FUNDAMENTOS DE JAVA

A First Simple Program

Let's start by compiling and running the short sample program shown here:

```
/* This is a simple Java program.*/

class Example
{
        // A Java program begins with a call to main().
        public static void main(String args[]) {
                System.out.println("Java lenguaje también para la Web.");
        }
 }
```

You will follow these three steps:
1. Enter the program.
2. Compile the program.
3. Run the program.
4. Handling Syntax Errors

1. abre bloc de notas y escribe el programa, sálvalo con nombre Example.java
   Desde línea de comando, posicionados en el directorio donde está el fuente:

   2. javac Example.java (compilar, generará un archive . class)
   3. java Example (ejecución)

A Second Simple Program

```
/* This demonstrates a variable.*/

class Example2
 {
        public static void main(String args[]) {
                int var1; // this declares a variable
                int var2; // this declares another variable
                 var1 = 1024; // this assigns 1024 to var1
                System.out.println("var1 contains " + var1);
                var2 = var1 / 2;
                System.out.print("var2 contains var1 / 2: ");
                System.out.println(var2);
         }
```

```
}
```

Another Data Type

```
/* This program illustrates the differences
between int and double. */

class Example3
{
        public static void main(String args[]) {
                int var; // this declares an int variable
                double x; // this declares a floating-point variable
                var = 10; // assign var the value 10
                x = 10.0; // assign x the value 10.0
                System.out.println("Original value of var: " + var);
                System.out.println("Original value of x: " + x);
                System.out.println(); // print a blank line
                // now, divide both by 4
                var = var / 4;
                x = x / 4;
                System.out.println("var after division: " + var);
                System.out.println("x after division: " + x);
        }
}
```

Ejercicio:

Si un galón son 3,7854118 litros, cuántos litros son 15 galones?

1. Data Types and Operators

Los tipos de datos son especialmente importantes en Java porque es un lenguaje fuertemente tipado. Esto significa que todas las operaciones son verificadas por el compilador para la compatibilidad de tipos. Las operaciones ilegales no serán compiladas. Por lo tanto, la verificación de tipo fuerte ayuda a prevenir errores y mejora la confiabilidad.

1.1. Java's Primitive Types

Java contiene dos categorías generales de tipos de datos integrados: orientado a objetos y no orientado a objetos.

El término primitivo se usa aquí para indicar que estos tipos no son objetos en un sentido orientado a objetos, sino más bien valores binarios normales. Estos tipos primitivos no son objetos debido a problemas de eficiencia. Todos los demás tipos de datos de Java se construyen a partir de estos tipos primitivos.

Type	Meaning
boolean	Represents true/false values
byte	8-bit integer
char	Character
double	Double-precision floating point
float	Single-precision floating point
int	Integer
long	Long integer
short	Short integer

Integers

Type	Width in Bits	Range
byte	8	–128 to 127
short	16	–32,768 to 32,767
int	32	–2,147,483,648 to 2,147,483,647
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

El tipo entero más utilizado es int. Las variables de tipo int a menudo se emplean para controlar bucles, indexar matrices y realizar cálculos enteros de propósito general.

Cuando necesite un número entero que tenga un rango mayor que int, use long.

```
/* Compute the number of cubic inches in 1 cubic mile. */
class Inches
{       public static void main(String args[]) {
                long ci;
                long im;
                im = 5280 * 12;
                ci = im * im * im;
                System.out.println("There are " + ci + " cubic inches in cubic mile.");
        }
}
```

Floating-Point Types

There are two kinds of floating-point types, float and double, which represent single- and double-precision numbers, respectively. Type float is 32 bits wide and type double is 64 bits wide.

Of the two, double is the most commonly used because all of the math functions in Java's class library use double values.

```
/* Use the Pythagorean theorem
to find the length of the hypotenuse given
the lengths of the two opposing sides.*/
class Hypot
{
        public static void main(String args[]) {
                double x, y, z;
                x = 3;
                y = 4;
                z = Math.sqrt(x*x + y*y);
                System.out.println("Hypotenuse is " +z);
        }
}
```

Ejercicio:

Dado el siguiente método de la clase Math:

static double	random() Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.

```
(int)(Math.random()*(maximo-minimo+1)+(minimo));
```

Escribe un programa en Java que busque y muestre en pantalla un número aleatorio comprendido entre 1.0 y 100.0 ambos incluidos.

Ejercicio:

Escribe un programa que calcule a qué distancia se ha producido un rayo cuyo trueno hemos oído 7,2 segundos después y sabiendo que el sonido viaja a unos 340 metros por segundo a través del aire.

Characters

En Java, los caracteres no son cantidades de 8 bits como en muchos otros lenguajes de computadora. En cambio, Java usa Unicode. Unicode define un conjunto de caracteres que puede representar todos los caracteres encontrados en todos los idiomas humanos. En Java, char es un tipo de 16 bits sin signo que tiene un rango de 0 a 65.536. El juego de caracteres ASCII estándar de 8 bits es un subconjunto de Unicode y varía de 0 a 127. Por lo tanto, los caracteres ASCII siguen siendo caracteres Java válidos.

A character variable can be assigned a value by enclosing the character in single quotes. For example, this assigns the variable ch the letter X:

```
char ch;
ch = 'X';
```

You can output a char value using a println( ) statement. For example, this line outputs the value in ch:

```
System.out.println("This is ch: " + ch);
```

Como char es un tipo de 16 bits sin signo, es posible realizar varias manipulaciones aritméticas en una variable char. Por ejemplo, considere el siguiente programa:

```
// Character variables can be handled like integers.

class CharArithDemo
{
	public static void main(String args[]){
		char ch;
		ch = 'X';
		System.out.println("ch contains " + ch+" y vale="+(int)ch);
		ch=(char) ch-4; // decremento ch
		System.out.println("ch is now " + ch);
		ch = 90; // give ch the value Z
		System.out.println("ch is now " + ch);
	}
}
```

The Boolean Type

The boolean type represents true/false values. Java defines the values true and false using the reserved words true and false.

```
// Demonstrate boolean values.
class BoolDemo
{
	public static void main(String args[]) {
		boolean b;
		b = false;
		System.out.println("b is " + b);
```

```
        b = true;
        System.out.println("b is " + b);

        // a boolean value can control the if statement
        if(b) System.out.println("This is executed.");

        b = false;
        if(b) System.out.println("This is not executed.");

        // outcome of a relational operator is a boolean value
        System.out.println("10 > 9 is " + (10 > 9));
    }
}
```

Ejercicio:

Escribe un programa que genere dos valores aleatorios en un intervalo cualquiera y muestre un mensaje que indique si es cierto o falso que el primer número generado es mayor que el segundo.

1.2. **Literals**

En Java, los literales se refieren a valores fijos que se representan en su forma legible por humanos. Por ejemplo, el número 100 es un literal. Los literales también se llaman comúnmente constantes.

Los literales de Java pueden ser de cualquiera de los tipos de datos primitivos. La forma en que se representa cada literal depende de su tipo. Por ejemplo, 'a' y '%' son constantes de caracteres.

Los literales enteros se especifican como números sin componentes fraccionarios. Por ejemplo, 10 y –100 son literales enteros.

Los literales de punto flotante requieren el uso del punto decimal seguido del componente fraccionario del número. Por ejemplo, 11.123 es un literal de coma flotante. Java también le permite usar notación científica para números de punto flotante.

Por defecto, los literales enteros son de tipo int. Si desea especificar un literal largo, agregue un l o un L. Por ejemplo, 12 es un int, pero 12L es un largo.

Por defecto, los literales de coma flotante son de tipo double. Para especificar un literal flotante, agregue una F o f a la constante. Por ejemplo, 10.19F es de tipo float.

Aunque los literales enteros crean un valor int de forma predeterminada, aún pueden asignarse a variables de tipo char, byte o short siempre que el valor que se asigne pueda representarse por el tipo de destino. Un literal entero siempre se puede asignar a una variable larga.

1.3. **Hexadecimal, Octal, and Binary Literals**

As you may know, in programming it is sometimes easier to use a number system based on 8 or 16 instead of 10. The number system based on 8 is called *octal*, and it uses the digits 0 through 7. In octal the number 10 is the same as 8 in decimal. The base 16 number system is called *hexadecimal* and uses the digits 0 through 9 plus the letters A through F, which stand for 10, 11, 12, 13, 14, and 15. For example, the hexadecimal number 10 is 16 in decimal. Because of the frequency with which these two number systems are used, Java allows you to specify integer literals in hexadecimal or octal instead of decimal. A hexadecimal literal must begin with **0x** or **0X** (a zero followed by an x or X). An octal literal begins with a zero. Here are some examples:

```
hex = 0xFF; // 255 in decimal
oct = 011; // 9 in decimal
```

As a point of interest, Java also allows hexadecimal floating-point literals, but they are seldom used.

Beginning with JDK 7, it is possible to specify an integer literal by use of binary. To do so, precede the binary number with a **0b** or **0B**. For example, this specifies the value 12 in binary: **0b1100**.

1.4. **Character Escape Sequences**

El encerrar constantes de caracteres entre comillas simples funciona para la mayoría de los caracteres de impresión, pero algunos caracteres, como el retorno de carro, plantean un problema especial cuando se utiliza un editor de texto. Además, ciertos otros caracteres, como las comillas simples y dobles, tienen un significado especial en Java, por lo que no puede usarlos directamente. Por estos motivos, Java proporciona secuencias de escape especiales, a veces denominadas constantes de caracteres de barra invertida, que se muestran en la Tabla 2-2. Estas secuencias se utilizan en lugar de los carcateres que representan.

Escape Sequence	Description
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line
\f	Form feed
\t	Horizontal tab
\b	Backspace
ddd	Octal constant (where *ddd* is an octal constant)

\uxxxx	Hexadecimal constant (where xxxx is a hexadecimal constant)

For example, this assigns **ch** the tab character:

```
ch = '\t';
```

The next example assigns a single quote to **ch**:

```
ch = '\'';
```

1.5. String Literals

Java supports one other type of literal: the string. A *string* is a set of characters enclosed by double quotes. For example,

"this is a test"

Is a string. You have seen examples of strings in many of the **println( )** statements in the preceding sample programs.

In addition to normal characters, a string literal can also contain one or more of the escape sequences just described. For example, consider the following program. It uses the **\n** and **\t** escape sequences.

```
// Demonstrate escape sequences in strings.
class StrDemo
{
        public static void main(String args[]) {
                System.out.println("First line\nSecond line");
                System.out.println("A\tB\tC");
                System.out.println("D\tE\tF") ;
        }
}
```

2. A Closer Look at Variables

Variables were introduced in Chapter 1. Here, we will take a closer look at them. As you learned earlier, variables are declared using this form of statement,

type var-name;

donde type es el tipo de datos de la variable y var-name es su nombre. Puede declarar una variable de cualquier tipo válido, incluidos los tipos simples que se acaban de describir, y cada variable tendrá un tipo. Por lo tanto, las capacidades de una variable están determinadas por su tipo. Por ejemplo, una variable de tipo booleano no se puede usar para almacenar valores de punto flotante. Además, el tipo de una variable no puede cambiar durante su vida útil. Una variable int no puede convertirse en una variable char, por ejemplo.

Todas las variables en Java deben declararse antes de su uso. Esto es necesario porque el compilador debe saber qué tipo de datos contiene una variable antes de poder compilar correctamente cualquier declaración que use la variable. También permite que Java realice una verificación de tipo estricta.

Initializing a Variable

En general, debe dar un valor a una variable antes de usarla. Otra forma es dándole un valor inicial cuando se declara. La forma general de inicialización se muestra aquí:

```
var = valor;
```

Aquí, valor es el valor que se le da a var cuando se crea var. El valor debe ser compatible con el tipo especificado.

Dynamic Initialization

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared. For example, here is a short program that computes the volume of a cylinder given the radius of its base and its height:

```
// Demonstrate dynamic initialization.
class DynInit
{
	public static void main(String args[]) {
		double radius = 4, height = 5;
		// dynamically initialize volume
		double volume = 3.1416 * radius * radius * height;
		System.out.println("Volume is " + volume);
	}
}
```

3. The Scope and Lifetime of Variables

Hasta ahora, todas las variables que hemos estado utilizando se declararon al comienzo del método main (). Sin embargo, Java permite que las variables se declaren dentro de cualquier bloque.

Como regla general, las variables declaradas dentro de un ámbito no son visibles (es decir, accesibles) para el código que se define fuera de ese ámbito. Por lo tanto, cuando declara una variable dentro de un ámbito, está localizando esa variable y protegiéndola del acceso y / o modificación no autorizados. De hecho, las reglas de alcance proporcionan la base para la encapsulación.

Los ámbitos se pueden anidar. Por ejemplo, cada vez que crea un bloque de código, está creando un nuevo ámbito anidado. Cuando esto ocurre, el alcance externo encierra el alcance interno. Esto significa que los objetos declarados en el alcance externo serán visibles para el código dentro del alcance interno. Sin embargo, lo contrario no es cierto.

```
// Demonstrate block scope.
class ScopeDemo
{
	public static void main(String args[]) {
		int x; // known to all code within main
		x = 10;
		if(x == 10) { // start new scope
			int y = 20; // known only to this block
			// x and y both known here.
			System.out.println("x and y: " + x + " " + y);
			x = y * 2;
			}
		// y = 100; // Error! y not known here
		// x is still known here.
		System.out.println("x is " + x);
	}
}
```

4. **Arithmetic Operators**

Operator	**Meaning**
+	Addition (also unary plus)
-	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
- -	Decrement

Ejercicio:

Escribe un programa que muestre un mensaje que indique si es cierto o falso que el número 364 es múltiplo del valor 13.

5. Increment and Decrement

The increment operator adds 1 to its operand, and the decrement operator subtracts 1. Therefore,

```
x = x + 1; is the same as x++;
```

and

```
x = x - 1; is the same as x--;
```

Both the increment and decrement operators can either precede (prefix) or follow (postfix) the operand. For example,

```
x = x + 1;
```

can be written as

```
++x; // prefix form
```

or as

```
x++; // postfix form
```

Sin embargo, cuando se usa un incremento o decremento como parte de una expresión más grande, hay una diferencia importante.
Cuando un operador de incremento o decremento precede **a su operando**, Java realizará la operación correspondiente antes de obtener el valor del operando para ser utilizado por el resto de la expresión. Si el operador sigue su a operando, Java obtendrá el valor del operando antes de incrementarlo o disminuirlo. Considera lo siguiente:

```
x = 10;
y = ++x;
```

In this case, **y** will be set to 11. However, if the code is written as

```
x = 10;
y = x++;
```

then **y** will be set to 10.

6. Relational and Logical Operators

The relational operators are shown here:

Operator	Meaning
= =	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The logical operators are shown next:

Operator	Meaning
&	AND
|	OR
^	XOR (exclusive OR) (iguales 0)
||	Short-circuit OR
&&	Short-circuit AND
!	NOT

In Java, all objects can be compared for equality or inequality using = = and !=. However, the comparison operators, <, >, <=, or >=, can be applied only to those types that support an ordering relationship. Therefore, all of the relational operators can be applied to all numeric types and to type **char**. However, values of type **boolean** can only be compared for equality or inequality, since the **true** and **false** values are not ordered. For example, **true > false** has no meaning in Java.

For the logical operators, the operands must be of type **boolean**, and the result of a logical operation is of type **boolean**. The logical operators, **&**, **|**, **^(xor o suma exclusiva)**, and **!**, support the basic logical operations AND, OR, XOR, and NOT, according to the following truth table:

p	q	p & q	p | q	p ^ q	!p
False	False	False	False	False	True
True	False	False	True	True	False
False	True	False	True	True	True
True	True	True	True	False	False

6.1. Short-Circuit Logical Operators

Java suministra versiones especiales de cortocircuito de sus operadores lógicos AND y OR que se pueden usar para producir un código más eficiente. Para entender por qué, considere lo siguiente. En una operación AND, si el primer operando es falso, el resultado es falso sin importar el valor que tenga el segundo operando. En una operación OR, si el primer operando es verdadero, el resultado de la operación es verdadero sin importar el valor del segundo operando. Por lo tanto, en estos dos casos no es necesario evaluar el segundo operando. Al no evaluar el segundo operando, se ahorra tiempo y se produce un código más eficiente.

El operador AND de cortocircuito es &&, y el operador O de cortocircuito es ||. Sus contrapartes normales son & y |. La única diferencia entre las versiones normal y de cortocircuito es que los operandos normales siempre evaluarán cada operando, pero las versiones de cortocircuito evaluarán el segundo operando solo cuando sea necesario.

7. The Assignment Operador

The *assignment operator* is the single equal sign, =.

```
var = expression;
```

8. Shorthand Assignments

Java provides special *shorthand* assignment operators that simplify the coding of certain assignment statements. Let's begin with an example. The assignment statement shown here

```
x = x + 10;
```

can be written, using Java shorthand, as

```
x += 10;
```

The general form of the shorthand is

```
var op = expression;
```

Thus, the arithmetic and logical shorthand assignment operators are the following:

+=	-=	*=	/=
%=	&=	|=	^=

9. Type Conversion in Assignments

In programming, it is common to assign one type of variable to another. For example, you might want to assign an **int** value to a **float** variable, as shown here:

```
int i;
float f;
i = 10;
f = i; // assign an int to a float
```

Cuando los tipos compatibles se mezclan en una asignación, el valor del lado derecho se convierte automáticamente al tipo del lado izquierdo. Por lo tanto, en el fragmento anterior, el valor en i se convierte en un float y luego se asigna a f. Sin embargo, debido a la estricta comprobación de tipos de Java, no todos los tipos son compatibles, y por lo tanto, no todas las conversiones de tipos están permitidas implícitamente. Por ejemplo, boolean e int no son compatibles.

Cuando un tipo de datos se asigna a otro tipo de variable, se realizará una conversión automática de tipos si

- The two types are compatible.
- The destination type is larger than the source type.

Cuando se cumplen estas dos condiciones, se produce una conversión al tipo mayor. Por ejemplo, el tipo int siempre es lo suficientemente grande para contener todos los valores de **bytes** válidos, y tanto **int** como **byte** son tipos **enteros**, por lo que se puede aplicar una conversión automática de byte a int.

LEGAL	ILLEGAL

class LtoD {
public static void main(String args[]) {
long L;
double D;
L = 100123285L;
D = L;
System.out.println("L and D: " + L + " " + D);
}
}
	// *** This program will not compile. ***
	class LtoD {
	public static void main(String args[]) {
	long L;
	double D;
	D = 100123285.0;
	L = D; // Illegal!!!
	System.out.println("L and D: " + L + " " + D);
	}
	}

9.1. **Casting Incompatible Types**

Aunque las conversiones automáticas de tipos son útiles, no satisfarán todas las necesidades de programación porque se aplican solo a las conversiones de ampliación entre tipos compatibles. Para todos los demás casos debes emplear un cast. Un cast es una instrucción para el compilador para convertir un tipo en otro. Por lo tanto, solicita una conversión de tipo explícita. Un cast tiene esta forma general:

```
double x, y;
// ...(int) (x / y)
```

Here, even though **x** and **y** are of type **double**, the cast converts the outcome of the expression to **int**. The parentheses surrounding **x / y** are necessary.

De lo contrario, el lanzamiento a int se aplicaría solo a la x y no al resultado de la división. El cast es necesario aquí porque no hay conversión automática de doble a int.

Cuando un cast implica una reducción de la conversión, la información puede perderse. Por ejemplo, al convertir un largo en un corto, la información se perderá si el valor de la longitud es mayor que el rango de un corto porque se eliminan sus bits de orden superior. Cuando un valor de punto flotante se convierte en un tipo entero, el componente fraccionario también se perderá debido al truncamiento. Por ejemplo, si el valor 1.23 está asignado a un número entero, el valor resultante simplemente será 1. El 0.23 se pierde.

The following program demonstrates some type conversions that require casts:

```
// Demonstrate casting.
class CastDemo {
        public static void main(String args[]) {
                double x, y;
                byte b;
                int i;
                char ch;
                x = 10.0;
                y = 3.0;
                i = (int) (x / y); // cast double to int
                System.out.println("Integer outcome of x / y: " + i);
                i = 100;
                b = (byte) i;
                System.out.println("Value of b: " + b);
                i = 257;
                b = (byte) i;
                System.out.println("Value of b: " + b);
                b = 88; // ASCII code for X
                ch = (char) b;
                System.out.println("ch: " + ch);
        }
}
```

10. **Operator Precedente**

Highest						
++ (postfix)	– – (postfix)					
++ (prefix)	– – (prefix)	~	!	+ (unary)	– (unary)	(*type-cast*)
*	/	%				
+	–					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
|						

&&	
||	
?:	
->	
=	op=
Lowest	

```
public static void main(String[] args) {
            int x=10,y,z=4;
            y=12+ ++z+ x++;
            //y=12+ ++z+10 y x ahora vale 11
            //y=12+ 5 +10 y ahora z vale 5
            System.out.println(y);
            System.out.println(x);
            System.out.println(z);

      }
```

11. **Expressions**

- Type Conversion in Expressions

Dentro de una expresión, es posible mezclar dos o más tipos diferentes de datos siempre que sean compatibles entre sí. Por ejemplo, puede mezclar corto y largo dentro de una expresión porque ambos son tipos numéricos. Cuando se mezclan diferentes tipos de datos dentro de una expresión, todos se convierten al mismo tipo. Esto se logra mediante el uso de las reglas de promoción de tipos de Java (reglas).

En primer lugar, todos los valores de char, byte y short se promueven en int. Entonces, si un operando es largo, la expresión completa se promueve a larga.

Si un operando es un operando flotante, la expresión completa se promueve a float.

Si alguno de los operandos es doble, el resultado es doble.

Es importante comprender que las promociones de tipo se aplican solo a los valores operados cuando se evalúa una expresión. Por ejemplo, si el valor de una variable de byte se promueve a int dentro de una expresión, fuera de la expresión, la variable sigue siendo un byte. La promoción de tipo solo afecta la evaluación de una expresión.

La promoción de tipo puede, sin embargo, conducir a resultados algo inesperados. Por ejemplo, cuando una operación aritmética implica dos valores de bytes, se produce la siguiente secuencia: Primero, los operandos de bytes se promueven a int. Entonces la operación tiene lugar, produciendo un resultado int.

```
// A promotion surprise!
class PromDemo {
```

```
        public static void main(String args[]) {
            byte b;
            int i;
            b = 10;
            i = b * b; // OK, no cast needed
            b = 10;
            b = (byte) (b * b); // cast needed!!
            System.out.println("i and b: " + i + " " + b);
        }
    }
```

Tipos de datos pequeños (*byte*, *short*, y *char*), se promueven primero a *int* siempre que sean utilizados con un operador aritmético binario en Java, incluso si ninguno de los operadores es un valor *int*.

- Spacing and Parentheses

Una expresión en Java puede tener parentesis y espacios para que sea más legible. Por ejemplo, las siguientes dos expresiones son las mismas, pero la segunda es más fácil de leer:

```
x=10/y*(127/x);
x = 10 / y * (127/x);
```

Los paréntesis aumentan la precedencia de las operaciones contenidas dentro de ellos, al igual que en álgebra. El uso de paréntesis redundantes o adicionales no causará errores ni ralentizará la ejecución de la expresión.

```
x = y/3-34*temp+127;
x = (y/3) - (34*temp) + 127;
```

Si dos valores tienen diferente tipo de dato, Java automáticamente promoverá uno de los valores al más grande de los dos tipos de datos.

1.Si uno de los valores es integral y el otro es un punto flotante, Java automáticamente promoverá el valor integral a un tipo de dato con valor de punto flotante.
2.Tipos de datos pequeños (*byte*, *short*, y *char*), se promueven primero a *int* siempre que sean utilizados con un operador aritmético binario en Java, incluso si ninguno de los operadores es un valor *int*.
3.Después de que todas las promociones hayan ocurrido y los operadores tengan el mismo tipo de dato, los valores resultantes tendrán el mismo tipo de dato como sus operadores promovidos.