



3 – Elementos básicos del lenguaje

3.1 - INTRODUCCIÓN

Palabras clave en Java

abstract	assert ^(1.4)	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum ^(5.0)	extends	final	finally	float
for	goto*	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp ^(1.2)	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

* Palabras clave no usadas

(X) Palabras añadidas en la versión X de Java

Los términos `null`, `true` y `false` no son palabras clave pero están reservados (no forman parte de la sintaxis pero tienen valor semántico)

Palabras restringidas en Java (desde Java9)

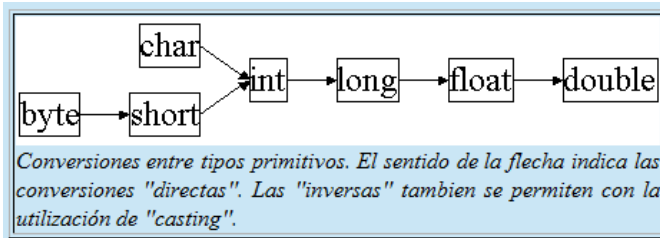
exports, module, open, opens, provides, requires, uses, with, to, transitive

3 – Elementos básicos del lenguaje

3.2 – TIPOS DE DATOS. IDENTIFICADORES Y LITERALES

Tipos PRIMITIVOS (no son objetos. Java es Híbrido)
 Son SIEMPRE IGUALES (no cambian con las plataformas)

Tipos primitivos		
Enteros [complemento a dos con signo]	byte	8 bits
	short	16 bits
	int	32 bits
	long	64 bits
Reales [IEEE 754]	float	32 bits
	double	64 bits
Caracteres [Unicode]	char	16 bits
Booleanos [dpte. implementación]	boolean	
no-tipo		
void		



```
float f;
double g=3.14159;
f=(float)g;

long l=32; //la constante 32 es int y se convierte automáticamente a long
char c=(char)l;
```

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

Tienen sus equivalentes como objetos.
 (Hay otros tipos sólo como objetos, p.ej. Binario, precisión infinita, etc.)

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

        // enteros
        byte maximoByte = Byte.MAX_VALUE;
        short maximoShort = Short.MAX_VALUE;
        int maximoInteger = Integer.MAX_VALUE;
        long maximoLong = Long.MAX_VALUE;

        // reales
        float maximoFloat = Float.MAX_VALUE;
        double maximoDouble = Double.MAX_VALUE;

        // otros tipos primitivos
        char unChar = 'S'; //existe Character
        boolean unBoolean = Boolean.TRUE; //también válido: boolean
        unBoolean=true

        // (aquí continuará la definición de la clase)
    }
}
```

Identificadores

$\{\text{letra}|_|\$\}\{\text{letra}|\text{digito}|_|\$\}^*$

[] ≡ opcional
{a|b} ≡ a ó b
* ≡ repetible (cero o más veces)

letra ::= cualquier carácter de escritura en cualquier idioma.

Digito ::= [0|1|2|3|4|5|6|7|8|9]

Los identificadores no pueden coincidir con palabras reservadas.

Literales

- Entero: $\{[+]|-\}$ [0|x|X|b|B] constante_entera [I|L]

“0” indica constante expresada en octal

“0x” o “0X” indica constante expresada en hexadecimal

“0b” o “0B” indica constante expresada en binario

“l” o “L” indica tamaño “long” (por defecto el tamaño es 32 bits)

constante_entera ::= digito digito*

Ejemplos de literales enteros:

101	+101	
-101		
1011	101L	(101 en 64 bits)
0101		(65 en octal)
0x101	0X101	(257 en hexadecimal)
0b101	0B101	(5 en binario)

- Real: $\{[+]|-\}$ parte_entera . parte_fraccionaria [$\{e|E\}\{[+]|-\}$ exponente] [f|F][d|D]

“f” o “F” indica tamaño “float”

“d” o “D” indica tamaño “double” (tamaño por defecto)

parte_entera, parte_fraccionaria, exponente ::= constante_entera

Ejemplos de literales reales:

37.091	+37.091	37.091D
37.091e12	37.091e+12	+37.091E+012
-37.091e12	-37.091e+12	-37.091E+012
37.091e-12	37.091e-12	+37.091E-012
37.091f	37.091F	←(37.091 en 32 bits)

- Booleano: {true|false}

- Caracteres: UNICODE ejemplos... (char c='x') (char c='\n') (char c='\u001C')

(Estos dos elementos tienen sentido en el “mundo de los objetos”)

- Cadenas: ejemplos... (String s="hola") (String s="\nhola\n")
- Objeto nulo: ejemplo... (String s=null)

<https://docs.oracle.com/javase/6/specs/jls/se7/html/jls-3.html>

Java es sensible a la capitalización, y no pone límites a la longitud de los identificadores.

Sobre estas características se “acuerdan” numerosas convenciones (no las exige el JDK ni los IDEs, pero las siguen los desarrolladores), p.ej. «esto» es un objeto, «Esto» es una clase, «setElement» es una rutina que tiene por función dar valor a un objeto o variable «element», etc.

Arrays. (y algo de syntactic sugar... o no.)

Como cualquier otro lenguaje algorítmico, Java tiene la posibilidad de manejar Arrays, es decir estructuras que almacenan de forma contigua un determinado número de elementos del mismo tipo o clase. Nótese que en los lenguajes clásicos existe otra estructura capaz de almacenar un grupo de elementos heterogéneos (denominada "struct" en C o "record" en Pascal) y que no existe en Java ya que es sustituida y ampliada con el concepto de clase.

Los arrays en Java son en realidad objetos, pero el lenguaje introduce una determinada sintaxis que permite realizar ciertas acciones de un modo natural y/o eficaz ("syntactic sugar")

```
// Declaración estilo C vs. estilo Java
int array1[]; //estilo C (y sólo declarado)
int[] array2=null; //estilo Java (declarado y definido como "No inicializado")

// Generación con tamaño predeterminado en tiempo de compilación
int[] array3=new int[10]; //el array ya existe, su contenido está indeterminado
int[] array4={1,2,3,5,7,11}; //el array existe, y su contenido está determinado

// El tamaño puede determinarse en tiempo de ejecución
array2=new int[2*numeroDeParejas()]; //el array ya existe, su contenido está indeterminado

// Asignacion y lectura
array3[5]=7;
int n=array3[5];

// Arrays multidimensionales
int[][] array2D1;
int[][] array2D2=new int[10][];
int[][] array2D3=new int[10][3];
int[][] array2D4={{1,2,3,4,5,6},{2,4,6},{3,6}};

// Asignaciones en Arrays multidimensionales
array2D2[5]=new int[3]; //generación de una segunda dimensión
array2D2[5][1]=7; //asignación de un valor concreto

// Arrays anónimos (generación en tiempo de ejecución)
array3= new int[116,28,496,8128]; //p.ej. en asignación
```

El campo "length"

```
class EchoParameters {
    public static void main(String[] args)
    {
        for (int i=0; i<args.length;i++)
            System.out.println(args[i]);
    }
}
```

*de la página anterior

La asignación de literales a Strings es también en cierto modo "syntactic sugar" porque nos facilita la generación de objetos como si no lo fuesen, aunque hay un detalle a tener en cuenta:

```
String s1="hola", s2="hola";
```

No son dos objetos String iguales, sino un sólo objeto String referenciado por dos identificadores

Más "syntactic sugar":

```
Double d1=5.0; //boxing
double d2=d1; //unboxing
```

Operador de asignación

Op	Uso	Operación
=	op1 = op2	Asigna el valor de op2 a op1

Operadores aritméticos

Op	Uso	Operación
+	op1 + op2	Suma op1 y op2 (también usado como concatenación de cadenas)
-	op1 - op2	Resta op1 de op2
*	op1 * op2	Multiplca op1 por op2
/	op1 / op2	Divide op1 por op2 (ojo a divisiones enteras/reales)
%	op1 % op2	Obtiene el resto de la división de op1 entre op2
++	++op op++	Incrementa op en 1; evalúa op antes del incremento Incrementa op en 1; evalúa op después del incremento
--	--op op--	decrementa op en 1; evalúa op antes del decremento decrementa op en 1; evalúa op después del incremento
+	+op	"Promueve" op a int si es byte, short, o char
-	-op	Niega op aritméticamente

++op, op++, --op, _op--

Sean x=1 e y=1

(z= ++x + y) resulta en x=2, y=1, z=3

Equivale a {x=x+1; z=x+y}

(z= x++ + y) resulta en x=2, y=1, z=2

Equivale a {z=x+y; x=x+1}

Lo mismo sucede con --op y op--

Operadores relacionales y condicionales

Op	Uso	"true" si...
>	op1 > op2	op1 es mayor que op2
>=	op1 >= op2	op1 es mayor o igual que op2
<	op1 < op2	op1 es menor que op2
<=	op1 <= op2	op1 es menor o igual que op2
==	op1 == op2	op1 es igual que op2
!=	op1 != op2	op1 es distinto que op2
&&	op1 && op2	op1 y op2 son "true" (evalúa op2 condicionalmente)
	op1 op2	op1 o op2 (o ambos) son "true" (evalúa op2 condicionalmente)
!	! op	op es "false"
&	op1 & op2	op1 y op2 son "true"
	op1 op2	op1 o op2 (o ambos) son "true"
^	op1 ^ op2	op1 o op2 (pero no ambos) son "true"

Operadores de desplazamiento y lógicos

Op	Uso	Operación
>>	op1 >> op2	Desplaza los bits de op1 a la derecha en op2 posiciones (desplazamiento aritmético)
<<	op1 << op2	Desplaza los bits de op1 a la izquierda en op2 posiciones
>> >	op1 >>> op2	Desplaza los bits de op1 a la derecha en op2 posiciones (desplazamiento lógico)
&	op1 & op2	"Y" lógico bit a bit
	op1 op2	"O" lógico bit a bit
^	op1 ^ op2	"O exclusivo" lógico bit a bit
~	~op	Complemento bit a bit

Op	Uso	Operación
?:	op1 ? op2 : op3	Si op1 es "true" evalúa op2, si es "false" evalúa op3
[]	type [] type [op] op1 [op2]	Declara un array de longitud indeterminada de elementos de tipo type Crea un array para op elementos de tipo type (usado en combinación con "new") Accede al elemento de índice op2 en el array op1 (el primer índice es 0)
.	op1 . op2	Es una referencia al miembro op2 de op1
()	op (params)	Declara o llama al método op con una lista de parámetros separados por comas que puede ser vacía.
(type)	(type) op	"Amolda" (cast) op al tipo type . Arroja una excepción si op no puede amoldarse
new	new op	Reserva espacio para un nuevo objeto op (se combina con una llamada a un método constructor)
instanceof	op1 instanceof op2	Devuelve el valor "true" si op1 es un objeto de clase op2

op1 ? op2 : op3

Sean x=1 e y=2

Si b=true; (z= b ? x : y) resulta en z=1

Si b=false; (z= b ? x : y) resulta en z=2

b, x, y pueden ser expresiones

(véase ejemplo en sentencia if-then-else)

Operadores de asignación combinados

Op	Uso	Equivalente a...
+=	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
&	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
>>>=	op1 <<<= op2	op1 = op1 <<< op2

Precedencia de operadores

Nivel de precedencia	Nombre	Operador	Asociatividad
16	Paréntesis	()	→
	Acceso a arreglo	[]	
	Acceso a miembros	.	
15	Post-incremento unario	++	←
	Post-decremento unario	--	
14	Pre-incremento unario	++	←
	Pre-decremento unario	--	
	Mas unario	+	
	Menos unario	-	
	Negación lógica unaria	!	
13	Negación/Complemento unario	~	←
	Creación de objetos	new	
12	Multiplicación	*	→
	División	/	
	Módulo	%	
11	Adición	+	→
	Sustracción	-	
	Concatenación de Strings	+	
10	Desplazamiento de bits a izquierda	<<	→
	Desplazamiento aritmético de bits a derecha.	>>	
	Desplazamiento lógico de bits a derecha.	>>>	
9	Menor que	<	→
	Menor o igual que	<=	
	Mayor que	>	
	Mayor o igual que	>=	
	Comparación de tipos	instanceof	
8	Igual a	==	→
	No igual a	!=	
7	Y binario	&	→
6	O exclusivo binario	^	→
5	O inclusivo binario		→
4	Y lógico con evaluación parcial	&&	→
	Y lógico con evaluación completa	&	
3	O lógico con evaluación parcial		→
	O lógico con evaluación completa		
2	Condicional ternario	?:	→
1	Asignación	=	←
	Suma y asignación	+=	
	Sustracción y asignación	-=	
	Multiplicación y asignación	*=	
	División y asignación	/=	
	Módulo y asignación	%=	
	Y binario y asignación	&=	
	O exclusivo binario y asignación	^=	
	O inclusivo binario y asignación	=	
	Desplazamiento de bits a izquierda y asignación	<<=	
	Desplazamiento Aritmético de bits a derecha y asignación	>>=	
Desplazamiento Lógico de bits a derecha y asignación	>>>=		
0	Flecha de expresión lambda	->	→

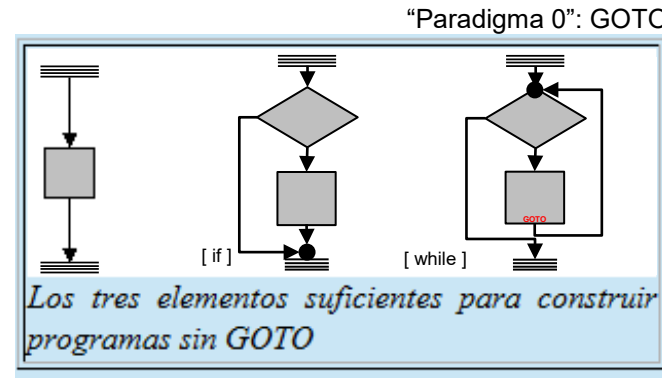
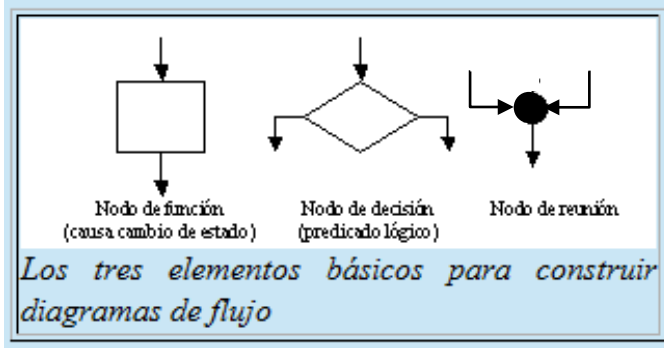
3 – Elementos básicos del lenguaje

3.3 – SENTENCIAS

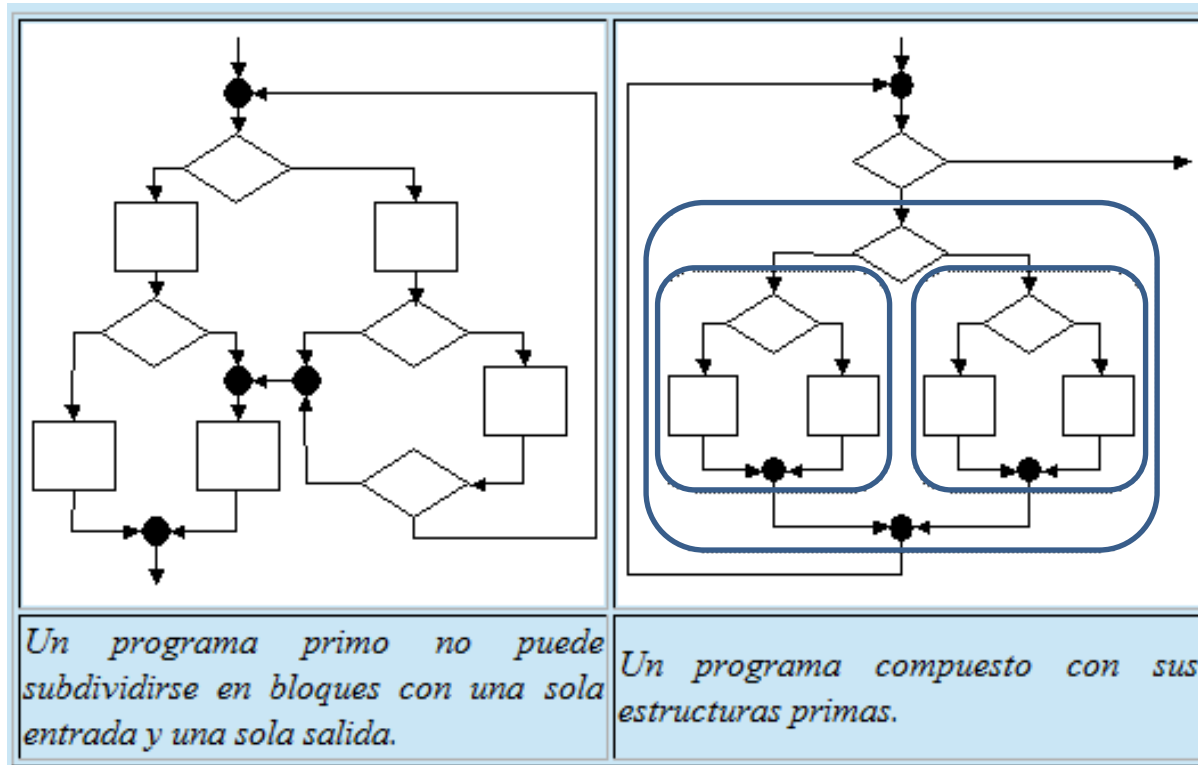
Antes de adentrarnos en la programación orientada a objetos veremos el conjunto de sentencias disponibles. Este es el material resultante del concepto (paradigma) conocido como “programación estructurada”.

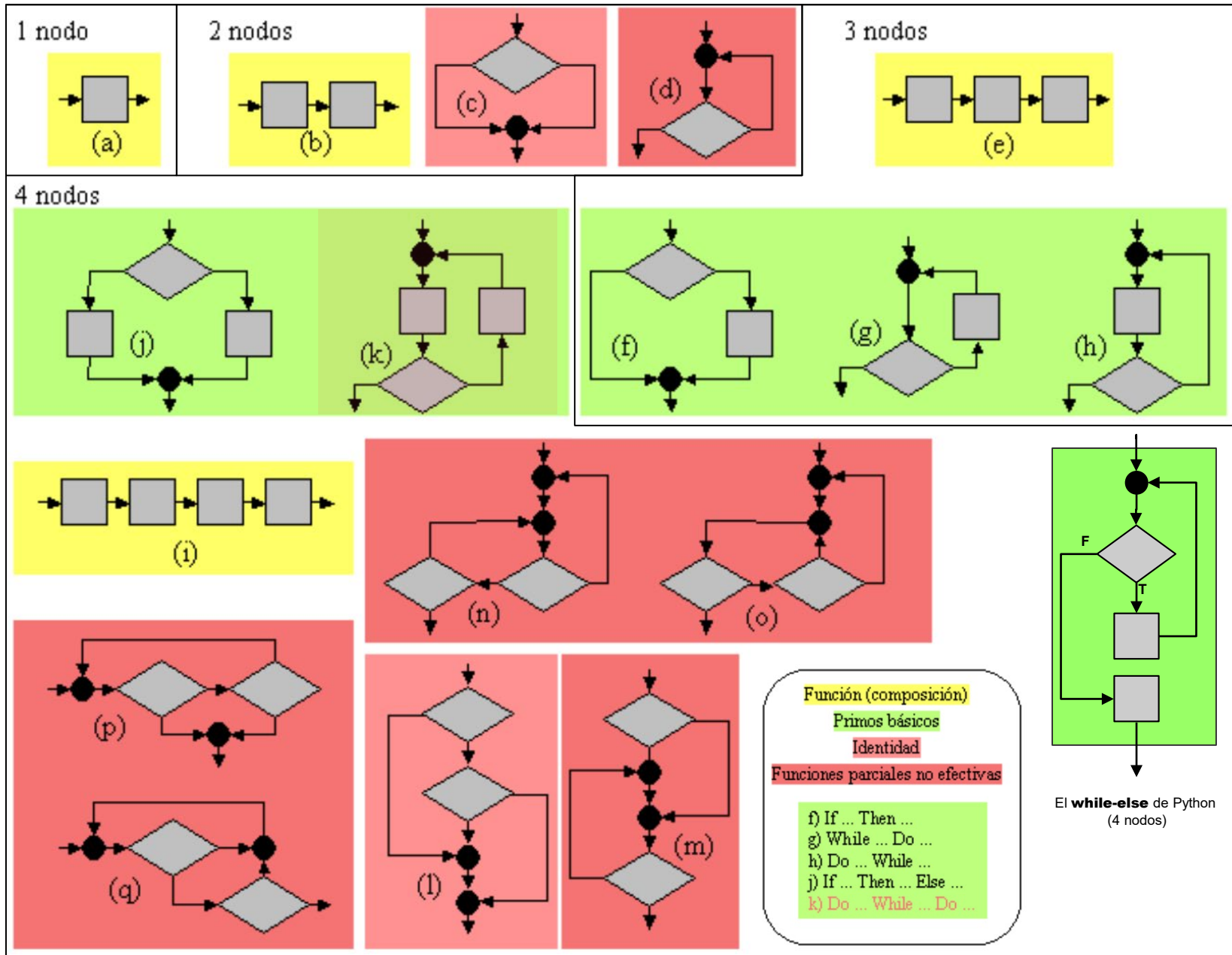
Antes de enumerar dichas sentencias veremos los fundamentos que dan lugar a las mismas

Programación estructurada.



“Paradigma 1”: programación estructurada ([el GOTO es pernicioso](#))





Break:

posibilidad de abortar estructuras. Particularmente ciclos.

En realidad no supone una desestructuración sino la inclusión de determinadas estructuras primas más complejas (con más de 4 nodos)

Return:

posibilidad de abortar rutinas.

En cierto modo es lo mismo, aunque no equivale a aceptar una estructura prima más.

Mecanismo de “excepciones”:

Una generalización de lo anterior que permite “abortos parametrizados”.

No necesariamente ligado a la orientación a objetos, pero típicamente presente en ese paradigma.

Rupturas de secuencia I

```

if (expresion) sentencia;
    
```

If - then

```

if (numeroBoleto==numeroPremiado)
    System.out.println(
        "has obtenido un premio");
    
```

```

if (expresion) sentencia1;
else sentencia2;
    
```

If - then - else

```

if (numeroBoleto==numeroSorteo)
    premio=1000;
else   premio=0;
    
```

Ejemplo tomado de Angel Franco: <http://www.sc.edu/sbweb/fisica/cursoJava/Intro.htm>

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

Utilizando el operador ternario:

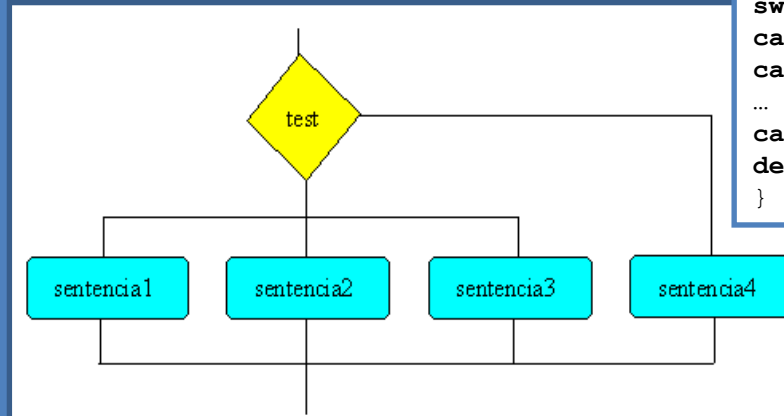
```
premio=(numeroBoleto==numeroPremiado)? 1000 :0;
```

Otro ejemplo. Para calcular un término i-ésimo de la serie :

$$f(x) = \sum_{i=0}^{N-1} (-1)^i g(x_i)$$

```
terminoIesimo=(i%2==0?1:-1) * g(x[i]);
```


Rupturas de secuencia II



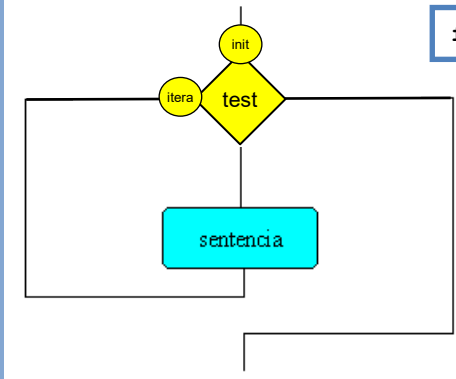
```
switch(expression) {  
  case valor_1: sentencia_1; * break;  
  case valor_2: sentencia_2; * break;  
  ...  
  case valor_N: sentencia_N; * break;  
  default:      sentencia_N+1; *  
}
```

switch

```
switch (mes) {  
  case 1:  
  case 3:  
  case 5:  
  case 7:  
  case 8:  
  case 10:  
  case 12: numDias = 31; break;  
  case 4:  
  case 6:  
  case 9:  
  case 11: numDias = 30; break;  
  case 2: if ( ((año % 4 == 0) && !(año % 100 == 0)) || (año % 400 == 0) )  
          numDias = 29;  
          else numDias = 28;  
          break;  
  default: numdias=0;  
}
```

Generalización:

```
If (expresion==valor_1) sentencia_1  
else if (expresion==valor_2) sentencia_2  
  else  
    ...  
    if (expresion==valor_N) sentencia_N  
    else sentencia_N+1;
```



for (inicialización; condición de mantenimiento; iteración) sentencia

```
for (int i = 0; i < 10; i++) System.out.println(i);
```

```
for (int i=20; i >= 2; i -= 2) System.out.println(i);
```

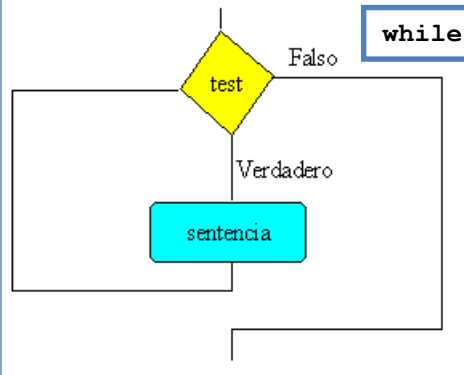
for

Hay otra versión del "for" ligada a colecciones

```
int[] indices={2,3,5,7,11};
```

```
for (int i: indices ) ...
```

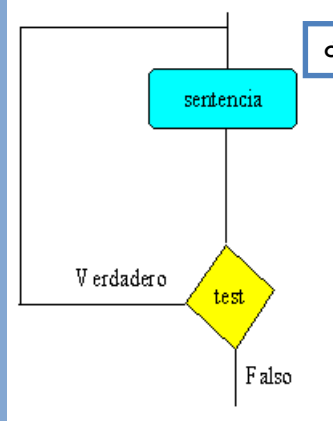
También usable con colecciones de objetos (se verá más adelante)



while (expresión) sentencia

while

```
int i=0;
while (i<10) {
  System.out.println(i);
  i++;
}
```



do sentencia **while** (expresion)

do while

```
int i=0;
do {
  System.out.println(i);
  i++;
} while (i < 10);
```

Ciclos

break, continue y etiquetas

```
for (int i = 0; i < 10; i++) {  
    //...otras sentencias  
    if (condicionFinal) break;  
    //...otras sentencias  
}
```

```
while (true) {  
    //...otras sentencias  
    if (condicionFinal) break;  
    //...otras sentencias  
}
```

```
nivelX:  
for (int i=0; i<20; i++) {  
    //...  
    while (j<70) {  
        //...  
        if (i*j==500) break nivelX;  
        //...  
    }  
    //...  
}
```

```
for (int i = 0; i < 10; i++) {  
    //...otras sentencias  
    if (condicionFinal) continue;  
    //...otras sentencias  
}
```

```
while (true) { //en algún punto un break  
    //...otras sentencias  
    if (condicionFinal) continue;  
    //...otras sentencias  
}
```

```
nivelX:  
for (int i=0; i<20; i++) {  
    //...  
    while (j<70) {  
        //...  
        if (i*j==500) continue nivelX;  
        //...  
    }  
    //...  
}
```

return

```
return ;  
return expresión;
```

(métodos)

```
atributos retorno nombre (parámetros) {  
    // sentencias  
}
```

Parámetros es una lista separada por comas de pares tipo/clase identificador

Ejemplo:

```
public static int suma(int a, int b) {  
    return a+b;  
}
```

Hay otras 2 sentencias:
try y **try-with-resources**
ligadas a objetos...
...por lo que se verán en el siguiente tema

Y una más:
assert
no sólo ligada a objetos sino al modelo de gestión de errores...
...por lo que se verá aún más adelante

```

public class Prueba {
public static void main(String[] args) {
    nivelX:
    for (int i=0; i<10; i++) {
        System.out.print("\nfor "+i+": ");
        int j=0;
        while (j<10) {
            if (i*j==32) break nivelX;
            System.out.print("("+i+"."+j+" ");
            j++; }
        System.out.println("for end"); }
    }
}

```

break con etiqueta

```
C:\>java Prueba
```

```

for 0: (0.0) (0.1) (0.2) (0.3) (0.4) (0.5) (0.6) (0.7) (0.8) (0.9) for end
for 1: (1.0) (1.1) (1.2) (1.3) (1.4) (1.5) (1.6) (1.7) (1.8) (1.9) for end
for 2: (2.0) (2.1) (2.2) (2.3) (2.4) (2.5) (2.6) (2.7) (2.8) (2.9) for end
for 3: (3.0) (3.1) (3.2) (3.3) (3.4) (3.5) (3.6) (3.7) (3.8) (3.9) for end
for 4: (4.0) (4.1) (4.2) (4.3) (4.4) (4.5) (4.6) (4.7)C:\>

```

```

public class Prueba {
public static void main(String[] args) {
    nivelX:
    for (int i=0; i<10; i++) {
        System.out.print("for "+i+": ");
        int j=0;
        while (j<10) {
            if (i*j==32) continue nivelX;
            System.out.print("("+i+"."+j+" ");
            j++; }
        System.out.println("for end"); }
    }
}

```

continue con etiqueta

```
C:\>java Prueba
```

```

for 0: (0.0) (0.1) (0.2) (0.3) (0.4) (0.5) (0.6) (0.7) (0.8) (0.9) for end
for 1: (1.0) (1.1) (1.2) (1.3) (1.4) (1.5) (1.6) (1.7) (1.8) (1.9) for end
for 2: (2.0) (2.1) (2.2) (2.3) (2.4) (2.5) (2.6) (2.7) (2.8) (2.9) for end
for 3: (3.0) (3.1) (3.2) (3.3) (3.4) (3.5) (3.6) (3.7) (3.8) (3.9) for end
for 4: (4.0) (4.1) (4.2) (4.3) (4.4) (4.5) (4.6) (4.7) for 5: (5.0) (5.1) (5.2) (5.3) (5.4) (5.5) (5.6) (5.7) (5.8) (5.9) for end
for 6: (6.0) (6.1) (6.2) (6.3) (6.4) (6.5) (6.6) (6.7) (6.8) (6.9) for end
for 7: (7.0) (7.1) (7.2) (7.3) (7.4) (7.5) (7.6) (7.7) (7.8) (7.9) for end
for 8: (8.0) (8.1) (8.2) (8.3) for 9: (9.0) (9.1) (9.2) (9.3) (9.4) (9.5) (9.6) (9.7) (9.8) (9.9) for end
C:\>

```

El while-else de Python en Java

El while-else de Python no tiene nada de especial a no ser que se produzca un break dentro del ciclo, en cuyo caso no se ejecuta la sentencia afectada por el else. Veámos cómo hacer esto con Java

Versión clásica:

```
boolean abortado=false;
while( <condición> ) {
    // sentencias...
    if (<se_da_condición_para_abortar>) {
        abortado=true;
        break;
    }
    // sentencias...
}
if (!abortado) // acción tras recorrer todos los elementos;
```

Una versión algo más interesante:

```
whileAndThen:{
while( hay_más_elementos_a_comprobar ){
    // sentencias...
    if (<se_da_condición_para_abortar>) break whileAndThen;
    // sentencias ...
}
// acción correspondiente al else de Python;
}
```

* He llamado **whileAndThen** a la etiqueta, y no **whileElse**, porque la palabra "else" de Python no es muy afortunada (tiene su lógica "interna", pero no es nada clara)

