UNIVERSIDAD POLITÉCNICA DE MADRID
ESCUELA UNIVERSITARIA DE INFORMÁTICA
LENGUAJES, PROYECTOS Y SISTEMAS INFORMÁTICOS
Luis Fernández Muñoz
setillo@eui.upm.es

Programación Orientada a Objetos en Java

Tema 2. PROGRAMACIÓN ESTRUCTURADA

ÍNDICE

1. COMENTARIOS

COMENTARIOS :

- Una o varias líneas

  No se permiten anidamientos

```
/* Comentario con
 múltiples líneas */
```

- Hasta el final de la línea

```
// Comentario de línea
```

2. PALABRAS RESERVADAS

Relación de palabras reservada:

abstract	default	goto	package	this
assert	do	if	private	throw
boolean	double	implements	protected	throws
break	else	import	public	transient
byte	enum	instanceof	return	true
case	extends	int	short	try
catch	false	interface	static	void
char	final	long	strictfp	volatile
class	finally	native	super	while
const	float	new	switch	
continue	for	null	synchronized	

Java es un lenguaje "case-sensitive" (diferencia entre mayúsculas y minúsculas => do ≠ Do)

3. TIPOS PRIMITIVOS

DEFINICIÓN: son conjuntos de valores atómicos.

Clasificación de tipos primitivos:

- *Enteros*: representan números enteros positivos y negativos con distintos rangos de valores.

  byte short int long

- *Reales: r*epresentan números reales positivos y negativos con distintos grados de precisión.

  float double

- *Lógicos:* representa un valor lógico (booleano).

  boolean

- *Caracteres*: representan un único carácter Unicode (extensión de ASCII).

  char

3. TIPOS PRIMITIVOS

Conjunto de valores de los tipos primitivos:

Tipo	Descripción	Valor mínimo a máximo.
byte	Entero con signo	-128 a 127
short	Entero con signo	-32768 a 32767
int	Entero con signo	-2147483648 a 2147483647
long	Entero con signo	-92211703685477580 8 a 9221170368547758 07
float	Real de simple precisión	+-3.40282347e+38 a +-1.40239846e-45
double	Real de doble precisión	+-1.79769313486231570e+308 a +-4.94065645841246544e-324
char	Caracteres Unicode	\u0000 a \uFFFF
boolean	Verdadero o falso	true o false

4. LITERALES DE VALORES PRIMITIVOS

DEFINICIÓN: son constantes anónimas que cumplen el formato del tipo primitivo al que pertenecen.

- *Enteros: s*e escriben en base decimal, octal o hexadecimal. Son de tipo **int**

```
Ej.: -21 (decimal) 025 (octal)
     0x15 (hexadecimal)
```

- *Reales*: se escriben con un punto decimal o con notación científica. Son de tipo **double**

```
Ej.: 5.0  .76  -3.14  56.34E-45
```

- *Booleanos*: se escriben mediante dos palabras reservadas. Son de tipo **boolean**

```
Ej.: true false
```

4. LITERALES DE VALORES PRIMITIVOS

• *Caracteres*: se escriben directamente entre comillas simples o mediante su representación Unicode en octal o hexadecimal

```
Ej.: 'f' 'P' '4' '*' ' ' '"'
     '\u0041'(hexadecimal)   '\101'(octal)
```

o mediante la notación de la tabla para los caracteres de escape.

Carácter	Significado	Carácter	Significado
'\t'	Tabulador	'\"'	Carácter comillas dobles
'\n'	Salto de línea	'\''	Carácter comilla simple
'\r'	Cambio de línea	'\\'	Carácter barra hacia atrás

```
Ej.: '\n'
```

5. DECLARACIÓN DE VARIABLES

DEFINICIÓN: son reservas de memoria estática que puede almacenar en cada instante uno de los valores de un cierto tipo primitivo.

```
<tipo> <variable> [ = <expresión[1]> ];
```

- *el nombre de la variable debe ser significativo, sin abreviaturas y debe ser en minúsculas salvo cuando se produce un cambio de palabra, donde la primera letra de la nueva palabra será en mayúscula.*

```
Ej.: int contador;
     boolean aprobadoParcial;
     char simboloNota;
```

[1] *las expresiones se verán más adelante .*

6. DECLARACIÓN DE CONSTANTES

DEFINICIÓN: son reservas de memoria estática que puede almacenar en único valor de un cierto tipo primitivo dado en su inicialización.

```
final <tipo> <variable> = <expresión[1]>;
```

- *el nombre de la constante debe ser significativo, sin abreviaturas y debe ser en mayúsculas separándose con un subrayado cuando se produce un cambio de palabra*

```
Ej.: final int MAXIMO = 100;
     final double PONDERACION = 0.8;
     final char SEPARADOR_FECHAS = '/';
```

[1] *las expresiones para inicializar una constante se verán más adelante .*

7. OPERADORES

DEFINICIÓN: son símbolos que realizan operaciones sobre valores de tipos primitivos, que son suministrados por el valor de las variables, de las constantes o de los literales de un tipo primitivo y devuelven otro valor también de tipo primitivo.

Clasificación según la posición respecto de los operandos:

- *Unarios Prefijos*: preceden a un único operando

```
<símbolo> <operando>
```

- *Unarios Sufijos:* suceden a un único operando

```
<operando> <símbolo>
```

- *Binarios* Infijos: entre los dos operandos

```
<operandoIzq> <símbolo> <operandoDer>
```

7. OPERADORES

Operadores Aritméticos: operan sobre valores numéricos del mismo tipo, y devuelven un valor del tipo de los operandos (menos byte, short y char, que devuelven int) ;

Operador	Uso	Descripción
+	<opI> + <opD>	Devuelve la suma de <opI> y <opD>
-	<opI> – <opD>	Devuelve la resta de <opD> de <opI>
*	<opI> * <opD>	Devuelve el producto de <opI> y <opD>
/	<opI> / <opD>	Devuelve el cociente de <opI> entre <opD>
%	<opI> % <opD>	Devuelve el resto de <opI> entre <opD>
+	+ <op>	Devuelve valor de <op>
-	– <op>	Devuelve el valor opuesto de <op>

7. OPERADORES

Operadores Relacionales: operan sobre dos valores del mismo tipo y devuelven un valor de tipo lógico;

Ope.	Uso	Descripción
>	<opI> > <opD>	Devuelve cierto si *opI* es mayor que *opD*
>=	<op1> >= <op2>	Devuelve cierto si *opI* es mayor o igual que *opD*
<	<op1> < <op2>	Devuelve cierto si *opI* es menor que *opD*
<=	<op1> <= <op2>	Devuelve cierto si *opI* es menor o igual que *opD*
==	<op1> == <op2>	Devuelve cierto si *opI* es igual que *opD*
!=	<op1> != <op2>	Devuelve cierto si *opI* es distinto de *opD*

7. OPERADORES

Operadores Lógicos: operan sobre valores lógicos y devuelven un valor del tipo lógico;

Operador	Uso	Descripción
&&	<opI> && <opD>	Devuelve cierto si ambos operandos son ciertos
||	<opI> || <opD>	Devuelve cierto si algún operando es cierto
!	! <op>	Devuelve el valor negado

8. EXPRESIONES

DEFINICIÓN: es un valor o una combinaciones de operandos y operadores cuya evaluación devuelve un valor de un tipo primitivo.

Reglas de construcción:

- los operandos serán los valores de variables, constantes, literales o el valor devuelto por otro operador;
- cada operador debe acompañarse en tipo, número y posición de operandos según su clase, carácter binario o unario, prefijo o sufijo;

Reglas de evaluación:

- no es recomendable hacer suposiciones sobre el orden de evaluación de los operandos, excepto en los operadores && y || que evalúan primero la parte izquierda;
- cuando un operando está situado entre dos operadores, será operando de aquel operador de mayor nivel de precedencia y, en caso de igualdad, por la asociatividad establecida para ese nivel;

8. EXPRESIONES

Tabla de Precedencia y Asociatividad:

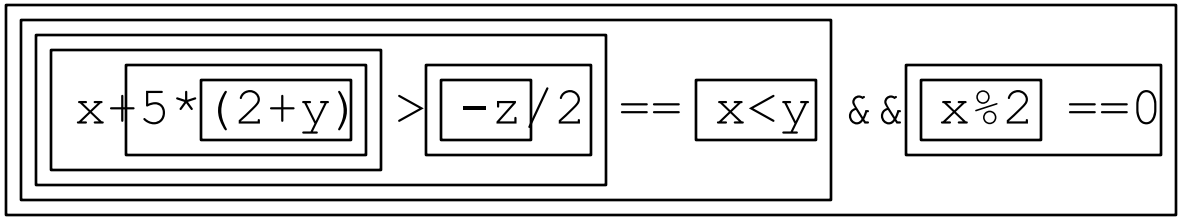
Nivel	Tipos de Operadores	Operadores	Asociatividad
max	Unarios	+ - !	Derecha > Izquierda
	Multiplicativos	* / %	Izquierda > Derecha
	Aditivos	+ -	Izquierda > Derecha
	Relacionales	< > <= >=	No asociativo
	Igualdad	== !=	Izquierda > Derecha
	Y Lógico	&&	Izquierda > Derecha
min	O Lógico	||	Izquierda > Derecha

Los paréntesis rompen las reglas de precedencia y asociatividad

8. EXPRESIONES

```
Ej.: int x = 5;
     int y = 5;
     int z = 5;
     boolean expresion = x+5*(2+y)>-z/2==x<y&&x%2==0;
```

Aplicación de las reglas de precedencia y asociatividad:



8. EXPRESIONES

Conversión de tipos:

• *Promoción:*
transforman un dato de un tipo a otro con el mismo o mayor espacio en memoria para almacenar información;

Origen	Destino
byte	short, int, long, float, double
short	int, long, float, double
char	int[1], long[1], float[1], double[1]
int	long, float, double
long	double
float	double

• *Contracción:*
transforman un dato de un tipo a otro con menor espacio en memoria para almacenar información con la consecuente posible pérdida de información;

Origen	Destino
byte	char
short	byte, char[1]
char	byte[1], short[1]
int	byte, short, char[1]
long	byte, short, char[1], int
float	byte, short, char[1], int, long
double	byte, short, char[1], int, long, float

[1] *código UNICODE.*

8. EXPRESIONES

Conversión de tipos:

- *Conversión implícita*:

  – por promoción cuando se combinan dos operandos de distinto tipo, se convierte el de menor precisión al de mayor precisión;

  – por promoción cuando se asigna un valor de un tipo de menor precisión a una variable de mayor precisión;

- *Conversión explícita*:

  - por promoción o contracción a través del operador de conversión de tipos (*cast*), cuyo nivel de precedencia es el de los operadores unarios*;*

```
(<tipo>) <operando>
```

```
Ej.: float f = (float) 5.5;
     int i = (int) f;
     byte b = (byte) (f/2);
```

9. ASIGNACIÓN

DEFINICIÓN: permite almacenar un valor de un tipo sobre una variable de su mismo tipo.

```
<variable> = <expresión>;
```

```
Ej.: int x = 5;
     int y = 8;
     int intercambio = x;
     x = y;
     y = intercambio;
```

• *la asignación en Java se considera un operador que devuelve el valor asignado. Su nivel de precedencia es el más bajo y su asociatividad es de derecha a izquierda*

9. ASIGNACIÓN

Operadores de Incremento/Decremento: operan sobre una variable de tipo numérico y devuelven un valor del tipo de la variable y su nivel de precedencia es el de los operadores unarios;

Op.	Uso	Descripción
++	<op> ++	Incrementa en 1 el valor de <op> Devuelve el valor de <op> antes de incrementar
++	++ <op>	Incrementa en 1 el valor de <op> Devuelve el valor de <op> después de incrementar
--	<op> --	Decrementa en 1 el valor de <op> Devuelve el valor de <op> antes de decrementar
--	-- <op>	Decrementa en 1 el valor de <op> Devuelve el valor de <op> después de decrementar

```
Ej.: contador++;
```

9. ASIGNACIÓN

Operadores de Acumulación: operan sobre una variable de un tipo numérico y un valor de tipo numérico y devuelven un valor del tipo de la variable; su nivel de precedencia y su asociatividad son los mismos que los del operador de asignación;

Operador	Uso	Equivale a
+=	<opI> += <opD>	<opI> = <opI> + <opD>
-=	<opI> -= <opD>	<opI> = <opI> – <opD>
*=	<opI> *= <opD>	<opI> = <opI> * <opD>
/=	<opI> /= <opD>	<opI> = <opI> / <opD>
%=	<opI> %= <opD>	<opI> = <opI> % <opD>

```
Ej.: int variable = 100;
     variable /= 2;
```

10. VECTORES DE TIPOS PRIMITIVOS

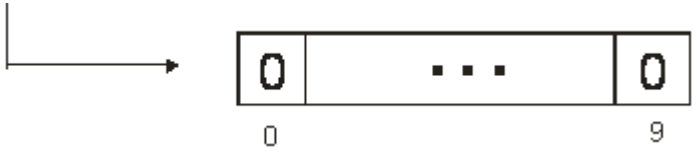
OPERADOR new: es unario prefijo cuyos operandos son vectores y matrices y devuelve la dirección de memoria donde se ha reservado el espacio para dicho vector o matriz; su nivel de precedencia es igual al de los operadores unarios.

Sintaxis a):

```
new <tipo> [ <expresión> ]
```

• donde la expresión debe devolver un valor de tipo entero que será la longitud del vector;

```
Ej.: new int[10];
```



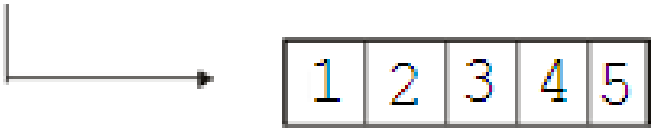
10. VECTORES DE TIPOS PRIMITIVOS

Sintaxis b):

```
new <tipo> [ ] { <expresión1>, … , <expresiónn> }
```

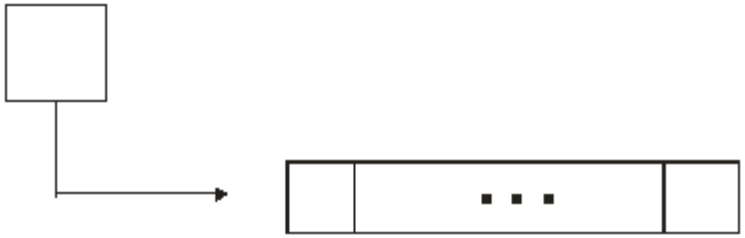
• donde el vector se crea con n elementos inicializados con los valores de la evaluación de las expresiones correspondientes;

```
Ej.: new int[] {1,2,3,4,5};
```



10. VECTORES DE TIPOS PRIMITIVOS

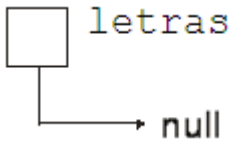
REFERENCIA a un vector de tipos primitivos: es una variable puntero que alberga la dirección del vector.



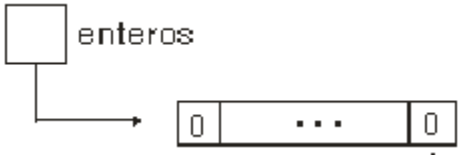
```
<tipo>[ ] <referencia> [ = <dirección> ];
```

- a falta de inicialización su dirección es **null**

```
Ej.: char[] letras;
```




```
Ej.: int[] enteros = new int[10];
```



10. VECTORES DE TIPOS PRIMITIVOS

Operadores:

- *<referenciaV> = <direcciónV>:* asigna la dirección a la referencia siendo del mismo tipo;
- *<direcciónV>**.length***: devuelve la longitud del vector;
- *<direcciónV-I> == <direcciónV-D>*: determina si dos direcciones a vectores del mismo tipo son iguales;
- *<direcciónV-I> != <direcciónV-D>*: determina si dos direcciones a vectores del mismo tipo son distintas;
- *<direcciónV>[<expresión>]*: accede a la variable que ocupa la posición dada por la expresión entera, numeradas de 0 al anterior a la longitud;

10. VECTORES DE TIPOS PRIMITIVOS

```
Ej.: int[] enteros = new int[] {3, 2, 1};
     int valor = enteros.length - 2;
     enteros[0] = enteros[valor] * enteros[2];
     // enteros[0] == 2
     int[] enteros2 = enteros;
     // Las dos variables comparten 3 enteros
     enteros = null;
     enteros2 = null;[1]
     enteros = new int[33];
     boolean prueba = ((enteros == null) ||
                       (enteros == enteros2));
     char[] vocales = [2]{'a','e','i','o','u'};
     vocales = new char[] {'A','E','I','O','U'};
```

[1] *se libera la memoria automáticamente cuando ninguna referencia apunta a un vector*

[2] *en una declaración se puede omitir el tipo del vector*

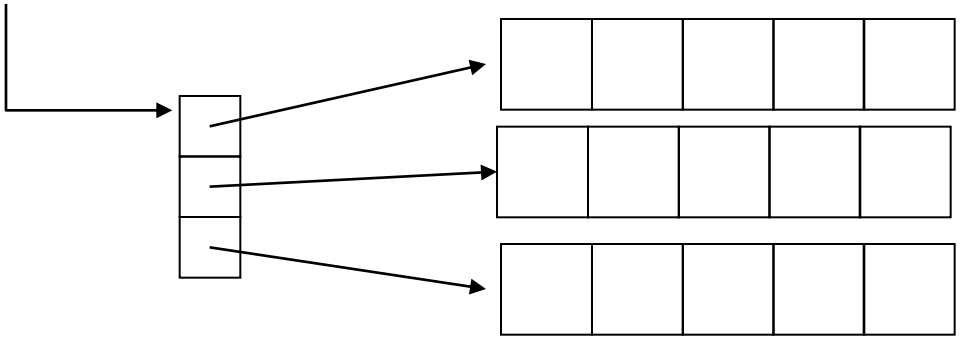
11. MATRICES DE TIPOS PRIMITIVOS

OPERADOR new:

Sintaxis c):

```
new <tipo> [ <expresiónF> ] [ <expresiónC> ]
```

• donde las expresiones F y C deben devolver un valor de tipo entero, que será el número de filas y columnas de la matriz;



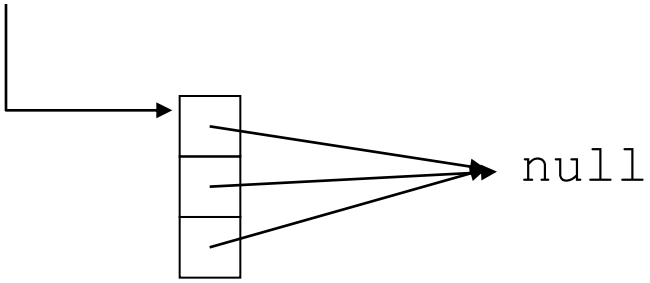
```
Ej.: new int[10][100];
```

11. MATRICES DE TIPOS PRIMITIVOS

Sintaxis d):

```
new <tipo> [ <expresiónF> ] [ ]
```

• donde las expresión F debe devolver un valor de tipo entero que será el número de filas de la matriz;




```
Ej.: new int[10][];
```

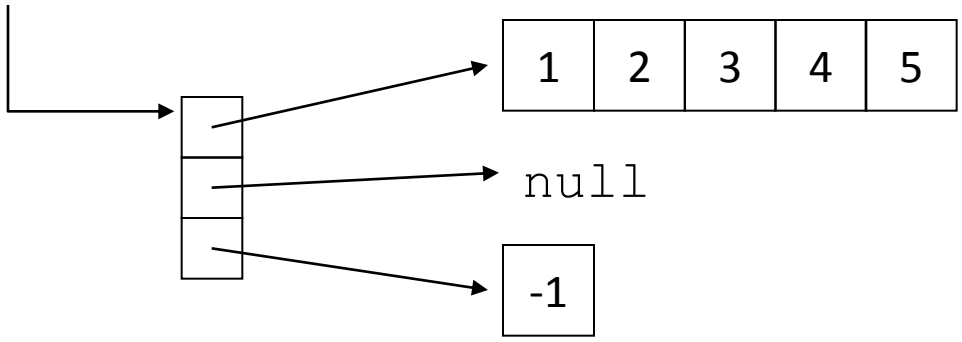
11. MATRICES DE TIPOS PRIMITIVOS

Sintaxis e):

```
new <tipo> [][] { { <expresión11>, … , <expresión1c1> },
                  …
                  { <expresión_f1>, … , <expresión_fcf> }}
```

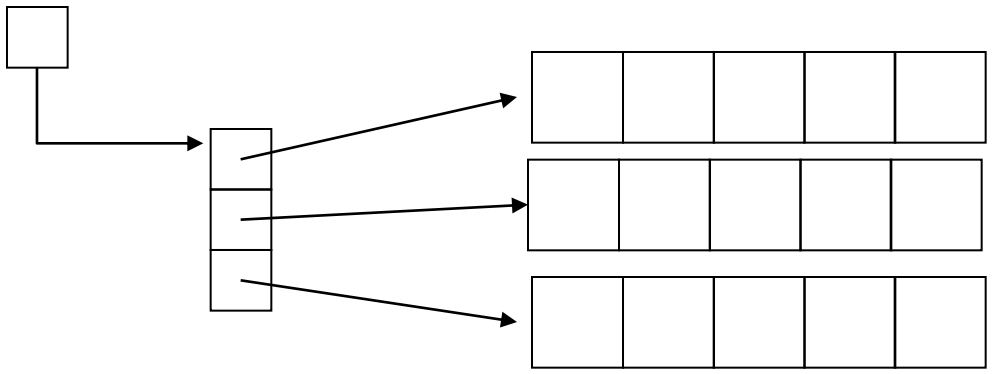
• donde el vector se crea con f filas y c1, … cf columnas inicializados con los valores de la evaluación de las expresiones correspondientes;

```
new int[][] {{1,2,3,4,5}, null, {-1}}
```



11. MATRICES DE TIPOS PRIMITIVOS

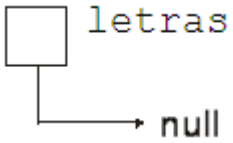
REFERENCIA a una matriz de tipos primitivos: es una variable puntero que alberga la dirección de un vector de referencias a vectores de tipos primitivos.



```
<tipo>[][] <referencia> [ = <dirección> ];
```

- a falta de inicialización su dirección es **null**

```
Ej.: char[][] letras;
```



11. MATRICES DE TIPOS PRIMITIVOS

Operadores:

- *<referenciaM> = <direcciónM>:* asigna la dirección a la referencia siendo del mismo tipo;
- *<direcciónM>**.length***: devuelve el número de filas de la matriz ;
- *<direcciónM-I> == <direcciónM-D>*: determina si dos direcciones a matrices del mismo tipo son iguales;
- *<direcciónM-I> != <direcciónM-D>*: determina si dos direcciones a matrices del mismo tipo son distintas;
- *<direcciónM>[<expresión>]*: accede a la referencia del vector de tipo primitivo que ocupa la posición dada por la expresión entera, numeradas de 0 al anterior a la longitud;

11. MATRICES DE TIPOS PRIMITIVOS

```
Ej.: int[][] matriz = new int[4][5];
     int[][] piramide = new int[3][];
     piramide[0] = new int[7];
     piramide[1] = new int[5];
     piramide[2] = new int[3];
     matriz[0][0] = piramide[2][2];
     matriz = piramide;
     // se pierden los 20 primeros enteros
     matriz[0] = matriz[2];[1]
     matriz[1] = matriz[2];[1]
     /* piramide y matriz mantienen 3 referencias
        a los mismos 3 enteros */
     matriz = null;
     piramide = null;[1]
```

[1] *se libera la memoria automáticamente cuando ninguna referencia apunta a un vector o matriz*

12. SENTENCIAS DE CONTROL DEL FLUJO DE EJECUCIÓN

Sentencia Secuencial:

```
{
   <sentencia1 o declaración1>;
   …
   <sentenciaN o declaraciónN>;
}
```

• *las declaraciones sólo se pueden referenciar dentro de la sentencia a partir de su declaración*

```
Ej.:{
        int x = 2;
        x += 2;
        int y = x + 2;
        y /= x;
    }
```

12. SENTENCIAS DE CONTROL DEL FLUJO DE EJECUCIÓN

Sentencia Condicional Simple:

```
if (<expresión>)
  <sentencia>
[ else
  <sentencia>
]
```

- *la evaluación de la expresión debe devolver un valor de tipo lógico*

```
Ej.: if (divisor != 0) {
        resultado = dividendo / divisor;
     }
```

12. SENTENCIAS DE CONTROL DEL FLUJO DE EJECUCIÓN

Sentencia Condicional Múltiple:

```
switch (<expresión>) {
      case <constante>:
            <sentencia>
            …
            [ break; ]
      …
      [ default:
            <sentencia>
            …
            [ break; ] ]
}
```

- *la evaluación de la expresión debe devolver un valor de tipo entero o carácter*

12. SENTENCIAS DE CONTROL DEL FLUJO DE EJECUCIÓN

```
Ej.: switch (nota) {
       case 0:
       case 1:
       case 2:
       case 3:
       case 4:
         letra = 'I';
         break;
       case 5:
       case 6:
         letra = 'A';
         break;
       …
       …
       case 7:
       case 8:
         letra = 'N';
         break;
       case 9:
       case 10:
         letra = 'S';
         break;
       default:
         letra = ' ';
         break;
}
```

12. SENTENCIAS DE CONTROL DEL FLUJO DE EJECUCIÓN

Sentencia Iterativa Indeterminada 0..N:

```
while (<expresión>)
   <sentencia>
```

- *la evaluación de la expresión debe devolver un valor de tipo lógico;*

```
Ej.: while (numero > 0) {
        numero /= 10;
        digitos++;
     }
```

12. SENTENCIAS DE CONTROL DEL FLUJO DE EJECUCIÓN

Sentencia Iterativa Indeterminada 1..N:

```
do {
   <sentencia>
} while (<expresión>);
```

- *la evaluación de la expresión debe devolver un valor de tipo lógico;*

```
Ej.: do {
        numero -= 10;
     } while (numero > 0);
```

12. SENTENCIAS DE CONTROL DEL FLUJO DE EJECUCIÓN

Sentencia Iterativa Determinada:

```
for (<expresión1>; <expresión2>; <expresión3>)
    <sentencia>
```

- *la expresión$_2$ debe devolver un valor de tipo lógico;*
- *la expresión$_1$ admite la declaración de variables locales que se pueden referenciar dentro del cuerpo del bucle;*

```
Ej.: boolean[] pares = new boolean[10];
     for (int i = 0; i < pares.length; i++) {
        pares[i] = i % 2 == 0;
     }
```