Guía práctica de estudio 02: Fundamentos y sintaxis del lenguaje

Elaborado por:
M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:
M.C. Alejandro Velázquez Mena

Guía práctica de estudio 02:

Fundamentos y sintaxis del lenguaje

Objetivo:

Implementar programas utilizando:

- Tipos de datos, variables, constantes.
- Expresiones (operadores, declaraciones, etc.)
- Estructuras de control de flujo (if/else, switch, for, while, etc.)

Introducción

Los lenguajes de programación tienen elementos básicos que se utilizan como bloques constructivos, así como reglas para que esos elementos se combinen. Estas reglas se denominan **sintaxis del lenguaje**. Solamente las instrucciones sintácticamente correctas pueden ser interpretadas por la computadora y los programas que contengan errores de sintaxis son rechazados por la máquina.

La **sintaxis** de un lenguaje de programación se define como el conjunto de reglas que deben seguirse al escribir el código fuente de los programas para considerarse como correctos para ese lenguaje de programación.

Los elementos básicos constructivos de un programa son:

- Palabras reservadas (propias de cada lenguaje).
- Identificadores (nombres de variables, nombres de funciones, nombre del programa, etc.)
- Caracteres especiales (alfabeto, símbolos de operadores, delimitadores, comentarios, etc.)
- Expresiones.
- Instrucciones.

Además de estos elementos básicos, existen otros elementos que forman parte de los programas, cuya comprensión y funcionamiento será vital para la correcta codificación de un programa, por ejemplo, tipos de datos y estructuras de control de flujo.

Los **tipos de datos,** hacen referencia al tipo de información que se trabaja, donde la unidad mínima de almacenamiento es el dato, también se puede considerar como el rango de valores que puede tomar una variable durante la ejecución del programa. Un tipo de datos define un conjunto de valores y las operaciones sobre estos valores.

Casi todos los lenguajes de programación explícitamente incluyen la notación del tipo de datos, aunque lenguajes diferentes pueden usar terminologías diferentes. La mayor parte de los lenguajes de programación permiten al programador definir tipos de datos adicionales, normalmente combinando múltiples elementos de otros tipos y definiendo las operaciones del nuevo tipo de dato.

Las **estructuras de control** permiten modificar el flujo de ejecución de las instrucciones de un programa. Todas las estructuras de control tienen un único punto de entrada. Las estructuras de control se puede clasificar en: secuenciales, transferencia de control e iterativas. Básicamente lo que varía entre las estructuras de control de los diferentes lenguajes es su sintaxis, cada lenguaje tiene una sintaxis propia para expresar la estructura.

NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.

Tipos de datos

En Java existen dos grupos de tipos de datos, tipos primitivos y tipos referencia.

Tipos de dato primitivos

Se llaman **tipos primitivos** a aquellos datos sencillos que contienen los tipos de información más habituales: valores booleanos, caracteres y valores numéricos enteros o de punto flotante.

Java dispone de ocho tipos primitivos:

Tipo	Definición
boolean	`true o false`
char	Carácter Unicode de 16 bits
byte	Entero en complemento a dos con signo de 8 bits
short	Entero en complemento a dos con signo de 16 bits
int	Entero en complemento a dos con signo de 32 bits
long	Entero en complemento a dos con signo de 64 bits
float	Real en punto flotante según la norma IEEE 754 de 32 bits
double	Real en punto flotante según la norma IEEE 754 de 64 bits

En Java al contrario que en C o C++ el tamaño de los tipos primitivos no depende del sistema operativo o de la arquitectura ya que en todas las arquitecturas y bajo todos los sistemas operativos el tamaño en memoria es el mismo.

Es posible recubrir los tipos primitivos para tratarlos como cualquier otro objeto en Java. Así por ejemplo, existe una clase envoltura del tipo primitivo **int** llamado **Integer**. La utilidad de estas clases se explicará en otro momento.

Tipos de dato referencia

Los **tipos de dato referencia** representan datos compuestos o estructuras, es decir, referencias a objetos. Estos tipos de dato almacenan las direcciones de memoria y no el valor en sí (similares a los apuntadores en C). Una referencia a un objeto es la dirección de un área en memoria destinada a representar ese objeto.

Variables

Una **variable** es un **nombre** que contiene un valor que **puede cambiar** a lo largo del programa. De acuerdo con el tipo de información que contienen, en Java hay dos tipos principales de variables:

1. Variables de tipos primitivos.
2. Variables referencia.

Desde el punto de vista del papel o misión en el programa, las variables pueden ser:

- Variables **miembro** de una clase: Se definen en una clase, fuera de cualquier método; pueden ser tipos primitivos o referencias.
- Variables **locales**: Se definen dentro de un método o más en general dentro de cualquier bloque entre llaves { }. Se crean en el interior del bloque y se destruyen al finalizar dicho bloque.

Una variable se define especificando el **tipo** y el **nombre** de dicha variable. Estas variables pueden ser tanto de tipos primitivos como referencias a objetos de alguna clase perteneciente al API de Java o generada por el usuario.

tipoDeDato *nombreVariable***;**

Ejemplo:

int *miVariable*; **float** *area*; **char** *letra*; **String** *cadena*; **MiClase** *prueba*;

Si no se especifica un valor en su declaración, las variables primitivas se inicializan a **cero** (salvo boolean y char, que se inicializan a false y '\0'). Análogamente las variables de tipo referencia son inicializadas por defecto a un valor especial: **null**.

Es importante distinguir entre la referencia a un objeto y el objeto mismo. Una referencia es una variable que indica dónde está guardado un objeto en la memoria (a diferencia de C/C++, Java no permite acceder al valor de la dirección, pues en este lenguaje se han eliminado los apuntadores). Al declarar una referencia todavía no se encuentra "apuntando" a ningún objeto en particular (salvo que se cree explícitamente un nuevo objeto en la declaración), y por eso se le asigna el valor **null**.

Si se desea que esta referencia apunte a un nuevo objeto es necesario crear el objeto utilizando el operador **new**.

Este operador reserva espacio en la memoria para ese objeto (variables y funciones).

También es posible igualar la referencia declarada a otra referencia a un objeto existente previamente.

Ejemplo:

```
MyClass unaRef;
unaRef = new MyClass();
MyClass segundaRef = unaRef;
```

Un tipo particular de referencias son los **arrays** o **arreglos**, sean éstos de variables primitivas (por ejemplo de enteros) o de objetos. En la declaración de una referencia de tipo **array** hay que incluir los **corchetes** [ ].

Ejemplo:

```
int [ ] vector;
vector = new int[10];
MyClass [ ] lista=new MyClass[5];
```

En Java todas las variables deben estar incluidas en una clase. En general las variables declaradas dentro de llaves **{ }**, es decir, dentro de un bloque, son visibles y existen dentro de estas llaves. Por ejemplo las variables declaradas al principio de un método existen mientras se ejecute el método; las variables declaradas dentro de un bloque if no serán válidas al finalizar las sentencias correspondientes a dicho if y las variables miembro de una clase (es decir declaradas entre las llaves { } de la clase pero fuera de cualquier método) son válidas mientras existe el objeto de la clase.

Constantes

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

final *tipoDato* nombreDeConstante = *valor;*

Ejemplo:

```
final double PI = 3.1416;
```

Entrada y salida de datos por consola

Una de las operaciones más habituales que tiene que realizar un programa es intercambiar datos con el exterior. Para ello el **API** de java incluye una serie de clases que permiten gestionar la entrada y salida de datos en un programa, independientemente de los dispositivos utilizados para el envío/recepción de datos.

Para el envío de datos al exterior se utiliza un flujo de datos de impresión o **print stream**. Esto se logra usando la siguiente expresión:

System.out.println("Mi mensaje");

De manera análoga, para la recepción o lectura de datos desde el exterior se utiliza un flujo de datos de entrada o **input stream**. Para lectura de datos se utiliza la siguiente sintaxis:

```
Scanner sc = new Scanner(System.in);
String s = sc.next( );          //Para cadenas
int x = sc.nextInt( );          //Para enteros
```

Para usar la clase Scanner se debe incluir al inicio del archivo la siguiente línea:

import java.util.Scanner;

Al finalizar su uso se debe cerrar el flujo usando el método **close**.

Para el ejemplo:

sc.close( );

Estas clases y sus métodos se verán más a detalle en la práctica de Manejo de archivos.

Operadores

Java es un lenguaje rico en operadores, que son casi idénticos a los de C/C++.

Operadores aritméticos:

Son operadores binarios (requieren siempre dos operandos) que realizan las operaciones aritméticas habituales: suma (**+**), resta (**-**), multiplicación (*), división (**/**) y resto de la división o módulo (%).

Operadores de asignación

Los operadores de asignación permiten asignar un valor a una variable. El operador de asignación por excelencia es el operador igual (**=**). La forma general de las sentencias de asignación con este operador es:

variable = expression;

Java dispone de otros operadores de asignación. Se trata de versiones abreviadas del operador (**=**) que realizan operaciones "acumulativas" sobre una variable.

Operador	Utilización	Expresión equivalente
+=	`op1 += op2`	`op1 = op1 + op2`
-=	`op1 -= op2`	`op1 = op1 - op2`
*=	`op1 *= op2`	`op1 = op1 * op2`
/=	`op1 /= op2`	`op1 = op1 / op2`
%=	`op1 %= op2`	`op1 = op1 % op2`

Operadores unarios

Los operadores más (**+**) y menos (**-**) unarios sirven para mantener o cambiar el signo de una variable, constante o expresión numérica. Su uso en Java es el estándar de estos operadores.

Operador instanceof

El operador **instanceof** permite saber si un objeto **pertenece o no** a una determinada clase. Es un operador binario cuya forma general es:

objectName ***instanceof*** *ClassName*

Este operador devuelve **true** o **false** según el objeto pertenezca o no a la clase.

Operador condicional ? :

Este operador, tomado de C/C++, permite realizar bifurcaciones condicionales sencillas. Su forma general es la siguiente:

booleanExpression ? res1 : res2

Donde se evalúa **booleanExpression** y se devuelve **res1** si el resultado es **true** y **res2** si el resultado es **false**. Es el único operador ternario (tres argumentos) de Java. Como todo operador que devuelve un valor puede ser utilizado en una expresión.

Operadores incrementales

Java dispone del operador incremento (**++**) y decremento (**--**). El operador (**++**) incrementa en una unidad la variable a la que se aplica, mientras que (**--**) la reduce en una unidad. Estos operadores se pueden utilizar de dos formas:

- Precediendo a la variable (por ejemplo: **++i**). En este caso primero se incrementa la variable y luego se utiliza (ya incrementada) en la expresión en la que aparece.
- Siguiendo a la variable (por ejemplo: **i++**). En este caso primero se utiliza la variable en la expresión (con el valor anterior) y luego se incrementa.

La actualización de contadores en ciclos **for** es una de las aplicaciones más frecuentes de estos operadores.

Operadores relacionales

Los operadores relacionales sirven para realizar **comparaciones** de igualdad, desigualdad y relación de menor o mayor. El resultado de estos operadores es siempre un valor **boolean** (**true** o **false**) según se cumpla o no la relación considerada.

Operador	Utilización	El resultado es true
>	op1 > op2	si op1 es mayor que op2
>=	op1 >= op2	si op1 es mayor o igual que op2
<	op1 < op2	si op1 es menor que op2
<=	op1 <= op2	si op1 es menor o igual que op2
==	op1 == op2	si op1 y op2 son iguales
!=	op1 != op2	si op1 y op2 son diferentes

Estos operadores se utilizan con mucha frecuencia en las estructuras de control.

Operadores lógicos

Los operadores lógicos se utilizan para construir **expresiones lógicas**, combinando valores lógicos (**true** y/o **false**) o los resultados de los operadores relacionales.

Operador	Nombre	Utilización	Resultado
&&	AND	op1 && op2	true si op1 y op2 son true. Si op1 es false ya no se evalúa op2
||	OR	op1 || op2	true si op1 u op2 son true. Si op1 es true ya no se evalúa op2
!	negación	! op	true si op es false y false si op es true
&	AND	op1 & op2	true si op1 y op2 son true. Siempre se evalúa op2
|	OR	op1 | op2	true si op1 u op2 son true. Siempre se evalúa op2

Debe notarse que en ciertos casos el segundo operando no se evalúa porque ya no es necesario (si ambos tienen que ser true y el primero es false, ya se sabe que la condición de que ambos sean true no se va a cumplir). Esto puede traer resultados no deseados y por eso se han añadido los operadores (&) y (|) que garantizan que los dos operandos se evalúan siempre.

Operador de concatenación de cadenas de caracteres (+)

El operador más (**+**) se utiliza también para concatenar cadenas de caracteres. Por ejemplo, para escribir una cantidad con un rótulo y valores puede utilizarse la sentencia:

System.out.println("El total asciende a " **+** *result* **+** *" unidades");*

Donde el operador de concatenación se utiliza dos veces para construir la cadena de caracteres que se desea imprimir por medio del método println( ). La variable numérica result es convertida automáticamente por Java en cadena de caracteres para poderla concatenar. En otras ocasiones se deberá llamar explícitamente a un método para que realice esta conversión.

Operadores a nivel de bits

Java dispone también de un conjunto de operadores que actúan a nivel de bits. Las operaciones de bits se utilizan con frecuencia para definir señales o flags, esto es, variables de tipo entero en las que cada uno de sus bits indica si una opción está activada o no.

Operador	Utilización	Resultado
>>	op1 >> op2	Desplaza los bits de op1 a la derecha una distancia op2
<<	op1 << op2	Desplaza los bits de op1 a la izquierda una distancia op2
>>>	op1 >>> op2	Desplaza los bits de op1 a la derecha una distancia op2 (positiva)
&	op1 & op2	Operador AND a nivel de bits
|	op1 | op2	Operador OR a nivel de bits
^	op1 ^ op2	Operador XOR a nivel de bits (1 si sólo uno de los operandos es 1)
~	~op2	Operador complemento (invierte el valor de cada bit)

Operador	Utilización	Equivalente a
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

Precedencia de operadores

El **orden** en que se realizan las operaciones es fundamental para determinar el resultado de una expresión. La siguiente lista muestra el orden en que se ejecutan los distintos operadores en una sentencia, de mayor a menor precedencia:

postfix operators	[ ] . (params) expr++ expr--
unary operators	++expr --expr +expr -expr ~ !
creation or cast	new (type)expr
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
conditional	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

En Java, todos los operadores binarios (excepto los operadores de asignación) se evalúan de **izquierda a derecha**. Los operadores de asignación se evalúan de derecha a izquierda, lo que significa que el valor de la derecha se copia sobre la variable de la izquierda.

Estructuras de control

Las estructuras de programación o estructuras de control permiten **tomar decisiones** o **realizar un proceso repetidas veces**. En la mayoría de los lenguajes de programación, este tipo de estructuras son comunes en cuanto a concepto, aunque su sintaxis varía de un lenguaje a otro. La sintaxis de Java coincide prácticamente con la utilizada en C/C++, lo que hace que para un programador de C/C++ no represente ninguna dificultad adicional.

Sentencias o expresiones

Una expresión es un conjunto de variables unidos por operadores. Son órdenes que se le dan a la computadora para que realice una tarea determinada. Una sentencia es una expresión que acaba en punto y coma (;). Se permite incluir varias sentencias en una línea, aunque lo habitual es utilizar una línea para cada sentencia.

Ejemplo:

```
i = 0; j = 5; x = i + j;     // Línea compuesta de tres sentencias
```

Estructuras de selección

Las estructuras de selección o bifurcaciones permiten ejecutar una de entre varias acciones en función del valor de una expresión lógica o relacional. Se tratan de estructuras muy importantes ya que son las encargadas de controlar el **flujo de ejecución** de un programa.

Existen dos bifurcaciones diferentes: **if** y **switch**.

IF / IF-ELSE

Esta estructura permite ejecutar un conjunto de sentencias en función del valor que tenga la expresión de comparación (se ejecuta si la expresión de comparación tiene valor **true**).

Las sentencias incluidas en el **else** se ejecutan en el caso de no cumplirse la expresión de comparación (**false**). Tiene la forma siguiente:

```
if (booleanExpression) {
        statements1;
} else {
        statements2;
}
```

Las llaves **{ }** sirven para agrupar en un bloque las sentencias que se han de ejecutar, y no son necesarias si sólo hay una sentencia dentro del **if**.

Si se desea introducir más de una expresión de comparación se usa **if / else if**. Si la primera condición no se cumple, se compara la segunda y así sucesivamente. En el caso de que no se cumpla ninguna de las comparaciones se ejecutan las sentencias correspondientes al **else**.

```
if (booleanExpression1) {
        statements1;
} else if (booleanExpression2) {
        statements2;
} else if (booleanExpression3) {
        statements3;
} else {
        statements4;
}
```

SWITCH

Se trata de una alternativa a la bifurcación **if /else if** cuando se compara la **misma expresión** con distintos valores.

Su forma general es la siguiente:

```
switch (expression) {
        case value1:
                statements1;
                break;
        case value2:
                statements2;
                break;
        case value3:
                statements3;
                break;
        case value4:
                statements4;
                break;
        case value5:
                statements5;
                break;
        case value6:
                statements6;
                break;
        [default:
                statements7;]
}
```

Las características más relevantes de **switch** son las siguientes:

- Cada sentencia **case** corresponde con un único valor de **expression**. No se pueden establecer rangos o condiciones sino que se debe comparar con valores concretos de tipo **int** (incluyendo a los que se pueden convertir a **int** como **byte, char y short**).
- No puede haber dos etiquetas **case** con el mismo valor.
- Los valores no comprendidos en ninguna sentencia case se pueden gestionar en **default**, que es **opcional**.
- En ausencia de **break**, cuando se ejecuta una sentencia case se ejecutan también todas las case que van a continuación, hasta que se llega a un **break** o hasta que se termina el **switch**.

Estructuras de repetición

Las estructuras de repetición, lazos, ciclos o bucles se utilizan para realizar un proceso **repetidas veces**. El código incluido entre las llaves **{ }** (opcionales si el proceso repetitivo consta de una sola línea), se ejecutará mientras se cumplan determinadas condiciones.

Hay que prestar especial atención a los ciclos infinitos, hecho que ocurre cuando la condición de finalizar el ciclo no se llega a cumplir nunca. Se trata de un fallo muy típico, habitual sobre todo entre programadores poco experimentados.

WHILE

Las sentencias **statements** se ejecutan mientras **booleanExpression** sea **true**.

```
while (booleanExpression) {
        statements;
}
```

DO-WHILE

Es similar al ciclo **while** pero con la particularidad de que el control está **al final del ciclo** (lo que hace que el ciclo se ejecute **al menos una vez**, independientemente de que la condición se cumpla o no). Una vez ejecutados los **statements**, se evalúa la condición: si resulta **true** se vuelven a ejecutar las sentencias incluidas en el ciclo, mientras que si la condición se evalúa a **false** finaliza el ciclo.

```
do {
        statements
} while (booleanExpression);
```

FOR

La forma general del **for** es la siguiente:

```
for (initialization; booleanExpression; increment) {
        statements;
}
```

La sentencia o sentencias **initialization** se ejecutan al comienzo del **for**, e **increment** después de **statements**. La **booleanExpression** se evalúa al comienzo de cada iteración; el ciclo termina cuando la expresión de comparación toma el valor **false**. Cualquiera de las tres partes puede estar vacía. La **initialization** y el **increment** pueden tener varias expresiones separadas por comas.

BREAK y CONTINUE

La sentencia **break** es válida tanto para las bifurcaciones como para los ciclos. Hace que se **salga inmediatamente** del ciclo o bloque que se está ejecutando, sin realizar la ejecución del resto de las sentencias.

La sentencia **continue** se utiliza en los ciclos (no en bifurcaciones). Finaliza la iteración "i" que en ese momento se está ejecutando (no ejecuta el resto de sentencias que hubiera hasta el final del bucle). Vuelve al comienzo del bucle y comienza la **siguiente iteración** (i+1).

Ejemplos usando estructuras de control:

Programa que suma los números pares comprendidos entre n1 y n2.

```
import java.util.Scanner;
public class SumaPares {
    public static void main(String[] args) {
        //Declaración de variables
        int n1 , n2;
        int suma = 0;
        int mayor, menor;
        Scanner sc = new Scanner(System.in);
        //Pedir datos al usuario
        System.out.println("Por favor introduzca un número entero");
        n1 = sc.nextInt();
        System.out.println("Introduzca otro número entero");
        n2 = sc.nextInt();
        //Validar cual es el número mayor y el menor
        if (n1 > n2){
            mayor = n1;
            menor = n2;
        } else {
            mayor = n2;
            menor = n1;
        }
        //Hacer un ciclo desde el menor hasta el mayor
        for(int i = menor; i <= mayor; i++){
            //Validar si es par para sumarlo
            if( i % 2 == 0){
                suma += i;
            }
        }
        //Imprimir el resultado
        System.out.println("La suma de los pares entre " + n1 + " y " + n2 +" es " + suma);
        sc.close();
    }
}
```

Programa que calcula el área de una figura geométrica dependiendo la opción seleccionada por el usuario en un menú que se repite hasta seleccionar la opción "Salir".

```
import java.util.Scanner;
public class FigurasGeometricas {
    public static void main(String[] args) {
        float area;
        int opcion;
        final float PI = 3.14159f;
        Scanner sc = new Scanner(System.in);
        do{
            System.out.println("Elige la opción");
            System.out.println("1.-Area de círculo");
            System.out.println("2.-Area de triángulo");
            System.out.println("3.-Area de cuadrado");
            System.out.println("4.-Salir");
            opcion = sc.nextInt();
            switch (opcion) {
            case 1:
                //Círculo
                System.out.println("Seleccionó el círculo");
                float radio = 15;
                area = PI * radio * radio;
                break;
            case 2:
                //Triángulo
                System.out.println("Seleccionó el triángulo");
                float base = 8, altura = 5;
                area = ( base * altura ) / 2;
                break;
            case 3:
                //Cuadrado
                System.out.println("Seleccionó el cuadrado");
                float lado = 10;
                area = lado * lado;
                break;
            case 4:
                //Salir
                System.out.println("Hasta luego");
                continue;
            default:
                //Ninguno de los anteriores
                System.out.println("Su elección no es correcta");
                continue;
            }
            System.out.println("El area es: " + area);
        } while (opcion != 4);
        sc.close();
    }
}
```

Bibliografía

Martín, Antonio
Programador Certificado Java 2.
Segunda Edición.
México
Alfaomega Grupo Editor, 2008

Sierra Katy, Bates Bert
SCJP Sun Certified Programmer for Java 6 Study Guide
Mc Graw Hill

Dean John, Dean Raymond.
Introducción a la programación con Java
Primera Edición.
México
Mc Graw Hill, 2009