UNIDAD DIDACTICA 3: INTRODUCCIÓN AL LENGUAJE JAVA

1.- ¿Qué es Java?

Java surgió en 1991 cuando un grupo de ingenieros de *Sun Microsystems* trataron de diseñar un nuevo lenguaje de programación destinado a electrodomésticos. La reducida potencia de cálculo y memoria de los electrodomésticos llevó a desarrollar un lenguaje sencillo capaz de generar código de tamaño muy reducido.

Debido a la existencia de distintos tipos de CPUs y a los continuos cambios, era importante conseguir una herramienta independiente del tipo de CPU utilizada. Desarrollaron un código "neutro" que no dependía del tipo de electrodoméstico, el cual se ejecutaba sobre una "*máquina hipotética o virtual*" denominada ***Java Virtual Machine*** (***JVM***). Era la ***JVM*** quien interpretaba el código neutro convirtiéndolo a código particular de la CPU utilizada. Esto permitía lo que luego se ha convertido en el principal lema del lenguaje: "*Write Once, Run Everywhere*". A pesar de los esfuerzos realizados por sus creadores, ninguna empresa de electrodomésticos se interesó por el nuevo lenguaje.

Como lenguaje de programación para computadores, ***Java*** se introdujo a finales de 1995. La clave fue la incorporación de un intérprete ***Java*** en la versión 2.0 del programa Netscape Navigator, produciendo una verdadera revolución en Internet. ***Java 1.1*** apareció a principios de 1997, mejorando sustancialmente la primera versión del lenguaje. ***Java 1.2***, más tarde rebautizado como ***Java 2***, nació a finales de 1998.

2.- El compilador de Java

Existen distintos programas comerciales que permiten desarrollar código ***Java***. La compañía ***Sun***, creadora de ***Java***, distribuye gratuitamente el *Java(tm) Development Kit* (***JDK***). Se trata de un conjunto de programas y librerías que permiten desarrollar, compilar y ejecutar programas en ***Java***.

Incorpora además la posibilidad de ejectar parcialmente el programa, deteniendo la ejecución en el punto deseado y estudiando en cada momento el valor de cada una de las variables (con el denominado ***Debugger***). Cualquier programador con un mínimo de experiencia sabe que una parte muy importante (muchas veces la mayor parte) del tiempo destinado a la elaboración de un programa se destina a la ***detección y corrección de errores***. Existe también una versión reducida del ***JDK***, denominada ***JRE*** (*Java Runtime Environment*) destinada únicamente a ejecutar código ***Java*** (no permite compilar).

Los ***IDE***s (*Integrated Development Environment*), tal y como su nombre indica, son entornos de desarrollo integrados. En un mismo programa es posible escribir el código ***Java***, compilarlo y ejecutarlo sin tener que cambiar de aplicación.

3.- Introducción al lenguaje Java

3.1.- Variables

Una ***variable*** es un ***nombre*** que contiene un valor que puede cambiar a lo largo del programa.

Nombres de Variables

Los nombres de variables en ***Java*** se pueden crear con mucha libertad. Pueden ser cualquier conjunto de caracteres numéricos y alfanuméricos, sin algunos caracteres especiales utilizados por ***Java*** como operadores o separadores (,.+-*/ etc.).

Existe una serie de ***palabras reservadas*** las cuales tienen un significado especial para ***Java*** y por lo tanto no se pueden utilizar como nombres de variables. Dichas palabras son:

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

Tipos Primitivos de Variables

Se llaman ***tipos primitivos*** de variables de ***Java*** a aquellas variables sencillas que contienen los tipos de información más habituales: valores *boolean*, *caracteres* y *valores numéricos* enteros o de punto flotante.

Java dispone de ocho tipos primitivos de variables: un tipo para almacenar valores ***true*** y ***false*** (***boolean***); un tipo para almacenar caracteres (***char***), y 6 tipos para guardar valores numéricos, cuatro tipos para enteros (***byte***, ***short***, ***int*** y ***long***) y dos para valores reales de punto flotante (***float*** y ***double***). Los rangos y la memoria que ocupa cada uno de estos tipos se muestran en la siguiente tabla:

Tipo de variable	Descripción
Boolean	1 byte. Valores true y false
Char	2 bytes. Unicode. Comprende el código ASCII
Byte	1 byte. Valor entero entre -128 y 127
Short	2 bytes. Valor entero entre -32768 y 32767
Int	4 bytes. Valor entero entre -2.147.483.648 y 2.147.483.647
Long	8 bytes. Valor entre -9.223.372.036.854.775.808 y 9.223.372.036.854.775.807
Float	4 bytes (entre 6 y 7 cifras decimales equivalentes). De -3.402823E38 a -1.401298E-45 y de 1.401298E-45 a 3.402823E38
Double	8 bytes (unas 15 cifras decimales equivalentes). De -1.79769313486232E308 a -4.94065645841247E-324 y de 4.94065645841247E-324 a 1.79769313486232E308

Los tipos primitivos de ***Java*** tienen algunas características importantes que se resumen a continuación:

1. El tipo ***boolean*** no es un valor numérico: sólo admite los valores ***true*** o ***false***. El tipo ***bolean*** no se identifica con el igual o distinto de cero, como en C/C++. El resultado de la expresión lógica que aparece como condición en un bucle o en una bifurcación debe ser ***boolean***.

2. El tipo ***char*** contiene caracteres en código UNICODE (que incluye el código ASCII), y ocupan 16 bits por carácter. Comprende los caracteres de prácticamente todos los idiomas.

3. Los tipos ***byte***, ***short***, ***int*** y ***long*** son números enteros que pueden ser positivos o negativos, con distintos valores máximos y mínimos. A diferencia de C/C++, en ***Java*** no hay enteros ***unsigned***.

4. Los tipos ***float*** y ***double*** son valores de punto flotante (números reales) con 6-7 y 15 cifras decimales equivalentes, respectivamente.

5. Se utiliza la palabra ***void*** para indicar la ausencia de un tipo de variable determinado.

6. A diferencia de C/C++, los tipos de variables en ***Java*** están perfectamente definidos en todas y cada una de las posibles plataformas. Por ejemplo, un ***int*** ocupa siempre la misma memoria y tiene el mismo rango de valores, en cualquier tipo de ordenador.

7. Existen extensiones de ***Java 1.2*** para aprovechar la arquitectura de los procesadores ***Intel***, que permiten realizar operaciones de punto flotente con una precisión extendida de 80 bits.

Cómo se definen e inicializan las variables

Una variable se define especificando el ***tipo*** y el ***nombre*** de dicha variable. Si no se especifica un valor en su declaración, las variable ***primitivas*** se inicializan a cero (salvo *boolean* y *char*, que se inicializan a ***false*** y '\0').

`Ejemplos de declaración e inicialización de variables:`

```
int x; // Declaración de la variable primitiva x. Se
inicializa a 0
int y = 5; // Declaración de la variable primitiva y. Se
inicializa a 5
```

3.2.- Operadores de Java

Operadores aritméticos

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

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. La Tabla 2.2 muestra estos operadores y su equivalencia con el uso del ***operador igual*** (=).

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

Tabla 2.2. Otros operadores de asignación.

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.

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:

1. *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.
2. *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.

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. La Tabla 2.3 muestra los operadores relacionales de ***Java***.

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

Tabla 2.3. Operadores relacionales.

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***. La Tabla 2.4 muestra los operadores lógicos de ***Java***. 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	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

Tabla 2.4. Operadores lógicos.

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 unas unidades 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.

Precedencia de operadores

El orden en que se realizan las operaciones es fundamental para determinar el resultado de una expresión. Por ejemplo, el resultado de *x/y*z* depende de qué operación (la división o el producto) se realice primero. La siguiente lista muestra el orden en que se ejecutan los distintos operadores en un sentencia, de ***mayor a menor*** precedencia:

1) Operadores incfrementale:++,--
2) Operadores unarios:+,-
3) Operadores de multiplicación/division
4) Operadores de suma/resta
5) Operadores relacionales
6) Operadores lógicos
7) Operadores de asignación

3.3.- Estructuras de programación

Sentencias o expresiones

Una ***expresión*** es un conjunto variables unidos por ***operadores***. Son órdenes que se le dan al computador 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. Por ejemplo:

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

Comentarios

Los comentarios son tremendamente útiles para poder entender el código utilizado, facilitando de ese modo futuras revisiones y correcciones. Además permite que cualquier persona distinta al programador original pueda comprender el código escrito de una forma más rápida. Se recomienda acostumbrarse a comentar el código desarrollado. De esta forma se simplifica también la tarea de estudio y revisión posteriores.

Java interpreta que todo lo que aparece a la derecha de dos barras "//" en una línea cualquiera del código es un comentario del programador y no lo tiene en cuenta. El comentario puede empezar al comienzo de la línea o a continuación de una instrucción que debe ser ejecutada. La segunda forma de incluir comentarios consiste en escribir el texto entre los símbolos /*…*/. Este segundo método es válido para comentar más de una línea de código. Por ejemplo:

```
// Esta línea es un comentario
int a=1; // Comentario a la derecha de una sentencia
// Esta es la forma de comentar más de una línea utilizando
// las dos barras. Requiere incluir dos barras al comienzo de
cada línea
/* Esta segunda forma es mucho más cómoda para comentar un
número elevado de líneas ya que sólo requiere modificar el
comienzo y el final. */
```

Bifurcaciones (Estructuras selectivas)

Bifurcación if

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***). Tiene la forma siguiente:

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

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***.

Bifurcación if else

Análoga a la anterior, de la cual es una ampliación. Las sentencias incluidas en el ***else*** se ejecutan en el caso de no cumplirse la expresión de comparación (***false***),

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

Bifurcación if elseif else

Permite introducir más de una expresión de comparación. 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 última ***else***.

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

Sentencia switch

Se trata de una alternativa a la bifurcación ***if elseif else*** 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:

1. Cada sentencia ***case*** se corresponde con un único valor de ***expression***. No se pueden establecer rangos o condiciones sino que se debe comparar con valores concretos.
2. Los valores no comprendidos en ninguna sentencia ***case*** se pueden gestionar en ***default***, que es opcional.
3. En ausencia de ***break***, cuando se ejecuta una sentencia ***case*** se ejecutan también todas las ***case*** ue van a continuación, hasta que se llega a un ***break*** o hasta que se termina el ***switch***.

Ejemplo:

```
char c = (char)(Math.random()*26+'a'); //
Generación aleatoria de letras minúsculas
System.out.println("La letra " + c );
switch (c) {
          case 'a': // Se compara con la letra a
          case 'e': // Se compara con la letra e
          casee 'i': // Se compara con la letra i
          case 'o': // Se compara con la letra o
          case 'u': // Se compara con la letra u
               System.out.println(" Es una vocal
");             break;
          default:
          System.out.println(" Es una consonante
");
```

Bucles (Estructuras repetitivas)

Bucle while

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

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

Bucle do while

Es similar al bucle ***while*** pero con la particularidad de que el control está al final del bucle (lo que hace que el bucle 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 bucle, mientras que si la condición se evalúa a ***false*** finaliza el bucle.

Este tipo de bucles se utiliza con frecuencia para controlar la satisfacción de una determinada condición de error o de convergencia.

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

Bucle for

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

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

que es equivalente a utilizar ***while*** en la siguiente forma,

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

La sentencia o sentencias ***initialization*** se ejecuta al comienzo del ***for***, e ***increment*** después de ***statements***. La ***booleanExpression*** se evalúa al comienzo de cada iteración; el bucle termina cuando la expresión de comparación toma el valor ***false***. Cualquiera de las tres partes puede estar vacía.

Sentencias break y continue

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

La sentencia ***continue*** se utiliza en los bucles (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).