Tema 4: Características del Lenguaje

4.1. Variables y Tipos de Datos.-

Las variables son las partes importantes de un lenguaje de programación: ellas son las entidades (valores, datos) que actúan y sobre las que se actúa.

Una declaración de variable siempre contiene dos componentes, el tipo de la variable y su nombre:

```
tipoVariable nombre;
```

4.1.1.Tipos de Variables.-

Todas las variables en el lenguaje Java deben tener un tipo de dato. El tipo de la variable determina los valores que la variable puede contener y las operaciones que se pueden realizar con ella.

Existen dos categorias de datos principales en el lenguaje Java: los tipos primitivos y los tipos referenciados.

Los tipos primitivos contienen un sólo valor e incluyen los tipos como los enteros, coma flotante, los caracteres, etc... La tabla siguiente muestra todos los tipos primitivos soportados por el lenguaje Java, su formato, su tamaño y una breve descripción de cada uno:

Tipo	Tamaño/Formato	Descripción
(Números enteros)		
byte	8-bit complemento a 2	Entero de un Byte
short	16-bit complemento a 2	Entero corto
int	32-bit complemento a 2	Entero
long	64-bit complemento a 2	Entero largo
(Números reales)		
float	32-bit IEEE 754	Coma flotante de precisión simple
double	64-bit IEEE 754	Coma flotante de precisión doble
(otros tipos)		
char	16-bit Caracter	Un sólo carácter
boolean	true o false	Un valor booleano (verdadero o falso)

Los tipos referenciados se llaman así porque el valor de una variable de referencia es una referencia (un puntero) hacia el valor real. En Java tenemos los arrays, las clases y los interfaces como tipos de datos referenciados.

4.2. Nombres de Variables.-

Un programa se refiere al valor de una variable por su nombre. Por convención, en Java, los nombres de las variables empiezan con una letra minúscula (los nombres de las clases empiezan con una letra mayúscula).

Un nombre de variable Java:

1. Debe ser un identificador legal de Java comprendido en una serie de caracteres Unicode. Unicode es un sistema de codificación que soporta texto escrito en distintos lenguajes humanos.Unicode perminte la codificación de 34.168 caracteres. Esto le permite utilizar en sus programas Java varios alfabetos como el Japonés, el Griego, el Ruso o el Hebreo. Esto es importante para que los programadores pueden escribir código en su lenguaje nativo.

2. No puede ser el mismo que una palabra clave o el nombre de un valor booleano (true or false).

3. No deben tener el mismo nombre que otras variables cuyas declaraciones aparezcan en el mismo ámbito.

La regla número 3 implica que podría existir el mismo nombre en otra variable que aparezca en un ámbito diferente.

Por convención, los nombres de variables empiezan por un letra minúscula. Si una variable está compuesta de más de una palabra, como 'nombreDato' las palabras se ponen juntas y cada palabra después de la primera empieza con una letra mayúscula.

4.3. Operadores.-

Los operadores realizan algunas funciones en uno o dos operandos. Los operadores que requieren un operador se llaman operadores unarios. Por ejemplo, ++ es un operador unario que incrementa el valor su operando en uno.

Los operadores que requieren dos operandos se llaman operadores binarios. El operador = es un operador binario que asigna un valor del operando derecho al operando izquierdo.

Los operadores unarios en Java pueden utilizar la notación de prefijo o de sufijo. La notación de prefijo significa que el operador aparece antes de su operando:

```
operador operando
```

La notación de sufijo significa que el operador aparece después de su operando:

```
operando operador
```

Todos los operadores binarios de Java tienen la misma notación, es decir aparencen entre los dos operandos:

```
op1 operator op2
```

Además de realizar una operación también devuelve un valor. El valor y su tipo dependen del tipo del operador y del tipo de sus operandos. Por ejemplo, los operadores aritméticos (realizan las operaciones de aritmética básica como la suma o la resta) devuelven números, el resultado típico de las operaciones aritmétcias.

El tipo de datos devuelto por los operadores aritméticos depende del tipo de sus operandos: si sumas dos enteros, obtendrás un entero. Se dice que una operación evalúa su resultado.

Es muy útil dividir los operadores Java en las siguientes categorías: aritméticos, relacionales y condicionales. lógicos y de desplazamiento y de asignación.

4.3.1. Operadores Aritméticos.-

El lenguaje Java soporta varios operadores aritéticos - incluyendo + (suma), - (resta), * (multiplicación), / (división), y % (módulo)-- en todos los números enteros y de coma flotante. Por ejemplo, puedes utilizar este código Java para sumar dos números:

```
sumaEsto + aEsto
```

O este código para calcular el resto de una división:

```
divideEsto % porEsto
```

Esta tabla sumariza todas las operaciones aritméticas binarias en Java:

Operador	Uso	Descripción
+	op1 + op2	Suma op1 y op2
-	op1 - op2	Resta op2 de op1
*	op1 * op2	Multiplica op1 y op2
/	op1 / op2	Divide op1 por op2
%	op1 % op2	Obtiene el resto de dividir op1 por op2

Nota: El lenguaje Java extiende la definición del operador + para incluir la concatenación de cadenas.

Los operadores + y - tienen versiones unarias que seleccionan el signo del operando:

Operador	Uso	Descripción
+	+op	Indica un valor positivo
-	- op	Niega el operando

Además, existen dos operadores de atajos aritméticos, ++ que incrementa en uno su operando, y -- que decrementa en uno el valor de su operando.

Operador	Uso	Descripción
++	op ++	Incrementa op en 1; evalúa el valor antes de incrementar
++	++ op	Incrementa op en 1; evalúa el valor después de incrementar
--	op --	Decrementa op en 1; evalúa el valor antes de decrementar
--	-- op	Decrementa op en 1; evalúa el valor después de decrementar

4.3.2. Operadores Relacionales y Condicionales.-

Los valores relacionales comparan dos valores y determinan la relación entre ellos. Por ejemplo, != devuelve true si los dos operandos son distintos.

Esta tabla sumariza los operadores relacionales de Java:

Operador	Uso	Devuelve true si
>	op1 > op2	op1 es mayor que op2
>=	op1 >= op2	op1 es mayor o igual que op2
<	op1 < op2	op1 es menor que op2
<=	op1 <= op2	op1 es menor o igual que op2
==	op1 == op2	op1 y op2 son iguales
!=	op1 != op2	op1 y op2 son distintos

Frecuentemente los operadores relacionales se utilizan con otro juego de operadores, los operadores condicionales, para construir expresiones de decisión más complejas. Uno de estos operadores es && que realiza la operación Y booleana . Por ejemplo puedes utilizar dos operadores relacionales diferentes junto con && para determinar si ambas relaciones son ciertas. La siguiente línea de código utiliza esta técnica para determinar si un indice de un array está entre dos límites- esto es, para determinar si el indice es mayor que 0 o menor que NUM_ENTRIES (que se ha definido préviamente como un valor constante):

```
0 < index && index < NUM_ENTRIES
```

Observa que en algunas situaciones, el segundo operando de un operador relacional no será evaluado. Consideremos esta sentencia:

```
((count > NUM_ENTRIES) && (System.in.read() != -1))
```

Si count es menor que NUM_ENTRIES, la parte izquierda del operando de && evalúa a false. El operador && sólo devuelve true si los dos operandos son

verdaderos. Por eso, en esta situación se puede deteminar el valor de && sin evaluar el operador de la derecha. En un caso como este, Java no evalúa el operando de la derecha. Así no se llamará a `System.in.read()` y no se leerá un carácter de la entrada estandard.

Aquí tienes tres operadores condicionales:

Operador	Uso	Devuelve true si
&&	op1 && op2	op1 y op2 son verdaderos
||	op1 || op2	uno de los dos es verdadero
!	! op	op es falso

El operador & se puede utilizar como un sinónimo de && si ambos operadores son booleanos. Similarmente, | es un sinonimo de || si ambos operandos son booleanos.

4.3.3. Operadores de Desplazamiento.-

Los operadores de desplazamiento permiten realizar una manipualción de los bits de los datos. Esta tabla sumariza los operadores lógicos y de desplazamiento disponibles en el lenguaje Java:

Operador	Uso	Descripción
>>	op1 >> op2	desplaza a la derecha op2 bits de op1
<<	op1 << op2	desplaza a la izquierda op2 bits de op1
>>>	op1 >>> op2	desplaza a la derecha op2 bits de op1(sin signo)
&	op1 & op2	bitwise and
|	op1 | op2	bitwise or
^	op1 ^ op2	bitwise xor
~	~ op	bitwise complemento

Los tres operadores de desplazamiento simplemente desplazan los bits del operando de la izquierda el número de posiciones indicadas por el operador de la derecha. Los desplazamientos ocurren en la dirección indicada por el propio operador. Por ejemplo:

```
13 >> 1;
```

desplaza los bits del entero 13 una posición a la derecha. La representación binaria del número 13 es 1101. El resultado de la operación de desplazamiento es 110 o el 6 decimal. Observe que el bit situado más a la derecha desaparece. Un desplazamiento a la derecha de un bit es equivalente, pero más eficiente que, dividir el operando de la izquierda por dos. Un desplazamiento a la izquierda es equivalente a multiplicar por dos.

Los otros operadores realizan las funciones lógicas para cada uno de los pares de bits de cada operando. La función "y" activa el bit resultante si los dos operandos son 1.

op1	op2	resultado
0	0	0
0	1	0
1	0	0
1	1	1

Supon que quieres evaluar los valores 12 "and" 13:

```
12 & 13
```

El resultado de esta operación es 12. ¿Por qué? Bien, la representación binaria de 12 es 1100 y la de 13 es 1101. La función "and" activa los bits resultantes cuando los bits de los dos operandos son 1, de otra forma el resultado es 0. Entonces si colocas en línea los dos operandos y realizas la función "and", puedes ver que los dos bits de mayor peso (los dos bits situados más a la izquierda de cada número) son 1 así el bit resultante de cada uno es 1. Los dos bits de menor peso se evalúan a 0 poque al menos uno de los dos operandos es 0:

```
 1101
&1100
------
 1100
```

El operador | realiza la operación O inclusiva y el operador ^ realiza la operación O exclusiva. O inclusiva significa que si uno de los dos operandos es 1 el resultado es 1.

op1	op2	resultado
0	0	0
0	1	1
1	0	1
1	1	1

O exclusiva significa que si los dos operandos son diferentes el resultado es 1, de otra forma el resultado es 0:

op1	op2	resultado
0	0	0
0	1	1
1	0	1
1	1	0

Y finalmente el operador complemento invierte el valor de cada uno de los bites del operando: si el bit del operando es 1 el resultado es 0 y si el bit del operando es 0 el resultado es 1.

4.3.4. Operadores de Asignación.-

Puedes utilizar el operador de asignación =, para asignar un valor a otro. Además del operador de asignación básico, Java proporciona varios operadores de asignación que permiten realizar operaciones aritmétcias, lógicas o de bits y una operación de asignación al mismo tiempo.

Especificamente, supon que quieres añadir un número a una variable y asignar el resultado dentro de la misma variable, como esto:

```
i = i + 2;
```

Puedes ordenar esta sentencia utilizando el operador +=.

i += 2;

Las dos líneas de código anteriores son equivalentes.

Esta tabla lista los operadores de asignación y sus equivalentes:

Operador	Uso	Equivale 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
|=	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

4.4. Expresiones.-

Las expresiones realizan el trabajo de un programa Java. Entre otras cosas, las expresiones se utilizan para calcular y asignar valores a las variables y para controlar el flujo de un programa Java. El trabajo de una expresión se divide en dos partes: realizar los cálculos indicados por los elementos de la expresión y devolver algún valor.

Una expresión es una serie de variables, operadores y llamadas a métodos (construida de acuerdo a la sintaxis del lenguaje) que evalúa a un valor sencillo.

El tipo del dato devuelto por una expresión depende de los elementos utilizados en la expresión. La expresión count++ devuelve un entero porque ++ devuelve un valor del mismo tipo que su operando y count es un entero. Otras expresiones

devuelven valores booleanos, cadenas, etc...

Una expresión de llamada a un método devuelve el valor del método; así el tipo de dato de una expresión de llamada a un método es el mismo tipo de dato que el valor de retorno del método. El método System.in.read() se ha declarado como un entero, por lo tanto, la expresión System.in.read() devuelve un entero.

La segunda expresión contenida en la sentencia System.in.read() != -1 utiliza el operador !=. Recuerda que este operador comprueba si los dos operandos son distintos. En esta sentencia los operandos son System.in.read() y -1.

System.in.read() es un operando válido para != porque devuelve un entero. Así System.in.read() != -1 compara dos enteros, el valor devuelto por System.in.read() y -1. El valor devuelto por != es true o false dependiendo de la salida de la comparación.

Como has podido ver, Java te permite construir expresiones compuestas y sentencias a partir de varias expresiones pequeñas siempre que los tipos de datos requeridos por una parte de la expresión correspondan con los tipos de datos de la otra. Tambíen habrás podido concluir del ejemplo anterior, el orden en que se evalúan los componentes de una expresión compuesta. Por ejemplo, toma la siguiente expresión compuesta:

```
x * y * z
```

En este ejemplo particular, no importa el orden en que se evalúe la expresión porque el resultado de la multiplicación es independiente del orden. La salida es siempre la misma sin importar el orden en que se apliquen las multiplicaciones.

Sin embargo, esto no es cierto para todas las expresiones. Por ejemplo, esta expresión obtiene un resultado diferente dependiendo de si se realiza primero la suma o la división:

```
x + y / 100
```

Puedes decirle directamente al compilador de Java cómo quieres que se evalúe una expresión utilizando los paréntesis (y). Por ejemplo, para aclarar la sentencia anterior, se podría escribir: (x + y)/ 100.

Si no le dices explícitamente al compilador el orden en el que quieres que se realicen las operaciones, él decide basándose en la precedencia asignada a los operadores y otros elementos que se utilizan dentro de una expresión.

Los operadores con una precedencia más alta se evalúan primero. Por ejemplo. el operador división tiene una precedencia mayor que el operador suma, por eso, en la expresión anterior x + y / 100, el compilador evaluará primero y / 100. Así

```
x + y / 100
```

es equivalente a:

```
x + (y / 100)
```

Para hacer que tu código sea más fácil de leer y de mantener deberías explicar e indicar con parétesis los operadores que se deben evaluar primero.

La tabla siguiente muestra la precedencia asignada a los operadores de Java. Los operadores se han listado por orden de precedencia de mayor a menor. Los operadores con mayor precedencia se evalúan antes que los operadores con un precedencia relativamente menor. Lo operadores con la misma precedencia se evalúan de izquierda a derecha.

Precedencia de Operadores en Java

operadores sufijo	[] . (params) expr++ expr--
operadores unarios	++expr --expr +expr -expr ~ !
creación o tipo	new (type)expr
multiplicadores	* / %
suma/resta	+ -
desplazamiento	<< >> >>>
relacionales	< > <= >= instanceof
igualdad	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
AND lógico	&&
OR lógico	||
condicional ? :	
asignación	= += -= *= /= %= ^= &= |= <<= >>= >>>=

4.5. Sentencias de Control de Flujo.-

Las sentencias de control de flujo determinan el orden en que se ejecutarán las otras sentencias dentro del programa. El lenguaje Java soporta varias sentencias de control de flujo, incluyendo:

Sentencias	palabras clave
toma de decisiones	if-else, switch-case
bucles	for, while, do-while
miscelaneas	break, return

4.5.1. La sentencia if-else.-

La sentencia if-else de java proporciona a los programas la posibilidad de ejecutar selectivamente otras sentencias basándose en algún criterio. Por ejemplo, supon que tu programa imprime información de depurado basándose en el valor de una

variable booleana llamada DEBUG. Si DEBUG fuera verdadera true, el programa imprimiría la información de depurado, como por ejemplo, el valor de una variable como x. Si DEBUG es false el programa procederá normalmente. Un segmento de código que implemente esto se podría parecer a este:

```
. . .
if (DEBUG)
     System.out.println("DEBUG: x = " + x);
. . .
```

Esta es la versión más sencilla de la sentencia if: la sentencia gobernada por if se ejecuta si alguna codición es verdadera. Generalmente, la forma sencilla de if se puede escribir así:

```
if (expresión)
     sentencia
```

Pero, ¿y si quieres ejecutar un juego diferente de sentencias si la expresión es falsa? Bien, puedes utilizar la sentencia else. Echemos un vistazo a otro ejemplo. Supon que tu programa necesita realizar diferentes acciones dependiendo de que el usuario pulse el botón OK o el botón Cancel en un ventana de alarma. Se podría hacer esto utilizando una sentencia if:

```
. . .
// Respuesta dependiente del botoón que haya pulsado el usuario
// OK o Cancel
. . .
if (respuesta == OK)
{
. . .
// Código para la acción OK
. . .
}
else
{
     . . .
     // código para la acción Cancel
     . . .
}
```

Este uso particular de la sentencia else es la forma de capturarlo todo. Existe otra forma de la sentecia else, else if que ejecuta una sentencia basada en otra expresión.

Por ejemplo, supon que has escrito un programa que asigna notas basadas en la puntuación de un examen, un Sobresaliente para una puntuación del 90% o superior, un Notable para el 80% o superior y demás. odrías utilizar una sentencia if con una serie de comparaciones else if y una setencia else para escribir este código:

```
int puntuacion;
String nota;

if (puntuacion >= 90)
{
     nota = "Sobresaliente";
}
else
     if (puntuacion >= 80)
     {
          nota = "Notable";
     }
     else
          if (puntuacion >= 70)
          {
               nota = "Bien";
          }
          else
               if (puntuacion >= 60)
               {
                    nota = "Suficiente";
               }
               else
               {
                    nota = "Insuficiente";
               }
```

Una sentencia if puede tener cualquier número de sentencias de acompañamiento else if. Podrías haber observado que algunos valores de puntuacion pueden satisfacer más una de las expresiones que componen la sentencia if. Por ejemplo, una puntuación de 76 podría evaluarse como true para dos expresiones de esta sentencia:puntuacion >= 70 y puntuacion >= 60. Sin embargo, en el momento de ejecución, el sistema procesa una sentencia if compuesta como una sóla; una vez que se ha satisfecho una condición (76 >= 70), se ejecuta la sentencia apropiada (nota = "Bien";), y el control sale fuera de la sentencia if sin evaluar las condiciones restantes.

4.5.2. La sentencia switch.-

La sentencia switch se utiliza para realizar sentencias condicionalmente basadas en alguna expresión. Por ejemplo, supon que tu programa contiene un entero llamado mes cuyo valor indica el mes en alguna fecha. Supon que también quieres mostrar el nombre del mes basándose en su número entero equivalente.

Podrías utilizar la sentencia switch de Java para realizar esta tarea:

```
int mes;
. . .
switch (mes)
{
     case 1:
               System.out.println("Enero");
               break;
     case 2:
               System.out.println("Febrero");
               break;
     case 3:
               System.out.println("Marzo");
               break;
     case 4:
               System.out.println("Abril");
               break;
     case 5:
               System.out.println("May0");
               break;
     case 6:
               System.out.println("Junio");
               break;
     case 7:
               System.out.println("Julio");
               break;
     case 8:   System.out.println("Agosto");
               break;
     case 9:   System.out.println("Septiembre");
               break;
     case 10:
               System.out.println("Octubre");
               break;
     case 11:
               System.out.println("Noviembre");
               break;
     case 12:
               System.out.println("Diciembre");
               break;
}
```

La sentencia switch evalúa su expresión, en este caso el valor de mes, y ejecuta la sentencia case apropiada. Decidir cuando utilizar las sentencias if o switch dependen del juicio personal. Puedes decidir cual utilizar basándose en la buena lectura del código o en otros factores.

Cada sentencia case debe ser única y el valor proporcionado a cada sentencia case debe ser del mismo tipo que el tipo de dato devuelto por la expresión proporcionada a la sentencia switch.

Otro punto de interes en la sentencia switch son las sentencias break después de cada case. La sentencia break hace que el control salga de la sentencia switch y continúe con la siguiente línea. La sentencia break es necesaria porque las sentencias case se siguen ejecutando hacia abajo. Esto es, sin un break explícito, el flujo de control seguiría secuencialmente a través de las sentencias case siguientes. En el ejemplo anterior, no se quiere que el flujo vaya de una sentencia case a otra, por eso se han tenido que poner las sentencias break. Sin embargo, hay ciertos escenario en los que querrás que el control proceda secuencialmente a través de las sentencias case. Como este código que calcula el número de días de un mes de acuerdo con el ritmico refrán que dice "Treinta tiene Septiembre...".

```
int mes;
int numeroDias;
. . .
switch (mes)
{
     case 1:
     case 3:
     case 5:
     case 7:
     case 8:
     case 10:
     case 12:
          numeroDias = 31;
          break;
     case 4:
     case 6:
     case 9:
     case 11:
          numeroDias = 30;
          break;
     case 2:
          if(((ano%4== 0) && !(ano % 100 == 0)) || ano % 400 == 0))
               numeroDias = 29;
          else
               numeroDias = 28;
          break;
}
```

Finalmente, puede utilizar la sentencia default al final de la sentencia switch para manejar los valores que no se han manejado explícitamente por una de las sentencias case.

```
int mes;
. . .
switch (mes)
{
     case 1:
```

```
            System.out.println("Enero");
            break;
      case 2:
            System.out.println("Febrero");
            break;

      …

      case 12:
            System.out.println("Diciembre");
            break;

      default:
            System.out.println("Ee, no es un mes válido!");
            break;
}
```

4.5.3. Sentencias de Bucle.-

Generalmente hablando, una sentencia while realiza una acción mientras se cumpla una cierta condición. La sintaxis general de la sentencia while es:

```
while (expresión)
      sentencia
```

Esto es, mientras la expresión sea verdadera, ejecutará la sentencia.

sentencia puede ser una sóla sentencia o puede ser un bloque de sentencias. Un bloque de sentencias es un juego de sentencias legales de java contenidas dentro de corchetes('{'y '}'). Por ejemplo, supon que además de incrementar contador dentro de un bucle while también quieres imprimir el contador cada vez que se lea un carácter. Podrías escribir esto en su lugar:

```
. . .
while (System.in.read() != -1)
{
      contador++;
      System.out.println("Se ha leido un el carácter = " + contador);
}
. . .
```

Por convención el corchete abierto '{' se coloca al final de la misma línea donde se encuentra la sentencia while y el corchete cerrado '}' empieza una nueva línea indentada a la línea en la que se encuentra el while.

Además de while Java tiene otros dos constructores de bucles que puedes utilizar en tus programas: el bucle for y el bucle do-while.

Primero el bucle for. Puedes utilizar este bucle cuando conozcas los límites del bucle (su instrucción de inicialización, su criterio de terminación y su instrucción de incremento). Por ejemplo, el bucle for se utiliza frecuentemente para iterar sobre los elementos de un array, o los caracteres de una cadena.

```
// a es un array de cualquier tipo
. . .
int i;
int length = a.length;
for (i = 0; i < length; i++)
{
     . . .
     // hace algo en el elemento i del array a
     . . .
}
```

Si sabes cuando estas escribiendo el programa que quieres empezar en el inicio del array, parar al final y utilizar cada uno de los elementos. Entonces la sentencia for es una buena elección. La forma general del bucle for puede expresarse asi:

```
for (inicialización; terminación; incremento)
     sentencias
```

inicialización es la sentencia que inicializa el bucle -- se ejecuta una vez al iniciar el bucle.

terminación es una sentecia que determina cuando se termina el bucle. Esta expresión se evalúa al principio de cada iteracción en el bucle. Cuando la expreión se evalúa a false el bucle se termina.

Finalmente, **incremento** es una expresión que se invoca en cada interacción del bucle.

Cualquiera (o todos) de estos componentes puden ser una sentencia vacía (un punto y coma).

Java proporciona otro bucle, el bucle do-while,que es similar al bucle while que se vió al principio, excepto en que la expresión se avalú al final del bucle:

```
do
{
     sentencias
}
while (Expresión Booleana);
```

La sentencia do-while se usa muy poco en la construcción de bucles pero tiene sus usos. Por ejemplo, es conveniente utilizar la sentencia do-while cuando el bucle debe ejecutarse al menos una vez. Por ejemplo, para leer información de un fichero, sabemos que al menos debe leer un carácter:

```
int c;
InputStream in;
. . .
do
{
     c = in.read();
     . . .
}
while (c != -1);
```

4.6. Arrays y Cadenas.-

Al igual que otros lenguajes de programación, Java permite juntar y manejar múltiples valores a través de un objeto array (matriz). También se pueden manejar datos compuestos de múltiples caracteres utilizando el objeto String (cadena).

4.6.1. Arrays.-

Esta sección te enseñará todo lo que necesitas para crear y utilizar arrays en tus programas Java.

Como otras variables, antes de poder utilizar un array primero se debe declarar. De nuevo, al igual que otras variables, la declaración de un array tiene dos componentes primarios: el tipo del array y su nombre. Un tipo de array incluye el tipo de dato de los elementos que va contener el array. Por ejemplo, el tipo de dato para un array que sólo va a contener elementos enteros es un array de enteros. No puede existir un array de tipo de datos genérico en el que el tipo de sus elementos esté indefinido cuando se declara el array. Aquí tienes la declaración de un array de enteros:

```
int[] arrayDeEnteros;
```

La parte int[] de la declaración indica que arrayDeEnteros es un array de enteros. La declaración no asigna ninguna memoria para contener los elementos del array.

Si se intenta asignar un valor o acceder a cualquier elemento de arrayDeEnteros antes de haber asignado la memoria para él, el compilador dará un error como este y no compilará el programa:

```
testing.java:64: Variable arraydeenteros may not have been initialized.
```

Para asignar memoria a los elementos de un array, primero se debe ejemplarizar el array. Se puede hacer esto utilizando el operador new de Java. (Realmente, los pasos que se deben seguir para crear un array son similares a los se deben seguir para crear un objeto de una clase: declaración, ejemplarización e inicialización.

La siguiente sentencia asigna la suficiente memoria para que arrayDeEnteros pueda contener diez enteros.

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

En general, cuando se crea un array, se utiliza el operador new, más el tipo de dato de los elementos del array, más el número de elementos deseados encerrado entre cochetes cuadrados ('[' y ']').

```
TipodeElemento[] NombredeArray = new TipodeElementos[tamanoArray];
```

Ahora que se ha asignado memoria para un array ya se pueden asignar valores a los elemetos y recuperar esos valores:

```
for (int j = 0; j < arrayDeEnteros.length; j ++)
{
     arrayDeEnteros[j] = j;
     System.out.println("[j] = " + arrayDeEnteros[j]);
}
```

Como se puede ver en el ejemplo anterior, para referirse a un elemento del array, se añade corchetes cuadrados al nombre del array. Entre los corchetes caudrados se indica (bien con una variable o con una expresión) el índice del elemento al que se quiere acceder. Observa que en Java, el índice del array empieza en 0 y termina en la longitud del array menos uno.

Hay otro elemento interesante en el pequeño ejemplo anterior. El bucle for itera sobre cada elemento de arrayDeEnteros asignándole valores e imprimiendo esos valores. Observa el uso de arrayDeEnteros.length para obtener el tamaño real del array. length es una propiedad proporcionada para todos los arrays de Java.

Los arrays pueden contener cualquier tipo de dato legal en Java incluyendo los tipos de referencia como son los objetos u otros array. Por ejemplo, el siguiente ejemplo declara un array que puede contener diez objetos String.

```
String[] arrayDeStrings = new String[10];
```

Los elementos en este array son del tipo referencia, esto es, cada elemento contiene una referencia a un objeto String. En este punto, se ha asignado suficiente memoria para contener las referencias a los Strings, pero no se ha asignado memoria para los propios strings. Si se intenta acceder a uno de los elementos de arraydeStrings obtendrá una excepción 'NullPointerException'

porque el array está vacio y no contiene ni cadenas ni objetos String. Se debe asignar memoria de forma separada para los objetos String:

```
for (int i = 0; i < arraydeStrings.length; i ++)
{
     arraydeStrings[i] = new String("Hello " + i);
}
```

4.6.2. Strings.-

Una secuencia de datos del tipo carácter se llama un string (cadena) y en el entorno Java está implementada por la clase String (un miembro del paquete java.lang).

```
String[] args;
```

Este código declara explícitamente un array, llamado args, que contiene objetos del tipo String. Los corchetes vacios indican que la longitud del array no se conoce en el momento de la compilación, porque el array se pasa en el momento de la ejecución.

El segundo uso de String es el uso de cadenas literales (una cadena de caracteres entre comillas " y "):

```
"Hola mundo!"
```

El compilador asigna implicitamente espacio para un objeto String cuando encuentra una cadena literal.

Los objetos String son inmutables - es decir, no se pueden modificar una vez que han sido creados. El paquete java.lang proporciona una clase diferente, StringBuffer, que se podrá utilizar para crear y manipular caracteres al vuelo.

4.6.3. Concatenación de Cadenas.-

Java permite concatenar cadenas facilmente utilizando el operador +. El siguiente fragmento de código concatena tres cadenas para producir su salida:

```
"La entrada tiene " + contador + " caracteres."
```

Dos de las cadenas concatenadas son cadenas literales: "La entrada tiene " y " caracteres.". La tercera cadena - la del medio es realmente un entero que primero se convierte a cadena y luego se concatena con las otras.