CONCEPTOS BASICOS DEL LENGUAJE JAVA

NOMENCLATURA GENERAL

- En Java **se distinguen** las letras mayúsculas y minúsculas.

- Las reglas del lenguaje respecto a los nombres de variables son muy amplias y permiten mucha libertad al programador, pero es habitual seguir ciertas normas que facilitan la lectura y el mantenimiento de los programas. Se recomienda seguir las siguientes instrucciones:

- En Java es habitual utilizar nombres con minúsculas, con las excepciones que se indican a continuación:
  - Cuando un nombre consta de varias palabras es habitual poner una a continuación de otra, poniendo con mayúscula la primera letra de la palabra que sigue a otra.
  - **Los nombres de clases e interfaces comienzan siempre con mayúscula**.
  - **Los nombres de objetos, los nombres de métodos y variables miembro, y los nombres de las variables locales de los métodos, comienzan siempre con minúscula.**

PALABRAS RESERVADAS DEL LENGUAJE

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

VARIABLES

- Las variables son posiciones de memoria donde se almacenan valores durante la ejecución de un programa, dicho valor puede cambiar a lo largo del mismo.

- De acuerdo con el tipo de información que contienen, en Java hay dos tipos principales de variables:

-**Variables de tipos de datos primitivos**: Están definidas mediante un valor único. Ej.: int, float, etc.

-**Variables de referencia**: Las variables de referencia son referencias a objetos de una determinada clase. Ej.: String, Integer, Date, etc.

- Desde el punto de vista de su rol 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 { }. Se crean en el interior del bloque y se destruyen al finalizar dicho bloque. Pueden ser también de tipos primitivos o referencias.

- Java es un lenguaje "fuertemente tipado" por lo que es necesario especificar el tipo de dato para cada una de las variables que se vayan a utilizar en un programa.

- El tipo de dato de una variable determina los valores que puede almacenar y las operaciones que se pueden hacer sobre ella. Para dar a una variable un tipo y un nombre, se utiliza una instrucción de declaración de la variable.

- Antes de poder usar una variable, es necesario declararla especificando un nombre y un tipo de dato. También debe ser inicializada en forma explicita. Puede inicializarse en la misma instrucción de declaración o posteriormente.

DECLARACION

tipo_dato nom_identificador [, nom_identificador];
tipo_dato nom_identificador [= valor] [, nom_identificador [= valor] ...];

Ejemplos:

```
Int  indice;
int num1,num2,num3;
int num1=92, num2=46;
String cadena = "Aquí va una cadena de caracteres";
```

VARIABLES MIEMBRO DE UNA CLASE O VARIABLES DE INSTANCIA

- Las variables de instancia se declaran dentro del cuerpo de la clase.

- Ambito: Las variables miembro de una ***clase*** son válidas mientras existe el objeto creado a partir de dicha clase.

```
public class Test
{
        private String cadena = "Cadena";
        public void metodo()
        {
           cadena = cadena + " de caracteres";
        }
}
```

VARIABLES LOCALES

- Se declaran y se utilizan dentro de las definiciones de los métodos o de un bloque. Un bloque es un conjunto de sentencias relacionadas encerradas entre llaves.

- Ámbito: Una variable podrá ser utilizada únicamente dentro del bloque en donde fue declarada y en los bloques anidados dentro del mismo. Una vez que el bloque se termina de ejecutar, la variable sale de su ámbito y deja de existir en memoria.

```
public class Test {
        public void metodo()
        {
          String cadenaLocal = "Cadena Local";
          int cantidadCaracteres = 15;
        }
}
```

VARIABLES DE REFERENCIAS

- Existen otros tipos de datos además de los denominados "primitivos" que son aquellos que se crean como objetos de las siguientes clases:

String
Integer
Double
Float
Char
Date

- Cada uno de los objetos creados a partir de estas clases posee métodos para manejar y comparar los datos que almacenan, brindando una mayor funcionalidad que los tipos primitivos. Están almacenadas dentro del paquete java.util.*

- También los arreglos y las interfaces son del tipo referencia. Representan una referencia a (la dirección de) el valor o conjunto de valores representados por la variable.

VARIABLES DE TIPOS DE DATOS PRIMITIVOS

- Una variable de tipo primitivo contiene un único valor de tamaño y formato apropiados para su tipo: un número entero, un número decimal, un carácter, un valor booleano, etc.

TIPOS DE DATOS PRIMITIVOS

TIPO	DESCRIPCION
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 de1.401298E-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

DECLARACION DE CONSTANTES

- Una variable de un tipo primitivo declarada como *final* no puede cambiar su valor a lo largo de la ejecución del programa. Puede ser considerada como una constante, y equivale a la palabra const de otros lenguajes. Debe ser inicializada en el mismo momento de la declaración.

```
public class Test

{
  public void metodo()
  {
        final double salarioPersona = 5300.50;

        ...
   }
}
```

LITERALES

- Una literal es un valor especificado en el código fuente.
- Las literales representan valores primitivos o cadenas de caracteres.
- A continuación se describen los siguientes tipos de literales:

- Numéricos
  -Enteros
  -Punto Flotante
- Booleanos
- Caracteres
  -Caracter simple
  -Cadenas de caracteres

LITERALES NUMERICOS

- ***Enteros***

  Tipo int (número entero) Ej.: **15**
  Tipo long (número entero seguido de la letra L) Ej.: **239562L**

  - También pueden ser expresados en notación:
  Octal: prefijando el número con 0.
  Hexadecimal: prefijando el número con 0x o 0X.

- ***Punto flotante***

- Expresa un número decimal de punto flotante. Debe cumplir con alguna de las siguientes condiciones:
  - Tener punto decimal: 1.23 (asume que es double)
  - Tener punto decimal y el sufijo F o f para indicar float: 1.23f, o el sufijo D o d para indicar double: 1.23d.
  - Tener la letra E ó e, para indicar notacion cientifica: 4.23E+21

LITERALES DE CARACTERES

- ***Un caracter simple:*** Encerrado entre comillas simples.

Ejemplo:

```
private char car1;
car1= 'c';
```

-Existen algunos caracteres especiales que requieren una notación distinta:

'\n' – nueva linea.
'\r' – retorno de carro.
'\t' – tab.
'\'' – comillas simples.
'\\' – barra invertida.
'\"' – comillas dobles.

- ***Cadenas de caracteres***: Encerrados entre comillas dobles.

Ejemplo:

```
private String cad1;
cad1 = "esto es una cadena de caracteres";
```

LITERALES BOOLEANOS

- ***true***
- ***false***

Ejemplo:
private boolean marca;
marca = **true**;

EXPRESIONES

- Una expresión es una serie de variables, operadores y llamadas a métodos (construidas de acuerdo a la sintaxis del lenguaje) que devuelven un único valor.

- El tipo de dato del valor regresado por una expresión depende de los elementos usados en la expresión

- Podemos escribir expresiones compuestas combinando expresiones simples. Cuando escribimos expresiones compuestas, debemos ser explícitos e indicar con paréntesis que operadores se deben evaluar primero. Si elegimos no utilizar paréntesis, luego la plataforma Java evaluará la expresión compuesta en el orden dictado por la precedencia de los operadores.

OPERADORES

- Los operadores son símbolos especiales que comunmente se utilizan en expresiones. Se pueden clasificar de la siguiente manera:

- **Operadores aritméticos**: Son operadores binarios (requieren siempre dos operandos) que realizan las operaciones aritméticas habituales:

suma (**+**)
resta (**-**)
multiplicación (*****)
división (**/**)
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.

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.

-Los operadores (++) y (--) unarios sirven para incrementar o decrementar una unidad respectivamente. Pueden prefijar o seguir a la variable.

Ejemplo:

```
public int incrementar(){
        //defino variable local dentro del bloque
        int var;
        //inicializo dicha variable
        var=10;
        //incremento 1 unidad al valor de dicha variable
        var++;
        return var;
}
```

- **Operadores relacionales:** Sirven para realizar comparaciones de igualdad, desigualdad y relación de menor o mayor. El resultado de estos operadores es siempre un valor ***booleano*** (***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

- **Operadores lógicos**: Se utilizan para construir ***expresiones lógicas***, combinando valores lógicos. La diferencia entre && y & y entre || y |, es que estos últimos evalúan siempre los dos operandos.

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
!	NOT	! 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

- **Operadores de String:** El operador (+) sirve para concatenar cadenas de caracteres.

Ejemplo:
```
String cad1,cad2;
cad1="Esto es una ";
cad2="cadena de caracteres unida";
System.out.println(cad1 + cad2);
```

- 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 izquierda se copia sobre la variable de la derecha.

COMENTARIOS EN JAVA

- Existen comentarios de una línea.

```
//Shopping shopping=new Shopping("ABASTO SHOPPING");
```

- Existen comentarios multilínea.

```
/*Local local1,local2,local3,local4,local5;
        local1=new Bar();
        local1.setNombre("LA CASCADA");
        local1.setTipoLocal("bar");
*/
```

- Existen comentarios de documentación.

```
/**
 * @author Perez Juan
 *
 */
```

ESTRUCTURAS DE PROGRAMACION

- Las ***estructuras de programación*** o ***estructuras de control*** permiten tomar decisiones y realizar un proceso repetidas veces. Son los denominados ***bifurcaciones*** y ***bucles***. 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.

BIFURCACIONES

- Las bifurcaciones o estructuras alternativas 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***
    - ***switch***

BIFURCACION IF

- ***BIFURCACION 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***). 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***. Tiene la siguiente forma:

```
if (ExpresionLogica) {
        sentencias;
}
```

- ***BIFURCACION 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***). Tiene la siguiente forma:

```
if (ExpresionLogica) {
        sentencias1;
}
else {
        sentencias2;
}
```

BIFURCACION IF

- ***BIFURCACION 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 las sentencias correspondientes al else. La sintaxis es la siguiente:

```
if (ExpresionLogica1) {
        sentencias1;
} else if (ExpresionLogica2) {
        sentencias2;
} else if (ExpresionLogica3) {
        sentencias3;
}
else {
        sentencias4;
}
```

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(variable){
    case valor1:
       sentencias;
       break;
    case valor2:
      sentencias;
      break;
       ...
default:
sentencias;
}
```

Ejemplo:

```
char car1 = 'm';
switch (car1) {
  case 'a':
        System.out.println(" Es la vocal a ");
        break;
  case 'e':
        System.out.println(" Es la vocal e ");
        break;
  case 'i':
        System.out.println(" Es la vocal i ");
        break;
  case 'o':
        System.out.println(" Es la vocal o ");
        break;
  case 'u':
        System.out.println(" Es la vocal u ");
        break;
  default:
        System.out.println(" Es una consonante "); }
```

BUCLES

- Un ***bucle*** se utiliza para realizar un proceso repetidas veces. Se denomina también ***estructura repetitiva*** o ***loop***. El código incluido entre las ***llaves*** {} (opcionales si el proceso repetitivo consta de una sola línea), se ejecutará mientras se cumpla unas determinadas condiciones. Hay que prestar especial atención a los bucles infinitos, hecho que ocurre cuando la condición de finalizar el bucle (***ExpresionLogica***) no se llega a cumplir nunca. Se trata de un fallo muy típico, habitual sobre todo entre programadores poco experimentados.
    - ***while***
    - ***do while***
    - ***for***
- ***BUCLE while***: Las sentencias que estan dentro del bloque {} se ejecutan mientras la expresion logica sea true.

```
while (ExpresionLogica) {
        sentencias;
}
```

- ***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 ejecutadas las ***sentencias***, 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.

```
do {
    sentencias
} while (ExpresionLogica);
```

- ***BUCLE for***: La ***inicializacion*** se ejecuta al comienzo del ***for***, e ***incremento*** después de ejecutarse las ***sentencias***. La ***ExpresionLogica*** 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. La ***inicializacion*** y el ***incremento*** pueden tener varias expresiones separadas por comas. La forma general del bucle ***for*** es la siguiente.

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

Ejemplo*:*

```
for(int i = 1, j = i + 10; i < 5; i++, j = 2*i) {
        System.out.println(" i = " + i + " j = " + j);
}
```

SENTENCIAS BREAK Y CONTINUE

- La sentencia ***break*** es válida tanto para las bifurcaciones como para los bucles. Hace que se salga inmediatamente de la repetitiva o bloque que se está ejecutando sin finalizar el resto de las sentencias.

- La sentencia ***continue*** se utiliza en los bucles (no en bifurcaciones). Finaliza la iteración 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.