Capítulo 2

Cuestiones básicas de sintaxis

En esta práctica se trabaja con los elementos de Java que permiten crear *programas estructurados*. Se explican los conceptos relacionados con variables, operadores, expresiones y sentencias. Muchas partes le recordarán a otros lenguajes (sobre todo si conoce C). No es extraño, la sintaxis de Java evolucionó desde la de este lenguaje. Sin embargo, existen diversas diferencias con C. No crea que ya lo sabe todo y lea con cuidado.

2.1. Variables y tipos

Antes de nada cabe destacar que en Java existen diferentes grupos de variables con características similares, pero no iguales son:

- Las propiedades, variables de instancia o campos no estáticos (depende de la bibliografía que use puede leer todos esos nombres). Como hemos visto los objetos almacenan sus propiedades en variables que se definen al principio de la clase. Por cada objeto de una clase que se instancia existirá en memoria espacio para cada una de estas variables.
- Las variables de clase o campos estáticos. Son las propiedades declaradas como `static`. Siempre que se use una clase existirá exactamente una copia de esta variable independientemente del número de objetos de la clase que se instancien.
- Variables locales. Los métodos pueden necesitar variables loacales para almacenar informaciones parciales. Estas variables se declaran dentro del bloque del método y son sólo visibles dentro del método.
- Parámetros. Los métodos, incluyendo el constructor, pueden declararse con parámetros de llamada.

Java diferencia entre los distintos tipos de variables por el lugar donde se declaran. En cualquier caso, Java es fuertemente tipado, por lo que debe declarar cualquier variable antes de su uso. Todos los grupos de variables se declaran siguiendo la misma sintaxis[1]:

```
[modificadores] tipo nombre_variable [= valor_inicial];
```

[1] Lo que se indica entre corchetes es opcional

Donde los modificadores incluyen los que se estudiaron en la práctica anterior.

En adición a variables, en Java también podemos tener constantes. Se declaran como una variable normal y corriente, pero tienen que inicializarse en el momento de la declaración. Para indicar su caracter de constante se les antepone el modificador `final`. En ellas se modifica ligeramente la convención de nombrado que se comentó en la práctica anterior. Sus nombres se escriben en mayúsculas y se emplea el caracter _ para separar diferentes palabras en caso de que el nombre sea de más de una palabra. Por ejemplo, la constante *numero pi* se podría declarar como `public static final float NUMERO_PI = 3.1516`.

2.2. Tipos de datos primitivos

Un tipo en Java determina tanto el tamño en memoria del dato como las operaciones que se pueden aplicar sobre la variable. En Java se emplean los siguientes tipos básicos:

- `byte`: 8 bits que codifican un valor entero en complemento a 2 (rango de valores desde -128 a 127).
- `short`: 16 bits que codifican un valor entero en complemento a 2 (rango de valores desde -32768 a 32767).
- `int`: 32 bits que codifican un valor entero en complemento a 2 (rango de valores desde -2147483648 a 2147483647).
- `long`: 64 bits que codifican un valor entero en complemento a 2 (rango de valores desde -9223372036858775808 a 9223372036854775807).
- `float`: 32 bits que codifican un valor real en coma flotante de precisión simple (standard 754 de IEEE). Como todos los números en coma flotante no es recomendable usarlo en cálculos que necesiten valores exactos ni en comparaciones de igualdad.
- `double`: 64 bits que codifican un valor real en coma flotante de doble precisión (standard 754 de IEEE). Como todos los números en coma flotante no es recomendable usarlo en cálculos que necesiten valores exactos ni en comparaciones de igualdad.
- `boolean`: Un valor booleano tiene sólo dos valores posibles `true` y `false`. Representan un bit de información aunque su tamaño en memoria no está definido.
- `char`: Un char son 16 bits que codifican caracteres Unicode. El valor mínimo es '\u0000' (o 0) y el máximo '\uffff' (o 65535).

Aunque no es un tipo básico puede considerarse también el tipo `String`. Este tipo está definido en la clase `java.lang.String`. Se trata en realidad de un objeto pero de características muy concretas. Son valores inmutables. Java crea un valor de tipo String para almacenar cadenas de caracteres que encuentra entre caracteres `''`. Así, podemos hacer cosas como `String s=''Hello World!''`.

2.2.1. Valores por defecto

Si estamos declarando una propiedad de una clase siendo su tipo un tipo básico, Java les asigna un valor por defecto. Los valores son 0 para los tipos que almacenan enteros, 0.0 para los reales, '\u0000' para los char, `null` para los String y `false` para los boolean. Se considera, sin embargo, una mala práctica confiar en la iniciación por defecto.

Las variables locales funcionan de un modo totalmente diferente. El compilador no inicializa nunca sus valores. Afortunadamente, el compilador nos avisa en el caso de tener variables locales no inicializadas.

2.2.2. Literales

En un programa Java también nos podemos encontrar valores literales. Son valores que se compilarán directemente en el código sin precisar nigún cálculo.

- Un literal entero es un valor entero. Si se escribe simplemente el valor se interpreta que es de tipo `int` (a no ser que no quepa en una variable `int` en cuyo caso será un literal `long`). Si finaliza en `L` o `l` se interpreta que es un valor `long` (aunque se pueden usar ambas formas se recomienda la mayúscula, pues `l` puede confundirse con `1`. Los literales enteros pueden escribirse en base decimal, en base hexadecimal (anteponiendo `0x` sin espacio) al valor o en binario (anteponiendo en este caso `0b`).
- Un literal en coma flotante es un valor con `.` decimal o un valor entero finalizado en `D`, `d`, `F` o `f`. Por defecto se entiende que el valor es double. Para indicar que es float se finaliza con `F` o `f`.
- Los literales de tipo caracter contienen caracteres Unicode (UTF-16). Pueden escribirse directamente o mediante secuencias de escape Unicode. Así para escribir el caracter C con circunflejo podemos poner 'u0108'. Java también soporta otras secuencias de escape como: \b (backspace), \t (tabulador), \n (salto de línea), \r (retorno de carro), \"(doble quote), \' (single quote) y \\(backslash).
- Los literales de tipo String pueden también contener los mismos caracteres que los `char`. También pueden contener el valor `null`. Valor que representa *no hay nada*.

A la hora de escribir literales numéricos se admite utilizar el caracter '_' como separador. Puede usarse para facilitar la lectura de números grandes. `123_456` es el valor 123456. Sólo puede aparecer entre dos dígitos

2.3. Tablas (arrays)

Una tabla en Java es un contenedor que contiene un número fijo de valores del mismo tipo. El tamaño de la tabla se fija cuando se crea. Las tablas son un tipo de datos muy distinto porque son también objetos. Para declarar una propiedad de tipo tabla basta con poner `[ ]` detrás del tipo, es decir

```
int[ ] unaTabla;
```

declara una variable, llamada `unaTabla` de tipo tabla de enteros. La instrucción anterior se limita a declarar un nombre para una variable que se referirá a valores de tipo tabla de enteros. Aún no se ha creado la tabla. Como las tablas son objetos deberemos instanciar el objeto para crearlo. Es en ese momento cuando se fija el tamaño. Es decir, una vez declarada la tabla, para que empiece a existir un objeto de ese tipo debemos hacer:

```
unaTabla = new int[10];
```

Nótese que estamos creando un nuevo objeto (por usar `new`) de tipo tabla de enteros con capacidad para 10 enteros.

Una vez instanciada la tabla podemos acceder a cualquiera de sus posiciones indicando el nombre de la variable y, entre corchetes, el desplazamiento desde la primera posición que necesitamos. Es decir, si queremos, por ejemplo, cargar el valor 10 en la primera posición de la tabla debemos hacer `unaTabla[0] = 10;`.

También podemos instanciar la tabla de modo implícito. Basta con informar de los elementos de la tabla en su iniciación. El tamaño del array será el adecuado para los valores que se le indiquen. Así

```
int [ ] unaTabla ={
    1, 10, 100, 1000, 10000, 100000, 1000000
};
```

crea una tabla de 7 posiciones con el valor 1 en la primera posición el 10 en la segunda ...

Java acepta también tablas multdimensionales. Para declarar una tabla multidimensional basta con poner un par de corchetes por cada dimensión que se desee. Obviamente habrá que emplear un índice por cada dimensión para acceder a un elemento. En Java un array multidimensinal es sólo una tabla cuyos elementos son tablas. Es decir, AL CONTRARIO QUE EN OTROS LENGUAJES como C es posible que cada fila sea de tamaño distinto. Pruebe el siguiente código:

```
class MultiDimArrayDemo {
    public static void main(String[] args) {
        String [][] nombres = {
            {''Señor'', ''Señorita'', ''Señora''},
            {''Perez'', ''Gomez''}
        };
        System.out.println(nombres[0][0] + nombres[1][0]);
    }
}
```

comprobará que todo funciona correctamente a pesar de que no hemos definido una tabla cuadrada.

Como hemos dicho, las tablas son un on objeto definido en `System`. Como objeto tiene un conjunto de propiedades definidas. Por ejemplo, con `unaTabla.length` podemos ver el tamaño de una tabla. También tenemos un método para copiar tablas de modo eficiente. System declara el método `public static void arraycopy(Object src, int srcPos,`

`Object dest, int destPos, int length)` que copia `length` elementos de `src` a partir de la posición `srcPos` en `dest` a partir de la posición `destPos`.

Una grán ventaja de la gestión que hace Java de las tablas respecto a otros lenguajes es que Java chequea en tiempo de ejecución los accesos a las posiciones de una tabla asegurando que nunca accedamos fuera de la memoria de la tabla. Compruébelo modificando el ejemplo anterior de modo que accedamos a alguna posición fuera de la tabla.

Lo que ha visto es una excepción, no un error. Java tiene un mecanismo de gestión de excepciones para reaccionar ante estas situaciones.

2.4. Operadores

El operador asignación. El operador asignación en Java es `=`. Tiene que estar situado entre un operando y una expresión que calcule un valor del tipo adecuado. Su efecto es calcular el valor de la expresión y cargarlo en el operando.

El operando puede ser una variable, una posición de una tabla, un objeto, etc.

Operadores aritméticos En Java puede emplear los operadores aritméticos habituales con su significado habitual. Es decir dispone de los operadores binarios: `+` (suma), `-` (resta), `*` (producto) y `/` (división). También puede usar el operador `%` para calcular el resto de una división entera.

Todos estos operadores pueden asignarse con en operador de asignación para escribir código más rápido. Así podemos escribir `x = x*2` como `x *= 2` (igual con el resto de operadores).

Java también dispone de los operadores unarios habituales: `+` para indicar que un valor es positivo (no se usa), `-` para cambiar el signo del valor que vaya por detrás.

Existen otros dos operadores unarios que pueden ponerse delante o detrás de un operando. Son el operador de incremento `++` y el de decremento `--`. Su funcionamiento requiere alguna explicación. Escriba el siguiente programa:

```
class DemoDeUnarios {
    public static int valor = 1;
    public static void main(String [] args){
        System.out.println(valor);
        System.out.println(valor++);
        System.out.println(valor);
        System.out.println(++valor);
        System.out.println(valor);
        System.out.println(valor--);
        System.out.println(valor);
        System.out.println(--valor);
        System.out.println(valor);
    }
}
```

El resultado le puede parecer extraño. El operador de incremento efectivamente incrementa el valor sobre el que se aplica, pero hay una diferencia entre ponerlo antes o despues del operando al que afecta. Si el valor que se está calculando se va a usar para algo, poner

`++` antes del operando hace que primero se calcule el nuevo valor y luego se use. Ponerlo detras significa que primero se usa el valor actual y luego se incrementa. Obviamente el operador decremento funciona de igual modo. Tenga cuidado con este comportamiento.

Operadores relacionales y lógicos. En Java se pueden comparar valores empleando los operadores `==` (igual que), `!=` (diferente), > (mayor que), `>=` (mayor o igual que), < (menor que) y `<=` (menor o igual que). Tenga mucho cuidado en no confundir la asignación `=` con el operador de comparación `==`.

También puede operar valores booleanos con operadores lógicos: `&&` es el operador lógico y, mientras que || es el operador lógico o. También dispone del operador lógico unario no `!`. Los operadores lógicos funcionan de modo que no siempre es necesario calcular expresiones completas. Si leemos un operador lógico o cuando la expresión es verdadera, se salta el siguiente operando. Lo mismo ocurre si llegamos a un operador lógico y con un valor falso.

Java dispone de otros dos operadores lógicos. Uno es el operador ternario `? :`. Este operador puede verse como una forma de programar un if-then-else en Java. La expresión `i = a>b?a:b;` asigna a `i` el máximo valor entre `a` y `b`, Es decir, este operador comprueba la expresion anterior al simbolo `?`. Si es verdadera su resultado es lo que indique lo que está antes del símbolo `:`, si es falsa, lo que indique lo de detrás del `:`.

El último operador lógico que proporciona Java es el operador `instanceof`. Se emplea para comprobar si un objeto es una instancia de una clase o subclase o si implementa una determinada interfaz (ya estudiaremos estas dos últimas cosas). Debe recordarse cuando use este operador que `null` no es instancia de nada.

Operadores a nivel de bit y de desplazamiento: En Java tenemos operadores que trabajan directamente con bits. El operador << desplaza a la derecha, manteniendo el signo, un patrón de bits el número de espacios que indique el segundo operando. Por su parte >> desplaza hacia la derecha, manteniendo el signo. Hay otro operador >>> que desplaza hacia la derecha completando con ceros (es decir, puede modificar el signo).

Hay otros operadores que afectan a bits. El operador unario `~` invierte todos los bits del operando al que se aplica. El binario `&` hace una operación lógica y bit a bit, | hace una operación o bit a bit y `^` hace una operación o exclusivo bit a bit.

Todos estos operadores también pueden combinarse con el operador de asignación.

El operador de concatenación +: El tipo `String` utiliza el operador `+` como operador de concatenación de cadenas. Es decir, podemos escribir expresiones como `String s = ''Hola '' + ''Mundo!''`. El resultado es que la variable s queda cargada con el valor `''Hola Mundo!''`. Este operador puede aparecer en cualquier sitio, por ejemplo, podemos escribir también `System.out.println(''Hola '' + ''Mundo!'')`. El resultado será que por pantalla aparezca el conocido mensaje.

2.4.1. Precedencia de operadores

Java dispone de multitud de operadores. A veces es complicado saber cómo se va a evaluar una expresión, para no llegar a errores conviene tener presente la siguiente tabla de precedencias. Cuanto más arriba aparece un operador mayor es su prioridad:

Operadores	Precedencia
Postfijos	`expr++ expr--`
Unarios	`++expr --expr + - ~ !`
Multiplicativos	`* / %`
Aditivos	`+ -`
Desplazamientos	`<< >> >>>`
Relacionales	`> < <= >= instanceof`
(des)Igualdades	`== !=`
Y bit a bit	`&`
O exclusivo bit a bit	`^`
O bit a bit	`|`
Y lógico	`&&`
O lógico	`||`
Ternario	`? :`
Asignaciones	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

Teniendo en cuenta esta tabla puede escribir expresiones correctas. Sin embargo, pueden ser difíciles de leer. En muchos caso se prefiere dejar claro el código utilizando paréntesis.

2.5. Sentencias

Una sentencia es una unidad completa de ejecución. Algunas son tan simples como una expresión finalizada con un `;` (se denominan sentencias de expresión). Por ejemplo, son sentencias si finalizan en `;`

- Las asignaciones;
- Cualquier uso de los operadores incremento y decremento;
- La invocación de un método;
- Las expresiones de instanciación de objetos;

Pero en Java hay otros tipos de sentenicas, las sentencias declarativas y las de control de flujo. Las declarativas consisten en la declaración de una variable (con o sin inicialización) y han aparecido ya múltiples veces. Las de control de flujo dirigen la ejecución del código de los métodos. Son las sentencias siguientes:

Los bloques: Recordamos que un bloque es un conjunto de sentencias rodeadas por llaves. Un bloque no es más que una composición secuencial de sentencias. Es decir, en un bloque se comienza a ejecutar la primera sentencia, cuando se acaba se ejecuta la segunda y así hasta que llegamos al final del bloque.

La sentencia alternativa *if - then*: Esta sentencia es la típica de todos los lenguajes de programación. Se escribe

```
if (expresion booleana) sentencia;
```

siendo el `;` necesario sólo si la sentencia no es un bloque. Cuando se ejecuta se evalúa la expresión booleana (que debe ir forzosamente entre paréntesis). Si el resultado es `true` se ejecuta la sentencia, si es `false` no se hace nada.

La sentencia alternativa *if - then - else*: Esta sentencia, también habitual en los lenguajes de programación. Se escribe

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

siendo el `;` necesario sólo si la sentencia2 no es un bloque. Cuando se ejecuta se evalúa la expresión booleana (que debe ir forzosamente entre paréntesis). Si el resultado es `true` se ejecuta la sentencia1, si es `false` se ejecuta la sentencia 2.

La sentencia alternativa `switch`: Esta sentencia permite elegir entre un conjunto de sentencias en función del resultado de evaluar una expresión. La expresión debe evaluarse en un tipo `byte`, `short`, `char`, `int`, un enumerado, String o algunas clases como `Byte`, `Short`, `Character` o `Integer`. La sintaxis de esta sentencia es

```
switch (expresion) {
    case valor1: Sentencia1; break;
    case valor2: Sentencia2; break;
    ...
    case valorn: SentenciaN; break;
    default: SentenciaNMas1; break;
}
```

Cuando se ejecuta se evalúa la expresión (que debe ir forzosamente entre paréntesis) y se compara con el `valor1`, si coinciden se ejecuta la `Sentencia1`. Si no coinciden se compara con el `valor2` .... Si el valor calculado no coincide con ninguno de los que aparecen en las partes `case` se ejecuta la sentencia del `default`. Es importante tener en cuenta el `break`. En esta sentencia, tras saltar a ejecutar una sentencia se ejecutan todas las siguientes hasta encontrar un `break`, por lo tanto es necesario para no ejecutar el código del siguiente caso. En el `default` no es realmente necesario

Cabe destacar que podemos tener diversos valores indicando que se salte a la misma sentencia. Para ello debemos poner `case valor1 case valor2 ... case valorn: Sentencia; break;`.

La sentencia iterativa `while`: Esta sentencia permite ejecutar un bloque de cófigo mientras cierta condición se mantenga cierta. Su sintaxis es:

```
while (expresion booleana) bloque
```

En la ejecución primero se evalúa la expresión booleana y se es cierta se ejecuta el bloque. Al finalizar se vuelve a evaluar la expresión booleana y si es cierta vuelve a ejecutarse el bloque. Se finaliza cuando se avalua la expresion a `false`.

La sentencia iterativa `do - while`: Esta sentencia es muy similar a la anterior, pero asegura la ejecución del bloque al menos una vez. Su sintaxis es:

```
do bloque while (expresion booleana);
```

En la ejecución primero se ejecuta el bloque; después se evalúa la expresión booleana y si es cierta se ejecuta el bloque de nuevo. Al finalizar se vuelve a evaluar la expresión booleana y si es cierta vuelve a ejecutarse el bloque. Se finaliza cuando se avalua la expresion a `false`.

La sentencia iterativa `for`: Esta sentencia permite iterar un bloque sobre un rango de valores. Se puede escribir de dos modos diferentes. El primero, muy similar al `for` de C es:

```
for (iniciacion; terminacion; incremento) bloque
```

Cuando ejecutamos esta sentencia, en primer lugar se ejecuta el código que aparezca en la parte de iniciacion. Posteriormente se comprueba la expresión de terminación. Si es falsa se sale de la sentencia, pero si es cierta se ejecuta el bloque. Tras ejecutar el bloque se ejecuta el incremento y se vuelve a comprobar la terminación. Si es falsa e sale de la sentencia, pero si es cierta ...

Es muy habitual que esta sentencia se escriba de modo similar a `for (int i=1; i<11;i++){...}`. Nótese que se ha declarado la variable que controla el bucle dentro de la iniciación. Esto suele ser muy cómodo y evita errores. El ámbito de la variable así declarada es el bloque del `for`.

Cuando trabajamos con tablas y colecciones hay otra forma de escribir este bucle. La sintaxis es:

```
for (variable: variableArray) bloque
```

En este caso, la `variable` recorre todas las posiciones de la `variableArray`.

Las sentencias `continue` **y** `break` Estas dos sentencias alteran el funcionamiento de las sentencias iterativas. La sentencia `continue` hace que se rompa la ejecución secuencial de código y se vaya a comprobar la condición de salida del bucle. La sentencia `break` finaliza la ejecución del bucle.

La sentencia `break` puede ser un tanto más compleja. Si tenemos múltiples bucles anidados podemos poner una etiqueta (`etiqueta1: for` antes de uno de ellos. Si dentro de cualquiera de los bucles anidados aparece la sentencia `break etiqueta1` se finalizará el bucle etiquetado (puede que ello suponga salir de muchos bucles internos).

La sentencia `return`: Esta sentencia finaliza la ejecución de un método. Tiene dos formas. Simplemente `return` o `return valor`. En el segundo caso, el método devuelve el valor indicado por la sentencia `return`. Obviamente el tipo del valor devuelto debe ser compatible con el que se declaró en la cabecera del método.

2.6. Prácticas a desarrollar

1. **Iniciación de variables** Conviene familiarizarse con los errores de Java para entenderlos cuando programe. Cree una clase ejecutable que contenga una propiedad estática de tipo entero. El método main debe limitarse a escribir por pantalla el

valor de la propiedad. No inicialice la propiedad. Compile y ejecute el programa. Posteriormente haga que el método main contenga una variable local no inicializada e imprima su valor. Trate de compilar y compruebe el error. ¿Entiende la diferencia en función de lo que ha leído?. Si no es así, pregunte.

2. **Leyendo desde la entrada stándard** Hasta ahora hemos visto dos formas de comunicarse con el usuario. Java tiene muchas. En esta práctica va a crear un programa que utiliza un objeto `Scanner` para ello (la clase Scanner se define en la librería `java.util`, en http://docs.oracle.com/javase/7/docs/api/ puede ver su documentación completa). Esta clase nos permite leer datos de diversas fuentes, como la entrada stándar (System.in) teniendo en cuenta su tipo. En el caso del ejercicio se solicitan enteros. Veamos como podemos implementar una clase (realmente, poco orientada a objetos) que realice esta función.

```
/**
 * Esta clase es un ejemplo de como leer datos
 * de la entrada standard
 */
import java.util.Scanner;
public class EjLectorDeEntrada {
    public static void main(String[] args){
        /* Instanciamos un objeto Scanner conectado con
         * la entrada standard */
        Scanner scan = new Scanner(System.in);
        System.out.print(''Introduce una cadena de texto: '');
        // Invocamos el metodo de Scanner para leer String
        String linea = scan.nextLine();
        System.out.println(''La linea escrita es: '' + linea);
        System.out.print(''Introduce un numero entero: '');
        // Invocamos el metodo de Scanner para leer int
        int num = scan.nextInt();
        System.out.println(''El numero escrito es: '' + num);
    }
}
```

3. **Alternativas e iteraciones**: Escriba un programa que, utilizando asteriscos, genere un triángulo isósceles o un cuadrado. La forma a construir así como las dimensiones vendrán dadas en forma de dato por parte del usuario (que indicará el número de líneas a visualizar). El margen izquierdo de la forma lo determina la última línea de la forma. Es decir, la ejecución del programa debe dejar en la pantalla lo siguiente:

```
Indique el numero de lados de la figura que desea visualizar
( 3 o 4): 3
Introduzca el numero de lineas de la figura: 4
    *
   * *
  * * *
 * * * *
```

Obviamente, este programa comienza leyendo datos del usuario. Haga que se solicite el número de lados hasta que sea correcto. Imprimir el cuadrado es un simple bucle que escriba el número de líneas indicadas. Cada línea es un bucle que escribe un asterisco seguido de un espacio. Para imprimir el triángulo puede usar también un bucle de bucles. El interno debe escribir un número de espacios en blanco que depende de la línea seguido de un número de asteriscos seguidos de espacio que también depende de la línea.

4. **Factorial** Utilice los tres métodos de comunicación con el usuario vistos hasta ahora para hacer tres programas que calculen el factorial de un valor.