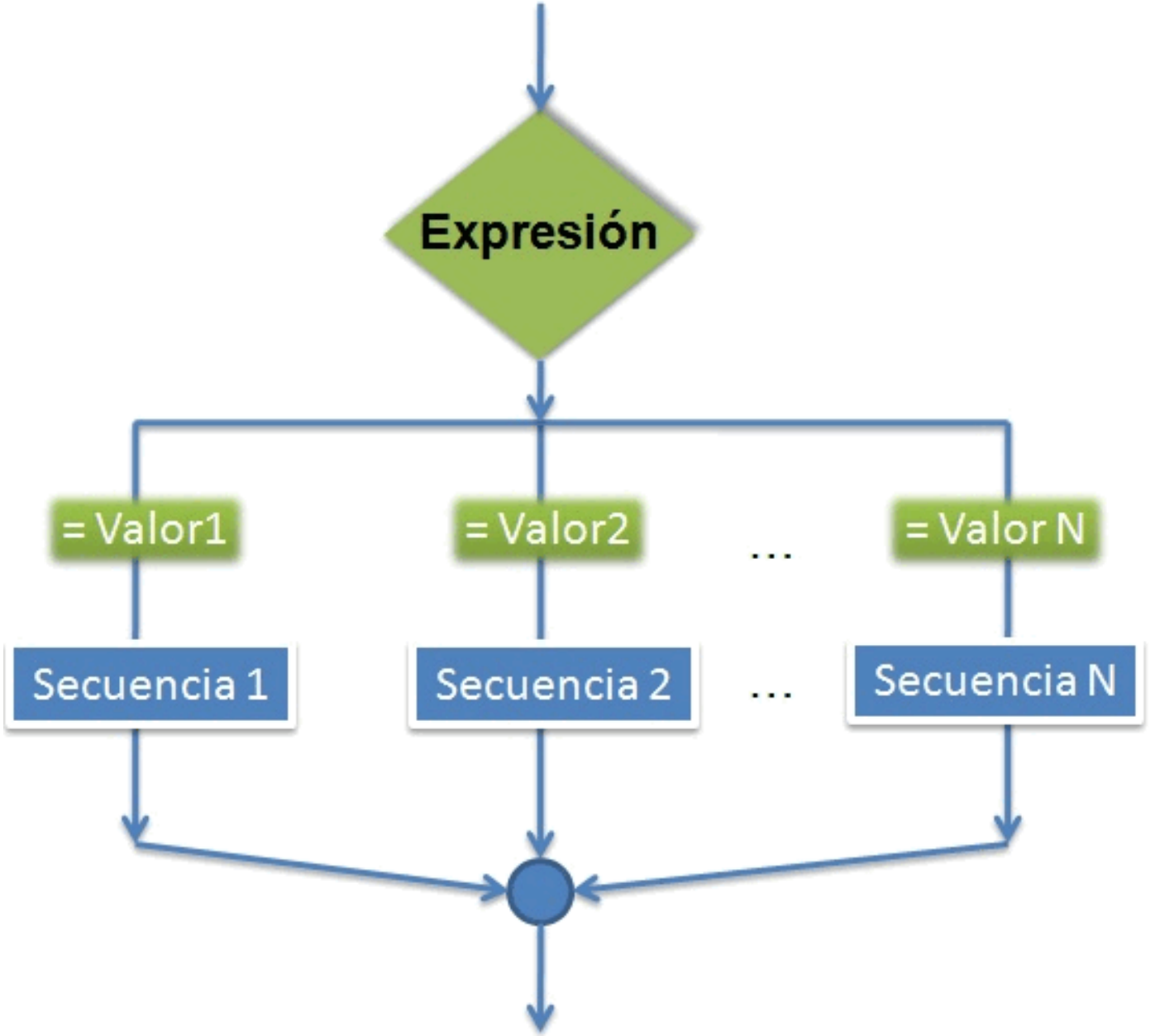
- Donde expresión debe ser del tipo char, byte, short o int, y las constantes de cada case deben ser de este tipo o de un tipo compatible.

- 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.

Estructura switch.



Estructura switch. Ejemplo

```
public class SwitchExample {
public static void main(String[ ] args) {
   //Declaring a variable for switch expression
   int number=20;
   //Switch expression
   switch(number){
   //Case statements
   case 10: System.out.println("10");
   break;
   case 20: System.out.println("20");
   break;
   case 30: System.out.println("30");
   break;
   //Default case statement
   default:System.out.println("Not in 10, 20 or 30");
   }
}
}
```

Estructuras de repetición.

Nuestros programas ya son capaces de controlar su ejecución teniendo en cuenta determinadas condiciones, pero aún hemos de aprender un conjunto de estructuras que nos permita repetir una secuencia de instrucciones determinada. La función de estas estructuras es repetir la ejecución de una serie de instrucciones teniendo en cuenta una condición.

A este tipo de estructuras se las denomina estructuras de repetición, estructuras repetitivas, bucles o estructuras iterativas. 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)**

Estructura for.

Estructura for.

Hemos indicado anteriormente que 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.

Estructura for.

Bucle con una sola sentencia y con un bloque de sentencias:

```
for (inicialización; condición; iteración)

   sentencia;  //Con una sola instrucción no es necesario utilizar llaves
```

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

   sentencia1;

   sentencia2;

   ...

   sentenciaN;

}
```

Donde:

- **inicialización** es una expresión en la que se inicializa una variable de control, que será la encargada de controlar el final del bucle.
- **condición** es una expresión que evaluará la variable de control. Mientras la condición sea falsa, el cuerpo del bucle estará repitiéndose. 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.

Estructura for. Ejemplo

```
//Java Program to demonstrate the example of for loop
//which prints table of 1
public class ForExample {
public static void main(String[ ] args) {
   //Code of Java for loop
   for(int i=1;i<=10;i++){
      System.out.println(i);
   }
}
```

Estructura for.

Ejemplo

```
public class NestedForExample {
public static void main(String[ ] args) {
//loop of i
for(int i=1;i<=3;i++){
//loop of j
for(int j=1;j<=3;j++){
      System.out.println(i+" "+j);
}//end of i
}//end of j
}
} }
```

Estructura for.

Ejemplo

```
//Java For-each loop example which prints the
//elements of the array
public class ForEachExample {
public static void main(String[ ] args) {
   //Declaring an array
   int arr[ ]={12,23,44,56,78};
   //Printing array using for-each loop
   for(int i:arr){
      System.out.println(i);
   }
}
}
```

Estructura for o for/in. Ejemplo

```
public class repetitiva_for_in {
     public static void main(String[] args) {
        // Declaración e inicialización de variables
        String[] semana = {"Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"};

        //Salida de información

        //Utilizamos ahora el bucle for/in
        for (String dia: semana){
        /* La cabecera del bucle incorpora la declaración de la variable dia
         * a modo de contenedor temporal de cada uno de los elementos que forman
         * el array semana.
         * En cada una de las iteraciones del bucle, se irá cargando en la variable
         * dia el valor de cada uno de los elementos que forman el array semana,
         * desde el primero al último.
         */
            System.out.println(dia);

        }
    }

}
```

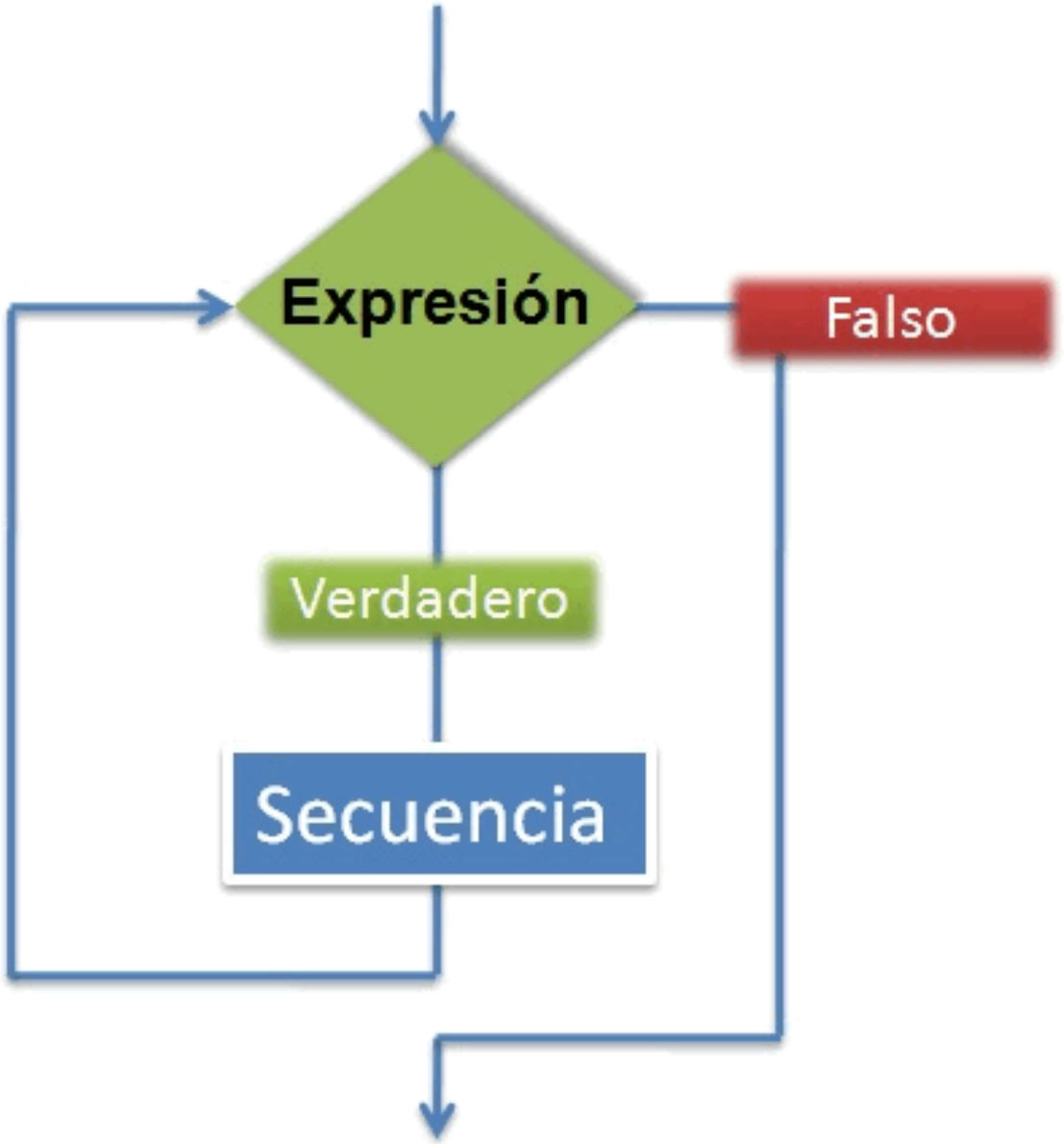
Estructura while.

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?**

```
while (condición)

sentencia;
```

```
while (condición) {

    sentencia1;

    …

    sentenciaN;

}
```

Estructura while.



Estructura while. Ejemplo

```
public class WhileExample {
public static void main(String[ ] args) {
   int i=1;
   while(i<=10){
      System.out.println(i);
   i++;
   }
}
}
```

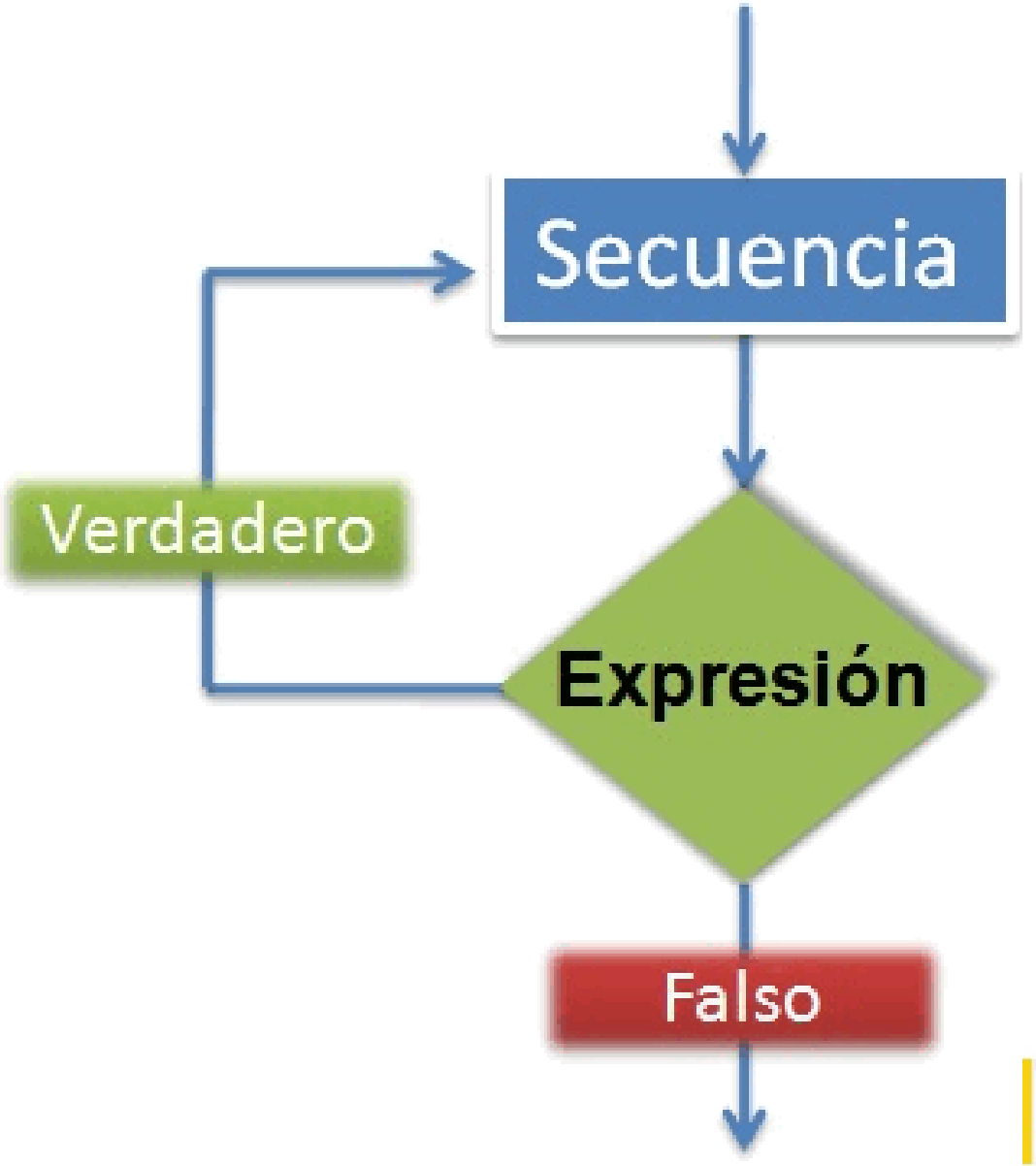
Estructura do-while.

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?**

```
do
   sentencia; //Con una sola sentencia no son necesarias las llaves

while (condición);
```

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

Estructura do-while.



Estructura do-while. Ejemplo

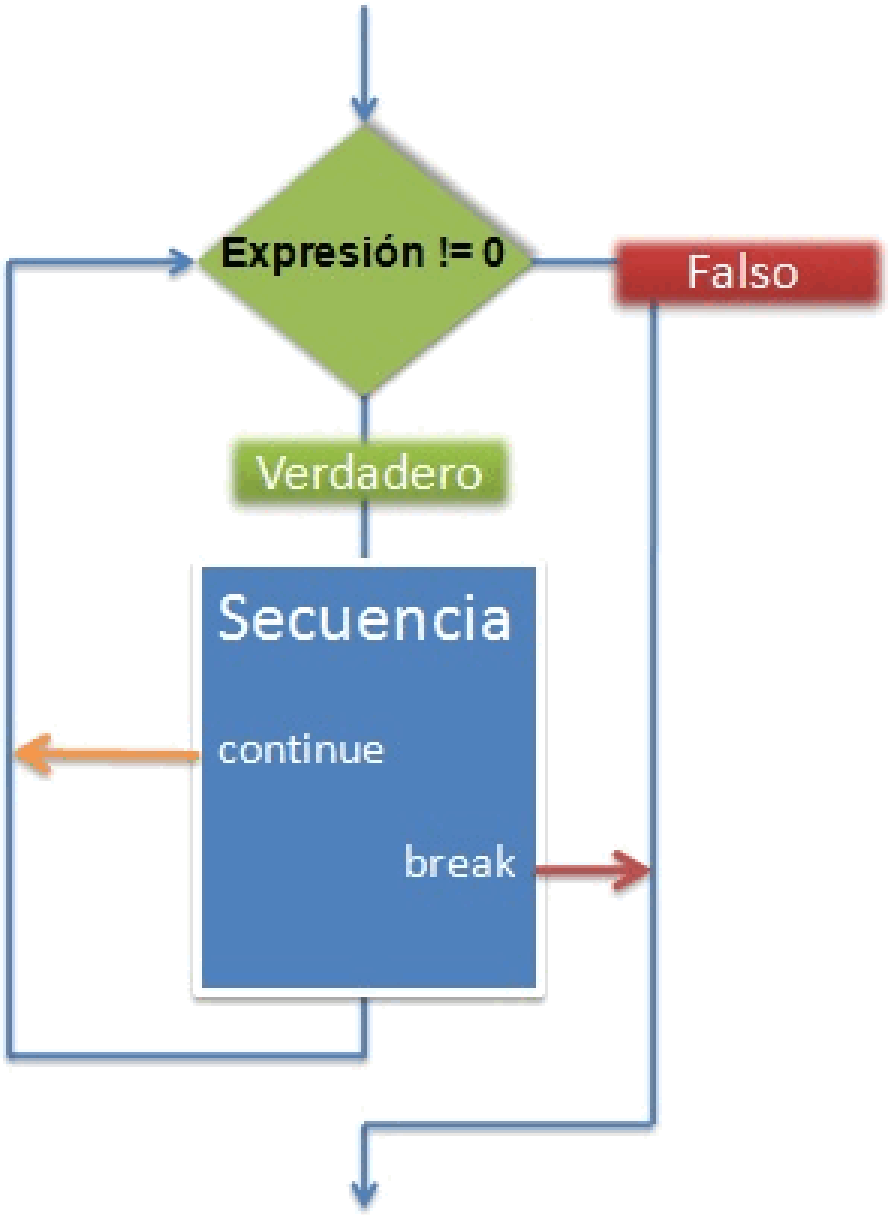
```
public class DoWhileExample {
public static void main(String[ ] args) {
   int i=1;
   do{
      System.out.println(i);
   i++;
   }while(i<=10);
}
}
```

Estructuras de salto.

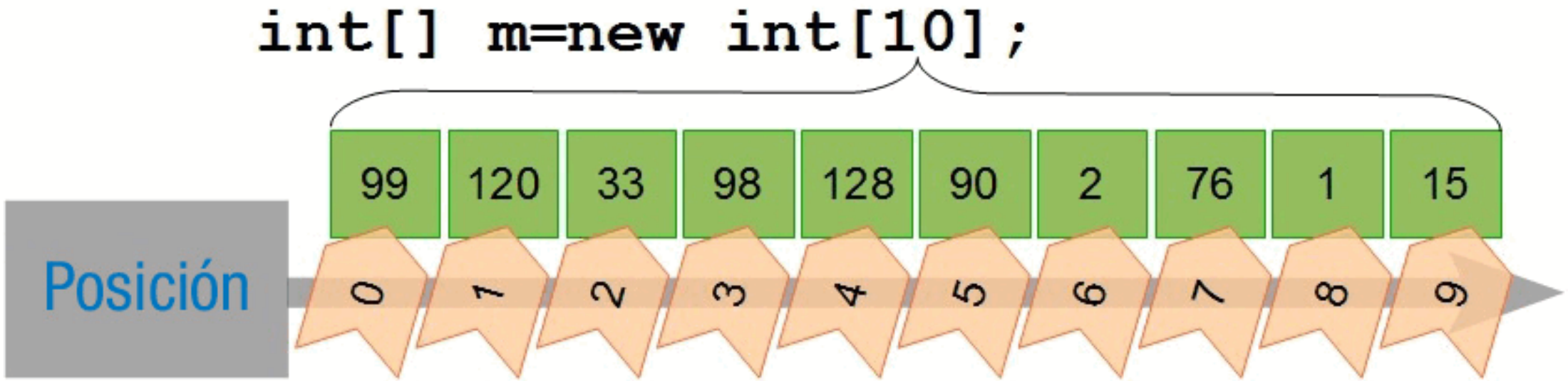
¿Saltar o no saltar? he ahí la cuestión. En la gran mayoría de libros de programación y publicaciones de Internet, siempre se nos recomienda que prescindamos de sentencias de salto incondicional, es más, se desaconseja su uso por provocar una mala estructuración del código y un incremento en la dificultad para el mantenimiento de los mismos. Pero Java incorpora ciertas sentencias o estructuras de salto que es necesario conocer y que pueden sernos útiles en algunas partes de nuestros programas.

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.

Estructuras de salto.



Arrays Unidimensionales.



Arrays Unidimensionales.

Un array es una estructura para guardar un conjunto de objetos de la misma clase. Se accede a cada elemento individual del array mediante un número entero denominado índice (index en inglés). 0 es el índice del primer elemento y n-1 es el índice del último elemento, siendo n, la dimensión del array.

Los arrays permiten almacenar una colección de objetos o datos del mismo tipo. Son muy útiles y su utilización es muy simple:

- **Declaración del array.** La declaración de un array consiste en decir "esto es un array" y sigue la siguiente estructura: "tipo[ ] nombre;". El tipo será un tipo de variable o una clase ya existente, de la cual se quieran almacenar varias unidades.

Arrays Unidimensionales.

- **Creación del array.** La creación de un array consiste en decir el tamaño que tendrá el array, es decir, el número de elementos que contendrá, y se pone de la siguiente forma: "nombre=new tipo[dimension]", donde dimensión es un número entero positivo que indicará el tamaño del array. Una vez creado el array este no podrá cambiar de tamaño.

```
int[] n; // Declaración del array.

n = new int[10]; //Creación del array reservando para el un espacio en memoria.

int[] m=new int[10]; // Declaración y creación en un mismo lugar.
```

Uso de arrays Unidimensionales.

Ya sabes declarar y crear de arrays, pero, ¿cómo y cuando se usan? Pues existen tres ámbitos principalmente donde se pueden usar, y los tres son muy importantes: **modificación** de una posición del array, **acceso** a una posición del array y **paso de parámetros**.

La modificación de una posición del array se realiza con **una simple asignación**. Simplemente **se especifica entre corchetes la posición a modificar después del nombre del array.** Veámoslo con un simple ejemplo:

```
int[] Numeros=new int[3]; // Array de 3 números (posiciones del 0 al 2).

Numeros[0]=99; // Primera posición del array.

Numeros[1]=120; // Segunda posición del array.

Numeros[2]=33; // Tercera y última posición del array.
```

Uso de arrays Unidimensionales.

El acceso a un valor ya existente dentro de una posición del array se consigue de forma similar, simplemente poniendo el nombre del array y la posición a la cual se quiere acceder entre corchetes:

```
int suma=Numeros[0] + Numeros[1] + Numeros[2];
```

```
System.out.println("Longitud del array: "+Numeros.length);
```

Uso de arrays Unidimensionales.

El tercer uso principal de los arrays, como se dijo antes, es en el **paso de parámetros.** Para pasar como argumento un array a una función o método, esta debe tener en su definición un parámetro declarado como array. Esto es simplemente que uno de los parámetros de la función sea un array. Veamos un ejemplo:

```
int sumaarray (int[] j) {

        int suma=0;

        for (int i=0; i<j.length;i++)

            suma=suma+j[i];

        return suma;

}
```

```
int suma=sumaarray (Numeros);
```

Uso de arrays Unidimensionales.

Ya sabes declarar y crear de arrays, pero, ¿cómo y cuando se usan? Pues existen tres ámbitos principalmente donde se pueden usar, y los tres son muy importantes: **modificación** de una posición del array, **acceso** a una posición del array y **paso de parámetros**.

La modificación de una posición del array se realiza con **una simple asignación**. Simplemente **se especifica entre corchetes la posición a modificar después del nombre del array.** Veámoslo con un simple ejemplo:

```
int[] Numeros=new int[3]; // Array de 3 números (posiciones del 0 al 2).

Numeros[0]=99; // Primera posición del array.

Numeros[1]=120; // Segunda posición del array.

Numeros[2]=33; // Tercera y última posición del array.
```

Inicialización.

Rellenar un array, para su primera utilización, es una tarea muy habitual que puede ser rápidamente simplificada. Vamos a explorar dos sistemas que nos van a permitir inicializar un array de forma cómoda y rápida.

En primer lugar, una forma habitual de crear un array y rellenarlo es simplemente a través de un método que lleve a cabo la creación y el rellenado del array. Esto es sencillo desde que podemos hacer que un método retorne un array simplemente indicando en la declaración que el valor retornado es tipo[ ], donde tipo de dato primitivo (int, short, float,...) o una clase existente (String por ejemplo). Veamos un ejemplo:

Inicialización.

```
static int[] ArrayConNumerosConsecutivos (int totalNumeros) {

        int[] r=new int[totalNumeros];

        for (int i=0;i<totalNumeros;i++) r[i]=i;

        return r;

}
```

```
int[] array = {10, 20, 30};

String[] diassemana= {"Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"};
```

Ejercicios