

Capítulo I Fundamentos del Lenguaje

Antes de iniciar este material es importante recordar que para cualquier desarrollo de software se requiere de cierta disciplina y conocimiento acerca del proceso y el ciclo de desarrollo de software, es recomendable que el alumno tenga conocimientos muy precisos sobre: Técnicas de diseño, técnicas para representar algoritmos, terminología orientada a objetos y sobre todo la suficiente creatividad para la resolución de problemas aplicando programas de computadora.

Pues bien empecemos, este material esta pensado para alumnos de cualquier carrera profesional que tenga que ver con la ciencia y el arte de programar computadoras, utilizando el paradigma orientado a objetos.

1.1 Introducción al lenguaje y a su entorno de desarrollo.

Antes que nada se debe conocer el lenguaje de programación a utilizar, así como el entorno que requerimos para poder hacer uso de el.

El lenguaje con el que se trabajan las explicaciones y ejercicios a lo largo de este material es Java.

Java es el lenguaje de programación que en la última decada a dado mucho de que hablar, a veces de manera positiva y otras pocas de forma negativa, pero es un lenguaje que esta en la boca de casi toda la gente que esta dentro de la industria de las tecnologías de información. Su historia se remonta a mediados de la decada de los 90's; todo inicia en enero de 1991 con el proyecto Green, en dicho proyecto se planteaba la idea de controlar dispositivos electrodomésticos por medio de programas de computadora, para la cual James Gosling, penso que C ni C++ eran los lenguajes apropiados, para esto, creo un nuevo lenguaje llamado Oak, Sun Microsystem tenía un nuevo producto pero que no habría de prosperar en los siguientes años; en 1994 buscando una utilidad para Oak se desarrolla un entorno para la Web llamado WebRunner, que no era otra cosa sino un browser de Internet con capacidad de ejecutar aplicaciones y no solo mostrar páginas escritas en HTML. El año de 1995 se le conoce como el año en que Java fue presentado ya como un lenguaje de programación; tomando el proyecto WebRunner se decide cambiar el nombre al proyecto lo cual genera lo que hoy se conoce como el lenguaje de programación Java; en principio Netscape y Sun Microsystem unen esfuerzos para hacer que Java se convierta en un estandar en la industria de las aplicaciones para la Web, en poco tiempo compañias como IBM, Oracle, Borland, Adobe Macromedia, Lotus, Spyglass e Intuit incorporan Java a sus productos, incluso Microsoft incorpora en su browser Internet Explorer a Java para la compatibilidad con miles de aplicaciones que comenzaron a hacer su aparición en Internet.

Fundamentos de programación orientada a objetos con Java

Hoy en día Java es uno de los lenguajes más populares, ya no solo es el lenguaje y sus aplicaciones para Internet, Java es una tecnología y plataforma de desarrollo que podemos encontrar en muchos de los sistemas computacionales, que van desde simples computadoras hasta sistemas empotrados, como son el horno de microondas, la lavadora, el refrigerador, el televisor, etc.

Dentro de las características de Java podemos mencionar las siguientes: lenguaje sencillo de aprender y aplicar, orientado a objetos, interpretado, robusto, seguro, de arquitectura multiplataforma, distribuido, portatil, multihebrado y dinámico. Cada una de estas características el intentar describirlas tendríamos que escribir todo un libro sobre el tema, así que espero poder mostar como algunas de las características mencionadas se pueden observar a lo largo de este material.

Sun Microsystem ofrece diversas ediciones de Java por lo cual debemos estar bien enterados, para saber cual es la edición que necesitamos para cada cosa que querramos hacer con este lenguaje de programación.

En primer lugar hablaremos de JRE(Java Runtime Environment), este es el ambiente en tiempo de ejecución que las aplicaciones Java requieren, este ambiente muchas veces se encuentra instalado en las computadoras, ya que existen algunos programas Java que lo requieren; con este ambiente no se pueden hacer desarrollos de aplicaciones en Java, por lo cual hay que tener mucho cuidado en pensar que bajando JRE del sitio oficial de Java ya se puede empezar a desarrollar aplicaciones, este ambiente Sun lo ofrece de manera gratuita así como las herramientas de desarrollo.

Las herramientas de desarrollo se les conoce como JDK (Java Development Kit) y estan clasificadas en tres categorias: El J2SE (Java 2 Standard Edition) que incluye todas las herramientas y bibliotecas estandar de Java para crear aplicaciones y applets. El J2EE (Java 2 Enterprise Edition) que incluye las bibliotecas y herramientas para crear Servlets, aplicaciones distriuidas, aplicaciones cliente/servidos multicapas, etc. El J2ME (Java 2 Micro Edition) que permite desarrollar midlets, aplicaciones en sistemas empotrados, aplicaciones inalambricas, etc.

En los tres JDK se utiliza el mismo lenguaje, las bibliotecas estandar con diferentes herramientas y bibliotecas específicas para cada una de los tipos de aplicaciones a desarrollar. El JDK que se utilizará a lo largo del material será el J2SE, este se ofrece por parte de Sun en diversas versiones que se han construido a lo largo de los años en que ha venido evolucionando Java, el material no hace referencia a alguna versión en específico, pero se sugiere consultar la página oficial de Java (java.sun.com) para conocer cual es la versión más reciente.

Desde la aparición de Java 2, las herramientas y utilerias son muy parecidas; un entorno de desarrollo de Java debe contener un conjunto de herramientas; más la configuración del ambiente para poder desarrollar y probar las aplicaciones que se elaboren. El JDK se puede obtener desde el sitio oficial de Sun mencionado anteriormente, existen versiones para diferentes plataformas como son Windows, Solaris, Linux, Macintosh, etc. Dependiendo de la plataforma es como se instala, pero una vez instalada se debe verificar que el ambiente esta correctamente configurado para poder usar el JDK.

Para el J2SE, la carpeta donde se haya decidido instalarlo, estará compuesta de la siguiente forma:

c:\j2sdk1.5.0\

bin\
include\
demo\
jre\
lib\

En la carpeta *bin* se localizan todas las utilerias para desarrollar aplicaciones y applets en Java así como herramientas para ejecutar aplicaciones y applets.

En la carpeta *include* se localizan archivos de cabecera para ser utilizados con sistemas heredados de C y C++.

En la carpeta *demo* se localizan ejemplos de demostración.

En la carpeta *jre* se localiza el ambiente para tiempo de ejecución de Java

En la carpeta *lib* se localizan las bibliotecas estandar de Java 2 standard edition

Es recomendable en el sistema verificar que la variable de entorno del sistema operativo PATH tenga descrita la ruta a la carpeta *bin* de Java.

Para el caso de Windows 95, 98 y ME, editar el archivo AUTOEXEC.BAT que se localiza en la raiz del sistema y agregar al final del archivo la siguiente linea.

SET PATH=c:\j2sdk1.5.0\bin;%PATH%

Para el caso de Windows NT, 2000 y XP, desde el panel de control activar la

Fundamentos de programación orientada a objetos con Java

configuración del sistema, en avanzados ver variables de entorno y modificar la variable PATH agregando al final el valor correspondiente. C:\j2sdk1.5.0\bin

Recordar que <C:\j2sdk1.5.0> es el nombre de la carpeta que se utilizó para instalar el JDK.

Otro de los elementos con los que se debe tener cuidado, es la variable de ambiente CLASSPATH, si es la primera vez que se instala una versión de Java no hay problema con dejar sin efecto esta variable, pero cuando estamos combinando versiones de Java diferentes y también estamos desarrollando una aplicación con varios archivos en diferentes carpetas, esta variable va a determinar donde buscará las clases de que estará compuesta nuestra aplicación; se agrega como una variable de entorno en el sistema y se le asignan las rutas donde se localizan las clases de una aplicación.

El proceso de implementar un programa de aplicación está regido por los siguientes pasos:

- a) Editar el programa fuente. Un programa fuente es un código escrito en un lenguaje de programación, al cual se le conoce como lenguaje anfitrión y debe cuidarse la sintaxis que determina el lenguaje. Java tiene un conjunto de reglas de sintaxis que permiten escribir de forma correcta programas en Java. Es muy importante hacer notar que no siempre un programa correctamente escrito funciona correctamente.
- b) Compilar el programa fuente. Este proceso toma un programa fuente y verifica que este correctamente escrito para posteriormente generar un programa escrito en un lenguaje que entiende la computadora que lo va a ejecutar. Para el caso de Java, el lenguaje del programa generado por el compilador se le conoce como bytecode, este se debe ejecutar en una computadora virtual, esto hace que los programas en Java sean transportados o portátiles, lo que quiere decir que al compilar y generar el bytecode correspondiente se puede ejecutar en cualquier plataforma (Windows, Linux, Solaris, etc.) siempre y cuando se tenga instalada la máquina virtual de Java (JVM), dentro de JRE se localiza una máquina virtual de Java.
- c) Ejecutar el programa generado por el compilador. En java, para ejecutar el programa, se requiere de una máquina virtual, la cual se encargará de llevar a cabo este proceso.

Existen algunas herramientas de terceros que nos ofrecen ambientes integrados de desarrollo (IDE's) que se integran con el JDK para facilitarnos la labor de desarrollo, algunos de estos entornos son: JCreator, JBuilder, JDeveloper, Visual Age, Sun Java Studio entre otros. Estos IDE's ofrecen editores de código,

Fundamentos de programación orientada a objetos con Java

directamente se enlazan al compilador y a la máquina virtual de Java para llevar a cabo el proceso, pero de igual forma se puede llevar a cabo el proceso utilizando solo las herramientas que nos ofrece el JDK y un editor de textos que nos ofrezca el sistema operativo.

Ahora sí, ya podemos empezar a trabajar haciendo un ejemplo:

La primera sugerencia, si es que se trabaja con algún ambiente de desarrollo es que estemos bien enterados que ha sido correctamente configurado con el JDK que deseamos utilizar; por el contrario si no trabajamos con alguno de estos ambientes estaremos bien preparados conociendo a la perfección el funcionamiento del sistema operativo con el cual vamos a trabajar.

Los ejemplos tratados en el material asumen que no se utiliza alguna herramienta en particular y solo el J2SE.

Editemos pues el primer programa en Java utilizando un editor de textos como el block de notas de Windows, que no ha sufrido muchos cambios desde sus orígenes.

```
import java.lang.*;
import java.util.*;

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

Salvar el programa en una carpeta preferentemente nueva que hayamos creado para guardar este primer programa, por ejemplo la carpeta programa1. Y debemos dar al nombre del archivo el nombre de la clase y con extensión java: Hola.java

Para compilar el programa debemos abrir una sesión de MS-DOS o Ventana de línea de comandos. Una vez en la línea de comandos cambiarse al directorio donde guardamos el archivo Hola.java

```
c:\programa1> javac Hola.java
```

Al terminar de compilar podemos verificar que en la carpeta aparece un nuevo archivo llamado Hola.class, este es el bytecode que generó el compilador, el cual puede ser ejecutado por la máquina virtual de Java. Para poder ejecutarlo se realiza lo siguiente:

c:\programa1> java Hola

El resultado que se muestra de la ejecución del programa, es el siguiente:

Hola Mundo

c:\programa1>

Con este primer ejemplo podemos analizar los elementos esenciales de un programa en Java. En primer lugar tenemos que decir que un programa al menos debe estar compuesto de una clase, la cual deberá tener el mismo nombre que el archivo donde se guardara el código fuente y la clase deberá ser publica para lo cual se utilizan las palabras reservadas class y public respectivamente.

Un programa en Java podrá llevar una serie de importaciones que indicarán las bibliotecas de clases que se utilizarán en el programa, para esto se hace uso de la palabra reservada import.

La clase principal de una aplicación Java deberá tener el método main, este método también se le conoce como método principal el cual deberá tener las propiedades de ser publico, estático y no retornar valor, para esto se utilizan las palabras reservadas public, static y void. El método main puede recibir parámetros desde la línea de comandos, desde donde se invoca a la clase para ser ejecutada por la máquina virtual Java, por lo cual es necesario declarar el parámetro que se puede utilizar, en este caso es una arreglo de cadenas que se declaran String[] args. El método main al igual que todos los métodos de una clase están delimitados por los simbolos de ambito o alcance que determinan los límites del método que son { para indicar el inicio del cuerpo del método y } para indicar el fin del cuerpo del método, todo lo que están dentro de estos simbolos son las declaraciones y sentencias que refieren a lo que realizará el método; en temas posteriores se explicará con mayor detalle el uso de los métodos al igual las declaraciones y sentencias que son permitidas dentro de un método.

Una clase siempre estará delimitada por los simbolos de ambito o alcance, que determina donde inicia y termina una clase. El símbolo { se utiliza para indicar el inicio de una clase y el símbolo } se utiliza para indicar la terminación de una clase. Dentro de una clase podrán ir declaraciones de datos miembros de

Fundamentos de programación orientada a objetos con Java

una clase así como los métodos de esta.

1.2Comentarios.

El elemento que nos permite describir en el idioma español las características de una aplicación así como descripción de elementos en particular, es el comentario que podemos insertar dentro del código de un programa; los comentarios sirven para generar la documentación de la aplicación y/o programa.

En java los comentarios pueden ser de una sola línea o de multiples líneas, el primero se utiliza para comentar alguna línea en particular de un programa o la declaración de las variables. Por el contrario los comentarios de multiples líneas se utilizan para describir a la aplicación y/o programa, así como los métodos que se encuentran dentro de una clase.

Ejemplo de comentario de una sola línea:

```
:  
int costos; // Describe el costo del producto  
:
```

Ejemplo de comentario de multiples líneas

```
/*  
   Clase de ejemplo para iniciar a programar en Java  
   Elaborada en la versión J2SE 1.5.0  
*/  
  

```

En el JDK de Java se tiene una herramienta llamada *javadoc*, que permite generar la documentación de las clases, datos miembros y métodos, esta herramienta utiliza los comentarios que se hacen para poder generar la documentación en HTML. Los comentarios para *javadoc* siempre tienen el formato siguiente:

```
/** comentario */
```

Fundamentos de programación orientada a objetos con Java

Es similar al comentario de multiples líneas, pero los utiliza la herramienta javadoc para poder generar la documentación, algunas etiquetas que se pueden utilizar en estos comentarios son las siguientes:

ETIQUETA	DESCRIPCION
@author <i>nombre</i>	Para especificar los datos del creador de elementos.
@deprecated <i>texto</i>	Para especificar elemento que ya no se usa y el texto deberá especificar que hacer.
@exception <i>texto de la excepción</i> @throws <i>texto de la excepción</i>	Para especificar excepciones cuando un elemento envia dicha excepción.
{@link <i>nombre de la etiqueta</i> }	Para insertar vínculos a otra página javadoc.
@param <i>nombre del texto</i>	Para describir los parámetros de los métodos.
@return <i>texto</i>	Para describir el valor de retorno de un método.
@see <i>referencia</i>	Para referencia cruzada con otra clase, paquete o documento.
@since <i>texto</i>	Para especificar que se añadio algo.
@serial <i>texto</i> @serialField <i>texto con nombre de tipo</i> @serialData <i>texto</i>	Para documentar la naturaleza serializable de una clase.
@versión <i>texto</i>	Para especificar la versión de un elemento.

El siguiente ejemplo muestra el uso de algunas de estas etiquetas:

```
import java.lang.*;  
import java.util.*;  
  
/**  
 * Ejemplo <code><h1>Hola</h1></code>solo para presentar uso de javadoc  
 * <p>  
 * <P> Esta clase permite ejecutar el primer programa en Java  
 * Solo es util por el que lo realizo  
 * @author Lic Rafael Herrera Garcia  
 * @version 1.0  
 */  
  
public class Hola{  
/**  
 *
```

Fundamentos de programación orientada a objetos con Java

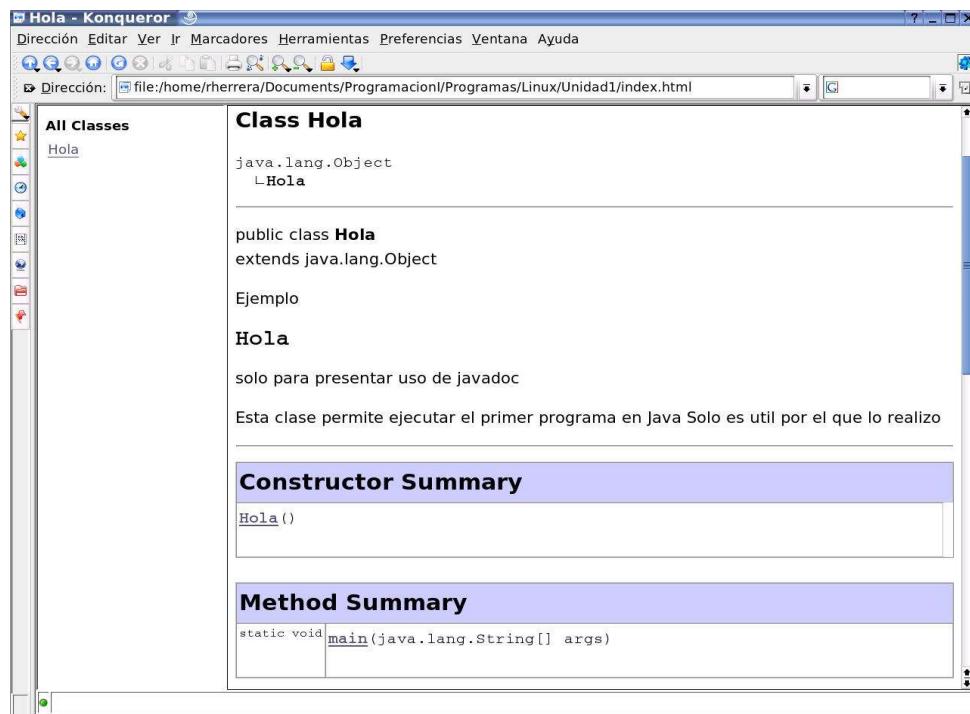
```
* @version 1.0
* @return tipo void ya que no regresa nada
* @param args se utiliza para pasar argumentos desde la linea de comandos
* <h4>Metodo main </h4>
* <p> version 1.0
* <p> Se ejecuta de forma inmediata una vez que se invoca a java Hola
* Directamente ejecuta las instrucciones que conforman el metodo
*/
public static void main(String [] args){

    System.out.println("Hola Mundo");
}
}
```

Una vez editado el programa, para generar la documentación se utiliza la utileria javadoc de la siguiente forma:

```
javadoc Hola.java
```

El ejecutar la utileria con el archivo a procesar, se genera de manera automática los archivos de documentación, donde el principal archivo es el nombrado index.html, el cual se puede cargar de forma directa en un browser de internet.



1.3 Variables y constantes.

En todo programa de computadoras deben existir áreas de memoria perfectamente identificadas donde se pueda guardar la información. A estas áreas de memoria se les identifica por un número conocido como dirección de memoria, pero se prefiere referenciar a estas por medio de nombres o identificadores.

Una variable es un identificador que permite referenciar una dirección de memoria, en dicha dirección se podrá guardar un valor y modificarlo durante la ejecución del programa.

Una constante es un identificador que permite referenciar un valor y no se puede modificar durante la ejecución del programa.

Las reglas de los identificadores en Java indican que se forman como una secuencia de caracteres y no existe un límite de cuantos caracteres deberán estar formados; se requiere que el primer carácter sea una letra a..z A..Z, el subrayado (_) o el símbolo de \$ y seguido de cualquier cantidad de caracteres pudiendo ser letras, dígitos y símbolos especiales que se encuentren dentro del UNICODE. Este último es el formato para representar caracteres, es un código de 16 bits lo cual permite casi cualquier símbolo dentro de un identificador, algunos ejemplos de identificadores válidos son los siguientes:

- número
- año
- costo_producción
- cadena_12
- \$peso40
- _\$45

Tanto las variables como las constantes deben ser declaradas en un programa, la sintaxis para ambas son las siguientes:

VARIABLE

[ambito] [static] tipoDeDato identificador;

Donde:

ambito : Es opcional; si se trata de una variable declarada dentro de un método no se necesita; si se trata de una variable de clase que también se le llama dato miembro de la clase este puede tomar los valores: *public*, *private* o *protected*; **public** indica que se trata de un dato miembro que su alcance puede ser utilizada de forma directa desde otras clases incluyendo la clase donde ha

sido definida, **private** indica que se trata de un dato miembro que su alcance es exclusivo de los miembros de la clase y no se permite su utilización en forma directa desde otras clases, **protected** indica que se trata de un dato miembro que su alcance es exclusivo de los miembros de la clase y por los miembros de las clases derivadas de la clase donde ha sido declarada, para mayor información en el tema de herencia se hará uso de este ámbito.

static: Es opcional y solo se utiliza para describir variables de clase; una variable de clase definida static indica que es única para todos los objetos que se creen e incluso se pueden acceder a ella sin que se creen objetos.

tipoDeDato: Es cualquier tipo de dato básico o definido por el usuario, por lo general los tipos de datos definidos por el usuario se dice que son estructuras de datos y en particular en Java se trata de clases definidas por el usuario o definidas por el sistema. Este tema se discutirá en la sección 1.3 de este capítulo.

Ejemplos de declaraciones de variables son las siguientes:

```
private int x;  
protected float costo;  
  
boolean bandera;
```

CONSTANTE

```
final [ámbito] tipoDeDato identificador = expresión;
```

Las diferencias con las variables es que utilizan la palabra reservada final, para indicar que su valor no cambiara y utilizan una expresión para indicar el valor que se le asignara a la constante. Muchas veces se prefiere que los identificadores de constantes estén en letras mayúsculas.

```
final int MAX_VALOR=45;  
final public float IVA=.15F;
```

El ámbito tiene la misma característica que en las variables, el tipo de dato de una variable restringe la expresión a utilizar para indicar el valor.

Las expresiones válidas en Java tienen que ver con los operadores que más adelante se detallaran; muchas veces la expresión es una literal como los ejemplos mostrados anteriormente. Las literales dependen del tipo de dato que se esté utilizando, así tenemos que:

Literales enteras

int, short, byte	Decimal dígitos entre 0-9 Octal Inicia con un 0 seguido por dígitos entre 0-7 Hexadecimal Inicia con 0x ó 0X seguido por dígitos 0-9 o letras a-f o A-F
long	Se agrega una L o l al final de una literal int.

Literales de punto flotante

float	Se debe agregar una f o F al final de una literal double
double	Cualquier combinación de dígitos separados con un punto en cualquier posición. Es valida la notación científica con la letra E o e; se puede utilizar al final las letras D o d.

Literales Booleana

boolean	true o false
---------	--------------

Literales de caracter	Cualquier caracter encerrado entre comillas simples ('') o cualquier secuencia de escape.
-----------------------	---

Secuencias de escape:

Secuencia	Descripción
\b	Retroceso
\t	Tabulador Horizontal
\n	Nueva línea
\f	Alimentación de página
\r	Retorno de carro
\”	Comillas dobles
\'	Comilla simple
\\\	Barra inversa
\DDD	Caracter en octal
\uHHHH	Caracter en hexadecimal

A continuación el programa muestra algunos ejemplos de constantes y su inicialización con literales apropiadas.

```
import java.lang.*;
public class demo0 {
    public static void main(String[] args) {

        final int numeroI=100;                      // Inicializa en decimal
        final float numeroF=4.56e-3F;                // Inicializa en notación científica
        final double numeroD=4.67E10;                 // Inicializa en notación científica
        final boolean Vbooleana=true;                 // Inicializa a true
        final short numeroS=0x4F;                     // Inicializa con valor en Hexadecimal
        final byte numeroB=034;                       // Inicializa con valor en Octal
        final char caracter='\'u0041';                // Inicializa con secuencia de escape

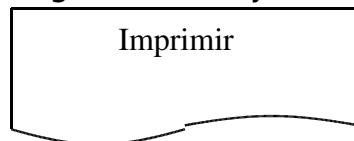
        System.out.println(numeroI);
        System.out.println(numeroF);
        System.out.println(numeroD);
        System.out.println(Vbooleana);
        System.out.println(numeroS);
        System.out.println(numeroB);
        System.out.println(caracter);
    }
}
```

1.4 Objetos que permiten E/S por consola.

Dos de las operaciones que permiten a los usuarios de una aplicación interactuar con ella son las de entrada/salida; estas operaciones permiten que por medio de dispositivos de entrada y/o salida el usuario pueda interactuar; los ejemplos más comunes son la pantalla del monitor que se caracteriza por ser de salida (imprimir, escribir, etc.); el teclado que se caracteriza por ser de entrada (leer); todo lenguaje de programación debe ofrecer mecanismos para llevar a cabo estas operaciones, Java no es la excepción.

Dentro de la biblioteca de clases `java.lang.*`; se localiza la clase `System`, esta ofrece un mecanismo para entrada y salida de datos. Para iniciar tomaremos el objeto `out` para indicar operaciones de salida de datos.

Su simbolo en un diagrama de flujo se representa con el símbolo:



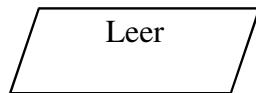
Fundamentos de programación orientada a objetos con Java

En Java se utiliza los métodos print y println del objeto out localizado en System, como se muestra a continuación.

```
System.out.print(cadena);
System.out.print(variable);
System.out.println(cadena);
System.out.println(variable);
```

La entrada de datos utiliza un flujo de entrada, para lo cual se ofrece el objeto in de la clase System, sin embargo su uso no es muy simple, para los primeros programas se utilizará la clase JOptionPane que se localiza en la jerarquía javax.swing.*., por lo cual se deberá importar esta jerarquía para poder hacer uso de esta clase. JOptionPane tiene un método que es showInputDialog que permite crear una caja de diálogo para capturar una cadena de caracteres; esta cadena capturada posteriormente podrá ser convertida al tipo de dato que más convenga al programador.

El símbolo del diagrama de flujo que representa la entrada de datos es el siguiente:



Un ejemplo de entrada de datos es la siguiente:

```
import java.lang.*;
import javax.swing.*;

public class demo1 {

    public static void main(String[] args) {

        String entrada;
        int numero;

        entrada=JOptionPane.showInputDialog("Dar un valor numérico entero:");
        numero=Integer.parseInt(entrada);

        System.out.println("El valor leido entero es: "+numero);
        System.exit(0);
    }
}
```

En este ejemplo se muestra como la entrada de datos se realiza con showInputDialog y lo guarda en una variable de tipo String y posteriormente la convierte utilizando el método parseInt de la clase Integer para guardar el valor como entero en la variable *numero*. Aquí la pregunta sería si tuviera que leer un valor de otro tipo (float, char, boolean, etc.) ¿Como tendría que realizarlo?; bueno la lectura es como una cadena de caracteres y esto no cambia showInputDialog, tendríamos que analizar de que forma queremos utilizar lo leido y convertirlo al formato deseado, por ejemplo:

```
import java.lang.*;
import javax.swing.*;

public class demo2 {

    public static void main(String[] args) {

        String entrada;
        int numerol;
        float numeroF;
        double numeroD;
        boolean VBoleana;
        short numeroS;
        byte numeroB;
        char caracter;

        entrada=JOptionPane.showInputDialog("Dar un valor numérico entero:");
        numerol=Integer.parseInt(entrada);
        entrada=JOptionPane.showInputDialog("Dar un valor numérico entero corto:");
        numeroS=Short.parseShort(entrada);
        entrada=JOptionPane.showInputDialog("Dar un valor numérico entero Byte:");
        numeroB=Byte.parseByte(entrada);
        entrada=JOptionPane.showInputDialog("Dar un valor numérico flotante:");
        numeroF=Float.parseFloat(entrada);
        entrada=JOptionPane.showInputDialog("Dar un valor numérico flotante doble:");
        numeroD=Double.parseDouble(entrada);
        entrada=JOptionPane.showInputDialog("Dar un valor booleano:");
        VBoleana=Boolean.valueOf(entrada).booleanValue();
        entrada=JOptionPane.showInputDialog("Dar un valor carácter:");
        caracter=entrada.charAt(0);

        System.out.println("El valor leido entero es: "+numerol);
        System.out.println("El valor leido short es: "+numeroS);
        System.out.println("El valor leido byte es: "+numeroB);
        System.out.println("El valor leido flotante es: "+numeroF);
        System.out.println("El valor leido double es: "+numeroD);
        System.out.println("El valor leido booleano es: "+VBoleana);
        System.out.println("El valor leido carácter es: "+caracter);

        System.exit(0);
    }
}
```

Aquí se muestra como se lee cada uno de los tipos de datos básicos y como se pueden convertir posteriormente para su uso.

1.5 Operadores.

Muchas de las operaciones que una computadora puede realizar estan asociadas con acciones que ejecuta la unidad aritmética y lógica. Un operador representa una abstracción de dichas acciones, así tenemos que un operador representa una operación que permite manipular valores de datos para obtener un resultado. Un operador por si solo no puede realizar algo, necesita de los datos que tendra que utilizar a los cuales se les llama *operandos*; estos pueden ser variables, constantes, literales o resultados obtenidos de invocaciones a métodos.

Los operadores por el tipo de operación que llevan a cabo se les clasifica en: aritméticos, relacionales, lógicos, a nivel de bits (bitwise) y especiales. La prioridad de los operadores indica el orden en que estos seran evaluados cuando se encuentren más de uno en una expresión, la más alta prioridad es 1 e indica que este será en primer lugar. Con la salvedad del autoincremento en postfijo que se evalua una vez que ha sido utilizado su valor. La prioridad de los operadores puede ser alterada por el uso de parentesis los cuales asocian operaciones que necesitemos ocurran antes que cualquier otras operaciones no importando que tengan menos prioridad.

Operadores Aritméticos

Representa operaciones aritméticas básicas (adición, substracción, multiplicación y división) en la siguiente tabla se resume su descripción y uso.

<i>Operador</i>	<i>Función</i>	<i>Valor de retorno</i>	<i>Prioridad</i>	<i>Asociatividad</i>	<i>Ejemplo</i>
<code>++, --</code>	Autoincremento y Autodecremento postfijo	-	1	Izquierda	<code>i++</code> <code>costo++</code>
<code>++,--</code>	Autoincremento y Autodecremento prefijo	-	2	Derecha	<code>++i</code> <code>++costo</code>
<code>+,-</code>	Unario positivo y Unario negativo	tipo de dato que interviene	2	Derecha	<code>+i</code> <code>-costo</code>

Fundamentos de programación orientada a objetos con Java

<i>Operador</i>	<i>Función</i>	<i>Valor de retorno</i>	<i>Prioridad</i>	<i>Asociatividad</i>	<i>Ejemplo</i>
*	Multiplicación	tipo de dato que interviene	4	Izquierda	i*costo
/	División	tipo de dato que interviene	4	Izquierda	costo/i
%	Residuo de división	tipo de dato que interviene	4	Izquierda	i%5
+,-	Adición y Resta	tipo de dato que interviene	5	Izquierda	costo+i i-5

Ejemplos del uso de los operadores aritméticos:

<i>Declaraciones</i>	<i>Ejemplos</i>	<i>Arbol de evaluación</i>	<i>Significado</i>
int z=10; short x=9; final byte y=12;	z=x+++z; x=y+8;	$ \begin{array}{c} z = x+++z \\ \diagdown \quad \diagup \\ 19 \\ 10 \end{array} $ $ \begin{array}{c} x = y + 8 \\ \diagdown \quad \diagup \\ 20 \\ 12 + 8 \end{array} $	a z se le asigna el resultado de 9+10 (19) y despues x se incrementa en 1(10) a x se le asigna y+8 (12+8) quedando 20 en x.
long a; float b; final int x=10;	a=x+100; b=++a-5*x; a= a / x; b= a / x;	$ \begin{array}{c} a = x + 100 \\ \diagdown \quad \diagup \\ 110 \end{array} $ $ \begin{array}{c} b = ++a - 5 * x \\ \diagdown \quad \diagup \quad \diagup \\ 111 - 50 \\ \diagdown \quad \diagup \\ 61 \end{array} $ $ \begin{array}{c} a = a / x \\ \diagdown \\ 11 \end{array} $ $ \begin{array}{c} b = a / x \\ \diagdown \\ 11.0F \end{array} $	a a se le asigna el resultado de x+100 (110) a a se incrementa en 1(111) y despues se multiplica 5*x10 (50) y por ultimo se resta 111-50 y se asigna a b (61) a a se le asigna a/x, divide (enteros) 110 / 10 (11) a b se le asigna a/x, divide (enteros) 110/10 (11.0F)

Declaraciones	Ejemplos	Arbol de evaluación	Significado
float x=10.5F; int y=15;	y = y / 2; x = y / 2; x= y / 2.0F; y = y%2; x= -x ++y;	y = y / 2 \ / 7 x = y / 2 \ / 3.0F x = y / 2.0F \ / 3.5F y = y%2 \ / 1 x= -x ++y \ / -3.5 + 6 \ / 2.5	divide (enteros) y/2, 15/2 y el resultado se guarda en y (7) divide (enteros) y/2, 7/2 y el resultado se guarda en x como flotante(3.0F) divide(flotantes) y/2.0F, 7/2.0F y el resultado se guarda en x (3.5F) divide(enteros) y%2, 7%2 y el residuo se guarda en y (1). Se pone signo negativo a x (-3.5). Se decrementa y en 1 (6), se suma -3.5+6 (2.5) y este resultado se guarda en x

Operadores Relacionales

Representan operaciones de comparación, que permiten verificar la relación entre valores, en la siguiente tabla se resume su descripción y uso.

Operador	Función	Valor de retorno	Prioridad	Asociatividad	Ejemplo
>, <, >=, <=	Compara si es mayor que, menor que, mayor o igual que y menor o igual que	Booleano	7	Izquierda	i>costo costo < 400 costo >= 300 i <= 10
==	Compara si es igual que	Booleano	8	Izquierda	costo == 200
!=	Compara si es diferente	Booleano	8	Izquierda	costo != 200

<i>Operador</i>	<i>Función</i>	<i>Valor de retorno</i>	<i>Prioridad</i>	<i>Asociatividad</i>	<i>Ejemplo</i>
?:	Devuelve uno de 2 valores posibles en función de un tercer valor	Booleano	14	Izquierda	i!=5? costo:costo-2

Ejemplos del uso de operadores relacionales:

<i>Declaraciones</i>	<i>Ejemplos</i>	<i>Arbol de evaluación</i>	<i>Significado</i>
int x=10; long y=100L; boolean r;	r= x >=10; r = y++ != x*10;	r = x >=10 true r = y++ != x*10 100 != false 101	Compara x con 10, indicando si es mayor o igual que (10>=10) guardando true en r. Multiplica x * 10, 10*10 (100), compara y (100) con 100 si es diferente que (100 != 100) guardando false en r y posteriormente incrementa y en 1(1001)
	r= ++y != x*10;	r=++y != x*10 101 != 100 true	Incrementa y en 1(101), Multiplica x * 10, 10*10 (100), compara y (101) con 100 si es diferente que (101 != 100) guardando true en r.
	x= x>10?x*5:x*10;	x= x>10?x*5:x*10 false 100	Compara x con 10, si es mayor que (10 > 10), evalua el resultado de la expresión y si es true ejecuta la expresión despues del signo (?) y si es falso ejecuta la expresión despues del signo(:). Guarda el resultado en x(100)

Operadores lógicos

Estos operadores permiten construir expresiones que representan enlaces lógicos y, o , la negación y la disyunción exclusiva. En la siguiente tabla se muestran su descripción y uso.

<i>Operador</i>	<i>Función</i>	<i>Valor de retorno</i>	<i>Prioridad</i>	<i>Asociatividad</i>	<i>Ejemplo</i>
!	Negación de una expresión	Booleano	2	Derecha	! (valor>4)
&	Conjunción lógica completa	Booleano	9	Izquierda	valor>4 & valor<9
^	Disyunción exclusiva	Booleano	10	Izquierdo	valor<9 ^ valor<4
	Disyunción lógica completa	Booleano	11	Izquierda	valor <4 valor>10
&&	Conjunción lógica incompleta	Booleano	12	Izquierda	valor>4 && valor<9
	Disyunción lógica incompleta	Booleano	13	Izquierda	valor <4 valor>10

Tabla de verdad de los operadores lógicos

<i>Operando A</i>	<i>Operando B</i>	<i>Resultado(&, &&)</i>
true	true	true
false	true	false
true	false	false
false	false	false

<i>Operando A</i>	<i>Operando B</i>	<i>Resultado(,)</i>
true	true	true
false	true	true
true	false	true
false	false	false

<i>Operando A</i>	<i>Operando B</i>	<i>Resultado(^)</i>
true	true	false
false	true	true
true	false	true
false	false	false

<i>Operando A</i>	<i>Resultado(!)</i>
true	false
false	true

Ejemplos de uso de operadores lógicos

<i>Declaraciones</i>	<i>Ejemplos</i>	<i>Arbol de evaluación</i>	<i>Significado</i>
int y=15; short x=5; boolean r;	r= y>=1 && y <=20; r = y>=1 x>=20; r = ! x< 5;	 	<p>Evaluá $y \geq 1$, si es verdadera, evalúa $y \leq 20$, una vez con los 2 resultados les aplica una conjunción y el resultado se guarda en r. Se dice que es incompleta ya que si la primera evaluación hubiera dado false no evalúa la segunda.</p> <p>Evaluá $y \geq 1$ si es verdadero sabe que al aplicar un operador $\$ con cualquier otro valor el resultado será verdadero así que ya no evalúa la segunda expresión, por eso se dice que es incompleta.</p> <p>Evaluá $x < 5$, el resultado lo niega($!$) en este caso false y el resultado será true y se guarda en r.</p>

Operadores a nivel de bits(bitwise)

Estos operadores se utilizan para manipular los bits de valores de tipo entero exclusivamente. Como su nombre indica trabaja a nivel de bits. En la tabla siguiente se muestra su descripción y uso.

<i>Operador</i>	<i>Función</i>	<i>Valor de retorno</i>	<i>Prioridad</i>	<i>Asociatividad</i>	<i>Ejemplo</i>
<code>~</code>	Complemento a 1	tipo de dato que interviene	2	Derecha	<code>~ valor</code>
<code>>>, <<</code>	Desplazamientos a la izquierda y derecha respectivamente	tipo de dato que interviene	6	Izquierda	<code>valor >> 3</code> <code>valor << 4</code>
<code>>>></code>	Desplazamiento a la derecha sin signo.	tipo de dato que interviene	6	Izquierda	<code>valor >>> 5</code>
<code>&</code>	Aplica un AND por pares de bits de los dos operandos	tipo de dato que interviene	9	Izquierda	<code>valor & 0xFF00</code>
<code> </code>	Aplica un OR por pares de bits de los dos operandos	tipo de dato que interviene	11	Izquierda	<code>valor 0x00F0</code>
<code>^</code>	Aplica un XOR por pares de bits de los dos operandos	tipo de dato que interviene	10	Izquierda	<code>valor ^ 0xFFFF</code>

Ejemplos de uso de operadores a nivel de bits

<i>Declaraciones</i>	<i>Ejemplo</i>	<i>Representación en la memoria</i>	<i>Significado</i>
<code>int x=0x00ff;</code>	<code>x = x && 0xfffff42;</code>	<pre>0000000000000000 0000000111111111 1111111111111111 1111111101000010 0000000000000000 000000001000010</pre>	Aplica bit por bit un operador and. <code>0x00ff</code> <code>0xfffff42</code> El resultado es 66. (<code>0x00000042</code>)

<i>Declaraciones</i>	<i>Ejemplo</i>	<i>Representación en la memoria</i>	<i>Significado</i>																																																																
int x; final byte y=0x0F;	x= ~y;	<table border="1"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> </table> <table border="1"> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	El valor original (15) se convierte de byte a int (32 bits) Al aplicar el complemento a 1 todos los 0 se convierten en 1 y viceversa.(-16)
0	0	0	0	0	0	0	0																																																												
0	0	0	0	0	0	0	0																																																												
0	0	0	0	0	0	0	0																																																												
0	0	0	0	1	1	1	1																																																												
1	1	1	1	1	1	1	1																																																												
1	1	1	1	1	1	1	1																																																												
1	1	1	1	1	1	1	1																																																												
1	1	1	1	0	0	0	0																																																												
int x,x2; byte y=0x0F; byte z=2;	x = y << 2;	<table border="1"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> </table> <table border="1"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> </table>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	Este operador desplaza los bits a la derecha, en este caso 2 y entran en su lugar 2 ceros al lado izquierdo quedando el valor 60 en decimal.
0	0	0	0	0	0	0	0																																																												
0	0	0	0	0	0	0	0																																																												
0	0	0	0	0	0	0	0																																																												
0	0	0	0	1	1	1	1																																																												
0	0	0	0	0	0	0	0																																																												
0	0	0	0	0	0	0	0																																																												
0	0	0	0	0	0	0	0																																																												
0	0	1	1	1	1	0	0																																																												
	x= x << 26;	<table border="1"> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table>	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Al hacer este corrimiento el valor resultante es negativo (-268,435,456).																																
1	1	1	1	0	0	0	0																																																												
0	0	0	0	0	0	0	0																																																												
0	0	0	0	0	0	0	0																																																												
0	0	0	0	0	0	0	0																																																												
	x2= x >>> 26;	<table border="1"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> </table>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	Al regresar los bits a la derecha pero sin signo el resultado es el original (60)																																
0	0	0	0	0	0	0	0																																																												
0	0	0	0	0	0	0	0																																																												
0	0	0	0	0	0	0	0																																																												
0	0	1	1	1	1	0	0																																																												
	x2= x >> 26;	<table border="1"> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> </table>	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	Al regresar los bits a la derecha con signo el resultado es -4.																																
1	1	1	1	1	1	1	1																																																												
1	1	1	1	1	1	1	1																																																												
1	1	1	1	1	1	1	1																																																												
1	1	1	1	1	1	0	0																																																												

Operadores especiales

<i>Operador</i>	<i>Función</i>	<i>Ejemplo</i>
(casting)	Convertir el valor de una expresión; tener cuidado con intentar convertir un flotante a entero.	a/(float)x
instanceof	Se utiliza para probar la clase de un objeto	ObjetoP instanceof persona
new	Se utiliza para crear objetos en tiempo de ejecución.	persona X; X= new persona();

El siguiente ejemplo muestra el uso de algunos operadores, al ejecutar el programar revisar los resultados para darles una interpretación correcta del uso que tienen.

```
import java.lang.*;
public class demo3 {
    public static void main(String[] args) {
        int numeroI;
        float numeroF;
        double numeroD;
        boolean VBoleana;
        short numeroS;
        long numeroL;
        byte numeroB;
        char caracter;
        String cadena;

        numeroI=10;
        numeroF=1.4F;
        numeroD=1.5;
        numeroS=20;
        numeroL=30;
        numeroB=1;
        caracter='A';
        VBoleana=true;
        cadena= new String("Hola Mundo");

        System.out.println(numeroI++);
        System.out.println(numeroF*numeroI);
        System.out.println(numeroD-numeroF);
        System.out.println(VBoleana & numeroI!=10);
        System.out.println(VBoleana && numeroI!=10);
        System.out.println(numeroS/numeroL);
        System.out.println(numeroS/(float)numeroL);
        System.out.println(numeroS % 3);
        System.out.println(numeroI << 2);
    }
}
```

```
System.out.println(numero1 >> 2);
System.out.println(caracter=='A');
System.out.println(cadena);
System.out.println(cadena instanceof String);
System.out.println(numeroB>1?50:60);
}
}
```

1.6 Tipos de datos.

Las variables y las constantes como se vio anteriormente deben ser de un tipo de dato, este se refiere al tipo de información que se desea representar, en una computadora todos los datos son información, pero no puede ser información arbitraria, esto quiere decir que los programas de computadoras solo trabajan con datos que estén perfectamente identificados por el lenguaje de programación, así que un lenguaje de programación debe ofrecer un conjunto de tipos de datos básicos con los cuales se permita representar los datos que el programa necesitará.

1.6.1 Fundamentales.

En Java los tipos de datos básicos también llamados fundamentales están descritos en la siguiente tabla:

<i>Tipo de dato</i>	<i>Descripción</i>	<i>Tamaño en bits</i>	<i>Valor mínimo</i>	<i>Valor Máximo</i>
boolean	Valor booleano	8 bits	true	false
byte	Entero con signo	8 bits	-128	128
short	Entero con signo	16 bits	-32768	32767
int	Entero con signo	32 bits	-2147483648	2147483647
long	Entero con signo	64 bits	-9223372036854775808	9223372036854775807
float	Flotante simple IEEE 754	32 bits	1.40239846 x 10 ⁻⁴⁵ -3.40282347 x 10 ³⁸	3.40282347 x 10 ³⁸
double	Flotante doble IEEE 754	64 bits	4.94065645841246544 x 10 ⁻³²⁴ -1.79769313486231570 x 10 ³⁰⁸	1.79769313486231570 x 10 ³⁰⁸
char	Caracter UNICODE	16 bits	\u0000	\uffff

Los operadores aritméticos solo operan sobre tipos de datos básicos, por lo cual se tiene que tener cuidado cuando se desea hacer alguna de estas operaciones con tipos de datos que no sean de este tipo.

1.6.2 Definidos por el usuario.

Un tipo de dato definido por el usuario no es más que una estructura de datos que describe un tipo de dato que no está disponible por el lenguaje; en el caso de Java una estructura de datos se define utilizando la definición de clase. Por otro lado en la biblioteca de Java existen muchas clases que se pueden utilizar y pasan a ser una conjunto de tipos de datos que fueron definidos por el desarrollo de Java pero que no son tipos de datos básicos; algunas de estas clases son los wrapper (clases envoltura) o clases de los tipos de datos básicos: Integer, Float, Double, Boolean, Character, Byte, Long, Short. En la documentación de las API's (Interfaz de programa de aplicación) de Java estas clases se encuentran descritas.

Un tipo definido por el usuario muchas de las veces también se le llama tipo de dato abstracto ya que no es un tipo de dato que provee el lenguaje y se construye partiendo de los tipos de datos básicos, es importante aclarar que un tipo de dato definido por el usuario no solo describe datos sino también las operaciones con las cuales dichos datos se podrán manipular. Las operaciones también son abstracciones de las operaciones básicas del lenguaje y se describen como métodos, estos y la definición de clases se describen en los capítulos siguientes. Por el momento es importante conocer que podemos utilizar otros tipos de datos aparte de los básicos.

1.7 Palabras reservadas.

Cada lenguaje de programación utiliza un conjunto de palabras que solo pueden ser utilizadas con un propósito, por lo cual no se permite que el programador haga uso de estas para otro propósito, a estas se les llama palabras reservadas, en el caso de Java el conjunto de palabras reservadas son las siguientes:

abstract	boolean	break	byte
case	catch	char	class
<i>const</i>	continue	default	do
double	else	extends	final
finally	float	for	goto
if	implements	import	instanceof
int	interface	long	native
new	package	private	protected
public	return	short	static

abstract	boolean	break	byte
strictfp	super	switch	synchronized
this	throw	throws	transient
try	void	volatile	while

1.8 Expresiones.

Una expresión es una forma de expresar una serie de operaciones que queremos que la computadora realice, esta formada por operadores y operandos, los operandos pueden ser variables, constantes o invocación a métodos.

Las reglas para escribir expresiones están en función del uso de los operadores vistos anteriormente, muchas veces una fórmula matemática, una ecuación o cualquier cálculo se representa mediante una o un conjunto de expresiones; por ejemplo:

Expresión Matemática	Expresión Computacional en Java
$\frac{(a+b)}{(c-d)}$	(a+b) / (c-d)
$\frac{f}{g} + \frac{k}{d}$	f/g + k/d
$\frac{(x+y)}{(x-5)} - \frac{(y-1)}{(y-x)}$	(x+y)/(x-5) - (y-1)/(y-x)
$f + \frac{X}{Z} - \frac{Z}{W}$	f+ x/z -z/w

1.9 Estructuras de control.

La ejecución de un programa sea cual sea el paradigma de programación requiere que las instrucciones sigan una secuencia de ejecución, esta secuencia normalmente es secuencial, esto quiere decir que ejecuta instrucción por instrucción aunque pueden existir otros tipos de secuencias; la secuencial es la más común.

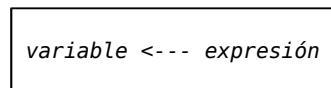
Una estructura de control determina la secuencia mediante la cual un programa se ejecutará. Los siguientes subtemas tratan las estructuras de control más comunes en la mayoría de los lenguajes de programación incluyendo a Java.

1.9.1 Asignación.

Es la estructura de control básica y permite modificar el contenido de una variable. La sintaxis en Java es la siguiente:

variable = expresión;

El símbolo representado en un diagrama de flujo para la asignación es el siguiente:



Donde **variable** es una variable previamente declarada y que este dentro del ámbito de donde se efectuará la asignación, puede ser una variable local de un método o una variable dato miembro; **expresión** debe seguir las reglas vistas anteriormente considerando que los tipos de datos del valor obtenido de la evaluación de la expresión sea compatible con el de la variable, considerar que una expresión numérica solo puede ser asignada a una variable numérica, una expresión booleana solo puede ser asignada a una variable booleana, etc. También se debe considerar que una expresión numérica entera puede ser asignada a una variable entera o de punto flotante, pero una expresión numérica de punto flotante solo puede ser asignada a una variable de punto flotante. Considerar a los operadores de asignación vistos anteriormente como casos especiales de asignación.

Los operadores de asignación adicionales son los que se muestran en la siguiente tabla:

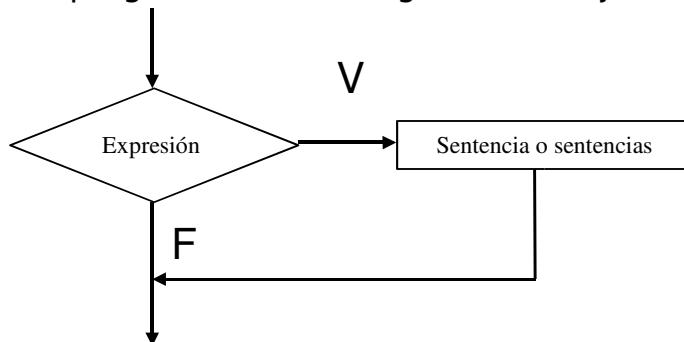
Operador	Función	Prioridad	Asociatividad	Ejemplo
=	Asignación	15	Derecha	a= 20;
=	Multiplica y asigna	15	Derecha	a=20; // a=a*20;
/=	Divide y asigna	15	Derecha	a/=10; //a=a/10;
%=	Resto y asigna	15	Derecha	a%-=10;//a=a%10;
+=	Suma y asigna	15	Derecha	a+=10;//a=a+10;
-=	Resta y asigna	15	Derecha	a=-10;//a=a-10;
>>=	Desplaza y asigna	15	Derecha	a>>=5;//a=a>>5;
<<=	Desplaza y asigna	15	Derecha	a<<=5;//a=a<<5;

Operador	Función	Prioridad	Asociatividad	Ejemplo
>>>=	Desplaza y asigna	15	Derecha	a>>>=3;//a=a>>>3;
=	O a nivel y bits y asigna	15	Derecha	a =0xff;//a=a 0xff;
&=	Y a nivel de bits y asigna	15	Derecha	a&=0xff;//a=a&0xff;

1.9.2 Selección.

La selección permite realizar bifurcaciones durante la ejecución de un programa; existen diferentes tipos de selección, tenemos la selección simple, doble, anidada y múltiple.

La selección simple consiste en evaluar una expresión para realizar una sola bifurcación, si el resultado de la expresión es verdadera se ejecuta una instrucción o un conjunto de sentencias para posteriormente seguir la ejecución del programa, en un diagrama de flujo se represente de la siguiente forma:



La sintaxis en Java es la siguiente:

if (*Expresión*)
 Sentencia ;

```

if (Expresión)
{
    // grupo de sentencias
    Sentencia1;
    :
}
```

Es muy importante aclarar que la Expresión deberá regresar un valor booleano. El siguiente ejemplo muestra como se utiliza esta sentencia al leer un número entero e imprime si es par.

Un número al dividirse entre 2 si el residuo es igual a 0 quiere decir que es par. `numero%2` obtiene el residuo de dividir numero entre 2.

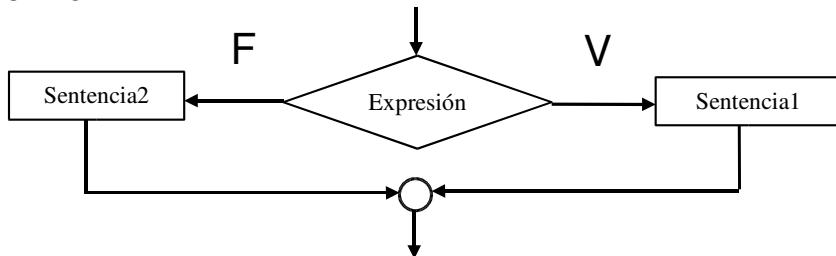
```
import java.lang.*;
import javax.swing.*;

public class Par {
    public static void main(String[] args) {
        int numero;
        String cadena;

        cadena= JOptionPane.showInputDialog("Dar un número:");
        numero=Integer.parseInt(cadena);

        if( numero%2 == 0)
            System.out.println(numero + " Es un número par");
        System.exit(0);
    }
}
```

La selección doble permite ejecutar sentencia(s) tanto si la expresión es verdadera como si es falsa; en un diagrama de flujo se representa de la siguiente forma:



La sintaxis en Java es la siguiente:

```
if (Expresión)
    Sentencia1;
else
    Sentencia2;

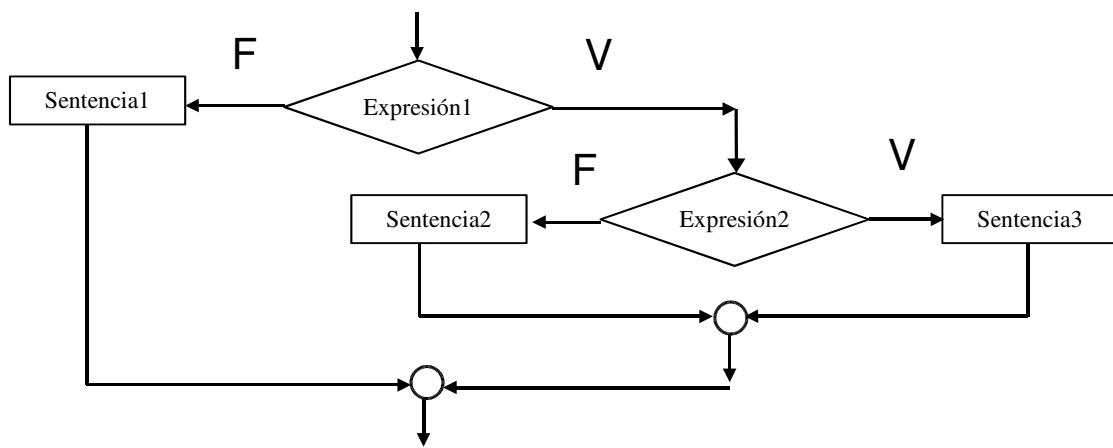
if(Expresión){
    // Grupo de sentencias parte verdadera
    Sentencia1;
    Sentencia2;
```

```
:  
}  
else {  
    // Grupo de sentencias parte falsa  
    Sentencia3;  
    Sentencia4;  
:  
}
```

Al igual que la anterior la Expresión debe regresar un valor booleano. El siguiente ejemplo muestra el uso de este tipo de selección cuando se lee una calificación de un alumno y se imprime si aprobo o reprobó; considerando que una calificación aprobatoria es mayor ó igual a 70 y reprobatoria es menor que 70.

```
import java.lang.*;  
import javax.swing.*;  
public class alumno {  
  
    public static void main(String[] args) {  
        float calificación;  
        String cadena;  
  
        cadena= JOptionPane.showInputDialog("Dar una calificación:");  
        calificación=Float.parseFloat(cadena);  
  
        if( calificación >= 70F)  
            System.out.println("Aprobo");  
        else  
            System.out.println("Reprobó");  
  
        System.exit(0);  
    }  
}
```

La selección anidada permite integrar cualquier tipo de selección como una sentencia dentro de otra selección las combinaciones que se pueden tener son muy variadas y no existe un límite de cuantas selecciones anidadas se pueden tener. Un ejemplo en un diagrama de flujo de una de las posibles combinaciones se muestra a continuación:



La sintaxis en Java para este caso de selección anidada es la siguiente:

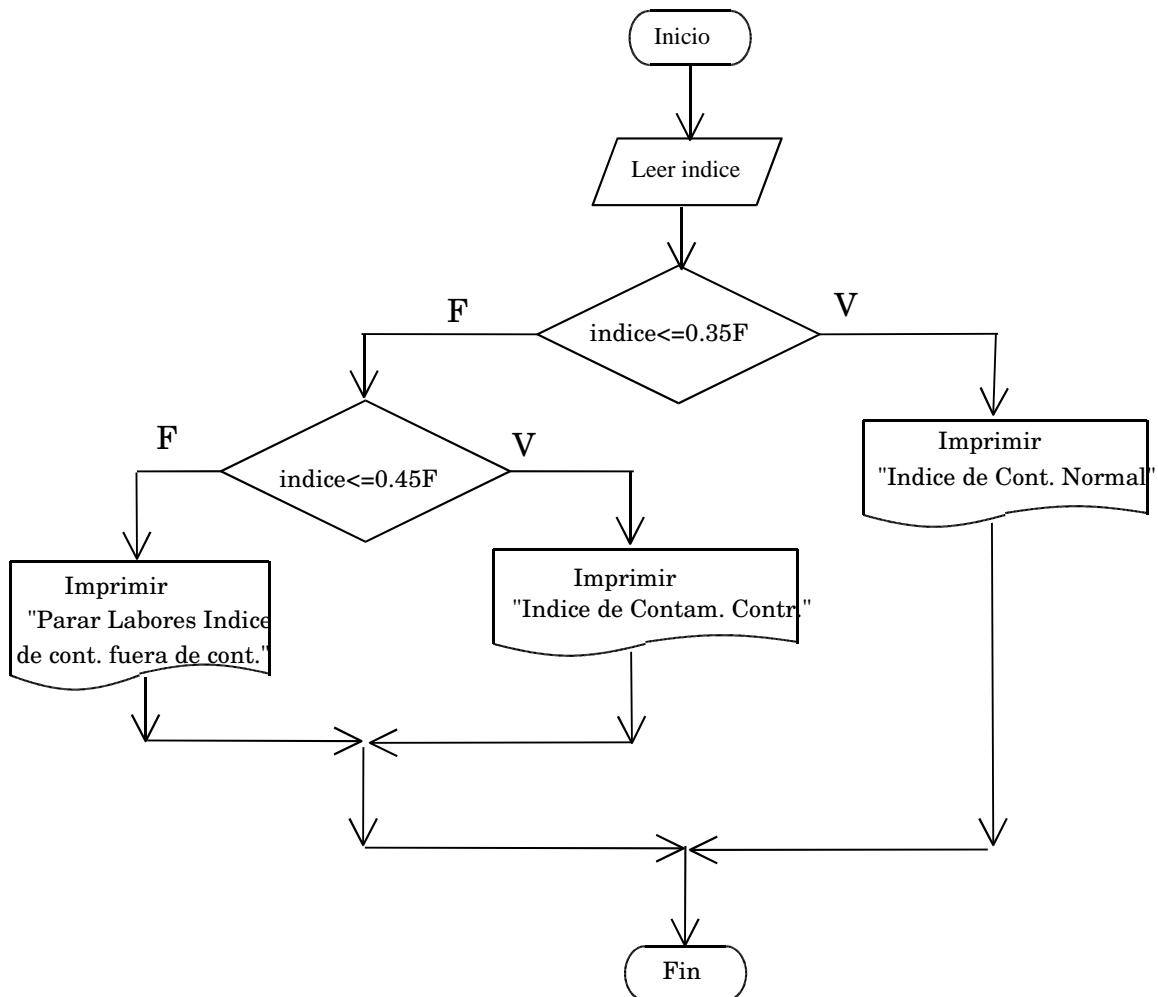
```

if(Expresión1)
  if(Expresión2)
    Sentencia3;
  else
    Sentencia2;
else
  Sentencia1;
  
```

Una selección anidada debe observar las mismas reglas que para las selecciones antes mencionadas; se debe tener cuidado con la sentencia else, esta cuando se utiliza hace referencia al if inmediato anterior. En el caso presentado el primer else hace referencia al último if anidado y el último else hace referencia al primer if.

Un ejemplo es el siguiente; se desea realizar un programa que lea el resultado del índice de contaminación de una empresa y se imprima un mensaje dependiendo del índice obtenido. Si los resultados arrojan un índice menor o igual a 0.35 se emite el mensaje de Índice normal de contaminación. Si el índice es mayor de lo normal pero inferior o igual a 0.45 se emite el mensaje Índice de contaminación superior a lo normal pero controlable; pero si el índice supera el límite de lo controlable se debe emitir un mensaje que indique para labores por que esta fuera de los límites controlados.

Fundamentos de programación orientada a objetos con Java



```

import java.lang.*;
import javax.swing.*;
public class Contaminacion {
    public static void main(String[] args) {
        float indice;
        String cadena;

        cadena= JOptionPane.showInputDialog("Dar indice de contaminación:");
        indice=Float.parseFloat(cadena);

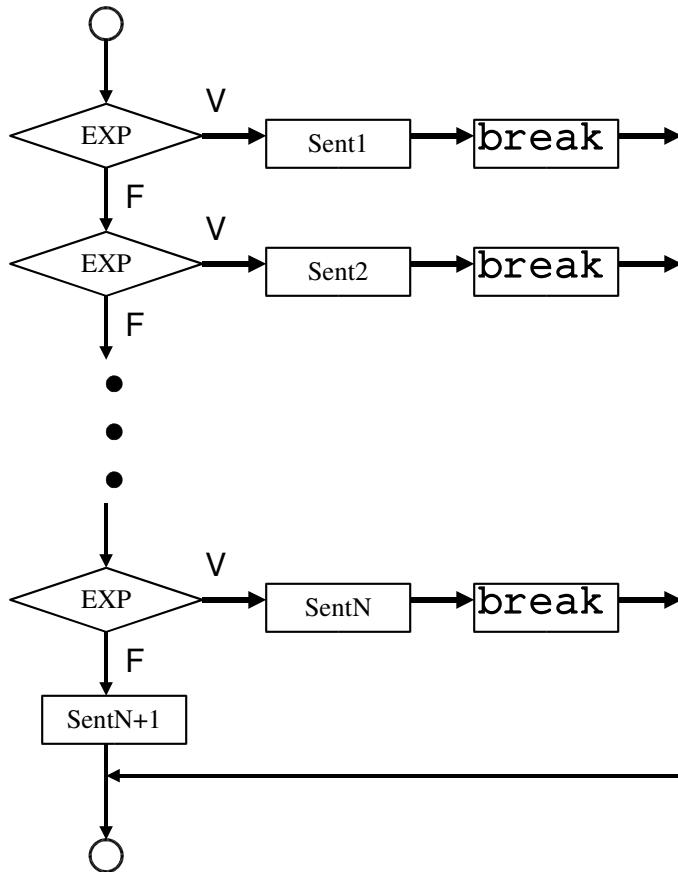
        if( indice <= 0.35F)
            System.out.println("Indice normal de contaminación");
        else if(indice <= 0.45F)
            System.out.println("Indice de contaminación controlable");
        else
            System.out.println("Parar labores: Indice de contaminación fuera de control");

        System.exit(0);
    }
}
  
```

Fundamentos de programación orientada a objetos con Java

Una selección multiple se utiliza para evaluar multiples selecciones sobre valores enteros o de tipo carácter(char), esta no se puede utilizar con algún otro tipo de dato.

Su descripción en un diagrama de flujo es el siguiente:



La sintaxis correspondiente en Java es la siguiente:

```
switch(EXP)
{
    case valor1: Sent1; break;
    case valor2: Sent2; break;
    :
    :
    case valorN: SentN; break;
    [default : SentN+1]
}
```

En este caso EXP, debe ser una expresión que retorne un valor entero (byte,

short o int no soporta valores long) o carácter (char). Los valores expresados en cada caso (case) deben ser del mismo tipo que EXP. La sentencia **break** hace que salte al final de **switch** la cláusula **default** es opcional y se utiliza para indicar si no es ningún caso anterior, se ejecuta la sentencia que está en esta sección y no requiere la sentencia **break**.

Muchas veces la selección múltiple se utiliza para revisar una variable entera o carácter y determinar las acciones a realizar; a diferencia de las anteriores selecciones si cualquiera de las sentencias en algún caso son más de una no se utilizan las llaves de ámbito ({ }).

Para exemplificar esta sentencia tenemos el siguiente programa, que lee un carácter e imprime si se trata de una vocal minúscula.

```
import java.util.*;
import java.lang.*;
import javax.swing.*;

public class vocal
{
    public static void main(String [] args){
        String cadena;
        char letra;
        cadena= JOptionPane.showInputDialog("Dar una letra");
        letra=cadena.charAt(0);
        switch(letra)
        {
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u': System.out.println(letra+" Es una vocal");break;
            default: System.out.println(letra+" No es una vocal");
        }
        System.exit(0);
    }
}
```

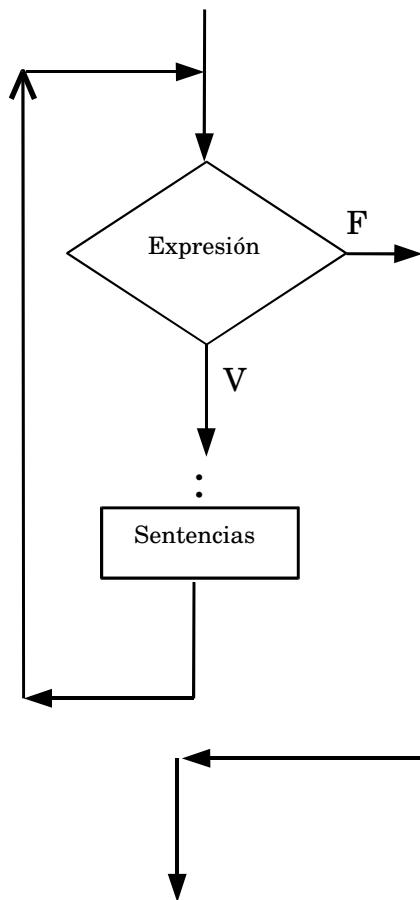
1.9.3 Iteración.

La interacción permite repetir una serie de instrucciones cuando así se requiera. Java ofrece tres clases de interacción; iteración con condición al inicio, iteración con condición al final e iteración con contador.

Iniciaremos con la iteración con condición al inicio; esta consiste en una selección donde si el resultado de la evaluación de la expresión es verdadera entra a ejecutar las sentencias que están dentro del ciclo y regresa a evaluar la expresión, termina de iterar cuando el resultado de la evaluación de la expresión

es falsa. Considerar que la expresión deberá cambiar de alguna forma dentro del cuerpo de la sentencia.

Su descripción en diagrama de flujo es la siguiente:



La sintaxis correspondiente en Java es la siguiente:

```
while(expresión)  
    sentencia;
```

```
while(expresión){  
    sentencia1;  
    sentencia2;  
    :  
    sentenciaN;  
}
```

Fundamentos de programación orientada a objetos con Java

El siguiente programa muestra como se usa esta sentencia; Calcula e imprime el monto a pagar de la cantidad leida de productos a comprar y su precio unitario, termina cuando se lee una clave de producto igual a -1.

```
import java.util.*;
import java.lang.*;
import javax.swing.*;

public class Tienda
{
    public static void main(String [ ] args){
        String cadena;
        int clave;
        float precio;
        int cantidad;
        float monto;
        clave=0;

        while(clave!=-1){
            cadena= JOptionPane.showInputDialog("Dar clave de producto:");
            clave=Integer.parseInt(cadena);

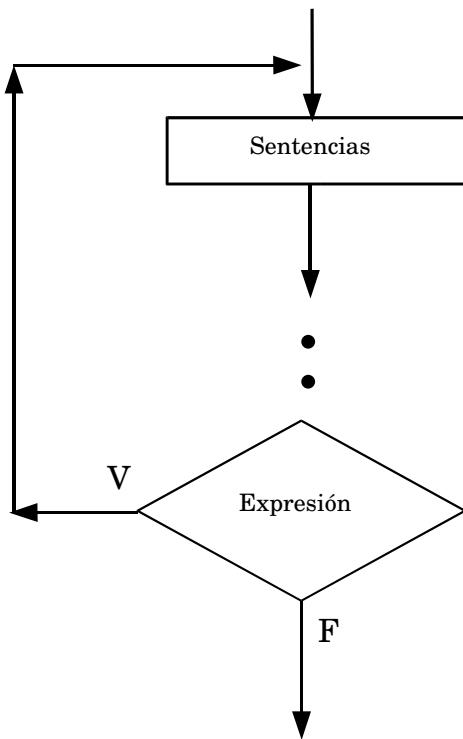
            if(clave!=-1){
                cadena= JOptionPane.showInputDialog("Dar cantidad a comprar:");
                cantidad=Integer.parseInt(cadena);
                cadena= JOptionPane.showInputDialog("Dar precio del producto:");
                precio=Float.parseFloat(cadena);
                monto=cantidad*precio;
                System.out.println("El monto a pagar del producto: "+clave);
                System.out.println("Cantidad comprada="+cantidad);
                System.out.println("Precio unitario =$"+precio);
                System.out.println("Monto ======>$"+monto);
            } // Fin de if

        }// Fin de while

        System.exit(0);
    } // Fin de main()
} // Fin de clase tienda
```

La iteración con condición al final por el contrario, tiene un cuerpo de iteración que al menos una vez queremos que se ejecute y al terminar tiene una selección con una expresión que se evalua, si el resultado es verdadero se repiten las instrucciones, en caso contrario termina de iterar.

Su descripción en diagrama de flujo es la siguiente:



La sintaxis correspondiente en Java es la siguiente:

```
do
    sentencia;
while(expresión);
```

```
do{
    sentencia1;
    sentencia2;
    :
    sentenciaN;
}while(expresión);
```

El siguiente programa muestra el mismo problema anterior pero resuelto con iteración con condición al final.

Fundamentos de programación orientada a objetos con Java

```
import java.util.*;
import java.lang.*;
import javax.swing.*;

public class Tienda2
{
    public static void main(String [ ] args){
        String cadena;
        int clave;
        float precio;
        int cantidad;
        float monto;

        do{
            cadena=JOptionPane.showInputDialog("Dar clave de producto:");
            clave=Integer.parseInt(cadena);

            if(clave!=-1){
                cadena=JOptionPane.showInputDialog("Dar cantidad a comprar:");
                cantidad=Integer.parseInt(cadena);
                cadena=JOptionPane.showInputDialog("Dar precio del producto:");
                precio=Float.parseFloat(cadena);
                monto=cantidad*precio;
                System.out.println("El monto a pagar del producto: "+clave);
                System.out.println("Cantidad comprada="+cantidad);
                System.out.println("Precio unitario =$"+precio);
                System.out.println("Monto ======>>$"+monto);
            } // Fin de if

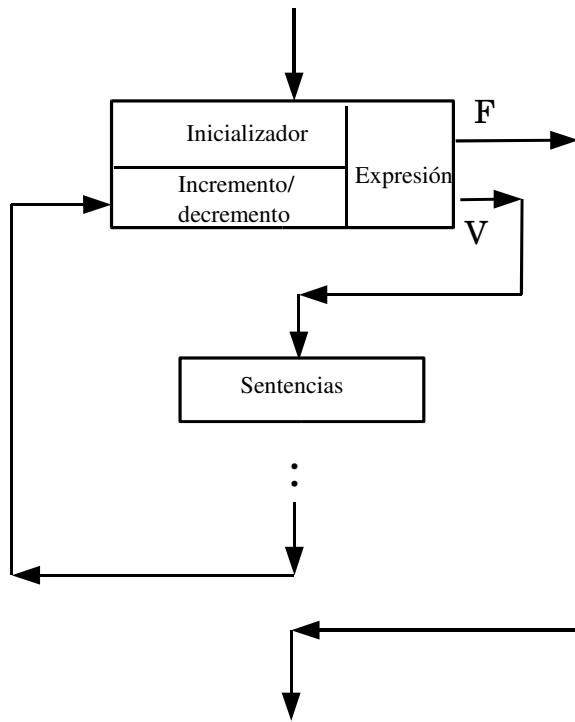
            }while(clave!=-1);// Fin de do-while

        System.exit(0);
    } // Fin de main()

} // Fin de clase tienda
```

La iteración con contador es una sentencia que repite una serie de operaciones utilizando un contador. El contador es una variable que se incrementa o decrementa de forma uniforme durante la ejecución de la sentencia según como sea la necesidad del programador.

Su descripción en diagrama de flujo es la siguiente:



La sintaxis correspondiente en Java es la siguiente:

```
for(inicializador; expresión; incremento/decremento)  
    Sentencia;
```

```
for(inicializador; expresión; incremento/decremento){  
    Sentencia1;  
    Sentencia2;  
    :  
    SentenciaN;  
}
```

El inicializador es una o varias sentencias de asignación que inicializan el o los contadores.

La expresión debe regresar un valor booleano que determinará si es verdadero que ejecute las sentencias de la iteración y en caso contrario termine.

El incremento/decremente es una o varias sentencias de asignación que indican de que forma se incrementan o decrementan los contadores.

Fundamentos de programación orientada a objetos con Java

El siguiente ejemplo muestra el uso de esta sentencia; el programa tabula los valores de la siguiente función matemática cuyos valores en x deberan ser 3,6,9,12,15,18, 21, 24, 27 y 30.

$$y = \frac{1}{x} + \frac{x}{3}$$

```
import java.util.*;
import java.lang.*;

public class Funcion
{
    public static void main(String [ ] args){
        float y;
        int x;

        for(x=3; x<=30; x=x+3){
            y=1.0F/x+x/3.0F;
            System.out.println(x+"\t"+y);
        } // fin de for

    } // fin de main
}
```

Se pueden utilizar las sentencias de iteración como elementos de otras sentencias incluso se pueden tener sentencias de iteración anidada, donde por ejemplo; se puede tener una sentencia de iteración con condición al inicio incluida dentro de una sentencia de iteración con contador, etc.

Las estructuras de control vistas hasta este momento pueden incluir declaración de variables lo cual les permite tener variables cuyo ámbito es la sentencia, por ejemplo:

```
do{
    int a;
    :
    :
}while(z!=10); // La variable a solo es visible dentro de la sentencia do.

while(true){
    float x; // La variable x solo es visible dentro de la sentencia while.
    :
}

for(int i=1; i<10;i++) // La variable i solo es visible dentro de la sentencia for.
    System.out.println(i*9);
```

```
if(a>10){  
    int vista; // La variable vista solo es visible en la sección verdadera de if.  
    :  
}
```

Existen dos sentencias que pueden alterar el funcionamiento de las sentencias de iteración; estas sentencias son *continue* y *break*.

La sentencia *break* rompe la ejecución de sentencias dentro de una sentencia de iteración haciendo que termine sin necesidad de evaluar la expresión condicional.

La sentencia *continue* forza a que se evalúe la expresión condicional que utilizan las sentencias *while*, *do* y *for*.

El siguiente ejemplo muestra el uso de la sentencia *continue*; resuelve el problema anterior validando que los valores en *x* no deban ser múltiplos de 4.

```
import java.util.*;  
import java.lang.*;  
  
public class Funcion2  
{  
    public static void main(String [ ] args){  
        float y;  
        int x;  
  
        for(x=3; x<=30; x=x+3){  
            if(x%4==0)  
                continue;  
            y=1.0F/x+x/3.0F;  
            System.out.println(x+"\t"+y);  
        } // fin de for  
    } // fin de main  
}
```

Capítulo II Arreglos unidimensionales y multidimensionales.

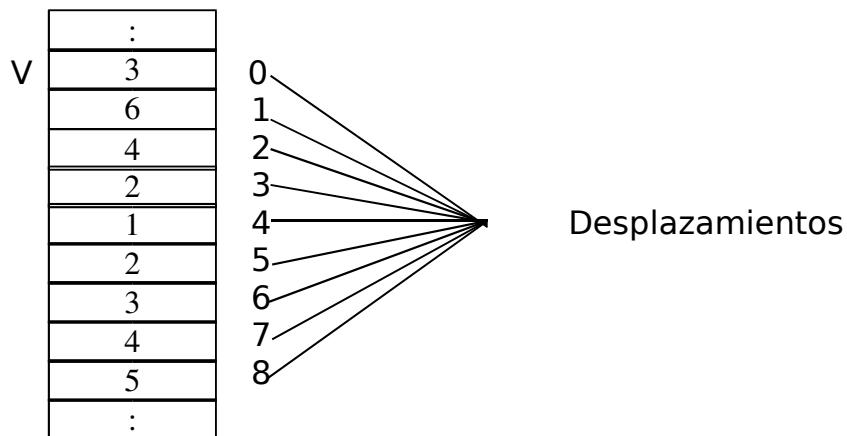
Los arreglos son un tipo de dato estructurado o también conocido como estructura de datos homogénea, ya que considera un conjunto de valores agrupados del mismo tipo de dato. Estos conjuntos de valores son de tamaño finito y pueden organizarse de diversas formas, como se verá a lo largo del capítulo.

2. 1 Arreglo Unidimensionales listas (vectores).

Un arreglo unidimensional es una estructura de datos lineal donde el conjunto de valores se encuentran organizados lógica y físicamente de la misma forma, a este tipo de arreglos también se les ha llamado vectores.

$$v=\{3,6,4,2,1,2,3,4,5\}$$

Dado v se dice que es un arreglo unidimensional o vector porque sus elementos pueden ser enumerados, el primer elemento es el 3, el segundo el 6, el tercero el 4, el cuarto el 2, el quinto el 1, el sexto el 2, el séptimo el 3, el octavo el 4 y el noveno el 5. Por lo tanto el arreglo se dice que es de nueve elementos y la referencia a cada uno de ellos se hace de la forma $v_0=3, v_1=6, v_2=4, v_3=2, v_4=1, v_5=2, v_6=3, v_7=4, v_8=5$. En el lenguaje Java siempre se hace referencia al elemento que se localiza en el desplazamiento 0 como el primer elemento, ya que dicho desplazamiento es relativo al origen del arreglo en la memoria.



2.1.1 Conceptos básicos.

Para declarar un arreglo unidimensional en Java se sigue cualquiera de las siguientes dos formas:

tipo de dato [] identificador;

tipo de dato identificador [];

Donde

tipo de datos : Puede ser cualquier tipo de dato básico (int, byte, short, long, float, double, boolean o char) o algún tipo definido por el programador (clase, estructura de datos, etc.)

identificador: Es el nombre con el cual el arreglo se referenciará.

Ejemplos:

```
int [ ] arreglo;           // Arreglo de valores int  
float arregloFloat [ ];   // Arreglo de valores float  
long arregloLong [ ];     // Arreglo de valores long  
Persona [ ] arregloP;    // Arreglo de objetos de la clase Persona
```

Un arreglo en Java en su declaración no se indica el tamaño que utilizará, este se especifica cuando se crea; los arreglos en Java siempre es necesario crearlos. Para crear un arreglo se pueden seguir tres notaciones:

```
int [ ] calificaciones={79,90,78,90,100}; // Asignando valores en la declaración
```

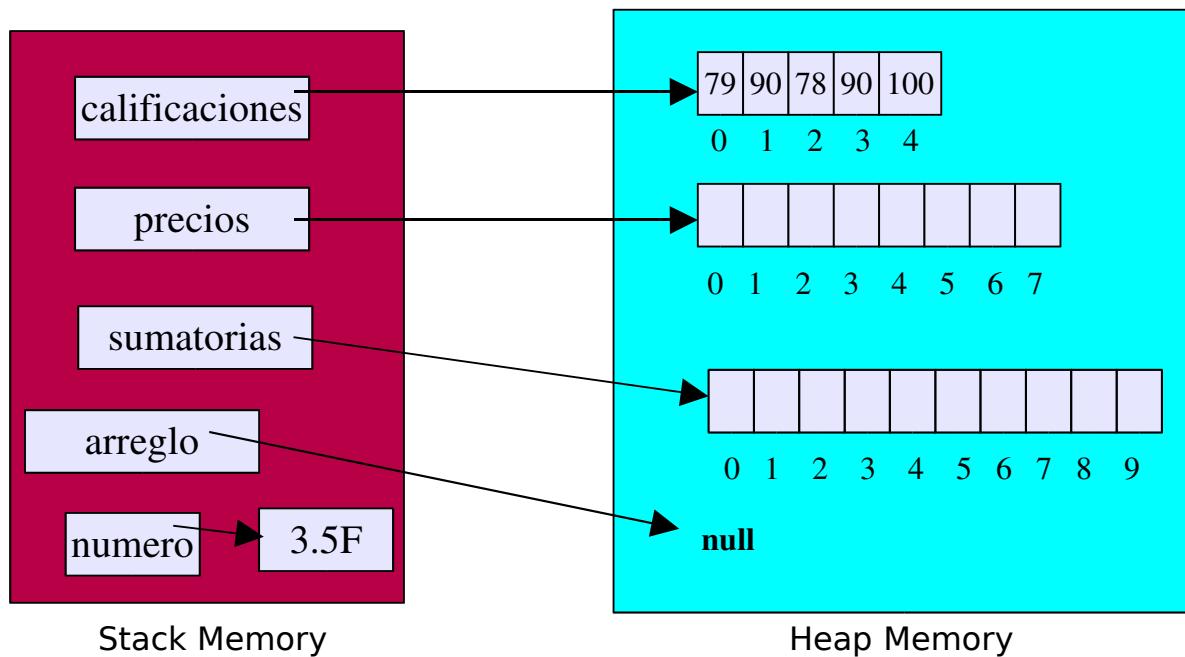
```
float [ ] precios;  
precios= new float[8]; // Creando el arreglo con el operador new
```

```
int [ ] sumatorias= new int[10]; // Crearlo junto con la declaración
```

```
short [ ] arreglo;  
float numero=3.5F;
```

La memoria en Java, se divide en dos secciones, Memoria de pila (stack memory) y Memoria de montículo (heap memory). Stack memory almacena todas las declaraciones de variables y constantes de un programa así como las localidades de memoria donde se almacenan los valores de variables de tipos de datos básicos. Heap memory almacena todos los objetos y valores creados con el

operador new, lo cual incluye a los valores de un arreglo, este se puede ver de la siguiente forma:



Como se puede observar en la figura, todos los elementos de un arreglo se almacenan en heap memory y las variables declaradas, en stack memory. Mientras un arreglo no se ha creado (En este ejemplo la variable nombrada arreglo) hace referencia al valor **null** indicando que no tiene un arreglo asignado.

En Java los arreglos son tratados como objetos por lo que también se les llama referencias; estos proveen un dato miembro que puede ser utilizado para referirse al tamaño del arreglo, llamado length.

Ejemplos:

```
int x;
int [ ] precios= new int[10];
x= precios.length; // El tamaño del arreglo se asigna a una variable int.

for(int i=0;i<precios.length;i++) //Se utiliza el dato length, para iterar
```

2.1.2 Operaciones.

Una vez declarado y creado un arreglo este puede utilizarse para realizar operaciones de guardar, recuperar, buscar, recorrer o lo que se guste realizar con él, según el programa donde se este utilizando y lo que deseemos que

representen sus valores. Por ejemplo podemos utilizar un arreglo para representar:

```
int [ ] edades;           // Edades de los alumnos del grupo
float [ ] pesos;          // Pesos de los alumnos del grupo
String [ ] nombres;        // Nombres de los alumnos del grupo
boolean [ ] correos_leidos; // Si un correo ha sido leido o no (true o false)
char [ ] simbolos;         // Simbolos de un alfabeto
long[ ] telefonos;        // Telefonos de los amigos
```

La siguiente lista muestra algunas de las operaciones más comunes que con un arreglo se pueden realizar:

- 1.- Creación del arreglo.
- 2.- Llenado del arreglo.
- 3.- Sumar los elementos de un arreglo numérico.
- 4.- Promediar los elementos de un arreglo numérico.
- 5.- Obtener cuantos elementos no se repiten en un arreglo.
- 6.- Obtener si un valor se encuentra dentro del arreglo.
- 7.- Obtener cuantas veces un valor se localiza dentro del arreglo.
- 8.- Destruir un arreglo.
- 9.- Obtener el numero de elementos que son pares en un arreglo numérico.
- 10.- Obtener el valor máximo o el mínimo de un arreglo.
- 11.- Obtener cuantas palabras tiene un arreglo de caracteres.

El llenado de un arreglo de esta clase, se realiza normalmente elemento por elemento iniciando en el primer elemento (Posición 0), para esto se utiliza un índice que indicará el elemento que será referenciado del arreglo, este índice puede ser una literal, constante, variable o expresión que retorne un valor entero. La forma de referenciar cada elemento utiliza la siguiente sintaxis:

nombreDelArreglo [índice]

Llenar un arreglo de flotantes que representan las 5 calificaciones de un alumno en una materia imprimiendo estas de la primera a la última y de la última a la primera. Se resuelve con el siguiente código:

```
import java.lang.*;
import javax.swing.*;

public class llenado {
    public static void main(String[] args) {
        float calificaciones[ ];
        int i;
        calificaciones = new float[5];
```

Fundamentos de programación orientada a objetos con Java

```
for(i=0;i<calificaciones.length;i++){
    String entrada= JOptionPane.showInputDialog("Dar calificacion "+(i+1));
    calificaciones[i]=Float.parseFloat(entrada);
}

System.out.println("Calificaciones de la primera a la ultima");

for(i=0;i<calificaciones.length;i++)
    System.out.println(calificaciones[i]);

System.out.println("Calificaciones de la ultima a la primera");
for(i=calificaciones.length;i>0;i--)
    System.out.println(calificaciones[i-1]);
System.exit(0);
}
```

En este ejemplo se muestra como recorrer el arreglo de inicio a fin y de fin a inicio, muchas de las operaciones requiere que se recorra en arreglo en cualquiera de los dos sentidos.

Otra de las operaciones que con un arreglo se pueden realizar son buscar y recorrer, estas operaciones son muy útiles para localizar y obtener información que se requiere del arreglo, por ejemplo si deseamos saber cual es la calificación máxima que obtubo el alumno se requiere recorrer todos los elementos del arreglo para localizar este valor, los métodos que se utilizan normalmente son diseñados por el programador. Para esto debemos diseñar un algoritmo que represente dicho método.

El método a utilizar que propondremos para buscar la calificación mayor es muy simple, supondremos que partiendo del primer elementos preguntamos si los elementos restantes son mayores que el y utilizando un área temporal para ir guardando el que nos vaya quedando como mayor.

Suponer :

```
float mayor; // variable temporal
mayor= calificaciones [0]; // Tomamos al primero en el área temporal como si fuera el mayor
for(i=1;i<calificaciones.length;i++) // El ciclo lo iniciamos con el segundo elemento
    if(calificaciones[i]>mayor) // Preguntamos si existe alguno que sea mayor que el anterior
        mayor=calificaciones[i]; // Si es verdad sustituimos el mayor por el nuevo
```

Cuando termina el ciclo se realizó la pregunta por todos los elementos del arreglo y obtiene en la variable mayor el mayor.

Para localizar el menor se tiene que realizar lo mismo pero con el

condicionante inverso (<).

```
float menor; // variable temporal  
menor= calificaciones [0]; // Tomamos al primero en el área temporal como si fuera el menor  
  
for(i=1;i<calificaciones.length;i++) // El ciclo lo iniciamos con el segundo elemento  
if(calificaciones[i]<mayor) // Preguntamos si existe alguno que sea menor que el anterior  
    mayor=calificaciones[i]; // Si es verdad sustituimos el menor por el nuevo
```

El programa completo quedaría de la siguiente forma:

```
import java.lang.*;  
import javax.swing.*;  
  
public class MaxMin {  
  
    public static void main(String[] args) {  
        float calificaciones[ ];  
        float mayor,menor;  
        int i;  
  
        calificaciones = new float[5];  
  
        for(i=0;i<calificaciones.length;i++){  
            String entrada= JOptionPane.showInputDialog("Dar calificacion "+(i+1));  
            calificaciones[i]=Float.parseFloat(entrada);  
        }  
  
        System.out.println("Calificacion Maxima");  
        mayor=calificaciones[0];  
  
        for(i=1;i<calificaciones.length;i++)  
            if(calificaciones[i]>mayor)  
                mayor=calificaciones[i];  
  
        System.out.println("Maxima="+mayor);  
  
        System.out.println("Calificacion Minima");  
        menor=calificaciones[0];  
  
        for(i=1;i<calificaciones.length;i++)  
            if(calificaciones[i]<menor)  
                menor=calificaciones[i];  
  
        System.out.println("Minima="+menor);  
        System.exit(0);  
    }  
}
```

EJERCICIOS:

- a) Diseña una forma diferente al descrito para obtener el mayor y el menor.
- b) Que modificaciones tendrías que realizar al programa para que imprimiera también que posiciones del arreglo tienen la calificación máxima y la calificación mínima
- c) ¿Podrías en un mismo ciclo obtener los valores mínimos y máximos? describe el programa como debería ser.
- d) Realiza un programa que obtenga cuantas veces se repite la calificación máxima y cuantas la calificación mínima.

2.1.3 Aplicaciones.

Las aplicaciones de los arreglos unidimensionales son muy diversas y su uso estará determinado por la capacidad el programador para describir la información de un problema en un arreglo.

Dado un vector de 10 elementos enteros, obtener e imprimir el promedio de ellos.

```
import java.lang.*;  
  
public class aplica1{  
    public static void main(String [ ] args){  
        int vector[ ] ={7,8,4,5,9,3,4,7,8,10};  
        int i;  
        int suma=0;  
        float promedio=0;  
  
        for(i=0;i<vector.length;i++)  
            suma=suma+vector[i];  
  
        promedio=suma/(float)vector.length;  
  
        System.out.println("Promedio="+promedio);  
    }  
}
```

Declara arreglo de 10 elementos y lo inicializa.
Declara indice para recorrer el vector
Declara acumulador para hacer la sumatoria
Declara donde se guardara el promedio calculado

Ciclo para recorrer el arreglo y realizar la sumatoria

Calcula el promedio

Imprime el resultado

Dado un vector de 15 elementos de punto flotante leidos desde el teclado, determine e imprima la mayor diferencia entre 2 elementos consecutivos.

<pre>import java.lang.*; public class aplica2{ public static void main(String [] args){ float vector[]= new float[15]; int i; float diferencia=0; float mayorD=0; for(i=0;i<vector.length;i++){ String aux= JOptionPane.showInputDialog("Dar numero"); vector[i]=Float.parseFloat(aux); } mayorD=vector[0]-vector[1]; for(i=1;i<vector.length-1;i++){ diferencia=vector[i]-vector[i+1]; if(diferencia > mayorD) mayorD=diferencia; } System.out.println("La mayor diferencia es="+mayorD); System.exit(0); } }</pre>	<p>Declara vector Declara indice Declara variable para diferencia Declara variable para mayor diferencia. Ciclo para leer el vector</p> <p>Calcula diferencia entre los 2 primeros elementos.</p> <p>Calcula diferencia elementos Si diferencia > que la anterior Sustituir la anterior</p> <p>Imprime diferencia.</p>
---	---

Dado un vector de 10 elementos enteros, imprima el vector sin los elementos que se encuentren repetidos y el total de valores que se imprimieron.

<pre>import java.lang.*; public class aplica3{ public static void main(String[] args){ int vector []= {4,5,4,3,6,5,7,2,3,9}; int aux[]= new int[10]; int i,j; int k=0; boolean band; for(i=0;i<vector.length;i++){ band=false; for(j=0;j<k;j++) if(vector[i]== aux[j]){ band=true; break; } if(! band){ aux[k]=vector[i]; k++; System.out.println(vector[i]); } } System.out.println("Total de elementos="+k); } }</pre>	<p>Declara el vector con valores iniciales. Declara vector auxiliar para guardar impresos Declara indices para recorrer vectores Declara indice para controlar valores impresos Bandera para buscar si se repito un valor.</p> <p>Ciclo para recorrer el arreglo original Inicializa bandera antes de buscar si se repite Ciclo para buscar en auxiliar si se repite Si lo encuentra cambia la bandera a true y sale del ciclo interno</p> <p>Si no lo encontro, lo guarda en el vector auxiliar, incrementa k y lo imprime</p> <p>Al terminar el ciclo externo en k tiene cuantos elementos imprimio.</p>
--	---

<p><i>Dado un vector de 10 caracteres, obtener e imprimir el numero de vocales que la componen.</i></p>	
<pre>import java.lang.*; public class aplica4{ public static void main(String [] args){ char [] vector={'r','i','t','r','e','a','i','y','o','p'}; int i; int vocales=0; for(i=0;i<vector.length;i++) switch(vector[i]){ case 'a': case 'e': case 'i': case 'o': case 'u': vocales++; } System.out.println("Vocales="+vocales); } }</pre>	<p>Declaración del vector Declaración del indice para recorrer el vector Declaración del contador de vocales</p> <p>Ciclo para recorrer el arreglo Pregunta si es una vocal para incrementar el contador.</p> <p>Imprime el contador de vocales.</p>

EJERCICIOS:

a) Dados 2 vectores de numeros enteros del mismo tamaño, sumar los i-enesimos elementos de ambos y guardar el resultado en otro vector, $c[0]=a[0]+b[0], \dots, c[n]=a[n]+b[n]$.

b) Dado 2 vectores de numeros enteros de distintos tamaños combinar los elementos en otro vector siguiendo la regla que para combinar $c[k] = \text{mayor}(a[i], b[j])$, esto quiere decir que un elemento para guardarse en el vector resultante, debe ser el mayor de los que quedan en los vectores, al final el vector resultante se deberá imprimir.

$$\begin{aligned} a &= \{5, 7, 3, 7, 2\} \\ b &= \{8, 3, 4, 6, 8, 9, 3, 2\} \end{aligned}$$

$$c = \{8, 5, 7, 3, 7, 3, 4, 6, 8, 9, 3, 2, 2\}$$

c) Dado un vector de n valores flotantes, hacer un programa que determine e imprima cual es la primera y última ocurrencia de un valor dentro del vector.

d) Describe como representarias la ocupación de los asientos de un autobus con 42 asientos.

- e) Tomando la representación descrita anteriormente, describe en Java el código para obtener cuantos lugares estan ocupados.
- f) Un doctor trabaja 9 horas atendiendo consultas solicitadas por sus pacientes, cada consulta le lleva 30 minutos como máximo, describe como representarias las consultas que tiene que atender en un día.
- g) Tomando la representación anterior, describe en Java el código para obtener cuantas citas tiene disponible el doctor en el día, y que no han sido reservadas.
- h) El IFE ha decidido hacer un recuento de cuantos ciudadano existen de cada edad entre los 18 y 35 años, para graficar dichos valores. Se necesita un programa en Java que lea la edad de cada ciudadano y al terminar de leer la edad del ultimo ciudadano imprima en forma horizontal una gráfica utilizando *, cuantos ciudadanos existen de cada edad.

2.2 Arreglo bidimensional.

Un arreglo bidimensional es un conjunto de arreglos unidimensionales, los cuales se pueden especificar de la forma:

A={ {5,8,3},{6,7,8},{9,3,2} }

o de la forma:

```
A={ {5,8,3},  
     {6,7,8},  
     {9,3,2}  
 }
```

Dado que A es bidimensional se puede decir que esta compuesto de 3 elementos y cada uno de ellos a su vez son arreglos de 3 elementos enteros. Se puede decir que $\text{arreglo}_0=\{5,8,3\}$, $\text{arreglo}_1=\{6,7,8\}$ y $\text{arreglo}_2=\{9,3,2\}$. Así también podemos decir que $\text{arreglo}_{00}=5$, $\text{arreglo}_{01}=8$, $\text{arreglo}_{02}=3$, $\text{arreglo}_{10}=6$, $\text{arreglo}_{11}=7$, $\text{arreglo}_{12}=8$, $\text{arreglo}_{20}=9$, $\text{arreglo}_{21}=3$ y $\text{arreglo}_{22}=2$.

Esto quiere decir que los arreglos dibimensionales pueden ser tratados como arreglos de otros arreglos o sus componentes como elementos individuales. Los arreglos bidimensionales se almacenan de forma secuencial a diferencia de su representación lógica. A esta clase de arreglos tambien se les llega a llamar matrices, por el concepto matématico que se utiliza para representarlos

Lógicamente:

$$A = \begin{bmatrix} 5 & 8 & 3 \\ 6 & 7 & 8 \\ 9 & 3 & 2 \end{bmatrix}$$

Una matriz tiene renglones y columnas, normalmente para identificar un elemento de la matriz se hace referencia al renglon y la columna donde se ubica, tanto los renglones como la columna se numeran de 0 en adelante.

Así tenemos que el elemento en el renglon 1 columna 2 es el 8, el elemento en el renglon 0 columna 2 es el 3, el elemento en la columna 1 renglon 1 es el 7, etc.

Las matrices tienen ciertas propiedades matemáticas, se dice que una matriz que tiene el mismo numero de renglones y columnas es cuadrada, el ejemplo anterior muestra una matriz cuadrada de 3 renglones por 3 columnas.

En una matriz cuadrada se dice que todos los elementos que están en la posición con el mismo numero de columna y renglon se les denomina diagonal de la matriz. En el ejemplo (5,7 y 2).

Una matriz se le denomina triangular superior si es una matriz cuadrada y todos sus elementos por encima de la diagonal incluyendo los elementos de la diagonal son diferentes de 0 y los elementos por debajo son igual a 0.

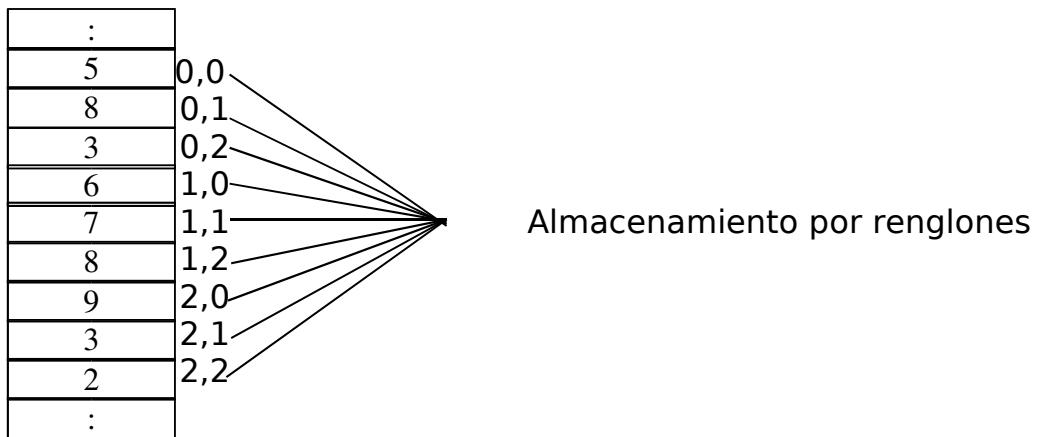
$$a = \begin{bmatrix} 4 & 5 & 9 & 5 \\ 0 & 6 & 7 & 4 \\ 0 & 0 & 5 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Una matriz se le denomina triangular inferior si es una matriz cuadrada y todos sus elementos por debajo de la diagonal incluyendo los elementos de la diagonal son diferentes de 0 y los elementos por encima son igual a 0.

$$a = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 7 & 4 & 0 & 0 \\ 8 & 9 & 4 & 0 \\ 5 & 3 & 2 & 1 \end{bmatrix}$$

Un arreglo bidimensional se puede representar almacenado por renglones o

por columnas, en la siguiente figura se muestra como se representa lógicamente almacenada una matriz por renglones.



2.2.1 Conceptos básicos.

En java los arreglos bidimensionales se declaran utilizando dos especificadores de dimensión:

tipo de dato [][] identificador;

tipo de dato identificador[][];

Se sigue las mismas especificaciones de los arreglos unidimensionales, la memoria cuando se crea un arreglo de este tipo se ve de la forma siguiente:

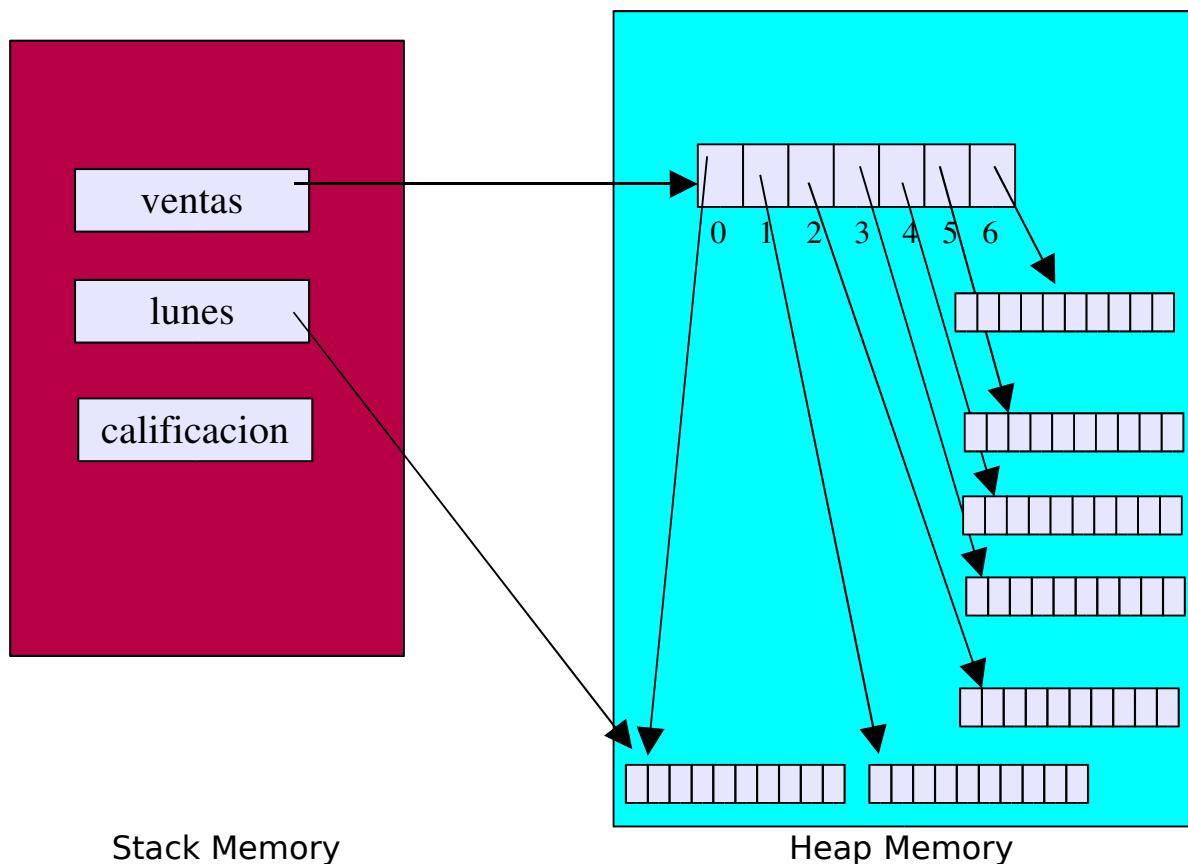
```
float [ ][ ] ventas;  
float [ ] lunes;
```

```
int [ ] [ ] calificacion;
```

```
ventas= new float[7][10];
```

```
calificaciones = new int [5][3];
```

```
lunes=ventas[0];
```



Este esquema muestra como se representa `ventas` en este esquema de memoria. Como se puede observar, `lunes` es un arreglo unidimensional y se le asigna directamente el primer renglon de la matriz llamada `ventas`. Por lo que tanto el primer renglon de `ventas` (`ventas[0]`) y `lunes` hacen referencia a los mismos datos. No se realizan copias de un arreglo a otro, cuando un arreglo se asigna a otro los dos arreglos hacen referencia a los mismo contenidos y no existen copias de unos con otros.

EJERCICIO:

- Elabora un esquema de memoria para representar la variable `calificaciones`.
- Elabora un esquema para representar las siguientes declaraciones

```
ventas= new float[1][12];
lunes = new float [4];
```

Los arreglos bidimensionales, necesitan de 2 indices para referenciar a cada uno de los elementos de la matriz y estos deben de ser enteros, siguiendo las

Fundamentos de programación orientada a objetos con Java

mismas características de los arreglos unidimensionales. Cuando se hace referencia a cada elemento por separado, para el llenado del arreglo, se deben utilizar dos ciclos, uno para llevar el control de los renglones y otro para el control de las columnas.

Suponiendo que los elementos del arreglo de *ventas*, representan las ventas realizadas durante la semana de 10 productos, cada renglon representa los días de la semana (0- Lunes, 1-Martes, 2-Miercoles, 3-Jueves, 4-Viernes, 5-Sabado y 6-Domingo) y las columnas los productos. Realizar un programa que lea las ventas totales realizadas por cada producto por cada día de la semana, y obtener e imprimir, el monto total de ventas realizadas el día Lunes de todos los productos y el promedio de ventas de la semana del producto 3.

```
import javax.swing.*;
import java.lang.*;
public class tienda {
    public static void main(String[] args) {
        float ventas[ ][ ];
        int r,c;
        float monto,promedio;
        String entrada;
        ventas= new float[7][10];
        for(r=0;r<ventas.length;r++)
            for(c=0;c<ventas[r].length;c++){
                entrada= JOptionPane.showInputDialog("Dar venta del día "+(r+1)+" del producto "+(c+1));
                ventas[r][c]=Float.parseFloat(entrada);
            }
        monto=0;
        for(c=0;c<ventas[0].length;c++)
            monto=monto+ventas[0][c];
        System.out.println("Monto total de ventas del dia Lunes de todos los productos="+monto);

        monto=0;
        for(r=0;r<ventas.length;r++)
            monto=monto+ventas[r][2];
        promedio=monto/ventas.length;

        System.out.println("El promedio de ventas del producto 3 en la semana =" +promedio);
        System.exit(0);
    }
}
```

EJERCICIOS:

- Modifica el programa para obtener e imprimir las ventas totales por día

de la semana de todos los productos.

- b) Modifica el programa para obtener el monto más bajo de las ventas por cada productos.

2.2.2 Operaciones

Un arreglo bidimensional, se utiliza para representar una gran variedad de tipos de información; se puede representar la información de un tablero de ajedrez, de un tablero de damas inglesas, la información de los alimentos de distintas especies en un zoológico, la información de un sistema de ecuacione, la información de una imagen, etc.

```
int [ ][ ] tablero;
int [ ][ ] colores;
float CantidadesAlimento[ ][ ];
char [ ][ ] damas;
```

Las operaciones sobre un arreglo de este tipo siempre estará en función de la información que se almacena en el y la utilidad que tendrá. Algunos ejemplos son:

- 1.- Buscar cuantos puntos de una imagen tienen un color específico.
- 2.- Cambiar los puntos verdes de una imagen por puntos rojos.
- 3.- Buscar donde esta el rey de color blanco dentro de un tablero.
- 4.- Buscar si una posición específica esta ocupada por una ficha.
- 5.- Sumar una columna específica.
- 6.- Promediar los elementos de una fila.
- 7.- Imprimir el arreglo por filas.
- 8.- Separar las filas de un arreglo bidimensional en elementos unidimensionales.
- 9.- Recorrer la diagonal de una matriz cuadrada para determinar su promedio.

Por ejemplo, suponer que tenemos un tablero de damas inglesas, representado por:

```
char [ ][ ] damas={{'n','v','n','v','n','v','n','v'},
{'v','n','v','n','v','n','v','n'},
{'v','v','v','v','v','v','v','v'},
{'v','v','v','v','v','v','v','v'},
{'v','v','v','v','v','v','v','v'},
{'v','v','v','v','v','v','v','v'},
{'v','v','v','v','v','v','v','v'},
{'b','v','b','v','b','v','b','v'},
{'v','b','v','b','v','b','v','b'}};
```

Queremos imprimir las posiciones donde se encuentran las fichas blancas que estan representadas por una 'b', el código para llevar a cabo esto será el siguiente:

```
for(int i=0;i<damas.length;i++)
    for(int j=0;j<damas[i].length;j++)
        if(damas[i][j]== 'b')
            System.out.println((""+i+"," +j+"));
```

Otra operación sería, dada la posición de una ficha negra, indicar si existe un movimiento que tenga una ficha que le permita saltarla.

```
int y=0; // Posición de la ficha en el tablero
int x=2;
char f=damas[y+1][x-1];
if(f != 'v')
    System.out.println("Existe ficha que saltar adelante y a la derecha");
f=damas[y+1][x+1];
if(f != 'v' )
    System.out.println("Existe ficha que saltar adelante y a la izquierda");
```

2.2.3 Aplicaciones

Al igual que los arreglos unidimensionales, los bidimensionales se aplican en una variedad muy diversa de aplicaciones y todo queda a la creatividad del programador para representar la información de un problema.

Fundamentos de programación orientada a objetos con Java

Dado un arreglo que representa los litros de leche que 4 vacas (Abeta,Ninfa,Bhel y Halfa) han dado en 5 días de la semana, entregar un reporte indicando cuales de ellas han sido las que menos leche han dado en cualquier día.

<pre>public class estable{ public static void main(String[] args){ int leche[][] ={{6,5,4,7,2}, {3,5,4,3,4}, {2,4,5,7,8}, {5,4,6,3,6}}; String [] vaca={"Abeta","Ninfa",Bhel","Halfa"}; int menor=10; int v,d; for(v=0;v<leche.length;v++) for(d=0;d<leche[v].length;d++) if(leche[v][d]<menor) menor=leche[v][d]; for(v=0;v<leche.length;v++) for(d=0;d<leche[v].length;d++) if(leche[v][d]== menor){ System.out.println(vaca[v]); break; } } }</pre>	<p>Define el arreglo que representa los litros de cada vaca en cada día de la semana. Los renglosnes indican la vaca y las columnas los dias.</p> <p>Define arreglo con los nombres de las vacas.</p> <p>Indica un numero mayor que los valores indicados</p> <p>Declara indices v(vacas) y d (dias) Ciclos para encontrar cual es el valor menor de litros dados por cualquier vaca en cualquier dia de la semana.</p> <p>Recorre arreglo para encontrar al menos una ocurrencia, para cada vaca, con el valor menor para imprimir el nombre correspondiente.</p>
---	--

Fundamentos de programación orientada a objetos con Java

Se ha realizado una serie de encuestas en la ciudad, sobre la preferencia de implementar el hoy no circula(sí o no), las personas encuestadas son mayores de edad entre 20 y 45 años. Diseñe un programa en Java que lea la edad y respuesta sobre la pregunta de la encuesta. El programa deberá terminar cuando se lea una edad igual a -1, cuando esto suceda deberá imprimir en forma tabular los resultados:

<pre>import javax.swing.*; public class encuesta { public static void main(String[] args) { int[][] resultado; int edad = 0; String resp; int i, j; resultado = new int[26][2]; do { resp = JOptionPane.showInputDialog("Dar edad de encuestado"); edad = Integer.parseInt(resp); if (edad >= 20 && edad <= 45) { resp = JOptionPane.showInputDialog("Desea el programa[s/n]:"); if (resp.compareTo("s") == 0) resultado[edad - 20][0]++; else resultado[edad - 20][1]++; } } while (edad != -1); System.out.println("Edad : SI NO"); System.out.println("====="); for(i=0;i<resultado.length;i++){ System.out.print((i + 20) + " : "); for (j = 0; j < resultado[i].length; j++) System.out.print(resultado[i][j] + " "); System.out.println(); } System.exit(0); } }</pre>	<p>Declara arreglo Declara variable para edad Declara indices Crea arreglo 26x2 Lee la edad Lee la respuesta Incrementa si es Si Incrementa si es No Termina cuando es -1 Imprime tabularmente</p>
--	--

EJERCICIOS:

- Modificar el programa del estable para que imprima los totales semanales de cada Vaca y cuales de ellas fueron las que más leche produjeron.
- Modifica el programa anterior para que también imprima el total de personas a favor y en contra.
- Modificar el programa anterior para que imprima una gráfica tabular horizontal con asteriscos(*).

2.3 Arreglos multidimensionales

La complejidad de los arreglos aumenta a medida que se incrementan sus dimensiones, su utilidad depende de la capacidad de abstracción del programador, hoy en día los arreglos multidimensionales permiten representar información para la toma de decisiones, el problema es como interpretarlos.

2.3.1 Conceptos básicos

En Java los arreglos multidimensionales se declaran con tantas dimensiones como quisieramos representar, al final de cuentas todos son vectores de arreglos.

```
int [ ][ ][ ] costos; // Arreglo de 3 dimensiones  
  
boolean [ ][ ][ ][ ] pruebas; // Arreglo de 4 dimensiones  
  
float [ ][ ][ ] ventas; // Suponer que son las ventas de un coorporativo  
  
// La primera dimension representa la región (América, Asia o Europa)  
// La segunda el producto (Tabcin, CitroC, HidroThe o GraC)  
// La tercera el tipo de venta (Mayoreo o menudeo)  
  
ventas = new float [3][4][2];
```

EJERCICIO:

- Dibujar el mapa de memoria del arreglo ventas.
- Hacer un programa que llene los elementos del arreglo ventas solicitando los datos desde el teclado

2.3.2 Operaciones

Las operaciones de los arreglo multidimensionales dependen de la naturaleza de la aplicación, sin embargo, también con ellos se pueden realizar busquedas, clasificar, separar, sumarizar, promediar y todo aquello que se nos pudiera ocurrir.

2.3.3 Aplicaciones

Como se expreso al principio una de las cualidades de los arreglos multidimensionales, es que ayudan a la toma de decisiones al manipularlos utilizando una o varias de sus dimensiones.

Tomando como ejemplo el arreglo ventas comentado anteriormente imprimir en orden ascendente las regiones, por el monto de ventas al mayoreo de mayor a menor.

<pre> public class regiones { public static void main(String[] args) { int ventas[][]={ {{400,300},{200,200},{300,100},{100,200}}, {{500,300},{200,250},{345,300},{400,356}}, {{345,450},{240,500},{45,125},{345,323}} }; String regiones[]={"America","Asia","Europa"}; int sumas[]= new int[3]; int r,p,t; for(r=0;r<ventas.length;r++) for(p=0;p<ventas[r].length;p++) sumas[r]= sumas[r]+ventas[r][p][0]; int mayor=sumas[0]; int m[]={0,-1,-1}; for(r=1;r<sumas.length;r++){ if(sumas[r]>mayor){ mayor=sumas[r]; m[0]=r; } if(m[0]==0){ if(sumas[1]>sumas[2]){ m[1] = 1; m[2] = 2; } else{ m[1]=2; m[2]=1; } }else if(m[0]==1){ if(sumas[0]>sumas[2]){ m[1]=0; m[2]=2; }else{ m[1]=2; m[2]=0; } }else{ if(sumas[0]>sumas[1]){ m[1]=0; m[2]=1; }else{ m[1]=1; m[2]=0; } } } for(r=0;r<regiones.length;r++) System.out.println(regiones[m[r]]+" = "+sumas[m[r]]); } } </pre>	<p>Declaración de arreglo; region, producto y tipo de venta.</p> <p>Declarar regiones Declarar sumas</p> <p>Calcular montos de ventas por mayoreo por region.</p> <p>Calcular la region mayor y poner indice en arreglo m.</p> <p>Calcular indice de segundo y tercer lugar.</p> <p>Imprime regiones según lugar que le corresponde.</p>
--	--

EJERCICIO

- a) Mejore el código para encontrar la lista de la región mayor a la menor.

Ejemplo de programa que usa un arreglo

El sistema para la protección del medio ambiente, tiene registrado los promedios de emisiones contaminantes de las 10 principales empresas de la región. El programa obtiene un reporte de los indices de contaminación promedio en la región, así como cuantas empresas están por debajo del promedio.

```
public class practica1 {  
    public static void main(String[ ] args) {  
  
        float cont[ ]={.23f,.56f,.23f,.53f,.20f,.67f,.23f,.54f,.23f,.23f};  
  
        int i;  
        int c;  
        float suma=0;  
        float promedio;  
  
        for(i=0;i<cont.length;i++)  
            suma=suma+cont[i];  
  
        promedio=suma/cont.length;  
        c=0;  
        for(i=0;i<cont.length;i++)  
            if(cont[i]<promedio)  
                c++;  
  
        System.out.println("El indice de contaminación promedio="+promedio);  
        System.out.println("La cantidad de empresas por debajo del promedio="+c);  
  
    }  
}
```

TEMAS ADICIONALES

I.- Métodos de la clase Arrays.

La clase Arrays, se localiza en el paquete `java.util.*`; al igual que muchas otras clases, esta contiene un conjunto de métodos que nos permiten manipular arreglos. En particular la clase Arrays, sus métodos proveen funcionalidad sobre arreglos de tipos de datos básicos, la siguiente lista muestra los principales métodos que se podrán utilizar.

METODO	DESCRIPCION
<pre>static int binarySearch(byte[] a, byte key) static int binarySearch(char[] a, char key) static int binarySearch(double[] a, double key) static int binarySearch(float[] a, float key) static int binarySearch(int[] a, int key) static int binarySearch(long[] a, long key) static int binarySearch(short[] a, short key) static int binarySearch(Object[] a, Object key)</pre>	Busca sobre el arreglo especificado (a) un valor, retorna la posición donde se localiza, en caso contrario retornara un valor negativo. Nota este método supone que el arreglo esta ordenado.
<pre>static boolean equals(boolean[] a, boolean[] a2) static boolean equals(byte[] a, byte[] a2) static boolean equals(char[] a, char[] a2) static boolean equals(double[] a, double[] a2) static boolean equals(float[] a, float[] a2) static boolean equals(int[] a, int[] a2) static boolean equals(long[] a, long[] a2) static boolean equals(short[] a, short[] a2) static boolean equals(Object[] a, Object[] a2)</pre>	Pregunta si dos arreglos son iguales; esto quiere decir que son del mismo tamaño y sus elementos son los mismos.. Retorna un valor true para indicar que si lo es y un false para el caso contrario.
<pre>static void fill(boolean[] a, boolean val) static void fill(boolean[] a, int l, int F, boolean val) static void fill(byte[] a, byte val) static void fill(byte[] a, int l, int F, byte val) static void fill(char[] a, char val) static void fill(char[] a, int l, int F, char val) static void fill(double[] a, double val) static void fill(double[] a, int l, int F, double val) static void fill(float[] a, float val) static void fill(float[] a, int l, int F, float val) static void fill(int[] a, int val) static void fill(int[] a, int l, int F, int val) static void fill(long[] a, long val) static void fill(long[] a, int l, int F, long val) static void fill(short[] a, short val) static void fill(short[] a, int l, int F, short val) static void fill(Object[] a, Object val) static void fill(Object[] a, int l, int F, Object val)</pre>	Rellena un arreglo con un valor específico, los parámetros l y F, indican desde y hasta donde deben llenar el arreglo. val indica el valor con el que se tiene que llenar cada posición.

Fundamentos de programación orientada a objetos con Java

METODO	DESCRIPCION
<pre>static void sort(byte[] a) static void sort(byte[] a, int fromIndex, int toIndex) static void sort(char[] a) static void sort(char[] a, int fromIndex, int toIndex) static void sort(double[] a) static void sort(double[] a, int fromIndex, int toIndex) static void sort(float[] a) static void sort(float[] a, int fromIndex, int toIndex) static void sort(int[] a) static void sort(int[] a, int fromIndex, int toIndex) static void sort(long[] a) static void sort(long[] a, int fromIndex, int toIndex) static void sort(short[] a) static void sort(short[] a, int fromIndex, int toIndex) static void sort(Object[] a) static void sort(Object[] a, int fromIndex, int toIndex) static void sort(Object[] a, Comparator c) static void sort(Object[] a, int fromIndex, int toIndex, Comparator c)</pre>	Ordena el vector o parte de el indicada por fromIndex y toIndex en forma ascendente.

```
import javax.swing.*;
import java.util.*;

public class busqueda {
    public static void main(String[] args) {
        int [ ]a={1,5,6,8,10,4,3,5,18,2};
        int [ ]b={4,6,7,8};

        float [ ] y= new float [10];

Arrays.fill(y,5.5F); // Rellena el arreglo en todas sus posiciones con el valor de 5.5

        System.out.println("Arreglos Originales");
        for(int i=0;i<a.length;i++)
            System.out.println(a[i]+\t+y[i]);

Arrays.sort(a); // Clasifica el arreglo en forma ascendente

        System.out.println("Arreglo Clasificado en orden Ascendente");
        for(int i=0;i<a.length;i++)
            System.out.println(a[i]);

        int N=Integer.parseInt(JOptionPane.showInputDialog("Dar numero a buscar:"));

        int l=Arrays.binarySearch(a,N); // Busca el valor de N en el arreglo a
        if(l>=0)
            System.out.println("Lo encontro en la posición "+l);
        else
            System.out.println("No lo encontro");
```

```

if(Arrays.equals(a,b)) // Compara si a y b son dos arreglos iguales.
    System.out.println("Son iguales");
else
    System.out.println("No son iguales");

System.exit(0);

}
}

```

II.- Métodos para cadenas de caracteres

La clase String ofrece la funcionalidad para la manipulación de cadenas de caracteres, esta clase provee un variedad de métodos para manipular cadenas de caracteres, a continuación se muestran los principales servicios:

METODO	DESCRIPCION
char charAt(int index)	Retorna el carácter localizado en la posición indicada por el argumento de la cadena actual.
int compareTo(String s)	Compara la cadena actual con la cadena s. Retorna un valor = 0 si son iguales, >0 Si la cadena actual es lexicográficamente mayor que s y <0 si la cadena actual es lexicográficamente menor que s.
int compareIgnoreCase(String s)	Compara la cadena actual con la cadena s, ignorando si son Mayusculas o minusculas. Retorna un valor = 0 si son iguales, >0 Si la cadena actual es lexicográficamente mayor que s y <0 si la cadena actual es lexicográficamente menor que s.
String concat(String str)	Concatena la cadena actual con str y retorna la nueva cadena.
boolean endsWith(String suffix)	Prueba si la cadena actual termina con el sufijo especificado por suffix.
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	Copia caracteres de la cadena actual en el arreglo de caracteres especificado por dst, iniciando del carácter srcBegin y terminando en el carácter srcEnd. Los caracteres iniciarán a ser copiados en el arreglo en la posición dstBegin
int hashCode()	Obtiene el valor hash de la cadena actual, retornándolo como un valor entero, este se calcula: $s[0]*31^{n-1} + s[1]*31^{n-2} + \dots + s[n-1]$ Donde S[i] : Es el i-enesimo carácter n: Es la longitud de la cadena ^: Representa exponente

Fundamentos de programación orientada a objetos con Java

METODO	DESCRIPCION
int indexOf(int ch)	Retorna la posición en la cadena donde se encuentra la primera ocurrencia del carácter ch.
int indexOf(int ch, int fromIndex)	Idem. al anterior solo iniciando en fromIndex.
int indexOf(String str)	Retorna la posición donde str inicia en la cadena actual, Si str no se localiza retorna -1,
int indexOf(String str, int fromIndex)	Idem. al anterior solo iniciando en fromIndex
int lastIndexOf(int ch) int lastIndexOf(int ch, int fromIndex) int lastIndexOf(String str) int lastIndexOf(String str, int fromIndex)	Retorna la posición de la última ocurrencia del carácter ch o la cadena str.
int length()	Retorna la longitud de la cadena
boolean matches(String regex)	Retorna si la cadena actual es válida o no, especificada por la expresión regular pasada como parámetro.
String replace(char oldChar, char newChar)	Retorna una cadena, con las ocurrencias del carácter oldChar, en la cadena actual sustituidos por newChar.
boolean startsWith(String prefix)	Prueba si una cadena inicia con una cadena como prefijo especificada por prefix.
String substring(int beginIndex) String substring(int beginIndex,int endIndex)	Retorna una subcadena de la cadena actual iniciando en beginIndex y terminando en endIndex.
char[] toCharArray()	Retorna un arreglo de caracteres conteniendo los caracteres de la cadena actual.
String toLowerCase()	Retorna una cadena con los caracteres todos en minúsculas de la cadena actual.
String toUpperCase()	Retorna una cadena con los caracteres todos en mayúsculas de la cadena actual.
String trim()	Retorna una cadena sin espacios en blanco al inicio y al final de la cadena, si es que los tuviera.

```

import java.lang.*;
public class Cadenas {
    public static void main(String[] args) {
        String cadena;
        char [ ]a;
        cadena= new String(" Instituto Tecnológico de Orizaba ");
        System.out.println(cadena.length());
    }
}
  
```

Fundamentos de programación orientada a objetos con Java

```
System.out.println(cadena.charAt(10));
System.out.println(cadena.compareTo("ITO"));
System.out.println(cadena.compareToIgnoreCase("INSTITUTOTECNOLÓGICODEORIZABA"));
System.out.println(cadena.concat("\tSistemas y computación"));
System.out.println(cadena.endsWith("ción"));
System.out.println(cadena.hashCode());
System.out.println(cadena.indexOf("Orizaba"));
System.out.println(cadena.replace('o','#'));
System.out.println(cadena.startsWith("ITO"));
System.out.println(cadena.lastIndexOf("gico"));
System.out.println(cadena.substring(10));
System.out.println(cadena.toLowerCase());
System.out.println(cadena.toUpperCase());
System.out.println(cadena.trim());

a=cadena.toCharArray();

for(int i=0;i<a.length;i++)
    System.out.println(a[i]);

}
```

}

III.- Métodos de la clase Random

La clase Random, se localiza en el paquete java.util, la generación de numeros aleatorios es muy importante para muchas aplicaciones, principalmente con las que tienen que ver con cálculo estadístico, probabilístico y la simulación. Esta clase es uno de los métodos mas comodos para generar numeros aleatorios. Se debe considerar que todos sus métodos deberan ser invocados por un objeto creado previamente.

METODO	DESCRIPCION
boolean nextBoolean()	Retorna un valor booleano generado aleatoriamente.
double nextDouble()	Retorna un valor double generado aleatoriamente.
float nextFloat()	Retorna un valor float generado aleatoriamente.
int nextInt()	Retorna un valor int generado aleatoriamente.
int nextInt(int n)	Retorna un valor int generado aleatoriamente, entre 0 y n-1.
long nextLong()	Retorna un valor long generado aleatoriamente.

Fundamentos de programación orientada a objetos con Java

METODO	DESCRIPCION
void setSeed(long seed)	Modifica la semilla para generar números aleatorios.

```

import javax.swing.*;
import java.util.*;

public class encuesta2 {

    public static void main(String[ ] args) {

        int[ ][ ] resultado;
        Random r= new Random();

        int edad = 0;
        String resp;
        int i, j, N;

        resultado = new int[26][2];
        resp = JOptionPane.showInputDialog("Dar cuantas encuestas se generan");
        N = Integer.parseInt(resp);

        for(i=1;i<=N;i++) {
            edad=r.nextInt(46);
            if (edad >= 20 && edad <= 45) {
                if (r.nextBoolean())
                    resultado[edad - 20][0]++;
                else
                    resultado[edad - 20][1]++;
            }
        }
        System.out.println("Edad : SI NO");
        System.out.println("=====");
        for(i=0;i<resultado.length;i++){
            System.out.print((i + 20) + " : ");
            for (j = 0; j < resultado[i].length; j++)
                System.out.print(resultado[i][j]+ " ");
            System.out.println();
        }
        System.exit(0);
    }
}

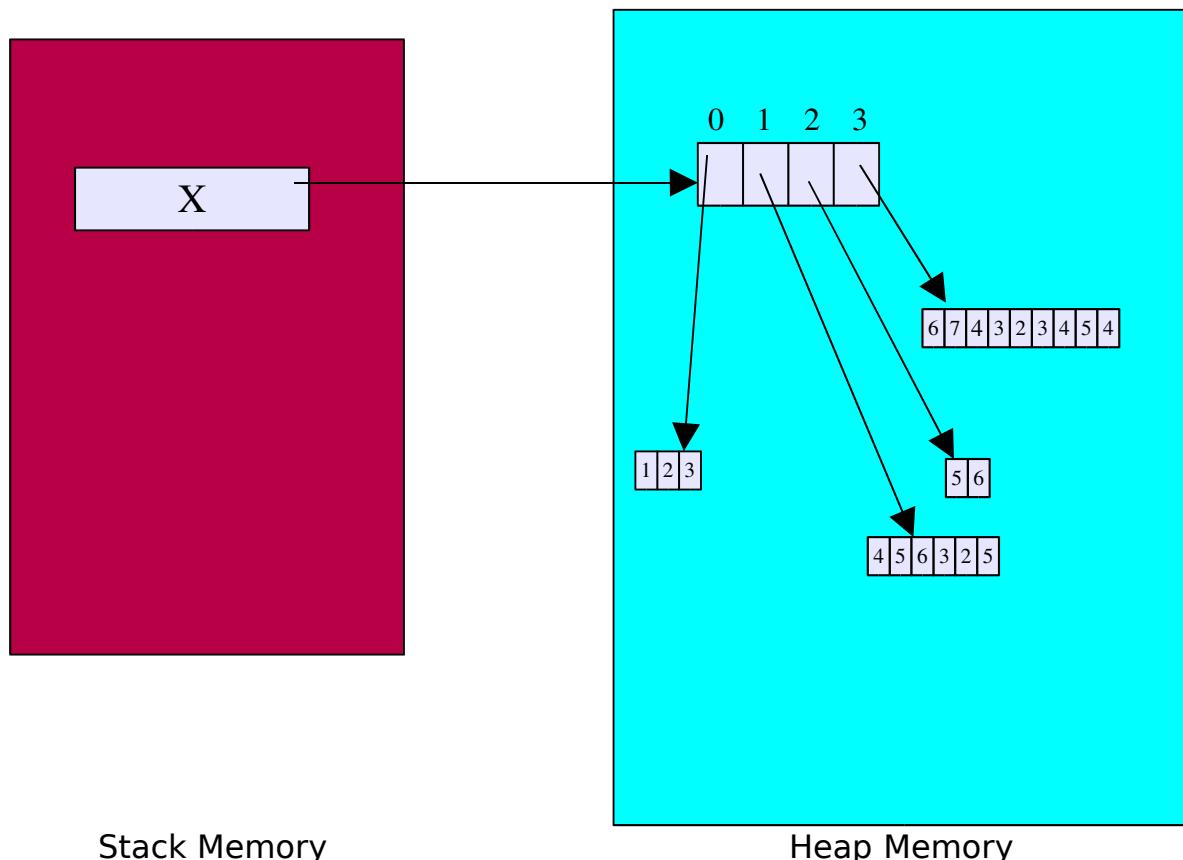
```

IV.- Arreglos bidimensionales con diversos tamaños de columnas por cada fila

Como se vio en el tema 2.2, los arreglos bidimensionales se les llama matrices también, ya que tienen un número específico de renglones y columnas; ningún renglón y ninguna columna queda sin elemento; pero Java permite que los arreglos bidimensionales puedan tener diferentes tamaños, cada renglón.

```
int [ ][ ] x = { {1,2,3},  
                 {4,5,6,3,2,5},  
                 {5,6},  
                 {6,7,4,3,2,3,4,5,4}  
};
```

Se puede decir que el arreglo *x* tiene 4 renglones, pero no es fácil identificar cuantas columnas tiene, ya que cada renglón tiene número diferente de elementos, un arreglo así se describe en la memoria de la siguiente forma:



El siguiente programa muestra como se puede recorrer el arreglo para

Fundamentos de programación orientada a objetos con Java

imprimirlo en forma matricial.

```
public class arreglo4 {  
    public static void main(String[] args) {  
        int[ ][ ] x= {{1,2,3},  
                      {4,5,6,3,2,5},  
                      {5,6},  
                      {6,7,4,3,2,3,4,5,4}};  
  
        int i,j;  
        for(i=0;i<x.length;i++){  
            for (j = 0; j < x[i].length; j++)  
                System.out.print(x[i][j] + "\t");  
            System.out.println();  
        }  
    }  
}
```

Capítulo III Métodos y mensajes

3.1 Atributos const y static.

Estos atributos (const y static) tienen diferentes concepciones en C++ y Java, incluso en Java const no es utilizada, en su lugar se utiliza la palabra final y aun así los significados no podemos considerarlos sinónimos.

La palabra **static** en Java es un atributo que se utiliza en la declaración de datos y métodos miembros de una clase. Si se trata de un dato este puede ser referenciado según el ámbito que tenga (public, private o protected) de forma indistinta con la restricción que los objetos comparten dicha variable como se vera más adelante, un dato static no requiere que un objeto se cree para poder referenciarlo, por otro lado los métodos static se dice que son métodos de clase y no de instancias u objetos, por lo que no podran utilizar datos miembros que no sean static de una clase. Para declarar elementos static dentro de un programa se sigue la siguiente sintaxis:

```
[ambito] static tipoDeDatos variable;
```

```
[ambito] static tipoDeDatos nombre_método ([parámetros]) {  
}
```

Por lo que hace a la palabra final, esta se utiliza en el caso de los datos para indicar que se trata de una constante.

```
final tipo variable[= expresión];
```

Ejemplos:

```
public class ejemplo{  
    static int numero;  
    final float x=3.45F;  
    long numero2;  
    static void cambia( ){  
        numero=56;  
        numero2=564523; // Error no es posible por regla de alcance  
    }  
}
```

3.2 Concepto de método.

Desde sus inicios la programación ha enfrentado los problemas de la complejidad y tamaño de los problemas que se requieren resolver por

computadora. La programación ha pasado por muchos estadios, siempre aprovechando los mecanismos desarrollados con anterioridad. La programación orientada a objetos no es la excepción.

En un principio la complejidad del problema, así como el tamaño de los programas, no requerían de grandes esfuerzos de abstracción por parte del programador y estos resolvían los problemas con unas cuantas líneas de código.

A medida que la solución crecía la complejidad aumentaba lo cual hacia más complicado tener la solución mejor en el menor tiempo posible. Para esto se echo mano de la abstracción a nivel de operación, entre mas complejo era un problema más abstracción se requería para resolverlo; fue así como el concepto de divide y vencerás se hizo presente. En un principio el diseño top-down que implica dividir un problema complejo en módulos funcionales más pequeños, manejables, entendibles y fáciles de implementar, fueron la solución.

Esto dio pie al subprograma, que ha recibido distintos nombres según el lenguaje que lo implemente (rutina, subrutina, procedimiento, función, etc.). Estos subprogramas ayudaban a dividir el problema en términos funcionales sin importar los datos que fueran operados por ellos.

En el caso de la programación orientada a objetos, el método hereda las características del subprograma pero hace hincapié que actúa sobre los datos de un objeto. Haciéndose énfasis que un método esta íntimamente ligado a datos sobre los que actúa, esto quiere decir que los datos y los métodos están encapsulados. En resumen un método es un subprograma con la intención de actuar para y por los datos a los cuales esta asociado.

Así podemos tener objetos específicos con métodos que proporcionan los servicios a la aplicación para que dichos objetos sean utilizados, ya que por los datos solamente no tienen razón de ser.

Como un paso intermedio de la programación estructurada a la programación orientada a objetos, se encuentra el concepto de tipo de dato abstracto (TDA). Un TDA se define como datos y operaciones (métodos, subprogramas) que determinan como se pueden manipular los datos, estos dos elementos definidos por el programador para resolver un problema específico.

Los métodos deben verse como fragmentos de código que tienen una función específica dentro de un programa y que pueden ser reutilizados.

Por ejemplo podemos definir un objeto llamado cuadrado, del cual la única información que se requiere representar es el tamaño del lado; las operaciones que pudiéramos implementar para manipular este dato serían: Inicializar el

tamaño del lado, modificar el tamaño del lado, obtener que valor tiene el lado, obtener el área del cuadrado y obtener el perímetro del cuadrado.

3.3 Declaración de métodos.

En java los métodos deben estar siempre declarados dentro de una clase y deben seguir las siguientes reglas al momento de declararlos:

```
declaraciónDeMétodo {  
    CuerpoDeMétodo  
}
```

Donde:

declaraciónDeMétodo debe estar especificado al menor por:

```
[ambito] [static] tipoDeDato nombreDeMétodo ( [listaDeArgumentos] )
```

Los elementos encerrados entre corchetes ([]) son opcionales y dependen de las necesidades del programador.

[ambito]: Este elemento es opcional, por defecto todos los métodos tienen alcance a nivel de paquete, quiere decir que son visibles desde cualquier clase que este dentro del mismo paquete. Otros ámbitos permitidos son public, private o protected.

[static] : Es un elemento opcional y será tratado en el tema 3.5.1.

tipoDeDato: Es cualquier tipo de dato básico, clase, arreglo o la palabra void, representa el tipo de valor que retornara el método. En la programación estructurada el concepto de procedimiento y función se utilizaba para diferenciar subprogramas que retornan o no retornan valor. En Java los métodos deben retornar valor cuando se especifica el tipo de valor que retornaran, los tipos incluyendo arreglos; un método que no retorna valor debe utilizarce la palabra void como tipo de dato. Cuando se utiliza el nombre de una clase indica que el valor de retorno es un objeto de la clase especificada.

[listaDeArgumentos]: Es una lista que especifica los argumentos o parámetros que el método requiere al momento de ser invocado (call). Un argumento es un valor, que es introducido al método para ser utilizado por el cuerpoDeMétodo, en alguna de sus sentencias. Los argumentos pueden ser valores de tipo básico u objetos de alguna clase específica incluyendo a los arreglos. Cada elemento de la lista va separado por una coma.

tipodeDato1 argumento1, tipodeDato2 argumento2,...., tipodeDatoN argumentoN

CuerpoDeMétodo: es un conjunto de declaraciones internas y sentencias que especifican la tarea que realiza el método. Las declaraciones son principalmente variables internas del método, estas solo son visibles cuando el control del programa es transferido al método, una vez que el método termina de ejecutarse las variables se destruyen. En cada invocación al método, este tiene la obligación de crear las variables nuevamente. Las sentencias que se pueden utilizar dentro del cuerpo del método son la asignación, la selección, la iteración, la invocación a métodos y el retorno de método, si el método no fue declarado con el tipo void. Para retornar el valor de un método se utiliza la palabra reservada return como lo especifica la sintaxis siguiente:

return expresión;

expresión: Puede ser una literal o constante, una variable o una expresión del tipo que
esta especificado el método.

Ejemplos:

```
// Método que suma dos valores enteros que son pasados como parámetros
int suma( int a, int b){
    return a+b;
}
```

```
//Método que obtiene la diferencia absoluta entre dos números enteros
int diferencia(int a, int b){
    int dif=0;
    if(a>b)
        dif=a-b;
    else
        dif=b-a;
    return dif;
}
```

```
//Método que obtiene el valor máximo de un arreglo de flotantes  
float maximo(float [ ] x){
```

```
    int i;  
    float max=x[0];  
  
    for(i=1;i<x.length;i++)  
        if(x[ i ] > max)  
            max=x[ i ];  
  
    return max;  
}
```

```
//Método que imprime los elementos de un arreglo de char en forma inversa
```

```
void imprime_reverse(char [ ] letras){  
int i;  
for(i=letras.length-1; i>=0; i --)  
    System.out.print(letras[ i ]);  
}
```

3.4 Llamadas a métodos (mensajes).

Un método por si solo no hace nada si no es invocado o llamado (call), como los métodos son miembros de una clase, para poder invocarlos primero se tiene que construir una instancia de la clase, las instancias de clases se les llama objetos. Cuando de una instancia se invoca a uno de sus métodos se dice que se envía un mensaje al objeto.

Los métodos static se invocan de otra forma, este tipo de métodos al ser de clase, no requiere que se creen instancias para ser invocados. Si estan dentro de la misma clase se invocan utilizando únicamente el nombre del método con los parámetros requeridos, si el método esta en otra clase se recomienda utilizar el nombre de la clase seguido de un punto y el nombre del método con los parámetros requeridos.

Por ejemplo si tengo la siguiente clase:

Fundamentos de programación orientada a objetos con Java

```
public class triangulo{  
    float getArea(float base, float altura){  
        float area;  
        area= base * altura / 2;  
        return area;  
    }  
  
    float getPerimetro(float l1, float l2, float l3){  
        float perimetro;  
        perimetro=l1+l2+l3;  
        return perimetro;  
    }  
}
```

Declaración del método `getArea`, tiene 2 argumentos flotantes, uno representa la base y otro la altura de un triángulo.

Se calcula el área del triángulo con los datos que fueron pasados como parámetros.

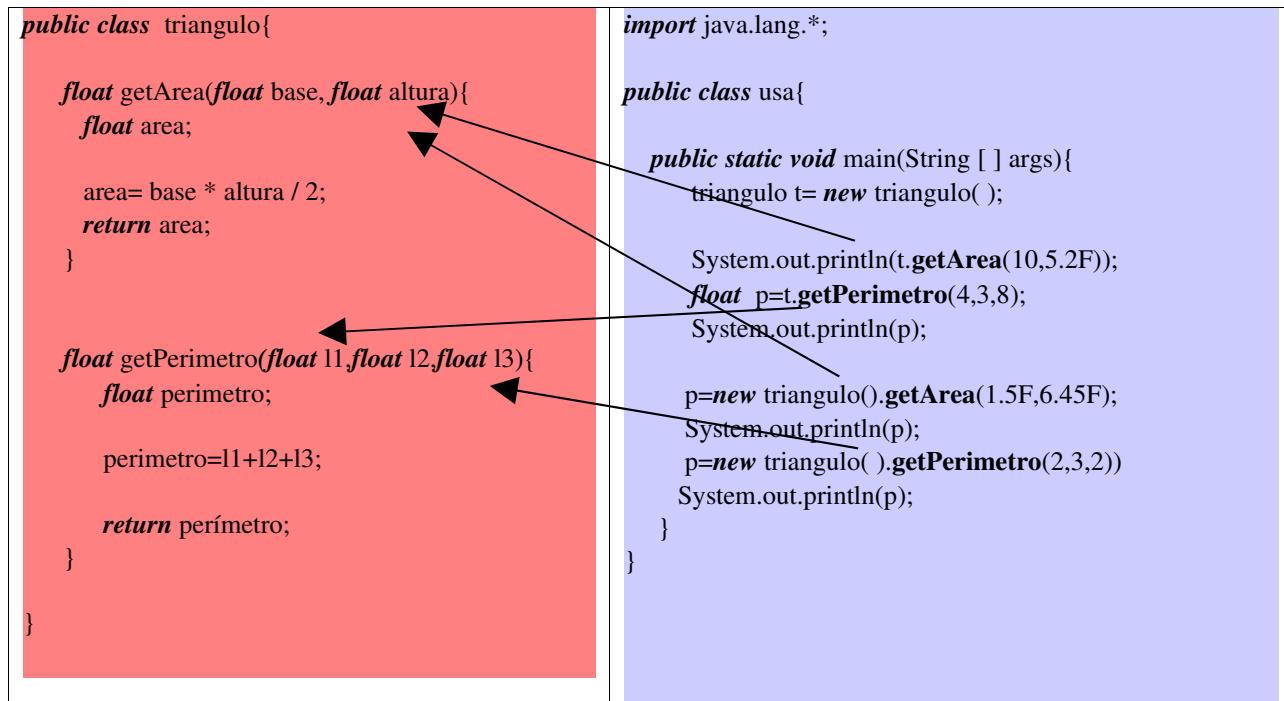
Retorna el valor calculado.

Declaración del método `getPerimetro`, Tiene 3 argumentos flotantes, cada uno representa la medida de cada lado de un triángulo.

Se calcula el perímetro del triángulo.

Retorna el valor calculado

Los dos métodos de la clase triángulo tienen un ámbito por default, alcance de paquete, para poder hacer referencia a los métodos, la aplicación debe de estar en el mismo paquete o directorio. El siguiente código explica como se puede invocar a esta clase de métodos.



3.5 Tipos de métodos.

La filosofía de la programación orientada a objetos, obliga a que los datos preferentemente estén protegidos contra modificaciones arbitrarias, para esto todo objeto debe proveer mecanismos de protección y una interfaz que le permita interactuar con las aplicaciones que lo utilizarán.

Una clase es la declaración genérica que los objetos necesitan al momento que se crean, cada objeto define sus propios datos y comparten los métodos. Los métodos al momento de ser invocados por un objeto estos definen sobre qué datos actúan. Una clase puede tener al menos 6 tipos de métodos que también son llamados servicios, que los objetos utilizarán en una aplicación:

- Métodos de crear e inicializar un objeto (Constructores, inicializadores).
- Métodos para modificar el estado de un objeto (Modificadores).
- Métodos para recuperar el estado de un objeto (Recuperadores).
- Métodos para cuestionar el estado de un objeto (Cuestionadores).
- Métodos manejadores de arreglos miembros de un objeto (Iteradores)
- Métodos liberadores de recuersos de objetos (Destructores)

Las clases deben ofrecer una serie de métodos genéricos para protección de los datos, a los métodos que afectan el estado de un objeto (modifica los valores de los datos del objeto) se les llama modificadores, a esta clase de métodos se le recomienda utilizar al definir su nombre el prefijo set seguido por el

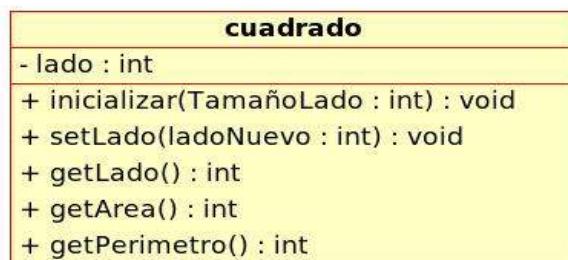
Fundamentos de programación orientada a objetos con Java

nombre del dato a ser alterado. Los métodos que proveen información acerca del estado de un objeto se les llama seleccionadores o recuperadores, a esta clase de métodos se le recomienda utilizar al definir su nombre el prefijo get seguido por el nombre del dato a ser recuperado.

Los métodos que pregunta sobre el estado de un objeto (Cuestionadores) se les recomienda utilizar el prefijo is.

Los métodos constructores, destructores e iteradores más adelante serán tratados.

Suponer la clase llamada cuadrado descrito en la sección 3.2, en un diagrama de clases podemos describirlo de la siguiente forma:



```
public class cuadrado{
    private int lado;
    public void inicializar(int TamañoLado){
        lado= TamañoLado;
    }
    public void setLado(int ladoNuevo){
        lado=ladoNuevo;
    }
    public int getLado( ){
        return lado;
    }
    public int getArea( ){
        return lado*lado;
    }
    public int getPerimetro( ) {
        return lado*4;
    }
}
```

La utilidad de una clase queda a cargo de la aplicación. Por ejemplo: Se necesita una aplicación que permite a un niño probar sus conocimientos acerca del perímetro y área de cuadrados. El programa generará lados de cuadrados en forma aleatoria y preguntará al niño acerca del área y el perímetro, cuando el

niño ya no deseé seguir probando sus conocimientos, la aplicación terminará imprimiendo la calificación otorgada al niño.

```
import java.util.*;
import javax.swing.*;

public class aplicaCuadrado {

    public static void main(String[] args) {
        cuadrado c= new cuadrado();
        int resp;
        Random r= new Random();
        int intentos=0;
        int exitos=0;
        c.inicializar(r.nextInt(20)+1);
        do{

            int a,p;
            a=Integer.parseInt(JOptionPane.showInputDialog("Dar área con lado ="+c.getLado()));
            p=Integer.parseInt(JOptionPane.showInputDialog("Dar perímetro con lado =" +c.getLado()));
            intentos++;
            if(a== c.getArea() && p== c.getPerimetro())
                exitos++;
            c.setLado(r.nextInt(20)+1);
            resp=Integer.parseInt(JOptionPane.showInputDialog("Deseas intentar?: 1: SI : 2:NO"));
        }while(resp!=2);

        System.out.println("Tu calificación="+(exitos/intentos*10));
        System.exit(0);
    }
}
```

3.5.1 Métodos const, static.

Como se comentó al principio, la palabra const no tiene una funcionalidad en Java; por otro lado la palabra static, se utiliza para declarar métodos de clase. Esto quiere decir que no pertenecen a instancias de objetos, sino a una clase, lo cual hace que no se necesiten crear objetos para referenciarlos. Un método static funciona como un método amigo, estos son llamados funciones amigas en C++.

Un método static, permite definir una funcionalidad que no actúa sobre un objeto específico, clases de la API de Java contienen una gran cantidad de métodos de este tipo. Por ejemplo la clase Math.

METODO	DESCRIPCION
static double abs(double a) static float abs(float a) static int abs(int a) static long abs(long a)	Obtiene el valor absoluto del valor a, pasado como parámetro.
static double acos (double num) static double asin (double num) static double atan (double num)	arco coseno, arco seno o el arco tangente de num
static double cos (double angulo) static double sin (double angulo) static double tan (double angulo)	coseno, seno o tangente de angulo
static double ceil (double num)	Techo de num, i.e., el entero más pequeño mayor o igual a num
static double exp (double pot)	Valor e a la pot
static double floor (double num)	Piso de num, i.e., el entero más grande menor o igual a num
static double pow (double num, double power)	num elevado a la power
static double random ()	Número aleatorio entre 0 (inclusive) y 1 (inclusive)
static double sqrt (double num)	La raíz de num, que debe ser positivo

Ejemplo de uso de métodos de la clase Math.

```
import java.lang.*;  
  
public class Matematicas {  
    public static void main(String[] args) {  
        int numeroI;  
        float numeroF;  
  
        numeroI=-400;  
        numeroF=1.5F;  
  
        System.out.println(Math.abs(numeroI));  
        System.out.println(Math.max(numeroI,numeroF));  
        System.out.println(Math.min(numeroI,numeroF));  
        System.out.println(Math.pow(numeroF,3));  
        System.out.println(Math.round(numeroF));  
        System.out.println(Math.sin(numeroF));  
        System.out.println(Math.sqrt(numeroF));  
        System.out.println(Math.PI);  
    }  
}
```

```
        System.out.println(Math.ceil(numeroF));
        System.out.println(Math.floor(numeroF));
    }
}
```

En un momento determinado se puede querer crear una clase en la que el valor de una variable de instancia sea el mismo (y de hecho sea la misma variable) para todos los objetos instanciados a partir de esa clase. Es decir, que exista una única copia de la variable de instancia. Se usará para ello la palabra clave **static**.

```
class Documento {
    static int version = 10;
}
```

El valor de la variable `version` será el mismo para cualquier objeto instanciado de la clase **Documento**. Siempre que un objeto instanciado de `Documento` cambie la variable `version`, ésta cambiará para todos los objetos.

De la misma forma se puede declarar un método como estático, lo que evita que el método pueda acceder a las variables de instancia no estáticas:

```
class Documento {
    static int version = 10;
    int numero_de_capitulos;

    static void añade_un_capitulo() {
        numero_de_capitulos++; // esto no funciona al ser un dato no static
    }

    static void modifica_version( int i ) {
        version++; // esto si funciona
    }
}
```

La modificación de la variable `numero_de_capitulos` no funciona porque se está violando una de las reglas de acceso al intentar acceder desde un método estático a una variable no estática.

3.5.2 Métodos normales y volátiles.

Los métodos de una clase preferentemente deben indicar el ámbito al cual se hace referencia, Java no ofrece mecanismos de volatilidad como sería el caso de C++, en este apartado se comentará sobre el uso de métodos públicos y privados, así como las referencias a datos miembros de las mismas clases.

Se dice que todo dato miembro de una clase, preferentemente deberá ser declarado private por seguridad de la información que los objetos almacenarán.

Los métodos podrán ser public, private o protected, un método public es visible desde cualquier otra clase no importando si está o no dentro de un paquete; un método private solo es accedido y visible por métodos miembros de la clase, a esta clase de métodos se le llama de uso doméstico y su funcionalidad es interna al objeto; un método protected es visible fuera de la clase siempre y cuando sea desde una clase derivada, los cuales serán tratados en el capítulo 6.

Cualquier método dentro de una clase debe ser reconocido como: constructor, destructor, modificador, seleccionador, cuestionador, iterador o bien como un método de uso doméstico.

Un método inicializador es aquel que inicializa el estado de un objeto, en el capítulo 5 se tratarán esta clase de métodos. Los métodos modificadores y seleccionadores son los casos más típicos de una clase. Cuando una clase contienen colecciones o conjuntos de objetos o datos es necesario implementar mecanismos que permitan recuperar elemento por elemento, a esta clase de métodos se les llama iteradores, estos serán tratados en el capítulo 6.

Todo método que se construya, siempre deberá pensarse en términos de la utilidad que este tendrá dentro de los objetos que lo utilizarán; ya sea que afecte el estado del objeto, que recupere el estado de este, que haga así mismo una operación de mantenimiento interno o que realice algún cálculo para recuperar algún estado que no tiene que ver directamente con los datos que describen su estado.

```
public tipoDeDatos nombreMétodo ([listadeArgumentos]){\n}
```

```
private tipoDeDatos nombreMétodo ([listadeArgumentos]){\n}
```

Como regla todos los métodos modificadores y seleccionadores deberán ser public. Un método modificador deberá tener al menos un argumento y preferentemente no retornará valor. Un método seleccionador deberá siempre tener tipo de retorno y preferentemente no tener argumentos.

Un método cuestionador también deberá ser public y deberá retornar un valor booleano, respondiendo a la pregunta que se le hace.

Fundamentos de programación orientada a objetos con Java

Ejemplo: Si tenemos en una aplicación de juego por computadora, cada vez que el usuario derriba un enemigo debe contabilizar los puntos que lleva, la siguiente clase representa al usuario.

```
public class usuario{  
    private String nombre;  
    private int puntos;  
    private int vidas;  
  
    public void inicializa(String n){ // Método inicializador  
        nombre=n;  
        puntos=0;  
        vidas=5;  
    }  
  
    public void eliminaVida ( ){ // Método modificador  
        if(vidas>=1)  
            vidas --;  
    }  
  
    private void aumentaVida( ){ // Método modificador de uso interno por el objeto  
        vidas ++;  
    }  
  
    public void aumentaPuntos(int cantidad){ // Método modificador  
        if (cantidad >500)  
            aumentaVida( ); // Solo aumenta vida cuando al aumentar puntos sobrepasa los 500.  
        puntos=puntos+cantidad;  
    }  
  
    public boolean tieneVidas( ){ // Método cuestionador  
        boolean band=false;  
        if(vidas>0)  
            band=true;  
        return band;  
    }  
  
    public String getNombre( ){ // Método recuperador  
        return nombre;  
    }
```

```
public int getPuntos( ){ // Método recuperador  
    return puntos;  
}  
}
```

3.6 Referencia this.

Esta palabra reservada es una referencia, this referencia el objeto actual sobre el que se está ejecutando un mensaje o método. Es útil para resolver ambigüedades que se pudieran dar entre argumentos y datos miembros de una clase y entre métodos de la clase actual y su clase base.

Ejemplo de ambigüedad entre un argumento y un dato miembro.

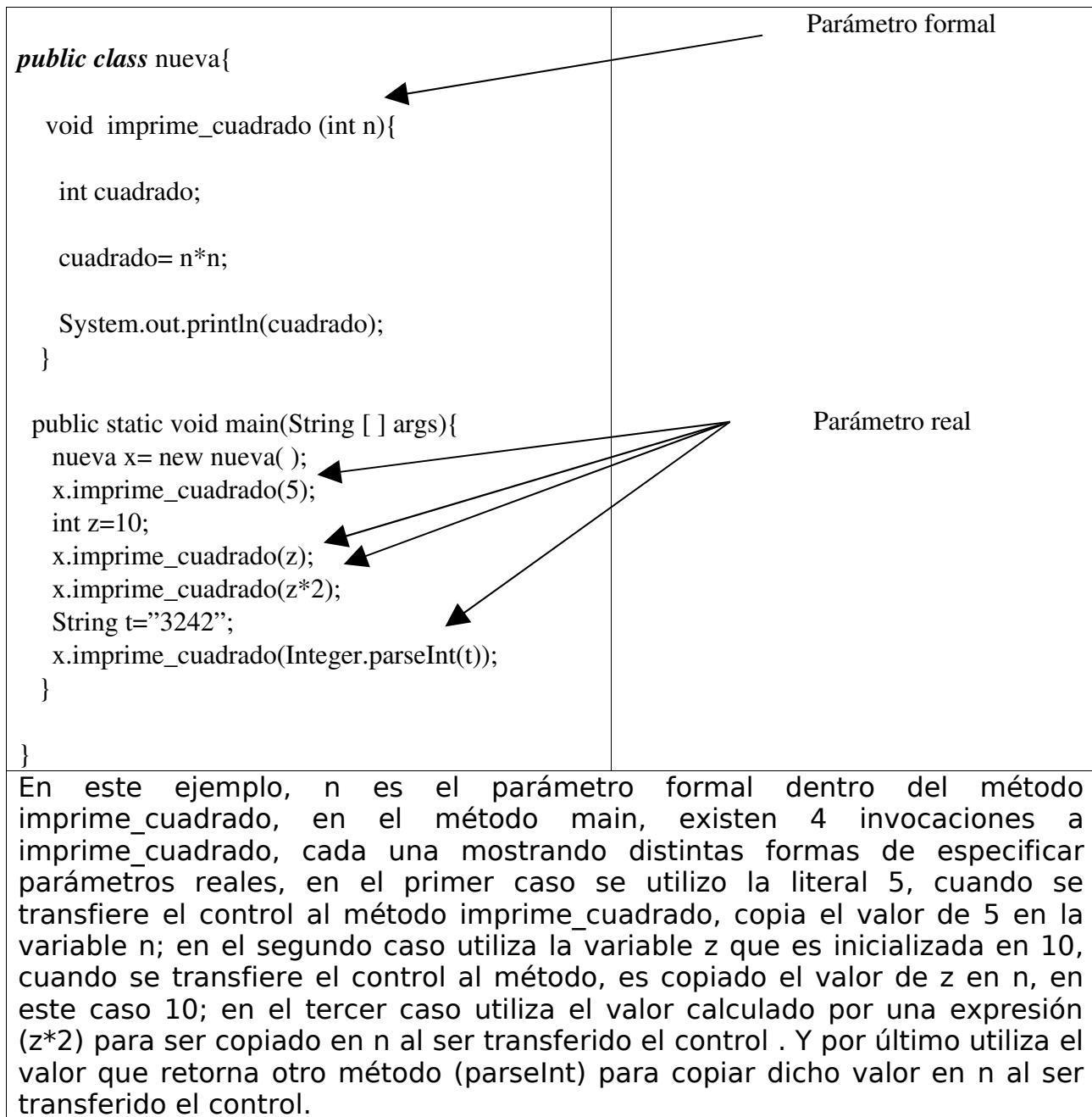
```
public class ejemplo {  
  
    private int d;  
  
    // El compilador no dará error, pero al dato miembro de la clase no se le asignara el dato.  
    public void setD(int d){  
        d=d ; // ambigüedad el argumento y el dato miembro se llaman igual.  
    }  
  
}
```

```
public class ejemplo {  
  
    private int d;  
  
    public void setD(int d){  
        this.d=d ; // ambigüedad resuelta con this. this.d hace referencia al dato de la clase  
    }  
  
}
```

3.7 Forma de pasar argumentos.

Los parámetros en un método se comportan como variables internas, solo se hacen visibles cuando es invocado el método. Su función es recibir los valores que son pasados en la invocación. Todos los parámetros definidos por tipos de datos básicos se dice que copian su valor del parámetro real (expresado en la invocación) al parámetro formal (expresado en la cabecera del método). Los

parámetros formales son los descritos en el encabezado del método y los parámetros reales son los que son utilizados en una invocación. A este tipo de paso de parámetros se le llega a llamar paso de parámetros por valor o por copia.



Por otro lado, en Java, a todas las variables definidas por tipos no básicos (clases o arreglos) se les llama referencias, así que los parámetros definidos por

Fundamentos de programación orientada a objetos con Java

estos tipos de datos se dice que son parámetros referencia y al mecanismo de copiar las referencias de los parámetros reales a los parámetros formales se les llama, paso de parámetro por referencia.

```
public class circunferencia{  
    float radio;  
    float area;  
    float perimetro;  
}
```

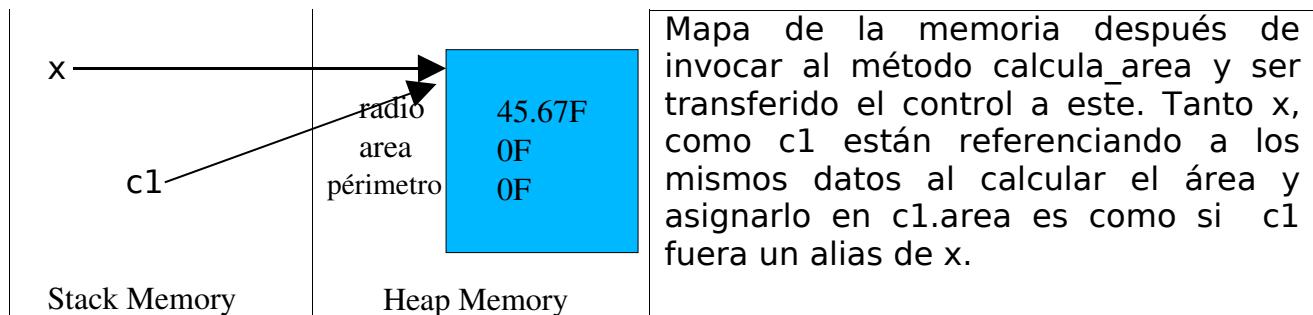
```
public class aplicación{  
  
    void calcula_area(circunferencia c1){  
        c1.area=3.1416 F*c1.radio*c1.radio;  
    }  
  
    void calcula_perimetro(circunferencia c){  
        c.perimetro=2*3.1416F*c.radio;  
    }  
  
}
```

```
public static void main(String[ ] args){  
    circunferencia x;  
    aplicación ap= new aplicación();  
    x= new circunferencia( );  
    x.radio=45.67F;  
  
    ap.calcula_area(x);  
    ap.calcula_perimetro(x);  
    System.out.println(x.area);  
    System.out.println(x.perimetro);  
}  
}
```

Parámetro formal

Parámetro real

En este ejemplo 2 métodos (`calcula_area` y `calcula_perimetro`) definen parámetro formal `c1` y `c` respectivamente, los cuales son declarados como referencias al no utilizar un tipo de dato básico, en el método `main` al invocar a los métodos indicados el parámetro real es un objeto (`x`) que previamente fue creado. Al ser `x` un objeto, lo que contiene es la dirección de la memoria donde el objeto se localiza cuando el método es invocado lo que copia es la dirección de memoria a la variable referencia indicada sea `c1` o `c` según el método utilizado.



3.8 Devolver un valor desde un método.

Todo método declarado de cualquier tipo de dato, obligatoriamente debe devolver un valor; dicho valor debe ser del tipo de dato declarado por el método. Para esto Java utiliza la sentencia `return`.

`return expresión;`

Un método puede tener más de una sentencia `return`, pero solo una será ejecutada. Una vez que se ejecuta la sentencia `return`, el control del programa retorna donde ocurrió la invocación, no importando que exista más de una sentencia `return`.

El caso de los métodos recursivos, se tiene que considerar, que el retorno algunas veces será a la misma sentencia. Un método recursivo es aquel que dentro del cuerpo del método tiene al menos una invocación así mismo. Muchos de los problemas que se resuelven recursivamente se pueden resolver con sentencias de control iterativas. Se debe tener mucho cuidado para saber cuando utilizar métodos recursivos y cuando no. La forma de expresar la solución de un problema en algunos casos, la recursividad lo expresa mejor que la forma iterativa, con la desventaja del uso de los recursos, principalmente la memoria.

El ejemplo típico de un método recursivo, es el método que calcula el factorial de un número ($n!$).

Por definición se dice que $n! = n * (n-1)!$
y $0! = 1$

Por lo tanto:

$$\begin{aligned}
 3! &= 3 * 2! = 3 * (2 * 1!) = 3 * (2 * (1 * 0!)) \\
 &= 3 * (2 * 1) = 3 * 2 \\
 &= 6
 \end{aligned}$$

En Java este método se puede expresar de la siguiente forma:

```
static long factorial (long n){  
    if(n==0)  
        return 1;  
    else  
        return n* (factorial (n-1)); // Invocación recursiva (A sí mismo)  
}
```

3.9 Estructura del código.

Todo método, no importando de que tipo sea, el código que le compone debe ser legible, claro y entendible. Todo algoritmo utilizado para representar la solución debe ser lo más general posible, esto implica el uso adecuado de parámetros y valor de retorno. El caso de los arreglos debe ser tomado en cuenta.

Una arreglo cuando es pasado como parámetro, el parámetro formal, recibe la referencia del parámetro real, lo cual indica que los dos referencian al mismo arreglo, el siguiente ejemplo muestra el método para obtener las veces que se repite un valor dentro de un arreglo:

```
static int getVecesRepite(int a[ ] , int valor){ // Tanto el arreglo como el valor a buscar son pasados  
                                              // Como parámetros  
    int c=0;  
    int i;  
    for(i=0;i<a.length;i++)  
        if(valor==a[i])  
            c++;  
    return c;  
}
```

El siguiente ejemplo muestra el método que obtiene de un arreglo, otro arreglo con los elementos del primero, pero sin valores repetidos.

```
static int [ ] getArregloSinRepetidos(int a[ ]){ // Arreglo pasado como parametro  
    int y[ ];           // Arreglo donde se guardan los elementos de a, pero sin valores repetidos  
    int x= new [a.length]; // Arreglo temporal del mismo tamaño de a.  
    int i,j=0;  
    for(i=0;i<a.length;i++)  
        if (getVecesRepite(x,a[i])==0) // Utiliza el método anterior  
            x[j++]=a[i];  
    y= new int[j]; // Crea el arreglo que retornara  
    for(i=0;i<j;i++) // Copia los elementos del temporal (x) a y  
        y[i]=x[i];  
    return y;         // Retorna el arreglo resultante.  
}
```

EJERCICIOS:

- a) Elaborar un método que se le pase un arreglo de flotantes y retorne el valor mayor.
- b) Elaborar un método que obtenga cuantas veces se repite el valor mayor en un arreglo de flotantes, utilizar una invocación al método anterior para resolver este problema.
- c) Elaborar un método que se le pase un arreglo de enteros largos (long) y obtenga un arreglo, con las posiciones donde se localiza un valor que también es pasado como parámetro.
- d) Elaborar un método que dado un arreglo de flotantes, retorne un arreglo con los valores sin la parte entera.
- e) Elaborar un método que dada una cadena de caracteres, retorne cuantas vocales tiene.
- f) Elaborar una método que dado un arreglo de enteros (int) y un valor, obtenga el número de veces que se localiza en una posición par dicho valor en el arreglo.
- g) En cierta aplicación se necesita utilizar objetos cuenta de ahorro, cada cuenta tiene numero de cuenta, a quien pertenece (nombre del cliente) y saldo. Se necesita construir una clase para describir una cuenta de ahorro, . Los objetos requieren proporcionar los servicios de modificar el nombre del cliente, asignar saldo, asignar el número de cuenta, hacer deposito, hacer retiro y consulta de saldo. Un retiro debe considerar que tiene los fondos suficientes para realizar el retiro. Indica que clase de método se utiliza para resolver cada caso.

Capítulo IV Constructor y destructor

Todo programa al ser cargado en memoria por el sistema operativo y serle transferido el control para su ejecución, tiene que crear el ambiente de ejecución, este consiste básicamente en inicializar áreas de memoria y ejecutar tareas para poder iniciar la ejecución del programa.

En la programación estructurada las rutinas (funciones y/o procedimientos) tienen que crear el ambiente para recibir los parámetros y regresar los valores, la tarea que se encarga de crear el ambiente para poder ejecutar la rutina se le llama prólogo. Cuando la rutina se termina de ejecutar, tiene que hacer una tarea para liberar la memoria y retornar la ejecución del programa donde fue invocada así como retornar el valor si es que fue especificado, a esta tarea se le llama epílogo. Tanto el prólogo como el epílogo son transparentes, el programador no trata con ellos, solo el compilador o interprete del lenguaje.

En el estilo de programación estructurada, un programa se ve como una serie de rutinas que se ejecutan para resolver un problema. La inicialización de variables, la destrucción de estas así como la liberación de los recursos utilizados por una rutina se les llama también; el problema de inicialización y limpiado (Initialization and cleanup)

4.1 Conceptos de métodos constructor y destructor.

En la programación orientada a objetos, de lo que se trata es que un programa se vea como un conjunto de objetos comunicándose por medio de mensajes, los objetos son la parte primordial de un problema, como lo fueron las rutinas en la programación estructurada. Cuando un objeto necesita crearse para poderlo utilizar dentro de un programa, es muchas de las veces, necesario inicializar el estado de este, esto consiste en inicializar los elementos que lo conforman (datos) y ejecutar una serie de tareas para preparar el ambiente para que el objeto sea utilizado por el programa. A esta tarea se le llama inicialización del objeto, dicha inicialización puede ser por invocación implícita o explícita. La invocación explícita requiere de un método que sea invocado directamente por el programador una vez creado el objeto. La invocación implícita es automática, y ocurre al momento de la creación del objeto y no necesita ser invocada por el programador.

Un constructor es un método cuya invocación es implícita, lo cual no requiere invocación por parte del programador, se encarga de inicializar el ambiente para que el objeto pueda ser utilizado por el programa.

Cuando un objeto ya no es necesario por parte de un programa, necesita liberar los recursos computacionales que este utilizando, principalmente la

memoria y autodestruirse. Probablemente algunos objetos necesiten hacer persistente su estado, mandar mensajes a otros objetos, etc.

Un destructor es un método cuya invocación es implícita y se encarga de limpiar la memoria y autodestruir al objeto que lo contiene.

Tanto el constructor, como el destructor actúan como el prólogo y el epílogo de una rutina, con la diferencia que el constructor y el destructor están asociados a objetos y no a rutinas.

4.2 Declaración de métodos constructor y destructor.

Java ofrece un constructor y un destructor por defecto para cada clase que se crea y no es necesario declararlo, pero también provee mecanismos para declarar constructores y un destructor en cada clase. La sintaxis para declarar un constructor es la siguiente:

```
public nombre_de_clase([lista de parámetros]){\n    // Cuerpo del constructor\n}
```

Como se puede observar el constructor recibe el mismo nombre de la clase, no tiene valor de retorno y puede tener lista de parámetros.

Ejemplo: Crear una clase para manejar una fecha, deberá tener un constructor que inicialice la fecha al 1ro. De Enero de 1900.

```
public class fecha{\n    private int dia;\n    private int mes;\n    private int año;\n\n    public fecha( ){ // Constructor declarado\n        dia=1;\n        mes=1;\n        año=1900;\n    }\n\n    public String get_fecha(){ // Método recuperador\n        return ""+dia+"/"+mes+"/"+año;\n    }\n\n    public static void main(String[ ] args){\n        fecha d= new fecha( ); // Crea un objeto fecha e invoca al constructor implícitamente\n        System.out.println(d.get_fecha( ));\n    }\n}
```

El constructor es invocado cuando un nuevo objeto es creado, en este ejemplo el operador new se encarga de crear un objeto de la clase fecha (new fecha()), y en ese momento invoca al constructor antes de asignar a la variable el objeto creado.

Un destructor es el método invocado al eliminarse un objeto. En algunos lenguajes, como C++, el programador es el responsable de eliminar los objetos que no utiliza destruyéndolos. En dichos lenguajes si un objeto no es destruido y queda “inaccesible” permanece en memoria de una manera “inútil”. En Java no hay destructores y el programador no debe preocuparse de eliminar sus objetos.

A lo largo de la ejecución de un programa Java, pueden “perderse” las referencias a los objetos. Una vez que un objeto no puede ser referenciado es inaccesible y es, simplemente, “basura”. Para recuperar la memoria que la “basura” está ocupando, el entorno de ejecución de Java dispone de un recolector de basura (garbage collection). El recolector de basura trabaja en un “segundo plano” y el programador desconoce cuando eliminará de memoria objetos inservibles. Para eliminar dichos objetos el recolector invoca para cada uno de ellos el método finalize.

El método finalize es un método de la clase Object y, por tanto, heredable y redefinible en nuestras propias clases. Puesto que podemos dotar de un comportamiento específico al método finalize de nuestras clases, podríamos sentirnos tentados de equipararlo a un método destructor. Eso sería un error puesto que: En los lenguajes con destructores el programador controla cuándo es invocado dicho método. En Java, el programador no invoca finalize sino es el recolector de basura quien lo invoca y no sabemos cuándo va a actuar éste.

El formato para reescribir el destructor de una clase es el siguiente:

```
protected void finalize(){  
    // Cuerpo del método destructor.  
}
```

4.3 Aplicaciones de constructores y destructores

Definir una clase para manejar la hora, que tenga un constructor que inicialice la hora, indicando hora, minutos y segundos como valores enteros separados, la clase deberá tener los siguientes métodos:

- 1.- Para incrementar la hora
- 2.- Para incrementar los minutos
- 3.- Para incrementar los segundos
- 4.- Para modificar la hora
- 5.- Para modificar los minutos
- 6.- Para modificar los segundos
- 7.- Para obtener la hora
- 8.- Para obtener los minutos
- 9.- Para obtener los segundos
- 10.- Para regresar la hora en el formato (hh:mm:ss)

```
public class hora{  
  
    private int hr;  
    private int min;  
    private int seg;  
  
    public hora(int h,int m,int s){ // Constructor de la clase  
        setHora(h); // Invoca a los método modificadores directos de la clase  
        setMinutos(m); // En su lugar pudo asignarse directamente a los datos de la clase  
        setSegundos(s);  
    }  
  
    public void incHora(){ // Método modificador que incremente la hora  
        if(hr<23)  
            hr++;  
        else  
            hr=0;  
    }  
  
    public void incMinutos(){ // Método modificador que incrementa los minutos  
        if(min<59)  
            min++;  
        else{  
            min=0;  
            incHora();  
        }  
    }  
}
```

Fundamentos de programación orientada a objetos con Java

```
public void incSegundos(){ // Método modificador que incremente los segundos
    if(seg<59)
        seg++;
    else{
        seg=0;
        incMinutos();
    }
}
public void setHora(int h){ // Método modificador
    if(h>=0 && h<24)
        hr=h;
}
public void setMinutos(int m){
    if(m>=0 && m<60)
        min=m;
}

public void setSegundos(int s){
    if (s>=0 && s<24)
        seg=s;
}

public int getHora(){ // Método recuperador
    return hr;
}

public int getMinutos(){
    return min;
}

public int getSegundos(){
    return seg;
}

public String getHoraFormat(){ // Método recuperador
    return ""+hr+":"+min+":"+seg;
}

public String toString( ){ // Método recuperador, reescrito de la clase Object
    return ""+hr+":"+min+":"+seg;
}
}
```

4.4 Tipos de constructores y destructores

En Java existen 3 tipos de constructores, el inicializador static, el inicializador de elementos no static y el constructor definido para este fin, comentado anteriormente en el punto 4.2.

Los datos miembros static, no pueden ser inicializados dentro de un constructor normal, para esto se utiliza un inicializador static o el inicializador por defecto, que cada clase puede tener. Es conveniente indicar que un inicializador no es propiamente un constructor. Por ejemplo:

```
public class ejemplo{  
  
    static int x;  
    static float y= 3.45F; // Inicializador por defecto  
    public ejemplo( ){ // Constructor  
        x++; // Toma el valor que le asigno alguno de los inicializadores  
    }  
  
    static{ // Inicializador static  
        x=1;  
    }  
}
```

El inicializador static a diferencia del constructor normal no tiene nombre y solo se le antepone la palabra reservada static, un inicializador static es resuelto por el compilador, la ejecución del programa antes de crear objetos de esta clase ya ejecuto el inicializador static, y solo es ejecutado una sola vez al momento de cargar la clase por parte de la máquina virtual de Java.

De igual forma los elementos no static definidos dentro de una clase, pueden ser inicializados ya sea dentro del constructor, utilizando el inicializador por defecto o utilizando el inicializador definido por el programador.

```
public class ejemplo2{  
  
    private int x;  
    private float y= 3.45F; // Inicializador por defecto  
    private double z;
```

```
{  
    z=0.234; // Inicializador definido por el programador  
}  
  
public ejemplo2( ){ // constructor de la clase  
    x=1;  
    y=y+x;  
    z=z*y;  
}  
}
```

En este ejemplo se muestra como se pueden utilizar los inicializadores con el constructor de la clase. Los inicializadores son ejecutados antes que el constructor sea invocado, a diferencia del inicializador static, cada vez que se crea un objeto se invoca primero al inicializador por default, después al inicializador definido por el programador y por ultimo al constructor.

EJERCICIOS:

- a) De la clase cuadrado del capítulo III, sustituir el método incializar, por un constructor.
- b) De la clase usuario del juego por computadora visto en el capítulo III, reflexionar, que elementos pueden ser inicializados por un incializador escrito por el programador y cuales deben ser inicializados por el constructor, describe como quedaria la clase.

Ejemplo de uso de constructores, inicializadores y destructores.

```
/*  
 * Practica para uso de Constructor, inicializador y destructor  
 *  
 * Este ejemplo muestra el uso de constructor, inicializador y destructor así  
 * como la invocación al recolector de basura, para forzar la invocación al destructor  
 */
```

Fundamentos de programación orientada a objetos con Java

```
import javax.swing.*;
import java.util.*;

class circunferencia{

    private int radio;
    private static int ejemplares;

    public circunferencia(int r){ // Constructor de la clase
        radio=r;
        ejemplares++;
    }

    {
        radio=10;          // Inicializador de elementos no static
    }

    static{
        ejemplares=0;    // Inicializador de elementos static
    }

    protected void finalize(){ // Método que actua como destructor.
        ejemplares--;
    }

    public int getRadio(){
        return radio;
    }

    public static int getEjemplares(){
        return ejemplares;
    }
} // Fin de clase circunferencia
=====
public class practica5{
    static Random r= new Random();

    static void agregar(circunferencia [] x){
        if(x[0].getEjemplares()<x.length)
            x[circunferencia.getEjemplares()]= new circunferencia(r.nextInt());
    }

    static void eliminar(circunferencia[] x){
        if(circunferencia.getEjemplares()>0)
            x[circunferencia.getEjemplares()-1]= null;
        System.gc(); // Invoca al recolector de basura
    }

    static void imprimeCuantos(circunferencia[] x){
        JOptionPane.showMessageDialog(null,"El total de circunferencias="+
            circunferencia.getEjemplares());
    }
}
```

Fundamentos de programación orientada a objetos con Java

```
static void imprimeEjemplares(circunferencia[] x){

    int i;
    for(i=0;i<circunferencia.getEjemplares();i++)
        JOptionPane.showMessageDialog(null,"Radio "+(i+1)+" = "+x[i].getRadio());
}

public static void main(String [] args){
    circunferencia[] c = new circunferencia[20];
    int opc;
    String cad="Menu";
    cad=cad+"\n1.- Agregar circunferencia";
    cad=cad+"\n2.- Eliminar circunferencia";
    cad=cad+"\n3.- Imprimir cuantos ejemplares existen";
    cad=cad+"\n4.- Imprimir los ejemplares";
    cad=cad+"\n5.- Terminar";
    cad=cad+"\n Dar opcion:[1..5]";
    do{
        opc=Integer.parseInt(JOptionPane.showInputDialog(cad));
        switch(opc){
            case 1: agregar(c); break;
            case 2: eliminar(c); break;
            case 3: imprimeCuantos(c); break;
            case 4: imprimeEjemplares(c);
        }
    }while(opc!=5);
    System.exit(0);
}
```

Capítulo V Sobrecarga

Una de las situaciones que los estilos de programación previos a la orientación a objetos no permite, es que dos subprograma tengan un mismo nombre para referirse a el. Esto se da por razones de ambigüedad que no pueden resolver dichos lenguajes de programación; por otro lado ofrecen que los símbolos de los operadores puedan utilizarse para describir distintas operaciones; por ejemplo en C, el operador +, es utilizado para representar la adición de números enteros, la adición de números flotantes, etc. dependiendo de los operandos que participen el compilador determinará el código interno que se ejecutará, este mecanismo donde un mismo símbolo o nombre se utiliza para representar distintas operaciones recibe el nombre de sobrecarga.

5.1 Conversión de tipos.

Muchas de las operaciones que se hacen en un lenguaje de programación, requieren la coherción de tipos de datos, por las incompatibilidades que pudieran tener los datos en ciertas operaciones. Java no es la excepción, aunque es un lenguaje altamente tipificado, realiza coherción de tipos de datos.

La coherción de tipos de datos, es una operación que permite alinear dos operandos que son de distintos tipos para poder llevar a cabo una operación, la coherción la resuelve el compilador de un lenguaje. La coherción de tipos de datos permite en ciertas operaciones hacer conversiones instantáneas para que la operación pueda llevarse a cabo, dicha coherción de tipos ocurre, como se comentó anteriormente, en tiempo de compilación. Por ejemplo:

Si tenemos la siguiente declaración:

```
int limite;
float maximo;
long suma;

limite=56;

suma=limite * 736674L; // limite y la literal son de tipos diferentes, uno es int y la literal es long
// Se alinean los datos para que ambos sean long y se ejecuta la operación
// Se dice que ocurre una coherción de tipos de datos en límime.

maximo = suma-34543434L; // Tanto suma como la literal son de tipo long, así que no se requiere
// coherción de tipos de datos, se lleva a cabo la operación y en este
// momento ocurre una coherción, para alinear el long resultante a float.
```

int limite; float maximo; long suma; limite=56; suma=limite * 736674L; maximo = suma-34543434L;	Declaración de variables Asigna una literal int a una variable int. Convierte limite a long y después lo multiplica por la literal long, asigna el resultado long a una variable long. Resta la variable long a la literal long, el resultado se convierte a float y se asigna a la variable maximo.
--	---

Ejercicio:

Indicar del siguiente código que coherciones de tipo existen y si son validas en Java.

int limite; byte x; short y; double z; char w; y=10; limite=100; x=y*4; z=y*3.4F; w=y; w=limite; w=94;	
---	--

No solo en los operadores se da la conversión de tipos de datos, también cuando se realiza una invocación a un método, los parámetros reales se tienen que alinear a los parámetros formales y si no corresponden deberán ser convertidos de acuerdo a la regla de tipificación de Java.

Regla	Ejemplos:
Todo valor en una expresión donde tiene menor precisión; siempre se convierte al inmediato superior.	int long long int short int int byte int int float float

Ejercicios:

- a) Buscar más ejemplos de coherción que son validos en Java.
- b) Hacer un programa donde se muestre que los ejemplos encontrados son validos.

5.2 Sobrecarga de métodos.

Un método sobrecargado es un método que tiene varias definiciones dentro de una misma clase, esto quiere decir que el nombre del método se repite. En java las reglas para la sobrecarga implican que un método sobrecargado deberá tener como requisito diferencias en sus argumentos o parámetros, los métodos sobrecargados podrán retornar el mismo tipo de dato. Los constructores en Java son métodos que pueden ser sobrecargados.

La firma de un método es la combinación del tipo de dato que regresa, su nombre y su lista de argumentos. La sobrecarga de métodos es la creación de varios métodos con el mismo nombre pero con diferentes firmas y definiciones. Java utiliza el número y tipo de argumentos para seleccionar cuál definición de método ejecutar. Java ve diferente los métodos sobrecargados con base en el número y tipo de argumentos que tiene el método y no por el tipo que devuelve.

Ejemplo

Métodos para calcular la suma de 3 valores diferentes, de diferentes tipos da datos:

```
/* Métodos sobrecargados */
int calculaSuma(int x, int y, int z){
    ...
}

double calculaSuma(double x, double y, double z){
    ...
}

float calculaSuma(double x, double y, double z){
    ...
}

/* Error: estos métodos no están sobrecargados */
int calculaSuma(int x, int y, int z){
    ...
}

double calculaSuma(int x, int y, int z){ // Los argumentos son el mismo número y del mismo tipo
    ...
}
```

Describir una clase fecha que pueda crear objetos donde: Unos se puedan inicializar con fecha 1/01/1900, otros con fecha indicada por el programador, donde el día, el mes y el año son pasados por parámetros en valores enteros separados y otros con fecha pasada como cadena de caracteres en el formato dd/mm/aaaa.

```

public class fecha{

    private int dia;
    private int mes;
    private int año;

    public fecha(){
        dia=1;
        mes=1;
        año=1900;
    }

    public fecha(int d,int m, int a){
        dia=d;
        mes=m;
        año=a;
    }

    public fecha(String f){
        dia=Integer.parseInt(f.substring(0,2));
        mes=Integer.parseInt(f.substring(3,5));
        año=Integer.parseInt(f.substring(6,10));
    }
    :
    :
}

```

Constructor por default

Constructor sobrecargado, recibe los valores como enteros (int)

Constructor sobrecargado, recibe los valores como String.

EJERCICIO

Diseñe e implemente una clase para representar un tanque de gas, con los atributos `limite_maximo`, `limite_minimo`, `indicador_llenado`. Donde, `limite_maximo` indica el límite para llenar completamente el tanque, el `limite_minimo` indica el límite para poder ser utilizado el tanque, normalmente este valor debe ser superior a 0. El `indicador_llenado` indica la marca a donde el tanque en un momento tiene gas que debe ser siempre un valor entre `limite_minimo` y `limite_maximo` siendo incluidos estos dos valores. La clase deberá tener un constructor para crear un tanque con límites 10 y 100 y el `indicador_llenado=100`; otro constructor para crear un tanque con los límites pasados como parámetros y el `indicador_llenado` ponerlo al `limite_minimo`. Los métodos que debe tener la clase son; `getLimiteMinimo`, `getLimiteMaximo`, `getIndicador`, `llena_tanque`, `getFaltanteParaLlenar`, `sacarGas`, `AlmacenarGas`.

5.3 Sobrecarga de operadores.

Aun cuando en Java los operadores están sobrecargados, no provee mecanismos al programador para que pueda construir sus propias operaciones con los operadores del lenguaje; en C++ si se provee el mecanismo con la palabra reservada operator.

```
class vector{  
  
    vector operator + (vector x){ // Sobre carga el operador +  
        :  
    }  
}
```

La invocación al operador sobrecargado seria la siguiente:

```
void main( ){  
  
    vector a,b;  
  
    b= a+b; // Invoca el método que sobrecarga al operador (+). Como se fuera una invocación  
             // b = a.+(b); El objeto a invoca al método + pasando como argumento b.  
}
```

Capítulo VI Herencia.

La programación orientada a objetos provee una serie de elementos que son indispensables para construir software de alta calidad. La abstracción así como la herencia son dos de los conceptos fundamentales, en este capítulo se abordará la herencia como mecanismo de abstracción y reutilización para la construcción de software.

6.1 Introducción a la herencia.

La herencia es el mecanismo de la programación orientada a objetos, que permite la reutilización de componentes de software, utilizando la extensión y la reescrituración de métodos.

Se dice que toda clase definida por el programador en Java, por default, es una subclase de la clase Object.

La clase Object ofrece una serie de elementos que pudieran ser utilizados por las clases que se crean; dichos elementos son los siguientes:

ELEMENTO	DESCRIPCION
protected Object clone()	Crea y retorna una copia del objeto actual.
public boolean equals(Object obj)	Indica si un objeto es igual al objeto actual.
protected void finalize()	Es el método que toda clase requiere para ser invocado por el recolector de basura.
public Class getClass()	Retorna en tiempo de ejecución la clase de un objeto.
public int hashCode()	Retorna el código hash del objeto actual
public void notify()	Notifica al hilo de ejecución del objeto si es que sucedió una espera a través del método wait().
public void notifyAll()	Notifica a todos los hilos de ejecución del objeto si es que sucedieron varias esperas a través del método wait().
public String toString()	Retorna una cadena que represente al objeto.
public void wait()	Avisa que un hilo de ejecución del objeto se espere.
public void wait(long timeout)	Avisa que un hilo de ejecución del objeto espere n milisegundos.

ELEMENTO	DESCRIPCION
void wait(long timeout, int nanos)	Avisa que un hilo de ejecución del objeto espere n milisegundos.

Ejemplo que muestra el uso de los métodos de la clase Object

```

import java.lang.*;

public class demostracion implements Cloneable{
    int numero;
    String nombre;

    public demostracion (int n,String nm){      // Constructor de la clase
        numero=n;
        nombre=nm;
    }

    public static void main(String[] args) {
        demostracion d1= new demostracion(25,"Susana Gonzalez");
        demostracion d2;

        try{
            d2 = (demostracion)d1.clone(); // Invoca a método de la clase Object
            d2.numero++;
            System.out.println(d2.nombre+" "+d2.numero);
            System.out.println(d1.equals(d2));
            System.out.println(d1.hashCode()+" "+d2.hashCode());
            System.out.println(d1==d2);
            System.out.println(d1.toString()+" "+d2.toString());
        }
        catch(CloneNotSupportedException e){
            System.err.println("Error: No existe clone");
        }
        d2 = d1;
        System.out.println(d2.nombre+" "+d2.numero);
        System.out.println(d1.equals(d2));
        System.out.println(d1.hashCode()+" "+d2.hashCode());
        System.out.println(d1==d2);
        System.out.println(d1.toString()+" "+d2.toString());
    }
}

```

En la herencia los elementos de una clase son heredados a otras clases donde se requiera reutilizarlos, los mecanismos de herencia en Java permiten que todos los elementos de una clase se hereden a otra aunque algunos de ellos no sean accesibles o visibles directamente.

6.2 Herencia simple.

La herencia simple es utilizada cuando una clase nueva tiene una sola clase de donde proviene sus elementos a reutilizar y no existe alguna otra clase que intervenga, en Java solo existe este tipo de herencia.

6.3 Herencia múltiple.

La herencia multiple es utilizada cuando una clase nueva tiene dos o mas clases de donde provienen sus elementos a reutilizar, el número de clases de donde se extiende es delimitado por el lenguaje de programación. Java no ofrece mecanismos de forma directa utilizando clases; para implementar la herencia multiple, como es el caso de C++ que si la ofrece, Java provee la interfaz, una interfaz es la declaración de elementos sin su implementación, estos quedaran a cargo de la clase que implemente dichos elementos, una clase en Java puede implementar varias interfaz aunque solo puede extender una sola clase.

6.4 Clase base y clase derivada.

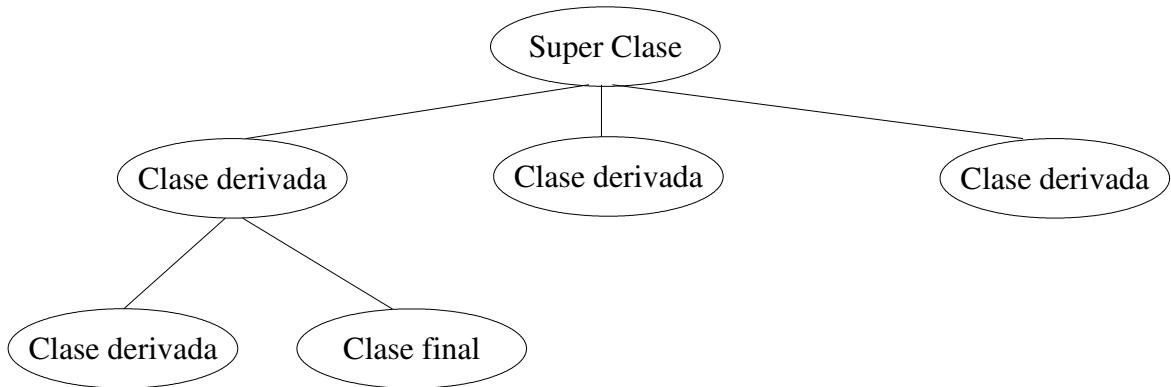
La herencia permite formar lo que se llama jerarquías de clases, para lo cual hay que diferenciar los distintos tipos de clases que se pueden observar en una jerarquía.

Una clase base es la clase que se extiende, heredando sus componentes a otras clases, en el caso de Java, la clase Object es la clase base de todas las clases.

Una clase derivada es la clase que extiende, heredando los componentes de una clase base, en el caso de Java, toda clase definida por el programador es una clase derivada de la clase Object.

Existen otros terminos que se utilizan para definir conceptos dentro de una jerarquía de clases, así tenemos que una super clase es la clase que inicia cualquier jerarquía de clases. Una clase derivada también recibe el nombre de clase hija y la clase base recibe el nombre de clase madre; dentro de una jerarquía de clases una clase hija, puede fungir como clase madre. Las clases al final de una generación pueden ser clases finales, esto quiere decir clases que no

se pueden extender.



6.4.1 Definición.

La herencia en Java se describe utilizando la palabra **extends** para clases e **implements** para interfaces.

6.4.2 Declaración.

Para declarar una clase derivada en Java se utiliza palabra reservada `extends` seguido de la clase base.

```
[ambito] class identificador extends claseBase{  
}
```

Ejemplo:

```
public class base{  
}  
  
public class derivada extends base{  
}
```

Fundamentos de programación orientada a objetos con Java

Uno de los ejemplos más típicos de Java son los applets, estos son pequeños programas que corren en contenedores de los visualizadores (browser's) de internet como Internet Explorer, Netscape Navigator, Konkeror, Mozilla, FireFox, etc. Un programa applet es un programa que se carga desde un servidor de internet y se ejecuta incrustado en el visualizador, aunque se pueden construir applets que se cargan de manera local y se ejecutan en la misma máquina.

```
import javax.swing.*;  
import java.awt.*;  
  
public class EjemploApplet extends JApplet { // Applet que se extiende de la clase JApplet  
  
    public void paint(Graphics g){  
  
        g.setColor(Color.BLUE);  
        g.draw3DRect(100,10,25,150,true);  
        g.drawString("Esto es un Applet",50,200);  
        g.drawOval(100,250,150,50);  
    }  
}
```

Este programa construye una clase llamada EjemploApplet que se extiende de la clase JApplet del paquete javax.swing.*. Reescribe el método paint que es heredado de JApplet y utiliza la funcionalidad de la clase base. Para poder ejecutar este applet, es necesario compilar el archivo y construir el documento contenedor, que será utilizado dentro del visualizador de internet, normalmente es una página web o también conocido como documento html.

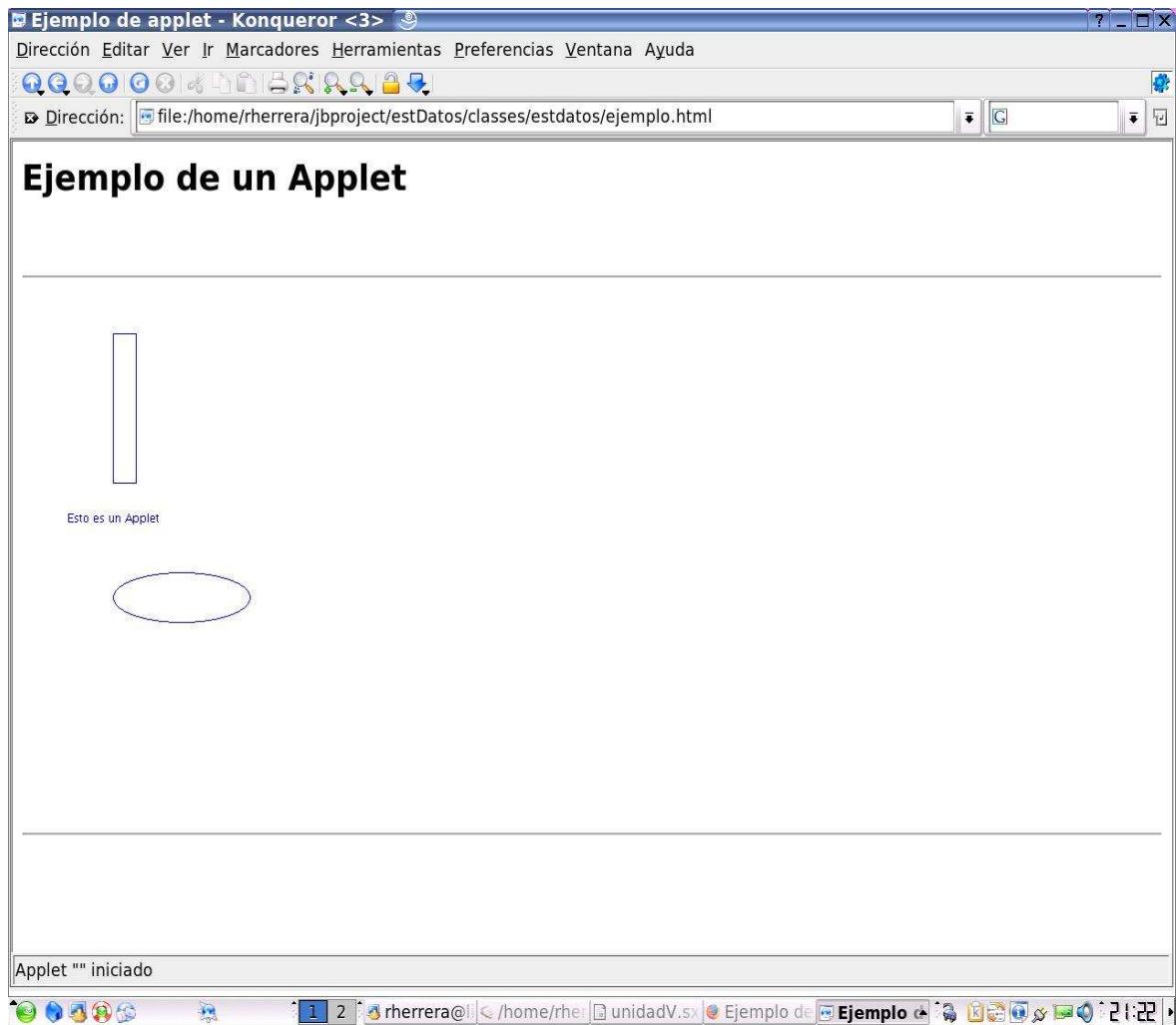
```
<html>  
  <head>  
    <title>Ejemplo de applet </title>  
  </head>  
  <body>  
    <h1> Ejemplo de un Applet</h1>  
    <br><hr><br>  
    <applet code="EjemploApplet.class" width=500 height=500>  
    </applet>  
    <br><hr><br>  
  </body>  
</html>
```

Fundamentos de programación orientada a objetos con Java



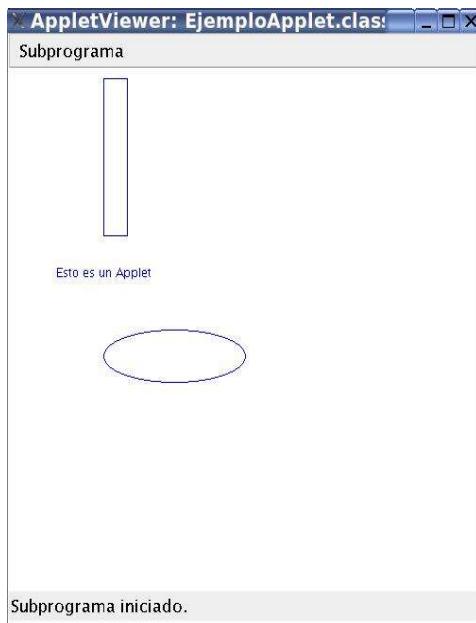
Applet corriendo en FireFox

Fundamentos de programación orientada a objetos con Java



Applet corriendo en Konkeror

Fundamentos de programación orientada a objetos con Java



Applet corriendo en el appletviewer

Otro ejemplo clásico de herencia es cuando desarrollamos aplicaciones basadas en ventanas (GUI), el siguiente ejemplo muestra el poder de la herencia utilizando swing.

```
import javax.swing.*;
import java.awt.*;

public class EjemploSwing extends JFrame { // Clase heredade de la clase JFrame

    public static void main(String[] args) {
        EjemploSwing e = new EjemploSwing();
        e.setTitle("Ejemplo de Ventana");
        e.setSize(600,600);
        e.setCursor(Cursor.E_RESIZE_CURSOR);
        e.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        e.show();
    }
}
```

Este ejemplo solo funciona con la versión 1.4.2 del JSDK en adelante, para versiones anteriores utilizar el siguiente código:

Fundamentos de programación orientada a objetos con Java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class EjemploSwing2 extends JFrame {

    public static void main(String[] args) {
        EjemploSwing e = new EjemploSwing();
        e.setTitle("Ejemplo de Ventana");
        e.setSize(600,600);
        e.setCursor(Cursor.E_RESIZE_CURSOR);
        e.addWindowListener(new WindowAdapter()
            { public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
        e.show();
    }
}
```



Esta es la salida que produce cualquiera de los dos códigos.

6.5 Parte protegida.

Todos los elementos (datos y métodos) de una clase base son heredados a la clase derivada, se tiene que considerar, que los elementos privados no son visibles por la clase derivada, solo son visibles los elementos públicos y los protegidos.

Un elemento protegido (declarado `protected`) tiene como función ser de uso exclusivo por objetos de la misma jerarquía de clases, no es visible fuera de la jerarquía de clase donde este declarado.

6.5.1 Propósito de la parte protegida.

El propósito de cualquier elemento declarado protegido es ocultar detalles de implementación fuera de la jerarquía de clases, es muy útil cuando queremos que elementos de una clase base, principalmente datos, sean accesibles directamente por una clase derivada pero no por cualquier otra clase, por ejemplo:

```
public class A{  
    private int x;  
    protected float y;  
  
    public void setX(int x){  
        this.x=x;  
    }  
    public void setY(float y){  
        this.y=y;  
    }  
  
    public int getX(){  
        return x;  
    }  
  
    public float getY(){  
        return y;  
    }  
}  
  
public class B extends A{  
  
    public float getYCuadrada(){  
        return y*y;  
    }  
}
```

En este ejemplo el dato `y`, es accesible por la clase `B` ya que es una clase derivada de la clase `A`. El dato `y` está declarado `protected`, lo cual no podría suceder con `x`.

6.6 Redefinición de los miembros de las clases derivadas.

Una clase derivada puede redefinir los miembros de una clase base, siempre y cuando la funcionalidad en la clase base no sea la necesaria para la clase derivada o en algunos casos la funcionalidad no es la completa y se requiere que se realicen otras tareas aparte de las definidas en la clase base.

La redefinición de miembros puede hacer que los miembros de la clase base no sean visibles ni accesibles en la clase derivada, para poder utilizar los miembros declarados en la clase base que han sido redefinidos en la clase derivada se utiliza la referencia **super**; esta referencia al igual que **this**, esta disponible para todas las clases.

Una redefinición que parámetros diferentes no oculta la definición en la clase base, se dice que solo se sobrecarga el método original.

```
class base{  
  
    private int n;  
    public base(int n){  
        this.n=n;  
    }  
    public int getCuadrado( ){ // Definición en clase base  
        return n*n;  
    }  
}  
  
class derivada extends base{  
  
    private int x;  
  
    public derivada( int x, int n){  
        super(n); // Invoca al constructor de la clase base  
        this.x=x;  
    }  
  
    public int getCuadrado( ){ // Redefinición en clase derivada oculta la definición de la clase base  
        return x*x;  
    }  
}
```

Fundamentos de programación orientada a objetos con Java

```
public int getXNCuadrado( ){
    return x*super.getCuadrado( ); // invoca al método de la clase base oculto, por redefinición
}

}

public class ejemploH{

public static void main(String [] args){
    derivada a= new derivada(3,6);

    System.out.println(a.getCuadrado());
    System.out.println(a.getXNCuadrado());
}
}
```

En este ejemplo el método getCuadrado de la clase base, se oculta en la clase derivada, pero desde esta se puede referenciar utilizando la referencia super.

6.7 Clases virtuales y visibilidad.

Las clases en Java no son virtuales, solo los métodos se permite que sean virtuales, un método virtual es aquel que permite redefinición en una clase derivada, por default todos los métodos no static en Java son virtuales, ya que permiten su redefinición en una clase derivada, lo que se tiene que observar es la visibilidad que esta redefinición tendrá en la clase derivada y en todos los objetos creados de la clase derivada.

Un método static no es virtual ya que pertenece a la clase y aunque se hereda a la clase derivada una redefinición del método no oculta el acceso al método en la clase base. Es conveniente hacer notar que un método static no puede hacer uso de las referencias this ni super.

6.8 Constructores y destructores en clases derivadas.

En Java los destructores no tienen un uso muy importante, así que en la herencia el uso de estos también no tiene mucha relevancia, sin embargo como se observó en el ejemplo del punto 6.6 el constructor de una clase derivada puede invocar al constructor de una clase base utilizando la referencia super con los argumentos que el constructor de la clase base requiera o solicite.

Los método redefinidos en las clases derivadas pueden de igual forma

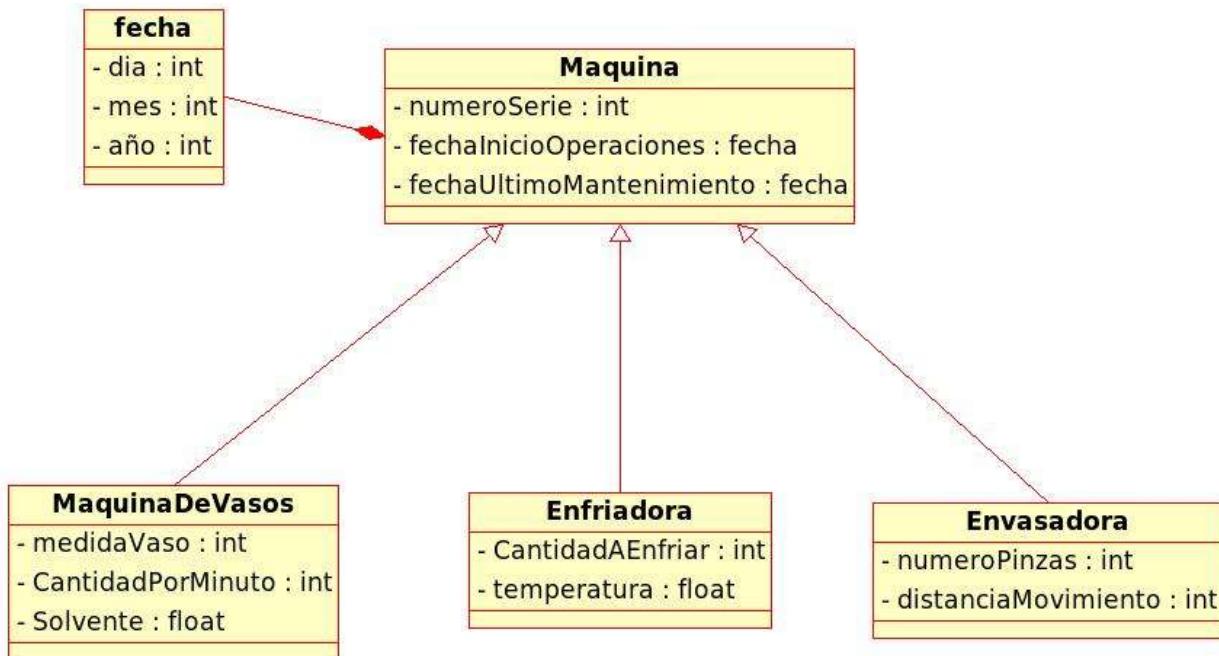
invocar a los métodos de la clase base anteponiendo la palabra super, seguida por el nombre del método.

6.9 Aplicaciones.

La herencia es util cuando en una aplicación queremos construir generalizaciones y espacializaciones. Esto quiere decir que se descubre en el marco de referencia (framework) del problema estas dos características, la generalización permite describir una clase que generaliza las clases de objetos de un problema y la especialización describe las clases especializadas de la generalización encontrada.

Por ejemplo, suponer que en un problema se tienen las máquinas que se utilizan para fabricar envases de plastico. Todas las máquinas se describen por un número de serie, fecha de inicio de operaciones, fechas de último mantenimiento; cada máquina tiene una utilidad específica; así tenemos las máquinas que fabrican vasos, de estas, aparte se describe la medida de vaso, el número de vasos por minuto y la cantidad de solvente plastico para fabricar los vasos. Otras máquinas se dedican a enfriar los productos que se fabrican de estas se describe cantidad de piezas que alcanza a enfriar por minuto y la temperatura de enfriamiento, por último tenemos las máquinas de envasado, estas se encargan de guardar los productos en cajas de carton para enviarlos al almacen de productos terminados, de estas máquinas se describe el número de pinzas que tiene para tomar y envasar el productor terminado y la distancia de movimiento máximo expresada en centímetros.

Aquí podemos descubrir los dos elementos, una generalización y tres especializaciones, utilizaremos un diagrama UML para representar esto, como sigue:



```

public class fecha{
    private int dia;
    private int mes;
    private int año;
    :
}

public class Maquina{
    private int numeroSerie;
    private fecha fechaInicioOperaciones;
    private fecha fechaUltimoMantenimiento;
    :
}

public class MaquinaDeVasos extends Maquina{
    private int medidaVaso;
    private int CantidadPorMinuto;
    private float Solvente;
    :
}
  
```

Fundamentos de programación orientada a objetos con Java

```
public class Enfriadora extends Maquina{  
    private int CantidadAEnfriar;  
    private float temperatura;  
    :  
}
```

```
public class Envasadora extends Maquina{  
    private int NumeroPinzas;  
    private int distanciaMovimiento;  
    :  
}
```

Capítulo VII Polimorfismo y reutilización

7.1 Concepto del polimorfismo

El polimorfismo al igual que la abstracción y la herencia, son conceptos claves en la programación orientada a objetos. El polimorfismo es el concepto que permite que los objetos tengan comportamientos de diferentes clases de objetos.

Todo lenguaje orientado a objetos debe ofrecer este mecanismo, Java no es la excepción. Un objeto polimórfico es aquel que tiene la cualidad de transformarse en tiempo de ejecución en un objeto diferente al original, siguiendo ciertas reglas que el lenguaje impone a esta clase de objetos. La transformación no quiere decir que el objeto cambia de estado sino que se convierte en un objeto de otra clase. Java impone restricciones para que un objeto sea polimórfico.

Un objeto polimórfico debe ser creado desde una jerarquía de clases previamente creada, esto quiere decir que el polimorfismo no se puede dar si no existe la herencia. Al menos para que exista polimorfismo en un objeto debe existir un método en la clase base de la jerarquía y que este sea abstracto.

Un método abstracto es aquel que solo tienen definición pero no implementación, esta última queda a cargo de alguna(s) subclase(s).

Para implementar el polimorfismo los lenguajes orientados a objetos utilizan el concepto de unión (binding). Muchos de los elementos de un lenguaje tienen un enlace o unión en algún momento en el tiempo; los tiempos de un programa son: tiempo de compilación y tiempo de ejecución, cualquier declaración de variable se dice que la unión entre el identificador y el tipo de dato se da en tiempo de compilación, pero la unión entre un valor y la variable se da en tiempo de ejecución; las uniones que ocurren en tiempo de compilación se les conoce como uniones tempranas (early binding) y a las uniones que se dan en tiempo de ejecución se les llama uniones tardías (late binding).

La resolución de a qué método se tiene que invocar cuando se hace referencia a este, se resuelve normalmente en tiempo de compilación, en los estilos de programación previos a la programación orientada a objetos solo puede ser así. En la programación orientada a objetos los lenguajes deben proveer mecanismos que le permitan resolver la invocación a ciertos métodos en tiempo de ejecución, esto para soportar el polimorfismo, el siguiente ejemplo muestra invocaciones a métodos resueltos en tiempo de compilación:

Fundamentos de programación orientada a objetos con Java

```
import java.util.*  
  
public class nuevo{  
  
    int a[ ] = new int [10];  
    Random r= new Random( );  
  
    void llena( ){  
        int i;  
        for(i=0;i<a.length;i++)  
            a[i]= r.nextInt(100);  
    }  
  
    public static void main(String [] args){  
        nuevo n= new nuevo( );  
        n.llena(); // El compilador resuelve a que método va a invocar.  
    }  
}
```

Aun cuando existiera sobrecarga esto de todas formas funciona:

```
import java.util.*  
  
public class nuevo{  
  
    Random r= new Random( );  
  
    static void llena( int x[ ]){  
        int i;  
        for(i=0;i<x.length;i++)  
            x[i]= r.nextInt(100);  
    }  
  
    static void llena(float x[ ]){  
        int i;  
        for(i=0;i<x.length;i++)  
            x[i]= r.nextFloat();  
    }  
  
    public static void main(String [] args){  
        int a[ ]= new int[10];
```

```
float b[ ]= new float[20];
nuevo n= new nuevo( );
n.llena(a); // El compilador resuelve a que método va a invocar.
n.llena(b); // Idem. el compilador resuelve que método va a invocar.

}
```

7.2 Clases abstractas.

Java provee al igual que otros lenguajes la capacidad de especificar métodos y clases abstract. Los métodos abstract como se comentó anteriormente son aquellos que están declarados sin especificar su implementación (cuerpo), ya que esta se realizará en las subclases donde se requiera implementar dicho método.

7.2.1 Definición.

Una clase abstract es aquella que al menos un método es declarado abstract, la sintaxis para declarar una clase abstract es la siguiente:

```
[public] abstract class nombreDeLa Clase {  
}
```

Para declarar un método abstract se sigue la siguiente sintaxis:

```
[ambito] abstract nombreDeMétodo ([argumentos]);
```

Suponer que tenemos una clase genérica llamada figura y en esta se declara un dato miembro para indicar el número de lados y dos métodos para obtener el área y el perímetro.

```
public abstract class figura {  
  
    private int lados;  
  
    public figura(int l){ // Constructor de la clase  
        lados=l;  
    }
```

Fundamentos de programación orientada a objetos con Java

```
public int getLados(){ // Método normal implementado.  
    return lados;  
}  
  
public abstract int getArea(); // Método abstract no implementado  
public abstract int getPerímetro(); // Método abstract no implementado  
}
```

7.2.2 Redefinición.

Los métodos declarados abstract en una clase deben ser implementados en una subclase, sino esta debe ser también abstract. La implementación de un método hace que se redefina el método.

Suponer la clase anterior y se crean 3 subclases, una llama cuadrado, otra trianguloEquilatero y otra triangulo.

Fundamentos de programación orientada a objetos con Java

```
public class cuadrado extends figura { // Declaración de subclase de la clase figura

    private int tamañoLado;

    public cuadrado(int tl){
        super(4); // Invocación al constructore de la clase base
        tamañoLado=tl;
    }

    public int getTamañoLado(){
        return tamañoLado;
    }

    public int getArea(){          // Implementación del método abstract
        return tamañoLado*tamañoLado;
    }

    public int getPerímetro(){ // Implementación del método abstract
        return tamañoLado*getLados();
    }
}
```

En este ejemplo se observar la implementación de los métodos abstractos declarados en la clase base llamada figura.

A continuación se muestra el ejemplo de la clase tirangulo equilatero, donde se muestra como se implementa de igual forma los métodos abstractos haciendo una redefinición de los métodos.

Fundamentos de programación orientada a objetos con Java

```
public class trianguloEquilatero extends figura {  
  
    private int base;  
    private int altura;  
  
    public trianguloEquilatero(int b,int a){  
        super(3);  
        base=b;  
        altura=a;  
    }  
  
    public int getBase(){  
        return base;  
    }  
  
    public int getAltura(){  
        return altura;  
    }  
  
    public int getArea(){  
        return base*altura/2;  
    }  
  
    public int getPerímetro(){  
        return base*getLados();  
    }  
}
```

Y por último se muestra la clase triangulo.

Fundamentos de programación orientada a objetos con Java

```
public class triangulo extends figura {  
  
    private int lado1;  
    private int lado2; // Base  
    private int lado3;  
    private int altura;  
  
    public triangulo(int l1,int l2,int l3,int a){  
        super(3);  
        lado1=l1;  
        lado2=l2;  
        lado3=l3;  
        altura=a;  
    }  
  
    public int getLado1(){  
        return lado1;  
    }  
  
    public int getLado2(){  
        return lado2;  
    }  
  
    public int getLado3(){  
        return lado3;  
    }  
  
    public int getAltura(){  
        return altura;  
    }  
  
    public int getArea(){  
        return lado2*altura/2;  
    }  
  
    public int getPerímetro(){  
        return lado1+lado2+lado3;  
    }  
}
```

Así podemos observar como se redefine un método abstract en las subclases, pero, ¿dónde está el polimorfismo?, hasta este momento no se ha tratado a este, ya que lo único que se ha realizado es la implementación de métodos abstract declarados en una clase abstract.

El polimorfismo se logra utilizando métodos abstract, ya que es una necesidad que tiene el polimorfismo para hacerse presente. El polimorfismo se define como la propiedad de objetos comportándose como otros objetos y bueno ¿que tienen que ver los métodos abstract con el polimorfismo?

Para que un objeto tenga un comportamiento diferente debe, en tiempo de ejecución hacer un binding con el o los métodos que realmente requiere para funcionar, para esto mostraremos un ejemplo de aplicación con la jerarquía de clases expuesto anteriormente.

```
import java.util.*;
import javax.swing.*;

public class Ejemplo {

    static Random r= new Random();

    //=====
    static cuadrado generaCuadrado(){
        cuadrado c;
        c= new cuadrado(r.nextInt(100));
        return c;
    }
    //=====
    static trianguloEquilatero generaTrianguloEquilatero(){
        trianguloEquilatero te;
        te= new trianguloEquilatero(r.nextInt(100),r.nextInt(100));
        return te;
    }
    //=====
    static triangulo generaTriangulo(){
        triangulo t;
        t= new triangulo(r.nextInt(100),r.nextInt(100),r.nextInt(100),r.nextInt(100));
        return t;
    }
}
```

Fundamentos de programación orientada a objetos con Java

```
//=====
public static void main(String[ ] args){

    figura [ ] x; // Declara una arreglo de la clase abstract
    int i,n;

    n= Integer.parseInt(JOptionPane.showInputDialog("Dar número de figuras:"));
    x= new figura[n]; // Crea el arreglo de la clase abstract

    ===== Genera las figuras aleatoriamente =====
    for(i=0;i<x.length;i++){
        n=r.nextInt(3);
        switch(n){
            case 0: x[i]=generaCuadrado();break; // Como el arreglo es de clase figura
            case 1: x[i]=generaTrianguloEquilatero();break; // No importa el objeto que se le
            case 2: x[i]=generaTriangulo();break; // Asigne siempre y cuando esten
        }                                         // en la misma jerarquía de clases
    }

    ===== Imprime las areas, perimetros y lados de cada figura en el arreglo
    String c="Lados  Area  Perímetro";
    for(i=0;i<x.length;i++)
        c=c+"\n"+x[i].getLados()+"      "+x[i].getArea()+"      "+x[i].getPerímetro();
    // La invocación a los métodos getArea y getPerímetro lo resuelve el tiempo de ejecución
    JOptionPane.showMessageDialog(null,c);

    System.exit(0);
}
}
```

La invocación al método **x[i].getArea()** lo resuelve en tiempo de ejecución, ya que no sabe el compilador que objeto tendrá cuando se realice la invocación, así que deja para tiempo de ejecución que se resuelva a qué método tendrá que invocar. De la misma forma sucede con **x[i].getPerímetro()**.

Como `getArea` y `getPerímetro` son métodos abstractos se puede realizar esto, `getLado` es un método definido e implementado en la clase `figura` así que se resuelve en tiempo de compilación. Se dice que estos métodos son polimórficos ya que realizan distintas cosas en función a qué objetos pertenecen y son resueltos en tiempo de ejecución.

7.3 Definición de una interfaz.

Una interfaz es una clase abstracta pura, esto quiere decir que todos sus métodos no están implementados. En Java los métodos de una interfaz no utilizan la palabra `abstract`. Una interfaz que define datos por default toman las características de ser `static` y `final`. Así que no es conveniente definir datos en una interfaz.

Para declarar una interfaz se sigue la sintaxis mostrada a continuación:

```
public interface nombreDeInterfaz{  
    // Cuerpo de la interfaz  
}
```

Ejemplo:

```
public interface Arreglo {  
  
    public void Agrega(Object x);  
    public boolean esLleno();  
    public boolean esVacio();  
    public void MueveAlPrincipio();  
    public void MueveAlFinal();  
    public Object ObtieneSiguiente();  
    public boolean EstoyEnFin();  
    public boolean EstoyEnInicio();  
}
```

Como puede observarse la interfaz `Arreglo` declara una serie de métodos pero sin su implementación.

7.4 Implementación de la definición de una interfaz.

Para que una interfaz pueda ser utilizada se requiere su implementación. Toda clase en Java puede implementar todas las interfaces que requiera. Para esto utiliza la palabra reservada ***implementation***.

```
public class claseAlImplementar implementation listaDeInterfaces{  
}
```

Ejemplo: Suponer una aplicación que requiera el manejo de un arreglo de Objetos `String` para manejar una lista de nombres y se utiliza la interfaz declarada anteriormente:

Fundamentos de programación orientada a objetos con Java

```
public class ListaNombres implements Arreglo {  
  
    private String [ ] nom;  
    private int i;  
    private int pos;  
  
    public ListaNombres(int n){ // Constructor de la clase  
        nom= new String[n];  
        i=0;  
        pos=-1;  
    }  
  
    public void Agrega(Object x) { // Método abstracto de una interfaz implementado  
        if(i<nom.length){  
            nom[i++]= (String) x;  
            pos=i-1;  
        }  
    }  
  
    public boolean esLleno() {  
        boolean band=false;  
        if(i==nom.length)  
            band=true;  
        return band;  
    }  
  
    public boolean esVacio(){  
        boolean band=false;  
        if(i==0)  
            band=true;  
        return band;  
    }  
  
    public void MueveAlPrincipio(){  
        if(i>0)  
            pos=0;  
    }  
}
```

Fundamentos de programación orientada a objetos con Java

```
public void MueveAlFinal(){
    if(i>0)
        pos=i-1;

}
public Object ObtieneSiguiente(){
    Object x=null;
    if(pos<i && pos!=-1) {
        x = nom[pos];
        pos++;
    }
    return x;
}

public boolean EstoyEnFin(){
    boolean band=false;
    if(pos==i)
        band=true;
    return band;
}

public boolean EstoyEnInicio(){
    boolean band=false;
    if(pos==0)
        band=true;
    return band;
}
```

Esta clase implementa la interface Arreglo y se puede utilizar ya en una aplicación, como la siguiente:

```
public class Aplicacion {

    public static void main(String [ ] args){
        Arreglo x= new ListaNombres(5); // x declarado de tipo Arreglo que es interface pero se
                                         // Asigna objeto de la clase ListaNombres
        x.Agrega("Juan Perez");      // La invocación a este método se resuelve en tiempo de
        x.Agrega("Maria Gutierrez"); // Ejecución (late binding)
```

```
x.Agrega("Benito Lopez");

x.MueveAlPrincipio();
while(!x.EstoyEnFin()){
    String cad=(String)x.ObtieneSiguiente();
    System.out.println(cad);
}
}
```

7.5 Reutilización de la definición de una interfaz.

Una interfaz se puede declarar en cualquier clase que así lo requiera, supongamos que necesitamos una lista de aulas, donde cada aula se declara por número de salón, número máximo de alumnos y si cuenta o no con equipo de proyección.

```
public class Aula {

    private int salon;
    private int no_alumnos;
    private boolean equipo=false;

    public void setSalon(int n){
        salon=n;
    }

    public void setAlumnos(int n){
        no_alumnos=n;
    }

    public void setEquipo(){
        equipo=true;
    }

    public int getSalon(){
        return salon;
    }

    public int getAlumnos(){
        return no_alumnos;
    }
}
```

Fundamentos de programación orientada a objetos con Java

```
public boolean getEquipo(){
    return equipo;
}
}
```

Tendriamos que redefinir los métodos de la interface Arreglo y esto quedaría de la siguiente forma:

```
public class ListaAulas implements Arreglo {
```

```
private Aula [ ] nom;
private int i;
private int pos;
```

```
public ListaAulas(int n){
    nom= new Aula[n];
    i=0;
    pos=-1;
}
```

```
public void Agrega(Object x) {
    if(i<nom.length){
        nom[i++] = (Aula) x;
        pos=i-1;
    }
}
```

```
public boolean EstoyEnFin() {
    boolean band=false;
    if(pos==i)
        band=true;
    return band;
}
```

Fundamentos de programación orientada a objetos con Java

```
public boolean EstoyEnInicio() {  
    boolean band=false;  
    if(pos==0)  
        band=true;  
    return band;  
}
```

```
public void MueveAlFinal() {  
    if(i>0)  
        pos=i-1;  
}
```

```
public void MueveAlPrincipio() {  
    if(i>0)  
        pos=0;  
}
```

```
public Object ObtieneSiguiente() {  
    Object x=null;  
    if(pos<i && pos!=-1) {  
        x = nom[pos];  
        pos++;  
    }  
    return x;  
}
```

```
public boolean esLleno() {  
    boolean band=false;  
    if(i==nom.length)  
        band=true;  
    return band;  
}
```

```
public boolean esVacio() {  
    boolean band=false;  
    if(i==0)  
        band=true;  
    return band;  
}  
}
```

En la siguiente aplicación se aplica polimorfismo:

```
public class Aplicacion2 {  
  
    public static void main(String [ ] args){  
        Arreglo x= new ListaNombres(5);  
        Arreglo y= new ListaAulas(10);  
  
        x.Agrega("Juan Perez");  
        x.Agrega("Maria Gutierrez");  
        x.Agrega("Benito Lopez");  
  
        Aula a;  
        a= new Aula();  
        a.setSalon(23);  
        a.setAlumnos(30);  
        y.Agrega(a);  
  
        a= new Aula();  
        a.setSalon(24);  
        a.setAlumnos(35);  
        a.setEquipo();  
        y.Agrega(a);  
  
        a= new Aula();  
        a.setSalon(25);  
        a.setAlumnos(40);  
        a.setEquipo();  
        y.Agrega(a);  
    }  
}
```

```
x.MueveAlPrincipio();
while(!x.EstoyEnFin()){
    String cad=(String)x.ObtieneSigiente();
    System.out.println(cad);
}

y.MueveAlPrincipio();
while(!y.EstoyEnFin()){
    a=(Aula)y.ObtieneSigiente();
    System.out.println(a.getSalon()+"\t"+a.getAlumnos()+"\t"+a.getEquipo());
}
}
```

7.6 Definición y creación de paquetes / librería.

Un paquete es un conjunto de clases que son utiles para construir aplicaciones, Java ofrece una gran variedad de paquetes que pone a nuestra disposición para ser utilizados por medio de la palabra reservada import. La nomenclatura utilizada por Java para declarar y utilizar paquetes es similar a los URL's de Internet.

El significado de un paquete esta relacionado con la estructura de directorios que se tiene en el sistema operativo que se este desarrollando, es por esto el motivo de evitar especificar una ruta completa para importar paquetes. Un paquete se considera como library o biblioteca de clases, en el pueden estar declaradas interfaces, clases bases, clases abstractas, subclases y clases solas que consideremos que tienen ciertas relaciones entre si.

Para construir un paquete solo tenemos que agregar a cada archivo que se integrara en el paquete la clausula package.

```
package utils.MisClases;

public class ClaseA{
:
:
:
}
```

Indica que la clase ClaseA forma parte del paquete utils.MisClases, y esto indica la estructura de directorios donde se localiza el archivo .class para esta clase, en un sistema Windows puede significar carpeta MisClases del directorio utils, de la raíz donde se localizaran las aplicaciones.

7.7 Reutilización de las clases de un paquete / librería.

Para que una clase pueda utilizar la clase ClaseA tendrá que importar el paquete o la clase de la siguiente forma:

```
import utils.MisClases.*;  
  
public class Aplicacion{  
  
    public static void main(String [ ] args){  
        ClaseA x= new ClaseA( );  
        :  
        :  
    }  
}  
  
import utils.MisClases.ClaseA;  
  
public class Aplicacion{  
  
    public static void main(String [ ] args){  
        ClaseA x= new ClaseA( );  
        :  
        :  
    }  
}
```

Capítulo VIII Excepciones.

8.1 Definición.

Una excepción es un imprevisto que surge durante la ejecución del programa, se dice que sucede algo excepcional y que no siempre ocurrira, solo en ciertos casos. En los estilos de programación previos a la programación orientada a objetos, las excepciones eran controladas por el sistema y pocas veces se le dejaba al programador encargarse de dichas tareas, en pocas ocasiones los lenguajes ofrecian mecanismos al programador para tratar excepciones.

Los lenguajes que ofrecian mecanismos para tratar excepciones lo hacian por medio de una serie de directivas del compilador para que al generar el código se ocupara de dar tratamiento diferente a ciertas partes del programa, pero si surgia la excepción, el sistema se encargaba de atender dicha excepción para abortar la aplicación y retornar el control al sistema operativo.

8.1.1 Que son las excepciones.

Formalmente, una excepción es un error imprevisto que surge durante la ejecución del programa y muchas de las veces el error no es facilmente manejado por el programador.

El ejemplo más típico de excepciones es la división por cero. Suponer el siguiente programa:

```
public class prueba{  
    public static void main(String [] args){  
        int a=10;  
        a=a/0;  
  
        System.out.println(a);  
    }  
}
```

En este caso el programa es muy explicito ya que asume que se realizará una división por 0, esto es un error, ningún computador puede realizar esta operación, ya que no puede representar al infinito. Si compilamos el programa y lo ejecutamos da error hasta el tiempo de ejecución cuando intenta ejecutar la división.

8.1.2 Clases de excepciones, excepciones predefinidas por el lenguaje.

Cada lenguaje define sus propias excepciones, el caso particular de Java, ofrece una jerarquía de clases para el manejo de excepciones, se inicia con la clase `java.lang.Throwable` de la cual se desprenden las clases `java.lang.Error` y `java.lang.Exception`.

Cada paquete implementa las excepciones para manejar cuestiones propias que ocurren en cada clase que conforman un paquete, algunas excepciones se muestran en la siguiente tabla:

PAQUETE	EXCEPCION	DESCRIPCION
java.lang	ArithmaticException	Ocurre cuando una operación aritmética no puede ocurrir por sobreflujo, división por cero, etc.
java.lang	ArrayIndexOutOfBoundsException	Ocurre cuando se intenta acceder a posiciones fuera del límite de un arreglo.
java.lang	ClassNotFoundException	Ocurre cuando una aplicación intenta cargar una clase y no la localiza.
java.lang	NullPointerException	Ocurre cuando se intenta acceder a un elemento de un arreglo cuando no se ha creado o a un objeto que tiene el valor de null.
java.lang	NumberFormatException	Ocurre cuando se intenta convertir de una cadena a un valor numérico y no está en el formato apropiado.
java.io	EOFException	Ocurre cuando se intenta leer desde un archivo o desde un flujo de entrada cuando se encuentra en el fin del archivo.
java.io	IOException	Ocurre cuando una operación de lectura/escritura no se pudo llevar a cabo correctamente

PAQUETE	EXCEPCION	DESCRIPCION
java.io	FileNotFoundException	Ocurre cuando se intenta abrir un archivo para leerlo y no se localiza.

8.1.3 Propagación.

Todas las operaciones de un programa pueden ser susceptibles a que ocurra una excepción, cuando esta surge, el programa debe transferir su control para que la excepción se propague por el programa y no deje al sistema colapsado, pudiendo este último regresar a un estado de consistencia el sistema.

Muchos de los métodos avisan que excepciones se pueden lanzar durante su ejecución. Por ejemplo el método parseInt:

```
public static int parseInt(String s) throws NumberFormatException
```

La declaración throws indica que excepciones pueden ocurrir durante la ejecución del método, en este caso indica que es la excepción NumberFormatException.

8.2 Gestión de excepciones.

Toda operación que se sabe lanza una excepción, es conveniente gestionarla para prever que no ocurra y de ocurrir sepamos que hacer durante la ejecución del programa para que este no aborde.

Para esto, Java ofrece las cláusulas **try, catch y finally**; try indica el código que se gestionará y catch indica el manejador de excepción a ser tratado, por cada excepción a ser tratada se utiliza una cláusula catch. La cláusula **finally** es opcional y se utiliza para ejecutar código si lanza o no una excepción en el bloque try, finally es útil para liberar recursos que se pudieran quedar atrapados en una excepción.

8.2.1 Manejo de excepciones.

El siguiente código gestiona la excepción vista anteriormente.

```
import javax.swing.*;
public class ejemplo{
    public static void main(String [ ] args){
        String cad;
        int i;

        cad=JOptionPane.showInputDialog("Dar un numero");

        try{                                // Inicia gestión de excepción
            i=Integer.parseInt(cad);          // Código donde puede ocurrir
        }

        catch(NumberFormatException excepcion){ // Clausula que gestiona la excepción ocurrida
            System.out.println(excepcion.getMessage()); // Código que se ejecuta si ocurre
            System.out.println(excepcion.getLocalizedMessage()); // la excepción
            System.out.println(excepcion.getStackTrace());
            excepcion.printStackTrace();
        }
    }
}
```

8.2.2 Lanzamiento de excepciones.

Las excepciones son lanzadas de manera automática por las rutinas que han sido identificadas en la biblioteca de clases de las API de Java para este fin. Pero el programador puede lanzar dichas excepciones utilizando la palabra reservada **throw**.

El siguiente ejemplo muestra como se lanza una excepcion por parte del programador

```
import javax.swing.*;
import java.lang.*;

public class ejemplo2{

    public static int x[ ]= new int[10];
```

```
public static int getElement(int i) throws ArrayIndexOutOfBoundsException { // Avisa que
    // Una excepción puede ocurrir en el método
    if(i>x.length)
        throw new ArrayIndexOutOfBoundsException( ); // Lanza la excepción
    return x[i];
}

public static void main(String [ ] args){
    try{
        System.out.println(getElement(20));
    }
    catch (ArrayIndexOutOfBoundsException AE){
        System.err.println("Error indice fuera de rango");
    }
}
```

Como se puede observar el formato para lanzar una excepción es el siguiente:

```
throw new claseException ( );
```

claseException es el nombre de la clase que tenemos que crear un objeto para lanzar la excepción.

La palabra throw requiere por fuerza un objeto Throwable o descendiente de él. El objeto que se cree y se lance con throw debe ser de la misma clase que espera alguna de las cláusulas catch después de la cláusula try.

8.3 Excepciones definidas por el usuarios.

Los programadores aparte de utilizar las excepciones que provee el lenguaje, en cierto momento pudieran declarar sus propios gestores de excepciones.

8.3.1 Clase base de las excepciones.

Las excepciones en Java se distribuyen a partir de la clase Throwable, de ella se extienden sus capacidades para crear las excepciones particulares. De la

clase Throwable se extiende la clase Exception, se puede decir que esta es la clase que debe servir como base para construir nuestros propios gestores de excepciones.

8.3.2 Creación de un clase derivada del tipo excepción.

```
public class ArrayOverflowException extends Exception {  
  
    public ArrayOverflowException(){  
        super("No existen mas lugares en el arreglo");  
    }  
  
    public ArrayOverflowException(String mensaje){  
        super(mensaje);  
    }  
}
```

Normalmente una nueva excepción al menos debe declarar los dos constructores, como se muestra en el ejemplo anterior y lo único que hacen es inicializar el mensaje, que utilizará el manejador de excepciones al momento de que sea detectado. Supongamos que la clase anterior sirve para manejar excepción cuando se intenta agragar un elemento mas a un arreglo que no tiene capacidad para más elementos.

8.3.3 Manejo de una excepción definida por el usuario.

Suponer el ejemplo del capítulo VII, donde se requiere manejar una lista de nombres.

```
public class ListaNombres implements Arreglo {  
  
    private String [ ] nom;  
    private int i;  
    private int pos;  
  
    public ListaNombres(int n){  
        nom= new String[n];  
        i=0;  
        pos=-1;  
    }  
}
```

Fundamentos de programación orientada a objetos con Java

```
public void Agrega(Object x) throws ArrayOverflowException{ // Declara excepción a lanzar
    if(i<nom.length){
        nom[i++] = (String) x;
        pos=i-1;
    }
    else
        throw new ArrayOverflowException ( ); // Lanza excepción si ocurre el problema
}

public boolean esLleno() {
    boolean band=false;
    if(i==nom.length)
        band=true;
    return band;
}

public boolean esVacio(){
    boolean band=false;
    if(i==0)
        band=true;
    return band;
}

public void MueveAlPrincipio(){
    if(i>0)
        pos=0;
}

public void MueveAlFinal(){
    if(i>0)
        pos=i-1;
}

public Object ObtieneSiguiente(){
    Object x=null;
    if(pos<i && pos!= -1) {
        x = nom[pos];
        pos++;
    }
    return x;
}
```

Fundamentos de programación orientada a objetos con Java

```
public boolean EstoyEnFin(){
    boolean band=false;
    if(pos==i)
        band=true;
    return band;
}

public boolean EstoyEnInicio(){
    boolean band=false;
    if(pos==0)
        band=true;
    return band;
}
```

Para utilizar el gestor de excepciones declarado y el método de ha sido protegido por la excepción (Aregar). Se necesita crear la aplicación como se muestra a continuación.

```
public class Aplicacion {
    public static void main(String [ ] args){
        ListaNombres x= new ListaNombres(3);

        try{ // Avisa que puede ocurrir excepción en alguna operación
            x.Agrega("Juan Perez");
            x.Agrega("Maria Gutierrez");
            x.Agrega("Benito Lopez");
            x.Agrega("Martin Guzman");
        }

        catch(ArrayOverflowException AE){ // Gestiona la excepción definida
            System.out.println(AE.getMessage());
        }

        x.MueveAlPrincipio();
        while(!x.EstoyEnFin()){
            String cad=(String)x.ObtieneSiguiente();
            System.out.println(cad);
        }
    }
}
```

Capítulo IX Flujos y archivos.

Los archivos permiten que la información que un programa procesa se puede hacer persistente. Esto quiere decir que exista aún cuando el programa termine de ejecutarse. Muchas aplicaciones requieren que la información se haga persistencia para ser utilizada posteriormente.

Java al igual que muchos lenguajes de programación orientados a objetos, proveen mecanismos de alto nivel de abstracción para la manipulación de archivos. Java ofrece diversos mecanismos para manipular archivos, en este capítulo se tratará exclusivamente uno que se localiza en el paquete java.io, es la clase RandomAccessFile.

9.1 Definición de Archivos de texto y archivos binarios.

Algunos lenguajes como C y C++ distinguen los archivos de texto y binarios, normalmente los de texto son de información plana (nombres, edades, etc.) y los binarios son de información codificada (imágenes, videos, fotos, etc.) Para tratar a cada uno de estos tipos de archivos el lenguaje ofrece mecanismos específicos. Java por el contrario no hace una distinción tal y permite que cualquiera de estos tipos de archivos sean tratados de la misma forma. El programador es el responsable de darle sentido a lo que lee de un archivo y lo que escribe en él.

9.2 Operaciones básicas en archivos texto y binario.

Todo tipo de archivos debe ofrecer ciertas operaciones básicas que el programador pueda utilizar para manejar archivos en sus aplicaciones, algunas de estas son: crear, abrir, cerrar, leer, escribir, recorrer el archivo etc.

9.2.1 Crear.

Para crear un archivo RandomAccessFile ofrece un constructor que permite abrir y si no existe crea el archivo. El siguiente ejemplo muestra como se crea un archivo llamado datos.dat. Es importante tener cuidado ya que el archivo se crea segun la carpeta actual a menos que se describa toda la ruta.

Este constructor es de la siguiente forma:

`RandomAccessFile(String file, String mode) throws FileNotFoundException`

file: Cadena que indica el nombre del archivo

mode: Indica el modo de apertura, las cadenas validas son: "r", "rw", "rwd"

o "rws"

```
import java.io.*;

public class CrearArchivo {

    public static void main(String [ ]args){

        RandomAccessFile archivo;

        try{
            archivo = new RandomAccessFile("datos.dat", "rw"); // Crea el archivo
            archivo.close(); // Cierra el archivo
        }
        catch (FileNotFoundException e){
            System.err.println("Error el archivo no se pudo crear");
        }

        catch(IOException e){
            System.err.println("Error no se pudo cerrar el archivo");
        }
    }
}
```

9.2.2 Abrir.

Para abrir un archivo que fue previamente creado se utiliza el mismo constructor, con el mode de apertura requerido.

9.2.3 Cerrar.

Para cerrar un archivo se utiliza el método close de la clase, en el ejemplo anterior se muestra el uso de este método.

9.2.4 Lectura y escritura.

Para leer y/o escribir desde o sobre un archivo, la clase RandomAccessFile ofrece una gran variedad de métodos. Estos métodos estarán dispuestos siempre y cuando el mode de apertura esté indicado correctamente. Un archivo abierto para leer ("r") no puede realizar operación de escritura.

Algunos métodos son los siguientes:

int readInt()	char readChar()	float readFloat()	double readDouble()
boolean readBoolean()	short readShort()	byte readByte()	long readLong()
String readLine()	void writeInt(int d)	void writeChar(char d)	
void writeFloat(float d)	void writeDouble(double d)	void writeBoolean(Boolean d)	
void writeShort(short d)	void writeByte(byte d)	void writeLong(long d)	
void writeChars(String d)			

El siguiente programa muestra como se graba una lista de nombres en el archivo de datos creado anteriormente.

```
import java.io.*;  
  
public class GrabaDatos {  
  
    public static void main(String [ ]args){  
        RandomAccessFile archivo;  
        try{  
            archivo = new RandomAccessFile("datos.dat", "rw");  
            archivo.writeBytes("Miguel Dominguez\n");  
            archivo.writeBytes("Pedro Perez\n");  
            archivo.writeBytes("Fernando Fernandez\n");  
            archivo.close();  
        }  
        catch (FileNotFoundException e){  
            System.err.println("Error el archivo no se pudo crear");  
        }  
        catch(IOException e){  
            System.err.println("Error no se pudo cerrar el archivo");  
        }  
    }  
}
```

El siguiente programa muestra el programa para leer el contenido del mismo archivo.

```
import java.io.*;  
  
public class LeeArchivo {  
    public static void main(String [ ]args){  
        RandomAccessFile archivo;  
        String cad;  
        try{  
            archivo = new RandomAccessFile("datos.dat", "rw");  
            while((cad=archivo.readLine())!=null)  
                System.out.println(cad);  
            archivo.close();  
        }  
        catch (FileNotFoundException e){  
            System.err.println("Error el archivo no se pudo crear");  
        }  
        catch(IOException e){  
            System.err.println("Error no se pudo cerrar el archivo");  
        }  
    }  
}
```

9.2.5 Recorrer.

Un archivo se organiza lógicamente como una secuencia de datos, almacenados contiguamente, su representación puede ser como una cinta donde se encuentran grabados datos:

Juan\n	82,45	34	true	Miram\n	45	20	false	Josefa\n	56,5	30	true	EOF
--------	-------	----	------	---------	----	----	-------	----------	------	----	------	-----



Apuntador al archivo

Cuando un archivo que esta creado se abre con el modo “r” o “rw”, el apuntador al archivo se inicia en el primer dato a leer, cada vez que ocurre la lectura de un dato el apuntador avanza, cuando llega al final del archivo EOF (End Of File), cualquier cosa que quiera leer no procederá ya que no existirán más datos que leer. Para recorrer un archivo es necesario saber que esta clase de archivos permite recorrer los datos en forma secuencial y/o en forma aleatoria.

Un archivo se recorre en forma secuencial, dato por dato de derecha a izquierda, y se recorre en forma aleatoria moviendo el apuntador al archivo a los datos que quisieramos leer.

Para esto la clase RandomAccessFile provee otros métodos:

long length() Obtiene el tamaño del archivo en número de bytes.

void setLength(long newlength) Modifica el tamaño del archivo

long getFilePointer() Obtiene la posición relativa al inicio del apuntador al archivo

void seek(long pos) Mueve el apuntador al archivo un desplazamiento desde el inicio del archivo

9.3 Aplicaciones.

Como ejemplo se resolverá el caso de un directorio telefónico que se almacena en un archivo.

```
import java.io.*;
import javax.swing.*;

public class Directorio {

    private static String nombre;
    private static String direccion;
    private static long telefono;
    private static RandomAccessFile a;

    public static void leer_datos(){
        nombre=JOptionPane.showInputDialog("Dar nombre");
        direccion=JOptionPane.showInputDialog("Dar dirección");
        telefono=Long.parseLong(JOptionPane.showInputDialog("Dar teléfono"));
    }
}
```

Fundamentos de programación orientada a objetos con Java

```
public static void guardar_disco(){
    try{
        a = new RandomAccessFile("dir.dat", "rw");
        a.seek(a.length());
        a.writeBytes(nombre+"\n");
        a.writeBytes(direccion+"\n");
        a.writeLong(telefono);
        a.close();
    }
    catch(FileNotFoundException e){
    }
    catch(IOException e){
    }
}
```

```
public static void Agregar(){
    leer_datos();
    guardar_disco();
}
```

```
public static void Imprimir(){
    try{
        a = new RandomAccessFile("dir.dat", "rw");
        while((nombre=a.readLine())!=null){
            direccion=a.readLine();
            telefono=a.readLong();
            JOptionPane.showMessageDialog(null,nombre+"\n"+direccion+"\n"+telefono);
        }
        a.close();
    }
    catch(FileNotFoundException e){
    }
    catch(IOException e){
    }
}
```

Fundamentos de programación orientada a objetos con Java

```
public static void buscar(){
    int pos=Integer.parseInt(JOptionPane.showInputDialog("Dar registro a buscar"));
    int i=0;
    try{
        a = new RandomAccessFile("dir.dat", "rw");
        while((nombre=a.readLine())!=null){
            direccion=a.readLine();
            telefono=a.readLong();
            i++;
            if(i==pos)
                JOptionPane.showMessageDialog(null,nombre+"\n"+direccion+"\n"+telefono);
        }
        a.close();
    }
    catch(FileNotFoundException e){
    }
    catch(IOException e){
    }
}

public static void main(String [ ] args){
    String cad=" Menu Principal\n";
    cad=cad+"1.- Agregar contacto\n";
    cad=cad+"2.- Imprimir contactos\n";
    cad=cad+"3.- Localizar por posicion\n";
    cad=cad+"4.- Salir\n Dar opcion:[1..4]:\n";
    int opc;
    do{
        opc=Integer.parseInt(JOptionPane.showInputDialog(cad));
        switch(opc){
            case 1: Agregar();break;
            case 2: Imprimir();break;
            case 3: buscar();
        }
    }while(opc!=4);

    System.exit(0);
}
```