ALGORITMOS Y ESTRUCTURAS DE DATOS
SEMANA Nº 2

OBJETIVOS DE LA SEGUNDA SEMANA

Clase teórica
- Lenguaje Java.
- Elementos del lenguaje, Gramatica, tipos de datos, variables, operadores, estructuras de contro, entrada y salida.

Clase práctica
- Ejercicios de clases con representación grafica (diagrama de clases y diagramas de flujo).

CLASE TEORICA

FASES EN EL DESARROLLO DE UN PROGRAMA

En esta materia aprenderá lo que es un programa, como escribirlo y que hacer para que la computadora lo ejecute y muestre los resultados perseguidos. Tambien adquirira conocimientos generales acerca de los lenguajes de programación utilizados para escribir programas. Despúes, nos centraremos en el lenguaje de programación *Java,* presentando sus antecedentes y marcando la pauta a seguir para realizar un programa sencillo.

QUE ES UN PROGRAMA

Probablemente alguna vez haya utilizado una computadora para escribir un documento o para divertirse con algun juego. Recuerde que en el caso de escribir un documento, primero tuvo que poner en marcha un procesador de textos, y que si quiso divertirse con un juego, lo primero que tuvo que hacer fue poner en marcha el juego. Tanto el procesador de textos como el juego son *programas* de computadora.

Poner un programa en marcha es sinonimo de ejecutarlo. Cuando ejecutamos un programa, nosotros solo vemos los resultados que produce (el procesador de textos muestra sobre la pantalla el texto que escribimos; el juego visualiza sobre la pantalla las imagenes que se van sucediendo) pero no vemos el guion seguido por la computadora para conseguir esos resultados. Ese guion es el programa.

Ahora, si nosotros escribimos un programa, entonces si que sabemos como trabaja y por que trabaja de esa forma. Esto es una forma muy diferente y curiosa de ver un programa de computadora, lo cual no tiene nada que ver con la experiencia adquirida en la ejecucion de distintos programas

Ahora, piense en un juego cualquiera. La pregunta es que hacemos si queremos enseñar a otra persona a jugar? Logicamente le explicamos lo que debe hacer; esto es, los pasos que tiene que seguir. Dicho de otra forma, le damos instrucciones de como debe actuar. Esto es lo que hace un programa de computadora. Un *programa* no es nada ma's que una serie de instrucciones dadas a la computadora en un lenguaje entendido por el, para decirle exactamente lo que queremos que haga. Si la computadora no entiende alguna instruccion, lo comunicará generalmente mediante mensajes visualizados en la pantalla.

LENGUAJES DE PROGRAMACON

Un programa tiene que escribirse en un lenguaje entendible por la computadora. Desde el punto de vista físico, una computadora es una maquina electrónica. Los elementos físicos (memoria, CPU, etc.) de que dispone la computadora para representar los datos son de tipo binario; esto es, cada elemento puede diferenciar dos estados (dos niveles de voltaje). Cada estado se denomina genéricamente *bit y* se simboliza por 0 o *1.* Por lo tanto, para representar y manipular información numérica, alfabetica y alfanumérica se emplean cadenas de *bits.* Segun esto, se denomina *byte* a la cantidad de informacion empleada por una computadora para representar un caracter; generalmente un *byte* es una cadena de ocho *bits.*

Así, por ejemplo, cuando un programa le dice a la computadora que visualice un mensaje sobre el monitor, o que lo imprima sobre la impresora, las instrucciones correspondientes para llevar a cabo esta acción, para que puedan ser entendibles por la computadora, tienen que estar almacenadas en la memoria como cadenas de *bits. Esto* hace pensar que escribir un programa utilizando ceros y unos (lenguaje maquina), llevaría mucho tiempo y con muchas posibilidades de cometer errores. Por este motivo, se desarrollaron los lenguajes *ensambladores.*

Un lenguaje *ensamblador* utiliza *códigos nemotécnicos* para indicarle al hardware (componentes ffsicos de la computadora) las operaciones que tiene que realizar. Un código nemotécnico es una palabra o abreviatura fácil de recordar que presenta una tarea que debe realizar el procesador de la computadora. Por ejemplo:

MOV AH, 4CH

El codigo *MOV* expresa una operacion consistente en mover alguna información desde una posición de memoria a otra.

Para traducir un programa escrito en *ensamblador* a lenguaje máquina (código binario) se utiliza un programa llamado *ensamblador* que ejecutamos mediante la propia computadora. Este programa tomará como datos nuestro programa escrito en lenguaje ensamblador y dará como resultado el mismo programa pero escrito en lenguaje máquina, lenguaje que entiende la computadora.

Cada modelo de computadora, dependiendo del procesador que utilice, tiene su propio lenguaje ensamblador. Debido a esto decimos que estos lenguajes estan orientados a la maquina.

Hoy en dia son mas utilizados los lenguajes orientados al problema o lenguajes de alto nivel. Estos lenguajes utilizan una terminología fácilmente comprensible que se aproxima mas al lenguaje humano. En este caso la traducción es llevada a cabo por otro programa denominado *compilador.*

Cada sentencia de un programa escrita en un lenguaje de alto nivel se descompone en general en varias instrucciones en ensamblador.

A diferencia de los lenguajes ensambladores, la utilizacion de lenguajes de alto nivel no requiere en absoluto del conocimiento de la estructura del procesador que utiliza la computadora, lo que facilita la escritura de un programa.

COMPILADORES

Para traducir un programa escrito en un lenguaje de alto nivel (programa fuente) a lenguaje máquina se utiliza un programa llamado *compilador.* Este programa tomará como datos nuestro programa escrito en lenguaje de alto nivel y dara como resultado el mismo programa pero escrito en lenguaje máquina, programa que ya puede ejecutar directa o indirectamente la computadora.

Por ejemplo, un programa escrito en el lenguaje C necesita del compilador C para poder ser traducido. Posteriormente el programa traducido podrá ser ejecutado directamente por la computadora. En cambio, para traducir un programa escrito en el lenguaje Java necesita del compilador Java; en este caso, el lenguaje máquina no corresponde a la de la computadora sino a la de una maquina ficticia, denominada maquina virtual Java, que sera puesta en marcha por la computadora para ejecutar el programa.

¿Qué es una máquina virtual? Una máquina que no existe físicamente sino que es simulada en una computadora por un programa. ¿Por qué utilizar una maquina virtual? Porque, por tratarse de un programa, es muy fácil instalarla en cualquier computadora, basta con copiar ese programa en su disco duro, por ejemplo. Y, ¿que ventajas reporta? Pues, en el caso de Java, que un programa escrito en este lenguaje y compilado, puede ser ejecutado en cualquier computadora del mundo que tenga instalada esa maquina virtual. Esta solución hace posible que cualquier computadora pueda ejecutar un programa escrito en Java independiente de la plataforma que utilice, lo que se conoce como transportabilidad de programas.

INTÉRPRETES

A diferencia de un compilador, un interprete no genera un programa escrito en lenguaje máquina a partir del programa fuente, sino que efectua la traducción y ejecución simultáneamente para cada una de las sentencias del programa. Por ejemplo, un programa escrito en el lenguaje *Basic* necesita el interprete *Basic* para ser ejecutado. Durante la ejecución de cada una de las sentencias del programa, ocurre simultáneamente la traducción.

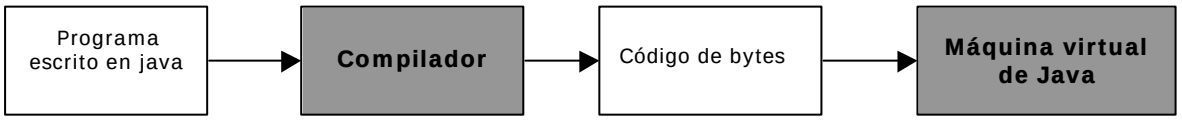
A diferencia de un compilador, un interprete verifica cada línea del programa cuando se escribe, lo que facilita la puesta a punto del programa. En cambio la ejecución resulta mas lenta ya que acarrea una traducción simultánea.

¿QUE ES JAVA?

Java es un lenguaje de programación de alto nivel con el que se pueden escribir tanto programas convencionales como para Internet.

Una de las ventajas significativas de Java sobre otros lenguajes de programación es que es independiente de la plataforma, tanto en código fuente como en binario. Esto quiere decir que el código producido por el compilador Java puede transportarse a cualquier plataforma (Intel, Sparc, Motorola, etc.) que tenga instalada una maquina virtual Java y ejecutarse. Pensando en Internet esta caracteristica es crucial ya que esta red conecta computadoras muy distintos. En cambio, C++, por ejemplo, es independiente de la plataforma solo en codigo fuente, lo cual significa que cada plataforma diferente debe proporcionar el compilador adecuado para obtener el código máquina que tiene que ejecutarse.

Segun lo expuesto, Java incluye dos elementos: un *compilador y* un *interprete.* El compilador produce un codigo de bytes que se almacena en un fichero para ser ejecutado por el interprete Java denominado *maquina virtual de Java.*



Los códigos de bytes de Java son un conjunto de instrucciones correspondientes a un lenguaje máquina que no es específico de ningún procesador, sino de la máquina virtual de Java. ¿Donde se consigue esta máquina virtual? Hoy en día casi todas las compañías de sistemas operativos y de navegadores han implementado máquinas virtuales según las especificaciones publicadas por *Sun Microsystems,* propietario de Java, para que sean compatibles con el lenguaje Java. Para las aplicaciones de Internet (denominadas *applets)* la máquina virtual está incluida en el navegador y para las aplicaciones Java convencionales, puede venir con el sistema operativo, con el paquete Java, o bien puede obtenerla a través de Internet.

¿Por que no se diseño Java para que fuera un interprete más entre los que hay en el mercado? La respuesta es porque la interpretación, si bien es cierto que proporciona independencia de la maquina, conlleva también un problema grave, y es la perdida de velocidad en la ejecución del programa. Por esta razón la solución fue disenar un compilador que produjera un lenguaje que pudiera ser interpretado a velocidades, si no iguales, si cercanas a la de los programas nativos (programas en código máquina propio de cada computadora), logro conseguido mediante la maquina virtual de Java.

Con todo, las aplicaciones todavía adolecen de una falta de rendimiento apreciable. Este es uno de los problemas que siempre se ha achacado a Java. Afortunadamente, la diferencia de rendimiento con respecto a aplicaciones equivalentes escritas en código máquina nativo ha ido disminuyendo hasta márgenes muy reducidos gracias a la utilizacion de compiladores JIT *(Just In Time* - compilacion al instante).

Un compilador JIT interacciona con la mquina virtual para convertir el código de bytes en código máquina nativo. Como consecuencia, se mejora la velocidad durante la ejecución. Sun sigue trabajando sobre este objetivo y prueba de ello son los resultados que se estan obteniendo con el nuevo y potente motor de ejecucion *HotSpot (HotSpot Perfomance Engine)* que ha diseñado, o por los microprocesadores especificos para la interpretación hardware de código de bytes.

CARACTERISTICAS

Hay muchas razones por las que Java es tan popular y útil. Aquí se resumen algunas características importantes:

- **Orientación a objetos**: Java está totalmente orientado a objetos. No hay funciones sueltas en un programa de Java. Todos los métodos se encuentran dentro de clases. Los tipos de datos primitivos, como

los enteros o dobles, tienen empaquetadores de clases, siendo estos objetos por sí mismos, lo que permite que el programa las manipule.

- **Simplicidad**: la sintaxis de Java es similar a ANSI C y C++ y, por tanto, fácil de aprender; aunque es mucho más simple y pequeño que C++. Elimina encabezados de archivos, preprocesador, aritmética de apuntadores, herencia múltiple, sobrecarga de operadores, struct, union y plantillas. Además, realiza automáticamente la recolección *de* basura, lo que hace innecesario el manejo explícito de memoria.
- **Robustez**: Por la forma en que está diseñado, Java no permite el manejo directo del hardware ni de la memoria (no permite modificar valores de punteros, por ejemplo), de modo que se puede decir que es virtualmente imposible "colgar" un programa Java. El intérprete siempre tiene el control. De hecho, el compilador es suficientemente inteligente como para no permitir una serie de acciones que podrían traer problemas, como usar variables sin inicializarlas, modificar valores de punteros directamente, acceder a métodos o variables en forma incorrecta, utilizar herencia múltiple, etc.
- **Compactibilidad**: Java está diseñado para ser pequeño. La versión más compacta puede utilizarse para controlar pequeñas aplicaciones. El intérprete de Java y el soporte básico de clases se mantienen pequeños al empaquetar por separado otras bibliotecas.
- **Portabilídad**: sus programas se compilan en el código de bytes de arquitectura neutra y se ejecutarán en cualquier plataforma con un intérprete de Java. Su compilador y otras herramientas están escritas en Java. Su intérprete está escrito en ANSI C. De cualquier modo, la especificación del lenguaje Java no *tiene* características *dependientes de* la implantación.
- **Multiplataforma**: Al ser un lenguaje que ejecuta el codigo sobre una maquina virtual, cambiando la maquina virtual es posible ejecutar un programa sobre una base diferente sin modificarlo.
- **Amigable para el trabajo en red**: Java tiene elementos integrados para comunicación en red, applets Web, aplicaciones cliente-servidor, además de acceso remoto a bases de datos, métodos y programas.
- **Soporte a GUI**: la caja de herramientas para la creación de ventanas abstractas -(Abstract Windowing Toolkit) de Java simplifica y facilita la escritura de programas GUI (Graphic User Interface) orientados a eventos con muchos componentes de ventana.
- **Carga y vinculación incremental dinámica**: las clases de Java se vinculan dinámicamente al momento de la carga. Por tanto, la adición de nuevos métodos y campos de datos a clases, no requieren de recompilación de clases del cliente.
- **Internacionalización:** los programas de Java están escritos en Unicode, un código de caracter de 16 bits que incluye alfabetos de los lenguajes más utilizados en el mundo. La manipulación de los caracteres de Unicode y el soporte para fecha/hora local, etcétera, hacen que Java sea bienvenido en todo el mundo.
- **Hilos**: Java proporciona múltiples flujos de control que se ejecutan de manera concurrente dentro de uno de sus programas. Los hilos permiten que su programa emprenda varias tareas de cómputo al mismo tiempo, una característica que da soporte a programas orientados a eventos, para trabajo en red y de animación.
- **Seguridad**: entre las medidas de seguridad de Java se incluyen restricciones en sus applets, implantación redefinible de sockets y objetos de administrador de seguridad definidos por el usuario. Hacen que las applets sean confiables y permiten que las aplicaciones implanten y se apeguen a reglas de seguridad personalizadas.

INSTALACION DE JAVA

Para configurar su ambiente Java, simplemente baje la versión mas reciente de JDK del sitio Web de JavaSoft:

http://www.javasoft.com/products/jdk/ e instálelo siguiendo las instrucciones.

PAQUETE DE CODIGO EJEMPLO JAVA

Un paquete de código ejemplo está disponible en el sitio web de la facultad en: http://labsys.frc.utn.edu.ar, dirigirse a los sitios de las cátedra/PPR-2003/Unidad 1.

REALIZACION DE UN PROGRAMA EN JAVA

Para desarrollar un programa hay que seguir los siguientes pasos:

1. Editar el programa
2. Compilarlo
3. Ejecutarlo
4. Depurarlo

Para poder escribir programas se necesita un entorno de desarrollo Java. *Sun Microsystems,* propietario de Java, proporciona uno de forma gratuita, *Java Development Kit* (JDK), que se puede obtener en la direccion de Internet: http://www.sun.com

Cómo crear un programa

Un programa puede ser una *aplicación* o un *applet.* En esta materia aprenderá principalmente a escribir aplicaciones Java. Empecemos con la creación de una aplicación sencilla: el clásico ejemplo de mostrar un mensaje de saludo.

Esta sencilla aplicación la realizaremos desde los dos puntos de vista comentados anteriormente: utilizando la interfaz de línea de ordenes del JDK y utilizando un entorno de desarrollo integrado.

Interfaz de Línea de ordenes

Empecemos por editar el fichero fuente Java correspondiente a la aplicación. Primeramente visualizaremos el editor de textos que vayamos a utilizar; por ejemplo, el *Bloc de notas o el Edit.* El nombre del fichero elegido para guardar el programa en el disco, debe tener como extension *Java;* por ejemplo *HolaMundoJava.*

Una vez visualizado el editor, escribiremos el texto correspondiente al programa fuente. Escríbalo tal como se muestra a continuación. Observe que cada *sentencia* del lenguaje Java finaliza con un *punto y coma y* que cada *linea del programa* se finaliza pulsando la tecla *Entrar (Enter o .--).*

```
class HolaMundo
{
	/* Punto de entrada a la aplicacion.
	* args: matriz de parametros pasados a la aplicacion
	* mediante la linea de ordenes. Puede estar vacia. */

	public static void main(String[] args)
	{
		System.out.println("Hola mundo!!!");
	}
}
```

¿Que hace este programa?

Comentamos brevemente cada línea de este programa. No apurarse si algunos de los términos no quedan muy claros ya que todos ellos se verán con detalle más adelante.

La primera línea declara la clase de objetos *HolaMundo,* porque el esqueleto de cualquier aplicación Java se basa en la definición de una clase. A continuación se escribe el cuerpo de la clase encerrado entre los caracteres { y }. Ambos caracteres definen un bloque de código. Todas las acciones que va a llevar a cabo un programa Java se colocan dentro del bloque de código correspondiente a su clase. Las clases son la base de los programas Java.

Las siguientes lineas encerradas entre /* y */ son simplemente un comentario. Los comentarios no son tenidos en cuenta por el compilador, pero ayudan a entender un programa cuando se lee.

A continuacion se escribe el metodo principal main. Observe que un método se distingue por el modificador () que aparece despues de su nombre y que el bloque de código correspondiente al mismo define las acciones que tiene que ejecutar dicho método. Cuando se ejecuta una aplicación, Java espera que haya un método main. Este método define el punto de entrada y de salida de la aplicación.

El metodo println del objeto out miembro de la clase System de la biblioteca Java, escribe como resultado la expresión que aparece especificada entre comillas. Observe que la sentencia completa finaliza con punto y coma.

Método main

Toda aplicación Java tiene un método denominado main, y solo uno. Este método es el punto de entrada a la aplicación y también el punto de salida. Su definición es como se muestra a continuación:

```
Public static void main (String[] args)
{
      // Cuerpo del método
}
```

Como se puede observar, el método **main** es publico (**public**), estático (**static**), no devuelve nada (**void**) y tiene como argumento un vector de tipo **String** que almacenará los argumentos pasados en la línea de órdenes cuando se invoca a la aplicación para su ejecución, concepto que estudiaremos posteriormente.

Guardar el programa escrito en el disco

El programa editado esta ahora en la memoria. Para que este trabajo pueda tener continuidad, el programa escrito se debe grabar en el disco utilizando la orden correspondiente del editor. Muy importante: el nombre del programa fuente debe ser el mismo que el de la clase que contiene, respetando mayusculas y minúsculas. En nuestro caso, el nombre de la clase es *HolaMundo,* por to tanto el fichero debe guardarse con el nombre *HolaMundo java.*

Compilar y ejecutar el programa

El siguiente paso es *compilar* el programa; esto es, traducir el programa fuente a código de bytes para posteriormente poder ejecutarlo. La orden para compilar el programa *HolaMundo java* es la siguiente:

```
javac HolaMundo.java
```

Previamente, para que el sistema operativo encuentre la utilidad *javac,* utilizando la linea de ordenes hay que añadir a la variable de entorno *path* la ruta de la carpeta donde se encuentra esta utilidad y otras que utilizaremos a continuación. Por ejemplo:

```
path=%path%;c:\java\j2sdk1.4.0\bin
```

La expresion *%path%* representa el valor actual de la variable de entorno *path.* Una ruta va separada de la anterior por un punto y coma.

Para compilar un programa hay que especificar la extension *Java. El* resultado de la compilación será un fichero *HolaMundo.class* que contiene el código de bytes que ejecutará la máquina virtual de Java.

Al compilar un programa, se pueden presentar *errores de compilación,* debidos a que el programa escrito no se adapta a la sintaxis y reglas del compilador. Estos errores se irán corrigiendo hasta obtener una compilación sin errores.

Para ejecutar el fichero resultante de la compilación, invocaremos desde la línea de ordenes al interprete de código de bytes *java* con el nombre de dicho fichero como argumento, en nuestro caso *HolaMundo, y* pulsaremos *Entrar* para que se muestren los resultados.

En la figura siguiente se puede observar el proceso seguido para ejecutar *HolaMundo* desde la línea de ordenes. Hay que respetar las mayúsculas y las minúsculas en los nombres que se escriben. Asi mismo, cabe resaltar que la extension. *class* no tiene que ser especificada.

Una vez ejecutado, se puede observar que el resultado es el mensaje: *Hola mundo!!!.*

Biblioteca de funciones

Java carece de instrucciones de E/S, de instrucciones para manejo de cadenas de caracteres, etc. con lo que este trabajo queda para la biblioteca de clases provista con el compilador. Una biblioteca es un fichero separado en el disco (con extension *lib, jar o dll)* que contiene las clases que definen las tareas más comunes,

para que nosotros no tengamos que escribirlas. Como ejemplo, hemos visto anteriormente el método println del objeto out miembro de la clase System. Si este método no existiera, seria labor nuestra el escribir el código necesario para visualizar los resultados en la ventana.

En el código escrito anteriormente se puede observar que para utilizar un método de una clase de la biblioteca simplemente hay que invocarlo para un objeto de su clase y pasarle los argumentos necesarios entre parentesis. Por ejemplo:

```
System.out.println("Hola mundo!!!");
```

Guardar el programa ejecutable en el disco

Como hemos visto, cada vez que se realiza el proceso de *compilacion* del programa actual, Java genera automáticamente sobre el disco un fichero *class.* Este fichero puede ser ejecutado directamente desde el sistema operativo, con el soporte de la máquina virtual de Java, que se lanza invocando a la utilidad *java* con el nombre del fichero como argumento.

Al ejecutar el programa, pueden ocurrir *errores durante la ejecución. Por* ejemplo, puede darse una division por cero. Estos errores solamente pueden ser detectados por Java cuando se ejecuta el programa y seran notificados con el correspondiente mensaje de error.

Hay *otro tipo de errores* que no dan lugar a mensaje alguno. Por ejemplo: un programa que no terrnine nunca de ejecutarse, debido a que presenta un lazo, donde no se llega a dar la condición de terminación. Para detener la ejecución se tienen que pulsar las teclas *Ctrl+C* (en un entorno integrado se ejecutará una orden equivalente *a Detener ejecución).*

Depurar un programa

Una vez ejecutado el programa, la solución puede ser incorrecta. Este caso exige un análisis minucioso de como se comporta el programa a lo largo de su ejecución; esto es, hay que entrar en la fase de *depuración* del programa.

La forma más sencilla y eficaz para realizar este proceso, es utilizar un programa *depurador. El* entorno de desarrollo de Java proporciona para esto la utilidad *jdb.* Este es un depurador de línea de órdenes un tanto complicado de utilizar, por lo que, en principio, tiene escasa aceptación. Normalmente los entornos de desarrollo integrados que anteriormente hemos mencionado incorporan las órdenes necesarias para invocar y depurar un programa.

Para depurar un programa Java debe compilarlo con la opcion -g. Por ejemplo, desde la línea de órdenes esto se haría asi:

```
javac -g Aritmetica.java
```

Desde un entorno integrado, habrá que establecer la opción correspondiente del compilador.

Una vez compilado el programa, se inicia la ejecución en modo depuración y se continua la ejecución con las órdenes tipicas de ejecución paso a paso.

Entorno de desarrollo integrado

Cuando se utiliza un entorno de desarrollo integrado lo primero que hay que hacer una vez instalado dicho entorno es asegurarse de que las opciones que indican las rutas de las herramientas Java, de las bibliotecas, de la documentacion y de los fuentes, o bien simplemente de la ruta donde se instalo el JDK, están establecidas.

La ubicación de estas opciones variará de acuerdo al entorno integrado que usted elija, generalmente puede estra en algún menú *Opciones o* equivalente.

Para editar y ejecutar la aplicaci6n *HolaMundo* anterior utilizando este entorno de desarrollo integrado, los pasos a seguir se indican a continuaci6n:

1. Suponiendo que ya esta visualizado el entorno de desarrollo, anadimos un nuevo fichero Java *(File, New),* editamos el c6digo que compone la aplicación y lo guardamos con el nombre *HolaMundo.java (File, Save).*
2. A continuaci6n, para compilar la aplicación, ejecutamos la orden *Compile* del menu *Compiler.*
3. Finalmente, para ejecutar la aplicación, suponiendo que la compilación se efectuó satisfactoriamente, seleccionaremos la orden *Run* del menu *Run.*

Si se implementa aplicaciones compuestas por varios ficheros fuente. En este caso, los pasos a seguir son los siguientes:

1. Suponiendo que ya esta visualizado el entorno de desarrollo, anadimos *(File, New)* y editamos cada uno de los ficheros que componen la aplicación y los guardamos con los nombres adecuados *(File, Save).*

2. Para compilar la aplicación visualizaremos el fichero que contiene la clase aplicación (la clase que contiene el método main) y ejecutamos la orden *Compile* del menu *Compiler.* Se puede observar que todas las clases que son requeridas por esta clase principal son compiladas automáticamente, independientemente de que estén, o no, en diferentes ficheros.

3. Finalmente, suponiendo que la compilación se efectuó satisfactoriamente, ejecutaremos la aplicación seleccionando la orden *Run,* del menu *Run.* Este paso exige que se esté visualizando el fichero que contiene la clase aplicación.

Cuando una aplicación consta de varios ficheros puede resultar más cómodo para su mantenimiento crear un proyecto. Esto permitirá presentar una ventana que mostrará una entrada por cada uno de los ficheros fuente que componen la aplicación. Para visualizar el código de cualquiera de estos ficheros, bastará con hacer doble clic sobre la entrada correspondiente.

Para crear un proyecto, los pasos a seguir son los siguientes:

1. Suponiendo que ya está visualizado el entorno de desarrollo, se añaden *(File, New)* y editan cada uno de los ficheros que componen la aplicación y se guardan con los nombres adecuados *(File, Save).*

2. Despues se crea un nuevo proyecto *(Project, New Project Workspace).* En la ventana que se visualiza, haciendo clic en el boton *Browse* correspondiente, se introduce un nombre para el proyecto y se selecciona el nombre de la clase aplicación (la clase que contiene el metodo main). Posteriormente, para visualizar la ventana del proyecto, se ejecuta la orden *Show Project Workspace Window* del menu *Project.*

3. El paso siguiente es añadir al proyecto el resto de los ficheros que lo componen. Para ello se ejecuta la orden *Edit Project* del menu *Project, y* a través del boton *Add Files* se añaden dichos ficheros.

4. Finalmente, para compilar y ejecutar la aplicación, se procede como se explicó anteriormente.

GRAMATICA DE JAVA

En general la gramática de java se parece a la del C/C++, vamos a ver los puntos más importantes para poder escribir un programa en java.

CONVENCIONES LEXICAS

SENTENCIAS SIMPLES

Una sentencia simple es la unidad ejecutable más pequeña de un programa Java. Las sentencias controlan el flujo u orden de ejecución. Una sentencia Java puede formarse a partir de una palabra clave (foir, while, if...else, etc), expresiones, declaraciones o llamadas a métodos. Toda sentencia simple termina con un punto y coma (;).

Por ejemplo:

```
x = y;
y = y + 1;
h=5;          g=10;
suma ();
```

SENTENCIAS COMPUESTAS O BLOQUE

Una sentencia compuesta o bloque, es una colección de sentencias simples incluidas entre llaves { } . Un bloque puede contener a otros bloques. Por ejemplo:

```
{
        ...
        {
          ...
        }
}
```

COMENTARIOS

En Java hay tres tipos de comentarios:

```
// comentarios para una sola línea

/* comentarios de una o
    más líneas
*/

/** comentario de documentación, de una o más líneas
*/
```

Los dos primeros tipos de comentarios son los que todo programador conoce y se utilizan del mismo modo. Los comentarios de documentación, colocados inmediatamente antes de una declaración (de variable o función), indican que ese comentario ha de ser colocado en la documentación que se genera automáticamente cuando se utiliza la herramienta de Java *javadoc*. Dichos comentarios sirven como descripción del elemento declarado permitiendo generar una documentación de nuestras clases escrita al mismo tiempo que se genera el código.

En este tipo de comentario para documentación, se permite la introducción de algunos tokens o palabras clave, que harán que la información que les sigue aparezca de forma diferente al resto en la documentación.

PALABRAS CLAVE

Las siguientes son las palabras clave que están definidas en Java y que no se pueden utilizar como indentificadores:

```
abstract    continue    for         new         switch
```

```
boolean     default     goto        null        synchronized
break       do          if          package     this
byte        double      implements  private     threadsafe
byvalue     else        import      protected   throw
case        extends     instanceof  public      transient
catch       false       int         return      true
char        final       interface   short       try
class       finally     long        static      void
const       float       native      super       while
```

PALABRAS RESERVADAS

Además, el lenguaje se reserva unas cuantas palabras más, pero que hasta ahora no tienen un cometido específico. Son:

```
cast        future      generic     inner
operator    outer       rest        var
```

SEPARADORES

Sólo hay un par de secuencias con otros caracteres que pueden aparecer en el código Java; son los separadores simples, que van a definir la forma y función del código. Los separadores admitidos en Java son:

() - paréntesis. Para contener listas de parámetros en la definición y llamada a métodos. También se utiliza para definir precedencia en expresiones, contener expresiones para control de flujo y rodear las conversiones de tipo.

{} - llaves. Para contener los valores de matrices inicializadas automáticamente. También se utiliza para definir un bloque de código, para clases, métodos y ámbitos locales.

[ ] - corchetes. Para declarar tipos matriz. También se utiliza cuando se referencian valores de matriz.

; - punto y coma. Separa sentencias.

, - coma. Separa identificadores consecutivos en una declaración de variables. También se utiliza para encadenar sentencias dentro de una sentencia for.

. - punto. Para separar nombres de paquete de subpaquetes y clases. También se utiliza para separar una variable o método de una variable de referencia.

TIPOS DE DATOS

Los tipos de datos en Java se clasifican en: tipos *primitivos* y tipos *referenciados.*

Tipos primitivos

Hay ocho tipos primitivos de datos que podemos clasificar en: tipos numéricos y el tipo **boolean.** A su vez, los tipos numéricos se clasifican en tipos enteros y tipos reales.

Tipos enteros: **byte, short, int, long** y **char.**
Tipos reales: **float** y **double.**

Cada tipo primitivo tiene un rango diferente de valores positivos y negativos, excepto el **boolean** que solo tiene dos valores: **true** y **false.** El tipo de datos que se seleccione para declarar las variables de un determinado programa dependerá del rango y tipo de valores que vayan a almacenar cada una de ellas y de si estos son enteros o fraccionarios.

Se les llama primitivos porque estan integrados en el sistema y en realidad no son objetos, lo cual hace que su uso sea mas eficiente. Mas adelante veremos también que la biblioteca Java proporciona las clases: **Byte, Character, Short, Integer, Long, Float, Double** y **Boolean,** para encapsular cada uno de los tipos expuestos, proporcionando asi una funcionalidad añadida para manipularlos.

La siguiente tabla muestra los tipos primitivos:

Tipo	*Tamaño/Formato*	**Descripción**
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
Float	32-bit IEEE 754	Punto flotante, precisión simple
double	64-bit IEEE 754	Punto flotante, precisión doble
char	16-bit caracter Unicode	Un carácter
boolean	true, false	Valor booleano (verdadero o falso)

El tamaño de estos tipos está fijado, siendo independiente del microprocesador y del sistema operativo sobre el que esté implementado. Esta característica es esencial para el requisito de la portabilídad entre distintas plataformas.

La razón de que se codifique los char con 2 bytes es para permitir implementar el juego de caracteres Unicode, mucho más universal que ASCII, y sus numerosas extensiones.

Java provee para cada tipo primitivo una clase correspondiente: Boolean, Character, Integer, Long, Float y Double.

¿Por qué existen estos tipos primitivos y no sólo sus objetos equivalentes? La razón es sencilla, por eficiencia. Estos tipos básicos son almacenados en una parte de la memoria conocida como el *Stack,* que es manejada directamente por el procesador a través de un registro apuntador *(stack pointer).* Esta zona de memoria es de rápido acceso, pero tiene la desventaja de que el compilador de java debe conocer, cuando está creando el programa, el tamaño y el tiempo de vida de todos los datos allí almacenados para poder generar código que mueva el *stack pointer,* lo cual limita la flexibilidad de los programas. En cambio, los objetos son creados en otra zona de memoria conocida como *Heap.* Esta zona es de propósito general y, a diferencia del *Stack,* el compilador no necesita conocer ni el tamaño, ni el tiempo de vida de los datos allí alojados. Este enfoque es mucho más flexible pero, en contraposición, el tiempo de acceso a esta zona es más elevado que el necesario para acceder al *stack.*

Aunque en la literatura se comenta que Java eliminó los punteros, esta afirmación es inexacta. Lo que no se permite es la aritmética de punteros. Cuando estamos manejando un objeto en Java, realmente estamos utilizando un *handie* a dicho objeto. Podemos definir un *handie* como una variable que contiene la dirección de memoria donde se encuentra el objeto almacenado.

Tipos referenciados

Hay tres clases de tipos referenciados: clases, interfaces *y arrays.* Todos ellos seran objeto de estudio en capitulos posteriores.

LITERALES

Un literal es la expresión de un valor de un tipo primitivo, de un tipo String (cadena de caracteres) o la expresion null (valor nulo o desconocido). Por ejemplo, son literales: *5, 3.14, `a', "hola" y* null. En realidad son valores constantes.

Un literal en Java puede ser: un entero, un real, un valor booleano, un caracter, una cadena de caracteres y un valor nulo.

IDENTIFICADORES

Los identificadores son nombres dados a tipos, literales, variables, clases, interfaces, metodos, paquetes y sentencias de un programa, cualquier cosa que el programador necesite identificar o usar.

En Java, un identificador comienza con una letra, un subrayado (_) o un símbolo de dólar ($). Los siguientes caracteres pueden ser letras o dígitos. **Se distinguen las mayúsculas de las minúsculas** y no hay longitud máxima.

Serían identificadores válidos:

identificador
nombre_usuario

Nombre_Usuario
_variable_del_sistema
$transaccion
y su uso sería, por ejemplo:

```
int contador_principal;
char _lista_de_ficheros;
float $cantidad_en_Ptas;
```

VARIABLES

Una variable representa un espacio de memoria para almacenar un valor de un determinado tipo. El valor de una variable, a diferencia de una constante, puede cambiar durante la ejecución de un programa. Para utilizar una variable en un programa, primero hay que declararla.

También podemos definirla como una unidad de almacenamiento identificada por un nombre y un tipo de dato, tiene una dirección (ubicación) dentro de la memoria y puede almacenar información según el tipo de dato que tenga asociado. Si imaginamos la memoria del computador como un conjunto de celdas, entonces podemos decir que una variable es una de esas celdas que el usuario identifica con un nombre.

Género de las variables

En informática hay dos maneras de almacenar las variables en un método:

se almacena el valor de la variable;

se almacena la dirección donde se encuentra la variable.

En Java:

los tipos elementales se manipulan directamente: se dice que se manipulan por valor;

los objetos se manipulan a través de su dirección: se dice que se manipulan por referencia.

Por ejemplo:

```
void Metodo() {
int var1  = 2;
Objeto var2 = new Objeto();
var1  = var1  + 3;
var2.agrega  (3);
}
```

En el ejemplo anterior, var1 representa físicamente el contenido de la variable mientras que var2 sólo representa la dirección que permite acceder a ella: se crea un objeto en algún lugar de la memoria y var2 representa físicamente el medio de acceder a él.

Cuando se añade 3 a var1, se modifica la variable var1, mientras que el método agrega (3) no modifica var2, sino el objeto designado por var2.

A priori preferirá var1, que le parecerá más simple. Es cierto, pero este mecanismo es mucho más restrictivo e impide por ejemplo compartir datos entre diferentes partes de la aplicación.

Además, el almacenamiento al estilo de var2 es indispensable cuando los objetos son algo más que tipos simples.

En Java, para todas las variables de tipo básico se accede al valor asignado a ellas directamente (no se conoce la dirección de memoria que ocupa). Para las demás (arrays, clases o interfaces), se accede a la variable a través de un puntero. El valor del puntero no es accesible ni se puede modificar como en C/C++; Java no lo necesita, además, esto atentaría contra la robustez del lenguaje.

De hecho, en Java no existen los tipos pointer, struct o union. Un objeto es más que una estructura, y las uniones no se hacen necesarias con un método de programación adecuado (además se evita la posibilidad de acceder a los datos incorrectamente).

Definición y declaracion

La declaracion introduce uno o más identificadores en un programa sin asignación de memoria. La definición de una variable, declara la variable y además le asigna memoria.

Generalmente las variables de tipos simples se declaran y definen al mismo tiempo, en cambio las variables objetos pueden declararse primero y luego definirse mas adelante en el programa (asignarle memoria).

Toda variable debe ser definida antes de ser utilizada.

La declaración de una variable esta compuesta por un identificador o nombre para esa variable, precedida por el tipo de dato que se puede almacenar en la misma.

En general:

Tipo Nombre o identificador;

- Los nombres o identificadores de las variables consisten en secuencias de letras y dígitos, donde el primer caracter debe ser siempre una letra.
- Java establece una diferencia entre las letras mayúsculas y minúsculas.

  PRIMERA Primera primera
- Ejemplos:

```
int xPos;
double Precio, Cantidad;
char asterisco;
int Resultado;
```

Asignaciones a variables

La operación de asignación está compuesta de tres elementos: la variable que va a recibir el valor, el operador de asignación y el valor que se va a asignar. La variable sólo podrá ser una que haya sido declarada. El operador de asignación es el símbolo = y por último el valor asignar, que puede ser un literal otra variable, una expresión, etc., siendo la única condición que el resultado sea del mismo tipo que la variable que va a recibir el valor, o de algún tipo compatible.

En general:

Variable = Constante | Expresión;

Por ejemplo:

```
suma = suma + 1;
precio = 1.05 * precio;
```

Inicialización de las variables

En Java no se puede utilizar una variable sin haberla inicializado previamente. El compilador señala un error cuando se utiliza una variable sin haberla inicializado.

Veamos un ejemplo. La compilación de:

```
import java.io.*;
class DemoVariable {
        public static void main (String argv[]) {
                int noInicializado;
                System.out.println("Valor del entero: "+noInicializado);
        }
}
```

provoca el error siguiente:

```
c: \ProgramasJava\cramatica>javac init.java
init.java:6: Variable noInicializada may not have been initialized
System.out.println ("Valor del entero:"+ noInicializado);

1 error
```

OPERADORES

Un operador es un elemento del lenguaje, generalmente representado por un símbolo, cuya finalidad es generar un resultado manipulando uno o dos operandos. Si el operador trabaja sobre un solo operando se dice que es unario, mientras que si lo hace sobre dos se llama binario.

OPERADORES ARITMETICOS

Estos operadores son binarios, a excepción de (cambio de signo) que es unitario.

+ : Suma	- : Resta
* : Multiplicación	/ : División
% : Resto de división entera	- : Cambio de signo
++ : incremento	-- : decremento

El orden de prioridades de estos operadores es como en las matemáticas : *, / y % tienen la prioridad mas alta.

Los operadores de incremento y decremento son unitarios y son los únicos que pueden se utilizados como pre y post fijos del operador.

Por ejemplo:

Operación	**Funcionamiento**
x++	suma 1 a x equivalente a x=x+1;
++x	ídem
y = x++	Asigna a "y" el valor de "x" y luego suma 1 a "x"
y = ++x	Suma 1 a "x" y luego asigna a "y" el nuevo valor de "x"
y = x--	Asigna a "y" el valor de "x" y luego resta 1 a "x"
y = --x	Resta 1 a "x" y luego asigna a "y" el nuevo valor de "x"

Supongamos : x = 5 Al finalizar la operación

Operación	**x**	**y**
y = x++	6	5
y = ++x	6	6
y = x--	5	4
y = --x	4	4

OPERADORES DE ASIGNACION

La operación de asignación consiste en “copiar” el valor de una expresión en una variable. Cuando decimos el valor de una expresión, queremos decir que este valor puede ser el resultado de alguna operación matemática, o bien el valor de otra variable y también el resultado devuelto por una función o mandato, aunque sea definido por el usuario.

En general:

```
variable = expresión;
```

Existen, como se deja traslucir de esta regla sintáctica, tres partes en una operación de asignación, a saber: primero el receptor, que siempre es el nombre de una variable. Segundo, el operador de asignación en sí y por último la expresión de la que ya se habló con anterioridad y que es la parte transmisora del valor asignado al receptor.

El destino, o la parte izquierda, de la asignación debe ser una variable (no una función, ni una constante).

El resultado de una operación de asignación es que el valor de la expresión que se encuentra a la derecha del operador (“=“) es asignado a la variable que se encuentra a la izquierda del mismo.

Una expresión con un operador de asignación puede ser utilizada en una expresión como la siguiente:

```
valor = 5 ;                    //asigna 5 a la variable valor
valor1 = 8 * (valor2 = 5);     // primero se asigna 5 a valor2. Este se multiplica
                               //por 8, por lo que el valor final de valor1 es 40.
```

ASIGNACIONES MULTIPLES

Se permite asignar a muchas variables el mismo valor utilizando asignaciones múltiples en una sola sentencia. En los programas a menudo se usa éste método para asignar valores comunes a las variables. Por ejemplo:

```
valor1 = valor2 = valor3 = 0 ;
```

ASIGNACIONES EN LA DECLARACION DE VARIABLES O INICIALIZACIÓN

Java permite asignar valores a las variables en el momento en que estas están siendo declaradas. A esta operación se la denomina Inicialización y tiene el mismo sentido que cuando inicializamos una variable previamente declarada; en cualquier punto del programa. Solo que en este caso las variables toman su valor en el momento de compilación. Para llevar a cabo esta tarea, basta con declarar la variable y en esa misma línea agregar el operador de asignación y el valor que se le quiere asignar.

Por ejemplo:

```
int valor1=0;
```

Es muy importante hacer notar el hecho de que no pueden combinarse las características de asignación múltiple y de inicialización ya que este lenguaje no lo permite, y esto se basa en el simple hecho de que si estamos declarando una variable, recién después de que esta haya tomado su valor en memoria, puede tomar su valor, y la expresión de asignación múltiple nos exige, indicar el valor de la variable al final de la sentencia, y si analizamos esto, le estaríamos pidiendo al compilador que realice una tarea imposible, ya que primero debe buscar lugar en memoria para todas las variables declaradas en el orden en que se declararon, y luego debe

retroceder para tomar el valor de la última y asignárselo a las demás en orden inverso al orden de declaración. Le dejamos un ejemplo para que Ud. piense y analice acerca de él, siempre haciendo hincapié en el hecho de que este ejemplo no es válido:

```
int valor1=valor2=valor3=10;
```

Además, y para finalizar, es de destacar el tema de que para definir muchas variables en una sola línea de programa, estas deben ir separadas por una "," y no por un "=".

OPERADORES DE ASIGNACION COMPUESTOS

Es un conjunto adicional de operadores de asignación que permiten expresar ciertos cálculos de modo más conciso.

+ =	Suma y Asignación	- =	Resta y Asignación
* =	Multiplicación y Asignación	/ =	División y Asignación
% =	Resto Modulo y Asignación		

Como podemos observar la mayoría de ellos resultan de la combinación del operador "=" (asignación directa) con operadores Aritméticos.

donde la expresión: x+=15
es equivalente a: x=x+15

OPERADORES RELACIONALES

Sirven para obtener una expresión que no es un resultado numérico sino una afirmación o negación de una determinada relación entre dos operandos.

> : Mayor que	< : Menor que
>= : Mayor o igual que	=< : Menor o igual que
== : Igual que (operador de igualdad)	
!= : Distinto de (operador de igualdad)	

OPERADORES LOGICOS

Permiten crear relaciones complejas. Se utilizan para vincular el resultado de distintas expresiones en donde intervienen operadores relacionales. Estos operadores siempre se evalúan de izquierda a derecha, deteniéndose la evaluación tan pronto como se produzca un fallo en la condición (para el and) o se cumpla (para el or). La prioridad de ejecución más alta corresponde a los operadores relacionales.

&& : y lógico (and) || : o lógico (or) ! : negación (not)

En la siguiente tabla aparecen los operadores que se utilizan en Java, por orden de precedencia:

operadores	
posfijos	[] . (parámetros) expr++ expr--
Operadores unarios	++expr --expr +expr –expr ~ !
Creación y "cast"	new (tipo)
Multiplicativos	* / %
Aditivos	+ -
Desplazamiento	<< >> >>>
Relacionales	< > <= >= instanceof
Igualdad	== !=
AND bit a bit	&
OR exclusivo bit a bit	^
OR inclusivo bit a bit	|
AND lógico	&&
OR lógico	||

Condicional	? :						
Asignación	=	+=	-=	*=	/=	%	=
	^=	&=	|=	<<=	>>=	>>>=	

Los operadores numéricos se comportan como esperamos. hay operadores unarios y binarios, según actúen sobre un solo argumento o sobre dos.

Operadores unarios

Incluyen, entre otros: +,-, ++, - -, ~, !, (tipo)
Se colocan antes de la constante o expresión (o, en algunos casos, después). Por ejemplo:

```
-cnt;              //cambia de signo; por ejemplo si cnt es 12 el
                       //resultado es -12 (cnt no cambia)
++cnt;             //equivalen a cnt+=1;
cnt++;
--cnt;             //equivalen a cnt-=1;
cnt--;
```

Operadores binarios

Incluyen, entre otros: +, -, *, /, %
Van entre dos constantes o expresiones o combinación de ambas. Por ejemplo:

```
cnt + 2            //devuelve la suma de ambos.
promedio + (valor/2)
horas / hombres;          //división.
acumulado % 3;            //resto de la división entera entre ambos.
```

Nota: + sirve también para concatenar cadenas de caracteres. Cuando se mezclan Strings y valores numéricos, éstos se convierten automáticamente en cadenas:

```
"La frase tiene " + cant + " letras"
```

se convierte en:

```
"La frase tiene 17 letras"     //suponiendo que cant = 17
```

Los operadores relacionales devuelven un valor booleano.

El operador = siempre hace copias de objetos, marcando los antiguos para borrarlos, y ya se encargará el garbage collector de devolver al sistema la memoria ocupada por el objeto eliminado.

Veamos algunos ejemplos:

[ ] define arreglos: `int lista [ ];`
(params) es la lista de parámetros cuando se llama a un método: `convertir (valor, base);`
new permite crear una instancia de un objeto: `new contador();`
(type) cambia el tipo de una expresión a otro: `(float) (total % 10);`
>> desplaza bit a bit un valor binario: `base >> 3;`
<= devuelve "true" si un valor es menor o igual que otro: `total <= maximo;`
instanceof devuelve "true" si el objeto es una instancia de la clase: `papa instanceof Comida;`
|| devuelve "true" si cualquiera de las expresiones es verdad: `(a<5) || (a>20).`

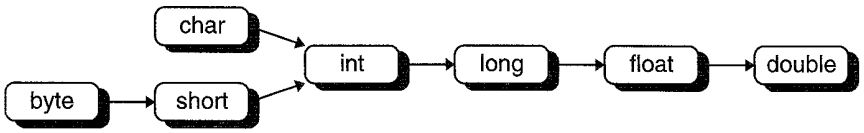
Expresiones numéricas

Una expresión es un conjunto de operandos unidos mediante operadores para especificar una operación determinada. Todas las expresiones cuando se evaluan retornan un valor. Por ejemplo:

```
a + 1
suma + c
cantidad * precio
7 * Math.sqrt(a) - b / 2   (sqrt indica raiz cuadrada)
```

Conversión entre tipos de datos

Cuando Java tiene que evaluar una expresión en la que intervienen operandos de diferentes tipos, primero convierte, solo para realizar las operaciones solicitadas, los valores de los operandos al tipo del operando cuya precisión sea mas alta. Cuando se trate de una asignación, convierte el valor de la derecha al tipo de la variable de la izquierda siempre que no haya perdida de información. En otro caso, Java exige que la conversión se realice explícitamente. La figura siguiente resume los tipos colocados de izquierda a derecha de menos a mas precisos; las flechas indican las conversiones implicitas permitidas:




```
//Conversion implicita
byte bDato = 1; short sDato = 0; int iDato = 0: long lDato = 0;
float fDato = 0; double dDato = 0;
sDato = bDato;
iDato = sDato;
lDato = iDato;
fDato = lDato;
dDato = fDato + lDato - iDato * sDato / bDato;
System.out.println(dDato); // resultado: 1.0
```

Java permite una conversion explicita (conversion forzada) del tipo de una expresión mediante una construcción denominada *cast,* que tiene la forma:

(tipo) expresion

Cualquier valor de un tipo entero o real puede ser convertido a o desde cualquier tipo numérico. No se pueden realizar conversiones entre los tipos enteros o reales y el tipo Boolean. Por ejemplo:

```
// Conversion explicita (cast)
byte bDato = 0; short sDato = 0; int iDato = 0; long lDato 0;
float fDato = 0; double dDato = 2;

fDato = (float)dDato;
lDato = (long)fDato;
iDato = (int)lDato;
sDato = (short)iDato;
bDato = (byte)(sDato + iDato - lDato * fDato / dDato);
System.out.println(bDato); // resultado: 2
```

La expresión es convertida al tipo especificado si esa conversión esta permitida; en otro caso, se obtendrá un error. La utilización apropiada de construcciones *cast* garantiza una evaluación consistente, pero siempre que se pueda, es mejor evitarla ya que suprime la verificación de tipo proporcionada por el compilador y por consiguiente puede conducir a resultados inesperados, o cuando menos, a una perdida de precisión en el resultado. Por ejemplo:

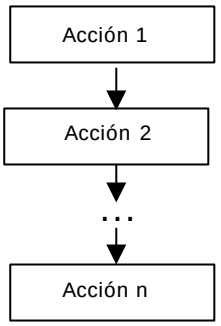
```
float r;
r = (float)Math.sgrt(10);  // el resultado se redondea perdiendo precisión ya que sqrt
                           //devuelve un valor de tipo double
```

ESTRUCTURAS DE CONTROL

Es la manera como se van encadenando, uniendo entre sí las acciones, dando origen a los distintos tipos de estructuras.

Estructura secuencial

La estructura secuencial es aquélla en la que una acción sigue a otra en secuencia. Las tareas se suceden de tal modo que la salida de una es la entrada de la siguiente y así sucesivamente hasta el fin del proceso.



Acción 1
Acción 2
...
Acción n

Estructuras selectivas (alternativas, de decisión)

Cuando el programa desea especificar dos o más caminos alternativos, se deben utilizar estructuras selectivas o de decisión. Una instrucción de decisión o selección evalúa una condición y en función del resultado de esa condición se bifurcará a un determinado punto.

Las estructura selectivas se utilizan para tomar decisiones lógicas; de ahí que se suelan denominar también estructuras de decisión o alternativas.

En la estructura selectiva se evalúa una condición y en función del resultado de la misma se realiza una opción u otra. Las condiciones se especifican usando expresiones lógicas. La representación de una estructura selectiva se hace con palabra en pseudocódigo, con una figura geométrica en forma de rombo.

Las estructuras selectivas o alternativas pueden ser:

Simples
Dobles.
Múltiples.

Alternativa simple

La estructura alternativa simple **si_entonces**, ejecuta una determinada acción cuando se cumple una determinada condición. La selección **si-entonces** evalúa la condición:

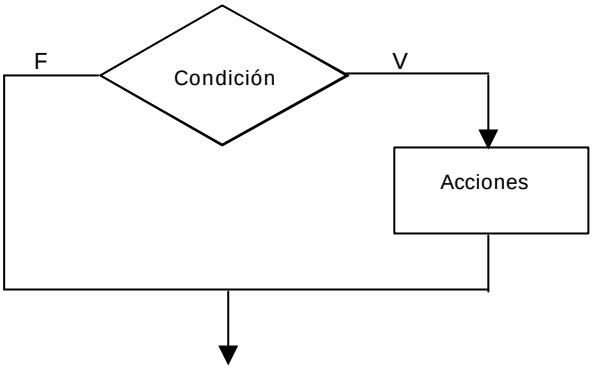
Si la condición es **verdadera,** entonces ejecuta la acción (simple o contar de varias acciones)
Si la condición es **falsa,** entonces no hace nada.

La representación gráfica de la estructura condicional simple se ve a continuación.

La estructura alternativa simple **si_entonces**, ejecuta una determinada acción cuando se cumple una determinada condición. La selección **si-entonces** evalúa la condición:

Si la condición es **verdadera,** entonces ejecuta la acción (simple o contar de varias acciones)
Si la condición es **falsa,** entonces no hace nada.

La representación gráfica de la estructura condicional simple se ve a continuación.



Si condición **Entonces**

Acciones

Fin_Si

Observe que las palabras del pseudocódigo Si y Fin_Si se alinean verticalmente indentando (sangrando) la acción o bloque de acciones.

Afirmemos esto con un ejemplo en Java: Sea generar dos números al azar, y si el segundo es mayor emitir un mensaje informándolo.

```
public class PruebaMay{
      static int numero1 = (int)(10000*Math.random());
      static int numero2 = (int)(10000*Math.random());
      public static void main(String args[]){
            System.out.println("Primer  numero " + numero1);
            System.out.println("Segundo numero " + numero2);
            if(numero2 > numero1) System.out.println("El segundo numero es el mayor");
      }
};
```

Primer proceso

```
Primer  numero 1337
Segundo numero 7231
El segundo numero es el mayor
Process Exit...
```

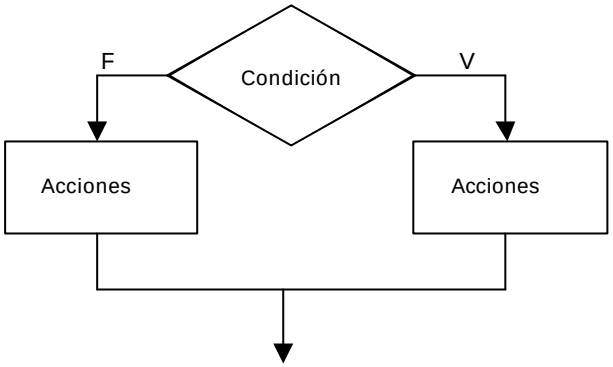
Segundo

```
Primer  numero 5410
Segundo numero 929
Process Exit...
```

Alternativa doble

La estructura anterior es muy limitada y normalmente se necesitará una estructura que permita elegir entre dos opciones o alternativas posibles, en función del cumplimiento o no de una determinada condición.

Si la condición es verdadera, se ejecuta la acciónes_1 y si es falsa, se ejecuta la acciónes_2.



Si Condición **Entonces**
    Acciones
Sino
    Acciones
Fin_Si

Observe que en el pseudocódigo las acciones que dependen de entonces y sino están indentadas en relación con las palabras Si y Fin_Si; este procedimiento aumenta la legibilidad de la estructura y es el medio idóneo para representar algoritmos.

Ejemplo Java: Generar dos números al azar indicando cual de ellos es el mayor.

```
public class PruebaMay{
       static int numero1 = (int)(10000*Math.random());
       static int numero2 = (int)(10000*Math.random());
       public static void main(String args[]){
              System.out.println("Primer  numero " + numero1);
              System.out.println("Segundo numero " + numero2);
              if(numero2 > numero1) System.out.println("El segundo numero es el mayor");
              else System.out.println("El primer numero es el mayor");
       }
};
```

Un par de procesos:

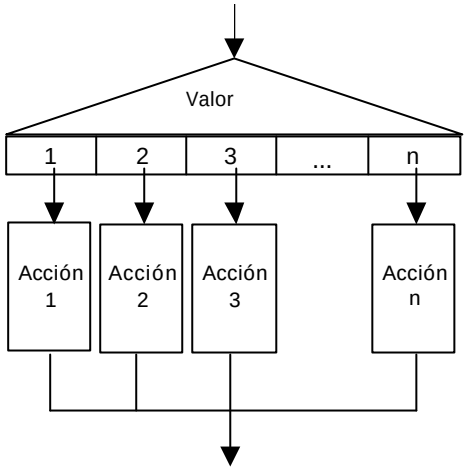
```
Primer  numero 5868
Segundo numero 654
El primer numero es el mayor
Process Exit...

Primer  numero 3965
Segundo numero 7360
El segundo numero es el mayor
Process Exit...
```

Alternativa múltiple

Con frecuencia, es necesario que existan más de dos elecciones posibles. Por ejemplo: en la resolución de la ecuación de segundo grado existen tres posibles alternativas o caminos a seguir, según que el discriminante sea negativo, nulo o positivo. Este problema, se podría resolver por estructuras alternativas simples o dobles, anidadas o en cascada; sin embargo, con este método, si el número de alternativas es grande puede plantear serios problemas de escritura del algoritmo y naturalmente de legibilidad.

La estructura de decisión múltiple evaluará una expresión que podrá tomar n valores distintos: 1, 2, 3,..., n. Según que elija uno de estos valores en la condición, se realizará una de las n acciones, o lo que es igual, el flujo del algoritmo seguirá sin determinado camino entre los n posibles.



Según Valor **Hacer**

Valor 1: Acción 1

Valor 2: Acción 2

Valor 3: Acción 3

...

Valor n: Acción n

Fin Según

En Java, su implementación requiere de las siguientes palabras reservadas: **switch case.. .break. . .default**

```
switch (expresión_entera) {
	case (valor1) : instrucciones_1; [break;]
	case (valor2) : instrucciones_2; [break;]
	...
	case (valorN) : instrucciones_N; [break;]
	default: instrucciones_por_defecto;
}
```

Ejemplo: Expresar **literalmente** el resto de la división entera por 6 de un numero generado al azar.

```
	public class Switch1{
		static int numero = (int)(1000*Math.random());
		static int resto  = numero%6;
		public static void main(String args[]){
			System.out.println("El numero random: " + numero);
			System.out.println("Su resto en la division: " + resto);
			switch (resto) {
				case (0):
					System.out.println("Confirmo, el resto es cero");
					break;
				case (1):
					System.out.println("Confirmo, el resto es uno");
					break;
				case (2):
					System.out.println("Confirmo, el resto es dos");
					break;
				case (3):
					System.out.println("Confirmo, el resto es tres");
					break;
				default:
					System.out.println("El resto es mayor que tres");
					break;
			}
		}
	};
```

Ejecutamos:

```
	El numero random: 326
	Su resto en la division: 2
	Confirmo, el resto es dos
```

```
Process Exit...
```

Si sacamos los “breaks” la alternativa múltiple es también acumulativa, ejemplo:

```
public class Switch2{
	static int numero = (int)(5*Math.random());
	public static void main(String args[]){
		System.out.println("El numero random: " + numero);
		switch (numero){
			case (5):
				System.out.println("Pasando por case (5)");
			case (4):
				System.out.println("Pasando por case (4)");
			case (3):
				System.out.println("Pasando por case (3)");
			case (2):
				System.out.println("Pasando por case (2)");
			default:
				System.out.println("Pasando por default");
		}
	}
};
```

y nos da:

```
El numero random: 4
Pasando por case (4)
Pasando por case (3)
Pasando por case (2)
Pasando por default
Process Exit...
```

Estructuras repetitivas (Iteración)

Las computadoras están especialmente diseñadas para todas aquellas aplicaciones en las cuales una operación o conjunto de ellas deben repetirse muchas veces. Un tipo muy importante de estructura es el algoritmo necesario para repetir una o varias acciones un número determinado de veces. Un programa que lee una lista de números puede repetir la misma secuencia de mensajes el usuario e instrucciones de lectura hasta que todos los números de un archivo se lean.

Las estructuras que repiten una secuencia de instrucciones un número determinado de veces se denominan *bucles* y se denomina *iteración* al hecho de repetir la ejecución de una secuencia de acciones.

Un bucle o lazo (loop) es un segmento de un algoritmo o programa, cuyas instrucciones se repiten un número determinado de veces mientras se cumple una determinada *condición.* Se debe establecer un mecanismo para determinar las tareas repetitivas. Este mecanismo es una condición que puede ser verdadera o falsa y que se comprueba una vez a cada paso o iteración del bucle.

Un bucle consta de tres partes:

- Decisión.
- Cuerpo del bucle.
- Salida del bucle.

Por ejemplo: Si se desea sumar una lista de números escritos desde teclado. El medio conocido hasta ahora es leer los números y añadir sus valores a una variable SUMA que contenga las sucesivas sumas parciales. La variable SUMA se hace igual a cero y a continuación se incrementa en el valor del número cada vez que uno de ellos se lea. El algoritmo que resuelve este problema es:

```
Inicio
   suma = 0
   leer número
   suma = suma + numero
```

```
        leer número
        suma =suma + numero
            ...
    fin
```

y así sucesivamente para cada número de la lista. En otras palabras, el algoritmo repite muchas veces las acciones.

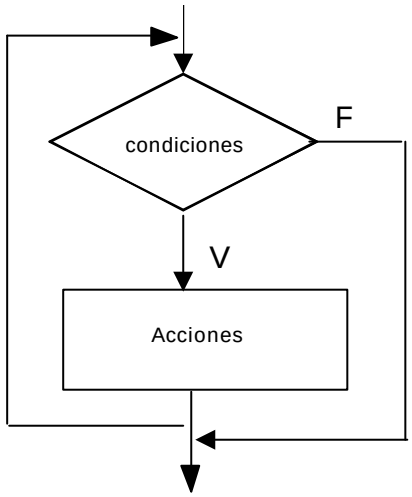
Las dos principales preguntas a realizarse en el diseño de un bucle son: ¿Qué contiene el bucle? y ¿Cuántas veces se debe repetir?

Cuando se utiliza un bucle para sumar una lista de números, se necesita saber cuántos números se han de sumar. Para ello necesitaremos conocer algún medio para detener el bucle. Para detener la ejecución de los bucles se utiliza una condición de parada. Los tres casos generales de estructuras repetitivas dependen de la situación y modo de la condición. La condición se evalúa tan pronto se encuentra en el algoritmo y su resultado producirá los tres tipos de estructuras citadas.

1. La condición de salida del bucle se realiza al principio del bucle (estructura mientras)
2. La condición de salida se origina al Final del bucle; el bucle se ejecuta mientras se verifique una cierta condición.
3. La condición de salida se realiza con un contador que cuenta el número de iteraciones.

Estructura mientras

La estructura repetitiva mientras es aquella en la que el bucle se repite mientras se cumple una determinada condición. La representación gráfica es:



Mientras condición **Hacer**
Acciones
Fin_Mientras

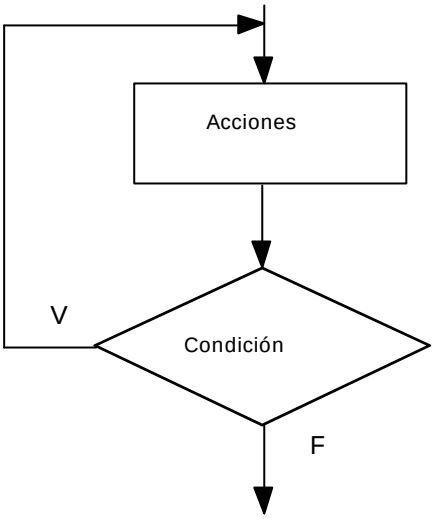
Cuando se ejecuta la instrucción mientras, la primera cosa que sucede es que se evalúa la condición (una expresión booleana). Si se evalúa falsa, ninguna acción se toma y el programa prosigue en la instrucción siguiente al cuerpo del bucle. Si la expresión booleana es *verdadera,* entonces se ejecuta el cuerpo del bucle, después de lo cual se evalúa de nuevo la expresión booleana. Este proceso se repite una y otra vez **mientras** la expresión **(condición)** sea verdadera.

La sintaxis en Java es:

```
while (condición) {
        instrucciones...
}
```

Estructura repetir o hacer mientras

Existen muchas situaciones en las que se desea que un bucle se ejecute al menos una vez antes de comprobar la condición de repetición. En la estructura **mientras** si el valor de la expresión booleana es inicialmente falsa, el cuerpo del bucle no se ejecutará; Por ello se necesitan otros tipos de estructuras repetitivas. La estructura **repetir** o **hacer mientras** se ejecuta mientras se cumpla una condición de permanencia que se comprueba al final del bucle.



Hacer

Acciones

Mientras Condición **Repetir**

Con una estructura **repetir**, el cuerpo del bucle se ejecuta siempre al menos una vez. Cuando una estructura **repetir** se ejecuta, la primera cosa que sucede es la ejecución del bucle y a continuación se evalúa la condición (una expresión booleana). Si se evalúa como verdadera, el cuerpo del bucle se repite y la condición se evalúa una vez. Después de cada iteración del cuerpo del bucle, la condición se evalúa; sí es falsa, el bucle termina y el programa sigue en la siguiente instrucción.

DIFERENCIAS DE LAS ESTRUCTURAS MIENTRAS Y REPETIR

Estructura Mientras	**Estructura Hacer o Repetir**
Es un ciclo de 0 a N, porque si la condición inicial no se cumple no se entra al ciclo, por lo tanto existe la posibilidad de que las aciones del ciclo no sean ejecutadas.	Es un ciclo de 1 a N, porque siempre se ejecuta al menos una vez las instrucciones del ciclo, si la primera vez la condición inicial es falsa, al evaluar la condición al final, se termina el ciclo pero ya se ejecutaron por lo menos una vez las acciones.

La sintaxis en Java es:

```
do {
        instrucciones...
} while (condición);
```

Estructura desde / para (Ciclo for)

En muchas ocasiones se conoce de antemano el número de veces que desean ejecutar las acciones de un bucle. En estos casos, en que el número de iteraciones es fija, se puede usar la estructura **desde** o **para.**

La estructura desde ejecuta las acciones del cuerpo del bucle un número especificado de veces y de modo automático controla el número de iteraciones o pasos a través del cuerpo del bucle, usando una o mas variables de control de ciclo.

La estructura está formada por cuatro partes:

1. Inicialización: Inicializa la o las variables de control.

2. Condición: Condición (expresión booleana) del ciclo, generalmente esta directamente relacionada con la o las variables de control.

3. Incremento: Incrementa la o las variable de control, los valores van evolucionando mientras el ciclo se repita.

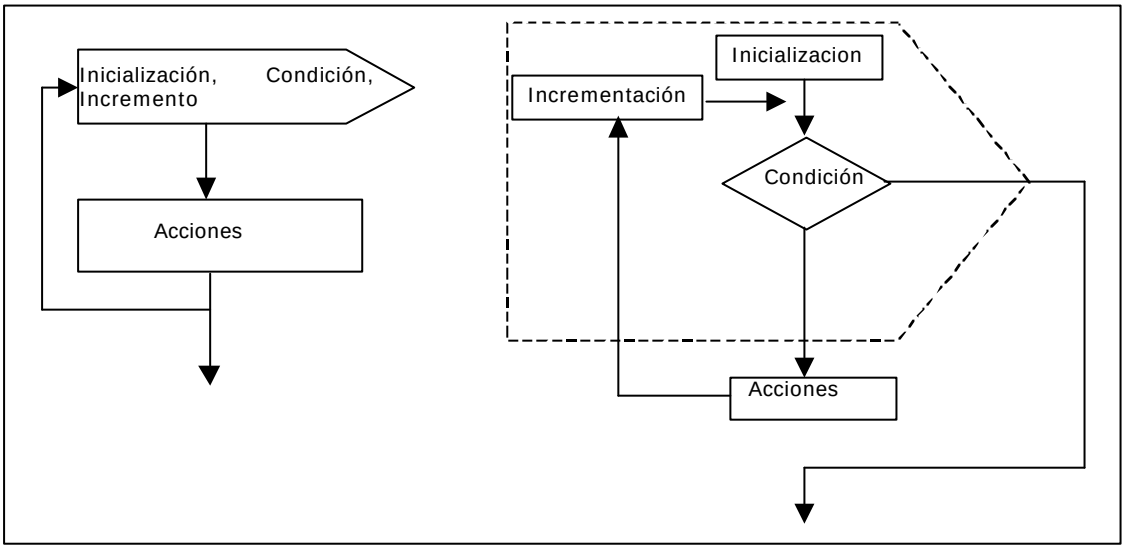
4. Cuerpo: Sentencia simple o compuesta que se repite de acuerdo a si se cumple la condición del ciclo o no.

La ejecución de la sentencia desde sucede de la siguiente forma:

1. Se inicializan la o las variables de control.
2. Se evalua la condición:
    a. Si el resultado es **true** (verdadero), se ejecuta el bloque de sentencias, se evalúa la expresión que da lugar al incremento de la o las variables de control y se vuelve al punto 2.

b. Si el resultado es **false** (falso), la ejecución de la sentencia se da por finalizada y se pasa el control a la siguiente sentencia del programa.

Consideraciones:
* El incremento de la variable de control siempre es 1 si no se indica expresamente lo contrario. Es posible que el incremento sea distinto de uno, positivo o negativo.
* La variable de control normalmente será de tipo entero y es normal emplear como nombre las letras: i, j, k.
* Al **incremento** se le suele denominar también paso.
* Todas las cláusulas (inicialización, condición, incrementación) son opcionales. Puede faltar una cualquiera, pueden faltar todas. Si no hay condición de permanencia significa que no saldremos nunca del ciclo, a menos que dentro del cuerpo existe una sentencia break o return.
* El ciclo for en realidad es un ciclo while ampliado con lugares específicos para inicialización e incrementación. Si queremos reemplazarlo por él, simplemente trasladamos la inicialización inmediatamente antes del ciclo y la incrementación debe ser lo ultimo dentro del cuerpo.
* La condición del ciclo puede ser cualquiera, no es obligatorio que sea una evaluación de la variable incrementada automáticamente.



La sintaxis en Java es:

```
for (inicialización; condición; incrementación) {
        instrucciones...
}
```

Control General del Flujo

Cuando se necesita interrumpir el control del flujo se puede utilizar:

- **break** [etiqueta] : Permite salir de la estructura alternativa múltiple o repetitiva.
- **continue** [etiqueta] : Permite saltar al principio de una ejecución repetitiva.
- **return** expr; : retorna desde una función.

Por ejemplo:

```
import java.io.*;
class bucles {
public static void main(String argv[]) {
        int i=0;
for (i=1; i<5; i++) {
        System.out.println("antes "+ i);
if (i==2) continue;
if (i==3) break;
        System.out.println("después "+i);
```

```
        }
        }
```

La salida es:

```
antes 1
después 1
antes 2
antes 3
```

Otras Instrucciones

Hay otras instrucciones que controlan el flujo del programa:

- **catch**, **throw**, **throws, try** y **finally** (se verá cuando tratemos excepciones)

ENTRADA Y SALIDA ESTANDAR

Métodos de salida.

Java tiene un objeto estático incorporado, llamado System.out que ejecuta la salida al dispositivo de "salida normal". Algunos sistemas operativos, (Unix/Linux), permiten a los usuarios redirigir la salida a archivos, o como entrada a otros programas, pero la salida predeterminada es a la ventana de consola Java. El objeto System.out es una instancia de la clase Java.io.PrintStream. Esta clase define métodos para una corriente de salida a memoria, un **buffer o memoria temporal,** que a continuación se vacía cuando la ventana de consola queda lista para imprimir caracteres. Esta clase también permite los siguientes métodos para salida simple: print(Object o*):* imprimir el objeto o usando su método toString.

Para acceder a este objeto hay que colocar en la primera línea del programa:

```
import java.io.*;
```

que importa el paquete java.io de la biblioteca estándar de Java.

`print(String` *`s):`* `imprimir la string s.`
`print(<base_type>` *`b):`* `imprimir el valor b del tipo base`
`println(String` *`s):`* `imprimir la string s, y cambio de renglón.`
`flush(): imprimir y vaciar el contenido del buffer de impresión.`

Por ejemplo, véase el siguiente fragmento de programa:

```
System.out.print("Valores Java: “);
System.out.print(3.1 41 6);
System.out.print(',');
System.out.print(15);
System.out.println("  (doubIe, char, int).");
```

Cuando se ejecuta, este fragmento origina la siguiente salida:

```
Valores Java: 3.1416,15 (double, char, int)
```

Naturalmente, esta salida aparecerá en la ventana de la consola de Java.

Métodos de entrada.

Así como hay un objeto especial para ejecutar salidas a la consola de Java, Java, también hay otro, llamado System.in, para ejecutar la entrada desde esa ventana. Técnicamente, la entrada proviene en realidad del dispositivo de “entrada Standard”, que por omisión es el teclado de la computadora, y la ventana de la consola reproduce sus caracteres. El objeto System.in es un ejemplo de la clase java.io.InputStream, que por definición avanza carácter a carácter. Esta forma de entrada es, normalmente, demasiado primitiva para ser cómoda, pero también se puede usar para definir otro objeto de entrada que la procese en forma fluida y con buffer, con las clases java.io.BuffereReader y java.io.InputStreamReader.

De todas maneras, la entrada en Java es un tanto laboriosa. Por ese motivo un profesor de la cátedra Paradigmas de Programación (Ing. Gustavo García) desarrolló una clase más "amigable". He aquí su documentación:

La clase In permite que una aplicación pueda leer datos de pantalla fácilmente.

Todos los métodos son estáticos por lo que no es necesario crear una instancia de esta clase para realizar la lectura de datos. Es más, para evitar la **innecesaria instanciación de In** su constructor se ha definido privado. Esto impide tanto la creación de objetos de esta clase como su derivación o extensión.

Para leer cada tipo de dato hay que usar el método correspondiente:

Tipo de dato	**Método**	**Ejemplo**
Entero	readInt()	`int i = In.readInt();`
Entero largo	readLong()	`long l = In.readLOng();`
Real	readFloat()	`float f = In.readFloat();`
Cadena de caracteres (sin espacios)	readString()	`String str = In.readString();`
Linea de caracteres	readLine()	`String line = In.readLine();`

El siguiente es un ejemplo de uso de los métodos de entrada de esta clase. Si ejecutamos el main() que sigue:

```
//  El siguiente es un ejemplo de uso de los metodos de entrada de esta
// clase. Si ejecutamos el main() que sigue:
//  <blockquote><pre>
import In;
class  pruebaIn{
    public static void main(String[] args) {
        long lng;
        int i;
        String str;
        String line;

        System.out.print("Introduzca un entero: ");
        i = In.readInt();
        System.out.print("Introduzca una cadena (sin espacios): ");
        str = In.readString();
        System.out.print("Introduzca un entero largo: ");
        lng = In.readLong();
        System.out.print("Introduzca una linea: ");
        line = In.readLine();
        System.out.println();
        System.out.println("Entero leido: " + i);
        System.out.println("Entero largo leido: " + lng);
        System.out.println("Cadena leida: " + str);
        System.out.println("Linea leida: " + line);
    }
}
```

```
utput | Build | Find in Files
E:\jdk1.3\bin\java.exe   pruebaIn
Working Directory = E:\Tymos\Catedras\AED2005\Unidad I\In\
Class Path =  .;E:\Kawa4.01\kawaclasses.zip;e:\jdk1.3\lib\tools.jar;e:\jdk1.
Introduzca un entero: 123
Introduzca una cadena (sin espacios): sinEspacios
Introduzca un entero largo: 123456789
Introduzca una linea: Esto es una linea

Entero leido: 123
Entero largo leido: 123456789
Cadena leida: sinEspacios
Linea leida: Esto es una linea
Process Exit...
```

La codificación de la clase In puede bajarla de nuestro sitio Labsys. No la pedimos en clase, pero si UD desea conocerla, es la siguiente:

```
//  Inicio código clase In

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

import java.util.StringTokenizer;
import java.util.NoSuchElementException;

public class In {
    /**
     * Objeto utilizado para parsear la linea introducida por el usuario
     * mediante el teclado.
     * @see #getNextToken(String)
     */
    private static StringTokenizer st;
```

```
/**
 * Objeto de I/O (entrada/salida) utilizado para leer los datos desde el
 * teclado.
 * @see #getNextToken(String)
 */
private static BufferedReader source;

/**
 * Constructor privado implementado para evitar instanciacion y
 * derivacion.
 */
private In() {}

/**
 * Metodo utilizado para leer cadenas de caracteres que no contengan
 * espacios. Los siguientes caracteres son considerados espacios:
 * <ul>
 *     <li><code>' '</code>: espacio en blanco.
 *     <li><code>'\t'</code>: tabulacion (tab).
 *     <li><code>'\n'</code>: nueva linea, equivalente al [Enter] (new-line).
 *     <li><code>'\r'</code>: retorno de carro (carriage-return).
 *     <li><code>'\f'</code>: avance de pagina (form-feed).
 * </ul>
 * @return la primer palabra completa que se haya introducido.
 * @see #readLine()
 */
public static String readString() {
    return getNextToken();
}

/**
 * Metodo utilizado para leer cadenas de caracteres que contengan
 * espacios en blanco y/o tabulaciones.
 * @return la linea introducida hasta que se encuentre un [Enter].
 * @see #readString()
 */
public static String readLine() {
    return getNextToken("\r\n\f");
}

/**
 * Metodo utilizado para leer numeros enteros de 32 bits.
 * @return el numero introducido o 0 (cero) si el valor introducido
 * no puede ser interpretado como numero entero. Esto ocurre cuando
 * se introducen letras o signos de puntuacion mezclados con el
 * numero.
 * @see #readLong()
 */
public static int readInt() {
    return (int)readLong();
}

/**
 * Metodo utilizado para leer numeros enteros de 64 bits.
 * @return el numero introducido o 0L (cero) si el valor introducido
 * no puede ser interpretado como numero entero. Esto ocurre cuando
 * se introducen letras o signos de puntuacion mezclados con el
 * numero.
```

```
     * @see #readInt()
     */
    public static long readLong() {
        long retVal = 0;
        try {
            retVal = Long.parseLong(getNextToken());
        } catch (NumberFormatException e) {}
        return retVal;
    }

    /**
     * Metodo utilizado para leer numeros reales de 32 bits de
     * precision.
     * @return el numero introducido o 0.0F (cero) si el valor introducido
     * no puede ser interpretado como numero entero. Esto ocurre cuando
     * se introducen letras o signos de puntuacion mezclados con el
     * numero. Los unicos signos permitidos son el - (menos) al
     * comienzo del numero y una unica aparicion del .
     * (punto decimal).
     * @see #readDouble()
     */
    public static float readFloat() {
        return (float)readDouble();
    }

    /**
     * Metodo utilizado para leer numeros reales de 64 bits de
     * precision.
     * @return el numero introducido o 0.0 (cero) si el valor introducido
     * no puede ser interpretado como numero entero. Esto ocurre cuando
     * se introducen letras o signos de puntuacion mezclados con el
     * numero. Los unicos signos permitidos son el - (menos) al
     * comienzo del numero y una unica aparicion del .
     * (punto decimal).
     * @see #readFloat()
     */
    public static double readDouble() {
        double retVal = 0.0;
        try {
            retVal = Double.valueOf(getNextToken()).doubleValue();
        } catch (NumberFormatException e) {}
        return retVal;
    }

    /**
     * Metodo utilizado para leer un caracter.
     * @return el primer caracter de la cadena introducida o '\0' (caracter nulo)
     * si se introdujo una cadena vacia.
     * @see #readString()
     */
    public static char readChar() {
        char car = '\0';
        String str;

        str = getNextToken("\0");
        if (str.length() > 0) {
            car = str.charAt(0);
            st = new StringTokenizer(str.substring(1));
```

```
            }
            return car;
        }

        /**
         * Metodo utilizado para obtener la siguiente palabra (o numero)
         * del objeto parseador de la entrada.
         * @return siguiente elemento del parseador de cadenas, considerando como
         * separadores a los caracteres de espacio (ver documentacion de
         * {@link #readString() readString()}).
         * @see #getNextToken(String)
         * @see #st
         * @see "Documentacion de la clase <code>StringTokenizer</code> en el
         * sitio oficial de Java: <a href="http://java.sun.com">http://java.sun.com</a>."
         */
        private static String getNextToken() {
            return getNextToken(null);
        }

        /**
         * Metodo utilizado para obtener el siguiente elemento del objeto
         * parseador de la entrada. Los elementos estaran definidos por el
         * delimitador que se recibe como parametro.
         * @param delim delimitador a utilizar durante el parseo de la entrada. Si
         * el parametro es <code>null</code> se tomarán los delimitadores
         * indicados en {@link #readString() readString()}.
         * @return siguiente elemento del parseador de cadenas, considerando como
         * separadores al par&aacute;metro recibido.
         * @see #getNextToken()
         * @see #st
         * @see "Documentaci&oacute;n de la clase <code>StringTokenizer</code> en el
         * sitio oficial de Java: <a href="http://java.sun.com">http://java.sun.com</a>."
         */
        private static String getNextToken(String delim) {
            String input;
            String retVal = "";

            try {
                if ((st == null) || !st.hasMoreElements()) {
                    if (source == null) {
                        source = new BufferedReader(new InputStreamReader(System.in));
                    }
                    input = source.readLine();
                    st = new StringTokenizer(input);
                }
                if (delim == null) {
                    delim = " \t\n\r\f";
                }
                retVal = st.nextToken(delim);
            } catch (NoSuchElementException e1) {
                // si ocurre una excepción, no hacer nada
            } catch (IOException e2) {
                // si ocurre una excepción, no hacer nada
            }

            return retVal;
        }
    }
```

```
//     Fin código clase In
```

LECTURA COMPLEMENTARIA

EL CUERPO DE LOS METODOS - ALGORITMOS

1. CONCEPTO

El objetivo fundamental de esta materia es enseñar a resolver problemas mediante una computadora. Un programador de computadora es antes que nada una persona que resuelve problemas, por lo que para llegar a ser un programador eficaz se necesita aprender a resolver problemas de un modo riguroso y sistemático.

La resolución de un problema exige el diseño de los objetos y de los algoritmos que resuelven el comportamiento de los mismos.

Ya vimos como resolver el diseño de los objetos, ahora veremos como resolver los métodos de los mismos, los pasos para la resolución de un problema son:

- Diseño del algoritmo que describe la secuencia ordenada de pasos, sin ambigüedades, que conducen a la solución de un problema dado. (Análisis del problema y desarrollo del algoritmo)
- Escritura del algoritmo en un lenguaje de programación adecuado.
- Ejecución y validación del programa por la computadora

Para llegar a la realización de un programa es necesario el diseño previo de un algoritmo, de modo que sin algoritmo no puede existir un programa.

Los algoritmos son independientes tanto del lenguaje de programación en que se expresan como de la computadora que los ejecuta. En cada problema el algoritmo se puede expresar en un lenguaje diferente de programación y, ejecutarse en una computadora distinta; sin embargo, el algoritmo será siempre el mismo. Así por ejemplo, en una analogía con la vida diaria, una receta de un plato de cocina se puede expresar en español, ingles o francés, pero cualquiera que sea el lenguaje, los pasos para la elaboración del plato se realizarán sin importar el cocinero.

En la ciencia de la computación y en la programación los algoritmos son más importantes que los lenguajes de programación o las computadoras. Un lenguaje de programación es tan sólo un medio para expresar un algoritmo y una computadora es sólo un procesador para ejecutarlo. Tanto el lenguaje de programación como la computadora son los medios para obtener un fin: conseguir que el algoritmo se ejecute y se efectúe el proceso correspondiente.

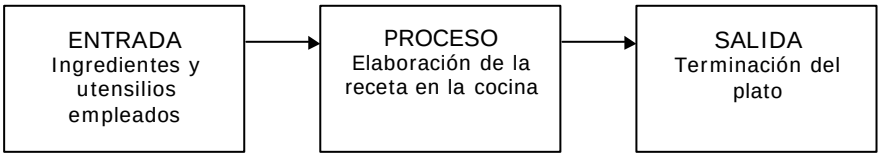
Dada la importancia del algoritmo en la ciencia de la computación, un aspecto muy importante será el diseño de algoritmo,

El diseño de la mayoría de los algoritmos requiere creatividad y conocimientos profundos de la técnica de la programación. En esencia, todo problema se puede describir por medio de un algoritmo.

Las características fundamentales que debe cumplir todo algoritmo son:

- Un algoritmo debe ser preciso e indicar el orden de realización de cada paso.
- Un algoritmo debe estar definido. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
- Un algoritmo debe ser finito. Si se sigue un algoritmo, se bebe terminar en algún momento; o sea debe tener un número finito de pasos.

La definición de un algoritmo debe describir tres partes: Entrada, Proceso y Salida. En el algoritmo de receta de cocina citada anteriormente se tendrá:



2. REPRESENTACION GRAFICA DE LOS ALGORITMOS

Para representar un algoritmo se debe utilizar algún método que permita independizar dicho algoritmo del lenguaje de programación elegido. Ello permita que un algoritmo pueda ser codificado indistintamente en cualquier lenguaje. Para conseguir este objetivo se precisa que el algoritmo sea representado gráfica o numéricamente, de modo que las sucesivas acciones no dependan de la sintaxis de ningún lenguaje de programación, sino que la descripción pueda servir fácilmente para la transformación en un programa, es decir, su codificación.

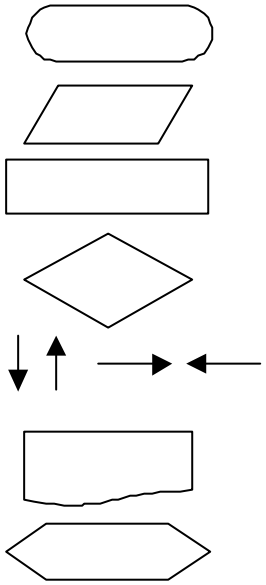
Los métodos para representar un algoritmo son:

Diagrama de flujo.
Pseudocódigo.
Lenguaje español.
Fórmulas.

DIAGRAMA DE FLUJO

Un diagrama de flujo es una de las técnicas de representación de algoritmo más antiguo y a la vez más utilizado. Un diagrama de flujo es un diagrama que utiliza los símbolos (cajas) estándar y que los pasos del algoritmo son escritos en el interior, donde son unidos mediante de flechas, denominadas líneas de flujo, que indican la secuencia en que se deben ejecutar.

Símbolos principales:

SIMBOLOS PRINCIPALES	FUNCION
	**Terminal** representa el comienzo "inicio' y el final "fin", de un programa. **Entrada/Salida**, cualquier tipo de introducción de datos en la memoria desde los periféricos. **Proceso**, cualquier tipo de operación que pueda originar cambio de valor, operaciones aritméticas, transferencia. **Decisión**, indica operaciones lógicas o de comparación entre datos y en función del resultado de la misma determina cuál de los distintos caminos alternativos del programa se debe seguir; normalmente tiene dos salidas. **Indicador de dirección o línea de flujo**, indica el sentido de ejecución de las operaciones. **Impresora**, se utiliza en ocasiones **en** lugar del símbolo de E/S **Subrutina**, indica comienzo de una subrutina o función

Los símbolos estándar normalizados son muy variados, sin embargo, los símbolos más utilizados representan:

Proceso
Decisión
Fin
Entrada / Salida
Dirección del flujo.

En un diagrama de flujo:

Existe una caja etiquetada "inicio", que es de tipo elíptico.
Existe otra caja etiquetada "fin" de igual forma que en a.
Si existen otras cajas, normalmente son rectangulares, tipo rombo o paralelogramo

Se puede escribir más de un paso del algoritmo en un sola rectangular. El uso de flechas significa que la caja no necesita ser escrita debajo de su predecesora. Sin embargo, abusar demasiado de esta flexibilidad conduce a diagramas de flujo complicados e ininteligibles.

PSEUDOCODIGO

El pseudocódigo es un lenguaje de especificación de algoritmos. El uso de tal lenguaje hace el paso de codificación final relativamente fácil.

El pseudocódigo nació como un lenguaje, similar al ingles y era un medio de representar básicamente las estructuras de control de programación estructurado. Se considera un primer borrador, dado que el pseudocódigo tiene que traducirse posteriormente a un lenguaje de programación. El pseudocódigo no puede ser ejecutado por una computadora. La ventaja del pseudocódigo es que en su uso en la planificación de un

programa, el programador se puede concentrar en la lógica y en las estructuras de control y no preocuparse de las reglas de un lenguaje especifico. Es también fácil modificar el pseudocódigo si se descubren errores o anomalías en la lógica del programa, mientras que en muchas ocasiones suele ser difícil el cambio en la lógica una vez que está codificado en un lenguaje de programación.

El pseudocódigo utiliza para representar las acciones sucesivas palabras reservadas en inglés, similares a sus homónimas en los lenguajes de programación, tales como **end, if-then-else, while, etc.** La escritura del pseudocódigo exige normalmente la indentación de diferentes líneas.

3. ESCRITURA DEL ALGORITMO

La escritura de un algoritmo mediante una herramienta de programación debe ser lo más clara posible y estructura de modo que su lectura facilite considerablemente el entendimiento del algoritmo y su posterior codificación en un lenguaje de programación.

Los algoritmos deben ser escritos en lenguajes similares a los programas. Por ahora, utilizaremos el lenguaje algoritmo basado en pseudocódigo.

Un algoritmo constará de dos componentes: una cabecera y un cuerpo.

La cabecera es una acción simple que comienza con las acciones de declaración que definen o declaran las variables y constantes que se utilizarán en la rutina.

El cuerpo es un bloque algoritmo que consta de las acciones ejecutables donde las acciones ejecutables son las acciones que posteriormente deberá realizar la computadora cuando el algoritmo sea convertido en programa ejecutable.

PROGRAMAS

1. CONCEPTO DE PROGRAMA

Un programa de computadora es un conjunto de instrucciones, ordenes dadas a la maquina, que producirán la ejecución de una determinada tarea.

2. PROGRAMAS TIPOS

Existen programas tipos que se encuentran en todas las aplicaciones y consisten en programas que responden siempre al mismo comportamiento, ellos son:

Altas

Realizan las operaciones de ingreso de datos, como alta de productos, clientes, proveedores, etc. Generalmente estas operaciones consisten en una carga de datos, verificación de esa carga y grabación de la misma en un archivo de datos o en una estructura en memoria.

Modificaciones

Realizan las operaciones de modificaciones de datos cargados en un archivo de datos o en una estructura en memoria, por ejemplo un archivo de productos, clientes, proveedores, etc. Generalmente estas operaciones consisten en una búsqueda del elemento a modificar, carga de las modificaciones, verificación de esa carga y grabación de las mismas en el archivo o estructura correspondiente.

Bajas

Realizan las operaciones de eliminación de datos cargados en un archivo de datos o en una estructura en memoria, por ejemplo un archivo de productos, clientes, proveedores, etc. Generalmente estas operaciones consisten en una búsqueda del elemento a eliminar, verificación del elemento y eliminación o borrado del mismo en el archivo correspondiente.

Consultas

Realizan las operaciones de consulta de datos cargados en un archivo de datos o en una estructura en memoria. Dichas consultas se pueden mostrar en pantalla, impresora o archivo, y pueden ser individuales o grupales.

Individuales: Realizan la consulta de un elemento en particular. Generalmente estas operaciones consisten en una búsqueda del elemento a consultar y muestra de los todos los datos del mismo o de los que solicite el usuario.

Grupales: Realizan la consulta de todos los elementos existentes en un archivo de datos o en una estructura en memoria. Generalmente estas operaciones consisten en la elección del archivo de datos a consultar y muestra de los todos los datos del mismo o de los que solicite el usuario.

Listados

Realizan las mismas operaciones que las consultas pero los resultados sólo se muestran en impresora.

Movimientos o transacciones

Realizan las operaciones de carga o modificación de datos de movimientos regulares, tales como, planillas de compras, ventas, asistencias, etc.

Carga: Implica la grabación de ese movimiento en un archivo de datos o en una estructura en memoria. Generalmente estas operaciones consisten en una carga de datos comunes al movimiento (número, fecha, etc.), carga del detalle del movimiento (ítem, cantidad, precio, etc.), verificación de esa carga y grabación de la misma en un archivo de datos o en una estructura en memoria.

Modificación: Implica la modificación de ese movimiento en un archivo de datos o en una estructura en memoria. Generalmente estas operaciones consisten en la búsqueda del movimiento a modificar, la verificación de esa modificación y la grabación de la misma en un archivo de datos o en una estructura en memoria. Las modificaciones de los movimientos o transacciones no siempre se realiza, depende del movimiento de que se trate.

Mantenimiento

Son programas auxiliares o adicionales que se agregan al programa principal o aplicación, sirve para realizar aquellas operaciones que el programador considere de importancia para completar la aplicación, tales como, programas de seguridad, de resguardo (Backup), etc.

3. ELEMENTOS BASICOS DE UN PROGRAMA

En programación se debe separar la diferencia entre el diseño del algoritmo y su implementación en un lenguaje específico. Por ello se debe distinguir claramente entre los conceptos de programación y el medio en que ellos se implementan en un lenguaje específico. Sin embargo, una vez que se comprendan los conceptos de programación y cómo utilizarlos, la enseñanza de un nuevo lenguaje es relativamente fácil.

Los lenguajes de programación tienen elementos básicos que se utilizan como bloques constructivos, así como reglas para las que esos elementos se combinan. Estas reglas se denominan *sintaxis* del lenguaje. Solamente las instrucciones sintácticamente correctas pueden ser interpretadas por la computadora y los programas que contengan errores de sintaxis son rechazados por la máquina.

Nosotros ya vimos que el elemento o unidad básica de un programa orientado a objetos es la clase, pero también existen ciertos elementos que se encuentran en todo programa y que veremos brevemente a continuación:

Palabras reservadas.
Comentarios.
Tipos de datos
Variables.
Operadores
Expresiones
Instrucciones.
Funciones.

Además de estos elementos básicos existen otros elementos que forman parte de los programas, cuya comprensión y funcionamiento será vital para el correcto diseño de algoritmo y naturalmente la codificación del programa.

Contadores.
Acumuladores.
Banderas.

El amplio conocimiento de todos los elementos de programación y el modo de su integración en los programas constituyen las técnicas de programación que todo buen programador debe conocer.

1. PALABRAS RESERVADAS: Son palabras propias ó nativas de cada lenguaje que el programador debe utilizar según las especificaciones del mismo.

2. COMENTARIOS: Forman parte de la documentación interna de un programa, se encuentran acompañando al código fuente en los lugares que se necesite un comentarios significativo del mismo.

3. TIPOS DE DATOS: Es una descripción del conjunto de valores que puede tomar un dato.

Los tipos de datos que se utiliza en los programas, en general, son los siguientes:

DATOS CARÁCTER: El tipo carácter es una sucesión de caracteres que se encuentran delimitados por una comilla o dobles comillas, según el tipo de lenguaje de programación.

DATOS NUMERICOS: El tipo numérico es el conjunto de los valores numéricos. Se representan como enteros y reales.

DATOS LOGICOS (BOOLEANOS) El tipo lógico, también denominado booleano, es aquel dato que sólo puede tomar uno de dos valores: cierto o verdadero(true) y falso (false)

4. VARIABLES: Una variable es una unidad de almacenamiento identificada por un nombre y un tipo de dato, tiene una dirección (ubicación) dentro de la memoria y puede almacenar información según el tipo de dato que tenga asociado. Es necesario definir las variables de un programa antes de que puedan ser utilizadas.

5. OPERADORES: Símbolos de operación que se utilizan para procesar, comparar o relacionar distintos valores.

OPERADORES ARITMÉTICOS: +, -, *, /
OPERADORES LÓGICOS: and, or.
OPERADORES RELACIONALES: >, <, >=, =<, =, !=
OPERADORES DE ASIGNACIÓN: =

6. EXPRESIONES: Es la combinación de operadores y operandos. Una expresión es evaluada por el lenguaje dando lugar principalmente a dos tipos de resultados, basándose en los cuales podemos dividir las expresiones en aritméticas y condicionales. Las primeras se caracterizan por generar un resultado numérico, mientras que las segundas sólo pueden devolver como resultado los valores verdadero (true) y falso (false).

Las expresiones son combinaciones de constantes, variables, símbolos de operación, paréntesis y nombre de funciones especiales. Las mismas ideas son utilizadas en notación matemática tradicional; por ejemplo:

$a + b*(b + 3) + (a / 10)$

Cada expresión toma un valor que se determina tomando los valores de las variables y constantes implicadas y la ejecución de las operaciones indicadas.

7. INSTRUCCIONES: Son las acciones que se ejecutan en un computador, generalmente hay instrucciones de inicio y fin, de asignación, de lectura, de escritura y de salto o bifurcación.

8. FUNCIONES : Una función, llamada también subrutina, es un conjunto de instrucciones o bloque de código independiente. En general poseen un nombre y un cuerpo que es donde se escriben las instrucciones. Estas funciones son lo que conforman los métodos de una clase.

9. CONTADORES: Es una variable que a partir de un valor inicial, aumenta o disminuye en una cantidad constante.

forma general
contador = valor inicial
contador = contador + paso *paso* es el valor constante que se suma o resta en cada repetición

ejemplo		
	c = 0	c=120
	c = c + 1	c = c + 10

Los procesos repetitivos son la base del uso de las computadoras. En estos procesos se necesitan normalmente contar los sucesos o acciones internas del bucle, como pueden ser los elementos de un fichero, el numero de iteraciones a realizar por el bucle, etc. Una forma de controlar un bucle es mediante un contador.

10. ACUMULADORES: Es una variable que a partir de un valor inicial, aumenta o disminuye en una cantidad no constante. Se utilizan para sumar o restar variables afines.

forma general
contador = valor inicial
contador = contador + cantidad *cantidad* se suma o resta en cada repetición

ejemplo		
	c = 0	cant=120
	c = c + cant	

Un acumulador es una variable cuya misión es almacenar cantidades variables resultantes de sumas sucesivas. Realiza la misma función que un contador con la diferencia de que el incremento o decremento de cada suma es variable en lugar de constante como en el caso del contador.

11. BANDERAS: Es una variable que puede tomar diversos valores a lo largo de la ejecución del programa y que permite comunicar información de una parte a otra del mismo. Por lo general toman dos valores (ej. 0 y 1) y se utilizan para saber si se ha ejecutado o no una parte del proceso. De acuerdo al valor que tenga en el momento en que se le testea se puede determinar si se ejecuta o no esa parte del proceso.

CLASE PRACTICA

Para la clase práctica se sugiere ejercicios que identifiquen clases y que grafiquen el cuerpo de los métodos, sin usar código todavía o alguno muy simple.

1.PROBLEMA

Dado el valor de los tres lados de un triángulo, calcular el perímetro.

ANALISIS

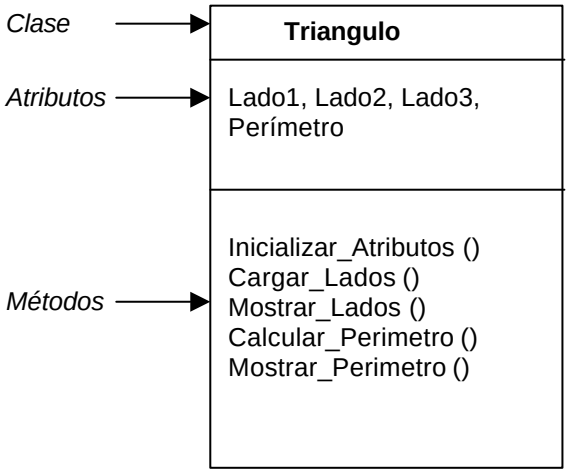
1. Identificar el objeto: identificar la entidad que engloba al problema, en nuestro ejemplo es el triángulo.

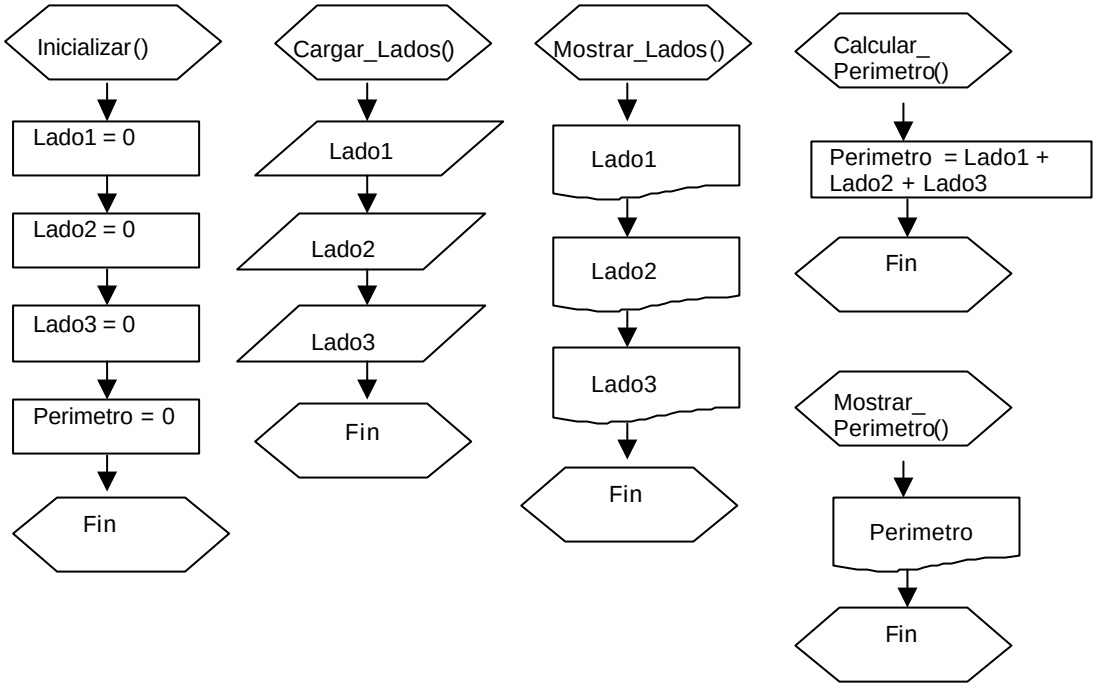
2. Identificar sus atributos: podemos decir que todo triángulo tiene como datos esenciales los tres lados, además, nuestro problema en particular necesitará el valor del perímetro, que es un datos calculable a partir de los tres lados y que puede figurar como atributo o no dependiendo si se quiere guardar su estado en el objeto. Como nuestro enunciado no nos restringe la forma de implementarlo vamos a definir como atributos los tres lados y el perímetro.

3. Identificar sus métodos: los métodos a elegir siempre dependerán de las operaciones que podemos realizar con los atributos, entonces podemos enumerar:

Inicializar los atributos: Inicializar en 0, por ejemplo, los lados y el perímetro.
Ingresar los lados: Se ingresa el valor de cada lado.
Mostrar los lados: Mostrar el valor de los lados.
Calcular el perímetro: formula que devuelve el perímetro del triángulo.
Mostrar el perímetro: Mostrar el valor del perímetro.

Graficando el resultado



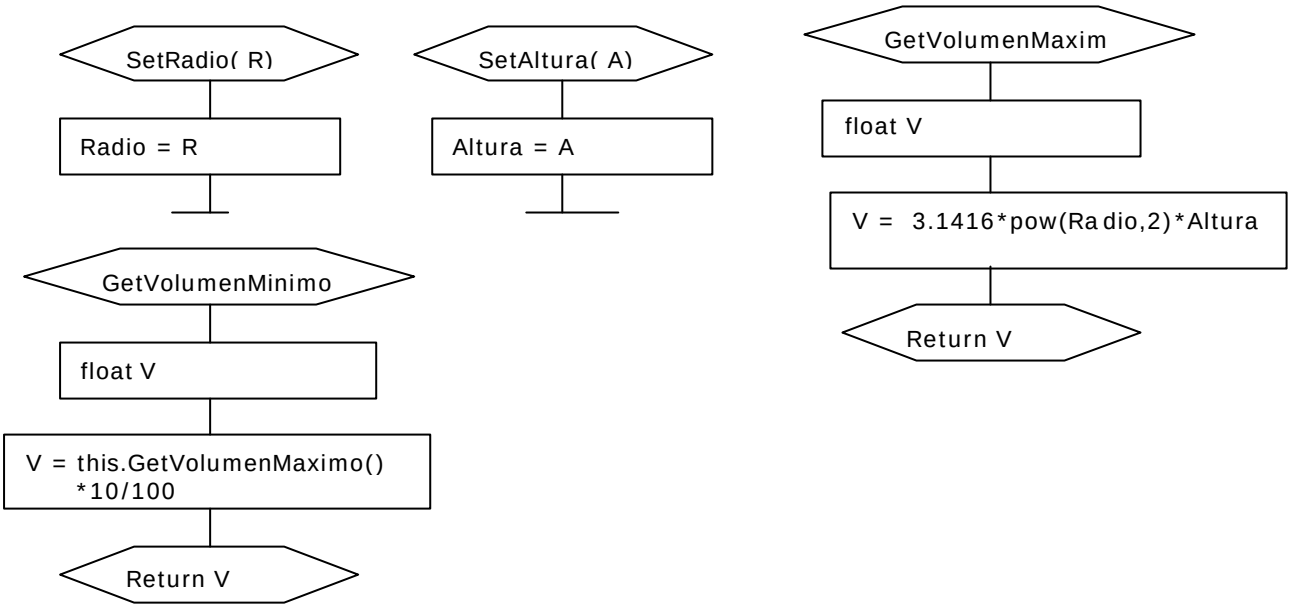


2. El dueño de un campo que almacena sus cosechas de granos en silos cilíndricos le pide que desarrolle un programa que le permita conocer el volumen máximo de un silo a partir de su radio y su altura.
El programa debe ser capaz también de informar el volumen mínimo sabiendo que este se define como el 10% del volumen máximo.
En un cilindro, su volumen interno se calcula como el número pi por el radio al cuadrado por la altura.

Solución.
Diagrama de clase:

Clase Silo
Radio Altura
Silo() SetRadio() SetAltura() GetVolumenMaximo() GetVolumenMinimo()

Diagrama de flujo de los métodos


SetRadio(R)
Radio = R
SetAltura(A)
Altura = A
GetVolumenMaxim
float V
V = 3.1416*pow(Radio,2)*Altura
Return V
GetVolumenMinimo
float V
V = this.GetVolumenMaximo()
*10/100
Return V