Programación Avanzada
UD2

Autor:
P. Pablo Garrido Abenza pgarrido@umh.es

Universidad Miguel Hernández

01 de Octubre de 2012

ÍNDICE GENERAL

CAPÍTULO 2

ELEMENTOS DEL LENGUAJE

Como en todos los lenguajes de programación, un programa es un fichero de texto que contiene una serie de instrucciones, las cuales deben cumplir unas normas (sintaxis). En este capítulo se describe la sintaxis del lenguaje Java para la declaración de variables, constantes, los operadores disponibles, estructuras de control del flujo de ejecución, etc. Al terminar este capítulo, el alumno podrá escribir programas Java ejecutables desde la consola del sistema, sin utilizar ninguna de las características de la Programación Orientada a Objetos (POO).

2.1. Declaración de variables y constantes

Una **variable** es una zona de memoria que permite almacenar un valor de algún tipo (numérico, cadena de caracteres, ...), y que se le asocia un nombre o identificador. Dependiendo del tipo de valor a almacenar (definido en la declaración de la variable), se reserva un espacio mayor o menor, que en Java es fijo independientemente de la plataforma en la que se ejecute el programa; por ejemplo, una variable de tipo entero (`int`) siempre ocupará 4 bytes, tanto en Windows como en Linux (cosa que no ocurre en otros lenguajes como C). El siguiente fragmento de código declara una variable `a` de tipo entero (`int`):

```
1     int a;
```

Más adelante se enumerarán los tipos primitivos que ofrece Java, se explicarán los posibles valores que pueden asignarse, y posibles conversiones entre ellos.

Por otra parte, también podemos declarar **constantes** que son muy similares a las variables con la excepción de que no pueden cambiar de valor durante la ejecución. Se declaran prácticamente igual que las variables, precediendo la declaración de la palabra reservada `final`, e inicializando el valor, que el caso de las variables era opcional:

```
1     final static int NUM_ALUMNOS = 50;
```

Podemos definir constantes dentro de una clase o dentro de un método. Si lo hacemos dentro de la clase conviene especificar el modificador `static`, para evitar que cada uno de los objetos de esta clase guarden una copia de la constante (todos los objetos compartirán el mismo valor). Dentro de un método no podremos especificar `static`, y el ámbito de la constante sólo será el método donde está definida.

La utilidad de las constantes es evitar el uso de *números mágicos* entre las líneas de nuestros programas, normalmente difíciles de recordar, consiguiendo programas más claros, fáciles de entender y mantener. La idea de *números mágicos* se entiende como valores constantes que aparecen en el código que nadie sabe su significado a primera vista, números que parecen puestos al azar, de ahí el nombre de *números mágicos*. En general, es una mala práctica de programación, y se aconseja el uso de constantes.

El siguiente ejemplo calcula el ratio de ordenadores disponibles por alumno, es decir, divide el número de alumnos (50) entre el número de ordenadores (30):

```
1    float ratio = (float) 50 / 30;
```

Utilizando constantes el código podría quedar como sigue:

```
1    final static int NUM_ALUMNOS = 50;
2    final static int NUM_ORDENADORES = 30;
3    ...
4    float ratio = (float) NUM_ALUMNOS / NUM_ORDENADORES;
```

Si alguien ajeno al programador original (o incluso él mismo algún tiempo después) intentase comprender lo que hace el primer fragmento de código, puede que le lleve más tiempo de la cuenta descrifrar esos números mágicos (50 y 30); por otro lado, utilizando constantes el código es más claro. Además, el código es más fácil de modificar ya que las constantes normalmente se definen al principio de la clase, por lo que si en algún momento hubiese que modificar el número de alumnos o el número de ordenadores, bastaría con modificar un único valor bien localizado, sobre todo si esos valores (50 y 30) se utilizan varias veces en el código.

Identificadores

Los identificadores son los nombres que se utiliza para identificar una variable, una constante, una función o método, una clase, etc. El nombre que puede utilizarse como identificador debe cumplir una serie de requisitos, aunque es mucho más flexible que en otros lenguajes:

- Debe comenzar por una letra (o subrayado '_' o el signo del dolar '$'), pero nunca con un dígito (`0..9`). Los siguientes caracteres ya pueden ser letras o digitos indistintamente.
- No tiene límites en cuanto a su longitud, es decir, puede tener cualquier número de caracteres.
- Java es sensible a las mayúsculas, es decir, la variable '`a`' es distinta de la variable '`A`'.
- Como Java utiliza el juego de caracteres Unicode [1] que incluye caracteres de muchos idiomas (incluyendo la '`ñ`' y la '`ç`') podemos utilizarlos todos para los identificadores, aunque podría sustituirse por su código Unicode (*uxxxx*) si portamos el código a otros sistemas.

[1] Para más información sobre Unicode podemos visitar la página web del organismo.

- Puede utilizarse cualquier nombre salvo las palabras reservadas o palabras clave (ver abajo).

La tabla 2.1 muestra las **palabras reservadas** por el lenguaje que no podremos utilizar como identificadores en los programas; solamente se podrán utilizar para el uso para el que fueron pensados.

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

Cuadro 2.1: Palabras reservadas (*keywords*)

Algunas palabras reservadas, aun no teniendo uso definido actualmente, tampoco pueden utilizarse como identificadores, como es el caso de `const` o `goto`, que se han reservado por mantener compatibilidad con C++ o para usos futuros (marcadas en naranja en la tabla). Para terminar, comentar que algunas de estas palabras reservadas han sido añadidas en nuevas versiones de Java: `strictfp` (v1.2), `assert` (v1.4), y `enum` (v1.5).

Con relación al uso de identificadores, aunque no es obligatorio se recomienda seguir las siguientes recomendaciones:

- Utilizar **nombres descriptivos**, para que el código sea autodocumentado, es decir, que sin necesidad de muchos comentarios entre el código se pueda comprender lo que hace.
- Para nombres de **variables** y **métodos**, escribirlos en minúscula, y si consta de varias palabras, separarlas poniendo la primera letra en mayúsculas (salvo la primera palabra que siempre irá en minúscula), es decir, no utilizar subrayados o guiones. Por ejemplo: `int diasMes;` o `añoBisiesto()`.
- Para nombres de constantes, hacerlo al contrario que para variables, es decir, escribirlos totalmente en mayúsculas, y separando las distintas palabras con subrayado. Por ejemplo: `final static int MESES_AÑO = 12;`

Alcance o visibilidad

Una **variable** puede declararse dentro de una clase (variable global), dentro de un método o función (variable local), o dentro de un bloque escrito dentro de un método (variable de bloque). Dependiendo de donde se haya declarado, la variable será visible o accesible desde todo el archivo o clase (variable global), o sólo desde un fragmento de código (variable local, o variable de bloque). Esto es lo que se conoce como *alcance*, *visibilidad* o *ámbito* de una variable, también válido para un objeto, pero de momento no entramos en aspectos de la Programación Orientada a Objetos (POO). Por tanto, el alcance de una variable u objeto es la porción de programa en la que se puede hacer referencia a dicha variable.

En el caso de declarar una variable dentro de un *bloque*, es decir, dentro de un conjunto de sentencias entre llaves `{...}`, su alcance será entre las dos llaves en las que se ha declarado. Esto no lo permitía el lenguaje C, y sería el caso de declarar una variable dentro de un bucle `for` o en una sentencia condicional `if`, por ejemplo. Sin embargo, esto no es tan sencillo como parece:

1. Cuando se anidan bloques, y un identificador de un bloque exterior tiene el mismo nombre que un identificador de un bloque interior, el compilador genera un error de compilación, para indicar que esa variable ya está definida.

2. Si una variable local o un parámetro de un método tiene el mismo nombre que una variable global (variable de clase), éste queda oculto hasta que el bloque termine de ejecutarse, es decir, tiene prioridad la variable definida dentro del bloque.

Vemos que en Java, una variable puede declararse en casi cualquier parte. No obstante, conviene seguir las siguientes recomendaciones para conseguir una mayor claridad en el código y un fácil mantenimiento del mismo:

- Declarar las variables como locales a un método siempre que sea posible, evitando el uso de variables globales para eliminar la posibilidad de que se produzcan efectos colaterales, es decir, que sean modificadas de forma indeseada dentro de algún otro método.

- No utilizar variables de bloque para evitar conflictos con otras variables que se llamen igual declaradas en un bloque superior (variable local o global). Sin embargo, algo que se utiliza con bastante frecuencia en los bucles `for` es declarar una variable en la misma sentencia cuando se trata simplemente de contar el número de iteraciones, como se muestra en el siguiente ejemplo.

```
1  public class MiProg {
2     static int a=0;  // a = variable global
3
4     public static void main (String args[]) {
5        int b=0;      // b = variable local
6
7        for (int i=0; i<50; i++) {   // i,c = variables de bloque (for)
8           // Generamos un valor aleatorio entre 1 y 100
9           int c = (Math.random()*100)+1;
10          // Sumamos el valor generado tanto en a como en b
11          a = a + c;
12          b = b + c;
13       }
14       // Mostramos el resultado de las variables visibles (a y b)
15       // En este punto, las variables de bloque i y c no son visibles
16       System.out.println ("Valor a=" + a);
17       System.out.println ("Valor b=" + b);
18    }
19 }
```

El ejemplo anterior ilustra la declaración de variables en todos los ámbitos posibles:

- **Variable global**: la variable `a` es una variable global accesible desde toda la clase (función `main()` o cualquier otra que hubiese dentro de la misma clase) [2].
- **Variable local**: la variable `b` es una variable local a la función `main()`, sólo visible dentro de ella. Si hubiese otra función dentro de la clase, podría declararse otra variable local a ella con el mismo nombre, y serían variables distintas, cada una visible únicamente en su propia función.
- **Variables de bloque**: la variable `i` se declara en la misma sentencia `for`, y sólo estará visible mientras dura el bucle; un vez terminado no se podrá consultar o imprimir el valor final de dicha variable, pues ya no está visible fuera del bloque correspondiente al bucle `for`. Este es un caso de variable de bloque que sí se aconseja, pues simplemente se utiliza para contar que el bucle se ejecute 50 veces y nunca se utiliza fuera del bucle. Sin embargo, la variable `c`, siendo también una variable de bloque y estando permitida su declaración, no se aconseja por los problemas de conflictos con otras variables que pudieran llamarse igual pero que están definidas en otros ámbitos superiores, tal y como se ha comentado anteriormente.

2.2. Tipos de datos

El lenguaje Java es un lenguaje fuertemente tipado, es decir, las variables deben declararse antes de su utilización, especificando el tipo de valores que podrá almacenar. La asignación de valores de un tipo diferente sólo se permite en ciertos casos, en ocasiones es necesario realizar una conversión explícita (casting), o en otras no es posible.

Existen 8 **tipos** de datos primitivos, es decir, soportados de forma implícita por el lenguaje, y cada uno de ellos tiene asociada una palabra reservada (ver tabla tabla 2.2). Es importante elegir correctamente el tipo de las variables según el valor máximo o mínimo que van a almacenar con objeto de no desperdiciar memoria, sobre todo si definimos un array de muchos valores. Caso de necesitar mayores valores enteros o más precisión en los valores reales, podemos hacer uso de las clases `BigInteger` y `BigDecimal`, respectivamente (consultar la librería de clases de Java (API Specification)).

En cualquier lenguaje de programación, cada tipo primitivo ocupa un determinado número de bytes [3] en memoria, pero este puede variar dependiendo de la plataforma donde se ejecute el programa. Por ejemplo, en el lenguaje C, una variable de tipo `int` puede ocupar 2 bytes en Windows, y sin embargo, en un entorno Unix puede ocupar 4 bytes. Esto puede ser un problema a la hora de migrar un programa de una plataforma a otra, y podría ser necesario modificar el código fuente. En Java no ocurre esto, ya que todos los tipos primitivos siempre ocupan lo mismo, independientemente de la plataforma donde se ejecute el programa, tal como se mostraba en la tabla 2.2.

En esa misma tabla se muestra una columna con el **valor por defecto**, que es el valor con el que se inicializa automáticamente una variable global (de clase). Por otro lado, las variables locales a un método no se inicializan automáticamente, cosa que también ocurre con el lenguaje C. Sin embargo, el compilador de Java detecta cuando una variable local se utiliza sin que previamente se le haya asignado un valor, dando

[2] Debe declararse `static` para que pueda utilizarse dentro de una función static, y la función `main()` debe serlo (esto no es relevante para comprender la visibilidad, y se explicará en el siguiente capítulo).

[3] 1 byte = 8 bits.

Tipo	Tamaño	Descripción	Valor mínimo	Valor máximo	Valor por defecto
`char`	2 bytes	Un carácter Unicode	0	65535	`'\0'` o `'\u0000'`
`byte`	1 byte	Entero de 8-bits	-2^7 = -127	$+2^7-1$ = +127	`0`
`short`	2 bytes	Entero corto	-2^{15} = -32768	$+2^{15}-1$ = +32767	`0`
`int`	4 bytes	Entero	-2^{31}	$+2^{31}-1$	`0`
`long`	8 bytes	Entero largo	-2^{63}	$+2^{63}-1$	`0L`
`float`	4 bytes	Real de precisión simple según IEEE 754	Precisión de 6 dígitos (aprox.)		`0.0F`
`double`	8 bytes	Real de precisión doble según IEEE 754	Precisión de 16 dígitos (aprox.)		`0.0D`
`boolean`	2 bytes	Booleano (si/no)	Valores `true` y `false`		`false`

Cuadro 2.2: Tipos de datos primitivos

un mensaje de error al compilar para que el programador la inicialice de forma explícita; el compilador del lenguaje C no genera tal error, permitiendo compilar un programa que, según el caso, podría proporcionar resultados inesperados por tener valor *basura* en las variables que no se hayan inicializado. Además de los valores mostrados en la tabla 2.2, existe un valor adicional con el que se inicializan las referencias a objetos, incluyendo las cadenas de caracteres (`String`): el valor `null`.

Valores constantes (literales)

A la hora de inicializar una variable o constante, o también, a la hora de asignar un valor nuevo a una variable, podemos utilizar una serie de valores o literales, que dependen del tipo de la variable. En esta sección vemos los distintos literales y formas alternativas de escribirlos en el código.

Según el tipo primitivo, los valores literales que pueden utilizarse son:

- **Caracteres**: un valor de tipo `char` es un carácter entre comillas simples, conteniendo cualquier carácter Unicode de 2 bytes (UTF-16). Una cadena de texto se representa entre comillas dobles, conteniendo una serie de caracteres (tipo `char`), y se almacena en un nuevo tipo llamado `String`, pero no se trata de un tipo primitivo sino una clase, y cada cadena de texto concreta será una instancia de la clase `String` (lo veremos más adelante).

```
1    char   letra  = 'a';       // Una letra
2    String nombre = "Pepe";    // Una cadena de caracteres
```

  Como valor de un `char` (ya sea como caracter individual, o como un carácter dentro de una cadena de caracteres) se puede escribir:

  - Cualquier carácter que nos permita el teclado: `'a'`, `'b'`, `'1'`, `'2'`, ...
  - Un carácter de escape (ver tabla 2.3): `'\r'`, `'\n'`, `'\\'`, ...
  - Un código Unicode de la forma `'\uhhhh'` o `'\nnn'` (siendo `h` un dígito hexadecimal `h=0..F`, y `n` un dígito octal `n=0..7`). Es posible asignar un valor entero en decimal (base 10) a una variable de tipo `char`, pero con la condición de realizar *casting* a `char`, ya que el valor especificado será un `int`, que tiene un mayor rango.

Carácter	Código Unicode	Descripción
'\r'	'\u000D'	Retorno de carro (carriage return)
'\n'	'\u000A'	Avance de línea (line feed)
'\f'	'\u000C'	Avance de página (form feed)
'\a'	'\u0007'	Pitido (alert / bell)
'\b'	'\u0008'	Retroceso (backspace)
'\t'	'\u0009'	Tabulador (tab)
'\\'		Barra invertida (*backslash*)

Cuadro 2.3: Carácteres de escape

Por ejemplo, para escribir la letra 'a', cuyo código Unicode (y ASCII) en decimal es 97, podemos escribir su código en hexadecimal '\u0061' o en octal '\141'. El siguiente ejemplo muestra 4 posibilidades de asignar la letra 'a' a una variable de tipo char:

```
1 char letra;
2 letra = 'a';          // Caracter normal
3 letra = (char) 97;    // Codigo Unicode (en decimal)
4 letra = '\u0061';     // Codigo Unicode (en hexadecimal)
5 letra = '\141';       // Codigo Unicode (en octal)
```

Dentro de una cadena de caracteres podemos escribir entre comillas dobles tanto caracteres normales, como caracteres de escape, como el código Unicode en hexadecimal u octal:

```
1     String nombre = "Pablo";
2     int    año    = 2012;
3     char   eñe    = '\u00f1';   // Caracter ñ en Unicode
4     // Uso de codigos Unicode y caracteres de escape (salto de linea)
5     System.out.print ("Nombre: "+nombre+" - A\u00f1o: "+año+"\r\n");
```

La salida del fragmento de programa anterior será la siguiente. El valor de la variable año, aunque sea numérica se transforma automáticamente a cadena de caracteres para imprimirse, es decir, se concatena como caracteres a la cadena a imprimir:

```
Nombre: Pablo - Año: 2012
```

Los 128 primeros caracteres de la codificación Unicode coinciden con los 128 primeros caracteres de la codificación ASCII, ampliamente utilizada en muchas computadoras. Para averiguar el código Unicode de cualquier símbolo podemos visitar la página web del organismo Unicode (sección *Code Charts*). Además, para los carácteres que podamos teclear (por ejempo, la letra 'ñ' o caracteres acentuados) también podemos ejecutar la utilidad native2ascii incluida en el JDK; ejecutaremos la utilidad sin especificar ninguna opción, ya que podríamos especificar un nombre de fichero a convertir, o utilizar una codificación distinta de la del sistema (para salir de la utilidad es necesario pulsar CTRL+C).

- **Valores enteros**: se trata de valores de tipo `byte`, `short`, `int`, y `long`. La forma de escribir los valores es similar en todos los casos, salvo para el tipo `long`, que hay que añadir la letra `L` al final para indicar que el valor es de tipo `long` en vez de `int`. Por defecto, el valor se expresa en el sistema decimal o base 10 (p.e. 25), pero en ciertos ámbitos (electrónica) puede ser de utilidad utilizar otros sistemas como el octal (p.e. `031`), o hexadecimal (p.e. `0x19`). Finalmente, Java SE 7 incorpora dos nuevas posibilidades: (1) utilizar también el sistema binario (p.e. `0b11001`), y (2) cuando un valor es muy largo, puede separarse mediante subrayados para conseguir mayor claridad en el código, por ejemplo, para separar los miles, centenas, grupos de 4 bits, etc. (ver variable `distanciaSol` en el siguiente ejemplo) o grupos de bits si se utiliza el sistema binario (grupos de 8-bits o 4-bits).

```
1     byte  codigoA = (byte) 65;          // Código Unicode de la letra 'A'
2     short edadJubilacion = (short) 80;  // Tipo byte podría quedarse corto
3     int   distanciaLuna = 400000;       // Distancia aprox. a la luna (km)
4     long  distanciaSol = 149_600_000L;  // Distancia aprox. al sol (km)
5     int   mascara = 0b11001;            // Valor 25 en binario (Java 7)
```

- **Valores reales o decimales**: tanto el tipo `float` como el `double` se utilizan para representar valores con decimales (reales) según el formato definido por el estándar IEEE 754. Unicamente se diferencian en la precisión del valor dependiendo de la cantidad de bits utilizados: (1) precisión simple (32-bits) utiliza 1 bit de signo, 8 de exponente, y 23 para la mantisa; (2) precisión doble (64-bits) utiliza 1 bit de signo, 11 para el exponente, y 52 para la mantisa. En Java una constante numérica expresada con decimales debe llevar un sufijo al final: `F` para indicar que de simple precisión (`float`) o `D` para indicar que es doble precisión (`double`); no obstante, cuando no lleva sufijo se supone que será `double`. Alternativamente, también se puede utilizar la notación científica, es decir, $mantisa\ x\ 10^{exponente}$.

```
1     float  horasTrabajo = 12.5F;      // Horas diarias de trabajo
2     double salarioMes = 600.15D;      // Salario mensual (euros)
3     double deuda = 1.5E6;             // Millon y medio de euros
```

- **boolean**: tan sólo se puede utilizar el valor `false` (falso) o `true` (cierto). Como veremos en la siguiente sección, no es posible asignar ningún otro valor ni siquiera realizando alguna conversión explícita.

```
1     boolean fin = true;
```

Conversiones de tipos

En ocasiones es necesario realizar asignaciones de un tipo primitivo a otro, o bien, tenemos una expresión en la que tenemos mezclados valores o variables de distinto tipo primitivo. En estos casos se necesita realizar conversión de un tipo a otro, bien de forma automática, o bien de forma explícita (*casting*). Java es un lenguaje fuertemente tipado, es decir, cada variable debe ser declarada previamente a su uso indicando qué tipo de datos almacenará. C también lo es, sin embargo, en Java hay ciertas asignaciones que se detectan en tiempo de compilación, impidiendo que el programa se compile, cosa que en C no ocurría por no ser tan estricto, pudiendo dar lugar a resultados inesperados durante la ejecución.

Para que una asignación de un valor o variable a otra variable de tipo diferente pueda ocurrir sin problemas, deben darse dos circunstancias (reglas de promoción):

1. Ambos tipos deben ser compatibles: la mayoría lo son, salvo el `boolean`.
2. Que el tipo destino tenga un mayor rango de representación para que no haya pérdida de información.

Cuando se dan las dos circunstancias anteriores, la asignación es permitida y si es necesario realizar alguna conversión se realiza de forma automática; cualquier otro caso no podrá realizarse la asignación, salvo que el programador de forma explícita realice la conversión mediante la técnica conocida como *casting*. Mediante el *casting* la expresión se convierte al tipo especificado, en el caso en que dicha conversión esté permitida (si cumple las reglas de promoción); en otro caso obtendremos un error en tiempo de compilación.

La figura 2.1 muestra las posibles conversiones automáticas (siguiendo el sentido de las flechas); si se realiza una asignación en sentido contrario al de las flechas, se deberá utilizar casting si realmente es lo que se desea, de lo contrario, el compilador generará un error de compilación: "*possible loss of precision*" (traducido, "*posible pérdida de precisión*").

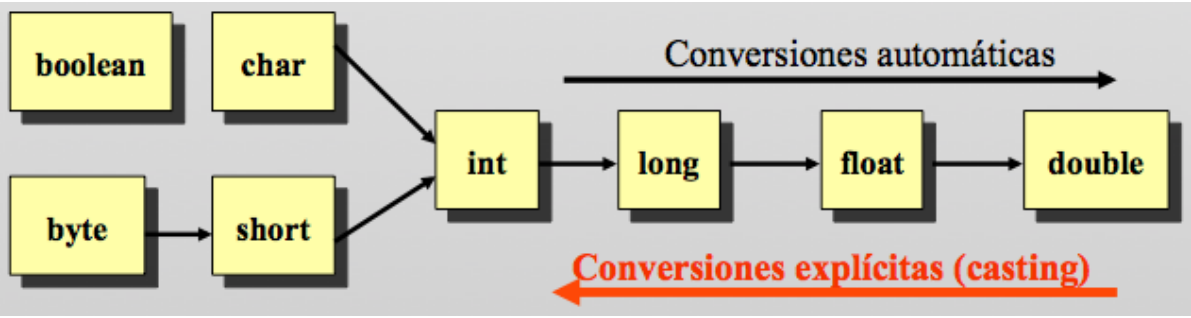

Figura 2.1: Conversiones automáticas y explícitas

En el siguiente ejemplo se declaran dos variables reales: una `float` y una `double`, que según la tabla 2.2 requieren 4 y 8 bytes, respectivamente. Después se intenta asignar el valor del `float` al `double` (no habiendo ningún problema), y después al contrario, siendo necesario el uso de *casting* en este caso. Lo mismo ocurriría si se utiliza una variable de tipo `int` y otra de tipo `long`, por ejemplo.


```
1  float  fValor = 5.2F;        // Sufijo F obligatorio
2  double dValor = 5.6D;        // Sufijo D opcional (double por defecto)
3
4  dValor = fValor;             // Conversión automática
5  fValor = (float) dValor;     // Casting obligatorio
```

La técnica del casting puede utilizarse para truncar los decimales de un número real, simplemente convirtiendo de forma explícita a entero.

```
1  double PI = 3.14159265;
2  int parteEntera = (int) PI;              // 3
3  double parteDecimal = PI - parteEntera;  // 0.14159265
```

Un tipo primitivo que en principio parecería incompatible con los enteros es el tipo `char`, ya que representa un caracter. Sin embargo, sí que es compatible, ya que los caracteres se almacenan internamente como un valor entero: su código Unicode. Cada símbolo, letra, número, etc. tiene asociado un código Unicode, por ejemplo, el `'a'` es el 97, el `'b'` el 98, ... El siguiente ejemplo hace un bucle para mostrar todas las letras del alfabeto, desde la `'a'` hasta la `'z'`, inicializando la variable `char` con `'a'`, y sumando el valor 1 en cada iteración. Sin embargo, hay que tener cuidado con esta suma, puesto que ese 1 es un valor entero de tipo `int` (4 bytes), teniendo por tanto un mayor rango que el tipo `char`; por ello, se puede observar en la línea 13 que al realizar la suma, el resultado es el de mayor rango (`int`), y por tanto, para asignarlo a una varible de tipo `char` es necesario realizar el *casting* apropiado.

```
1  public class Alfabeto {
2
3     // Numero de letras del alfabeto
4     final static int NUM_LETRAS = 24;
5
6     public static void main (String args[]) {
7        char c = 'a';  // Primera letra
8
9        for (int i=0; i<NUM_LETRAS; i++) {
10          // Imprimimos letra
11          System.out.print (c);
12          // Pasamos a la siguiente letra
13          c = (char) (c + 1);
14       }
15       System.out.println ("");
16    }
17 }
```

Debemos evitar el uso de *casting*, pues anulamos la verificación de tipo proporcionada por el compilador, pudiendo producirse resultados inesperados, o como mínimo, una pérdida de información en el resultado. Además de realizar casting entre tipos primitivos es posible realizar *casting* también entre clases, lo cual se explicará en el capítulo correspondiente a Programación Orientada a Objetos.

2.3. Comentarios

Es muy recomendable escribir comentarios de texto junto con el código propiamente dicho de cualquier programa, los cuales servirán para aportar alguna aclaración de una parte del código que el programador considere oportuna para facilitar su comprensión: variables utilizadas, pasos en un algoritmo implementado, etc. En Java hay 3 posibilidades para insertar comentarios:

1. Comentario de una o varias líneas: `/* ... */`
2. Comentario hasta fin de línea: `// ...`
3. Comentario de la utilidad `javadoc`: `/** ... */`

Los dos primeros tipos de comentarios ya existían en C / C++, y contienen texto libre que será ignorado por el compilador, es decir, el texto no sigue ninguna norma. Si recordamos el primer ejemplo mostrado `HolaMundo`, se utilizaban ambos tipos de comentarios:

```
1  /* ========================
2   * Archivo: HolaMundo.java
3   * ======================== */
4  class HolaMundo {
5     public static void main (String args[]) {
6        // Mostramos un mensaje
7        System.out.println ("Hola mundo!");
8     }
9  }
```

En cuanto al tercer tipo de comentarios para javadoc, el compilador los trata de forma similar al primer tipo, es decir, ignora todo el texto incluido entre `/**` y `*/`. Sin embargo, el código fuente puede ser procesado por una utilidad llamada `javadoc` que viene incluida en el JDK, cuyo objetivo es generar una serie de páginas web (.html) incluyendo los comentarios de ese tipo, del estilo a la documentación on-line de Java (ver figura 2.2). Se obtendrá un índice con las distintas clases definidas por una aplicación, y al seleccionar una de ellas nos aparecerá su contenido (campos, constructores, métodos, argumentos que recibe cada método, valor retornado, etc.), con el contenido de los comentarios insertados.

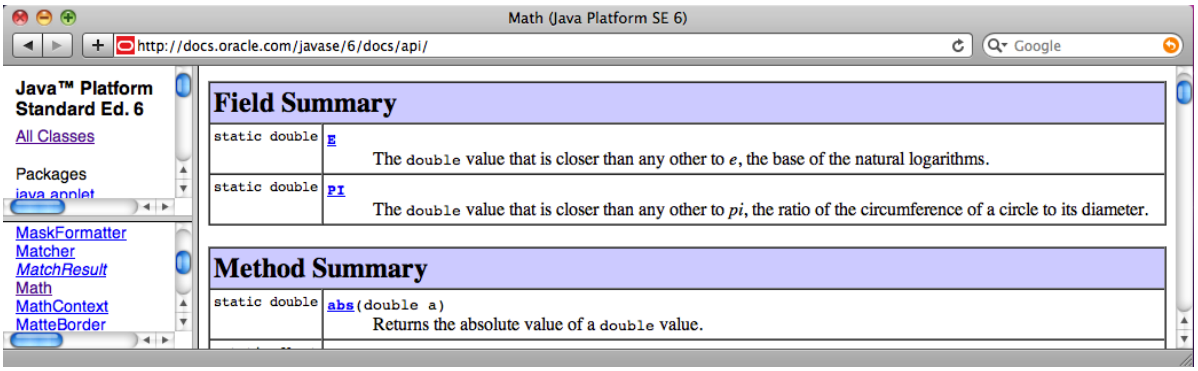

Figura 2.2: Documentación Java API

De momento no se explica cómo escribir este tipo de comentarios, pues en este caso el texto insertado como comentario sí que debe seguir ciertas normas para que la utilidad `javadoc` funcione correctamente; para ampliar información puede consultarse el siguiente enlace.

2.4. Operadores

Los **operadores** son unos símbolos que permiten realizar alguna operación sobre variables, normalmente una o dos variables. También existe un operador ternario, como se mostrará en las siguientes secciones. Los operadores pueden clasificarse en varios grupos:

- Asignación y aritméticos.
- Relacionales y condicionales.
- Bits.

Operador de asignación y aritméticos

El **operador de asignación** `'='` se utiliza para asignar un valor a una variable o inicializar una constante.

```
int    edad = 18;                // Declaración e inicialización de variable
float temperatura = 37.0F;      // Declaración e inicialización de variable
final static double PI = 3.14159265D;        // Declaración la constante PI
```

Al igual que es posible declarar varias variables del mismo tipo en la misma sentencia, también es posible inicializar variables a un mismo valor. Los 4 siguientes fragmentos de código realizan la misma tarea, pero dependiendo de la situación puede que nos convenga un método u otro:

```
// Declaración e inicialización
// individual
int a = 0;
int b = 0;
int c = 0;
```

```
// Declaración múltiple
int a, b, c;
// Inicialización individual
a = 0;
b = 0;
c = 0;
```

```
// Declaración múltiple
int a, b, c;
// Inicialización múltiple
a = b = c = 0;
```

```
// Declaración múltiple e
// inicialización individual
int a=0, b=0, c=0;
```

IMPORTANTE: no confundir el operador de asignación `'='` con el comparador de igualdad `'=='`, el cual se explicará en la siguiente sección.

Cuando lo que se asigna a una variable es una expresión (por ejemplo, una fórmula matemática), se hace uso de los **operadores aritméticos**, que se resumen en la tabla 2.4.

Operador	Descripción	Ejemplo
+	Suma	`a = a + 12;`
-	Resta	`a = a - 10;`
*	Multiplicación	`diasAño = 30 * 12;`
/	División	`diasMes = 365 / 12;`
%	Módulo (resto división entera)	`resto = 365% 12;`
++	Preincremento / Postincremento	`a++; ++a;`
-	Predecremento / Postdecremento	`a-; -a;`

Cuadro 2.4: Operadores aritméticos

Todos estos operadores también existen en su correspondiente versión asignación, para operar con una variable y asignar el resultado en ella misma: +=, -=, *=, /=, y %=. Harían algo similar a los últimos 2 operadores de la tabla: ++ y -, salvo que éstos últimos siempre suman o restan el valor 1. Los siguientes ejemplos serían todos equivalentes, es decir, el valor final de la variable a será 1:

```
1  int a = 0;
2  a = a + 1;
```

```
1  int a = 0;
2  a += 1;
```

```
1  int a = 0;
2  a++;
```

Para operar con otros valores distintos de 1, como se ha dicho anteriormente, ya no se podría utilizar los operadores ++ o -. Por ejemplo, para restar 5 podríamos hacerlo de dos formas:

```
1  int a = 0;
2  a = a - 5;
```

```
1  int a = 0;
2  a -= 5;
```

El operador suma '+' además de sumar valores numéricos (enteros y reales) también se utiliza para la **concatenación de cadenas de caracteres** cuando uno de los dos operandos sobre los que actúa es una cadena (String). Esto es muy útil para mostrar valores por la pantalla, ya que la conversión de valores numéricos a cadena es automática. Aunque las cadenas las veremos más adelante, vamos a mostrar unos ejemplos relacionados con esto. Este primer ejemplo (izquierda) muestra un fragmento de código que imprime un texto seguido del valor que tenga la variable a (en la derecha):

```
1      int a = 5;
2      a = a + 1;
3      System.out.println ("Resultado: " + a);
```

```
Resultado: 6
```

Este segundo ejemplo, muy similar al anterior, ilustra la diferencia entre el uso del operador '+' como suma o como concatenación:

```
1      int a = 5;
2      System.out.println ("Resultado: " + (a+1));
```

```
Resultado: 6
```

El resultado de este fragmento será idéntico al anterior, pero en este caso, la suma del valor 1 se hace en la misma línea de la llamada al método println(); en esa línea podemos ver dos operadores '+', el primero de ellos actúa como concatenación de cadenas (igual que antes), pero el segundo realiza una suma (a+1), y el valor resultante se muestra por pantalla (el valor final de la variable 'a' no varía en este caso). Si en el ejemplo anterior no hubiésemos utilizado paréntesis, el resultado hubiera sido diferente, puesto que se hubieran considerado los dos operadores '+' como de concatenación, concatenando el valor de la variable 'a' (5) seguido del valor 1 convertido a cadena. Esto se muestra en el siguiente código:

```
1     int a = 5;
2     System.out.println ("Resultado: " + a + 1);
```

```
Resultado: 51
```

Cuando se explique el uso de cadenas de caracteres se ampliará este punto.

Una cosa a tener en cuenta con el **operador de división** `'/'` es que si ambos operandos son enteros (no de tipo real `float` o `double`), el resultado no tendrá decimales, aunque la variable a la que se asigne el resultado sea real, ya que se realizará una división entera, truncando los decimales, y ese valor truncado será el que se asigne finalmente a la variable. El siguiente ejemplo calcula el área de un triángulo, y el resultado debería ser `17.5`, pero se obtiene `17.0` debido a que lo que hay a ambos lados del operador `'/'` es entero.

```
1     int base = 5;
2     int altura = 7;
3     double area = (base * altura) / 2;  // ¿ 17.5 ?
4     System.out.println ("Area del triangulo: " +
          area);
```

```
Area del triangulo: 17.0
```

La solución sería hacer que alguno de los dos operandos sea `double`:

- El operando de la izquierda (dividendo) utilizando la técnica de conversión o *casting*.

```
1     double area = ((double) (base * altura)) / 2;
```

- El operando de la derecha (divisor) simplemente escribiendo el valor 2 como una constante `double`.

```
1     double area = (base * altura) / 2.0D;
```

Operadores relacionales y condicionales

Los **operadores relacionales** pueden utilizarse para comparar variables o valores (igualdad, desigualdad, menor, menor o igual, mayor, o mayor igual). El resultado de la comparación será un valor lógico o *booleano* (`true` o `false`), y podrá utilizarse en cualquier lugar donde se necesite un valor lógico, como por ejemplo, como condición en un `if`, `while`, etc., o simplemente asignarse a una variable de tipo `boolean`.

Operador	Descripción	Ejemplo
==	Relación de igualdad	`a == b`
!=	Relación de desigualdad	`a != b`
<	Relación menor que	`a < b`
<=	Relación menor o igual que	`a <= b`
>	Relación mayor que	`a > b`
>=	Relación mayor o igual que	`a >= b`
!	Negación lógica (NOT)	`!expresion`
&&	Condicional Y lógico (AND)	`expr1 && expr2`
||	Condicional O lógico (OR)	`expr1 || expr2`

Cuadro 2.5: Operadores lógicos

El siguiente ejemplo utiliza una sentencia `if` para averiguar si la temperatura es menor que 0º C, es decir, si estamos bajo cero:

```
public static void main (String args[]) {
   double temperatura = ...;
   ...
   if ( temperatura < 0.0 ) {
      // Bajo cero
      System.out.println ("Estamos bajo cero");
   }
}
```

En el ejemplo anterior tan sólo comprobamos el valor de una expresión booleana (lógica); utilizando los **operadores condicionales** && y || podemos averiguar si se cumple más de una expresión. Con el operador && (Y lógico) comprobamos si son ciertas (`true`) las dos expresiones sobre las que actúa, mientras que con el operador || (O lógico) comprobamos si alguna de las dos expresiones es cierta (una o las dos). Ambos operadores pueden ir encadenándose para la comprobación de tantas expresiones como se necesite; aunque no suele ser obligatorio, por claridad o posibles problemas con la prioridad de los operadores conviene encerrar entre paréntesis cada una de las expresiones.

Por ejemplo, vamos a comprobar si un determinado valor de temperatura está dentro o fuera de un rango o intervalo, utilizando dos formas distintas: (1) comprobando si está dentro del rango, y en caso contrario estará fuera (a la izquierda), y (2) comprobando si está fuera del rango, y en caso contrario estará dentro (a la derecha).

```
final static double MINIMO = 10.0D;
final static double MAXIMO = 20.0D;
double temperatura = ...;
...
if ( (temperatura >= MINIMO) &&
     (temperatura <= MAXIMO) ) {
   // Dentro del rango
} else {
   // Fuera del rango
}
```

```
final static double MINIMO = 10.0D;
final static double MAXIMO = 20.0D;
double temperatura = ...;
...
if ( (temperatura < MINIMO) ||
     (temperatura > MAXIMO)) {
   // Fuera del rango
} else {
   // Dentro del rango
}
```

Si el resultado de una comparación (`true` o `false`) se asigna a una variable de tipo `boolean`, si posteriormente queremos comprobar su valor, podemos compararlo con los valores `true` o `false`, o también de forma abreviada simplemente con el nombre de la variable, o utilizando el operador de **negación lógica** `!` justo delante como se muestra a continuación:

```
boolean a;
...
if (a==true) {
   // Cierto
}
```

```
boolean a;
...
if (a) {
   // Cierto
}
```

```
boolean a;
...
if (a==false) {
   // Falso
}
```

```
boolean a;
...
if (!a) {
   // Falso
}
```


Operadores de bits

Java dispone de una serie de operadores que permiten manipular valores enteros a bajo nivel, permitiendo realizar desplazamientos hacia la izquierda y hacia la derecha de un número concreto de bits, así como ciertas operaciones lógicas entre ellos (NOT, AND, OR, XOR). Los operadores se resumen en la tabla 2.6, y se aplican a valores enteros o compatibles con ellos, es decir, a los tipos primitivos: `byte`, `short`, `int`, `long`, y `char`. Los ejemplos presentados muestran valores escritos utilizando el sistema de numeración binario introducido desde la versión Java 7, es decir, comienzan con el prefijo `0b` seguidos de los dígitos `0` o `1`, tal como se mostró en la sección 2.2. Sin embargo, es indistinto inicializar una variable, por ejemplo, con el valor 4 que con `0b0100`, y podemos utilizar estos operadores exactamente igual; aquí se utiliza el sistema binario por claridad. Caso de utilizar una versión anterior a Java 7, el sistema binario no está disponible, por lo tendría que utilizarse cualquiera de los otros sistemas: octal, hexadecimal, o decimal.

Operador	Descripción	Ejemplo
<<	Desplazamiento a la izquierda	`int n=0b0100; n = n << 1; // n=0b1000`
>>	Desplazamiento a la derecha (copiando el bit de signo)	`int n=0b1100; n = n >> 1; // n=0b1110`
>>>	Desplazamiento a la derecha con relleno de 0's (en vez de copiar el bit de signo)	`int n=0b1100; n = n >>> 1; // n=0b0110`
~	NOT (inversión o complemento)	`int a=0b0101; c=~a; // c=0b1010`
&	AND (bit a bit)	`int a=0b0101, b=0b1100, c=a & b; // c=b0100`
|	OR (bit a bit)	`int a=0b0101, b=0b1100, c=a | b; // c=b1101`
^	OR exclusiva o XOR (bit a bit)	`int a=0b0101, b=0b1100, c=a ^ b; // c=0b1001`

Cuadro 2.6: Operadores de bits

Todos estos operadores también existen en su correspondiente versión asignación, funcionando de forma similar a +=, *=, ..., es decir, asignando el valor final a la primera de las variables sobre las que actúa. En concreto, existen los operadores: <<=, >>=, >>>=, &=, |=, y ^= (para todos salvo ~, ya que no tendría sentido).

Podemos comprobar desde los ejemplos de la tabla 2.6 que cada vez que desplazamos un bit hacia la izquierda equivale a multiplicar por 2; por el contrario, cada vez que desplazamos un bit hacia la derecha equivale a dividir por 2 (ignorando los decimales si los hubiera). En el caso de desplamiento a la derecha tenemos el operador >> y el operador >>>. La diferencia entre ambos es que >> inserta por la izquierda el valor que tenga el bit de signo, es decir, el bit MSB: 0 si es positivo y 1 si es negativo (extiende el bit de signo); por otro lado, el operador >>> actúa como el <<, es decir, insertando 0 (en este caso por la izquierda).

En cuanto a los otros operadores, las operaciones ~ (NOT), & (AND), | (OR) y ^ (XOR) se describen mejor mediante una tabla de verdad, que sería la que se aplica a cada par de bits del mismo peso de las dos variables o valores a comparar:

a	NOT
0	1
1	0

a	b	AND
0	0	0
0	1	0
1	0	0
1	1	1

a	b	OR
0	0	0
0	1	1
1	0	1
1	1	1

a	b	XOR
0	0	0
0	1	1
1	0	1
1	1	0

El operador AND & se suele utilizar como máscara, es decir, para averiguar si un determinado bit de un valor entero vale 1 o 0. Por ejemplo, el siguiente programa comprueba si el bit de menor peso (LSB) de la variable `n` es 1 o 0. Imaginemos que `n` vale `11` en decimal, es decir, `0b1011` en binario, que como hemos dicho antes, es indistinto qué sistema de numeración utilicemos. Al realizar la operación AND con la máscara `0b0001`, el resultado será todo 0 salvo el bit LSB que no sabemos si será 0 o 1; en este caso sería: `0b1011 & 0b0001 = 0b0001`. Vemos que el resultado no es 0, indicando que el bit LSB del valor de `n` es 1.

```
int n = 11;  // 0b1011
int mascara=0b0001;
int resultado;
...
resultado = n & mascara;
if (resultado != 0) {
   // El bit LSB es 1
} else {
   // El bit LSB es 0
}
```

2.5. Control de flujo de ejecución

Un programa es una lista de instrucciones que normalmente se ejecutan una tras otra de forma secuencial, comenzando siempre por la función `main()`. La ejecución puede saltar a otra función, que de nuevo se ejecutará secuencialmente, y retornará a la función que la invocó una vez finalizada. Sin embargo, para cualquier tarea se necesita modificar en algún momento el orden de ejecución, por ejemplo, para tomar un camino u otro en función de una condición (sentencias `if` o `switch`), o repetir un grupo de sentencias mientras se cumpla una determinada condición (bucles `for`, `while`, `do-while`).

Sentencia condicional: `if`

La sentencia `if` permite bifurcar la ejecución según una condición booleana (entre paréntesis): si se evalúa como `true` la ejecución continuará por el bloque del `if`; si se evalúa como `false`, no entrará en el bloque del `if`, y, si se ha especificado la parte opcional `else` (a la derecha), se saltará a dicho bloque.

```
if ( ... ) {

}
```

```
if ( ... ) {

} else {

}
```

Por ejemplo, para averiguar si un determinado valor de temperatura es menor que un determinado valor, y activar una variable que indique que hay que encender la calefacción podemos hacer lo de la izquierda. Para averiguar el mayor de dos números podemos hacer lo de la derecha:

```
double  temp = ...;
boolean calefaccion;
if (temp < 0.0D) {
   calefaccion = true;
} else {
   calefaccion = false;
}
```

```
int mayor (int a, int b) {
   if (a > b) {
      return (a);
   } else {
      return (b);
   }
}
```

Cuando se explicaron los operadores del lenguaje (sección 2.4) vimos que existía un operador ternario, que en ocasiones puede utilizarse en lugar de la sentencia `if`. Los 2 siguientes bloques de código son equivalentes al ejemplo anterior de la temperatura:

```
double  temp = ...;
boolean calefaccion;
calefaccion =
   ((temp < 0.0D) ? true : false);
```

```
double  temp = ...;
boolean calefaccion;
calefaccion = (temp < 0.0D);
```


En el caso de querer comprobar varias condiciones o varios valores de una variable pueden anidarse o encadenarse varias sentencias `if`. El siguiente ejemplo es una función que calcula el número de días de un mes, primero con `if` anidados (izquierda), y luego con `if` encadenados (derecha). Evidentemente, en este ejemplo no conviene el uso sentencias `if` de forma anidada (independientes); con el uso de sentencias `if` encadenadas se consigue un código más compacto y con menos sangrado [4]. En el siguiente punto se presenta la estructura `switch`, que es aun más apropiada para estos casos, aunque sólo se puede utilizar para ciertos tipos de datos primitivos: `char` o `int`.

```
// Obtener el numero de dias
// de un mes (1..12)
int numDias (int m) {
   int dias;

   if ( m == 1 ) {
      dias = 31;  // Enero
   } else {
      if ( m == 2 ) {
         dias = 28;  // Febrero
      } else {
         if ( m == 3 ) {
            ...
         } else {
            ...
            if ( m == 12 ) {
               // Diciembre
               dias = 31;
            } else {
               // Mes desconocido
               dias = -1;
            }
            ...
         }
      }
   }
   return (dias);
}
```

```
// Obtener el numero de dias
// de un mes (1..12)
int numDias (int m) {
   int dias;

   if ( m == 1 ) {
      dias = 31;  // Enero
   } else if ( m == 2 ) {
      dias = 28;  // Febrero
   } else if ( m == 3 ) {
      ...
   } else if ( m == 12 ) {
      dias = 31;  // Diciembre
   } else {
      // Mes desconocido
      dias = -1;
   }
   return (dias);
}
```

Aunque en estos ejemplos hemos escrito siempre las llaves `{..}` para cada `if` y para cada `else`, éstas sólo son necesarias cuando hay más de una sentencia; en el caso de que sólo haya una no es obligatorio. Esto se aplica, no sólo a las sentencias `if`, sino a cualquier otra estructura de control: bucles `for`, `while`, etc.

[4] Técnica utilizada en programación que consiste en desplazar cada nuevo bloque de código a la derecha varios espacios en blanco (o un carácter tabulador) con objeto de mejorar la legibilidad del código fuente, los cuales son ignorados por el compilador. De este modo, dada una llave de inicio de bloque `{` es fácil saber cuál es su correspondiente llave de fin de bloque `}`, y viceversa.

Para finalizar con la sentencia `if`, comentar que la condición puede ser una condición múltiple, es decir, varias condiciones unidas mediante los operadores condicionales && (AND), || (OR), o su negación mediante el operador ! (NOT). Cuando se explicaron estos operadores ya se mostró algún ejemplo (sección 2.4). Algunos ejemplos similares que utilizan condiciones múltiples serían los siguientes;

```
int edad = ...;
if ((edad >= 18) && (edad <= 30)) {
   // Joven
}
```

```
int edad = ...;
if ((edad < 12) || (edad > 80)) {
   // Niño o anciano
}
```

Cuando hay varias condiciones, se evalúan comenzando desde la izquierda hacia la derecha, pero puede que no lleguen a evaluarse todas las condiciones, ya que el compilador Java realiza una **evaluación en cortocircuito**, es decir, en el momento que ya se sepa el resultado de la condición global, aunque aun queden más condiciones por evaluar se detiene su evaluación. Podrían darse dos casos, o la mezcla de ambos:

1. En una serie de condiciones unidas mediante el operador && (AND), en el momento en que una de las condiciones se haya evaluado a `false`, el resultado final ya será `false`, independientemente del valor del resto de las condiciones pendientes de evaluar.

2. En una serie de condiciones unidas mediante el operador || (OR), en el momento en que una de las condiciones se haya evaluado a `true`, el resultado final ya será `true`, independientemente del valor del resto de las condiciones pendientes de evaluar.

Al evaluarse en cortocircuito, en el momento en que se de una de las condiciones anteriores ya no se sigue evaluando el resto de condiciones, para mejorar el rendimiento. Esto normalmente es transparente al programador y no sería necesario conocer que esto es así, pero si en alguna de esas condiciones que no se han llegado a evaluar, se realizase alguna asignación, puesto que no se ha evaluado tampoco se llegaría a realizar dicha asignación. Por ello, debemos evitar realizar asignaciones dentro de las mismas condiciones. Supongamos el fragmento de código siguiente, en el que vemos que no se cumple la primera condición; en estas circunstancias la segunda condición no llega a ser evaluada, no incrementándose el valor de la variable `edad` como sería de esperar:

```
int edad = 15;
...
if ((edad >= 18) && (edad++ <= 20)) {
   // Joven
}
```

Siguiendo con el ejemplo anterior de la función `numDias()`, habrá meses con 31 días, otros con 30 y uno con 28 (no hay más casos, si no tenemos en cuenta años bisiestos). Con la sentencia `if` podríamos reescribir dicha función utilizando condiciones múltiples como se muestra en el siguiente ejemplo.

```
// Numero de dias de un mes
int numDias (int m) {
   int dias;

   if ((m==1) || (m==3)  || (m==5)  || (m==7) ||
       (m==8) || (m==10) || (m==12)) {
      // Enero, Marzo, Mayo, Julio, Agosto, Octubre, Diciembre
      dias = 31;
   } else if ((m==4) || (m==6) || (m==9) || (m==11))
      // Abril, Junio, Septiembre, Noviembre
      dias = 30;
   } else if (m==2) {
      // Febrero
      dias = 28;
   } else {
      dias = -1;
   }
   return (dias);
}
```

Sentencia condicional: `switch`

En los casos en los que hay que comparar una variable o expresión con varios valores (como era el caso de la función `numDias()` del punto anterior), es más recomendable utilizar la sentencia `switch` que `if` anidados o encadenados. La estructura general de una sentencia `switch` es la de la izquierda, que sería equivalente al uso de sentencias `if` encadenadas como aparece a la derecha:

```
switch (variable) {
   case valor1:
      sentencias1;
      break;
   case valor2:
      sentencias2;
      break;
   ...
   case valorN:
      sentenciasN;
      break;
   default:
      sentenciasOTROS;
}
```

```
if ( variable == valor1 ) {
   sentencias1;
} else if ( variable == valor2 ) {
   sentencias2;
   ...
} else if ( variable == valorN ) {
   sentenciasN;
} else {
   sentenciasOTROS;
}
```

La función `numDias()` quedaría del siguiente modo utilizando una sentencia `switch`:

```
// Numero de dias de un mes
int numDias (int m) {
   int dias;

   switch (m) {
      case 1:   // Enero
         dias = 31;
         break;
      case 2:   // Febrero
         dias = 28;
         break;
      ...
      case 12:  // Diciembre
         dias = 31;
         break;
      default:
         dias = -1;
    }
    return (dias);
}
```

```
// Obtener el numero de dias
// de un mes (1..12)
int numDias (int m) {
   switch (m) {
      case 1:   // Enero
         return (31);
      case 2:   // Febrero
         return (28);
      ...
      case 12:  // Diciembre
         return (31);
      default:
         return (-1);
    }
}
```

Como se adelantó en el punto anterior, la sentencia `switch` sólo puede utilizarse para ciertos tipos de datos primitivos: `char` o `int`. Sin embargo, desde la versión Java 7 también es posible utilizar cadenas de caracteres (`String`).

Normalmente, cada claúsula `case` debe terminar con una sentencia `break` como última de sus sentencias, de lo contrario, la ejecución seguiría con el siguiente `case`, y así sucesivamente hasta encontrar un `break` o el fin de la sentencia `switch`. Es decir, en el ejemplo anterior de la izquierda, si no hubiésemos escrito las sentencias `break` y el mes fuese 2 (`m=2`), la ejecución entraría en el '`case 2`', y asignaría `dias=28`, pero al no encontrar un `break`, entraría también en el '`case 3`', '`case 4`', ... y así sucesivamente en todos, terminando en el `default` que hace la asignación `dias=-1`, que será el valor que retornará finalmente, en lugar del valor 28 que debería haber retornado.

Una alternativa al uso de `break` para este caso concreto es la que aparece en la parte derecha del ejemplo anterior, en la que se utiliza la sentencia `return`, ya que la función `numDias()` lo único que hace es retornar un valor. Cuando se encuentra la sentencia `return`, la ejecución no sólo sale del `switch` sino de la función completa, retornando el valor especificado. Esta solución, no puede utilizarse siempre, ya que después del `switch` la función podría necesitar realizar más tareas. Además, cuando una función retorna un valor, es decir, no es de tipo `void` (en este caso de tipo `int`), debemos tener la seguridad de que siempre haya una sentencia `return` del tipo apropiado, independientemente del flujo de ejecución seguido; por ejemplo, podríamos haber olvidado escribir la sentencia `return` dentro de la claúsula `default`. No obstante, el compilador detecta estos casos y nos avisa convenientemente con un error de compilación: "*missing return*".

Como es de suponer, la sentencia `switch` no permite realizar comparaciones múltiples mediante operadores condicionales como sí permitía la sentencia `if`, es decir, sólo se comparan valores concretos, uno en cada `case`. No obstante, sí que se puede implementar algo equivalente a utilizar el operador `||` (OR). En ocasiones puede que para distintos casos sea necesario ejecutar las mismas acciones, ilustrando un caso de por qué la sentencia `break` no es obligatoria tras cada `case`. Por tanto, podríamos reescribir la función `numDias()` utilizando la sentencia `switch` del siguiente modo.

```
// Numero de dias de un mes
int numDias (int m) {
   int dias;

   switch (m) {
      case 1:   // Enero
      case 3:   // Marzo
      case 5:   // Mayo
      case 7:   // Julio
      case 8:   // Agosto
      case 10:  // Octubre
      case 12:  // Diciembre
         dias = 31;
         break;
      case 4:   // Abril
      case 6:   // Junio
      case 9:   // Septiembre
      case 11:  // Noviembre
         dias = 30;
         break;
      case 2:   // Febrero
         dias = 28;
         break;
      default:
         dias = -1;
   }
   return (dias);
}
```

```
// Obtener el numero de dias
// de un mes (1..12)
int numDias (int m) {
   switch (m) {
      case 1:   // Enero
      case 3:   // Marzo
      case 5:   // Mayo
      case 7:   // Julio
      case 8:   // Agosto
      case 10:  // Octubre
      case 12:  // Diciembre
         return (31);
      case 4:   // Abril
      case 6:   // Junio
      case 9:   // Septiembre
      case 11:  // Noviembre
         return (30);
      case 2:   // Febrero
         return (28);
      default:
         return (-1);
   }
}
```

Vemos que en todos los ejemplos presentados existe una claúsula `default` al final, que se ejecuta en el caso de que no se cumpla ninguno de los casos (`case`) previstos. Aunque no es obligatoria, se aconseja escribirla siempre, para evitar que el programa aborte si se produjera un caso no previsto.

```
   switch ( ... ) {
      case 1: ...
      default:;  // Sentencia nula
   }
```


Bucles: `for`, `while`, `do-while`

Los bucles son unas estructuras de control que permiten ejecutar un bloque de código una serie de veces, mientras se cumpla una determinada condición. Cada una de las veces que se ejecutan las sentencias del bucle se le denomina **iteración**. En Java disponemos de tres sentencias para implementar bucles:

- Bucle `for`: se distinguen 3 partes, las sentencias de inicialización (separadas por comas `','`), una condición tal que mientras se cumpla se seguirá ejecutando el bucle, y las sentencias de incremento que se ejecutan justo al final de cada iteración (también separadas por comas `','`). La primera y última parte es opcional, pero si se eliminan ambas, quizás sea mejor utilizar un bucle `while`.

```
for (inicializacion; condicion; incremento) {
   sentencias;
}
```

- Bucle `while`: las sentencias se ejecutarán mientras se cumpla la condición, la cual se evalúa antes de cada iteración, por lo que las sentencias podrían no ejecutarse ninguna vez si no se cumple la condición.

```
inicializacion;
while (condicion) {
   sentencias;
   incremento;
}
```

- Bucle `do-while`: es similar al anterior, salvo que la condición se evalúa tras cada iteración, no antes, por lo que las sentencias se ejecutarán al menos una vez.

```
inicializacion;
do {
   sentencias;
   incremento;
} while (condicion);
```

Los 3 bucles son muy similares. Como ejemplo, vamos a escribir un programa que imprima los números 1..10, utilizando los 3 tipos de bucles:

```
for (int i=1; i<=10; i++) {
   System.out.println (i);
}
```

```
int i=1;
while (i<=10) {
   System.out.
       println (i);
   i++;
}
```

```
int i=1;
do {
   System.out.print
        (i);
   i++;
} while (i<=10);
```

```
1 2 3 4 5 6 7 8 9 10            (un valor en cada linea)
```


La diferencia entre los bucles `while` y `do-while` es el momento en el que se evalúa la condición. Mientras que el bucle `while` la comprueba antes de cada iteración, el bucle `do-while` la comprueba justo después. Según esto, se podría dar el caso de que con el bucle `while` no se ejecute ninguna vez las sentencias si no se cumple la condición; en cambio, con el bucle `do-while` al menos siempre se ejecutan una vez.

Aunque los ejemplos anteriores producen el mismo resultado en los 3 casos, piénsese que ocurriría si el valor inicial de la variable `i` fuese `i=20`... Respuesta: tanto con el bucle `for` como con `while` no se imprimiría nada, ya que no se cumple la condición y no entra al bucle; sin embargo, como con el bucle `do-while` las sentencias siempre se ejecutan al menos una vez, se imprimirá el valor 20 y luego terminará, puesto que no cumple la condición.

Sentencias de salto: `break`, `continue`, `return`

`break`

Cuando se explicó la estructura `switch` se presentó la sentencia `break`, que permitía salir de dicha estructura. Del mismo modo, cuando la sentencia `break` se escribe dentro de un bucle (cualquiera de los 3 bucles anteriores), la ejecución saldrá del bucle. El siguiente ejemplo debería imprimir los valores 1..10, pero al llegar al 5 termina (no llega a imprimirlo).

```
for (int i=1; i<=10; i++) {
   if (i == 5) break;  // Salida del bucle
   System.out.print (i);
}
System.out.println ("");
```

```
1 2 3 4
```

En el caso de tener bucles anidados, siempre se sale del bucle más cercano, el que contiene al `break`.

```
for (int i=1; i<=3; i++) {
   for (int j=1; j<=5; j++) {
      if (j == 3) break;  // Salida bucle j
      System.out.print ("("+i+","+j+") ");
   }
}
System.out.println ("");
```

```
(1,1) (1,2) (2,1) (2,2) (3,1) (3,2)
```

Junto con la sentencia `break` se puede especificar una etiqueta como destino del salto, pudiendo entonces realizar saltos multinivel, es decir, terminando no sólo el bucle más interior sino cualquier otro.

```
for (int i=1; i<=3; i++) {
   for (int j=1; j<=5; j++) {
      if (j == 3) break fin;  // Salto
      System.out.print ("("+i+","+j+") ");
   }
}
fin: System.out.println ("");
```

```
(1,1) (1,2)
```


`continue`

La sentencia `continue` se puede escribir dentro un bucle para finalizar la iteración actual, pero siguiendo dentro del bucle. El siguiente ejemplo imprime los valores 1..10 pero se salta la iteración 5.

```
for (int i=1; i<=10; i++) {
   if (i == 5) continue;
   System.out.print (i);
}
System.out.println ("");
```

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

Al igual que la sentencia `break`, en caso de utilizarse dentro de bucles anidados afectará al bucle más interior, salvo que se haya especificado una etiqueta como destino, en cuyo caso podría afectar a otro bucle.

IMPORTANTE: el uso de etiquetas con las sentencias `break` o `continue` no se recomienda, pues rompe los esquemas de la programación estructurada.

`return`

Con esta sentencia se pone fin a la ejecución de la función (método) actual, retornando a la función que la llamó, justo a continuación de su llamada. Todos los métodos (salvo los de tipo `void`) deben retornar un valor del tipo apropiado, lo cual se hace en la misma sentencia `return`.

```
// Retornamos desde funcion void
return;
```

```
// Retornamos desde funcion int
return (30);
```

Bucle `for-each`

En la versión Java 5 se introdujo un nuevo tipo de bucle, el bucle `for-each`, muy útil para recorrer *arrays* y colecciones (`Collection`), conceptos que se explicarán más adelante. De momento se describe en esta parte para tenerlo como referencia, aunque de momento no lo utilizaremos.

El siguiente fragmento de código declara e inicializa un array de 7 elementos y luego lo recorre mostrando su contenido. Aunque podría hacerse con los bucles tradicionales, se puede comprobar que para estos casos el código se simplifica bastante.

```
String diasSemana[] = {"Lunes", "Martes", "Miercoles", "Jueves",
                       "Viernes", "Sabado", "Domingo"};
// Recorremos todos los elementos y los mostramos
for (String s : diasSemana) {
   System.out.println (s);
}
```

En los capítulos correspondientes a *arrays* y colecciones (`Collection`) se mostrará de nuevo algún ejemplo del bucle `for-each`.

2.6. Argumentos desde la línea de órdenes

Es posible pasar valores a un programa cuando se ejecuta desde línea de órdenes (consola). La forma de gestionar los argumentos pasados al programa en Java es parecida a como los gestiona C, aunque hay alguna diferencia. Todos los programas Java han de definir obligatoriamente una función `main()`, que será el punto de entrada al programa:

```
public class MiClase {
   public static void main (String args[]) {
      ...
   }
}
```

El parámetro `args` que recibe la función `main()` es una matriz o array de cadenas de caracteres (objetos `String`). Cuando se ejecuta el programa desde la línea de órdenes, este array se rellena automáticamente con los valores de los argumentos que especificamos a continuación del nombre del programa. Esta matriz se trata de una matriz cualquiera, por lo que podremos acceder a sus parámetros por medio de un índice, y saber el número de elementos mediante la propiedad `length`. Aunque aun no se ha explicado el uso de arrays y de objetos `String`, dado su parecido al lenguaje C podemos escribir un primer ejemplo. Vamos a escribir un programa que simplemente imprima por pantalla los argumentos recibidos:

```
// Archivo: Argumentos.java
public class Argumentos {
   public static void main (String[] args) {
      // Obtenemos el numero de argumentos recibidos
      System.out.println ("Numero de argumentos: " + args.length);
      // Recorremos todos los argumentos recibidos para imprimirlos
      for (int i=0; i < args.length; i++) {
         System.out.println ("El argumento " + i + " es:" + args[i]);
      }
   }
}
```

Compilamos el programa desde la ventana del DOS en Windows, y lo ejecutamos especificando varios argumentos (separados por espacio en blanco), los cuales son tratados como cadenas de caracteres `String` dentro del programa, aunque fueran valores numéricos (como el último de ellos):

```
C:\ javac Argumentos.java
C:\ java Argumentos
Numero de argumentos: 0
C:\ java Argumentos primero segundo tercero 1234
Numero de argumentos: 4
El argumento 0 es: primero
El argumento 1 es: segundo
El argumento 2 es: tercero
El argumento 3 es: 1234
```


Comparativa con el lenguaje C

Vamos a comparar en detalle la forma de gestionar los argumentos del lenguaje Java y del lenguaje C. Aunque se hace de forma muy similar, hay ciertas diferencias:

1. En C, la función `main()` recibe 2 argumentos (o ninguno): el número de argumentos pasados desde línea de órdenes (`int`) y un array de cadenas de caracteres (`char*`), aunque también podría no escribirse nada en caso de no ser necesario utilizar los argumentos. Por otro lado, Java recibe únicamente el array de cadenas (`String`), ya que a partir de él se puede obtener el número de argumentos (`args.length`), y además, es obligatorio, es decir, aunque no se utilice siempre debe especificarse.

```c
// Lenguaje C
int main () { ... }
int main (int argc, char *argv[]) {
    ...
}
```

```java
// Lenguaje Java
public static void main (String
    args[]) {
    ...
}
```

2. En C, los argumentos empiezan con el elemento `argv[1]` del array de argumentos, ya que el elemento `argv[0]` contiene siempre el nombre del programa; en Java, los argumentos desde línea de órdenes empiezan en el elemento `args[0]`, ya que no se incluye el nombre del programa.

El primer ejemplo anterior puede reescribirse en C para compararlo con Java:

```c
// Archivo: argumentos.c
void main (int argc, char *argv[]) {
    // Obtenemos el numero de argumentos recibidos
    printf ("Numero de argumentos: %d \r\n", argc-1);
    // Recorremos todos los argumentos recibidos para imprimirlos
    for (int i=1; i < argc; i++) {
        printf ("El argumento %d es: %s \r\n", i, argv[i]);
    }
}
```

Lo más destacable es que en lenguaje C hay que restar 1 al valor indicado por `argc` a la hora de mostrar el número de argumentos, ya que el primero siempre es el nombre del programa (como se ha dicho antes), y por esta misma razón, en el bucle `for` comenzamos por el valor 1, es decir, se comienza con el argumento `argv[1]`, no con `argv[0]`.

Ejemplos

A continuación se muestra unos ejemplos de paso de argumentos desde línea de órdenes a los programas. Se hace uso de alguna función o técnica que, aunque aun no se ha explicado, por su similitud con el lenguaje C se espera que se puedan comprender (como es el caso de los arrays).

Ejemplo 1

Vamos a escribir un programa que reciba desde la línea de órdenes un único argumento, que será un valor numérico entero `'n'`, y nos calcule la suma de los `'n'` primeros números naturales, mostrándola por pantalla. En este caso hay que tener en cuenta que los argumentos que se reciben desde línea de órdenes son cadenas de caracteres (`String`), y debemos convertir la cadena recibida a un valor entero, para lo que utilizamos la función `Integer.parseInt()`. Tal cual se muestra el código, el programa generará un error en el caso de que el argumento recibido no se un valor entero válido; más adelante se presentarán técnicas para solventar este problema (gestión de excepciones).

```
1  // Archivo: SumaNaturales.java
2  public class SumaNaturales {
3
4     public static void main (String args[]) {
5        int n, suma;
6
7        if (args.length != 1) {
8           System.out.println ("Hay que pasar un argumento");
9           System.exit (1);
10       }
11       n = Integer.parseInt (args[0]);
12       for (int i=1; i<=n; i++) {
13          suma += i;
14       }
15       System.out.println ("La suma de los " + n +
16           " primeros numeros naturales es " + suma);
17    }
18 }
```

Compilamos y ejecutamos el ejemplo anterior:

```
C:\ javac SumaNaturales.java
C:\ java SumaNaturales
Hay que pasar un argumento
C:\ java SumaNaturales 5
La suma de los 5 primeros numeros naturales es 15
```


Ejemplo 2

Para finalizar, vamos a escribir un programa que reciba una cadena de texto como argumento desde línea de órdenes, que se supone que está escrita en minúsculas y la transformará a mayúsculas. El procedimiento es recorrer cada carácter de la cadena mediante un bucle, y restarle 32, que es la diferencia entre el código Unicode de la letra `'a'` (97) y el de la letra `'A'` (65). Por simplicidad, no comprobaremos si cada uno de los caracteres a transformar es una letra en minúsculas.

```
1  public class Mayusculas {
2
3     public static void main (String args[]) {
4        String cadenaMinusculas, cadenaMayusculas;
5        char   letra;
6        int    longitud;
7
8        if (args.length == 0) {
9           System.out.println ("Hay que pasar un argumento");
10          System.exit (1);
11       }
12       cadenaMayusculas = "";
13       cadenaMinusculas = args[0];
14       longitud = cadenaMinusculas.length();
15       for (int i=0; i<longitud; i++) {
16          // Obtenemos letra i-esima con el metodo charAt de los String
17          letra = cadenaMinusculas.charAt (i);
18          // Convertimos a mayusculas (restando 32)
19          letra = (char) (letra - 32);
20          // Concatenamos a la cadena
21          cadenaMayusculas = cadenaMayusculas + letra;
22       }
23       System.out.println (cadenaMinusculas + " --> " + cadenaMayusculas);
24    }
25 }
```

Un ejemplo de ejecución podría ser:

```
C:\ javac Mayusculas.java
C:\ java Mayusculas
Hay que pasar un argumento
C:\ java Mayusculas hola
hola --> HOLA
```


GLOSARIO DE ACRÓNIMOS

Siglas	Significado
ANSI	American National Standards Institute
AOT	Ahead-Of-Time (técnica de compilación)
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AWT	Abstract Window Toolkit
CISC	Complex Instruction Set Computing
DBMS	DataBase Management System
DLL	Dynamic Linked Library
ETSI	European Telecommunications Standards Institute
GNU	GNU Not Unix
GPL	GNU General Public License
GUI	Graphical User Interface
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronic Engineer
ISO	International Standards Organization
JAR	Java ARchive
J2SE	Java 2 Platform, Standard Edition (~ Java SE)
J2EE	Java 2 Platform, Enterprise Edition (~ Java EE)
J2ME	Java 2 Platform, Micro Edition (~ Java ME)
JCP	Java Community Process
JDBC	Java DataBase Conectivity
JDK	Java Development Kit (~ Java 2 SDK y J2SDK)
JFC	Java Foundation Classes
JIT	Just-In-Time (técnica de compilación)
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface
JRE	Java Runtime Environment
JVM	Java Virtual Machine
KNI	K Native Interface (~ JNI)
ODBC	Open Database Connectivity
OSI	International Organization of Standarization
PC	Personal Computer

PDA	Personal Digital/Data Assistant
RDBMS	Relational DataBase Management System
RISC	Reduced Instruction Set Computing
ROM	Read Only Memory
SDK	Standard Development Kit
SQL	Structured Query Language
SRAM	Static Random Access Memory
URL	Uniform Resource Locator
UTF-16	Unicode Transformation Format (16-bit)
WORA	Write Once, Run Anywhere