unemeb
Instituto de nuevas Tecnologías

Nivel Básico

JAVA 4

Objetivos de JAVA 5

Entornos de Funcionamiento 6

Máquina Virtual de JAVA 6

JRE 7

Enlace de descarga 8

Lenguaje de Programación (Programming Language) 9

Tipos de datos 10

Conceptos básicos 11

Clasificación para los tipos de datos 12

Operaciones de los tipos de datos 13

Algunos comentarios sobre tipos de datos y precedencia 13

Conversión de tipos 17

Estructuras de Control Condicional (Si y Selección) 20

Condicional Simple: Si – Entonces – FSi (*if*) 21

Condicional Compuesto. Si – Entonces – Sino – FSi (*if - else*) 21

Condicional Anidado 22

Selección Múltiple: Selección – Sino – FSelección (*switch*) 23

Estructuras de control iterativas 26

Bucles o Ciclos 27

Estructura iterativa Para… FinPara (*for … next … endfor*) 27

Estructura iterativa Mientras… Hacer … FinMientras (*while*) 29

Estructura iterativa Repetir… Hasta (*do - while*) 32

Procedimientos 34

Acciones 36

Funciones 38

Llamada (o invocación) de las acciones o funciones 40

Tipos y pases de parámetros 41

Tipos de datos estructurados 42

Estructura de datos (ED) 42

Arreglo 43

Registro 49

Archivo 50

¿Qué es Java?

Java es la base para prácticamente todos los tipos de aplicaciones de red, además del estándar global para desarrollar y distribuir aplicaciones móviles y embebidas, juegos, contenido basado en web y software de empresa. Con más de 9 millones de desarrolladores en todo el mundo, Java le permite desarrollar, implementar y utilizar de forma eficaz interesantes aplicaciones y servicios.
Desde portátiles hasta centros de datos, desde consolas para juegos hasta súper computadoras, desde teléfonos móviles hasta Internet, Java está en todas partes.

- El 97% de los escritorios empresariales ejecutan Java
- El 89% de los escritorios (o computadoras) en Estados Unidos ejecutan Java
- 9 millones de desarrolladores de Java en todo el mundo
- La primera opción para los desarrolladores
- La primera plataforma de desarrollo
- 3 mil millones de teléfonos móviles ejecutan Java
- El 100% de los reproductores de Blu-ray incluyen Java
- 5 mil millones de Java Cards en uso
- 125 millones de dispositivos de televisión ejecutan Java
- 5 de los 5 principales fabricantes de equipos originales utilizan Java ME

El lenguaje de programación Java fue originalmente desarrollado por James Gosling de Sun Microsystems (la cual fue adquirida por la compañía Oracle) y publicado en el 1995 como un componente fundamental de la plataforma Java de Sun Microsystems. Su sintaxis deriva mucho de C y C++, pero tiene menos facilidades de bajo nivel que cualquiera de ellos. Las aplicaciones de Java son generalmente compiladas a bytecode (clase Java) que puede ejecutarse en cualquier máquina virtual Java (JVM) sin importar la arquitectura de la computadora subyacente.

Es un lenguaje de programación de propósito general, concurrente, orientado a objetos y basado en clases que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible.

Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo (conocido en inglés como *WORA*, o "*write once, run anywhere*"), lo que quiere decir que el código que es ejecutado en una plataforma no tiene que ser recompilado para correr en otra. Java es, a partir del 2012, uno de los lenguajes de programación más populares en uso, particularmente para aplicaciones de cliente-servidor de web, con unos 10 millones de usuarios reportados.

OBJETIVOS DE JAVA

El lenguaje Java se creó con cinco objetivos principales:

1. Debería usar el paradigma de la programación orientada a objetos.

2. Debería permitir la ejecución de un mismo programa en múltiples sistemas operativos.

3. Debería incluir por defecto soporte para trabajo en red.

4. Debería diseñarse para ejecutar código en sistemas remotos de forma segura.

5. Debería ser fácil de usar y tomar lo mejor de otros lenguajes orientados a objetos, como C++.

Para conseguir la ejecución de código remoto y el soporte de red, los programadores de Java a veces recurren a extensiones como CORBA (Common Object Request Broker Architecture), Internet Communications Engine o OSGi respectivamente.

ENTORNOS DE FUNCIONAMIENTO

El diseño de Java, su robustez, el respaldo de la industria y su fácil portabilidad han hecho de Java uno de los lenguajes con un mayor crecimiento y amplitud de uso en distintos ámbitos de la industria de la informática:

1. En dispositivos móviles y sistemas empotrados
2. En el navegador web
3. En sistemas de servidor

4. En aplicaciones de escritorio
5. Plataformas soportadas

MÁQUINA VIRTUAL JAVA

Una máquina virtual Java (en inglés Java Virtual Machine, JVM) es una máquina virtual de proceso nativo, es decir, ejecutable en una plataforma específica, capaz de interpretar y ejecutar instrucciones expresadas en un código binario especial (el bytecode Java), el cual es generado por el compilador del lenguaje Java.

El código binario de Java no es un lenguaje de alto nivel, sino un verdadero código máquina de bajo nivel, viable incluso como lenguaje de entrada para un microprocesador físico. Como todas las piezas del rompecabezas Java, fue desarrollado originalmente por Sun.

La JVM es una de las piezas fundamentales de la plataforma Java. Básicamente se sitúa en un nivel superior al hardware del sistema sobre el que se pretende ejecutar la aplicación, y este actúa como un puente que entiende tanto el bytecode como el sistema sobre el que se pretende ejecutar. Así, cuando se escribe una aplicación Java, se hace pensando que será ejecutada en una máquina virtual Java en concreto, siendo ésta la que en última instancia convierte de código bytecode a código nativo del dispositivo final.

La gran ventaja de la máquina virtual java es aportar portabilidad al lenguaje, de manera que desde Sun Microsystems se han creado diferentes máquinas virtuales java para diferentes arquitecturas, y, así, un programa .class escrito en Windows puede ser interpretado en un entorno Linux. Tan solo es necesario disponer de dicha máquina virtual para dichos entornos. De ahí el famoso axioma que sigue a Java: "escríbelo una vez, ejecútalo en cualquier parte", o "Write once, run anywhere".

JRE

El JRE (Java Runtime Environment, o Entorno en Tiempo de Ejecución de Java) es el software necesario para ejecutar cualquier aplicación desarrollada para la plataforma Java. El usuario final usa el JRE como parte de paquetes software o plugins (o conectores) en un navegador Web. Sun ofrece también el SDK de Java 2, o JDK (Java Development Kit) en cuyo seno reside el JRE, e incluye herramientas como el

compilador de Java, Javadoc para generar documentación o el depurador. Puede también obtenerse como un paquete independiente, y puede considerarse como el entorno necesario para ejecutar una aplicación Java, mientras que un desarrollador debe además contar con otras facilidades que ofrece el JDK.

ENLACE DE DESCARGA

https://netbeans.org/downloads/

LENGUAJE DE PROGRAMACIÓN (*PROGRAMMING LANGUAGE*)

En computación es cualquier lenguaje artificial que puede utilizarse para definir una secuencia de instrucciones, a fin de que puedan ser procesadas por un computador.
Conjunto de caracteres, reglas, palabras y operaciones con significados previamente asignados y que permiten escribir programas.

La definición de un lenguaje de programación cubre tres aspectos:

1. Léxico: definen los símbolos que sirven para la redacción de un programa y las reglas para la formación de palabras en el lenguaje. Por ejemplo, 10 es un número entero.
2. Sintaxis: conjunto de reglas que permiten organizar las palabras del lenguaje en frases, por ejemplo, la operación de división se define como Dividendo/Divisor.
3. Semántica: definen las reglas que dan sentido a una frase.

Los principales tipos de lenguajes de programación utilizados en la actualidad son:

1. Lenguajes de máquina.
2. Lenguajes de bajo nivel y traductores (lenguaje ensamblador, compiladores, intérpretes).
3. Lenguajes de alto nivel (C++, C#, Visual Basic, Java, Turbo Pascal, Prolog, SQL, HTML, JavaScript, VBScript, PHP, VB.Net, Fortran, Delphi, etc.).

Objetivos:

Conocer sobre conceptos asociados a tipos de datos elementales, su clasificación, operadores y su precedencia, la conversión implícita y explícita de dato.

Puntos:

1. Conceptos de dato, tipo de dato, variables, constantes, expresiones, precedencia de operadores.
2. Clasificaciones de los tipos de datos.
3. Operaciones de tipos de dato elementales (entero, real, carácter, string, booleano).
4. Precedencia de operadores. Conversión de tipos, implícita y explícita.

CONCEPTOS BÁSICOS

1. **Dato:** diferentes entidades u objetos de información con que trabaja un programa. Determina el conjunto de valores que la entidad puede almacenar, los operadores que puede usar y las operaciones definidos sobre ellos.

2. **Tipo de Dato:** define el conjunto de valores que un elemento o un objeto (una variable, constante, expresión o función) de dicho tipo puede asumir y las operaciones asociadas a tales valores. Es un conjunto de entidades o de objetos y las operaciones definidas sobre ellos.

3. **Variable:** nombre asignado a una entidad que puede adquirir **un valor cualquiera** dentro de un conjunto de valores. Es decir, una entidad cuyo valor puede cambiar a lo largo del programa. En un programa de computador se puede asumir que una variable es una posición de memoria donde los valores asignados pueden **ser reemplazados o cambiados por otros valores** durante la ejecución del programa.

4. **Constante:** nombre asignado a una entidad al cual se asigna un valor que mantiene sin cambios durante el programa. En java se denotan las constantes con la palabra reservada final.
   Ejemplo:

   ```
   final int pi=3.1416;
   ```

   Aquí estamos declarando una variable de tipo entero llamada ‘’pi’’ cuyo valor será 3.1416 durante toda la ejecución del algoritmo.

5. **Expresiones:** Son combinaciones de constantes, variables, símbolos de operación, paréntesis y nombres de funciones o acciones. Cada expresión toma un valor que se determina evaluando los valores de sus variables, constantes y operadores. Una expresión consta de *operandos* y *operadores*. Las expresiones se pueden clasificar en Aritméticas, Lógicas o Carácter. Así tenemos:
   a. Una expresión aritmética, arroja resultados de tipo numérico (*entero* o *real*).

   b. Una expresión relacional o una expresión lógica, arrojan resultados de tipo lógico (*booleanos*).
   c. Una expresión carácter, arroja resultados de tipo carácter (caracteres simples o *string*).
6. **Precedencia de operadores (prioridad definida entre los operadores):** Indica en qué orden debe aplicarse diferentes operaciones sobre un conjunto de valores. Permiten aplicar los operadores en el orden correcto.

CLASIFICACIONES PARA LOS TIPOS DE DATOS

1. **Tipos de datos primitivos o elementales**: tipos básicos incluidos en cada lenguaje de programación. En el lenguaje de programación Java, también son llamados tipos integrados. Y los más usados son: entero, real, lógico, carácter y cadena de caracteres. En java se describen de la siguiente manera:
   a. int: Solo almacena números enteros. Ejemplo: 1, 59, -78.
   b. float: Almacena números enteros y reales. Ejemplo: 1, 59, -78, 5.69, -365.489.
   c. bool: Almacena solo true o false (Verdadero/falso).
   d. char: Almacena cualquier alfa numérico (comillas simples). Ejemplo: ‘a’, ‘A’, ‘,’, ‘@’.
   e. string: Almacena una cadena de caracteres (comillas dobles). Ejemplo: ‘’ hola”, “a”, “¿cómo estás?”.

2. **Tipos de datos estructurados**: tipos basados o construidos a partir de tipos de datos primitivos por ejemplo, arreglo, registro, archivo.
3. **Tipos de datos abstractos (TDA):** tipos de datos definidos por el usuario y las operaciones abstractas aplicadas sobre ellos. Los TDA apoyan el concepto de ocultamiento de la información. Esconden los detalles de la representación y permiten el acceso a los objetos sólo a través de sus operaciones, son ejemplos las representaciones de los TDA Lista, Cola, Pila, Árbol y la representación que hace el Enfoque Orientado a Objeto mediante atributos y métodos.

OPERACIONES DE LOS TIPOS DE DATOS ELEMENTALES

1. **Operación**: Acción por medio de la cual se obtiene un resultado de un operando. Ejemplos: sumar, dividir, unir, restar.
2. **Operando**: número, texto, valor lógico, variable o constante sobre la cual es ejecutada una operación.
3. **Operador**: símbolo que indica la operación que se ha de efectuar con el operando, por ejemplo, + / - * > == ≠ ≥ =.

Algunos comentarios sobre tipos de datos y precedencia

Los lenguajes C, C++ y Java requieren que todas las variables tengan un tipo asociado antes de que puedan ser usadas en un programa. Los lenguajes con esta propiedad son llamados **lenguajes de tipos estrictos**.

A diferencia de C y C++, los tipos primitivos de Java son portátiles entre todas las plataformas que reconocen a Java. Esto permite a los programadores Java escribir los programas una sola vez sin saber ni preocuparse en cuál plataforma de computadora se ejecutará el programa. Por el contrario, los programadores en C y C++ a menudo tenían que escribir varias versiones de los programas para trabajar con las diversas plataformas de computadoras por que no se garantizaba que los tipos de datos primitivos fueran idénticos o se representaran igual en todas las computadoras.
Como los diseñadores de Java querían que fuera un lenguaje portátil o transportable, decidieron usar estándares internacionales reconocidos para los formatos de caracteres (*char*, estándar Unicode) y para los números de punto flotante (*flot,* IEEE 754).

Los operadores de expresiones contenidas dentro de pares de paréntesis ( ) se evalúan primero. Así el programador puede usar paréntesis para obligar a una evaluación en el orden que desee.

Si una expresión contiene varias operaciones de multiplicación, división o residuo, o varias operaciones de suma o resta, los operadores se aplican de izquierda a derecha (asociatividad). Entre ellos tienen el mismo nivel de precedencia.

Operadores	Nombres	Orden en que se evalúan
		Más alta prioridad. Las expresiones entre

( )	Paréntesis o Corchetes	paréntesis se evalúan primero. Si los paréntesis están anidados, la expresión más interna se evalúa primero. Si hay varios pares de paréntesis en el mismo nivel, se evalúan de izquierda a derecha.
^	Exponente (equiv. a **)	**Operadores matemáticos** Entre ellos el mismo nivel de precedencia. Si hay varios se evalúan de izquierda a derecha.
– (unario)	Menos unario	
* / div mod	Multiplicación real División real División entera o cociente (div) Residuo o resto entero (mod)	
+ –	Suma, Resta	
	DesplazamientoIzq, DesplazamientoDer	**Operadores de desplazamiento de Registro** Desplazamiento a la izquierda o a la derecha en un archivo o en una cadena de caracteres
< ≤ > ≥	Menor, Menor o igual,	**Operadores relacionales.** Si hay varios se evalúan de

	Mayor, Mayor o igual	izquierda a derecha. Entre ellos el mismo nivel de precedencia.
== ≠	Igual, Diferente o distinto de	
No	No lógico (*not*)	**Operadores lógicos**
Y	Y lógico (*and*)	
O	O lógico (or)	
+	Concatenación	**Operador de cadena** Permite la concatenación de valores de tipo entero, real o lógico con valores de tipo caracteres o con cadenas de caracteres (string). El resultado de la concatenación es un string. Se utiliza principalmente para crear mensajes.
=	Asignación	**Menor prioridad.** La asignación de valores o resultados a una variable o constante es la última operación que se realiza.

CONVERSIÓN DE TIPOS

La conversión de tipos es el proceso de cambiar un valor de un tipo de dato a otro. Por ejemplo, el string o cadena "1579874" se puede convertir a un número entero, o se puede cambiar un número real a un string o a un entero.

Las conversiones de tipo pueden ser de **ampliación** o de **restricción**:

- Las conversiones de **ampliación** transforman un valor de un tipo de dato a otro más grande, por ejemplo transforman un valor entero (conjunto más pequeño) a un número real (que es un conjunto más grande). Estas conversiones no producen desbordamiento o pérdida de datos.

- Las conversiones de **restricción** permiten transformar un valor de un conjunto de datos más grande a uno más pequeño, por ejemplo, transformar un número real a un entero o transformar una cadena a carácter. Si transformamos el valor X=23,14587 a entero obtendríamos como resultado 23, lo cual significa pérdida de información ya que se pierde la precisión de los decimales; transformar el string m=”casa” a carácter significaría quedarnos solo con el carácter ‘c ’ del inicio de la cadena. Estos tipos de transformación son poco convenientes ya que pueden implicar pérdida de información y solo deben ser usados cuando sea estrictamente necesario.

Las conversiones por ampliación o por restricción pueden a su vez ser **explícitas** o **implícitas**:

- **Conversión implícita.** La mayoría de las conversiones, como la asignación de un valor a una variable, se producen automáticamente. El tipo de datos de la variable determina el tipo de datos de destino de la conversión de expresión. En otros casos, la conversión implícita viene dada tanto por los tipos de datos como por los operadores utilizados.

- **Conversión explícita.** Para convertir explícitamente una expresión a un tipo de datos concreto, se utilizan funciones que se asumen predefinidas y disponibles en java como mostraremos a continuación. Las conversiones explícitas requieren más escritura que las implícitas, pero proporcionan más seguridad con respecto

a los resultados. Además, las conversiones explícitas pueden controlar conversiones con pérdida de información.

Las conversiones con pérdida de información tienen lugar cuando el tipo de datos original no tiene un análogo en el tipo de destino de la conversión. Por ejemplo, la cadena "Pedro" no se puede convertir en un número. En estos casos, algunos lenguajes de programación devuelven un valor predeterminado cuando se usa la función de conversión de tipo, por ejemplo el valor **NaN** o el número cero, indicando con estos valores que la conversión de tipos falló.

Algunos tipos de conversiones, como de una cadena a un número, tardan bastante tiempo. Cuantas menos conversiones utilice el programa, más eficaz será.

Veamos la conversión de una cadena a un entero y flotante:

```
String Letras=”10”;
Float.parseFloat(Letras);
Integer.parseInt(Letras);
```

Ahora veamos de un tipo entero a una cadena y real:

```
int Numero=7;
String auxNumero;
auxNumero=String.valueOf(Numero);
Float.parseFloat(auxNumero);
```

ESTRUCTURAS DE CONTROL CONDICIONAL (SI Y SELECCIÓN)

Objetivos:

Conocer la sintaxis, semántica y uso de las principales estructuras de control básicas:

- Condicional simple, Condicional doble, Condicional anidado.
- Selección múltiple.

Puntos:

Las estructuras de control o formas de composición de acciones, son los mecanismos mediante los cuales es posible construir nuevas acciones a partir de otras. En esta sección estudiaremos las siguientes:

- Condicional simple.
- Condicional compuesto.
- Condicional anidado.
- Selección múltiple.

CONDICIONAL SIMPLE: Si – Entonces – FSi (*if*)

El condicional simple **Si – Entonces – Fsi** ejecuta un conjunto de instrucciones si cumple la condición evaluada en el Si.

Sintaxis y comportamiento:

En la instrucción

Si <condición> **Entonces**
<instrucciones S1>;
FSi;

se evalúa la <condición> y:

- Si la condición es verdadera, ejecutan o realizan las instrucciones del bloque S1.
- Si la condición es falsa, no se realiza ninguna instrucción del Si.

En java seria de la siguiente manera:

```
if(condición)
{
<instrucciones S1>;
}
```

CONDICIONAL COMPUESTO: Si – Entonces – Sino – Fsi (*if* – *else*)

Permite elegir entre dos opciones o alternativas posibles, en función de que se cumpla o no la condición expresada en el Si.

Sintaxis y Comportamiento:

En la instrucción

Si <condición> **Entonces**
<instrucciones S1>;
Sino
<instrucciones S2>;
FSi;

se evalúa la <condición> y:

- Si la condición es verdadera, ejecutan o realizan las instrucciones del bloque S1.
- Si la condición es falsa, se ejecutan o realizan las instrucciones del bloque S2

En Java seria de la siguiente manera:

```
if(condición)
{
        <instrucciones S1>;
}else{
        <instrucciones S2>;
}
```

CONDICIONAL ANIDADO

Permite incluir dentro del cuerpo de la instrucción Si a otras instrucciones Si simples o compuestas. Se utiliza para elegir entre varias opciones o alternativas en función del cumplimiento o no de las diferentes condiciones que se van verificando en cada instrucción Si.

Comportamiento: en cada condicional se cumple el mismo comportamiento que se ha indicado para el condicional simple o anidado.
Ejemplo:

```
public class EjemplosIFAnidados
{
        public static void main(String[]args)
        {
                int edad = 5;
                if(edad >= 18)
                {
                        System.out.println("Tienes 18 o mas");
                }
                else if(edad >= 15)
                {
                        System.out.println("Tienes 15 años o más pero menos de 18")
```

```
            }else if(edad >= 10)
            {
                System.out.println("Tienes 10 años o más pero menos de 15")
            }else
                System.out.println("Eres un niño");
            }
        }
}
```

SELECCIÓN MÚLTIPLE: Selección – Sino – Fselección (*switch*)

La selección múltiple permite evaluar una condición o expresión que puede tomar muchos valores distintos. Se ejecutaran las instrucciones correspondientes al caso que se cumple. La principal ventaja de la estructura Selección es que permite crear algoritmos que sean legibles y evitar la confusión creada por el anidamiento de muchos bloques SI.

Comentario: se evalúa cada condición de la selección y se realiza el bloque de instrucciones correspondientes a la condición que se cumple.

En java sería de la siguiente manera:

```
switch(numero) {
 case 1:
    <instrucciones S1>;
    break;
 case 2:
    <instrucciones S2>;
    break;
 case 3:
    <instrucciones S3>;
    break;
 case 4:
    <instrucciones S4>;
    break;
```

```
        case 5:
           <instrucciones S5>;
           break;
        default:
           <instrucciones S6>;
           break;
        }
```

Cuando la variable numero toma el valor entero 1 se cumple el bloque <instrucciones S1> correspondiente al case 1, después de ejecutarse la última instrucción del bloque S1 se ejecuta la instrucción break; y luego continua la ejecución del programa a partir de la llave.

En el caso de que la variable número no esté dentro del rango de valores entre 1-5 Entrará en el **default** y se controlará con el bloque de instrucciones S6 ese error

Ejemplo:

```
public class Meses {
   public static void main (Spring[] args) {
          int (month) = 8;
          switch (month) {
                    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("Mayo"); 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;
          }
    }
```

```
}
```

Objetivos:

Conocer la sintaxis, comportamiento y utilidad de las principales estructuras iterativas utilizadas en programación.

Puntos:

- Bucles o ciclos:
  - Para (for).
  - Repetir (do - while).
  - Mientras (while).

BUCLES O CICLOS

Un **bucle, lazo, ciclo o loop** (en inglés) es un segmento de algoritmo o programa (una secuencia de instrucciones) que se repiten un determinado número de veces mientras se cumple una determinada condición, en otras palabras, un bucle o ciclo es un conjunto de instrucciones que se repiten mientras una condición es verdadera o existe.

A cada repetición del conjunto de acciones se denomina **iteración**.

Para que un bucle no se repita indefinidamente debe tener una condición de **parada** o de **fin**. Esta condición de parada o de fin se verifica cada vez que se hace una iteración. El ciclo o loop llega a su fin cuando la condición de parada se hace verdadera.

La condición de parada puede estar al principio de la estructura repetitiva o al final.
Al igual que las estructuras de selección simple o compuesta (los bloques si – entonces – sino – fsi), en un algoritmo pueden utilizarse varios ciclos. Estos ciclos pueden ser independientes (una a continuación de otro) o anidados (ciclos dentro de ciclos).

Para representar los bucles o lazos utilizaremos en el curso las estructuras de control: Para, Repetir y Mientras.

1. **Estructura iterativa Para... FinPara *(for... next... end for*):**

   Es una estructura iterativa que es controlada por una variable (llamada también **variable índice**), la cual se incrementa o decrementa hasta llegar a un valor límite o valor final que representa la condición de parada.

   La estructura **Para** comienzan con un valor inicial de la variable índice, las acciones especificadas para el ciclo se ejecutan un número determinado de veces, a menos, que el valor inicial de la variable índice sea mayor que el valor límite que se quiere alcanzar.

   Se recomienda usarlo: la estructura **Para** es recomendada cuando se conoce el número de veces que se deben ejecutar las instrucciones del ciclo, es decir, en los casos en que el número de iteraciones es fijo y conocido.

   Sintaxis del Para:

   Para variable_índice = valor1 **hasta** valor2 **en** inc/dec **hacer**
   <inst 1>
   …
   <inst n>
   FinPara;

   for (int i=0; i<=10; i++)
   {
   <inst 1>
   …
   <inst n>

   }

   Ejemplo 1:

   /*

```
* programa que muestra los números del 1 al 10
*/
public class Ejemplo0For {
    public static void main(String[] args) {
        int i;
        for(i=1; i<=10;i++)
            System.out.println(i + " ");
    }
}
```

Ejemplo 2**:**

```
/*
 * programa que muestra el valor de a, b y su suma mientras que la suma de
 * ambas es menor de 10. En cada iteración el valor de a se incrementa en
 * 1 unidad y el de b en 2
 */
public class Ejemplo3For {
    public static void main(String[] args) {
        int a, b;
        for(a = 1, b = 1; a + b < 10; a++, b+=2){
            System.out.println("a = " + a + "  b = " + b + "  a + b = " + (a+b));
        }
    }
}
```

2. **Estructura iterativa Mientras – Hacer – FinMientras (*while*):**

   Es una estructura iterativa que permite verificar la condición de **entrada** al ciclo **antes** del cuerpo de instrucciones a repetir.

   Como la evaluación de la condición de entrada se realiza al **inicio** del bloque **Mientras**, puede ocurrir que las instrucciones del ciclo no se realicen ni siquiera 1 vez, a diferencia del **Repetir**, donde el bloque de instrucciones se realiza al

menos 1 vez porque la condición de parada se verifica al final. Las instrucciones del Mientras se pueden realizar 0 o más veces antes de que se cumpla la condición de terminar el ciclo.

El conjunto de instrucciones dentro del **Mientras – FinMientras** se ejecuta cuando la condición de entrada del principio se cumple (es verdadera). Dicho de otro modo, el ciclo de instrucciones dentro del **Mientras** se va a detener cuando la condición se haga falsa.

Sintaxis del Mientras:

Mientras <expresión_condición_de_entrada> **hacer**
<instrucción 1>;
....
< instrucción n>;
<actualización variable(s) usada(s) en la cond. entrada>;
FinMientras;

En java se implementaría de la siguiente forma:

```
while(condición)
{<instrucción 1>;
....
< instrucción n>;
};
```

Se recomienda usarlo: la estructura **Mientras** es recomendada cuando tienes que verificar la condición de entrada al inicio y si se cumple, entonces, entrar al ciclo y realizar sus instrucciones

Ejemplo 3:

```
/*
 * Programa que lee números hasta que se lee un negativo y muestra la
 * suma de los números leídos
 */
```

```
import java.util.*;
public class Ejemplo1While {
    public static void main(String[] args) {
        int suma = 0, num;
        Scanner sc = new Scanner(System.in);
        System.out.print("Introduzca un número: ");
        num = sc.nextInt();
        while (num >= 0){
            suma = suma + num;
            System.out.print("Introduzca un número: ");
            num = sc.nextInt();
        }
        System.out.println("La suma es: " + suma );
    }
}
```

3. **Estructura iterativa Repetir… Hasta (*do-while*):** Ejecuta un bloque de instrucciones varias veces hasta que se cumple la condición que es verificada al final del bucle.

   Las instrucciones dentro del ciclo **Repetir** se van a realizar mientras la condición de parada evaluada al final sea falsa. Dicho de otro modo, el ciclo se va a detener cuando la condición de parada se haga verdadera.

   Sintaxis del Repetir:

   Repetir
   < instrucción 1>;
   ...
   < instrucción n>;
   <actualización variable(s) usada(s) en la cond. parada>;
   Hasta <expresión_condición_de_parada>;

   En java se implementaría de la siguiente forma:

```
do
{
}while(condición):
```

   Se recomienda usarlo: la estructura **Repetir** es recomendada cuando las instrucciones del ciclo se pueden realizar al menos 1 vez antes de comprobar la condición de parada.

```
/*
 * Programa que lee un número entre 1 y 10 ambos incluidos
 */
import java.util.*;
public class Ejemplo2DoWhile {
    public static void main(String[] args) {
        int n;
```

```
        Scanner sc = new Scanner( System.in );
        do {
                System.out.print("Escribe un número entre 1 y 10:");
          n = sc.nextInt();
        }while (n<1 || n >10);
        System.out.println("Ha introducido: " + n);
    }
}
```

Objetivos:

- Conocer la sintaxis utilizada para la representación de acciones, funciones y parámetros.
- Conocer los tipos de pase de parámetro y sus implicaciones.

Puntos:

- Procedimientos
- Acciones
- Funciones
- Tipos y pase de Parámetros

PROCEDIMIENTOS

La definición de procedimientos permite asociar un nombre a un bloque de instrucciones. Luego podemos usar ese nombre para indicar en algún punto de un algoritmo que vamos a utilizar ese bloque de instrucciones, pero sin tener la necesidad de repetirlas, sólo **invocando** al procedimiento por su nombre.

Los procedimientos pueden ser clasificados en acciones o funciones. Las **acciones** se caracterizan por no retornar valores al algoritmo que las llama, mientras que las **funciones** retornan un valor. Sin embargo, aunque las acciones no retornan valores, si pueden informar al algoritmo que las llamó (a veces llamado **algoritmo principal**) de cambios realizados por sus instrucciones en algunos valores a través de una herramienta que se llama **pase de parámetros**. Los **parámetros** permiten utilizar la misma secuencia de instrucciones con diferentes datos de entrada. Utilizar parámetros es opcional.

Cuando entre las instrucciones de un algoritmo vemos el nombre de un procedimiento (acción o función), decimos que estamos llamando o invocando al procedimiento.

Los procedimientos facilitan la programación modular, es decir, tener bloques de instrucciones que escribimos una vez pero que podemos llamar y utilizar muchas veces en varios algoritmos. Una vez terminada la ejecución de un procedimiento (acción o función), se retorna el control al punto de algoritmo donde se hizo la llamada, para continuar sus instrucciones.

ACCIONES

Conjunto de instrucciones con un nombre que pueden ser llamadas a ejecución cuando sea necesaria. No retornan valores.

Sintaxis de la definición formal de la Acción

Acción <Nombre> [(lista parámetros formales)]
// comentario sobre lo que la acción hace
< definición de variables locales >
< instrucción 1 de la acción >
...
< instrucción n de la acción >
FAcción <Nombre>;

donde:

- Acción… FAcción delimitan el inicio y fin de la acción.
- <Nombre> es el identificar de la acción.
- [<lista de parámetros formales>] son objetos o variables que utilizan las instrucciones dentro de la acción, pero que no tienen valor hasta que la acción no es llamada y utilizada. Esta lista es opcional por eso aparece entre corchetes.
- <definición de las variables locales> es el conjunto de valores que se usan dentro de la acción.
- <instrucciones> a veces también llamadas simplemente acciones, es la secuencia de instrucciones a ser ejecutadas por la acción.

En java se implementaría de la siguiente manera:

Public void <nombre>(<lista de parámetros>)
{
// comentario sobre lo que la acción hace
< definición de variables locales >
< instrucción 1 de la acción >
...

```
< instrucción n de la acción >
}
```

Ejemplo:

El método cajaTexto recibe un String y lo muestra rodeado con un borde.

```
import java.util.*;
public class MetodoVoid {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      String cadena;
      System.out.print("Introduce cadena de texto: ");
      cadena = sc.nextLine();
      cajaTexto(cadena); //llamada al método
   }
  /**
   * método que muestra un String rodeado por un borde
   */
   public static void cajaTexto(String str){
      int n = str.length();
      for (int i = 0; i < n + 4; i++){
         System.out.print("#");
         }
         System.out.println();
         System.out.println("# " + str + " #");
         for (int i = 0; i < n + 4; i++){
             System.out.print("#");
         }
         System.out.println();
   }
}
```

FUNCIONES

Al igual que las acciones con conjuntos de instrucciones con un nombre, pero se caracterizan por **retornar** (enviar o devolver) **un valor** al algoritmo que la llama.

Como el resultado de la función es retornado al algoritmo principal, debe usarse una variable para almacenar este resultado, es decir, en una variable del algoritmo principal se "captura" el valor retornado por la función. Luego el valor almacenado en la variable puede ser utilizado por el algoritmo que llama a la función.

Sintaxis de la definición formal de la Función

Función <Nombre> [(lista parámetros formales)] **:** <tipo dato retorno>
// comentario sobre lo que la función hace
< definición de variables locales >;
< instrucciones de la función >;
retornar (<variable, constante o expresión compatible
con el tipo de retorno >);
FFunción <Nombre>;

donde:

- Función … FFunción delimitan el inicio y fin de la función.
- <Nombre> es un identificador (nombre) válido
- **:** operador usado para expresar que la función retornará un valor del tipo de dato que se indica luego.
- [<lista de parámetros formales>] son objetos o variables que utilizan las instrucciones dentro de la acción, pero que no tienen valor hasta que la acción no es llamada y utilizada. Esta lista es opcional por eso aparece entre corchetes.
- <tipo de dato de retorno> indica el tipo de dato del valor que la función retornará al algoritmo que la llamó (por ejemplo, podría retornar un entero, ó un string, o un objeto, etc.) retornar instrucción predefinida utilizada para indicar el lugar y el valor a retornar por parte de la función.

En java se implementaría de la siguiente manera:

public int <nombre>(<lista de parámetros>)

```
{
// comentario sobre lo que la función hace
< definición de variables locales >;
< instrucciones de la función >;
return (<valor de retorno>)
}
```

Comentario: el valor de retorno obligatoriamente debe coincidir con el tipo de dato de retorno de la función.

Ejemplo:

Función que sume dos números:

```
import java.util.*;
public class Metodos1 {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int numero1, numero2, resultado;
      System.out.print("Introduce primer número: ");
      numero1 = sc.nextInt();
      System.out.print("Introduce segundo número: ");
      numero2 = sc.nextInt();
      resultado = sumar(numero1, numero2);
      System.out.println("Suma: " + resultado);
   }
   public static int sumar(int a int b){
        int c;
          c = a + b;
          return c;
   }
}
```

LLAMADA (O INVOCACIÓN) DE LAS ACCIONES O FUNCIONES

La llamada a una acción o función es una instrucción que permite la ejecución de la secuencia de sus instrucciones. Consiste en indicar el nombre y los parámetros actuales que van a ser utilizados.

Los parámetros actuales que se indican en la llamada de la acción (por ejemplo en el algoritmo principal), deben corresponderse con los parámetros formales de la definición de la acción o /función. Por ello, la cantidad de parámetros actuales debe ser igual a la cantidad de parámetros formales y del mismo tipo de dato.

En el caso de java:

Una función o acción se invoca de la siguiente manera:
<Nombre función/acción > (<pase de parámetros>);

TIPOS Y PASES DE PARÁMETROS

Los tipos de parámetros son:

1. **Parámetros actuales:** Son los valores indicados en la llamada a la acción o función en el algoritmo principal. Son los valores que se desean pasar desde el algoritmo principal a las instrucciones de la acción o función.

2. **Parámetros formales:** Son los nombres dados a los parámetros en la definición formal de la acción o función. Con estos nombres se conocerán a los valores de los parámetros dentro de la acción o función y funcionan como variables locales dentro de ellos.

Los pases de parámetros se pueden clasificar de la siguiente forma:

1. **Pase de parámetros por valor**: El parámetro actual no es modificado si se modifica el parámetro formal dentro del procedimiento, ya que ambos parámetros ocupan posiciones diferentes en memoria. Esto se debe a que el parámetro actual se evalúa y el resultado se copia en el correspondiente parámetro formal, que ocupa otra posición en memoria. Por ello, cuando se

regresa de la acción o función al algoritmo desde donde se hace la llamada los parámetros actuales mantienen su valor original.

Ejemplo:

Suma_dos_numeros(2,3);

O también

Suma_dos_numeros(a, b);

2. **Pase de parámetros por referencia:** El parámetro actual sufre los mismos cambios que el parámetro formal. El parámetro actual no puede ser ni una constante ni una expresión. Ambos parámetros ocupan la misma posición en memoria. Por ello, cuando se regresa de la acción/función al algoritmo desde donde se hace la llamada los parámetros actuales han cambiado. En java, para pasar un valor por referencia se puede hacer por medio de un puntero o el símbolo &.

TIPOS DE DATOS ESTRUCTURADOS

1. **Estructura de Datos (ED):** Es una herramienta mediante la cual es posible almacenar datos estructurados en la memoria del computador, permitiendo guardar datos conformados por varios elementos y manipularlos en forma sencilla. Estas estructuras de datos están formadas por más de un elemento, donde estos pueden ser todos del mismo tipo de dato (ED homogéneas como los arreglos y los archivos) o de tipos de datos diferentes (ED heterogénea, como los registros y los objetos).

   Las estructuras de datos permiten el almacenamiento de información de manera organizada en la memoria principal del computador o en algún dispositivo externo (memoria auxiliar).

2. **Arreglos:** Estructuras de datos conformada por una sucesión de celdas, que permite almacenar en la memoria principal del computador un conjunto finito de elementos (hay un número máximo conocido) que tienen el mismo tipo de dato (son homogéneos).

Para hacer referencia a cualquiera de las celdas del arreglo es necesario el nombre del arreglo y el valor de uno de los elementos perteneciente al conjunto de **índices** asignado, lo que permite tener acceso aleatorio.

a. Características básicas de los Arreglos

i. **Homogéneo**: los elementos del arreglo son todos del **mismo tipo de dato.**
ii. **Ordenado**: cada elemento del arreglo puede ser identificado por el índice que le corresponde. El índice no es más que un entero que pertenece a un intervalo finito y determina la posición de cada elemento dentro del arreglo.
iii. **Acceso Secuencial o Directo**: El acceso a cada elemento del arreglo se realiza recorriendo los anteriores según el orden en que están almacenados o de manera directa (operación selectora), indicando el valor del índice del elemento requerido.
iv. **Sus elementos son tratados como variables simples:** Una vez seleccionado algún elemento del arreglo este puede ser utilizado en acciones de asignación, como parte de expresiones o como parámetros al llamar a acciones o funciones como si se tratara de una variable del tipo de dato simple declarado para todos los elementos del arreglo.
v. **Uso de índice:** El índice es un valor de tipo entero (número entero o carácter con un código entero equivalente) que determina la posición de cada elemento en el arreglo. La cantidad de índices determina las dimensiones del arreglo. Así un arreglo unidimensional tiene un índice, un arreglo bidimensional tiene dos índices y uno n-dimensional tiene *n* índices.

b. Declaración de los arreglos: Para declarar un arreglo se debe indicar:

i. El nombre del arreglo.
ii. El tipo base: tipo de dato de todos los componentes del arreglo
iii. El tipo índice: intervalo de valores que podrá tomar el índice del arreglo, indicando el límite inferior (Li) y el límite superior (Ls) del rango

<tipo de dato del arreglo><nombre>[ ];

Ejemplo:

```
int arreglo[];
arreglo[]=new int [tamaño];
```

c. **Arreglos Unidimensionales (o Vectores):** Tipos de Datos Estructurados (TDE) donde todos sus elementos pertenecen al mismo tipo y existe una correspondencia uno a uno de cada elemento con un subconjunto de los enteros (índices).

i. **Operaciones básicas en Arreglos Unidimensionales**

1. **Operación constructora**: Permite asociarle al nombre de un arreglo un valor estructurado de las mismas dimensiones del arreglo, y con componentes del mismo tipo. Esto sólo se puede hacer en la declaración.
2. **Operación selectora**: permite hacer referencia a un elemento individual del arreglo, mediante el índice unívoco del elemento. El índice es evaluado, por lo que puede ser una constante, variable o expresión.
3. **Recorrido secuencial:** Esta operación se realiza cuando se utiliza una estructura de control iterativa para tratar todos y cada uno de los elementos del arreglo de acuerdo al orden en que se almacenan. El tratamiento es el mismo para todo el arreglo, y se aplica tanto cuando se desea leer el arreglo, buscar un elemento en un arreglo, listar todo el contenido del mismo, y muchos otros.

   El recorrido puede ser de manera ascendente (el más utilizado), partiendo del límite inferior hasta el superior e incrementando en uno; o de manera descendente, partiendo del límite superior hasta el inferior y decrementando en uno.

   La estructura de control **Para** es la más conveniente para realizar de manera el recorrido, indicando solo el límite inferior y superior, pues el incremento en uno del índice es

automático. Sin embargo, si existiera alguna condición adicional o si se necesita usar una condición lógica para finalizar el recorrido, no basta con llegar al límite superior del índice, hay que cambiar de la estructura iterativa Para, y utilizar un **Mientras** o un **Repetir.**

d. Arreglos Bidimensionales (Matrices o Tablas): Un arreglo bidimensional (tabla o matriz) es un arreglo que tiene dos índices. Para localizar o almacenar un valor en el arreglo se deben especificar **dos subíndices** (dos posiciones), uno para la fila y otro para la columna. Los diferentes tipos de índices no necesitan ser subrangos del mismo tipo. Los elementos se referencian con el formato:

<identificador>[3,4]

donde:

i. <identificador>: es el nombre del arreglo o de la variable
ii. [3,4]: indica el elemento de la fila 3 y columna 4

e. Declaración de Arreglos Bidimensionales: Al igual que en los arreglos unidimensionales, los arreglos bidimensionales **también se pueden crear con declaraciones de variable o de tipo**, el cambio, es que se debe indicar dos intervalos de índices: uno para las filas y otro para las columnas. Estos intervalos o subrangos no tienen que ser de la misma dimensión.

<tipo de dato del arreglo> <nombre>[ ][ ];

Ejemplo:

```
int arreglo[][];
arreglo[][]=new int [fila][columna];
```

f. Operaciones básicas en arreglos bidimensionales: Las operaciones son las mismas que para arreglos unidimensionales, pero utilizando dos subíndices para identificar a cada elemento. Estas son:

i. Operación constructora.

ii. Operación selectora.
iii. Recorrido Secuencial

Cualquier proceso que queramos realizar sobre un arreglo bidimensional involucrará a los dos subíndices. Se usan dos ciclos anidados para recorrer todo el arreglo.

g. Algoritmos de búsqueda en Arreglos: La búsqueda de un elemento, es una operación muy frecuente en cuanto a programación se trata, ya que permite verificar la existencia de un elemento dentro de una estructura de datos. En el caso de los arreglos unidimensionales, una operación de búsqueda puede implicar recorrer, de manera secuencial, desde 1 hasta *N* elementos. El resultado de un algoritmo de búsqueda puede ser:

i. Un lógico indicando que existe (verdad) o no (falso) el elemento buscado.
ii. Un número entero que indique la posición o índice donde está el elemento en el arreglo. Si no existe devuelve -1, 0 o cualquier valor fuera del intervalo definido para el arreglo.
iii. Un número entero, que indique la cantidad de veces que se encuentra el elemento buscado (0 o más repeticiones), devuelve cero (0) si el elemento no existe.

h. Búsqueda lineal: Es la forma más simple de buscar un elemento y consiste en examinar secuencialmente uno a uno hasta encontrar el buscado o haber revisado todos los elementos sin éxito.

Ejemplo:

```
public static int busquedaLineal(int X, int [] A)
{
        int i = 0;
        int n = A.length -1;
        while (i < n && A[i] != X)
```

```
            i++;
            /* Si se encuentra el elemento se devuelve su posición sino
            se devuelve -1 (indica que el elemento no está)*/
        if (A[i] == X)
        return i;
        else
        return -1;
}
```

i. **Algoritmo de ordenamiento:** El ordenamiento es una de las tareas que se efectúa con más frecuencia cuando se desarrollan programas de tratamiento de información estructurada. Sin embargo esta tarea puede llegar a consumir mucho tiempo de ejecución. Por esta razón, muchos investigadores de la computación han desarrollado cientos de algoritmos de ordenamiento que buscan hacer más eficiente el proceso.

Ejemplo:

```
public void sort(double [] vector) {
    for(int i = 0; i < vector.length-1; i++){
        int idx = i;
        for(int j= i +1; j < vector.length ; j++){
                    if(vector[j] < vector[idx]){
            idx = j;
          }
         }
        if (i != idx){
           double aux = vector[idx];
           vector[idx] = vector[i];
           vector[i] = aux;
           }
        }
    }
```

Su lógica es sencilla, compara cada elemento con sus siguientes, guardando el incide del elemento de menor valor y al final lo intercambia.

3. **Registros:** Estructura de datos formada por una colección finita de elementos llamados campos, no necesariamente homogéneos (del mismo tipo) y que permiten almacenar una serie de datos relacionados entre sí bajo un nombre y una estructura común.

   a. **Características básicas de los Registros:**
      i. Permiten almacenar un grupo de elementos bajo un nombre y una estructura común.
      ii. Los elementos (campos) de un registro no tienen que ser homogéneos, de hecho, generalmente son de diferentes tipos.
      iii. No están disponibles en todos los lenguajes de programación, razón por la cual muchas veces es necesario simularlo o definirlo.
      iv. Cada campo del registro se comporta como una variable simple, de manera que puede ser usado en una expresión de asignación, como parte de otra expresión, en operaciones o como parámetro al invocar una acción o función.

   b. **Operaciones básicas en Registros:**
      i. **Operación Constructora:** Permite asociarle al nombre de un registro un dato estructurado, el cual se corresponde componente a componente con la declaración del registro. Esta operación **permite inicializar** los campos del registro.
      ii. **Operación Selectora:** Permite referenciar o seleccionar un campo particular del registro.

   c. **Ejemplo de un registro en java:**

```
class Persona
{
    Int edad;
    String nombre;
    }
```

4. **Archivos:** Un archivo puede definirse como una secuencia de elementos del mismo tipo, que residen generalmente en memoria auxiliar. Los archivos son utilizados cuando el volumen de datos a procesar es grande y no cabe en la memoria central del computador. Adicional a la memoria principal, los archivos pueden ser almacenados en dispositivos de memoria auxiliar como discos, disquetes, cintas, cartuchos zip, cd, pen drive o memory flash, entre otros.
    a. **Características básicas de los Archivos:**
        i. Los archivos son utilizados cuando se desea que los datos puedan recuperarse aún después de haber apagado la máquina y también cuando se manejan grandes volúmenes de información.
        ii. Residen en memoria auxiliar, la cual generalmente, es más económica, grande, pero lenta que la memoria principal del computador.
        iii. Los elementos que almacena un archivo pueden ser elementales (texto, números) o estructurados (arreglos, registros). Para propósitos del curso, sólo se puede leer un archivo elemento por elemento.
        iv. La cantidad de componentes que constituyen el archivo, es en teoría, ilimitada (no se conoce a priori). Si tiene cero elementos se dice que es un archivo vacío.
        v. Cuando la estructura de un archivo no parece contener elementos de un solo tipo, si decimos que todos los elementos del archivo es un gran registro, no rompe la definición anterior (es un caso particular de un archivo con un solo elemento de tipo registro).
        vi. Los archivos más comunes son los archivos secuenciales. Los elementos de un archivo secuencial están dispuestos uno tras otro, y para acceder un elemento en particular, hay que pasar por todos los anteriores. Por esos se les llama de acceso secuencial, a diferencia de los arreglos que son de acceso aleatorio. Sin embargo, los elementos del archivo también puedes ser extraídos de manera directa, pero este objetivo escapa del alcance del curso.

    b. **Declaración de Archivos:** Consiste en indicar el nombre de la variable que será utilizada en el algoritmo para hacer Referencia al archivo.

i. **Declaración por Variable:** se declara la variable tipo archivo como si se tratara de una variable de tipo de dato simple.
ii. **Declaración por Tipo:** al igual que los arreglos y los registros, se antecede la palabra reservada **Tipo** a la declaración y luego se declaran las variables del tipo.

c. Operaciones básicas en Archivos Secuenciales: Estas operaciones se consideran operaciones primitivas y reservadas del pseudo-código. Entre las operaciones básicas en archivos secuenciales tenemos las siguientes:

i. **Abrir el Archivo:** Ubica y prepara el archivo para poder trabajar con él.

Acción AbrirArchivo (Ref Archivo A; String <ruta_del_archivo>, <Argumentos>)

donde:

- **<ruta_del_archivo>** indica la ruta en memoria o al menos el nombre del archivo con el cual se va a trabajar, por ejemplo, C/Mis documentos/datosprueba.txt o DatosParticipantes.doc.
- **<Argumentos>** es uno o más de las siguientes palabras reservadas:
  - Escritura: indica que el archivo se abre de solo escritura.
  - Lectura: indica que el archivo de abre de solo lectura.
  - Añadir: indica que el archivo se abre de escritura pero todo lo que se escriba se añade al final del archivo.
  - Texto: indica que el archivo a abrir es un archivo de texto.
  - Binario: indica que el archivo a abrir es un archivo binario (un archivo diferente a un archivo de texto).

Los argumentos pueden ser combinados con el operador lógico *y*. Por ejemplo: AbrirArchivo(A, “prueba.txt”, Lectura y Texto).

En el parámetro argumentos, normalmente se indica como mínimo, uno de los tipos de archivo (lectura, escritura, añadir) y uno de los tipos de datos para sus elementos (texto, binario). Los argumentos también son llamados flags.

Comentario: El archivo se abre una sola vez en el algoritmo, al principio del mismo y debe ser cerrado al finalizar el mismo.

ii. **Cerrar el archivo:** Cuando se desea dejar de trabajar con un archivo, debe cerrarse para que esté disponible para otras aplicaciones. Sólo los archivos que están abiertos pueden cerrarse. Si un archivo no se cierra se puede perder información en el mismo.

Acción CerrarArchivo(Ref Archivo A)

iii. **Fin de archivo (FDA, EOF):** indica si no hay más elementos para leer en el archivo. Sólo se usa para archivos de lectura. Retorna verdadero si se alcanzó el fin del archivo y falso en caso contrario.

Función FDA(Ref Archivo A) : lógico

iv. **Leer del archivo:** Lee un elemento. Aunque una lectura no modifica el archivo físicamente en disco, la variable Archivo si es modificada (por ejemplo, cambia la posición actual del cursor del archivo), por lo tanto debe pasarse por Referencia.

Acción LeerArchivo(Ref Archivo A, Ref <elemento_tipo> x)

v. **Escribir en el archivo:** escribe un elemento en el archivo.

Acción EscribirArchivo(Ref Archivo A, <elemento_tipo> x)

d. Recorrido secuencial para tratar un archivo: Supongamos que utilizamos dos archivos de texto: **A** es de lectura y será el archivo a recorrer a través de operaciones de lectura, **B** es de escritura y se utilizará para mostrar los resultados.

La idea fundamental es recorrer el archivo de entrada o lectura una sola vez a través del uso de una estructura de iteración que finalice cuando la **operación FDA** retorne verdadero. Los ciclos **Mientras** o **Repetir** son los más adecuadas para este recorrido, el **Para** no resulta tan conveniente, a menos que conozcamos cuantos registros tiene el archivo o el problema este limitado a leer una cantidad determinada de ellos. El archivo de salida, si se pide que se genere, se creará en el ciclo de recorrido ya que la idea es realizar la lectura del archivo origen y la creación del archivo resultado al mismo tiempo, es decir, en un solo ciclo.

La acción de impresión del archivo de salida requiere que el archivo abierto como escritura sea cerrado y reabierto como de lectura, de manera que sea recorrido y en cada paso se escriba el contenido de cada uno de sus registros.

A continuación se suministrará un esquema para este recorrido, que si bien no es 100% general o no es aplicable para todos los problemas de recorrido de archivos, puede servir como base para la resolución de los ejercicios clásicos donde se usa un archivo de entrada, se genera un archivo de salida y se realizan cálculos adicionales. En el ejemplo se contarán la cantidad de caracteres existentes en el archivo de entrada.

Una de tantas maneras de implementar archivos en java es la siguiente:

Creación de un archivo:

```
File archivo = new File ("C:\\archivo.txt");
```

Leer un de un archivo:

```
class LeeFichero {
   public static void main(String [] arg) {
      File archivo = null;
      FileReader fr = null;
      BufferedReader br = null;
```

```
      try {
        // Apertura del fichero y creacion de BufferedReader para
poder
        // hacer una lectura comoda (disponer del metodo
readLine()).
        archivo = new File ("C:\\archivo.txt");
        fr = new FileReader (archivo);
        br = new BufferedReader(fr);

        // Lectura del fichero
        String linea;
        while((linea=br.readLine())!=null)
          System.out.println(linea);
      }
      catch(Exception e){
        e.printStackTrace();
      }finally{
        // En el finally cerramos el fichero, para asegurarnos
        // que se cierra tanto si todo va bien como si salta
        // una excepcion.
        try{
          if( null != fr ){
            fr.close();
          }
        }catch (Exception e2){
          e2.printStackTrace();
        }
      }
    }
}
```

Escribir en un archivo:

```
public class EscribeFichero
```

```
        {
            public static void main(String[] args)
            {
                FileWriter fichero = null;
                PrintWriter pw = null;
                try
                {
                    fichero = new FileWriter("c:/prueba.txt");
                    pw = new PrintWriter(fichero);

                    for (int i = 0; i < 10; i++)
                        pw.println("Linea " + i);

                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                  try {
                  // Nuevamente aprovechamos el finally para
                            // asegurarnos que se cierra el fichero.
                            if (null != fichero)
                              fichero.close();
          } catch (Exception e2) {
            e2.printStackTrace();
          }
        }
    }
}
```

Si usamos sólo **FileImputStream, FileOuoutStream, FileReader o FileWriter**, cada vez que hagamos una lectura o escritura, se hará físicamente en el disco duro. Si escribimos o leemos pocos caracteres cada vez, el proceso se hace costoso y lento, con muchos accesos a disco duro.

Los **BufferedReader**, **BufferedInputStream**, **BufferedWriter** y **Buffered OutputStream** añaden un buffer intermedio. Cuando leamos o escribamos, esta clase controlará los accesos a disco:

- Si vamos escribiendo, se guardará los datos hasta que tenga bastantes datos como para hacer la escritura eficiente.
- Si queremos leer, la clase leerá muchos datos de golpe, aunque sólo nos dé los que hayamos pedido. En las siguientes lecturas nos dará lo que tiene almacenado, hasta que necesite leer otra vez.

Esta forma de trabajar hace los accesos a disco más eficientes y el programa correrá más rápido. La diferencia se notará más cuanto mayor sea el fichero que queremos leer o escribir.