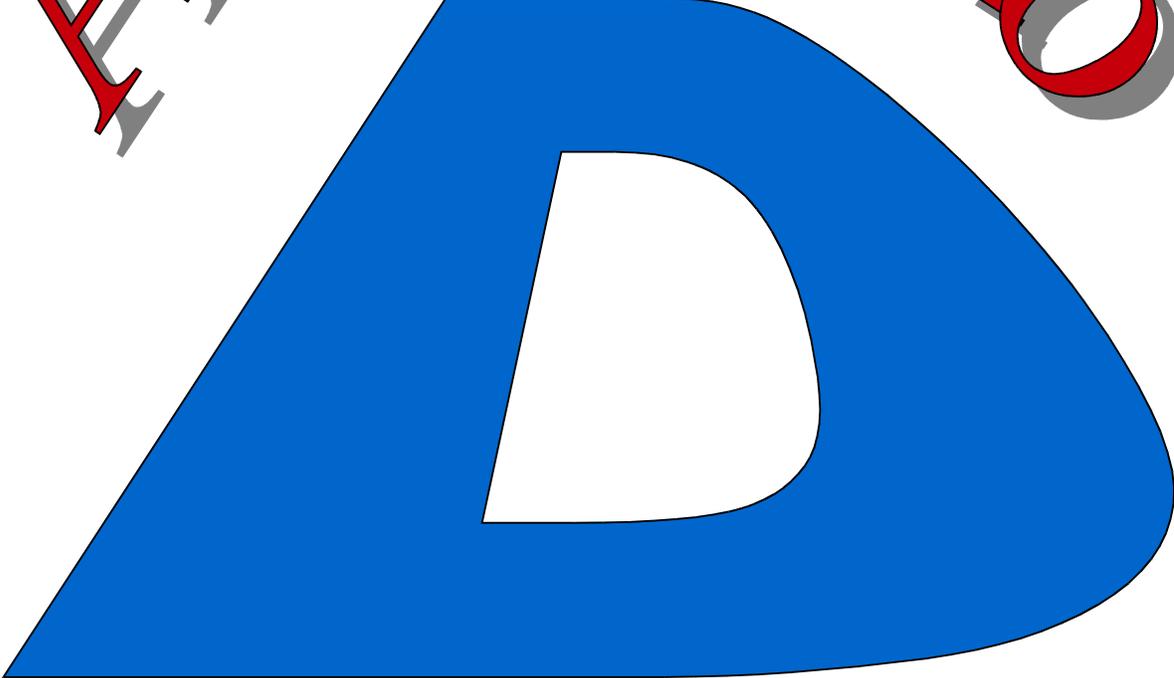


Algoritmo



Versión Draft
9.3

Estructuras de Datos en Java

Índice de contenido

Introducción a la Programación.....	8
La importancia de la lógica.....	8
La Importancia de la Información.....	11
La Informática.....	12
La computadora.....	13
El hardware.....	15
Componentes Básicos de las Computadoras.....	18
Microprocesador.....	18
Tarjeta Principal.....	18
Tarjeta de video.....	18
Memoria RAM.....	18
Disco Duro.....	18
Torre.....	19
Dispositivos.....	19
Periféricos.....	19
El Software.....	20
Los Lenguajes de Programación.....	21
Traductores de lenguaje.....	22
La programación.....	23
Historia de los lenguajes de programación.....	24
Paradigmas de Programación.....	25
El lenguaje de Programación Java.....	26
Práctica # 1 - Introducción a la Informática.....	27
Programación Orientada a Objetos.....	28
Introducción.....	28
Pensar en objetos: una analogía.....	29
Objetos y clases.....	30
Práctica # 2 - Clases y Objetos.....	32
La abstracción.....	33
¿Cómo se piensa en objetos?.....	34

Atributos y Métodos.....	36
Atributos.....	36
El dato.....	37
La Variable.....	37
Tipos de Datos en Java.....	37
Tabla de tipos de datos primitivos.....	38
Métodos.....	40
Creando nuestra primera clase utilizando BlueJ.....	42
¿Pero... qué dijimos que es un programa?.....	42
¿Qué es BlueJ?.....	42
1) Crear un proyecto.....	44
2) Crear una clase.....	44
Convenciones sobre el nombre de las clases.....	46
3) Crear objetos.....	47
El método constructor.....	51
Algunas convenciones sobre los nombres de métodos.....	53
Interactuando con objetos en BlueJ.....	55
Colecciones de Objetos.....	56
La clase ArrayList.....	56
Constructores.....	57
Métodos.....	57
Declaración de un ArrayList de enteros.....	57
Creación de un ArrayList de enteros.....	57
Añadiendo elementos al ArrayList.....	58
Conociendo el tamaño del ArrayList.....	58
Obteniendo un elemento del ArrayList.....	58
Borrando un elemento del ArrayList.....	58
Relaciones entre objetos.....	59
Composición (tiene un).....	59
Herencia (es un).....	60
Recursividad.....	63
Definición.....	63
Tipos de Recursión.....	69
Recursividad Simple.....	69
Recursividad Múltiple.....	69
Recursividad Anidada.....	69

Recursividad Cruzada o Indirecta.....	70
Reglas fundamentales de la recursividad.....	71
a) Caso base:	71
b) Progreso:.....	71
c) Regla de diseño:.....	71
d) Regla del interés compuesto:.....	72
La Pila de Recursión.....	73
Llamada a una función.....	73
Llamada a una función recursiva.....	73
Análisis de Algoritmos.....	74
Definición.....	74
Factores que influyen en la eficiencia de un algoritmo.....	75
El Hardware	75
El Software.....	75
La longitud de entrada.....	75
Análisis de Algoritmos.....	76
Elementos de un algoritmo.....	76
Enunciados simples (sentencias).....	77
Enunciados compuestos (estructuras de control).....	78
Selectivas.....	78
IF.....	78
IF-ELSE.....	80
Repetitivas.....	82
FOR.....	82
WHILE.....	84
DO - WHILE.....	85
Orden de Magnitud (Notación O grande).....	86
Definición conceptual del Orden de Magnitud.....	86
Propiedades del Orden de Magnitud.....	87
Estructuras de Datos.....	88
Definición.....	88
Operaciones.....	88
Navegar por la estructura.....	88
Búsqueda.....	88
Consulta de la información.....	88
Copia parcial o total.....	88

Prueba.....	89
Modificación.....	89
Inserción.....	89
Eliminación.....	89
Verificar si es vacía.....	89
Clasificación.....	90
Estructuras de Datos Interna y Externa.....	90
Estructuras de Datos Estáticas.....	90
Estructuras de Datos Dinámicas.....	90
Estructuras de Datos Lineales y No Lineales.....	91
Objetivos.....	91
Estructuras de Datos Lineales.....	92
Listas.....	92
Operaciones.....	92
Definición recursiva.....	93
La clase Lista.java.....	95
Constructores	95
Métodos.....	95
Verificar si es vacía.....	95
Insertar un dato al final.....	96
Insertar un dato al inicio.....	96
Insertar un dato en la k-ésima posición de la lista.....	96
Eliminar un dato de la lista.....	97
Obtener el tamaño de la lista.....	97
Buscar un dato sobre la lista.....	97
Obtener un elemento de cualquier posición de la lista.....	98
Obtener en formato cadena a la lista.....	98
Modificar el elemento de una posición dada.....	99
Eliminar el elemento de una posición dada.....	99
Genericidad o Parametrización de tipos.....	101
Definición de Lista en base a Nodos.....	103
Lista de Simple Enlace.....	103
La clase ListaSE.java.....	106
Constructor.....	106
Métodos.....	106
Verificar si es vacía.....	106

Insertar un dato al inicio.....	106
Insertar un dato al final.....	107
Obtener el tamaño de la lista.....	107
Obtener el elemento de una posición dada.....	108
Eliminar el elemento de una posición dada.....	108
Buscar un elemento en la lista.....	109
Lista de Doble Enlace.....	109
La clase ListaDE.java.....	111
Constructor	111
Métodos.....	111
Insertar un elemento al inicio.....	111
Insertar un elemento al final.....	112
Obtener los elementos de la lista en formato cadena.....	112
Eliminar y obtener el primer elemento de la lista.....	113
Eliminar y obtener el último elemento de la lista.....	113
Buscar un elemento en la lista.....	114
Buscar un elemento en la lista.....	114
Lista Circular de Simple Enlace.....	115
La clase ListaSEC.java.....	116
Constructor.....	116
Métodos.....	116
Verificar si es vacía.....	116
Insertar un dato al inicio.....	116
Insertar un dato al final.....	117
Obtener en formato cadena.....	117
Lista Circular de Doble Enlace.....	118
La clase ListaDEC.java.....	118
Constructor.....	118
Métodos.....	119
Verificar si es vacía.....	119
Insertar un dato al inicio.....	119
Insertar un dato al final.....	120
Obtener la lista en formato cadena.....	120
Pilas.....	121
Operaciones.....	121
Aplicaciones de las Pilas	124

Colas.....	125
Operaciones.....	126
Aplicaciones de las Colas	127
Estructuras de Datos No Lineales.....	128
Introducción.....	128
Árboles.....	128
Algunos Conceptos.....	129
Árboles Binarios	131
Operaciones.....	131
Definición recursiva.....	131
La clase ArbolB.java.....	132
Constructor	133
Métodos.....	133
Verificar si el árbol está vacío.....	133
Insertar un elemento en el árbol en una ubicación aleatoria.....	133
Obtener el peso del árbol binario.....	134
Obtener el tamaño del árbol binario.....	134
Buscar un elemento en el árbol binario.....	134
Calcular la altura del árbol binario.....	135
Recorrido de un árbol binario.....	135
Recorrido preorden.....	135
Recorrido enorden.....	136
Recorrido postorden.....	136
Árboles Binarios de Búsqueda.....	137
Árboles Binarios de Expresión.....	138
Bibliografía.....	139

Introducción a la Programación

La importancia de la lógica

Cuando era niño, alguna vez me coloqué mal los zapatos, en un instante en que toda mi familia estaba en un afán de salir rápido. Una tía vió que tenía puestos los zapatos al revés y me regañó fuertemente diciéndome que porqué me había puesto los zapatos así, que no tenía un poquito de lógica.



Luego de ponerme adecuadamente los zapatos, es decir, colocando el zapato donde debería ser, zapato izquierdo para el pie izquierdo, zapato derecho para el pie derecho, comencé a pensar que realmente era posible que no tuviera lógica porque me parecía increíble que yo no hubiera detectado que para colocarse correctamente los zapatos, sólo había un camino (y que además era muy sencillo). Además de esta conclusión (tan obvia) también llegué a otra (no tan obvia) y era el hecho de que es más difícil ponerse mal los zapatos que ponérselos bien o, dicho en otras palabras, es muy muy muy fácil colocarse correctamente los zapatos.

En la medida en que los años fueron pasando y fui creciendo, me dí cuenta que son muchas pero muchas las cosas obvias que por un extraño error no hacemos y vuelve a mi mente el recuerdo de mi tía diciéndome: - *Es que no tienes lógica o qué!!!*

Dediqué mi carrera universitaria al estudio de las Ciencias de la Computación, precisamente porque allí encontré porqué era tan importante aquello de la lógica. Sin embargo y luego de buscar muchas definiciones de Lógica llegué a una que en mi concepto refleja realmente el sentido de esa palabra. Pregunté a una amiga: - *¿Qué*

es para ti la lógica? Me respondió en un lenguaje muy popular: - *Pues, ehh.. la lógica es... es... es... como algo muy lógico.* De hecho su respuesta no me satisfizo. Estaba incluyendo en la definición el término a definir o sea que no me había dicho nada. Pero cuando le pregunté: - *¿Por qué te es difícil definirlo..?* Me respondió: - *Es que no es fácil definir algo tan lógico.* O sea que ella tenía clara la concepción del término, sencillamente no sabía cómo definirlo.

Pregunté a Don José, un viejo zapatero que durante varios años lo había visto llegar todas las mañanas a armar su "cambuche" desde donde atendía, para desarmarlo en las horas de la noche. El me respondió: - *Pues hombre, lo único que le puedo decir es que lógico es todo aquello que no es ilógico.* Su definición me parecía muy racional pero de hecho seguía siendo distante de lo que yo estaba esperando. Sin embargo yo veía que el proceso de armar su "cambuche" que constaba de unos nueve soportes y dos plásticos (desde donde él atendía a su clientela) le tomaba a él solamente unos diez minutos. Un día le dije que me dejara armárselo y al cabo de media hora me dijo un poco molesto: - *¿Quieres armarlo rápido..?* . Yo le respondí que sí, entonces me dijo: - *Pues póngale lógica a esos palos y verá lo sencillo que es.* Luego de un buen rato y de haber colmado la paciencia de Don José pude armarlo, pero lo que si noté fue que él tenía muy clara la definición de Lógica porque luego de armado vi que realmente era muy sencillo y que todo lo que necesitaba era ponerle lógica. Mi inquietud siguió sin resolverse porque yo quería tener una definición concreta y exacta de ese término.

Pregunté a mi profesora de Lenguaje y me entregó una excelente definición de diccionario: - *Lógica es la rama del conocimiento que nos permite determinar que algo está aprobado por la razón como bien deducido o bien pensado.* Para mí era una definición exacta y era suficiente con que apareciera en el Pequeño Larousse para que yo no la discutiera. Sin embargo me exigía más razonamientos de los necesarios para entender tal definición (al menos en esos días), pues me parecía inaudito que la definición de Lógica fuera muy compleja, es decir no fuera tan lógica. Esa misma razón era la que no me había animado a buscar una definición de diccionario sino buscar una expresión tan sencilla y tan lógica que no me exigiera muchos razonamientos.

En esa búsqueda por una definición que dejara satisfechas mis expectativas caí en las fauces de un buen matemático. De hecho, sabía que tenía que conocer y

poder definir qué era la Lógica. Cuando le pregunté al respecto me respondió diciendo: - *Lógica es la Ciencia que estudia la estructura, fundamentos y uso de las expresiones del conocimiento humano.* Realmente era la definición más exacta que yo había recibido hasta ese momento, pero al igual que la definición del Pequeño Larousse, me exigía demasiados razonamientos como para poder digerirla. No puedo discutir que considero que es la definición más exacta y precisa acerca de lo que es Lógica pero mi nivel de conocimiento no me permitía procesar adecuadamente, en esos tiempos, esta definición.

Le pregunte a alguien, un transeúnte, un desconocido qué era la Lógica y su respuesta desprevenida y silvestre me gustó porque pude entenderla muy fácilmente: - *Yo considero la lógica como una serie coherente de ideas y razonamientos.* Compartí con él dicha definición y me pareció acertada. Además que en todo mi entorno pude descubrir que todas las personas a quienes les preguntaba tenían, por lo menos, muy claro el concepto de lo que era la Lógica así en algunos casos no la pudieran definir de una manera clara.

Finalmente y luego de tantas definiciones, busqué a mi padre. Un hombre del campo para quien los avances tecnológicos le corrían por la espalda porque no eran su afán. Me miraba sin parpadear cada que yo iniciaba algunas de mis disertaciones sobre la tecnología y su relación con el mundo moderno. Para él el mundo moderno no era sino un cúmulo de problemas en vez de soluciones. Yo pensaba lo contrario. Sin embargo me arriesgué a preguntarle: - *Papá, para usted qué es la lógica...?* y él mirándome con la extrañeza de la pregunta me dijo: - *Pues es la forma más OBVIA y más FÁCIL de hacer algo.* Y ví que todas las definiciones que hasta el momento había recibido, unas provenientes de la vida cotidiana y otras extractadas de libros especializados en el tema, se resumían en ésta última. Eso es la LÓGICA. Algunos libros citan que la base para la solución de problemas a través de la computadora es tener muy buena lógica. Yo iría mas allá. Considero que la base para ello es, sencillamente, ser muy lógicos o sea poder vislumbrar el camino más obvio y más fácil para lograr un objetivo.

La lógica es la forma más OBVIA y más FÁCIL de hacer algo.



La Importancia de la Información

En sentido general, *la información* es un conjunto organizado de datos, que constituyen un mensaje sobre un determinado ente o fenómeno. De esta manera, si por ejemplo organizamos datos sobre un país, tales como: número de habitantes, densidad de población, nombre del presidente, etc. y escribimos por ejemplo, el capítulo de un libro, podemos decir que ese capítulo constituye información sobre ese país. Cuando tenemos que resolver un determinado problema o tenemos que tomar una decisión, empleamos diversas fuentes de información (como podría ser el capítulo mencionado de este libro imaginario), y construimos lo que en general se denomina conocimiento o información organizada que permite la resolución de problemas o la toma de decisiones.

En muy distintas épocas el progreso de la humanidad ha sido posible gracias a: el fuego, la rueda, la agricultura, la industria y la tecnología. Pues bien, ahora entramos en la **Era de la Información**, porque es a través de la información como llegan a nuestras vidas las grandes soluciones... para los grandes problemas.

Nuestras vidas giran en torno a la *información* que podemos percibir de nuestro entorno. La vida es una constante decisión, debemos elegir en cada momento de nuestras vidas, y las decisiones claramente deben estar basadas en información. La información se produce cuando, de alguna manera, logramos interpretar datos, es decir, señales de nuestro entorno. Por otra parte, la información debe tener una serie de características implícitas para que podemos utilizarla, debe ser fidedigna, a tiempo, etc.

Hoy en día la ciencia que se encarga del estudio de la información es la Informática.



La Informática

La informática es la Ciencia que estudia, aplica y optimiza el tratamiento eficiente de la información.

¿Sabes lo que significa el tratamiento eficiente de la información..? Sencillamente que es la ciencia que se ocupa de que la información, cualquiera que sea su procedencia o su destino, cumpla con dos objetivos:

- **Veracidad** : "Toda información debe ser verdad (es decir veraz)". De nada te sirve que vayas al Banco y solicites tu saldo e inmediatamente y sin ninguna demora te den un saldo que no corresponde a la realidad.
- **Oportunidad** : "Toda información debe llegar en el momento indicado (o sea oportunamente)". De nada te sirve que en el Banco te digan que tu verdadero saldo te lo entregan en 10 meses.

¿Porqué cada que se habla de Informática se relaciona inmediatamente el concepto con computadoras..? Pues sencillamente porque en la actualidad los computadores son los dispositivos que mejor pueden cumplir con el objetivo de la Oportunidad ya que trabajan a velocidades impresionantemente altas (millonésimas de segundo). Y quien es el encargado de cumplir con la veracidad...? Pues el ser humano que es quien planea, organiza, **programa** todo lo que la computadora va a entregar como información.



La computadora

Las computadoras u ordenadores, son herramientas esenciales en muchas áreas: industria, gobierno, ciencia, educación..., en realidad en casi todos los campos de nuestras vidas. El papel de los programas es esencial; sin una lista de instrucciones a seguir, la computadora es virtualmente inútil. Los lenguajes de programación nos permiten escribir esos programas y, por consiguiente, comunicarnos con las computadoras.



Desde un punto de vista simple, una computadora procesa datos y los convierte en información significativa.



El programa es la fuerza conductora de cualquier tarea que hace una computadora. Un programa es una lista de instrucciones detalladas que indican a la computadora lo que ha de hacer. La computadora no puede hacer nada sin un programa.

Los programas modernos producen información en muchos formatos. Estos programas reproducen música, se comunican con otras computadoras en diferentes formas. La salida se envía a la pantalla y/o a la impresora que constituyen los dispositivos más usuales para acceso a la computadora: es decir, los datos de

entrada y los datos de salida pueden ser, realmente, cualquier cosa, texto, dibujos, sonidos...

Una computadora consta de dos partes bien diferenciadas, hardware y software. El **hardware** consta de las partes físicas, tangibles de la computadora. El **software** consta de programas, también llamados aplicaciones, que contienen instrucciones que la computadora "ejecuta" o "corre".

Una computadora necesita tanto del hardware como del software para poder ser usada real y prácticamente. El hardware de la computadora sin el software que le acompaña es como si tuviéramos un libro con páginas en blanco. La portada, contraportada y las páginas interiores constituyen el hardware del libro, pero el libro no es útil sin ningún texto, el software.

Software



Hardware



El hardware

Las computadoras están compuestas en su mayoría por transistores, en donde los transistores en su nivel más básico hacen lo mismo que un interruptor, dejan o no pasar energía.



Nomenclatura:

Transistor encendido
ON = 1 = bit

Transistor apagado
OFF = 0 = bit

bit = es la mínima cantidad de información posible de ser manipulada.

Ya que el bit es muy pequeño se vio la necesidad de agruparlos, para su mención, en cantidades más grandes.

8 bits = 1 byte

Debido a que las necesidades informáticas son cada vez más grandes se usan agrupaciones más y más grandes, por ejemplo:

1 bit (b) = 1 cero o 1 uno
1 Byte (B) = 8 bits (b)
1 Kilo Byte (Kb) = 1024 bytes
1 Mega Byte (Mb) = 1024 Kb
1 Giga Byte (Gb) = 1024 Mb
1 Tera Byte (Tb) = 1024 Gb
1 Peta Byte (Pb) = 1024 Tb
etc..

Otra cosa que interesa a la hora de hablar de computadoras es la **velocidad del procesador**. Todos somos impacientes y queremos que nuestra computadora

funcione lo más rápidamente posible.

Existen diferentes factores que determinan con qué rapidez la computadora ejecuta los trabajos. La Velocidad del Procesador es un factor. Pero ¿qué es lo que determina la velocidad del procesador?

La Velocidad del Procesador es afectada por:

Reloj Reloj del Sistema = Un pulso electrónico que es usado para sincronizar el procesamiento. (Entre pulso y pulso solamente puede tener lugar una sola acción).

Medido en megahertz (MHz) donde 1 MHz = 1 millón de ciclos por segundo o gigahertz (GHz) donde 1 GHz = 1 ciclo de mil millones por segundo.

De esto es lo que están hablando cuando dicen que una computadora es una máquina de 2.8 Ghz. La velocidad de su reloj es de 2.8 mil millones de ciclos por segundo.

Cuanto más grande el número = más rápido el procesamiento

Ancho del Bus = Es la cantidad de datos que la CPU puede transmitir en cada momento hacia la memoria principal y a los dispositivos de entradas y salidas. (todo camino para conducir bits es un bus).

Un bus de 8 bits mueve en cada instante 8 bits de datos.

El ancho del Bus puede ser de 8, 16, 32, 64, o 128 bits, hasta ahora.

Piense en ello como "cuántos pasajeros (bits) puede haber en determinado momento, dentro del autobús a fin de trasladarse de una parte de la computadora a otra."

Cuanto más grande sea el número = más rápida será la transferencia de datos.

Tamaño de la palabra = Una palabra es la cantidad de datos que la CPU puede procesar en un ciclo de reloj.

Un procesador de 8 bits puede manejar 8 bits cada vez. Los procesadores pueden

ser, hasta ahora, de 8, 16, 32, o 64 bits.

Cuanto más grande sea el número = más rápida será el procesamiento

Es necesaria la plena coincidencia entre el tamaño de la palabra, tamaño del bus y el reloj. No serviría de nada tener un bus que entregue 128 bits cada vez, si la CPU solo puede, utilizar 8 bits y tiene un reloj de baja velocidad. Se formaría una enorme fila de datos esperando poder salir del bus! Cuando las computadoras se saturan como en ese caso, pueden suceder cosas indeseables con sus datos.

Es como si los espectadores esperaran en una larga cola para entrar al cine; algunos podrían irse!!

Componentes Básicos de las Computadoras

Microprocesador

Es el elemento más importante dentro del computador, normalmente las personas conocen al equipo por el nombre del procesador y se define: la velocidad del proceso, calidad, calentamiento y rendimiento en general.

Tarjeta Principal

Es el lugar donde se coloca el microprocesador, generalmente es la tarjeta más grande dentro del equipo, determina la calidad, estabilidad del sistema y compatibilidad del hardware.

Tarjeta de video

Debido al desarrollo de juegos y programas cada vez más complejos las necesidades gráficas crecen en forma abrumadora, tal es la necesidad que los fabricantes se ven obligados a desarrollar Tarjetas y Ranuras para videos especiales para dar más velocidad a los gráficos afectando en especial a la estructura de la tarjeta principal.

Memoria RAM

La memoria principal o RAM (Random Access Memory, Memoria de Acceso Aleatorio) es una memoria temporal, donde se guardan los datos que se están utilizando en el momento presente (todo lo que se ve en pantalla), mientras más memoria mejor. La diferencia entre la RAM y otros tipos de memoria de almacenamiento, como los disquetes o los discos duros, es que la RAM es mucho más rápida y que se borra al apagar el equipo.

Disco Duro

Sirven como unidad de almacenamiento masivo, es donde se guarda toda la información, incluyendo el sistema operativo y no se borra al apagar el equipo.

Torre

Llamado también *Case* o *Caja* o *Gabinete*. Antiguamente el case no era importante, ahora no sólo sirve para que el equipo se vea bien sino que se convirtió en un elemento básico para el buen funcionamiento de la computadora.

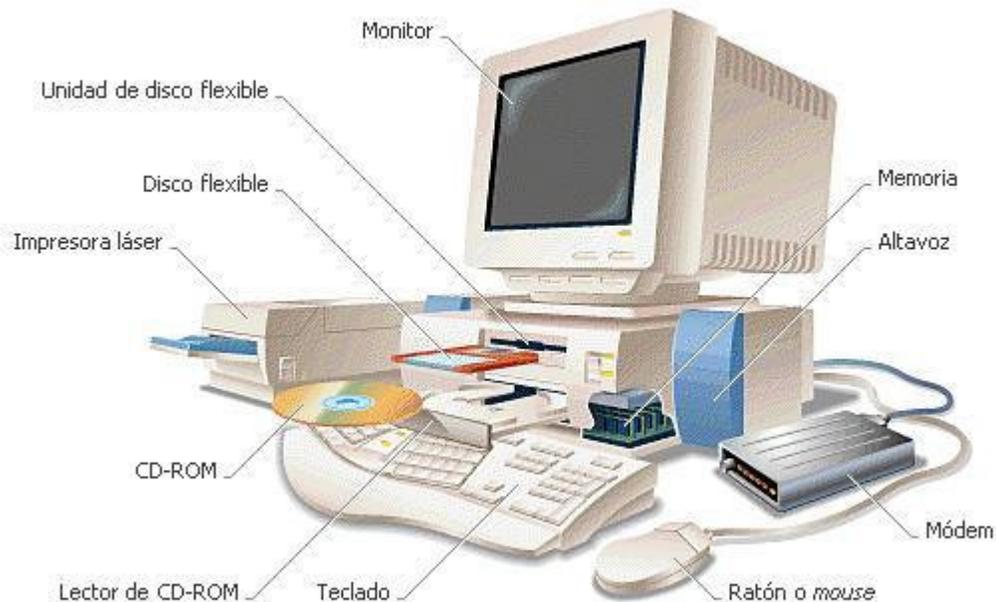
Dispositivos

Pueden o no necesitarse, dependiendo del uso, estos son: Floppy Drive (disquetera), CD-ROM (lector), CD-RW (quemador), DVD (lector de DVD), DVD-RW (quemador DVD), ZIP Drive, LS 120, lectores de tarjetas múltiples (SD, MS, PCMCIA, etc.)

Periféricos

Suelen ser los dispositivos que se conectan externamente al equipo como: Monitor, Teclado, Mouse, Impresora, Scanner, Joystick / gamepad, cámaras, etc.

A continuación una computadora y sus partes explicadas:



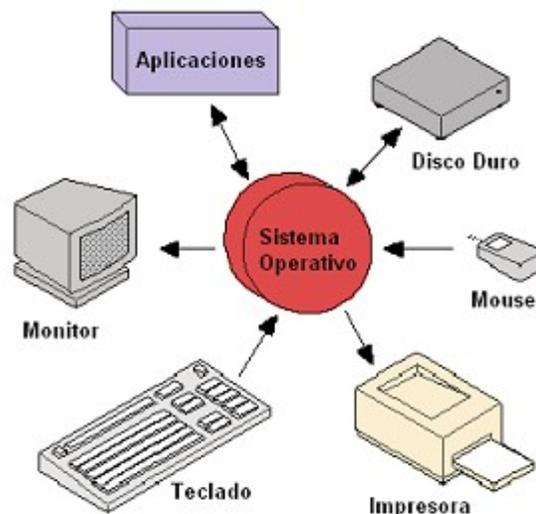
El Software

Las operaciones que debe realizar el hardware son especificadas por una lista de instrucciones, llamadas programas o software. El software se divide en dos grandes grupos: *software del sistema* y *software de aplicaciones*.

El **software del sistema** es el conjunto de programas indispensables para que la máquina funcione. Estos programas son, básicamente, el sistema operativo, los editores de texto, los compiladores/intérpretes (lenguajes de programación) y los programas de utilidad.



Uno de los programas más importantes es el **sistema operativo**, que sirve, esencialmente para facilitar la existencia, ejecución y producción programas. El sistema operativo dirige las operaciones globales de la computadora, instruye a la computadora para ejecutar otros programas y controla el almacenamiento y recuperación de archivos (programas y datos) de dispositivos de almacenamiento.



Un programa es un conjunto de instrucciones que controlan (dirigen) a una computadora; más formalmente, un programa de computadora es un conjunto de introducciones internas utilizadas para ejecutarse en una computadora y que produzca un resultado concreto. Otro término para un programa o conjunto de programas es **software**; ambos términos se utilizan indistintamente. El proceso de escribir un programa, o **software**, se denomina **programación** y el conjunto de instrucciones que se pueden utilizar para construir un programa se denomina **lenguaje de programación**. Así pues, los lenguajes utilizados para escribir programas de computadoras son los lenguajes de programación y programadores son los escritores y diseñadores de programas.

Los Lenguajes de Programación

Los lenguajes de programación sirven para escribir programas que permiten la comunicación persona/máquina.

El propósito de un lenguaje de programación es permitir a las personas comunicarse con una computadora. Los lenguajes de las personas y los lenguajes de la máquina son muy diferentes, ya que las características y posibilidades de las personas y de las máquinas son muy diferentes. Los lenguajes de programación permiten a las personas escribir en un lenguaje que sea más apropiado a las características humanas y se puedan traducir al lenguaje máquina de diferentes tipos de máquinas.

Los principales tipos de lenguajes utilizados:

- lenguajes máquina
 - instrucciones binarias (ceros y unos)
 - sólo la máquina puede entenderlo
- lenguajes de bajo nivel (assembler)
 - instrucciones nemotécnicas a las binarias
 - pueden entenderlo las personas
- lenguajes de alto nivel
 - instrucciones en lenguaje fácil de leer por las personas
 - es fácil escribir y entender para las personas

Los lenguajes de programación de alto nivel existentes hoy en día¹ son muy numerosos, aunque la práctica demuestra que los más usados se reducen a:

- C	- PYTHON	- JAVASCRIPT
- C#	- C++	- RUBY
- JAVA	- PHP	

Unos programas especiales llamados traductores (compiladores o intérpretes) convierten las instrucciones escritas en un determinado lenguaje de programación en instrucciones escritas en el lenguaje que la máquina se sabe y entiende muy bien (0 y 1, bits).

El propósito de un lenguaje informático (lenguaje de computadora) es permitir a las personas comunicarse con una computadora. Los lenguajes de los humanos y los lenguajes de la máquina son muy diferentes, ya que las características y posibilidades de las personas y de las máquinas son muy diferentes.

Traductores de lenguaje

Los traductores de lenguaje son programas que traducen los programas fuente escritos en un lenguaje entendible por personas a un lenguaje que la máquina entienda.

Los traductores se dividen en:

- **Intérpretes:** que son traductores que toman un programa fuente, lo traducen y a continuación lo ejecutan.
- **Compiladores:** generan un programa equivalente que la máquina será capaz de interpretar.

1 Diciembre de 2007

La programación

Después que se ha diseñado y pensado como resolver el problema en papel, se debe comenzar el proceso de introducir el programa en un archivo en el disco duro de la computadora. La introducción y modificación de su programa en un archivo se hace utilizando un simple editor de texto o un editor especializado que hace la tarea de escribir un programa una tarea muy fácil.

El programa que se introduzca en el editor, estará escrito en Java o C++ o cualquier otro, pero ni C++ ni Java son lenguajes máquina, muy al contrario, como ya se mencionó, son lenguajes de alto nivel diseñados para hacer más fácil la programación que utilizando el lenguaje máquina. Pero a la vez, una computadora no entiende los lenguajes de alto nivel. En consecuencia, un programa escrito en un lenguaje de alto nivel debe ser traducido a un lenguaje que la máquina pueda comprender. Los lenguajes que la computadora puede comprender (más o menos directamente) son los lenguajes de bajo nivel. La traducción de un programa escrito en un lenguaje de alto nivel, como C++ o Java, a un lenguaje que pueda entender la computadora se hace mediante otro programa conocido como **compilador**.

Los lenguajes de bajo nivel que la computadora puede entender directamente se conocen como lenguajes ensamblador o lenguajes máquina. En realidad, aunque son muy similares y en ocasiones se les considera sinónimos, tienen algunas pequeñas diferencias. El lenguaje que la computadora puede comprender directamente es el lenguaje máquina. El lenguaje ensamblador es casi la misma cosa, pero necesita un paso adicional para que la traducción pueda ser entendida por la máquina. Si un compilador traduce su programa de alto nivel a algún lenguaje de bajo nivel, no es exactamente lenguaje de máquina, se necesita una pequeña traducción adicional antes de ser ejecutado, pero normalmente este proceso suele ser automático y no es problema.

Hacer un programa en un lenguaje de programación es escribir el programa, compilarlo y luego ejecutarlo.



Historia de los lenguajes de programación

La historia de la informática y de la computación se ha caracterizado fundamentalmente en la existencia de centenares de lenguajes de programación, existen muchos realmente, de ahí que se suele hablar de "La Babel de los lenguajes de programación", aunque de estos centenares de lenguajes, son pocos los lenguajes que han pisado fuerte, que han tenido un impacto significativo.

Muchos de los lenguajes de programación actuales tienen sus raíces en los lenguajes que nacieron a finales de los cincuenta y primeros años de los sesenta, tales como: COBOL, FORTRAN, BASIC, LOGO. Estos lenguajes representaron la primera alternativa a los lenguajes ensambladores. En la década de los setenta y primeros años de los ochenta emergieron nuevos lenguajes como: ALGOL, BASIC, SIMULA, SMALLTAK. A mediados de los ochenta y principios de los noventa se presentaron C, C++, JAVA. Y uno de los últimos lenguajes que se vino en el año 2000 es C#.

Paradigmas de Programación

La programación como cualquier otro arte, por ejemplo Las Artes Marciales, posee diferentes estilos, técnicas o modos de pensar diferentes para programar:

- **Programación Estructurada, Procedural o Imperativa**, en el que los programas se dividen en procedimientos independientes con acceso total a datos comunes. Ejemplos de estos tipos de programas son: C, FORTRAN, BASIC.

Algoritmos + Estructuras de Datos = Programas

- **Programación Funcional**, en el que el resultado de un cálculo es la entrada del siguiente, así hasta que se produce el valor deseado. HASKELL es un ejemplo moderno.

Funciones + Estructuras de Datos = Programas

- **Programación Orientada a Objetos**, en el que los datos se consideran la parte más importante del programa, de modo que se agrupan en objetos. Los objetos modelan las características de los problemas del mundo real, su comportamiento ante estas características y su forma de interactuar con otros elementos. SMALLTALK, JAVA son algunos de los lenguajes que soportan este paradigma.

Objetos + Mensajes = Programas

El lenguaje de Programación Java

Java surgió en 1991 cuando un grupo de ingenieros de *Sun Microsystems* trataron de diseñar un nuevo lenguaje de programación destinado a electrodomésticos.

La historia va más o menos así: En 1991, Sun Microsystems financió un proyecto de investigación corporativo interno. Un resultado del proyecto fue la creación de un lenguaje basado en C y C++ que su creador, James Gosling, llamó Oak, inspirado por un roble (oak, en inglés) que crecía afuera de su ventana en Sun. Más tarde se descubrió que ya existía un lenguaje de programación llamado Oak. Un día cuando un grupo de empleados de Sun visitó una cafetería cerca, alguien sugirió el nombre Java (como se conoce también en inglés al café), y el nombre gusto mucho. Sun anunció formalmente a Java en una conferencia importante en mayo de 1995.

Trece años después de su lanzamiento oficial, Java se ha convertido en un estándar de la industria, en un lenguaje de programación para desarrollo de aplicaciones tanto de propósito general como de Internet y también en un lenguaje para comenzar la formación en programación, al tener características adecuadas para el aprendizaje.

Otras cosas más sobre java:

 <i>James Gosling</i>	 <i>Taza de café</i>	 <i>Duke</i>
Creador de Java	Logo de Java	Mascota de Java

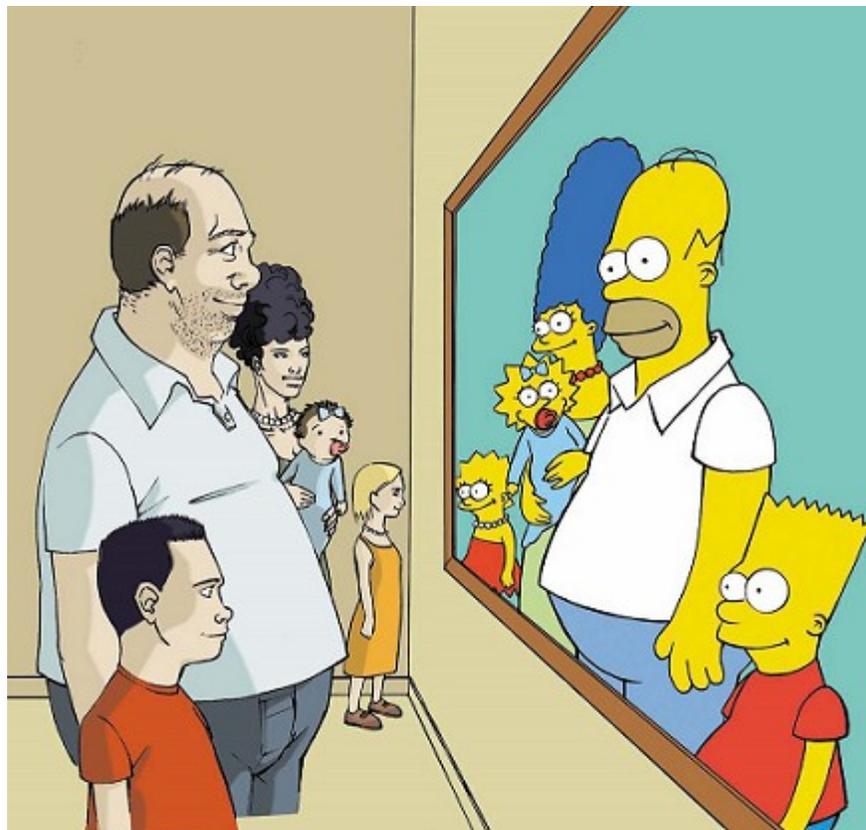
Práctica # 1 - Introducción a la Informática

1. Define los siguientes términos:
 - a) programa de computadora
 - b) programación
 - c) lenguaje de programación
 - d) lenguaje de alto nivel
 - e) lenguaje máquina
 - f) lenguaje ensamblador
 - g) compilador
 - h) intérprete
2. Averigua y enumera los diferentes tipos de lenguajes de programación utilizados a lo largo de las diferentes generaciones de computadoras.
3. Averigua y describe las características más sobresalientes de Java.
4. ¿Crees que influye el conocimiento del hardware en la formación del aprendizaje de la programación?
5. Haz una lista de sistemas operativos que conozcas.

Programación Orientada a Objetos

Introducción

La programación orientada a objetos (POO), es una de las más grandes ideas en el campo de la programación durante los años recientes. Todo se reduce a organizar los programas en formas que van a plasmar el mundo real para representarlo en un mundo de computadora.



La clave en ésta forma de programar son los objetos que nos rodean. Estos objetos pueden ser cosas físicas como personas, casas, sillas, etc. o cosas lógicas como el clima, la hora, la fecha, etc.

La POO nos permitirá resolver problemas de la vida real usando objetos de la vida real.

Pensar en objetos: una analogía

Vamos a hablar de los rastis. Los rastis o legos, si es que no pasas mucho tiempo con niños, son pequeños bloques de construcción de plástico en varios colores y tamaños. Éstos tienen pequeñas partes circulares en un lado, que encaja en pequeños orificios circulares en otros rastis para unirse, de manera ajustada, a fin de crear formas más grandes. Con diferentes partes de rastis (ruedas, máquinas, bisagras y poleas), los niños pueden construir castillos, automóviles, robots o cualquier otra figura que puedas imaginarte.

Aquí está otro ejemplo: puedes entrar a una tienda donde venden partes de computadoras y con un poco de conocimiento básico (y con frecuencia algo de ayuda de tu cuate), ensamblar una PC completa con varios componentes: una tarjeta madre, un chip CPU, una tarjeta de video, un disco duro, un teclado, etc. Lo ideal sería que, cuando termines de ensamblar todas las diferentes unidades individuales, tengas un sistema en donde todas las unidades crean un sistema más grande, con el cual puedes solucionar los problemas por los que compraste en un principio la computadora.

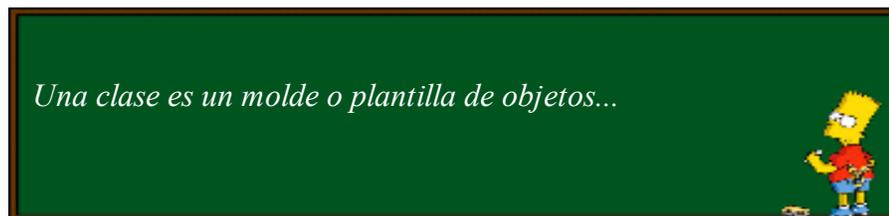
En su estructura interna (su implementación), cada uno de estos componentes puede ser muy complicado y ser de distintos fabricantes que utilizan distintos métodos de diseño. Pero no necesitamos saber cómo funcionan los componentes, qué hace cada chip en la tarjeta, o cómo, cuando presiona la tecla A, se envía una "A" a la computadora. Cada componente que se utiliza es una unidad individual, y todo lo que interesa es cómo interactúan las unidades entre sí. ¿Encajará esta tarjeta de video en las ranuras de la tarjeta madre y funcionará este monitor con esta tarjeta de video? ¿Se comunicará cada componente en particular con los comandos correctos a los demás con quienes interactúan, para que cada parte de la computadora se entienda con todas las demás? Una vez que conocemos las interacciones que hay entre los componentes y la forma en que coinciden con aquellas, es sencillo armar una computadora.

¿Qué tiene que ver esto con la programación? Todo. La POO funciona de esta manera. Al utilizarla, el programa general está formado por componentes individuales (objetos) numerosos y diferentes; cada uno de los cuales realizará su papel en el programa y todos se comunicarán en formas predefinidas.

Objetos y clases

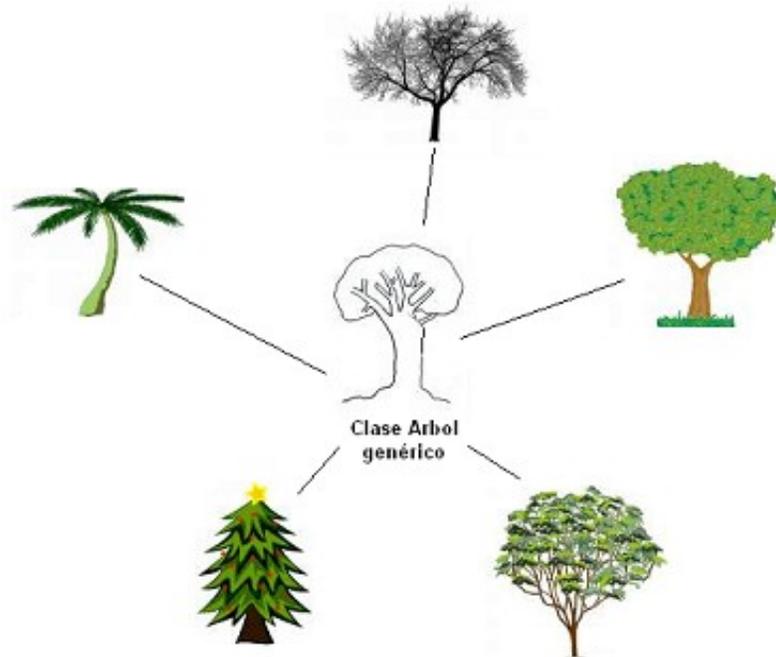
La programación orientada a objetos está modelada sobre cómo, al igual que en el mundo real, los objetos con frecuencia están formados por varios tipos de objetos más pequeños. Esta capacidad de combinar objetos, sin embargo, sólo es un aspecto muy general de la programación orientada a objetos. Ésta proporciona otros conceptos y características para hacer más fácil y más flexible la creación y utilización de objetos, donde la más importante de estas características son las clases.

Una clase es una plantilla para múltiples objetos con características similares. Las clases comprenden todas las características de una serie particular de objetos.



Vamos a ver cual es la diferencia básica entre clases y objetos. Por ejemplo, si nos ponemos a hablar de palmeras, eucaliptos, pinos... de qué estamos hablando realmente? Pues de árboles. Entonces podríamos contar con una clase Arbol que describa las características de todos los árboles (tienen hojas y raíces, crecen, crean clorofila, etc). La clase Arbol sirve como un modelo abstracto para el concepto de un árbol; para alcanzar y sujetar, o interactuar con él, o cortarlo tendrá que poseer una instancia concreta del mismo. Por supuesto, en cuanto tenga una clase árbol, puede crear muchas instancias del mismo, y cada diferente instancia puede tener diversas características (bajo, alto, frondoso, cambia de hojas en el otoño), mientras todavía se compone y se reconozca como árbol.





Así que, ¿cuál es la diferencia exacta entre una instancia y un objeto? En realidad ninguna. Objeto es el término más general, pero tanto las instancias como los objetos son la representación concreta de una clase. De hecho, los términos instancia y objeto con frecuencia se utilizan en forma indistinta en el lenguaje POO. Una instancia de un árbol y un objeto árbol son lo mismo.

Práctica # 2 - Clases y Objetos

1. Completa la siguiente tabla colocando el tipo de objetos o ejemplos de objetos (mínimamente 3) y el nombre de clase según la nomenclatura Java (Nombre propio, empezando con mayúscula y singular). Después para cada clase, realizar un diagrama de clase indicando mínimamente 2 atributos.

<i>Tipo de Objetos</i>	<i>Clase Java</i>	<i>Objetos</i>
Puntos	Punto	(5,2) - (1,-3) - (0,1)
Países		
		Java, C#, C++
Docentes		
		10:30 - 15:45 - 8:00
Planetas		
		14/09/2007, 25/12/2006, 14/08/2003
Sistemas Operativos		
		$13X^2$, $-7X^3$, $3XY^5$
Celulares		
		5 Bs, 10 \$us, 1 Bs
Universidades		
		Hitachi de 80 Gb Maxtor de 120 Gb Samsung de 160 Gb
		Algebra I, Calculo II, Física III
		14/7, 1/2, 3/4

Antes de continuar veamos un concepto que es muy importante en programación y sobre todo en programación orientada a objetos.

La abstracción

La abstracción visto desde el punto de vista de la programación orientada a objetos expresa las características esenciales de un objeto dentro un contexto en particular, las cuales distinguen al objeto de los demás. Además de distinguir entre los objetos provee límites conceptuales. Entonces se puede decir que la encapsulación separa las características esenciales de las no esenciales dentro de un objeto. Si un objeto tiene mas características de las necesarias los mismos resultarán difíciles de usar, modificar, construir y comprender.

La misma genera una ilusión de simplicidad dado a que minimiza la cantidad de características que definen a un objeto.



El contexto en el que se analiza al objeto es importante, ya que de ello dependerá la abstracción que realicemos. Por ejemplo, la importancia que tiene la cabeza para un psicólogo es muy diferente a la importancia que tiene la cabeza para un peluquero.



Vs.



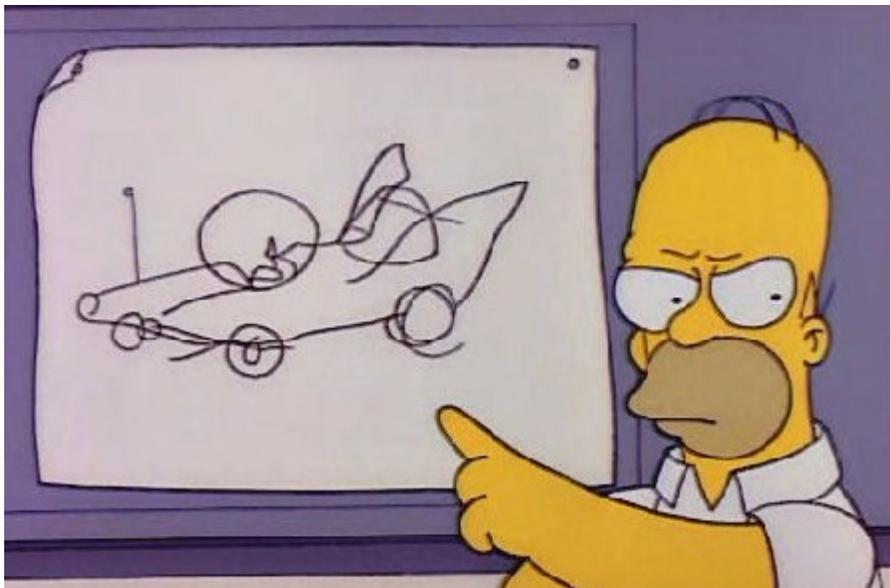
EJERCICIO.- Indica la forma de llegar a tu casa. Para éste ejercicio puedes utilizar cualquier forma de representación.

Durante años, los programadores se han dedicado a construir aplicaciones muy parecidas que resolvían una y otra vez los mismos problemas. Para conseguir que los esfuerzos de los programadores puedan ser utilizados por otras personas se creó la POO, que es una forma de programar que enfatiza descubrir los objetos del mundo real en un contexto particular, simulándolos en objetos computacionales.

La POO no es difícil, pero es una manera especial de pensar, a veces subjetiva de quien la programa, de manera que la forma de hacer las cosas puede ser diferente según el programador. Aunque podamos hacer los programas de formas distintas, no todas ellas son correctas, lo difícil no es programar orientado a objetos sino programar bien. Programar bien es importante porque así podemos aprovechar todas las ventajas de la POO.

¿Cómo se piensa en objetos?

Pensar en términos de objetos es muy parecido a cómo lo haríamos en la vida real. Por ejemplo vamos a pensar en un coche para tratar de modelarlo en un esquema de POO. Diríamos que el coche es el elemento principal que tiene una serie de características (atributos), como podrían ser el color, el modelo o la marca. Además tiene una serie de funcionalidades asociadas (métodos), como pueden ser ponerse en marcha, parar o estacionarse.



Pues en un esquema POO el coche sería el nombre general (clase) que utilizamos para referirnos a un conjunto de objetos con las mismas características y las mismas funcionalidades; los atributos serían las características como el color o el modelo y los métodos serían las funcionalidades asociadas como ponerse en marcha o parar.

Por poner otro ejemplo vamos a ver cómo modelaríamos en un esquema POO a las fracciones ($\frac{3}{4}$, $\frac{1}{2}$, etc.). un objeto fracción, es esa estructura matemática que tiene un numerador y un denominador que divide al numerador, por ejemplo $3/2$.

Fracción será el nombre que le demos a esta clase de objetos y tendrá dos atributos, el numerador y el denominador. Luego podría tener varios métodos como simplificarse, sumarse, restarse, multiplicarse, dividirse con otra fracción o número, etc.

Estos objetos se podrán utilizar en los programas, por ejemplo en un programa de matemáticas harás uso de objetos fracción y en un programa que gestione un taller de coches utilizarás objetos coche.

Atributos y Métodos

Ahora bien, cada clase está, por lo general, compuesta de dos componentes:

- **atributos**, indican características comunes de los objetos
- **métodos**, indican comportamiento similar de los objetos

Atributos

Los atributos son las características individuales que diferencian a un objeto de otro, y determinan la apariencia, estado u otras cualidades de ese objeto.

Por ejemplo, suponiendo que queremos modelar a las personas, creamos una clase llamada `Persona`, y para ver cuales serían sus atributos pensemos en una persona, un individuo en particular y seguro muy conocido para ti: Homero Simpson... Vamos a decir que Homero está viejo, seguramente te imaginarás a Homero muy adulto, calvo, de unos 50 años. Es decir, apenas yo digo Homero está viejo, y ya entras en una serie de razonamientos lógicos y concluyentes para captar claramente lo que yo quise decir. ¿Qué fue lo que describí de Homero? Muy sencillo, describí un atributo.

A veces los atributos pueden llegar a describirse de una forma muy relativa. Ya que si Homero tuviera unos 30 años no sería tan viejo para personas de la misma edad, para personas más jóvenes si sería algo viejo y para los niños si sería muy viejo. Cuando se describe de esa forma un atributo depende mucho de quien da el concepto. Igual como cuando un hombre dice que una determinada mujer es hermosa, pues la que para él es hermosa puede que para otros no lo sea tanto. Por esta razón es mejor expresar los atributos de manera que sean tasados a través de una escala ya que estos los hace manejables y no relativos. Esto dará lugar a la presencia de un concepto muy importante en Informática: El **Dato**. En lugar de decir Homero está viejo podríamos decir Homero tiene 40 años. En este caso, a pesar de que los razonamientos y conclusiones sean las mismas, podemos dejar al libre concepto de quien sea.

El dato

El dato es un atributo codificado en términos entendibles a un sistema computarizado, en condiciones que sean manejables y comparables de manera casi absoluta.

Sin embargo un dato, solo, no significa nada a menos que se tenga claridad de cuál es el atributo que él está describiendo. Por ejemplo, si yo dijera que el dato es Z. ¿Qué pensarías que significa este dato? La cantidad de computadoras que tengo o la cantidad de novias que tengo o la cantidad de autos que tengo o la edad que tengo o ... realmente no tendrías idea del significado de dicho dato. Para ello, vendrá sobre nosotros un concepto que aclarará las cosas: **la variable**.

La Variable

Una variable se representa con un nombre significativo al atributo que representa, un identificador que hace referencia a un lugar de la memoria principal y almacena un dato.

Pero es necesario especificar que tipos de dato se puede almacenar en una variable, para eso existen los tipos de datos que se encargan de restringir el dato que puede almacenarse en una variable.

Tipos de Datos en Java

En Java se definen 2 tipos de datos:

- tipos primitivos
- tipos referencia

Los **tipos primitivos** son 8 agrupados en cuatro categorías:

- ✓ lógico: boolean, almacena valores o datos lógicos como true o false.
- ✓ texto: char, almacena valores o datos caracteres como 'a'
- ✓ entero: byte, short, int, long, almacena valores numéricos enteros.
- ✓ real: float, double, almacena valores numéricos reales.

Tabla de tipos de datos primitivos

Tipo	Contiene	Defecto	Tamaño	Rango de Valores	
				Min	Max
boolean	True o false	False	8 bits	-	-
char	Caracter Unicode	\u0000	16 bits	\u0000	\uFFFF
byte	Entero con signo	0	8 bits	-128	127
short	Entero con signo	0	16 bits	-32768	32768
int	Entero con signo	0	32 bits	-2147483648	2147483647
long	Entero con signo	0	64 bits	-9223372036854775808	9223372036854775808
float	IEEE 754 estándar punto flotante	0.0	32 bits	±3.40282347E+38	±1.40239846E-45
double	IEEE 754 estándar punto flotante	0.0	64 bits	±1.79769313486231570E+308	±4.94065645841246544E-324

Los **tipos referencia** son sólo referencias a otros objetos. En realidad lo que hacemos cuando creamos una clase para modelar objetos, es crear un nuevo tipo de dato objeto, más conocido como tipos referencia.

Algunos tipos referencia de ejemplo:

Tipo	Referencia	Tamaño
String	Literales o cadenas de texto	32 bits
Integer	Un entero con signo	32 bits
Long	Un entero grande con signo	32 bits
Short	Un entero corto con signo	32 bits
Double	Un número real	32 bits
Float	Un número real	32 bits
Character	Un caracter Unicode	32 bits
Persona	Un objeto persona	32 bits
Coche	Un objeto coche	32 bits
Fraccion	Un objeto fraccion	32 bits

Volviendo al ejemplo, nuestra frase inicial Homero está viejo, que luego se convirtió en Homero tiene 40 años, podríamos ahora enunciarla como La edad de Homero es 40. En este instante, a pesar de que podemos obtener las mismas conclusiones que en las frases pasadas, ya tenemos identificado de manera clara y

con un nombre el atributo que se está describiendo. Éste es el concepto de **Variable o Campo** que no es más que el nombre que se le coloca a un dato para identificar el atributo que está describiendo. Además podemos especificar el tipo de dato que tendría esta variable que sería int puesto que la edad es un valor entero.

Así en nuestra frase La edad de Homero es 40, tenemos tres atributos claramente identificados (tres...???) Sí, tres atributos, tres campos, tres variables: el primero de ellos es la edad, atributo con el cual hemos estado realizando toda la explicación, el segundo es el nombre de la persona de quien estamos hablando y ya sabemos que es Homero y el tercero es el sexo pues podemos asegurar que Homero es de sexo masculino.

De tal forma que si organizáramos la información de la que hemos hablado hasta el momento podríamos hacer un pequeño esquema como el que se presenta a continuación:

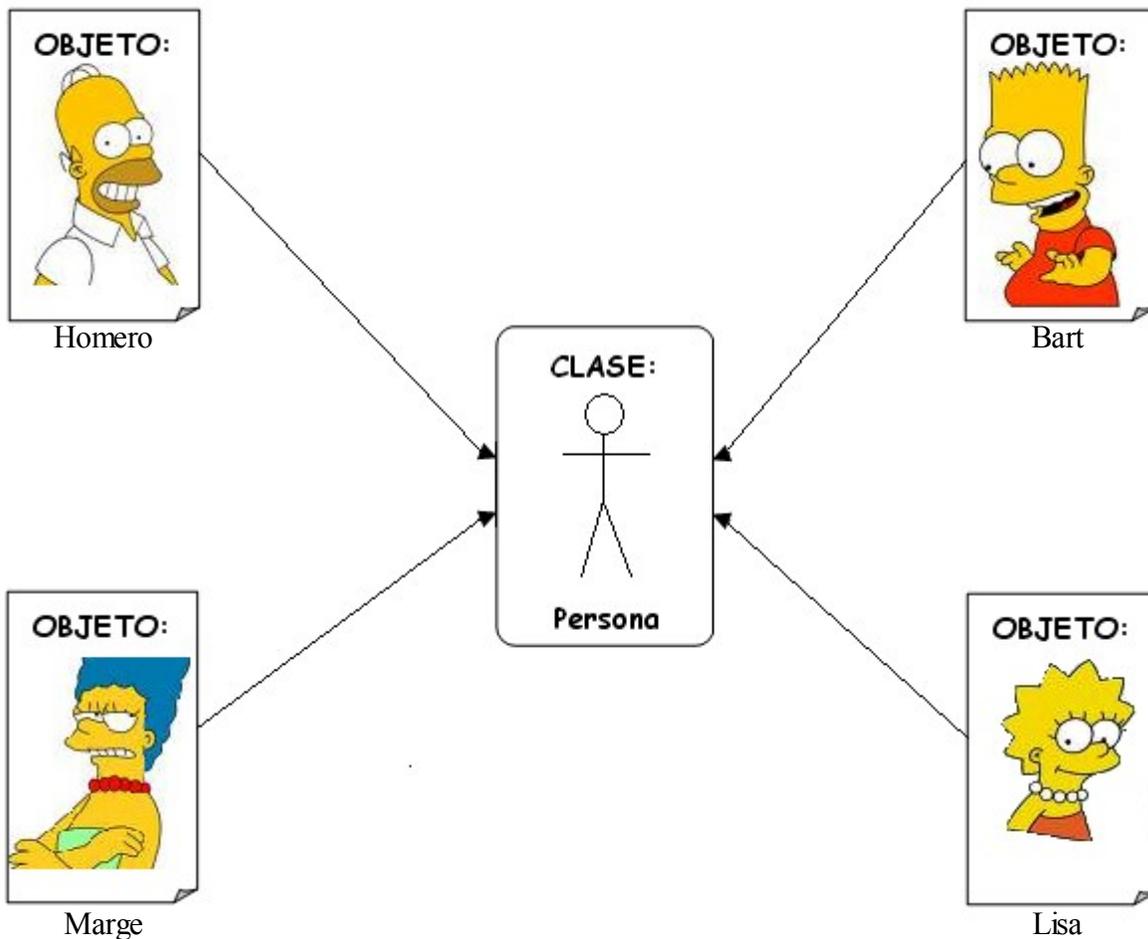
Los atributos de la persona:

- nombre de tipo Literal (String)
- edad de tipo entero o (Int)
- sexo de tipo Literal (String)

Nombre	Edad	Sexo
Homero	40 años.	Masculino

Incluso podríamos tener información de otros objetos del mismo tipo:

Nombre	Edad	Sexo
Homero	40 años	Masculino
Bart	10 años	Masculino
Lisa	8 años	Femenino
Marge	35 años	Femenino



Métodos

Los métodos reflejan el comportamiento de los objetos. El comportamiento de clase determina qué instancias de esa clase requieren cambiar su estado interno o cuándo esa instancia es llamada para realizar algo por otra clase u objeto. El comportamiento es la única manera en que los objetos pueden hacerse algo a sí mismos o tener que hacerles algo.

Algunos comportamientos para nuestra clase de ejemplo:

- **saludar** como *hola soy Homero*
- **dameTuEdad** como *40* o *15*

Los métodos son la interfaz de comunicación del objeto con el mundo exterior, permiten mandar mensajes a los objetos para que realicen cosas.

Por ejemplo: Si alguna vez jugaste Mario Bross o un video juego parecido, recordarás que para hacer que Mario salte o bote fuego había que apretar algunos botones del mando, pues así es la utilización de los métodos de un objeto, en este caso para hacer que el objeto Mario salte, pues le mandamos un mensaje desde el mando para esto, aprentando un botón, si quisieramos que bote fuego, pues apretamos otro botón.

Los métodos no siempre afectan a un solo objeto; los objetos también se comunican entre sí mediante el uso de métodos. Un objeto puede llamar a métodos en otro objeto para avisar sobre los cambios en el ambiente o para solicitarle a ese objeto que cambie su estado.

Creando nuestra primera clase utilizando BlueJ

Hasta el momento hemos repasado algunos conceptos que son importantes, pero dejemos a un lado la parte teórica y vayámonos a lo práctico. En esta sección aprenderemos a crear nuestra clase de ejemplo llamada Persona, a fin de que se observe como se definen los atributos y métodos. Cuando escribamos nuestra primera clase, estaremos haciendo nuestro primer programa orientado a objetos.

¿Pero... qué dijimos que es un programa?

En el sentido informático se dice que un programa es una secuencia de instrucciones que se escriben y que la computadora puede interpretar y ejecutar¹.

El escribir un programa (editar) normalmente se lo hace en un editor de textos como el Bloc de Notas de Windows. La interpretación que hace la máquina de lo que escribimos no es directa (en algunos casos), sino que se lo traducimos a un lenguaje que pueda comprender. El proceso de traducir es más conocido como compilar y es lo que tenemos que hacer siempre antes de ejecutar nuestros programas Java.

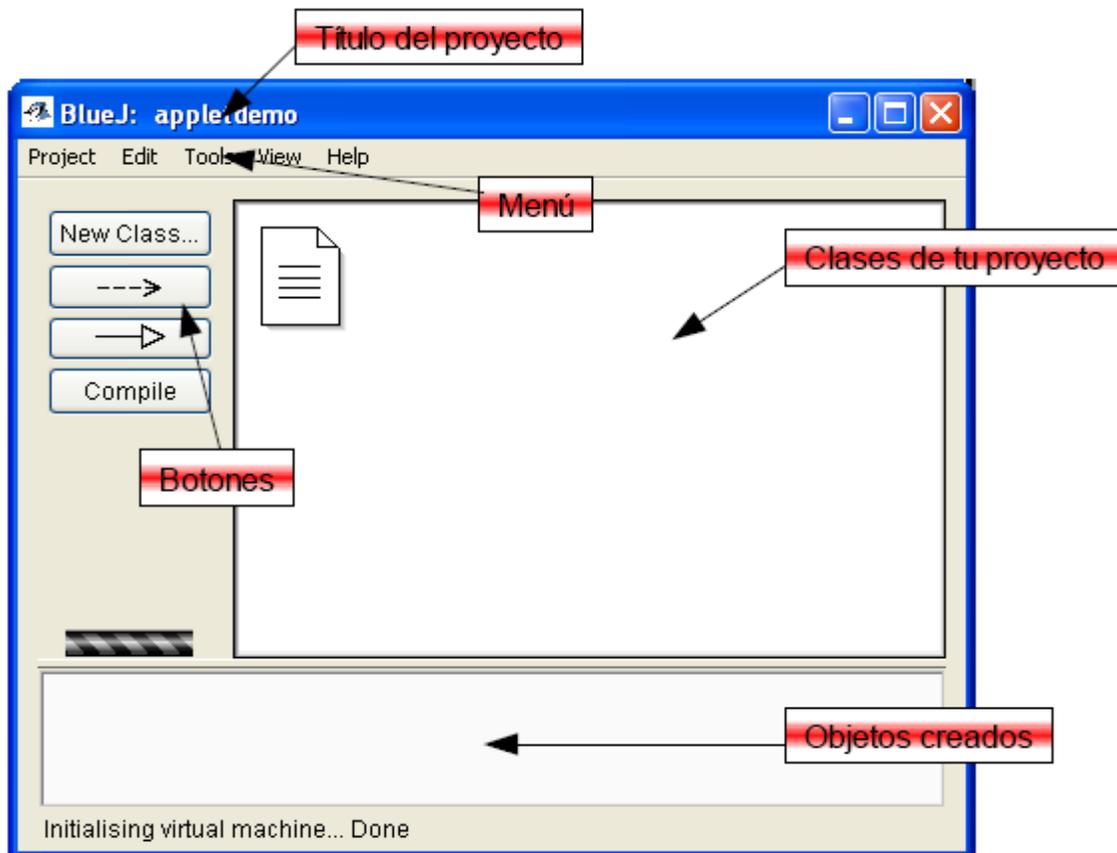
¿Qué es BlueJ?

BlueJ es un IDE (Entorno de Desarrollo Integrado), que facilita mucho el proceso de crear un programa (editar, compilar y ejecutar) y el proceso de entender un programa (ver gráficamente a los objetos, las clases y sus relaciones).



¹ Según la Wikipedia <http://wikipedia.org/>

La siguiente figura muestra y explica la ventana principal de BlueJ y sus principales áreas de trabajo:



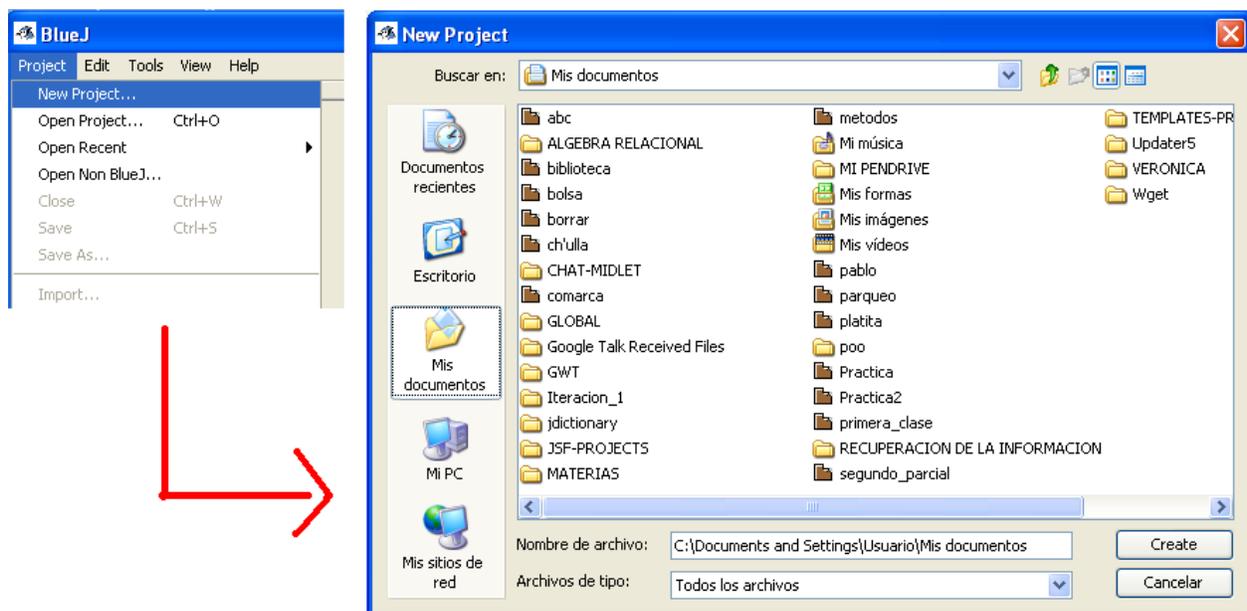
1. Sobre el **título** no es mucho lo que se puede decir, cuando trabajas con BlueJ creas un nuevo proyecto para trabajar, a ese tienes que darle un nombre, el cual se muestra en esta parte.
2. En el **menú** tenemos 5 secciones, Project (proyecto), Edit (edición), Tools (herramientas), View (ver) y Help (ayuda), vamos a describir solo lo que vamos a utilizar más.
 1. En **project** podemos crear, guardar y abrir proyectos, crear ejecutables y también salir del programa principalmente.
 2. En **edit** podemos crear clases, flechas de uso o de herencia (ya vamos a ver que son).
 3. En **Tools** podemos ver las preferencias de BlueJ y compilar principalmente.
 4. **View** sirve para mostrar diferentes cosas que tiene BlueJ como debugger, terminal y code pad.

5. **Help** creo que todos sabrán para que es... si quieres encontrar información sobre BlueJ o clases java.

Ahora sí, empecemos a crear nuestra primera clase con la ayuda del buen BlueJ. ¿Estámos listos? Lo primero será crear un proyecto, crear una clase y crear los objetos.

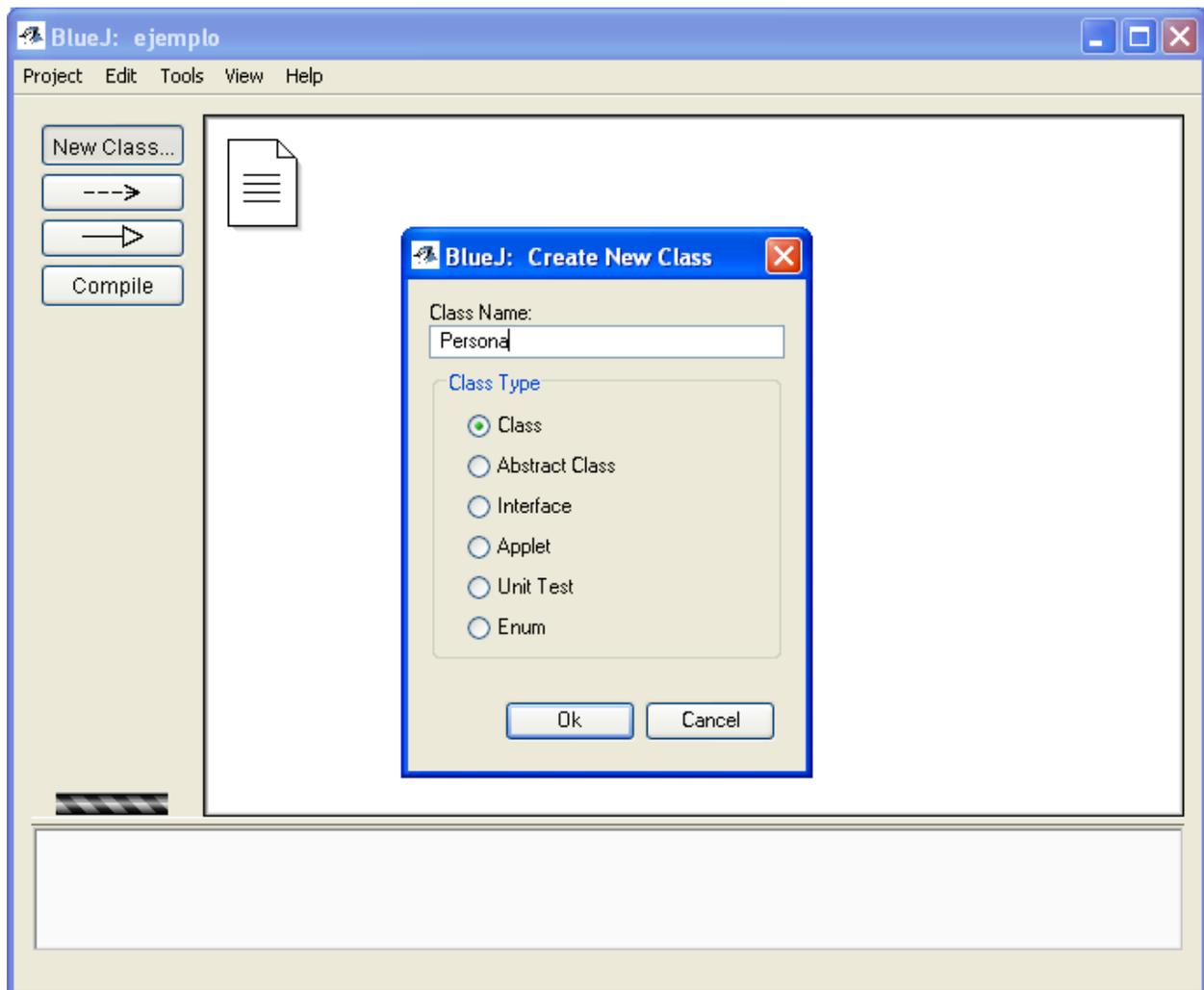
1) Crear un proyecto

Un proyecto para BlueJ es una carpeta donde se colocarán todos los archivos de nuestro programa. Para crear un proyecto nos vamos al menú Project, luego seleccionamos la opción New Project, el mismo nos abrirá un cuadro de diálogo para crear nuestro proyecto donde querramos. El nombre de proyecto puede ser cualquier cosa.



2) Crear una clase

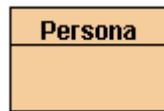
Existen varias formas de crear una clase en BlueJ, la más sencilla es hacer click en el botón New Class y aparecerá un cuadro de diálogo donde debemos colocar el nombre de la clase y elegir que tipo de clase será. Esto creará un archivo con el mismo nombre que la clase y de extensión `.java`. Para nuestros primeros ejercicios podemos obviar la parte de elegir el tipo de clase.



Convenciones sobre el nombre de las clases

- ✓ Debe ser significativo
- ✓ Empezar con mayúscula (como un nombre propio)
- ✓ Estar escrito en singular

Una vez colocado el nombre en el cuadro de diálogo, aparecerá una cajita en el área de las clases, ésta representará a la clase en cuestión.



Debemos de editar el código por defecto que tiene ésta clase, para lo cual haremos doble click sobre la cajita, aparecerá el editor de esta clase mostrando código que tal vez, de momento, no entendamos ni j, así que borraremos todo lo que esté escrito en el editor para hacer nuestro propio código.

Una vez borrado todo, escribimos lo siguiente:

```
class Persona
{
}
```

Y pues bien, hemos creado una clase. Claro que no hace mucho en este momento, pero ésta es una clase en Java en su forma más sencilla.

Ésta es la sintaxis general para definir a una clase.

```
class <NombreClase>
{
    // Declaración de atributos
    <tipo> <variable> ;
    // Declaración de métodos
    <tipo> <nombreMétodo> ( <parámetros> )
    { ... }
}
```

El nombre del archivo Java debe coincidir con el de la clase definida en él
<NombreClase>.java

3) Crear objetos

Una vez hecha la clase ya es posible crear objetos. En BlueJ esto es una tarea fácil. Sólo tenemos que hacer click derecho sobre la clase y elegir la opción new Persona (para nuestro ejemplo).



Haciendo lo mismo pero con código:

```
[...]  
Persona per;  
per = new Persona( );  
[...]
```

Siguiendo con la creación de nuestra clase Persona, ahora veamos algunos atributos (variables de instancia) para esta clase.

¿Qué tiene toda persona?

- nombre
- edad

Una vez identificado los atributos para una clase, es necesario indicar el tipo de dato que guardarán sus objetos.

...Volviendo al ejemplo:

- **nombre**, como Homero, Bart, Lisa, Marge. El tipo de dato más adecuado sería **String**.
- **edad**, como 40, 10, 8, 38. El tipo de dato más adecuado sería **int**.

Y de esa manera colocamos los atributos en nuestra clase.

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

Ahora nos toca ver algunos métodos.

¿Qué hace toda persona?

- saludar
- decir si es mayor de edad o no
- decir su nombre
- decir su edad
- modificar edad

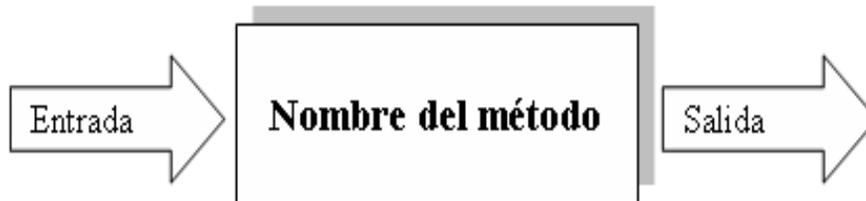
Como los métodos son las cosas que saben hacer los objetos, y para invocar a ese método se lo hace a través de mensajes, veamos que tipo de mensajes existen para ver como debería estar estructurado un método.

Los mensajes pueden ser:

- con respuesta. Ejemplo: dime tu nombre.
- sin respuesta. Ejemplo: levántate. (No esperamos que nos de una respuesta, simplemente realiza una acción)
- con información adicional (parámetros). Ejemplo: ¿Te gusta el color ? (El parámetro sería el color)
- sin información adicional (sin parámetros). Ejemplo: ¿Eres mayor de edad? (No

se necesita ningún parámetro para realizar ese mensaje)

A un método podría vérselo de la siguiente forma (como si fuese una caja):



Donde la entrada, sería la información adicional del mensaje (parámetros) y la salida sería el retorno del mensaje (retorno del método).

La sintaxis general de un método es:

```
[public]<tipo de dato del retorno> <nombre del método> ([parámetros])  
{  
    <instrucciones>  
}
```

Donde:

Tipo de dato del retorno:	Si el método devuelve algo, éste tiene un tipo de dato, que es un tipo de dato válido en Java (primitivo o clase).
Nombre del método:	es el nombre, a través del cual se invocará al método.
Parámetros:	la información adicional al método.
Instrucciones:	aquí indicamos cómo el objeto hace la operación. Si el método tiene respuesta, entonces deberá existir una instrucción: <code>return <expresion>;</code>

Volviendo a nuestro ejemplo, veamos como son los métodos de la clase Persona:

- *saludar*: tiene respuesta (String) y no necesita información adicional.

```
public String saludar()
{
    return "Hola soy "+nombre+" y tengo "+edad+" años ";
}
```

- *decir si es mayor de edad o no*: tiene respuesta (boolean) y no necesita información adicional.

```
public boolean esMayorDeEdad()
{
    return edad >= 18;
}
```

- *decir su nombre*: tiene respuesta (String) y no necesita parámetros.

```
public String decirNombre()
{
    return nombre;
}
```

- *decir su edad*: tiene respuesta (int) y no necesita parámetros.

```
public int decirEdad()
{
    return edad;
}
```

- *modificar su edad*: no tiene respuesta (void) y necesita saber la nueva edad (int)

```
public void modificarEdad(int nuevaEdad)
{
    edad = nuevaEdad;
}
```

El método constructor

- Es un método especial que sirve para inicializar los atributos.
- Tiene el mismo nombre que la clase.
- No tiene tipo de retorno (ni siquiera void)

```
class Persona
{
    String nombre;
    int edad;

    public Persona(String n, int e)
    {
        nombre = n;
        edad = e;
    }
}
```

Tiene el mismo nombre que la clase y no definen tipo de retorno.

- Se invoca automáticamente cuando se crea un objeto.

```
Persona p = new Persona("Pepito",10);
```

La implementación completa de la clase Persona sería:

```
class Persona
{
    String nombre;
    int edad;

    public Persona(String n, int e)
    {
        nombre = n;
        edad = e;
    }

    public String saludar()
    {
        return "Hola me llamo "+nombre+" y tengo "+edad+" años";
    }

    public boolean esMayorDeEdad()
    {
        return edad >= 18;
    }

    public String decirNombre()
    {
        return nombre;
    }

    public int decirEdad()
    {
        return edad;
    }

    public void modificarEdad(int nuevaEdad)
    {
        edad = nuevaEdad;
    }
}
```

Algunas convenciones sobre los nombres de métodos

- se escriben con minúscula. Ej: saludar
- si su nombre involucra varias palabras, la primera letra de cada letra (menos la primera), con mayúscula. Ej: decirNombre
- Para todos aquellos métodos que se encargan de obtener valores de atributos, coloca el prefijo `get` seguido del nombre del atributo. Ej: En nuestro ejemplo, en lugar de escribir `decirNombre`, deberíamos colocar `getNombre`.
- Para todos aquellos métodos que se encargan de modificar valores de los atributos, coloca el prefijo `set` seguido del nombre del atributo. Ej: En nuestro ejemplo, en lugar de escribir `modificarEdad` deberíamos colocar `setEdad`.
- Si el método va a devolver un valor lógico, el nombre del método debería escribirse como si fuese una pregunta al objeto. Ej: `esMayorDeEdad ?`

La implementación siguiendo las convenciones de métodos:

```
class Persona
{
    String nombre;
    int edad;

    public Persona(String n, int e)
    {
        nombre = n;
        edad = e;
    }

    public String saludar()
    {
        return "Hola me llamo "+nombre+" y tengo "+edad+" años";
    }

    public boolean esMayorDeEdad()
    {
        return edad >= 18;
    }

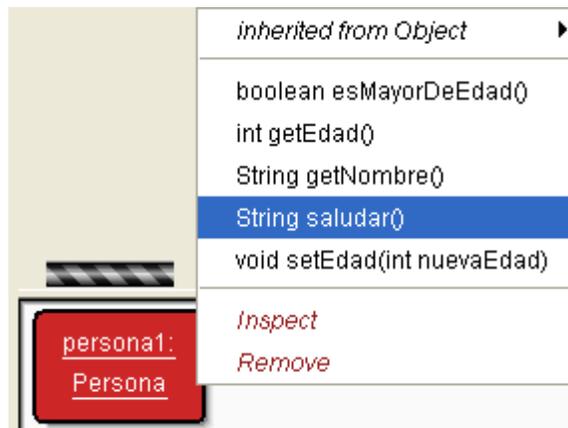
    public String getNombre()
    {
        return nombre;
    }

    public int getEdad()
    {
        return edad;
    }

    public void setEdad(int nuevaEdad)
    {
        edad = nuevaEdad;
    }
}
```

Interactuando con objetos en BlueJ

Para interactuar con los objetos que vayamos creando con BlueJ, sólo tenemos que hacer click derecho sobre el objeto creado y ver en el menú contextual la lista de métodos (mensajes), que le podemos hacer a la instancia.



En código ésto se hace a través del operador **punto (.)**

```
<objeto>.<algunMétodo> ();
```

Ejemplo:

```
String s = per.saludar();  
per.setEdad(13);  
String nom = per.getNombre();
```

Colecciones de Objetos

Una colección de objetos es sencillamente un conjunto de cosas del mismo tipo. Esta idea de agrupar cosas no es nueva para ti. Ya conoces el significado de estas expresiones: "un juego de libros", "una colección de sellos", "una cadena de tiendas", "un conjunto de ejercicios". Algunas veces utilizamos otras palabras tales como rebaño, manada, cuadrilla, equipo, compañía, grupo, club, clase o familia para referirnos a colecciones y conjuntos. Tú mismo eres un elemento de grupo o conjunto tales como éstos: tu familia o tu clase.

En Java, existen colecciones de objetos que también son considerados otros objetos, están modelados por clases o por estructuras primitivas. `ArrayList` es una clase que modela a un grupo especial de colecciones.

La clase `ArrayList`

La clase `ArrayList` almacena un conjunto de objetos del mismo tipo a manera de lista. Java tiene una suite completa de objetos "Collection" que almacenan objetos de varias maneras. `ArrayList` es el tipo de colección más famoso y comúnmente usado.

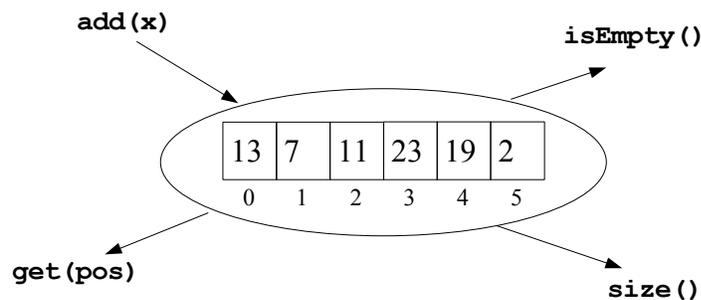


Figura 1: Jerarquía de la clase `ArrayList`

A continuación presentaremos los constructores y métodos más importantes de esta clase, para un detalle más exacto y completo revisar los JAVADOCS.

Constructores

<i>Constructor</i>	<i>Descripción</i>
<code>ArrayList()</code>	Construye una lista vacía con una capacidad inicial de 10.
<code>ArrayList(int)</code>	Construye una lista vacía con una capacidad inicial especificada.

Métodos

<i>Tipo de retorno</i>	<i>Nombre del Método</i>	<i>Descripción</i>
boolean	<code>add(E)</code>	Añade el elemento dado al final de la lista.
void	<code>add(int,E)</code>	Añade el elemento dado en la posición especificada.
E	<code>get(int)</code>	Retorna el elemento de la posición dada
void	<code>set(int,E)</code>	Establece un nuevo elemento E en la posición especificada.
boolean	<code>isEmpty()</code>	Verifica si la lista está vacía.
E	<code>remove(int)</code>	Remueve el elemento de la posición dada.
int	<code>size()</code>	Retorna el número de elementos de la lista.

Ejemplo de operaciones más utilizadas:

Declaración de un ArrayList de enteros

```
ArrayList<Integer> numeros;
```

Creación de un ArrayList de enteros

```
numeros = new ArrayList<Integer>();
```

Añadiendo elementos al ArrayList

```
numeros.add(13);  
numeros.add(7);  
numeros.add(11);
```

Lógicamente el ArrayList se vería así:

13	7	11
0	1	2

Cada elemento ocupa una posición en el ArrayList que es accedida mediante un índice que empieza en 0.

Conociendo el tamaño del ArrayList

```
int tamaño = numeros.size(); // 3
```

Obteniendo un elemento del ArrayList

```
Integer x = numeros.get(0); // 13
```

Borrando un elemento del ArrayList

```
Integer x = numeros.remove(0); //13
```

Lógicamente el ArrayList quedaría así:

7	11
0	1

Existen otras clases que modelan colecciones de objetos como HashMap, HashSet, Vector, etc. todas ellas propias de Java. También existen estructuras primitivas para manejar conjuntos de elementos conocidas como arreglos.

Relaciones entre objetos

Los 2 tipos de relaciones más importantes entre objetos son:

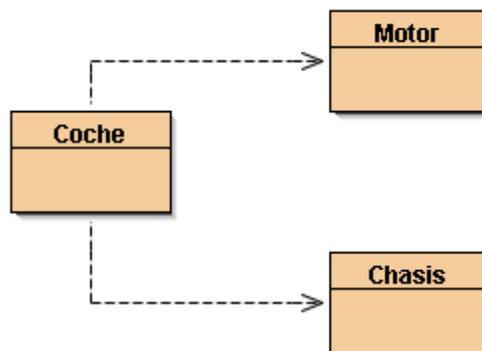
- La Composición
- La Herencia

Composición (tiene un)

Un objeto puede estar formado a partir de otros objetos. Por ejemplo un país, está formado de muchas personas, un continente está formado de muchos países, el mundo está formado de muchos continentes, etc.

Este tipo de relación responde afirmativamente a la pregunta:

¿<Contenedor> **tiene un** <Contenido>?



```
class Motor
{
    [...]
}
```

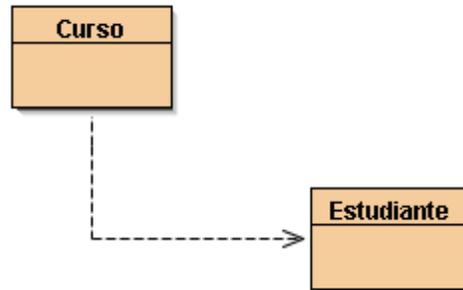
```
class Chasis
{
    [...]
}
```

```
class Coche
{
    Motor m;
    Chasis c;
    [...]
}
```

¿Un Coche tiene un Motor? --> SI

¿Un Coche tiene un Chasis? --> SI

¿<Contenedor> **tiene muchos** <Contenido>?



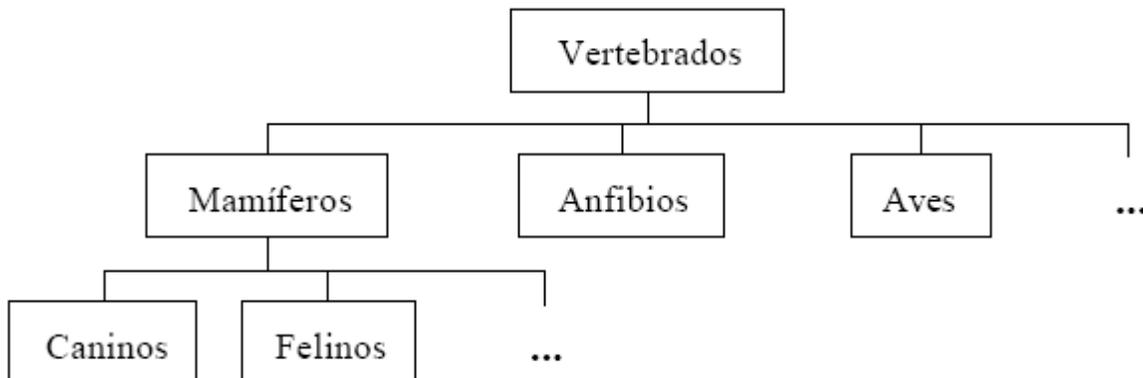
```
class Curso
{
    ArrayList<Estudiante> estudiantes;
}
```

```
class Estudiante
{
    [...]
}
```

¿Un Curso tiene muchos Estudiantes? --> SI

Herencia (es un)

La herencia es uno mecanismo, por medio del cual una clase, llamada *subclase*, se deriva de otra, llamada entonces *superclase*, de manera que extiende su funcionalidad. La idea es la de partir de las situaciones más generales e ir *derivando* hacia las más particulares, creando categorías, de la misma forma en que piensa el ser humano.



Las subclasses "heredan" características y métodos de las superclases (excepto los constructores).

Supongamos, por ejemplo, que tenemos la clase Aula y la clase Laboratorio definidas como sigue:

```
class Aula
{
    String nombre;
    int capacidadMaxima;

    int numAsientos;

    [...]

}
```

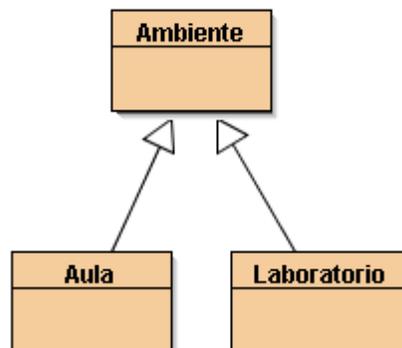
```
class Laboratorio
{
    String nombre;
    int capacidadMaxima;

    int numComputadoras;

    [...]

}
```

Las partes comunes se pueden agrupar en una misma clase, manteniendo las otras dos clases con las partes no comunes y heredando de esta nueva clase con la palabra reservada extends.



```
class Ambiente
{
    String nombre;
    int capacidadMaxima;

    [...]

}
```

```
class Aula extends Ambiente
{
    int numAsientos;

    [...]

}
```

```
class Laboratorio extends Ambiente
{
    int numComputadoras;

    [...]

}
```

¿Un Aula **es un** Ambiente? --> SI

¿Un Laboratorio **es un** Ambiente? --> SI

Recursividad

Definición

Recursión es una técnica de programación en el cual un método puede llamarse a sí mismo. Esto seguramente suena como algo raro de hacer o un grave error del que escribió esto. Sin embargo, la recursión una de las más interesantes y una de las técnicas sorprendentemente efectivas en programación.

Algo es recursivo si se define en términos de sí mismo (cuando para definirse hace mención a sí mismo). Para que una definición recursiva sea válida, la referencia a sí misma debe ser relativamente más sencilla que el caso considerado.

Ejemplo: definición de un número natural.

- ✓ El $N^{\circ} 0$ es natural.
- ✓ El $N^{\circ} n$ es natural si $n-1$ lo es.

En un algoritmo recursivo distinguimos como mínimo 2 partes:

a) Caso trivial, base o de fin de recursión:

Es la respuesta más sencilla a un problema. Es un caso donde el problema puede resolverse sin tener que hacer uso de una nueva llamada a sí mismo. Evita la continuación indefinida de las partes recursivas.

b) Parte puramente recursiva:

Relaciona el resultado del problema con resultados de casos más simples. Se hacen nuevas llamadas a la función, pero están más próximas al caso base.

Ejemplo:

Un problema clásico: El factorial de un número.

Todos conocemos que la definición matemática del factorial de un número n es la siguiente:

$$n! = 1*2*3*\dots*n \text{ ó } n! = n*(n-1)*(n-2)*\dots*1$$

Por definición matemática, también sabemos que:

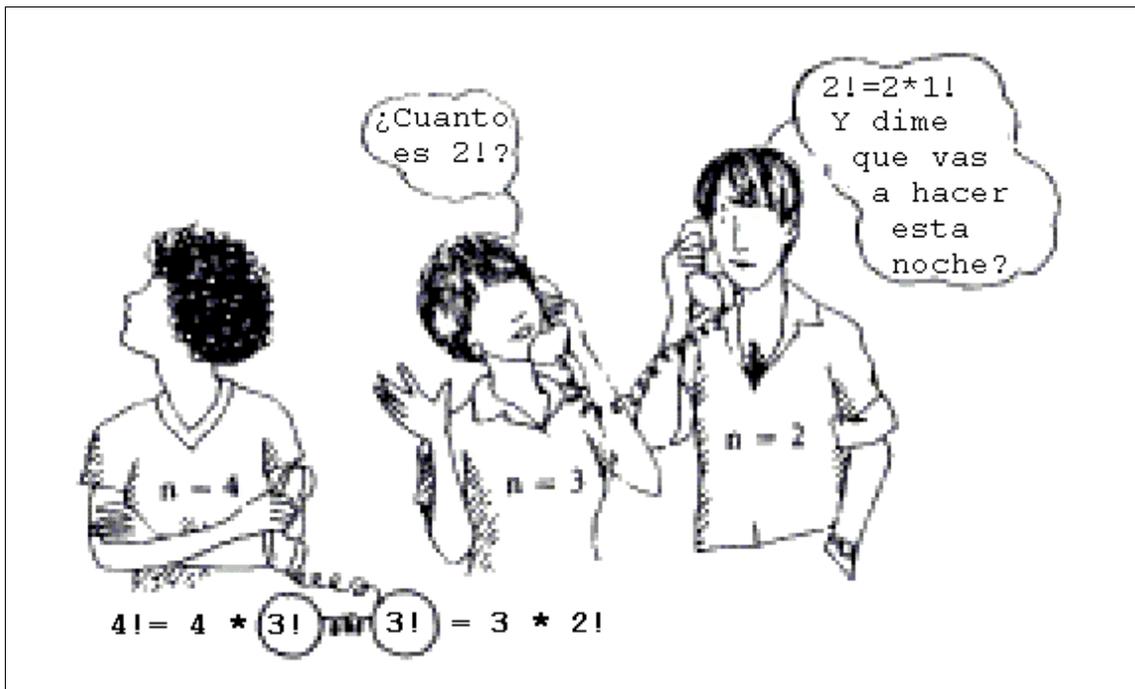
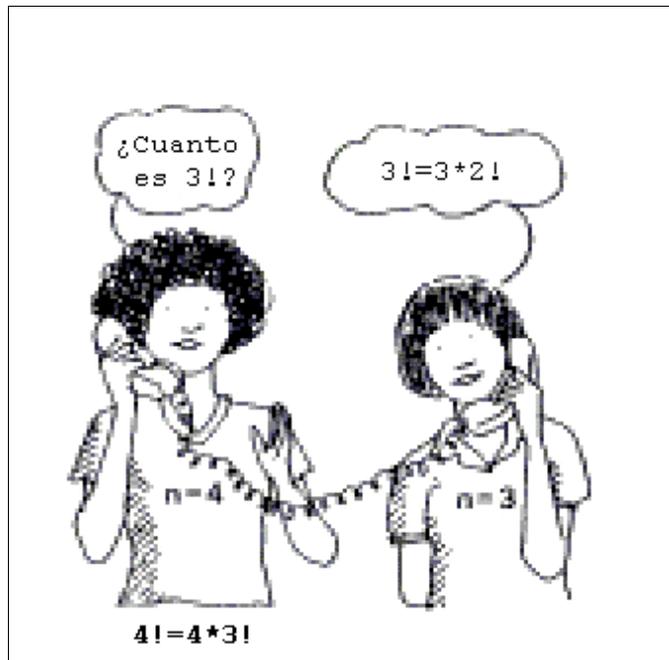
$$0! = 1$$

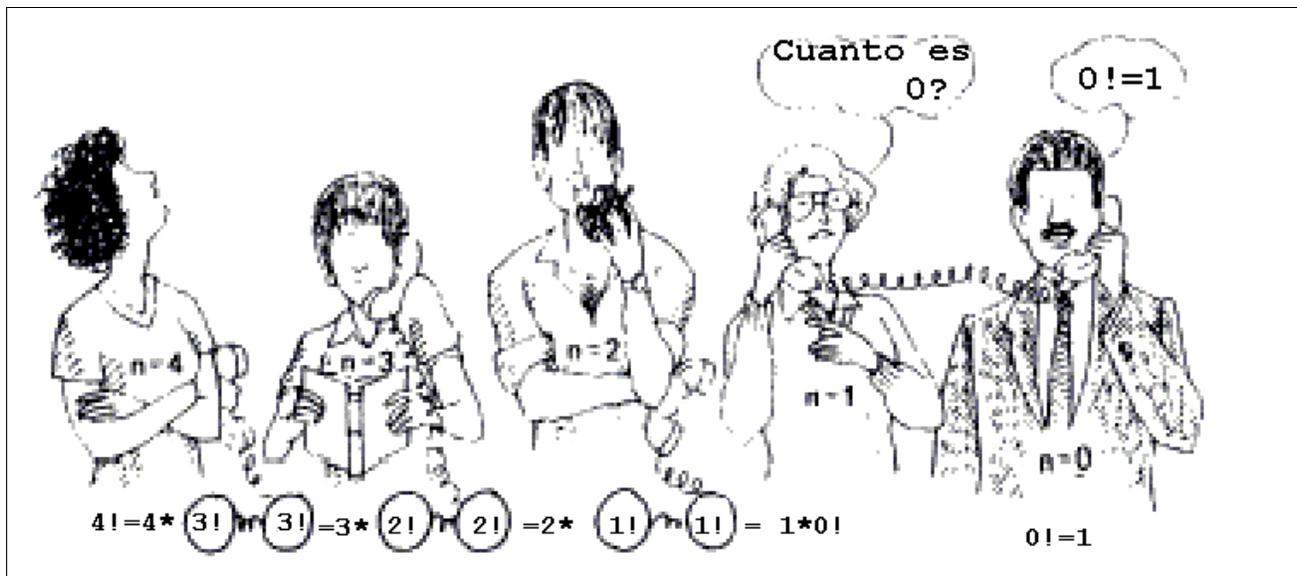
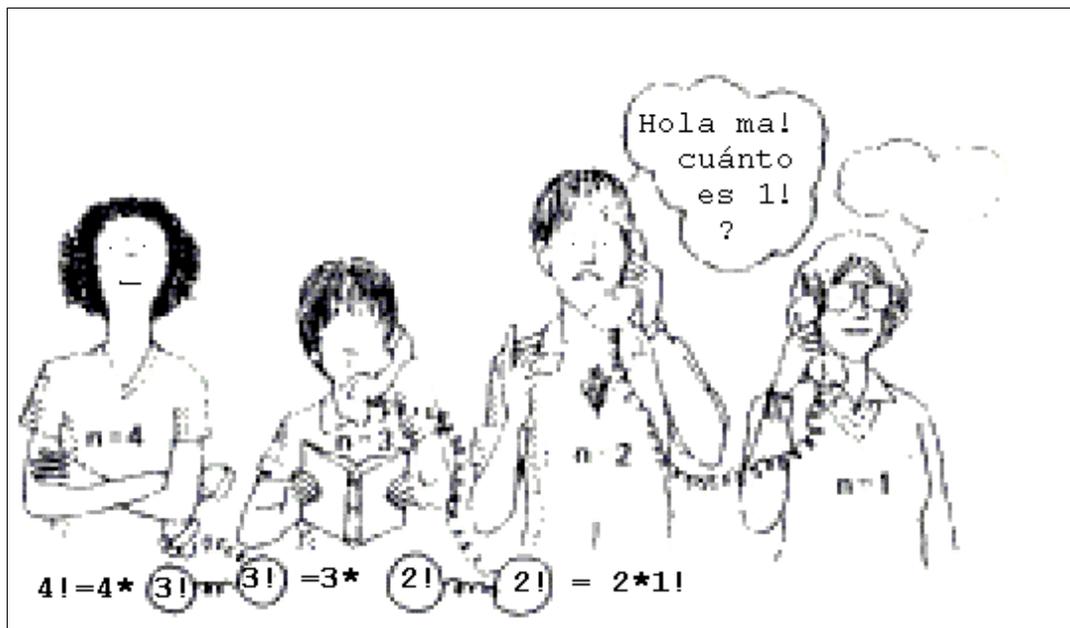
Con lo que tendríamos la siguiente la siguiente definición de factorial:

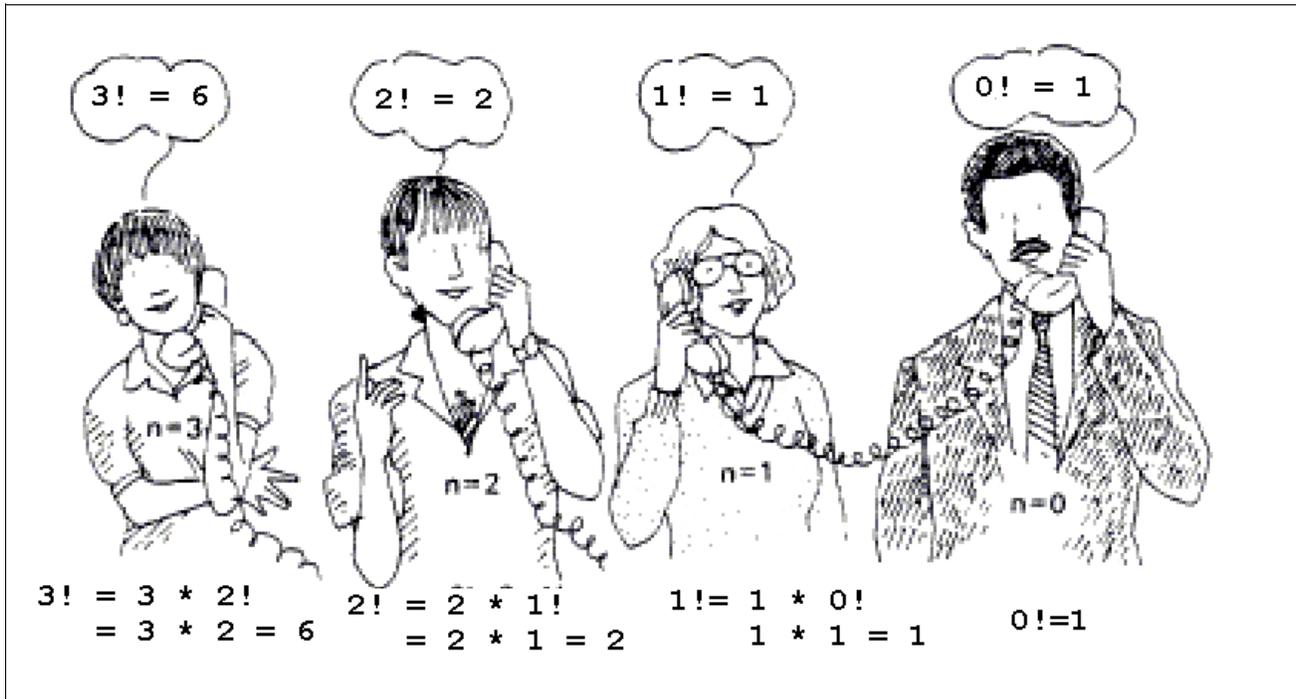
$$n! = \begin{cases} 1 & \text{Si } n = 0 \longrightarrow \text{Caso base} \\ n*(n-1)! & \text{Otros casos} \longrightarrow \text{Parte recursiva} \end{cases}$$

Un problema que puede resolverse de manera recursiva, debe tener por lo menos 1 caso base y 1 parte recursiva, sino no hay recursión.

Para ver más o menos como funciona la recursividad, vamos a ver una analogía de como funciona la recursividad con el siguiente ejemplo:







Como en el ejemplo, las **llamadas recursivas** siempre tienden hacia el caso base, hasta la forma de solucionar más fácil y una vez ahí, se van devolviendo los resultados de las llamadas hasta llegar a la llamada inicial.

Ahora, para implementarlo en un lenguaje de programación como Java, sólo tenemos que traducir nuestros casos bases y partes recursivas:

$$n! = \begin{cases} 1 & \text{Si } n = 0 \longrightarrow \text{Caso base} \\ n * (n-1)! & \text{Otros casos} \longrightarrow \text{Parte recursiva} \end{cases}$$

En Java tendríamos un método que recibe como parámetro un número y devuelve como resultado otro número:

```
public int factorial(int n)
{
    if (n==0)
        return 1;
    else
        return n*factorial(n-1);
}
```

→ *Caso base*

→ *Parte recursiva*

Prueba de Escritorio

Veamos como podemos ver el funcionamiento de nuestro método recursivo haciendo una pequeña prueba de escritorio. Probemos intentando calcular el factorial de 4.

<i>n</i>	<i>factorial(4)</i>	<i>A la IDA</i> ↓	<i>A la VUELTA</i> ↑
4	4 * factorial(3)	Se ejecuta la parte recursiva	Se devuelve el resultado de evaluar 4*6 = 24
3	3 * factorial(2)	Se ejecuta la parte recursiva	Se devuelve el resultado de evaluar 3*2 = 6
2	2 * factorial(1)	Se ejecuta la parte recursiva	Se devuelve el resultado de evaluar 2*1 = 2
1	1 * factorial(0)	Se ejecuta la parte recursiva	Se devuelve el resultado de evaluar 1*1 = 1
0	1	Se ejecuta el caso base	Se devuelve 1

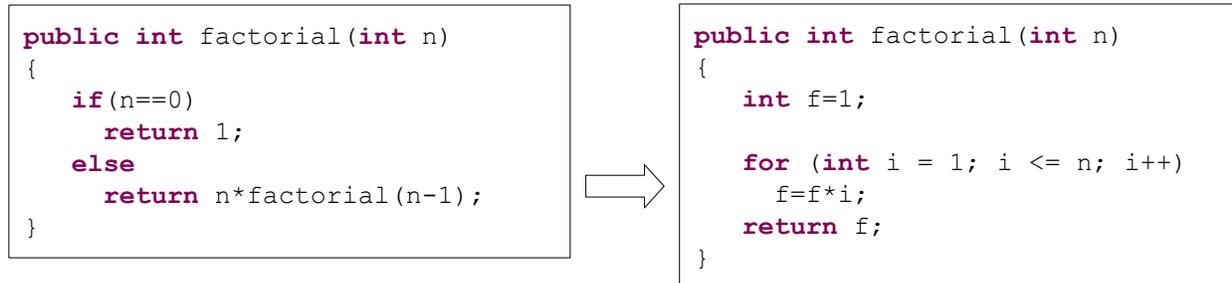
```
class Factorial
{
    public int factorial(int n)
    {
        if(n==0)
            return 1;
        else
            return n*factorial(n-1);
    }
}
```

```
class Principal
{
    public static void main(String[] args)
    {
        Factorial f = new Factorial();
        int res = f.factorial(4);
        System.out.println(res); //24
    }
}
```

Tipos de Recursión

Recursividad Simple

Aquella en cuya función sólo aparece una llamada recursiva. Se puede transformar con facilidad en algoritmos iterativos. Ej: factorial



Recursividad Múltiple

Se da cuando hay más de una llamada a sí misma dentro del cuerpo de la función, resultando más difícil de hacer de forma iterativa. Ej: Fibonacci.

```
public int fibonacci(int n)
{
    if(n==1)
        return 0;
    if(n==2)
        return 1;
    else
        return fibonacci(n-1)+fibonacci(n-2);
}
```

Recursividad Anidada

En algunos de los argumentos de la llamada hay una nueva llamada a sí misma. Ej: La función de Ackerman:

```
public int ackerman (int m, int n)
{
    if (m == 0)
        return n+1;
    if (n == 0)
        return ackerman(n-1, 1);
    else
        return ackerman( m-1, ackerman(m, n-1) );
}
```

Recursividad Cruzada o Indirecta

Son algoritmos donde una función provoca una llamada a sí misma de forma indirecta, a través de otras funciones. Ej: Par o Impar:

```
public boolean par(int n)
{
    if (n == 0)
        return true;
    else
        return impar(n - 1);
}

public boolean impar(int n)
{
    if (n == 0)
        return false;
    else
        return par(n - 1);
}
```

Reglas fundamentales de la recursividad

a) Caso base:

Siempre se deben tener algunos casos base, que se puedan resolver sin recursión.

Esta regla nos garantiza que un programa recursivo va a terminar siempre. Cuando un programa recursivo se ejecuta, muchas funciones quedan pendientes producto de recursión misma. Pero todas las llamadas recursivas se realizarán hasta que se alcance un caso base, a partir del cual se retorne en el otro sentido con resultados concretos de cada llamada hasta la primera llamada recursiva.

b) Progreso:

*Para los casos que deben resolverse recursivamente,
la llamada recursiva siempre debe tender hacia un caso base.*

Esta regla nos asegura que las llamadas recursivas no se alejen del caso base, más al contrario se dirijan hacia él.

c) Regla de diseño:

Todas las llamadas recursivas funcionan correctamente.

Esta regla es importante porque significa que cuando se diseñan programas recursivos, generalmente no es necesario conocer los detalles del funcionamiento interno si se tiene que rastrear la gran cantidad de llamadas recursivas. Con frecuencia, es en extremo difícil seguir la secuencia real de llamadas recursivas. Por supuesto, en muchos casos esto indica el buen uso de la recursión, ya que la computadora se usa para encargarse de los detalles complicados.

d) Regla del interés compuesto:

El trabajo nunca se debe duplicar resolviendo el mismo ejemplar de un problema en llamadas recursivas separadas.

Esta regla nos prohíbe repetir el trabajo de la recursión inutilmente, haciendo una llamada recursiva con el mismo ejemplar más de una vez.

Un claro ejemplo de un problema resuelto recursivamente que no cumple con ésta regla es fibonnaci:

```
public int fibonacci(int n)
{
    if (n==1)
        return 0;
    if (n==2)
        return 1;
    else
        return fibonacci(n-1)+fibonacci(n-2);
}
```

El anterior programa halla el enésimo número de la serie de fibonacci:

0, 1, 1, 2, 3, 5, 8, etc.
1 2 3 4 5 6 7 ...

Para una llamada: fibonacci(4) = fibonacci(3) + **fibonacci(2)**
fibonacci(3) = **fibonacci(2)** + fibonacci(1)

Y como vemos para este caso sencillo, existe doble llamada recursiva para hallar el fibonacci(2).

La Pila de Recursión

La memoria del ordenador se divide (de manera lógica, no física) en varios segmentos:

- ✓ **Segmento de código:** Parte de la memoria donde se guardan las instrucciones del programa en código máquina.
- ✓ **Segmento de datos:** Parte de la memoria destinada a almacenar las variables estáticas.
- ✓ **Montículo:** Parte de la memoria destinada a las variables dinámicas.
- ✓ **Pila del programa:** Parte destinada a las variables locales y parámetros de la función que está siendo ejecutada.

Llamada a una función

- 1) Se reserva espacio en la pila para los parámetros de la función y sus variables locales.
- 2) Se guarda en la pila la dirección de la línea de código desde donde se ha llamado a la función.
- 3) Se almacenan los parámetros de la función y sus valores en la pila.
- 4) Al terminar la función, se libera la memoria asignada en la pila y se vuelve a la instrucción actual.

Llamada a una función recursiva

En el caso recursivo, cada llamada genera un nuevo ejemplar de la función con sus correspondientes objetos locales:

- 1) La función ejecutará normalmente hasta la llamada a sí misma. En ese momento se crean en la pila nuevos parámetros y variables locales.
- 2) El nuevo ejemplar de función comienza a ejecutarse.
- 3) Se crean más copias hasta llegar a los casos bases, donde se resuelve directamente el valor, y se va saliendo liberando la memoria hasta llegar a la primera llamada (última en cerrarse).

Análisis de Algoritmos

Definición

Algoritmo es sinónimo de procedimiento computacional y es fundamental para las ciencias de la computación¹.

Un algoritmo es una secuencia finita de instrucciones, cada cual con un significado concreto y cuya ejecución genera un tiempo finito. Un algoritmo debe terminar en un tiempo finito.

Algoritmo es toda receta, proceso, rutina, método, etc. que además de ser un conjunto de instrucciones que resuelven un determinado problema, cumple las siguientes condiciones:

1. **Ser finito.** La ejecución de un algoritmo acaba en un tiempo finito; un procedimiento que falle en la propiedad de la finitud es simplemente un procedimiento de cálculo.
2. **Ser preciso.** Cada instrucción de un algoritmo debe ser precisa; deberá tenerse en cuenta un rigor y no la ambigüedad, esta condición es la definibilidad: cada frase tiene un significado concreto.
3. **Posee entradas.** Las entradas se toman como un conjunto específico de valores que inicializan el algoritmo.
4. **Posee salida.** Todo algoritmo posee una o más salidas; la salida es la transformación de la entrada.
5. **Ser eficaz.** Un algoritmo es eficaz cuando resuelve el problema.
6. **Ser eficiente.** Un algoritmo es eficiente cuando resuelve el problema de la mejor manera posible, o sea utilizando la mínima cantidad de recursos.

1 Del libro: Algoritmos Análisis & Diseño - R. Lecca

Una vez que tenemos un algoritmo que resuelve un problema y podemos decir que es de alguna manera correcto, un paso importante es tener idea de la cantidad de recursos, como tiempo de procesador o espacio en la memoria principal que requerirá.

El objetivo de este capítulo es:

- Conocer los factores que influyen en la eficiencia de un algoritmo.
- Aprender a calcular el tiempo que emplea un algoritmo.
- Aprender a reducir el tiempo de ejecución de un programa de días o años a fracciones de segundo.

Factores que influyen en la eficiencia de un algoritmo

Podemos tomar en cuenta muchos factores que sean externos al algoritmo como la computadora donde se ejecuta (hardware y software) o factores internos como la longitud de entrada del algoritmo. Veamos algunos de estos factores.

El Hardware

Por ejemplo: procesador, frecuencia de trabajo, memoria, discos, etc.

El Software

Por ejemplo: sistema operativo, lenguaje de programación, compilador, etc.

La longitud de entrada

El enfoque matemático considera el tiempo del algoritmo como una función del tamaño de entrada. Normalmente, se identifica la longitud de entrada (tamaño de entrada), con el número de elementos lógicos contenidos en un ejemplar de entrada, por ejemplo: en un algoritmo que calcula el factorial de un número, la longitud de entrada sería el mismo número, porque no es lo mismo calcular el factorial de 4 que calcular el factorial de 1000, las iteraciones que tenga que hacer el algoritmo dependerá de la entrada. De igual manera se puede considerar como longitud de

entrada: al tamaño de un arreglo, el número de nodos de una lista enlazada, el número de registros de un archivo o el número de elementos de una lista ordenada). A medida que crece el tamaño de un ejemplar del programa, generalmente, crece el tiempo de ejecución. Observando cómo varía el tiempo de ejecución con el tamaño de la entrada, se puede determinar la *tasa de crecimiento* del algoritmo, expresado normalmente en términos de n , donde n es una medida del tamaño de la entrada.

La tasa de crecimiento de un problema es una medida importante de la eficiencia ya que predice cuánto tiempo se requerirá para entradas muy grandes de un determinado problema.

Para que un algoritmo sea eficiente, se debe optimizar el tiempo de ejecución y el espacio en la memoria, aunque se producirá la optimización de uno a costa del otro.

Análisis de Algoritmos

El análisis de algoritmo que hacemos en este capítulo toca únicamente el punto de vista temporal (tiempo de ejecución de un algoritmo) y utilizamos como herramienta el lenguaje de programación Java.

Es difícil realizar un análisis simple de un algoritmo que determine la cantidad exacta de tiempo requerida para ejecutarlo. La primera complicación es que la cantidad exacta de tiempo dependerá de la implementación del algoritmo y de la máquina real en que se ejecuta. El análisis normalmente debe ser independiente de la computadora (hardware y software) y del lenguaje o máquina que se utilice para implementar el algoritmo. La tarea de calcular el tiempo exacto requerido suele ser bastante pesado.

Elementos de un algoritmo

Un algoritmo es un **conjunto de instrucciones** ordenados de manera lógica que resuelven un problema. Éstas instrucciones a su vez pueden ser: enunciados simples (sentencias) o enunciados compuestos (estructuras de control). El tiempo de ejecución dependerá de como esté organizado ese conjunto de instrucciones, pero

nunca será constante.

Es conveniente utilizar una función $T(n)$ para representar el número de unidades de tiempo tomadas por un algoritmo de cualquier entrada de tamaño n .

Ahora bien, para calcular el $T(n)$ de un algoritmo, vamos a ver cómo calcular el tiempo de ejecución de las instrucciones que podría tener un algoritmo.

Enunciados simples (sentencias)

Para los enunciados simples como:

- La asignación de un valor a una variable.
- La invocación a otro método.

Podríamos calcular minuciosamente el tiempo de ejecución, pero es una tarea que se la dejamos a los grandes matemáticos que hacen cálculos precisos, pero para nosotros todo enunciado simple se ejecutará en 1 (una unidad de tiempo que podría ser 1 segundo o 1 nanosegundo...) y listo :).

Ejemplo:

```
1 //Algoritmo para mostrar el mayor de 2 # enteros positivos.
2 //Sin utilizar estructuras de control :)
3
4 a = in.nextInt(); —————▶ 1
5
6 b = in.nextInt(); —————▶ 1
7
8 m = (Math.abs(a+b)+Math.abs(a-b))/2; —————▶ 1
9
10 System.out.println("El mayor es: "+m); —————▶ 1
```

$T(n) = 4$ Se lee: El tiempo de ejecución es 4

Enunciados compuestos (estructuras de control)

Todas las estructuras de control tienen 2 partes: una cabecera que puede ser evaluado como una sentencia simple y un cuerpo que puede tener un conjunto de instrucciones y será evaluado de acuerdo al tipo de instrucciones que tenga.

Para las estructuras de control:

- selectivas (if, if-else, switch)
- repetitivas (for, while, do-while)

Haremos un repaso de la sintaxis y semántica operacional que sea general para otros lenguajes de programación.

Selectivas

IF

Sintaxis:

```
Cabecera: if(<expresión lógica>
           {
Cuerpo:    //instrucciones
           }
```

Semántica:

Si el resultado de evaluar la expresión lógica de la cabecera es verdad entonces se ejecutan las instrucciones del cuerpo.

Ahora bien, ¿cómo calculamos su tiempo de ejecución? Pues como en el caso de las sentencias simples diremos que la cabecera se ejecuta en 1 unidad de tiempo y luego calculamos el tiempo de ejecución del cuerpo.

```
Cabecera: if(<expresión lógica> } → 1
Cuerpo:   { //instrucciones } →  $T_c(n)$ 
```

Ejemplo:

```
1 //Algoritmo para mostrar el valor absoluto de un # leído del teclado
2
3 x = LeerTeclado.readInt(); —————▶ I
4
5 if(x<0) —————▶ I
6 {
7     x = -x; —————▶ I
8 }
9
10 System.out.println("El valor absoluto es: "+x); —————▶ I
```

Análisis en base al promedio o en base al peor caso

Un algoritmo puede ejecutarse más rápido ante ciertos datos de entrada y lento ante otros. En dichos casos nos hacemos la siguiente pregunta: ¿Utilizar un análisis basado en el promedio o en el peor caso?. En el primer caso, requerimos definir una distribución de probabilidades de todo el conjunto de datos de entrada, el cuál típicamente es una tarea difícil. En cambio, en el segundo caso, no se requieren esas herramientas. Por lo tanto, en el resto del capítulo, vamos a caracterizar tiempos de ejecución en términos del peor caso.

Tomando en cuenta el algoritmo anterior, si el número fuese positivo sólo se ejecutarían las líneas 3, 5 y 10, en cambio si fuese negativo se ejecutarían las líneas 3, 5, 7 y 10. Entonces podemos ver que el tiempo de ejecución puede tomar 2 casos:

- el mejor de los casos (cuando se ejecuta la menor cantidad de instrucciones)
- y el peor de los casos (cuando se ejecuta la mayor cantidad de instrucciones)

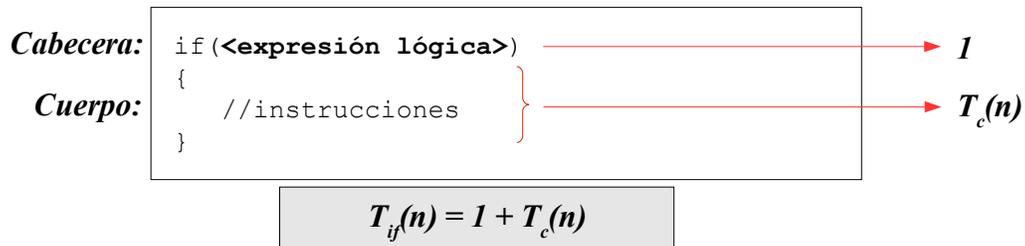
Para nuestro ejemplo anterior el tiempo de ejecución sería:

MEJOR DE LOS CASOS: $T(n) = 3$

PEOR DE LOS CASOS: $T(n) = 4$

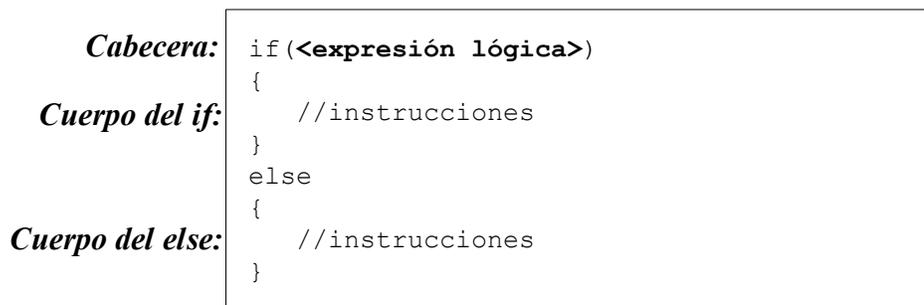
Normalmente suele hacerse el análisis del tiempo de ejecución tomando en cuenta el peor de los casos ya que estaríamos subestimando al algoritmo si nos basáramos en el mejor de los casos.

A partir de ahora todo nuestro análisis se hará en base al peor de los casos.



IF-ELSE

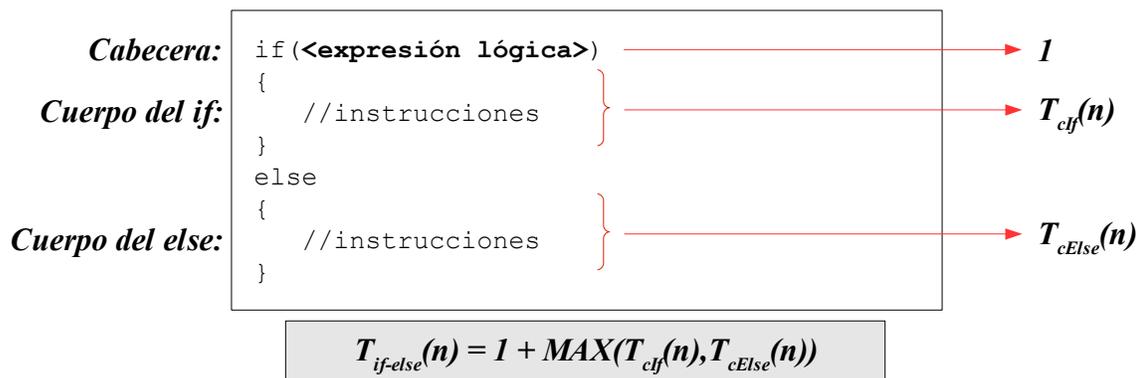
Sintaxis:



Semántica:

Si el resultado de evaluar la expresión lógica de la cabecera es verdad entonces se ejecutan las instrucciones del cuerpo if en caso contrario se hacen las instrucciones del cuerpo else.

¿Cómo calcular el tiempo de ejecución de una estructura if-else? Pues como nuestro análisis es en el peor de los casos, tomamos en cuenta el mayor de los tiempos de ejecución del cuerpo if y del cuerpo else.



Ejemplo:

```
1 //Algoritmo que muestra ordenado de forma descendente 2 números.
2
3 if(x>y) → 1
4 {
5     System.out.println(x+", "+y); → 1
6 }
7 else
8 {
9     System.out.println("Los números son"); → 1
10    System.out.println(y+", "+x); → 1 } 2
11 }
```

Repetitivas

FOR

Sintaxis:

```
Cabecera: for(<inicialización>; <expresión lógica>; <actualización>)
Cuerpo:   {
           //instrucciones
           }
```

Semántica:

Se realiza la instrucción de inicialización, luego se evalúa la expresión lógica, si ésta es verdadera entonces se realizan las instrucciones del cuerpo, después se pasa a la actualización de variable y vuelve a evaluarse la expresión lógica.

¿Cómo calcular el tiempo de ejecución de una estructura de control for? Para realizar nuestro análisis, consideraremos que un for tiene básicamente 2 partes: una cabecera y un cuerpo. Estableceremos una fórmula que nos dirá cuantas veces se ejecuta la cabecera porque conociendo cuantas veces lo hace podemos saber cuantas veces entra hacer las instrucciones de su cuerpo.

```
Cabecera: for(<inicialización>; <expresión lógica>; <actualización>) →  $x$ 
Cuerpo:   {
           //instrucciones }  $T_c(n)$  →  $*(x-1)$ 
           }
```

Si decimos que la cabecera se ejecuta un número x de veces entonces se entrarán a hacer las instrucciones del cuerpo $x-1$ veces.

$$x = \frac{\text{distancia}}{(\text{valor que actualiza})} + 2$$

Donde distancia es lo que recorre la variable para llegar desde la inicialización hasta ya no cumplir la expresión lógica.

$$T_{for}(n) = x + T_c(n) * (x-1)$$

Ejemplo:

```
1 //programa que muestra los números naturales de 1 hasta n
2
3 for (int i = 1; i <= n; i++) → x
4 {
5     System.out.println(i);
6 }
7
8
```

El problema se reduce a conocer cuantas veces se ejecuta la cabecera del for. Veamos algunos pasos para saber este valor:

1. Conocer la distancia que recorre la variable i .
Como i se mueve desde 1 hasta n , entonces la distancia sería $n-1$.
2. Conocer el valor que actualiza.
La variable i se incrementa de 1 en 1, entonces el valor que actualiza es 1.
3. Reemplazar en la fórmula el valor de la distancia y el valor que actualiza:

$$x = \frac{\text{distancia}}{(\text{valor que actualiza})} + 2 \implies x = \frac{(n-1)}{1} + 2 \implies x = n + 1$$

$$T_{for}(n) = x + T_c(n) * (x-1)$$

$$T_{for}(n) = (n-1) + 1 * ((n-1) - 1)$$

$$T_{for}(n) = 2n - 3$$

WHILE

Sintaxis:

```
Cabecera: while(<expresión lógica>
{
Cuerpo:    //instrucciones
}
```

Semántica:

Se evalúa la expresión lógica, si ésta es verdad se ejecutan las instrucciones de su cuerpo y luego se regresa a evaluar la cabecera nuevamente. Esto se repite mientras la expresión lógica sea verdadera.

Para calcular el tiempo de ejecución de un while, también es importante conocer las veces que se ejecuta su cabecera y las veces que entrará a realizar su cuerpo será una vez menos.

```
Cabecera: while(<expresión lógica>) →  $x$ 
{
Cuerpo:    //instrucciones }  $T_c(n)$  →  $*(x-1)$ 
}
```

$$T_{while}(n) = x + T_c(n) * (x-1)$$

DO - WHILE

Sintaxis:

```
Cuerpo: do  
{  
    //instrucciones  
Cabecera: }while(<expresión lógica>);
```

Semántica:

Se ejecutan las instrucciones del cuerpo y luego recién se evalúa la expresión lógica de la cabecera, si ésta es verdad se ejecutan las instrucciones de su cuerpo nuevamente. Esto se repite mientras la expresión lógica sea verdadera.

Una estructura de control do-while se diferencia de un while en que las veces que entra a su cuerpo y las veces que evalúa su cabecera siempre van a ser lo mismo.

```
Cuerpo: do  
{  
    //instrucciones }  $T_c(n) \rightarrow *x$   
Cabecera: }while(<expresión lógica>);  $\rightarrow x$ 
```

$$T_{do-while}(n) = x + T_c(n)*x$$

Orden de Magnitud (Notación O grande)

Cuando se tratan de algoritmos, el cálculo detallado del tiempo de ejecución de todas las operaciones primitivas llevaría mucho tiempo. Además, ¿qué importancia tendría el número de instrucciones primitivas ejecutadas por un algoritmo?

Es más útil en el análisis de algoritmos, ver la velocidad de crecimiento del tiempo de ejecución como una función del tamaño de la entrada n , en lugar de realizar cálculos detallados. Es más significativo saber que un algoritmo crece, por ejemplo, proporcionalmente a n , a razón de un factor constante pequeño que depende del hardware o software y puede variar en un cierto rango, dependiendo de una entrada n específica. Esto lo que se conoce como orden de magnitud " $O(g(n))$ " o notación asintótica o notación "O grande".

El orden de magnitud se utiliza para comparar la eficiencia de los algoritmos. La notación "O grande", se centra en el **término dominante** (es decir, en aquel que más crece cuando n aumenta), e ignora el resto de términos. Por lo tanto, desprecia también los coeficientes de los términos.

Por ejemplo para un algoritmo de tiempo de ejecución = $an^2 + bn + c$

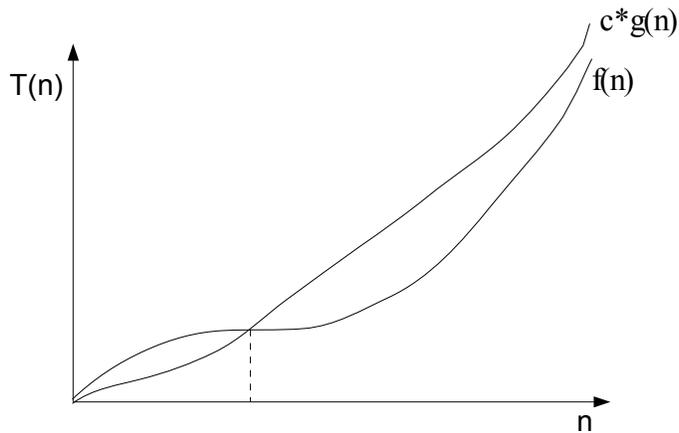
Su orden de magnitud sería:

$$O(an^2 + bn + c) = O(an^2) = O(n^2)$$

Definición conceptual del Orden de Magnitud

Sean $f(n)$ y $g(n)$ funciones definidas sobre enteros no negativos. Diremos que $f(n)$ es $O(g(n))$ si existe un número real constante $c > 0$ y un entero constante $n_0 >= 1$, tal que $f(n) \leq c * g(n)$ para todo entero $n \geq n_0$.

Por consiguiente, $g(n)$ es un límite superior para $f(n)$, como se ilustra en la siguiente figura.



El orden de magnitud es ampliamente utilizado para caracterizar los tiempos de ejecución en términos de la longitud de entrada n , el cual varía de problema en problema, pero es usualmente una noción intuitiva del "tamaño" del problema. Asimismo, el orden de magnitud nos permite ignorar factores constantes y términos de orden menor y apuntar a los componentes principales de una función que influyen en su crecimiento.

Cuando se dice que $f(n)$ es del $O(g(n))$, se está garantizando que la función $f(n)$ crece a una velocidad no mayor que $g(n)$; así $g(n)$ es una cota superior de $f(n)$.

Propiedades del Orden de Magnitud

1. $O(f(x)) + k = O(f(x))$
2. $O(k f(x)) = O(f(x))$
3. $O(f(x)) * O(g(x)) = O(f(x) * g(x))$
4. $O(f(x)) + O(g(x)) = \max(O(f(x)), O(g(x)))$

Estructuras de Datos

Definición

Una estructura de datos es un conjunto de elementos, del mismo tipo, que se relacionan entre sí y que se pueden operar como un todo.

El componente básico de la estructura de datos es la celda; las estructuras de datos se implementan a través de los lenguajes (en nuestro caso, lo haremos con Java), y son un modelo que caracteriza y permite almacenar y utilizar una determinada organización de datos.

Operaciones

Sobre una estructura de datos se puede efectuar diferentes tipos de operaciones, entre las más importantes están:

Navegar por la estructura

Esta es una operación básica que garantiza que se puede recuperar información almacenada.

Búsqueda.

Permite determinar si un elemento se encuentra o no en la estructura.

Consulta de la información.

Permite obtener información de uno o más elementos de la estructura.

Copia parcial o total.

Mediante esta operación se puede obtener total o parcialmente una estructura con características similares a la original.

Prueba.

Permite determinar si uno o varios elementos cumplen determinadas condiciones.

Modificación.

Permite variar parcial o totalmente el contenido de la información de los elementos de la estructura.

Inserción.

Es aquella mediante la cual se incluye un nuevo elemento en la estructura.

Eliminación.

Como su nombre lo indica, es la que permite suprimir elementos de la estructura.

Verificar si es vacía

Permite determinar si existen o no elementos sobre la estructura.

Clasificación

Estructuras de Datos Interna y Externa

Si una estructura de datos reside en la memoria central de la computadora se denomina **estructura de datos interna**. Recíprocamente, si reside en un soporte externo, se denomina **estructura de datos externa**.

Las estructuras de datos internas pueden ser de dos tipos:

- ✓ Estructuras de Datos Estáticas.
- ✓ Estructuras de Datos Dinámicas.

Estructuras de Datos Estáticas

Tienen un número fijo de elementos que queda determinado desde la declaración de la estructura en el comienzo del programa. Ejemplo los arreglos.

Las estructuras de datos estáticas, presentan dos inconvenientes:

- x La reorganización de sus elementos, si ésta implica mucho movimiento puede ser muy costosa. Ejemplo: insertar un dato en un arreglo ordenado.
- x Son estructuras de datos estáticas, es decir, el tamaño ocupado en memoria es fijo, el arreglo podría llenarse y si se crea un arreglo de tamaño grande se estaría desperdiciando memoria.

Estructuras de Datos Dinámicas

Las estructuras de datos dinámicas nos permiten lograr un importante objetivo de la programación orientada a objetos: la reutilización de objetos.

Al contrario de un arreglo, que contiene espacio para almacenar un número fijo de elementos, una estructura dinámica de datos se amplía y contrae durante la ejecución del programa.

A su vez, este tipo de estructuras se pueden dividir en dos grandes grupos según la forma en la cual se ordenan sus elementos.

- Lineales
- No lineales

Estructuras de Datos Lineales y No Lineales

Entre las lineales: listas, pilas y colas. Las no lineales: árboles y grafos.



Objetivos

El objetivo del presente capítulo es hacer énfasis en tres aspectos:

- ✓ Presentar las diferentes estructuras de datos.
- ✓ Desarrollar habilidades necesarias en ti, para construir los algoritmos que manipulan de manera eficiente estas estructuras.
- ✓ Y por último, que seas capaz de seguir una metodología para diseñar (definir y justificar) las estructuras de datos más adecuadas para resolver un problema.

Estructuras de Datos Lineales

En éste tipo de estructuras los elementos se encuentran ubicados secuencialmente. Al ser dinámica, su composición varía a lo largo de la ejecución del programa que lo utiliza a través de operaciones de inserción y eliminación.

Dependiendo del tipo de acceso a la secuencia, haremos la siguiente distinción:

- **Listas:** podemos acceder (insertar y eliminar) por cualquier lado.
- **Pilas:** sólo tiene un único punto de acceso fijo a través del cual se añaden, se eliminan o se consultan elementos.
- **Colas:** tiene dos puntos de acceso, uno para añadir y el otro para consultar o eliminar elementos.

Listas

Una lista en su sentido amplio, es un conjunto de elementos del mismo tipo donde cada elemento tiene un único predecesor (excepto el primero) y un único sucesor (excepto el último) y cuyo número de elementos es variable.

$$S = \{a, b, c, d, \dots z\}$$

Ejemplo: Una lista de letras

Operaciones

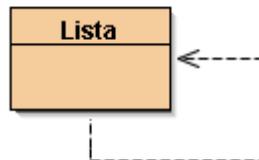
Entre las operaciones permitidas están:

- Verificar si es vacía.
- Insertar, eliminar o localizar un elemento.
- Determinar el tamaño de la lista.
- Etc.

En fin, una lista podría tener todo tipo de operaciones que le permitan ser manipulada como un arreglo por ejemplo.

Veamos dos maneras de definir e implementar a una lista. La primera, trata de una definición recursiva de la lista y la segunda, se basa en el concepto de celda o nodo¹.

Definición recursiva.



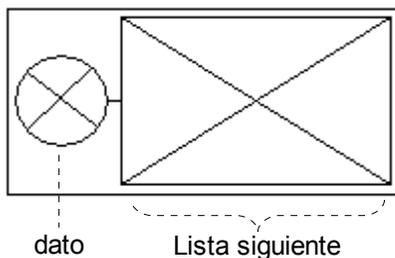
En esta forma de implementación, una lista puede definirse como un dato seguido de otra lista o como vacía.

Ejemplo: La lista [1, 2, 3] de manera recursiva puede verse así:

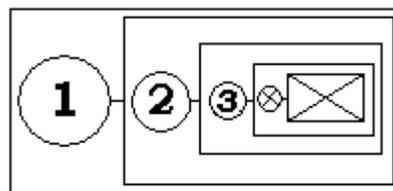
[1, 2, 3] = 1 → [2,3]	: 1 seguido de la lista [2, 3]
[2, 3] = 2 → [3]	: 2 seguido de la lista [3]
[3] = 3 → []	: 3 seguido de una lista vacía

De manera gráfica:

a) Lista vacía



b) Lista no vacía



Entonces una lista tiene como atributos a un dato² y la referencia a la siguiente lista.

Lista vacía: Cuando el dato y la referencia siguiente no existen (son null).

Lista no vacía: Cuando existe el dato y la siguiente lista por lo menos es vacía.

1 Algo que explicaremos más adelante.

2 Que puede ser de cualquier tipo, dependiendo de que tipos de datos querramos almacenar en nuestra lista (números, caracteres, etc)

Ahora pensemos en la implementación; si quisieramos una lista de números enteros, entonces el tipo de dato del dato de la lista sería `int` o `Integer`. De igual manera si quisieramos que la lista almacene cadenas, entonces el tipo sería `String`, y si quisieramos objetos personas, pues sería `Persona` y así sucesivamente. Nos conviene colocar el tipo de dato como `Object`¹ ya que de esa manera permitimos que pueda almacenarse todo tipo de elementos (números, caracteres, objetos en general).

El modelo UML sería el siguiente:



Y su implementación:

```
public class Lista
{
    private Object dato;
    private Lista sig;

    public Lista()
    {
        dato = null;
        sig = null;
    }

    public boolean vacia()
    {
        return sig == null;
    }

    public void insertarFinal(Object x)
    {
        if (vacía())
        {
            dato = x;
            sig = new Lista();
        } else
            sig.insertarFinal(x);
    }

    [...]
}
```

1 Esto se debe a que `Object` es la superclase de todas las clases en Java.

La clase *Lista.java*

Constructores

```
/**
 * Constructor por defecto, crea una lista vacia.
 */
public Lista()
{
    dato = null;
    sig = null;
}

/**
 * Crea una lista con un sólo dato.
 * @param d es el único dato de la lista.
 */
public Lista(Object d)
{
    dato = d;
    sig = new Lista();
}

/**
 * Crea una lista a partir de otra
 * @param otra
 */
public Lista(Lista otra)
{
    dato = otra.dato;
    sig = otra.sig;
}
```

Métodos

Verificar si es vacía

```
/**
 * Verifica si la lista está vacía.
 * Una lista está vacía, cuando el dato
 * y la referencia a la siguiente lista son null.
 */
public boolean vacia()
{
    return dato==null&&sig == null;
}
```

Insertar un dato al final

```
/**
 * Añade un nuevo elemento al final de la lista.
 * @param x es el nuevo elemento a ser insertado
 */
public void insertarFinal(Object x)
{
    if (vacía())
    {
        dato = x;
        sig = new Lista();
    }
    else
        sig.insertarFinal(x);
}
```

Insertar un dato al inicio

```
/**
 * Añade un nuevo elemento al principio de la lista.
 * @param x es el nuevo elemento a ser insertado
 */
public void insertarInicio(Object x)
{
    sig = new Lista(this);
    dato = x;
}
```

Insertar un dato en la k-ésima posición de la lista

```
/**
 * Inserta un nuevo dato sobre la lista en la posición especificada
 * La primera posición es 0. Si la posición destino no existe,
 * entonces el nuevo dato se inserta al final
 * @param x es el dato a insertar
 * @param k es la posición donde debe insertarse
 */
public void insertar(Object x, int k)
{
    if(vacía() || k==0)
        insertarInicio(x);
    else
        sig.insertar(x, k-1);
}
```

Eliminar un dato de la lista

```
/**
 * Elimina el dato que sea igual al especificado
 * @param x es el dato a buscar
 * @return el dato eliminado
 */
public Object eliminar(Object x)
{
    Object res = null;
    if(!vacía())
    {
        if(dato.equals(x))
        {
            res = dato;
            dato = sig.dato;
            sig = sig.sig;
        }
        else
            res = sig.eliminar(x);
    }
    return res;
}
```

Obtener el tamaño de la lista

```
/**
 * Obtiene el tamaño de la lista
 * @return la cantidad de elementos de la lista
 */
public int tamaño()
{
    if (vacía()) return 0;
    return 1 + sig.tamaño();
}
```

Buscar un dato sobre la lista

```
/**
 * Busca un elemento en la lista
 * @param x es el elemento a ser buscado
 * @return el resultado de comprobar si el dato está en la lista.
 */
public boolean buscar(Object x) {
    if (vacía()) return false;
    if (dato.equals(x)) return true;
    return sig.buscar(x);
}
```

Obtener un elemento de cualquier posición de la lista

```
/**
 * Permite obtener un elemento de la lista
 * de una posición especificada
 * @param pos es la posición del objeto
 * @return el elemento de esa posición
 */
public Object obtener(int pos)
{
    if(vacia()) return null;

    if(pos==0) return dato;

    return sig.obtener(pos-1);
}
```

Obtener en formato cadena a la lista

```
/**
 * Devuelve en cadena a la lista [a,b,c...z]
 * @return todos los elementos en formato cadena
 */
public String toString()
{
    String res = "[" + resto() + "]";
    return res;
}

//método recursivo que obtiene todos los elementos de la lista
private String resto()
{
    String res = "";

    if (!vacia())
    {
        res = dato.toString();

        if (!sig.vacia())
            res = res + ", " + sig.resto();
    }
    return res;
}
```

Modificar el elemento de una posición dada

```
/**
 *
 * @param pos La posición que debe modificarse
 * @param x El nuevo elemento
 */
public void modificar(int pos, Object x)
{
    if(vacia() || pos<0) return;
    if(pos==0)
    {
        dato = x;
    }
    else
    {
        sig.modificar(pos-1, x);
    }
}
```

Eliminar el elemento de una posición dada

```
/**
 *
 * @param pos la posición del elemento
 * @return el elemento de la posición
 */
public Object eliminar(int pos)
{
    if( vacia() ) return null;

    if(pos==0)
    {
        Object res = dato;
        dato = sig.dato;
        sig = sig.sig;
        return res;
    }
    else
        return sig.eliminar(pos-1);
}
```

Podríamos parar ahí la implementación de métodos y ver un ejemplo de utilización de la lista. Por ejemplo, crear una lista y llenarla. con qué?, pues con lo que sea, recuerda que es una lista de objetos.

```
[...]  
Lista lista = new Lista();  
lista.insertarFinal("Hola Nena");  
lista.insertarFinal(13);  
lista.insertarFinal(true);  
System.out.println(lista);  
[...]
```

Y con esto tendríamos una lista, parecida a esto:

```
[ "Hola Nena", 13, true ]  
      0         1     2
```

Como ves, podemos insertar cualquier tipo de objeto sobre ésta lista, y vemos que cada elemento, ocupa una posición dentro de la lista siguiendo el estándar de Java para los índices, el primer elemento está en la posición 0, el segundo en la posición 1 y así sucesivamente.

Pero que tal si quisieramos obtener un elemento de la lista, por ejemplo el primero. Entonces nuestro código inocente sería algo como esto:

```
[...]  
String x = lista.obtener(0); -> Error, un object no es un String  
[...]
```

Tendríamos un error, pues recordemos que lo que devuelve el método obtener, es un Object y no un String como pensabamos, bueno, pero eso se puede arreglar así:

```
[...]  
String x = (String)lista.obtener(0);  
[...]
```

Lo arreglamos haciendo una simple conversión de tipos (casting).

La ventaja de tener una lista de objetos, es tener una lista que puede

almacenar todo tipo de datos, números, caracteres, etc. La desventaja es a la hora de acceder a esos datos, no saber exactamente a que tipo de dato estamos accediendo. La siguiente línea muestra un ejemplo claro de esto:

```
[...]  
String x = (String) lista.obtener(1);  
[...]
```

Este código compila correctamente, intentamos obtener el segundo elemento de la lista, pero al ejecutar el código nos aparece el siguiente error:

```
java.lang.ClassCastException: java.lang.Integer  
[...]
```

Que significa que el casting que intentabamos hacer no es válido. Esto es debido a que si nos fijamos que elemento realmente está en la posición 1, vemos que no es una cadena sino un entero y lo estamos obligando a que sea cadena, esto no lo permite Java.

Una solución no muy buena (pésima) para el anterior problema, sería tener una implementación específica para cada tipo de dato. Si quisieramos una lista de números enteros, entonces en nuestra anterior implementación modificar el tipo de dato de `Object` a `int` y trabajar todas las operaciones para este tipo de dato; el problema de hacerlo de esta manera es que tendríamos una implementación de lista para cada diferente tipo de dato. Una solución mejor, y ahora posible de hacerlo en Java desde la versión 1.5 es parametrizar el tipo de dato de nuestra estructura.

Genericidad o Parametrización de tipos

Consiste en introducir un parámetro en la definición del tipo para poder usarlo con distintos tipos, esto quiere decir que en lugar de escribir especificaciones, representaciones e implementaciones independientes para cada tipo, se escribe una sola, donde el tipo se reemplaza por un parámetro.

El tipo de dato también es variable

Haremos otra implementación de lista con parametrización de tipo. O sea una `Lista<E>` donde E^1 es el parámetro genérico formal de "cualquier tipo".

```
public class Lista<E>
{
    private E dato;
    private Lista<E> sig;

    public Lista()
    {
        dato=null;
        sig=null;
    }

    public boolean vacia(){
        return sig==null;
    }

    public void insertarFinal(E x)
    {
        if(vacia())
        {
            dato = x;
            sig = new Lista<E>();
        }
        else
            sig.insertarFinal(x);
    }

    public int tamaño()
    {
        if(vacia()) return 0;
        return 1 + sig.tamaño();
    }

    public E obtener(int pos)
    {
        if(vacia()) return null;
        if(pos==0) return dato;
        return sig.obtener(pos-1);
    }
    [...]
}
```

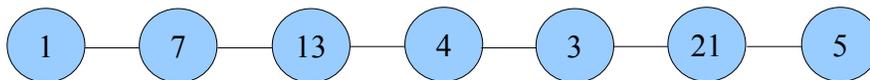
Ahora, si necesitamos tener una lista para manejar enteros y que todas sus operaciones sean para enteros, escribimos lo siguiente:

1 Podría haber sido cualquier letra o palabra, es una variable.

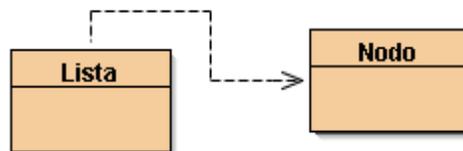
```
[...]  
Lista<Integer> lista = new Lista<Integer>();  
lista.insertarFinal(13);  
lista.insertarFinal("Hola"); -> Error de compilación, el método insertarFinal  
de Lista<Integer> sólo acepta números enteros  
[...]
```

Definición de Lista en base a Nodos

Otra forma de ver a una lista es como un conjunto de celdas o nodos que se encuentran enlazados entre si.

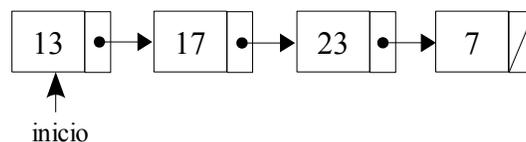


Y dependiendo de los enlaces y como se los tenga, podemos tener una lista de simple enlace, lista de doble enlace, lista de simple enlace circular o lista de doble enlace circular.

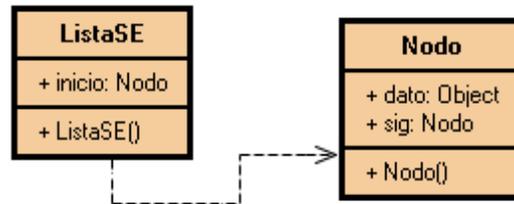


Lista de Simple Enlace

Cada nodo de la lista tiene 2 campos: un campo que tiene el dato y otro que tiene la referencia del siguiente nodo.



Los nodos de la lista son enlazados por medio de los campos enlaces. El último nodo de una lista de simple enlace, por definición no tiene siguiente o no existe o en términos de programación es NULL.

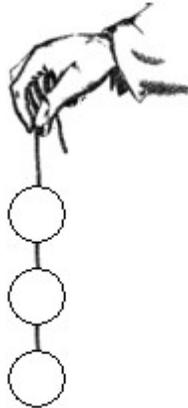


A continuación se muestra la implementación de un *Nodo* de simple enlace genérico, esto para poder parametrizar el *tipo de dato* de dato.

```
class Nodo<E>
{
    private E dato;
    private Nodo<E> sig;

    public Nodo(E n)
    {
        dato = n;
        sig = null;
    }
    public E getDato()
    {
        return dato;
    }
    public Nodo<E> getSig()
    {
        return sig;
    }
    public void setDato(E x)
    {
        dato=x;
    }
    public void setSig(Nodo<E> x)
    {
        sig=x;
    }
}
```

Para tener una lista enlazada con estos nodos, basta con tener la referencia solamente del primero, al igual que para sostener un conjunto de pelotitas enlazadas solo tenemos que agarrarnos de la primera pelotita.



```
class ListaSE<E>
{
    private Nodo<E> inicio;

    [...]
}
```

La clase *ListaSE.java*

Constructor

```
/**
 * Crea una lista vacia
 *
 */
public ListaSE()
{
    inicio = null;
}
```

Métodos

Verificar si es vacía

```
/**
 * Verifica si la lista está vacia
 * @return el resultado de ver si la lista está vacía
 */
public boolean vacia(){
    return inicio==null;
}
```

Insertar un dato al inicio

```
/**
 * Inserta un dato al inicio de toda la lista
 * @param x es el nuevo dato para la lista
 */
public void insertarInicio(E x){
    Nodo<E> nuevo = new Nodo<E>(x);
    if(!vacía())
        nuevo.setSig(inicio);
    inicio = nuevo;
}
```

Insertar un dato al final

```
/**
 * Inserta un nuevo dato al final de toda la lista
 * @param x es el dato a insertar al Final
 */
public void insertarFinal(E x)
{
    Nodo<E> nuevo = new Nodo<E>(x);

    if(vacia())
    {
        inicio = nuevo;
    }
    else
    {
        Nodo<E> actual = inicio;

        while(actual.getSig() != null)
        {
            actual = actual.getSig();
        }

        actual.setSig(nuevo);
    }
}
```

Obtener el tamaño de la lista

```
/**
 * Cuenta la cantidad de elementos de la lista
 * @return el número de elementos de la lista
 */
public int tamaño()
{
    int contador=0;

    for(Nodo<E> x=inicio; x!=null; x=x.getSig())
    {
        contador++;
    }

    return contador;
}
```

Obtener el elemento de una posición dada

```
/**
 * @param pos La posición del elemento
 * @return el elemento de la posición pos
 */
public E obtener(int pos)
{
    int i=0;
    Nodo<E> x = inicio;
    while(x!=null && i!=pos)
    {
        x = x.getSig();
        i++;
    }
    return x==null? null : x.getDato();
}
```

Eliminar el elemento de una posición dada

```
/**
 * @param pos La posición del elemento
 * @return el elemento de la posición pos
 */
public E eliminar(int pos)
{
    int i=0;
    Nodo<E> an = null;
    Nodo<E> ac = inicio;

    while(ac!=null && i!=pos)
    {
        an = ac;
        ac = ac.getSig();
        i++;
    }
    E res = null;
    if(ac!=null)
    {
        res = ac.getDato();
        if(an==null)
            inicio = inicio.getSig();
        else
            an.setSig(ac.getSig());
    }
    return res;
}
```

Buscar un elemento en la lista

```
/**
 *
 * @param d el elemento a buscar
 * @return el resultado de ver si elemento está o no
 */
public boolean buscar(E d)
{
    Nodo<E> x = inicio;

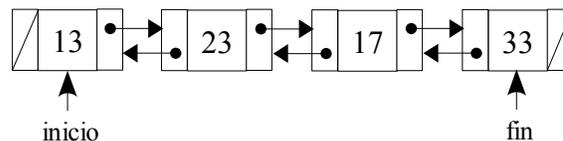
    while(x!=null&&!x.getDato().equals(d))
    {
        x = x.getSig();
    }

    return x!=null;
}
```

Algo que cabe resaltar en esta nueva implementación es el hecho de que no usamos recursividad para realizar los métodos ya que no es necesario, algo que no se puede evitar en la definición recursiva de Lista.

Lista de Doble Enlace

En las listas lineales estudiadas anteriormente el recorrido de ellas sólo podía hacerse en un único sentido: de izquierda a derecha. Sin embargo en muchas ocasiones es necesario recorrer la lista en ambas direcciones. Este tipo de listas se denomina *Listas de Doble Enlace* o *Listas Doblemente Enlazadas*. En este tipo de listas cada nodo ha de tener 3 campos: un campo que tiene el dato, otro que tiene la referencia del siguiente nodo y otro que tiene la referencia del nodo anterior.



Los nodos de la lista son enlazados por medio de los campos enlaces. El último nodo de una lista de doble enlace, no tiene siguiente y el primer nodo de la lista no tiene anterior. Veamos una implementación de nodo de doble enlace:

```
class NodoD<E>
{
    private E dato;
    private NodoD<E> sig;
    private NodoD<E> ant;

    public NodoD(E n)
    {
        dato = n;
        sig = null;
    }

    public E getDato()
    {
        return dato;
    }

    public void setDato(E x)
    {
        dato=x;
    }

    public NodoD<E> getSig()
    {
        return sig;
    }

    public void setSig(NodoD<E> x)
    {
        sig=x;
    }

    public NodoD<E> getAnt()
    {
        return ant;
    }

    public void setAnt(NodoD<E> ant)
    {
        this.ant = ant;
    }
}
```

La clase *ListaDE.java*

```
class ListaDE<E>
{
    private NodoD<E> inicio, fin;

    [...]
}
```

Constructor

```
/**
 * Constructor por defecto, crea una lista vacia.
 */
public ListaDE()
{
    inicio = null;
    fin = null;
}
```

Métodos

Insertar un elemento al inicio

```
/**
 *
 * @param x El elemento a insertar
 */
public void insertarInicio(E x)
{
    NodoD<E> nuevo = new NodoD<E>(x);

    if(vacia())
        fin = nuevo;
    else
    {
        nuevo.setSig(inicio);
        inicio.setAnt(nuevo);
    }

    inicio = nuevo;
}
```

Insertar un elemento al final

```
/**
 *
 * @param x El elemento a insertar
 */
public void insertarFinal(E x)
{
    NodoD<E> nuevo = new NodoD<E>(x);

    if(vacia())
        inicio = nuevo;
    else
    {
        fin.setSig(nuevo);
        nuevo.setAnt(fin);
    }
    fin = nuevo;
}
```

Obtener los elementos de la lista en formato cadena

```
/**
 * @return Retorna todos los elementos de la lista en formato cadena
 */
public String toString()
{
    String res="";

    for (NodoD<E> x=inicio; x!=null; x=x.getSig())
    {
        res = res + x.getDato()+" ";
    }

    return res;
}
```

Eliminar y obtener el primer elemento de la lista

```
/**
 *
 * @return Devuelve y elimina el primer elemento de la lista
 */
public E eliminarInicio()
{
    E res=null;
    if(!vacía())
    {
        res=inicio.getDato();
        if(inicio==fin)
            inicio=fin=null;
        else
        {
            inicio = inicio.getSig();
            inicio.setAnt(null);
        }
    }
    return res;
}
```

Eliminar y obtener el último elemento de la lista

```
/**
 *
 * @return Devuelve y elimina el último elemento de la lista
 */
public E eliminarFin()
{
    E res=null;

    if(!vacía())
    {
        res=fin.getDato();

        if(inicio==fin)
            inicio=fin=null;
        else
        {
            fin = fin.getAnt();
            fin.setSig(null);
        }
    }

    return res;
}
```

Buscar un elemento en la lista

```
/**
 *
 * @param pos La posición del elemento a borrar
 * @return El elemento de la posición dada
 */
public E eliminar(int pos)
{
    if(pos==0) return eliminarInicio();
    if(pos==tamaño()-1) return eliminarFin();
    else{
        NodoD<E> x = inicio;
        while(x!=null&&pos!=0)
        {
            x = x.getSig();
            pos--;
        }
        E res=null;
        if(x!=null)
        {
            res = x.getDato();
            x.getAnt().setSig(x.getSig());
            if(x.getSig()!=null)
                x.getSig().setAnt(x.getAnt());
        }
        return res;
    }
}
```

Buscar un elemento en la lista

```
/**
 *
 * @param d el elemento a buscar
 * @return el resultado de ver si elemento está o no
 */
public boolean buscar(E d)
{
    NodoD<E> x = inicio;

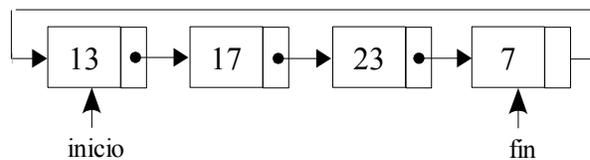
    while(x!=null&&!x.getDato().equals(d))
    {
        x = x.getSig();
    }

    return x!=null;
}
```

Lista Circular de Simple Enlace

Una lista circular es una lista lineal, en la que el último nodo enlaza con el primero. Entonces es posible acceder a cualquier elemento de la lista desde cualquier punto dado. Las operaciones sobre una lista circular resultan más sencillas, ya que se evitan casos especiales.

Cuando recorremos una lista circular, diremos que hemos llegado al final de la misma, cuando nos encontremos de nuevo en el punto de partida.



Las listas circulares presentan las siguiente ventajas respecto de las listas enlazadas simples:

- Cada nodo de una lista circular es accesible desde cualquier otro nodo de ella. Es decir, dado un nodo se puede recorrer toda la lista completa.
- Las operaciones de concatenación y división de listas son más eficaces con listas circulares.

Un inconveniente, por el contrario, es:

- Si no se tienen cuidado con las operaciones, se pueden tener ciclos infinitos.

```
class ListaSEC<E>
{
    private Nodo<E> inicio;
    private Nodo<E> fin;

    [...]
}
```

La clase *ListaSEC.java*

Constructor

```
/**
 * Crea una lista vacia
 *
 */
public ListaSEC()
{
    inicio = fin = null;
}
```

Métodos

Verificar si es vacía

```
/**
 * Verifica si la lista está vacia
 * @return el resultado de ver si la lista está vacía
 */
public boolean vacia(){
    return inicio==null&&fin==null;
}
```

Insertar un dato al inicio

```
/**
 *
 * @param x El nuevo elemento
 */
public void insertarInicio(E x)
{
    Nodo<E> nuevo = new Nodo<E>(x);

    if(vacia())
        fin = nuevo;
    else
        nuevo.setSig(inicio);

    inicio = nuevo;
    fin.setSig(inicio);
}
```

Insertar un dato al final

```
/**
 *
 * @param x Es el nuevo elemento
 */

public void insertarFinal(E x)
{
    Nodo<E> nuevo = new Nodo<E>(x);

    if(vacia())
        inicio = nuevo;
    else
        fin.setSig(nuevo);

    fin = nuevo;
    fin.setSig(inicio);
}
```

Obtener en formato cadena

```
/**
 * Devuelve la lista en formato cadena
 */

public String toString()
{
    if(vacia()) return "";

    String res = inicio.getDato()+" ";

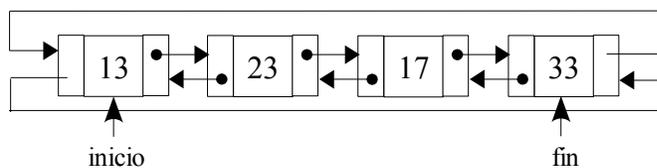
    for (Nodo<E> x=inicio.getSig(); x!=inicio; x=x.getSig())
    {
        res = res + x.getDato()+" ";
    }

    return res;
}
```

Lista Circular de Doble Enlace

Una lista circular de doble enlace es una lista lineal doblemente enlazada, en la que el último nodo tiene como siguiente al primero y el primer nodo tiene como anterior al último nodo. Esto mejora el acceso a cualquier elemento desde cualquier parte y en cualquier sentido.

Cuando recorremos una lista circular doblemente enlazada, diremos que hemos llegado al final de la misma, cuando nos encontremos de nuevo en el punto de partida.



```
class ListaDEC<E>
{
    private NodoD<E> inicio;
    private NodoD<E> fin;

    [...]
}
```

La clase *ListaDEC.java*

Constructor

```
/**
 * Crea una lista vacia
 *
 */
public ListaDEC()
{
    inicio = fin = null;
}
```

Métodos

Verificar si es vacía

```
/**
 * Verifica si la lista está vacía
 * @return el resultado de ver si la lista está vacía
 */
public boolean vacia()
{
    return inicio==null&&fin==null;
}
```

Insertar un dato al inicio

```
/**
 *
 * @param x El nuevo elemento
 */
public void insertarInicio(E x)
{
    NodoD<E> nuevo = new NodoD<E>(x);

    if(vacia())
        fin = nuevo;
    else
    {
        nuevo.setSig(inicio);
        inicio.setAnt(nuevo);
    }

    inicio=nuevo;
    inicio.setAnt(fin);
    fin.setSig(inicio);
}
```

Insertar un dato al final

```
/**
 *
 * @param x El nuevo elemento
 */
public void insertarFinal(E x)
{
    NodoD<E> nuevo = new NodoD<E>(x);

    if(vacia())
    {
        inicio = nuevo;
        fin = nuevo;
    }
    else
    {
        fin.setSig(nuevo);
        nuevo.setAnt(fin);
        fin = nuevo;
    }

    inicio.setAnt(fin);
    fin.setSig(inicio);
}
```

Obtener la lista en formato cadena

```
/**
 * Devuelve la lista en formato cadena
 */
public String toString()
{
    if(vacia()) return "";

    String res = inicio.getDato()+" ";

    for (NodoD<E> x=inicio.getSig(); x!=inicio; x=x.getSig())
    {
        res = res + x.getDato()+" ";
    }

    return res;
}
```

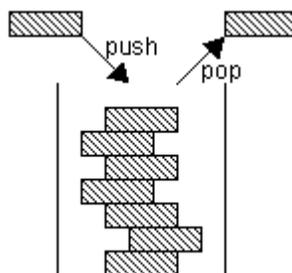
Pilas

Una pila es un concepto aplicable sobre estructuras de datos lineales, en la que la inserción y eliminación de datos se realiza sólo por un extremo denominado cima o tope.

Podemos hacernos una imagen más gráfica, pensando en una pila de bandejas en una cafetería, una pila de platos en un fregadero, una pila de latas en un expositor de un supermercado: en cualquiera de estos ejemplos, los elementos se retiran y se añaden por un mismo extremo. En una pila de platos podríamos intentar retirar uno de los intermedios con el consiguiente peligro de derrumbe. Sin embargo, en una estructura de datos de tipo pila, esto no es posible. El único elemento de la pila que podemos retirar es el situado en el tope de la misma, y si queremos retirar otro, será necesario previamente haber borrado todos los situados "por encima" de él.

Las pilas siguen la filosofía LIFO (Last In First Out), porque su característica principal es que el Último en Entrar es el Primero en Salir (UEPS).

Se define el tope o cima de la pila al único dato visible de la estructura que sería el último que se colocó (el que está encima).



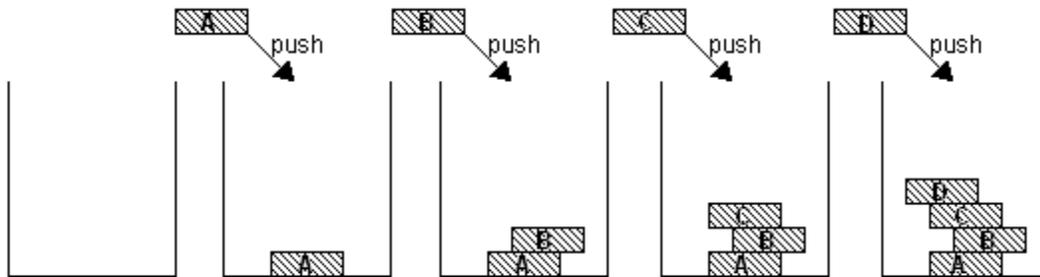
Operaciones

Entre las operaciones permitidas están:

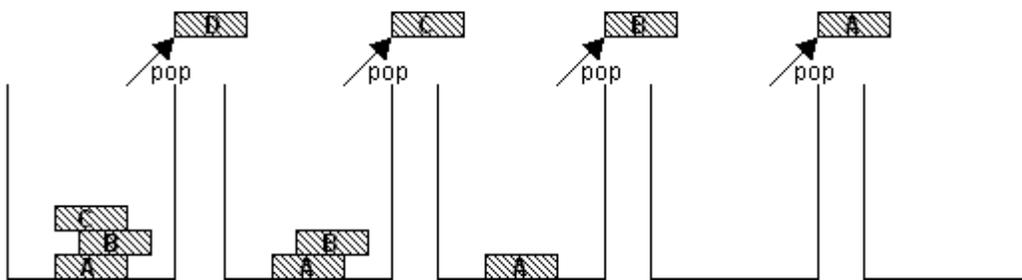
- Verificar si es vacía (empty)
- Insertar un dato sobre la pila (push)
- Eliminar un dato de la pila (pop)
- Inspeccionar la cima (top)

Veamos mediante un ejemplo gráfico, como cambia el contenido de una pila conforme aplicamos sucesivamente las operaciones de inserción y eliminación:

a) **Push**eando (apilando) elementos sobre la pila



b) **Pop**eando (desapilando) elementos de la pila



En el ejemplo anterior podemos observar que el tope (el elemento que está encima de todos) varía si operamos sobre la pila (si añadimos o eliminamos elementos). Si la pila está vacía, es decir, si no tiene ningún elemento, no hay tope o no está definido.

Las operaciones sobre las pilas deberían tardar una cantidad constante de tiempo en ejecutarse, independientemente del número de elementos apilados. De forma análoga, encontrar el periódico de hoy en una pila de periódicos es rápido, independientemente de lo alta que se la pila. Sin embargo, el acceso arbitrario a una pila no se soporta de forma eficiente, por lo que no se cataloga como opción.

Lo que hace que las pilas sean útiles es que hay muchas aplicaciones en las que sólo es necesario acceder al elemento más recientemente insertado.

A continuación se muestra una implementación de Pila dinámica, recursiva, genérica y además no heredable (esto por hacer respetar el concepto de pila).

```
public final class Pila<E>
{
    private E tope;
    private Pila<E> resto;

    public Pila()
    {
        tope = null;
        resto = null;
    }

    private Pila(Pila<E> otra)
    {
        tope = otra.tope;
        resto = otra.resto;
    }

    public boolean empty()
    {
        return tope==null&&resto==null;
    }

    public void push(E x)
    {
        resto = new Pila<E>(this);
        tope = x;
    }

    public E pop()
    {
        E res=null;
        if(!empty())
        {
            res = tope;
            tope = resto.tope;
            resto = resto.resto;
        }
        return res;
    }

    public E top()
    {
        return tope;
    }
}
```

Aplicaciones de las Pilas

Las pilas son utilizadas ampliamente para solucionar una amplia variedad de problemas. Se utilizan en compiladores, sistemas operativos y en programas de aplicación. Veamos algunas de las aplicaciones más interesantes.

- ✓ **Llamadas a métodos.**- Cuando se realizan llamadas a métodos, el programa principal debe recordar el lugar donde se hizo la llamada, de modo que pueda retornar allí cuando el método se haya terminado de ejecutar.
- ✓ **Almacenamiento temporal de páginas Web.**- Los navegadores de Internet almacenan las direcciones de las páginas recientemente visitadas en una pila. Cada vez que un usuario visita un nuevo sitio, las direcciones son introducidas en la pila de direcciones. El navegador después permite al usuario volver a las páginas visitadas previamente usando un botón.
- ✓ **El mecanismo de "deshacer-rehacer" en los editores de texto.**- Los editores de texto utilizan usualmente un mecanismo de "deshacer" (undo) que cancela las operaciones editadas actualmente para volver a las operaciones previas. Para ello se utiliza una pila.

En general, las aplicaciones de esta estructura de datos serán aquellas en las que los datos se manejen de modo dinámico siguiendo una política LIFO.

Colas

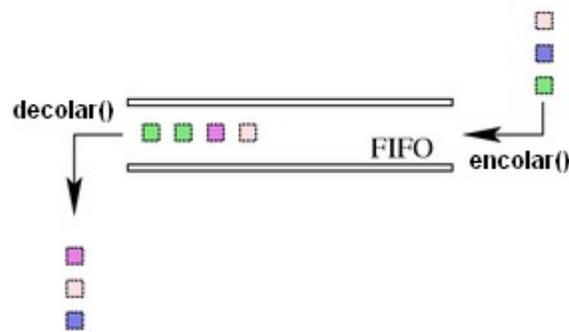
El concepto de cola es ampliamente utilizado en la vida real. Cuando nos situamos ante la taquilla del cine para obtener nuestra entrada, o cuando esperamos en el autoservicio de un restaurante solemos hacerlo en una cola. Esto significa que formamos una fila en la que el primero que llega es el primero en obtener el servicio y salir de la misma. Esta política de funcionamiento se denomina FIFO (First In First Out), es decir, el primer elemento en entrar es el primer elemento en salir (PEPS).



En la vida real puede perfectamente ocurrir que alguien pretenda saltarse su turno en una cola, o incluso que abandone la misma antes de que le toque el turno. Sin embargo, en ambos casos se está incumpliendo la política de funcionamiento de la cola y, estrictamente hablando, ésta deja de serlo.

Las colas son otro tipo de concepto también aplicables sobre estructuras de datos lineales. En una cola las eliminaciones se realizan por el principio (inicio) y las inserciones se realizan en el otro extremo (fin).

Se usan para almacenar datos que necesitan ser procesados según el orden de llegada.



Existen varias formas de implementar a una cola , de forma estática utilizando arreglos o bien, de manera dinámica utilizando el concepto de listas. De cualquier manera las operaciones sobre colas también deberían tardar una cantidad de tiempo constante en ejecutarse, independientemente del número de elementos que existan sobre ella.

Operaciones

Entre las operaciones permitidas están:

- Verificar si es vacía
- Insertar un dato sobre la cola (encolar)
- Eliminar un dato de la cola (decolar)
- Inspeccionar el primero (primero)

A continuación se muestra la implementación de Cola siguiendo el concepto de lista en base a nodos:

```
final class Cola<E>
{
    private Nodo<E> inicio, fin;

    public Cola() { inicio = fin = null; }

    public boolean vacia() { return inicio==null; }

    public void encolar(E n)
    {
        Nodo<E> nuevo = new Nodo<E>(n);
        if(vacia())
            inicio = nuevo;
        else
            fin.setSig(nuevo);
        fin=nuevo;
    }

    public E decolar()
    {
        if(vacia()) return null;
        E res=inicio.getDato();
        inicio = inicio.getSig();
        return res;
    }

    public E primero() { return vacia()?null:inicio.getDato(); }
}
```

Aplicaciones de las Colas

En informática existen numerosas aplicaciones de las colas. Por ejemplo, en un sistema de tiempo compartido suele haber un procesado central y una serie de periféricos compartidos: discos, impresoras, lectores de CD, etc. Los recursos se comparten por diferentes usuarios y se utiliza una cola para almacenar los programas o las peticiones de los diferentes usuarios que esperan su turno de ejecución. El procesador central atiende -normalmente- por riguroso orden de llamada del usuario; por tanto, todas las llamadas se almacenan en una cola.

Estructuras de Datos No Lineales

Introducción

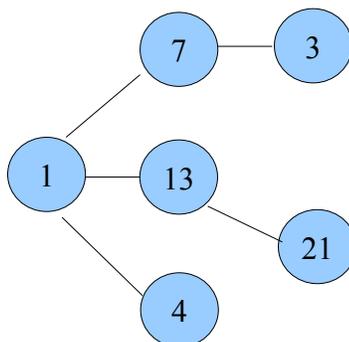
Las estructuras dinámicas lineales de datos tienen grandes ventajas de flexibilidad sobre las representaciones contiguas; sin embargo tienen un punto débil, son listas secuenciales, es decir, están dispuestas de modo que es necesario moverse a través de ellas una posición cada vez. Esta linealidad es típica de cadenas, de elementos que pertenecen a una sola dimensión. En éste capítulo se tratarán las **estructuras de datos no lineales**, que resuelven los problemas que plantean las listas lineales y en las que cada elemento puede tener diferentes elementos "siguientes", que introducen el concepto de estructuras de bifurcación.

Las principales estructuras de este tipo son:

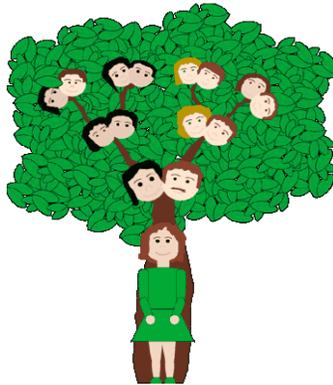
- **Árboles:** los elementos están organizados como un verdadero árbol.
- **Grafos:** los elementos están organizados como una red de datos.

Árboles

En su sentido amplio, un árbol permite almacenar información y organizarla de forma que tengan sucesores o elementos siguientes, como hijos en una forma de las ramas de un árbol.

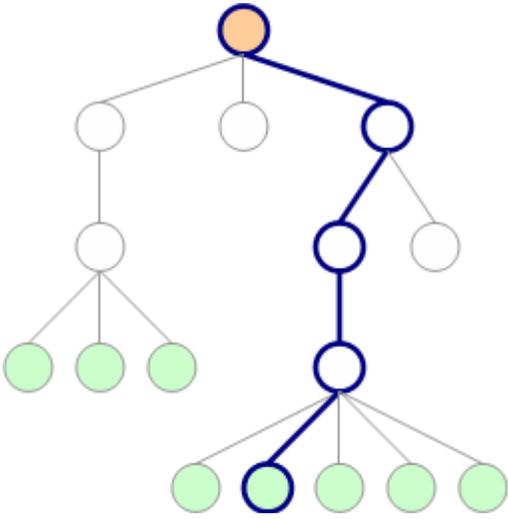


Todo el mundo tiene claro el concepto de árbol, al menos en su aspecto botánico. Sin embargo, los árboles no son sólo eso de lo que estamos rodeados cuando nos perdemos en un bosque, sino que también se usan en otros muchos ámbitos. Así por ejemplo, todos hemos manejado alguna vez el concepto de árbol genealógico, o hemos visto clasificaciones jerárquicas como las del reino animal. En todos esos casos manejamos el concepto de árbol.



Árbol genealógico

Algunos Conceptos

	<p>Raíz. -Es el primer nodo del árbol y es por donde se accede a él (sólo tiene sucesores), es decir, la <i>cabeza</i> del árbol siempre será la raíz.</p>	
	<p>Nodo Hoja: Aquel nodo que no tiene hijos o sucesores.</p>	
	<p>Nodo Interno. -Aquel nodo que tiene un antecesor y por lo menos un sucesor (ni raíz ni hojas).</p>	
	<p>Altura.- Es la cantidad de nodos que se recorren para llegar desde la raíz hasta el nodo hoja más alejado de todo el árbol.</p>	

Grado. - Cantidad máxima de hijos que puede tener cualquier nodo.

Nivel. - Número de generaciones que se está de la raíz. La raíz está en un nivel = 0.

Ruta. - Camino que se recorre para llegar de un nodo a otro.

Subárbol. - Cualquier nodo puede ser considerado como la raíz de un subárbol.

Peso. - Es la cantidad de nodos hoja del árbol.

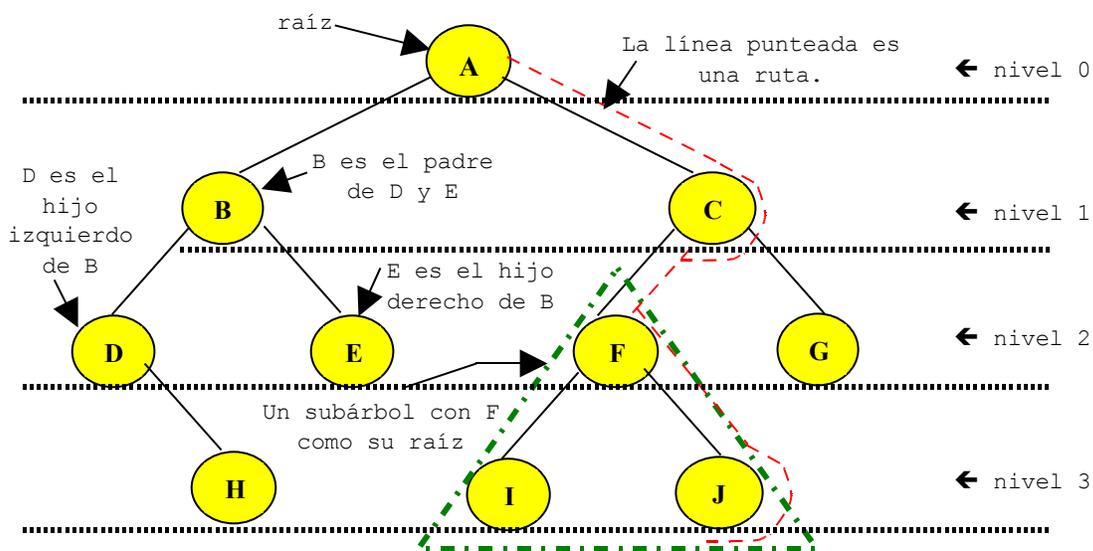
Visita. - Cuando se accede al dato de un nodo.

Recorrido. - Cuando se visita todos los nodos de un árbol en algún orden en especial.

Nodo completo. - Un nodo es completo cuando tiene todos sus hijos o no tiene a ninguno.

Arbol Completo. - Un árbol es completo cuando tiene todos sus nodos completos.

Ejemplo:



H, E, I, J Y G son nodos hoja.

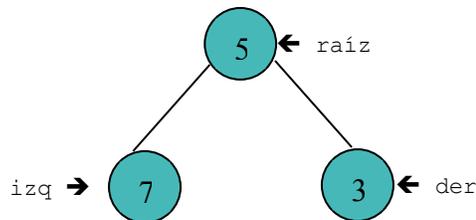
La altura del anterior árbol es 4.

De acuerdo al grado de un árbol, podemos hacer una clasificación en:

- árboles binarios (grado=2) y
- n-arios (grado>2).

Árboles Binarios

Los árboles binarios son un tipo especial de árbol donde cada nodo puede tener a lo mucho 2 hijos (grado = 2). Los dos hijos de cada nodo en un árbol binario son llamados hijo izquierdo e hijo derecho.



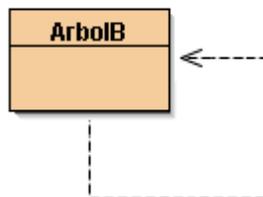
Operaciones

Entre las operaciones permitidas están:

- Verificar si es árbol vacío.
- Insertar, eliminar o localizar un elemento.
- Determinar el tamaño (número de elementos) de la estructura.
- Etc.

Al igual que en listas, vamos a ver dos maneras de definir e implementar a un árbol. La primera, trata de una definición recursiva y la segunda, se basa en el concepto de nodo.

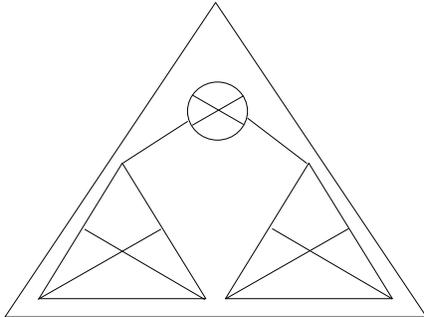
Definición recursiva.



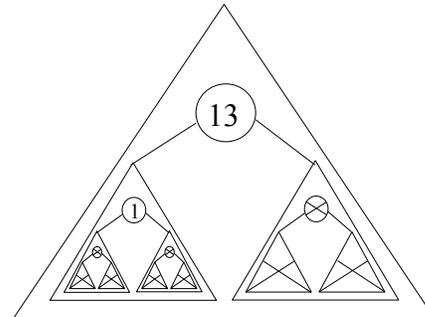
En esta forma de implementación, un árbol puede definirse como un dato (al que llamamos raíz) seguido de un subárbol izquierdo y otro subárbol derecho.

De manera gráfica:

a) Arbol Binario vacío



b) Arbol Binario no vacío



Árbol vacío: Cuando el árbol no tiene raíz, ni izquierdo ni derecho.

Árbol no vacío: Cuando existe una raíz y tiene 2 subárboles (por lo menos vacíos).

El modelo:



La clase ArbolB.java

```
public class ArbolB<E>
{
    private E raiz;
    private ArbolB<E> izq;
    private ArbolB<E> der;

    [...]

}
```

Constructor

```
/**
 * Construye un árbol binario vacío
 */
public ArbolB()
{
    raiz = null;
    izq = null;
    der = null;
}
```

Métodos

Verificar si el árbol está vacío

```
/**
 *
 * @return El resultado de verificar si está vacío
 */
public boolean vacio()
{
    return raiz==null&&izq==null&&der==null;
}
```

Insertar un elemento en el árbol en una ubicación aleatoria

```
/**
 *
 * @param x Es el elemento a insertar
 */
public void insertarAleatorio(E x)
{
    if( vacio() )
    {
        raiz = x;
        izq = new ArbolB<E> ();
        der = new ArbolB<E> ();
    }
    else if(Math.random()<0.5)
        izq.insertarAleatorio(x);
    else
        der.insertarAleatorio(x);
}
```

Obtener el peso del árbol binario

```
/**
 * Calcula el peso del árbol.
 * El peso de un árbol es igual al número de nodos hoja
 * @return el peso del árbol
 */
public int peso()
{
    if( vacio() ) return 0;
    if( izq.vacio() && der.vacio() ) return 1;
    return izq.peso() + der.peso();
}
```

Obtener el tamaño del árbol binario

```
/**
 * Calcula el tamaño del árbol.
 * El tamaño del árbol es la cantidad de nodos que existe
 * @return el tamaño del árbol
 */
public int tamaño()
{
    if( vacio() ) return 0;
    return 1 + izq.tamaño() + der.tamaño();
}
```

Buscar un elemento en el árbol binario

```
/**
 *
 * @param x El elemento a buscar
 * @return El resultado de ver si está
 */
public boolean buscar(E x)
{
    if(vacio()) return false;
    if(raiz.equals(x)) return true;
    return izq.buscar(x) || der.buscar(x);
}
```

Calcular la altura del árbol binario

```
/**
 * Método que calcula la altura de un árbol
 * altura = cantidad de nodos desde la raíz hasta el nodo hoja
 * más alejado
 * @return la altura del árbol
 */
public int altura()
{
    if( vacio() ) return 0;
    return 1 + Math.max( izq.altura(), der.altura() );
}
```

Recorrido de un árbol binario

Recorrer un árbol consiste en acceder una sola vez a todos sus nodos. Esta operación es básica en el tratamiento de árboles y nos permite, por ejemplo, imprimir toda la información almacenada en el árbol, o bien eliminar toda esta información o, si tenemos un árbol de números, sumar todos los valores, etc.

En el caso de los árboles binarios, el recorrido de sus distintos nodos se debe realizar en tres pasos:

- Visitar la raíz.
- Visitar el subárbol izquierdo.
- Visitar el subárbol derecho.

Estas tres acciones repartidas en diferentes órdenes proporcionan los diferentes recorridos del árbol: *preorden*, *enorden* y *postorden*. Su nombre refleja el momento en que se visita el nodo raíz.

Recorrido preorden

- ✓ Visitar la raíz
- ✓ Recorrer el subárbol izquierdo en preorden
- ✓ Recorrer el subárbol derecho en preorden

Recorrido enorden

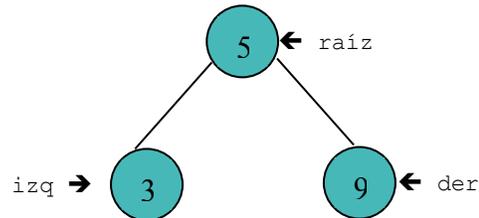
- ✓ Recorrer el subárbol izquierdo en enorden
- ✓ Visitar la raíz
- ✓ Recorrer el subárbol derecho en enorden

Recorrido postorden

- ✓ Recorrer el subárbol izquierdo en postorden
- ✓ Recorrer el subárbol derecho en postorden
- ✓ Visitar la raíz

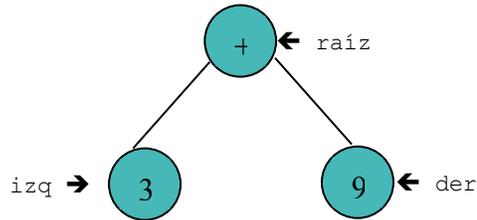
Árboles Binarios de Búsqueda

Son árboles binarios que tienen los datos ordenados de tal manera que todos los datos que están en el subárbol izquierdo son menores que la raíz y todos los datos que están en el subárbol derecho son mayores o iguales que la raíz.



Árboles Binarios de Expresión

Son árboles binarios que almacenan y representan expresiones aritméticas.



NOTA

Para cualquier reclamo, sugerencia o jalón de orejas escríbeme al correo del proyecto ayukin@gmail.com o a mi correo xjkwak@gmail.com.

Esta es una versión borrador y necesito de tu colaboración para poder terminarlo así que no dudes en escribirme (es en serio). Atte. CDMT

Última actualización: 01-01-2009 (Feliz año nuevo!!!)

Bibliografía

1. "Data Structures and Algorithms in Java"
Mitchell Waite y Robert Lafore (1998)
2. "Estructura de Datos y Algoritmos"
Allen Weiss Mark. Addison-Wesley (1995)
3. "Estructuras de Datos y de la Información"
Andrés Muñoz (Universidad de Chile - 2003)
4. "San Google" - <http://www.google.com>