ELEMENTOS DEL LENGUAJE

1. SINTAXIS BÁSICA

Vamos a comentar algunos aspectos sintácticos generales:

- **Lenguaje sensible a mayúsculas/minúsculas.** El compilador Java hace distinción entre mayúsculas y minúsculas, no es lo mismo escribir *public* que Public, de hecho, utilizar la segunda forma en vez de la primera provocaría un error de compilación al no reconocer public como palabra reservada. La distinción entre mayúsculas y minúsculas no sólo se aplica a las palabras reservadas del lenguaje, también a los nombres de variables y métodos.

- **Las sentencias finalizan con "`;`".** Ya lo hemos visto en los ejemplos anteriores, toda sentencia Java debe terminar con el carácter ";".

- **Los bloques de instrucciones se delimitan con llaves `{}`.** Esto también lo hemos podido comprobar en los ejemplos anteriores, donde el contenido de una clase y el código de un método se delimitaban mediante dichos símbolos.

- **Comentarios de una línea y multilínea.** En Java, un comentario de una línea va precedido por "`//`" mientras que los que ocupan varias líneas se delimitan por "`/*`" y "`*/`" (figura 1).

```
//comentario de una línea
/*comentario de
   varias líneas*/
```

Fig. 1 - Comentarios en un programa Java.

2. SECUENCIAS DE ESCAPE

Hay determinados caracteres en Java que, o bien no tienen una representación explícita, o bien no pueden ser utilizados directamente por el hecho de tener un significado especial para el lenguaje. Para poder utilizar estos caracteres dentro de un programa Java se utilizan las secuencias de escape.

Una secuencia de escape está formada por el carácter "****" seguido de una letra, en el caso de ciertos caracteres no imprimibles, o del carácter especial. La **tabla** de la figura 2 contiene las principales secuencias de escape predefinidas de Java.

Secuencia de escape	***Significado***
\b	**Retroceso**
\n	**Salto de línea**
\t	**Tabulación horizontal**
****	**Barra invertida **
\’	**Comilla simple**
\”	**Comilla doble**

Fig. 2 - Secuencias de escape

Por ejemplo si quisiéramos mostrar:

Java *“mañana”*

VB *“tarde”*

utilizaríamos la instrucción:

```
System.out.println (“Java\t\”mañana\”\nVB\t\”tarde\””);
```

3. TIPOS DE DATOS PRIMITIVOS

Toda la información que se maneja en un programa Java puede estar representada, bien por un objeto o bien por un *dato básico o de tipo primitivo.* Java soporta ocho tipos de datos primitivos que se indican en la figura 3.

Entre los ocho tipos de datos primitivos no encontramos ninguno que represente una cadena de caracteres, el motivo es que en Java éstas se tratan mediante objetos, concretamente, objetos de la clase **String.** Más adelante abordaremos el tratamiento de cadenas.

Tipo básico	*Tamaño*
byte	**8 bits**
short	***16 bits***
int	**32 bits**
long	**64 bits**
char	**16 bits**
float	**32 bits**
double	**64 bits**
boolean	**-**

Fig. 3 - Tipos primitivos Java

Los tipos de datos primitivos se pueden organizar en cuatro grupos:

- **Numéricos enteros.** Son los tipos *byte, short, int* y *long*. Los cuatro representan números enteros con signo.
- **Carácter.** El tipo *char* representa un carácter codificado en el sistema unicode.
- **Numérico decimal.** Los tipos *float* y *double* representan números decimales en coma flotante.
- **Lógicos.** El tipo *boolean* es el tipo de dato lógico, los dos únicos posibles valores que puede representar un dato lógico son *true* y *false*. *true y false* son palabras reservadas de Java, a diferencia de otros lenguajes, no existe una equivalencia entre estos valores y los números enteros. En cuanto al tamaño en número de bits del tipo *boolean*, éste es dependiente de la máquina virtual.

4. VARIABLES

Una variable es un espacio físico de memoria donde un programa puede almacenar un dato para su posterior utilización.

Tipos de datos de una variable

En Java las variables pueden utilizarse para tratar los dos tipos de datos indicados anteriormente.

- **Tipos primitivos.** Las variables que se declaran de un tipo primitivo almacenan el dato en sí, tal y como se indica en la figura 4.

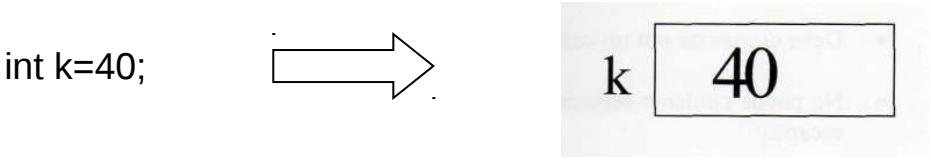
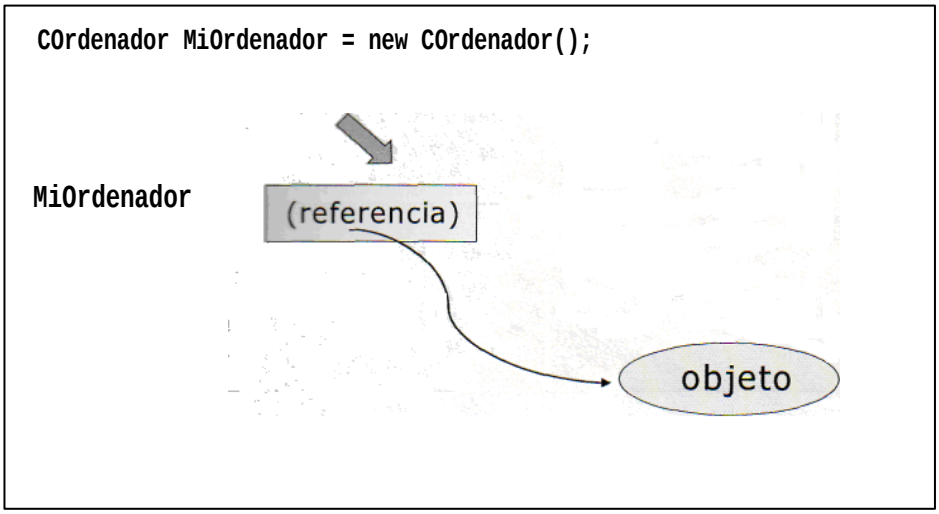


Fig. 4 - Variable conteniendo un tipo primitivo

- **Tipo objeto.** Los objetos en Java también se tratan a través de variables, sólo que, a diferencia de los tipos primitivos, una variable de tipo objeto no contiene al objeto como tal sino una referencia al mismo (figura **5).** No debe importarnos el valor exacto contenido en la variable, tan solo tenemos que saber que a través del operador "**.**" podemos acceder con ella a los métodos del objeto referenciado, utilizando la expresión:

```
variable_objeto.metodo()
```



Fíg;. 5 - Variable conteniendo una referencia a un objeto

Declaración de variables

Como se desprende de los anteriores ejemplos, una variable se declara de la forma:

```
tipo_dato nombre_variable;
```

Un nombre de variable válido debe cumplir las siguientes reglas:

- Debe comenzar por un carácter alfabético.
- No puede contener espacios, signos de puntuación o secuencias de escape.
- No puede utilizarse como nombre de variable una palabra reservada a Java.

También es posible declarar en una misma instrucción varias variables de igual tipo:

```
tipo variable1, variable2, variable3;
```

La figura 6 muestra algunos ejemplos de declaraciones de variables válidas y no válidas.

Válidas	No válidas
`int k, cod;`	`boolean 7q; //comienza por un número`
`long p1;`	`int  num uno; //contiene un espacio`
`char cad_2;`	`long class; //utiliza una palabra reservada`

Fig. 6 Ejemplos de declaraciones de variables

Asignación

Una vez declarada una variable se le puede asignar un valor siguiendo el formato:

variable=expresión;

dónde expresión puede ser cualquier expresión Java que devuelva un valor acorde con el tipo de datos de la variable:

```
int p, k, v;

p = 30;

k = p + 20;

v = k * p;
```

También es posible asignar un valor inicial a una variable en la misma instrucción de declaración:

```
int num = 5; //declara la variable y la inicializa

int p, n = 7; //declara dos variables y sólo inicializa
              //la segunda
```

Literales

Un literal es un valor constante que se puede asignar directamente a una variable o puede ser utilizado en una expresión.

Existen cuatro tipos de literales básicos, coincidiendo con los cuatro grupos en los que se pueden dividir los tipos básicos Java, esto es, numéricos enteros, numéricos decimales, lógicos y carácter. A la hora de utilizar estos literales en una expresión hay que tener en cuenta lo siguiente:

- **Los literales numéricos enteros se consideran como tipo int.** Dado que todo literal entero es un tipo int, una operación como:

```
byte b = 10;
```

intentaría asignar un número de tipo int a una variable de tipo byte, lo que a priori podría provocar un error de compilación. Sin embargo, como veremos más adelante, en expresiones de ese tipo Java realiza una conversión implícita del dato al tipo destino, siempre y cuando el dato "quepa" en la variable.

- **Los literales numéricos decimales se consideran como *tipo* double.** Una asignación de tipo:

```
float p = 3.14;
```

provoca un error de compilación al intentar asignar un dato double a una variable float que tiene un tamaño menor. Para evitar el error, se debe utilizar la letra "f' a continuación del número:

```
float p = 3.14f;
```

lo que provoca una conversión del número *double a float.*

- **Los literales boolean son *true yfalse.*** Estas palabras reservadas no tienen equivalencia numérica, por lo que la siguiente instrucción provocará un error de compilación de incompatibilidad de tipos:

```
boolean b = 0;
```

- **Los literales de tipo char se escriben entre comillas simples.** Se puede utilizar la representación del carácter o su valor unicode en hexadecimal, precedido de la secuencia de escape \u:

```
char car = '#';
char p = '\u03AF';
```

Dado que un carácter es realmente un número entero, también puede asignarse directamente a una variable char el literal entero correspondiente a la combinación unicode del carácter:

```
char c = 231; //se almacena el carácter cuyo
              //código unicode es 231
```

Ámbito de las variables

Según dónde esté declarada una variable, ésta puede ser:

- **Campo o atributo.** Se les llama así a las variables que se declaran al principio de la clase (figura 7), fuera de los métodos. Estas variables son compartidas por todos los métodos de la clase y, suelen declararse como *private* para limitar su uso al interior de la clase, si bien en algunas ocasiones es necesario asignarles un modificador de acceso

que permita su utilización desde el exterior de la misma. Las **variables atributo pueden ser utilizadas sin haber sido inicializadas de manera explícita,** ya que se inicializan implícitamente cuando se crea un objeto de la clase.

- **Variable local.** Son variables que se declaran dentro de un método, su ámbito de utilización está restringido al interior del método y no admiten ningún tipo de modificador (figura 7). Una variable local se crea en el momento en que se hace la llamada al método, destruyéndose cuando finaliza la ejecución de éste (se dice que la variable sale del ámbito). Una variable local también puede estar declarada en el interior de un bloque de instrucciones, sólo que, en este caso, su uso está restringido al interior de ese bloque. **Toda variable local tiene que ser inicializada explícitamente antes de ser utilizada.**

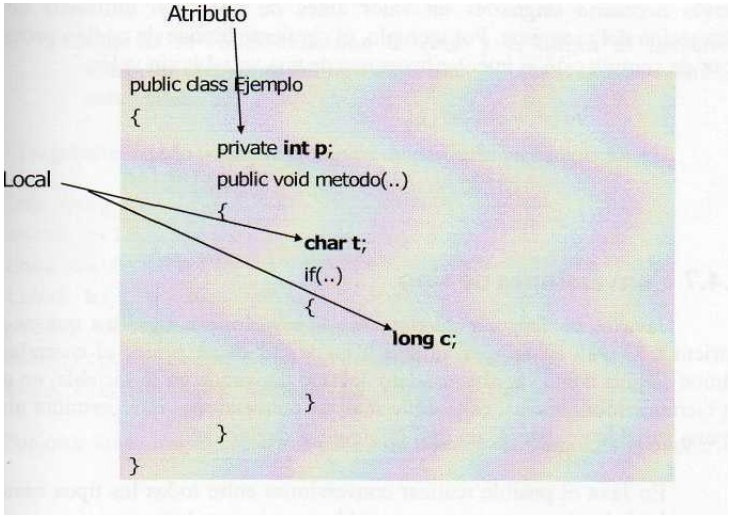


Fig. 7 - Lugares en los cuales se puede declarar una variable en Java

Valores por defecto de una variable

Como ya se ha mencionado anteriormente, las variables de tipo atributo son inicializadas implícitamente antes de su utilización. El valor que toma la variable cuando ésta es inicializada de forma automática, conocido como valor por defecto o predeterminado, depende del tipo que se ha declarado la variable.

La tabla de la figura 8 representa los valores de inicialización de variable según su tipo.

Tipo de *variables*	*Valor por defecto*
byte, short int, long	***0***
char	**'\u0000'**
float, double	**0.0**
boolean	**False**
objeto	**Null**

Fig. 8 - Valores a los que se inicializa una variable implícitamente

Las variables locales en cambio no son inicializadas de forma implícita, siendo necesario asignarles un valor antes de que sean utilizadas en alguna instrucción del

programa. Por ejemplo, el siguiente bloque de código provocará un error de compilación al intentar hacer uso de una variable sin valor:

```
void método()
{
      int n;
      n = n + 1; //error de compilación
}
```

Conversiones de tipo

Java es un lenguaje fuertemente tipado, lo que significa que es bastante estricto a la hora de asignar valores a las variables. A priori, el compilador sólo admite asignar a una variable un dato del tipo declarado en la variable, no obstante. en ciertas circunstancias, es posible realizar conversiones que permitan almacenar en una variable un dato de un tipo diferente al declarado.

En Java es posible realizar conversiones entre todos los tipos básicos, excepción de boolean, que es incompatible con el resto de tipos.

Las conversiones de tipo pueden realizarse de dos maneras: **implícitamente o explícitamente.**

CONVERSIONES IMPLÍCITAS

Las conversiones implícitas se realizan de manera automática, es decir, el valor o expresión que va a asignar a la variable es convertido automáticamente al tipo de ésta por el compilador, antes de almacenarlo en la variable. El siguiente código es un ejemplo de conversión implícita:

```
int i;byte b = 30;
i = b;
```

En este ejemplo, el dato de tipo *byte* almacenado en la variable b es convertido en un *int* antes de asignarlo a la variable b.

Para que una conversión pueda realizarse de forma automática (implícitamente), **el tipo de la variable destino debe ser de tamaño igual o superior al tipo de origen,** si bien esta regla tiene dos excepciones:

- Cuando la variable destino es entera y el origen es decimal *(float* o *double)*, la conversión no podrá ser automática.

- Cuando la variable destino es *char* y el origen es numérico, independientemente del tipo específico, la conversión no podrá ser automática.

El siguiente listado contiene ejemplos de conversiones implícitas:

```
int k = 5, p;short s = 10;
char c = 'ñ';
float h;
p = c; //conversión implícita char a int
h = k; //conversión implícita int a float
k = s; //conversión implícita short a int
```

Por otro lado, los siguientes intentos de conversión implícita provocarían un error:

```
int n;long c = 20;
float ft = 2.4f;
char k;
byte s = 4;
n = c; //error, no se puede convertir implícitamente
       //long a int
k = s; //error, no se puede convertir implícitamente
       // byte a char
n = ft; //error, no se puede convertir implícitamente
        //float a int
```

CONVERSIONES EXPLÍCITAS

Cuando no se cumplan las condiciones para una conversión implícita, ésta podrá realizarse explícitamente utilizando la expresión:

```
variable_destino = (tipo_destino) dato_origen;
```

Con esta expresión le estamos diciendo al compilador que convierta ***dato_origen*** a ***tipo_destino*** para que pueda ser almacenado en ***variable_destino***.

A esta operación se la conoce como ***casting*** o estrechamiento ya que al convertir un dato de un tipo en otro tipo de tamaño inferior se realiza un estrechamiento que, en algunos casos, puede provocar una pérdida de datos, aunque ello no provocará errores de ejecución. Los siguientes son ejemplos de conversiones explícitas:

```
char c;
byte k;
int p = 400;
double d = 34.6;
c = (char)d; //se elimina la parte decimal (truncado)
k = (byte)p; //se produce una pérdida de datos
             //pero la conversión es posible
```

En el caso de los objetos no es posible realizar conversiones entre los mismos, sin embargo, es posible asignar un objeto de una clase a una variable de clase diferente. Para que esto sea posible es necesario que exista una **relación de herencia** entre las clases, por lo que este asunto se tratará en el capítulo dedicado a la Programación Orientada a Objetos.

Constantes

Una constante es una variable cuyo valor no puede ser modificado. Para definir una constante en Java se utiliza la *palabra final*, delante de la declaración del tipo, siguiendo la expresión:

```
final tipo nombre_cte = valor;
```

Por ejemplo:

```
final double pi = 3.1416;
```

Una constante se define en los mismos lugares en los que se puede declarar una variable: al principio de la clase y en el interior de un método.

Suele ser bastante habitual que derterminadas constantes tengan que poder ser utilizadas desde el exterior de la clase donde se han declarado; para que además no sea necesario crear objetos de la misma para hacer uso de estas constantes, se declaran como campos públicos y estáticos. Éste es el caso de la constante **PI,** declarada en la clase **Math** de Java estándar de la forma:

```
public static final double PI = 3.1416;
```

5. OPERADORES

Los operadores son símbolos que permiten realizar operaciones con los datos de un programa. Java dispone de una amplia variedad de operadores, a continuación estudiaremos los más importantes, clasificándolos según el tipo de operación que realizan.

Aritméticos

Se utilizan para realizar operaciones aritméticas dentro de un programa, actuando sobre valores de tipo numérico.

El cuadro de la figura 9 muestra los operadores aritméticos más importantes, incluyendo una breve descripción de su función y un ejemplo de utilización.

Sobre el operador +, hay que decir que, además de sumar números, se puede utilizar también para la concatenación o unión de cadenas:

```
System.out.println("Hola " + "adiós "); //mostraría Hola adiós
```

También puede utilizarse para unir un texto con el contenido de una variable que no sea de tipo texto, puesto que **Java convierte automáticamente a texto los operandos que no sean de tipo cadena** antes de realizar la concatenación. El siguiente programa muestra en pantalla el cuadrado de un número cualquiera almacenado en una variable:

```
public class Cuadrado{
   public static void main(String []args)
   {
      int r, n = 5;
      r = n * n;
      System.out.println("El cuadrado de " + n + " es " + r);
   }
}
```

Al ejecutar el programa anterior se mostrará la frase:

El cuadrado de 5 es 25

Operador	Descripción	Ejemplo
+	Suma dos valores numéricos	int c; c=2+5;
-	Resta dos valores numéricos.	Int c; c=2-5;
*****	Multiplica dos números	Int c; c=2*5;

/	Divide dos números. El tipo de resultado depende de los operandos, pues en **el** caso de que ambos sean enteros, el resultado de la división siempre será entero.	int a=8,b=5;
		float x, y;
		x=a/b; // El resultado es 1
		y=a/3.0f; //El resultado es 2.66

%	Calcula el resto de la división entre dos números.	int c; c=7%2; //El resultado es 1
++	Incrementa una variable numérica en una unidad y deposita el resultado en la variable.	int c=3 c++; //Equivale a c=c+1
--	Decrementa una variable en una unidad y deposita el resultado en la variable.	int c=3 c--; //Equivale a c-c-1

Fig. 9- Operadores aritméticos

En cuanto a los operadores de incremento (++) y decremento (--), hay que tener cierta precaución al utilizarlos dado que es posible situarlos delante de la variable (prefijo), o después de la variable (posfijo), lo que puede condicionar el resultado final de una expresión. Veamos el siguiente ejemplo (los números de línea se indican a efectos de clarificar la explicación):

```
1   int a = 3, b, s;
2   b = a++;
3   System.out.println{"b vale " + b); //b vale 3
4   s = ++b * ++a;
5   System.out.println("s vale " + s); //s vale 20
```

En la línea 2, el operador de incremento está colocado después de la variable, esto implica que primero se asignará el valor actual de la variable "a" a la variable “b”,' realizando posteriormente el incremento de "a". Por otro lado, en la línea 4, dado que el operador está situado delante de la variable, se realizará primero el incremento de las variables *"a"* y *"b" y* después se multiplicarán sus contenidos.

Como norma general, al utilizar operadores de incremento y decremento en una expresión compleja, debemos tener en cuenta los siguientes puntos:

- Las expresiones Java se evalúan de izquierda a derecha.
- Si el operador se sitúa delante de la variable, se ejecutará primeramente la operación que representa antes de seguir evaluando la expresión.
- Si el operador se sitúa después de la variable, se utiliza el valor actual de la variable para evaluar la expresión y a continuación se ejecuta la operación asociada al operador.

Asignación

Además del clásico operador de asignación (=), Java dispone de un conjunto de operadores que permiten simplificar el proceso de operar con una variable y asignar el resultado de la operación a la misma variable.

La tabla de la figura 10 muestra estos operadores y la función que realiza

Operador	*Descripción*	*Ejemplo*
=	Asigna la expresión de la derecha, al operando situado a la izquierda del operador.	int c; c=8*5; //Asigna el resultado de la operación a la variable c
+=	Suma la expresión de la derecha, a la variable situada a la izquierda del operador.	intc=4; c+=5; //Equivale a realizar c=c+5
-=	Resta la expresión de la derecha a la variable situada a la izquierda del operador.	int c=3; c-=2; //Equivalea realizar c=c-2
***=**	Multiplica la expresión de la derecha con la variable y deposita el resultado en la variable.	int a=5 a* =2; //Equivale a realizar a=a*2
/=	Divide la variable situada a la izquierda entre la expresión de la derecha, depositando el resultado en la variable.	int c=7; c/=2; //Equivale a realizar c=c/2
%=	Calcula el resto de la división entre la variable situada a la izquierda y la expresión de la derecha, depositando el resultado en la variable.	intc=6; c%=3; //Equivale a realizar c=c %3

Fig. 10 - Operadores de asignación

ASIGNACIÓN DE REFERENCIAS Y ASIGNACIÓN DE VALORES

Es importante en este punto destacar la diferencia existente entre realizar una operación de asignación entre variables de tipo objeto y realizar la misma operación entre variables de tipo básico.

En el caso de un tipo básico, por ejemplo un entero, esta operación implica que el dato contenido en una variable se copia en la otra. Esto mismo sucede si la variable es de tipo objeto, ahora bien, en este caso debemos saber que **lo que** se **está copiando es la referencia al objeto, no el objeto.** Por tanto, nos encontramos con que **no tenemos dos copias del objeto, sino un único objeto referenciado por dos variables** (figura 11).

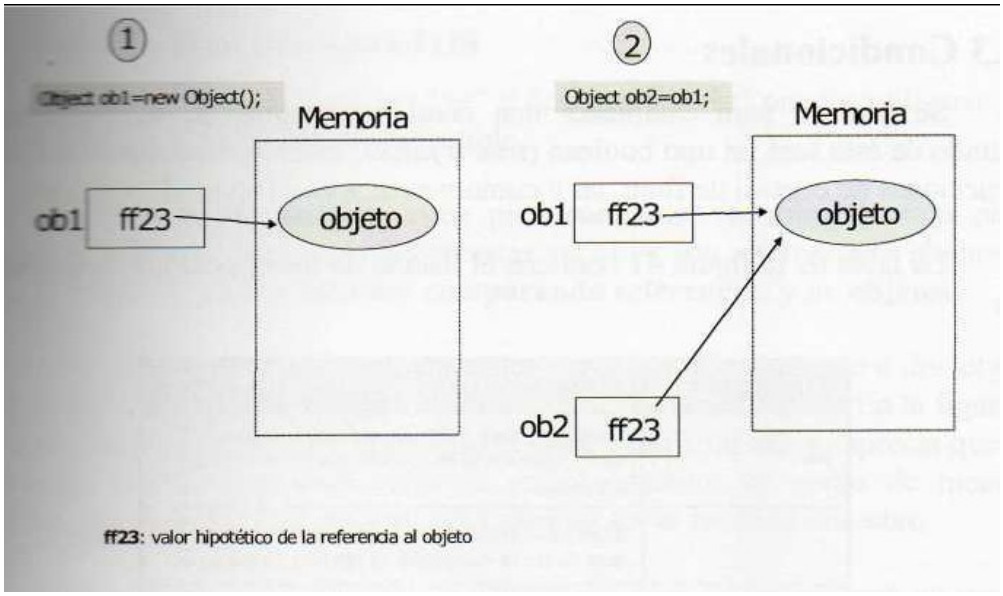


Fig. 11 - Asignación de una variable objeto a otra

Como ejemplo concreto, en la figura 12 podemos comparar el efecto producido al realizar una asignación entre variables enteras con el que tiene lugar en la asignación entre variables String.

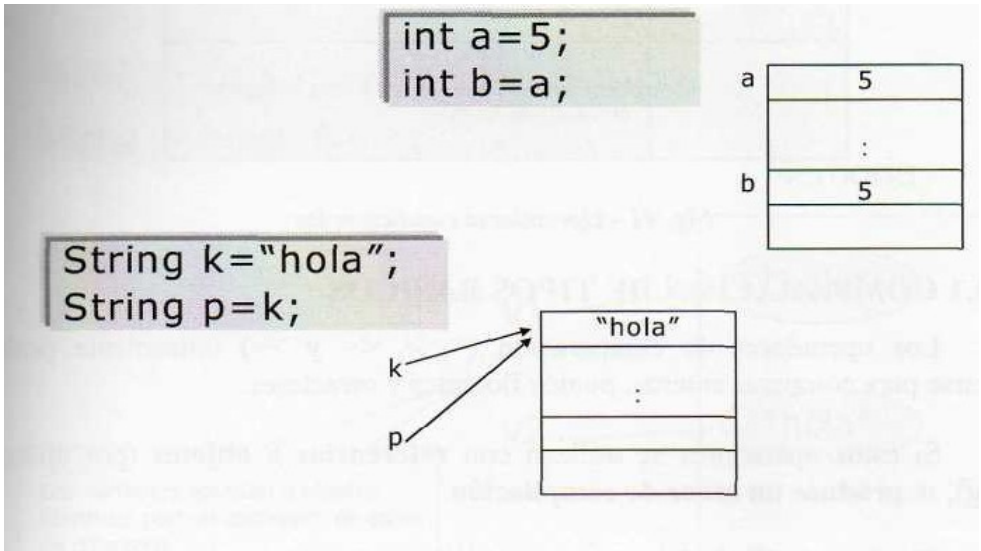


Fig- 12 - Comparativa entre la asignación con variables enteras y con variables Strin

Condicionales

Se utilizan para establecer una condición dentro de un programa, el resultado de ésta será un tipo boolean *(true o false)*. Estos operadores se utilizan en instrucciones de control de flujo, tal y como veremos en el apartado siguiente.

La tabla de la figura 13 contiene el listado de los operadores condicionales Java.

Operador	***Descripción***
==	Compara dos valores, en caso de que sean iguales el resultado de la operación será *true.*
<	Si el operando de la izquierda es menor que el de la derecha, el resultado es *true.*
>	Si el operando de la izquierda es mayor que el de la derecha, el resultado es *true.*
<=	Si el operando de la izquierda es menor o igual que el de la derecha, el resultado es *true*
>=	Si el operando de la izquierda es mayor o igual que el de la derecha, el resultado es *true.*
!=	Si el valor de los operandos es diferente, el resultado es *true.*

Fig. 13 - Operadores condicionales

COMPARACIÓN DE TIPOS BÁSICOS

Los operadores de comparación (<, >, <= y >=) únicamente podrán utilizarse para comparar enteros, puntos flotantes y caracteres.

Si estos operadores se utilizan con **referencias a objetos** (por ejemplo String), se **produce un error de compilación:**

```
String s = "hola";
char c = 'k';
íf(c <= 20) //OK
if(s > "adios") //Error de compilación
```

IGUALDAD DE OBJETOS

Los operadores de igualdad "**==**" y desigualdad "***!=***" pueden utilizarse para cualquier tipo de dato compatible.

Ahora bien, si los utilizamos para comparar variables de tipo objeto ***msxzx**** recordar que lo que contienen estas variables son referencias a objetos, no bjetos en sí, por tanto, estamos comparando referencias y no objetos.

Esto implica que podemos tener dos variables referenciando a dos objetos iguales y que la condición de igualdad de las variables resulte falsa. En la figura 14 se nuestra un ejemplo de esto utilizando la clase String, en ella se aprecia que hay dos objetos idénticos (mismo valor de texto), situados en zonas de memoria diferentes, por lo que las referencias a los mismos serán también distintas.

Para comprobar la igualdad de objetos, las clases proporcionan un método llamado *equals*, en el que cada clase implementa su propio criterio de igualdad. Por tanto, **si queremos saber si dos objetos de una determinada clase son iguales, se debe utilizar el método *equals***, no el operador de comparación "**==**", el cual solamente daría un resultado verdadero si las dos variables apuntan al mismo objeto. Más adelante se verá la aplicación de este método en algunas clases básicas de Java.

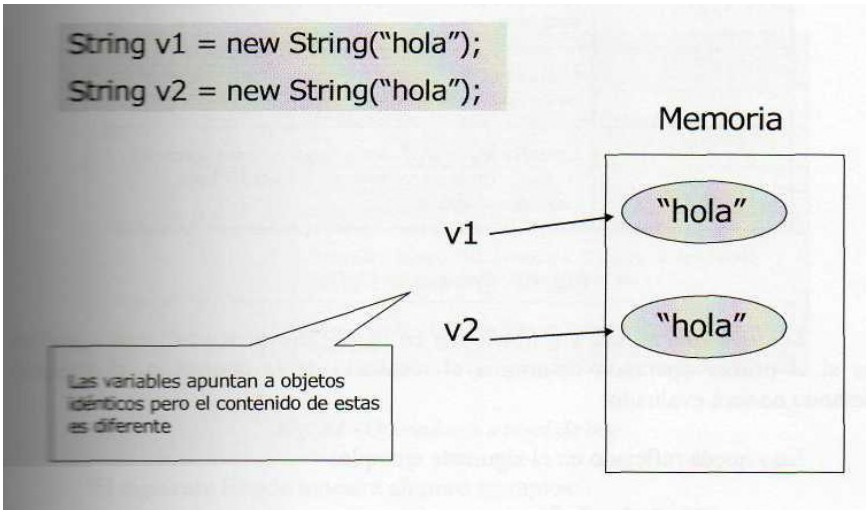


Fig. 14 - Variables diferentes apuntando a objetos iguales

Un último apunte sobre la utilización del operador "==" con referencias:

Solamente está permitida la comparación de referencias del mismo tipo de objetos, si se comparan dos variables de tipos de objetos diferentes, se producirá un error de compilación.

El siguiente bloque de código no compilará:

```
Integer i = new Integer(25);
String s = new String("hola");
if(i == s){} //¡Error de compilación al comparar
             //referencias de distinta clase!
```

Lógicos

Operan con valores de tipo boolean, siendo el resultado también de tipo boolean.

La tabla de la figura 15 muestra los tres operadores lógicos de Java.

Operador	*Descripción*
&&	Operador lógico AND. El resultado será true si los dos operandos son true, en cualquier otro caso el resultado será false
||	Operador lógico OR. El resultado será true si alguno de los operandos es true
!	Operador lógico NOT. Actúa sobre un único operando boolean, dando como resultado el valor contrario al que tenga el operando

Fig. 15 - Operadores lógicos

Los operadores && y || funcionan en modo "cortocircuito", esto significa que si el primer operando determina el resultado de la operación, el segundo operando no será evaluado.

Esto queda reflejado en el siguiente ejemplo:

```
int p = 4, f = 2;
if ((p > 0) || (++f > 0)){
    p++;
}
System.out.println("p vale "+p);
System.out.println("f vale "+f};
```

Al ejecutarse este código se imprimirá en pantalla lo siguiente:

p vale 5

f vale 2

Lo que demuestra que la expresión incluida en el segundo operando de la OR **no llega a ejecutarse** pues, al ser *true* el primer operando (p>0), el resultado de la operación será directamente *true.*

Operadores a nivel de bits

Existe una "versión" de operadores lógicos que no operan en modo cortocircuito. Se trata de los operadores lógicos a nivel de bits, que, además de evaluarlos dos operandos de la expresión, su principal característica es **que operan** a nivel **de bits, pudiendo ser el tipo de los operandos tanto bolean como entero.**

En la tabla de la figura 16 tenemos los cuatro operadores a nivel de bits existentes.

Operador	***Descripción***
&	Operador lógico AND. Realiza la operación AND entre los operandos, bit a bit.
|	Operador lógico OR. Realiza la operación OR entre los operandos, bit a bit.
^	Operador lógico OR exclusiva. Realiza la operación OR exclusiva entre los operandos, bit a bit.
~	Operador NOT. Invierte el estado de los bits del operando.

Fig. 16 - Operadores a nivel de bits

El siguiente listado muestra algunos ejemplos:

```
int k = 5, p = 7;
boolean b = true;
long c = 5;
System.out.println("b vale " + b);
System.out.println("c vale " + c);
System.out.println("k & p vale "+(k & p));
```

La ejecución de este código generará:

b vale true
c vale 5
k & p vale 5

Operador instanceof

Opera únicamente con referencias a objetos y se utiliza para saber si un objeto pertenece a un determinado tipo. La forma de utilizarlo es:

referencia_objeto* instanceof *clase

El resultado de la operación es *true* si el objeto pertenece a la clase especificada o a una de sus subclases.

Por ejemplo, al ejecutar el siguiente código:

```
String  s = "Hola";
if(s instanceof String}{
System.out.println("Es una cadena");
```

Se mostrará en pantalla el texto:

Es una cadena

Operador condicional

Se trata de un operador ternario (consta de tres operandos) cuya función es asignar un valor entre dos posibles a una variable, en función del cumplimiento o no de una condición. Su formato es:

tipo variable = (condicion)?valor_si_true:valor_si_false

Si la condición resulta verdadera (el resultado es *true)*, se almacenará en la variable el resultado de la expresión *valor_si_true*, si no, se almacenará *valor_si_false*

La ejecución del siguiente bloque de código mostrará en pantalla la palabra "par".

```
int k = 4;
String s = (k % 2 == 0)?"par":"impar";
System.out.println(s);
```

EL RECOLECTOR DE BASURA DE JAVA

Aunque no se trata de un tema relacionado propiamente con la sintaxis del lenguaje, el estudio del recolector de basura resulta fundamental para comprender lo que sucede con los objetos de Java cuando dejan de estar apuntados o referenciados por una variable.

El recolector de basura (Garbage Collector) es una aplicación que forma parte de la JVM y cuyo objetivo es liberar de la memoria los objetos no referenciados. Cuando un objeto deja de estar referenciado, se le marca como "basura", a partir de entonces la memoria que ocupa puede ser liberada por el recolector (figura 17).

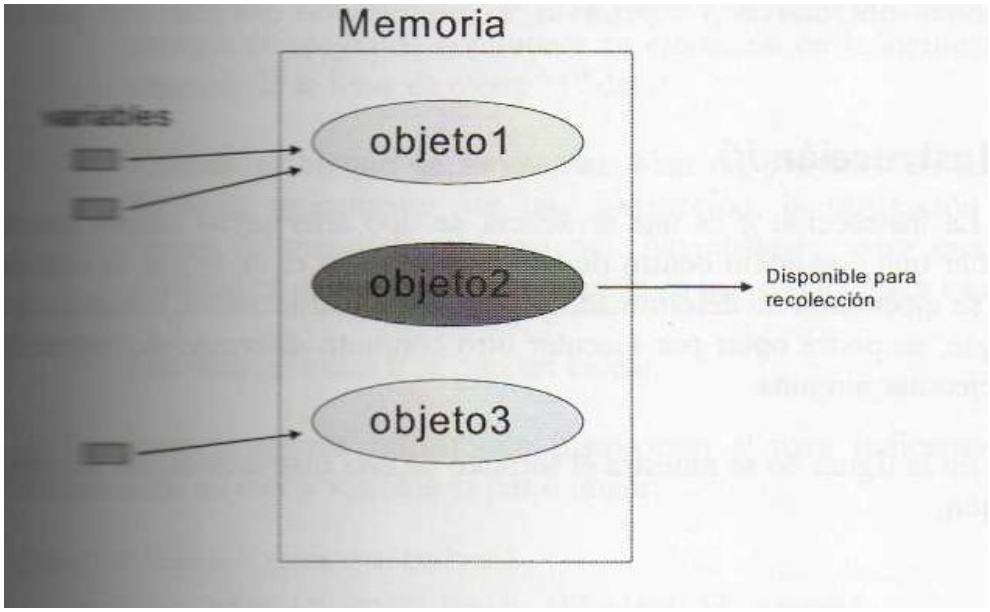


Fig. 17 - Limpieza de objetos no referenciados

La decisión de en qué momento se va a ejecutar el recolector de basura depende de la implementación de la máquina virtual, por tanto, en Java no es posible saber en qué momento exacto un objeto será destruido, sólo es posible determinar a partir de qué momento esto puede suceder (cuando se elimine la última referencia al mismo).

Si un objeto va a dejar de ser utilizado en un programa, conviene eliminar las referencias al mismo para que sea marcado como "basura". Esto se puede hacer asignando el valor *null a* la variable o variables que apuntan al objeto:

```
Integer i = new Integer(30); //se crea el objeto
.
.
i = null; //pérdida de referencia
```

Aunque no asignemos explícitamente el valor *null* a la variable que apunta al objeto, en el momento en que ésta salga de ámbito, se perderá la referencia y de la misma forma que antes, el objeto será marcado como "basura".

EVITE ESTOS ERRORES

Pon atención a estos errores:

- **No distinguir mayúsculas y minúsculas** En Java, Casa y casa **son identificadores distintos.**
- **Omisión del punto y coma** Todas las sentencias de Java deben terminar con punto y coma.
- **Comentarios anidados** Recuerde que los comentarios no se pueden anidar, es decir, dentro de un comentario no puede aparecer el símbolo /*.
- **Comentarios mal finalizados** Un comentario debe comenzar con el símbolo /*y finalizar con */. Si se olvida alguno de estos símbolos se provocará un error..
- **Empleo de variables no definidas** Es también muy común utilizar en los programas variables no definidas, bien porque se ha olvidado su definición o bien porque existe algún problema con las mayúsculas y las minúsculas.
- **Cadenas de caracteres sin dobles comillas** Cuando se utiliza una cadena de caracteres en un programa como literal y no se abre o cierra adecuadamente utilizando ", se produce un error de compilación.
- **Olvido de las llaves en la función** main Todo las sentencias de la función main, y en general de todas las funciones, deben ir encerradas entre { y }.
- **Intentar asignar tipos distintos sin especificadores de conversión** Cuando no se especifica, o se emplea un especificador de conversión incorrecto, en la asignación de variables de dos tipos distintos, se obtendrá un error de compilación.