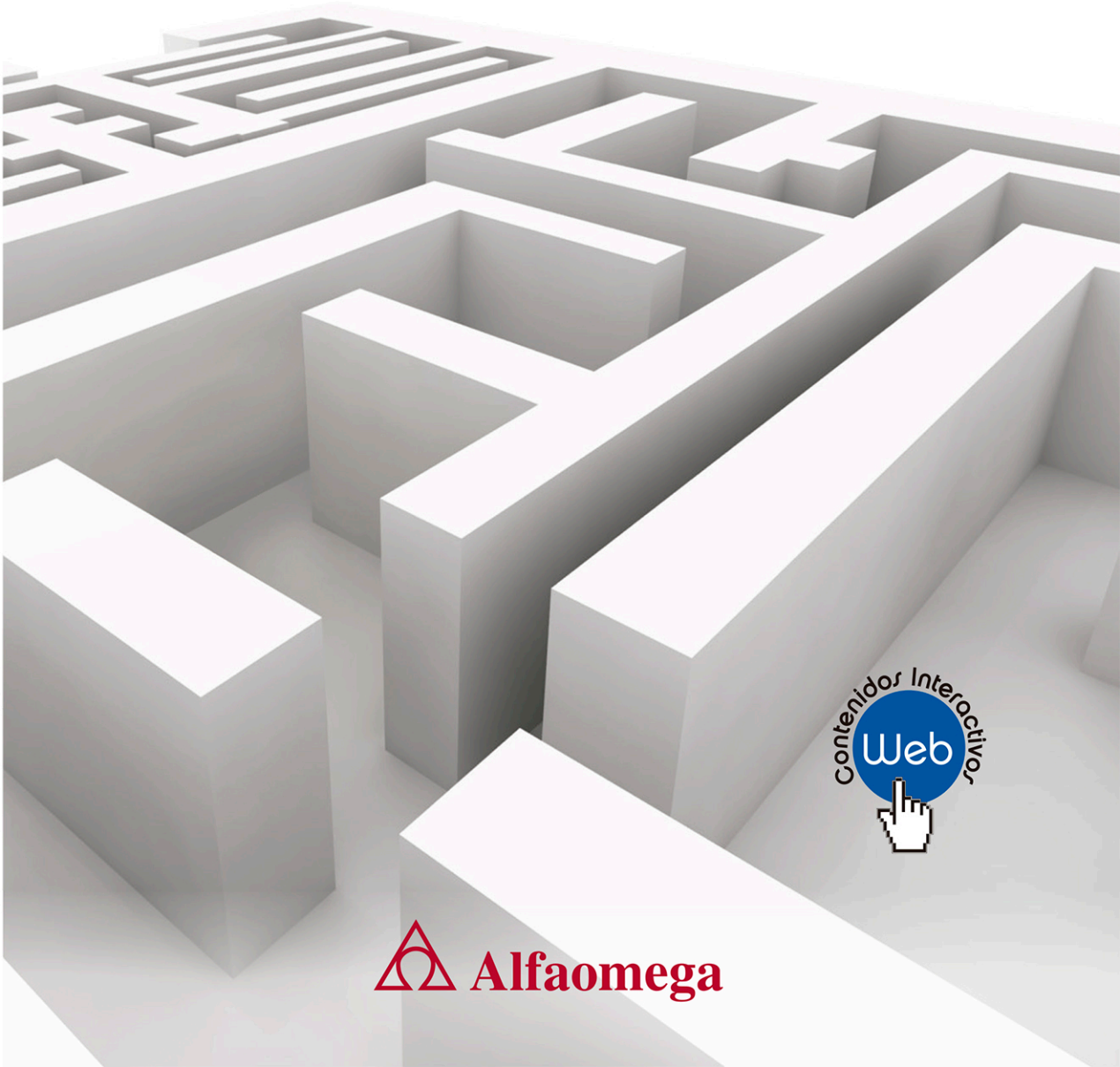
PROGRAMACIÓN ORIENTADA A OBJETOS Y ESTRUCTURA DE DATOS A FONDO

IMPLEMENTACIÓN DE ALGORITMOS EN JAVA

ING. PABLO AUGUSTO SZNAJDLEDER



Alfaomega

Programación orientada a objetos y estructura de datos a fondo

Implementación de algoritmos en Java

Ing. Pablo Augusto Sznajdleder

Alfaomega

Sznajdleder, Pablo Augusto
Programación orientada a objetos y estructura de datos a fondo : Implementación de algoritmos en Java / Pablo Augusto Sznajdleder. - 1a ed . - Ciudad Autónoma de Buenos Aires : Alfaomega Grupo Editor Argentino, 2017.
336 p. ; 16.8 x 23 cm.

ISBN 978-987-3832-27-7

1. Programación. I. Título.
CDD 005.3

Edición: Damián Fernández
Revisión de estilo: Vanesa García
Diagramación: Diego Ay
Revisión de armado: Damián Fernández

Internet: http://www.alfaomega.com.co

Av. Córdoba 1215 Piso “10”, Buenos Aires, Argentina, C.P. 1055.
ISBN **978-958-778-337-7**
Queda hecho el depósito que prevé la ley 11.723

Empresas del grupo:

Argentina: Alfaomega Grupo Editor Argentino, S. A.
Av. Córdoba 1215 Piso “10”, Buenos Aires, Argentina, C.P. 1055
Tel.: (54-11) 4811-7183/0887 / E-mail: ventas@alfaomegaeditor.com.ar

México: Alfaomega Grupo Editor, S. A. de C.V.
Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores. C.P. 06720, Del. Cuauhtemoc. Ciudad de México.
Tel.: (52-55) 5575-5022 - Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396.
E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S. A.
Calle 62 N° 20-46, Bogotá, Colombia
Tel. (57-1)7460102 - Fax: (57-1) 2100122
E-mail: cliente@alfaomegacolombiana.com

Chile: Alfaomega Grupo Editor, S. A.
Av. Providencia 1443, Oficina 24, Santiago de Chile, Chile
Tel.: (56-2) 235-4248/2947-5786 – Fax: (56-2) 235-5786
E-mail: agechile@alfaomega.cl

"El Capitán América"
por Octaviano Sznajdleder

Esta obra no hubiera sido posible sin el apoyo incondicional
de mi esposa Analía y mi hijo Octaviano.

Para ellos está dedicada, con todo el amor de mi corazón.

Quiero agradecer:

a Analía Mora, Hernán Mora, Nahuel Mora y Pablo Bergonzi por el permanente apoyo técnico.

a Damián Fernández por confiar e impulsar mis proyectos editoriales, y por la oportunidad de convertirme en autor.

a Octaviano Sznajdleder por las ilustraciones.

A todos ellos... ¡**Gracias**!

Mensaje del Editor

Los conocimientos son esenciales en el desempeño profesional. Sin ellos es imposible lograr las habilidades para competir laboralmente. La universidad o las instituciones de formación para el trabajo ofrecen la oportunidad de adquirir conocimientos que serán aprovechados más adelante en beneficio propio y de la sociedad. El avance de la ciencia y de la técnica hace necesario actualizar continuamente esos conocimientos. Cuando se toma la decisión de embarcarse en una vida profesional, se adquiere un compromiso de por vida: mantenerse al día en los conocimientos del área u oficio que se ha decidido desempeñar.

Alfaomega tiene por misión ofrecerles a estudiantes y profesionales conocimientos actualizados dentro de lineamientos pedagógicos que faciliten su utilización y permitan desarrollar las competencias requeridas por una profesión determinada. Alfaomega espera ser su compañera profesional en este viaje de por vida por el mundo del conocimiento.

Alfaomega hace uso de los medios impresos tradicionales en combinación con las tecnologías de la información y las comunicaciones (TIC) para facilitar el aprendizaje. Libros como este tienen su complemento en una página Web, en donde el alumno y su profesor encontrarán materiales adicionales, información actualizada, pruebas (test) de autoevaluación, diapositivas y vínculos con otros sitios Web relacionados.

Esta obra contiene numerosos gráficos, cuadros y otros recursos para despertar el interés del estudiante, y facilitarle la comprensión y apropiación del conocimiento.

Cada capítulo se desarrolla con argumentos presentados en forma sencilla y estructurada claramente hacia los objetivos y metas propuestas. Cada capítulo concluye con diversas actividades pedagógicas para asegurar la asimilación del conocimiento y su extensión y actualización futuras.

Los libros de Alfaomega están diseñados para ser utilizados dentro de los procesos de enseñanza-aprendizaje, y pueden ser usados como textos guía en diversos cursos o como apoyo para reforzar el desarrollo profesional.

Alfaomega espera contribuir así a la formación y el desarrollo de profesionales exitosos para beneficio de la sociedad.

Pablo Augusto Sznajdleder
Es Ingeniero en Sistemas de Información egresado de la Universidad Tecnológica Nacional (UTN.BA, 1999).
Es profesor ordinario con categoría de asociado en la cátedra de "Algoritmos y estructura de datos" y, promotor y profesor en la materia optativa "Algoritmos complejos para estructuras avanzadas"; ambas materias en UTN.BA.

Es autor de los libros *JEE a fondo* (Alfaomega, 2015), *Algoritmos a fondo* (Alfaomega 2013), *Java a fondo* (Alfaomega, 2010/12/15) y *HolaMundo.pascal* (CEIT, 2007).

Se desempeñó como instructor Java para Sun Microsystems y Oracle obteniendo, en 1997, las certificaciones SCJP y SCJD; estas fueron las primeras certificaciones Java acreditadas en Argentina y estuvieron entre las primeras logradas en latinoamérica.

Actualmente, se desempeña en el ámbito profesional como consultor e instructor en tecnologías Java/JEE proveyendo servicios de *coaching* y capacitación en las empresas líderes del país.

En el ámbito académico, es candidato del "Programa de Maestría en Ingeniería de Sistemas de Información" de la Universidad Tecnológica Nacional (UTN.BA); e Investigador Tesista del Laboratorio de Investigación, Desarrollo e Innovación en Espacios Virtuales de Trabajo de la Universidad Nacional de Lanús.

www.thejavalistener.com

Revisor técnico: Alberto Templos Carbajal
Es Ingeniero en computación de la Facultad de Ingeniería de la UNAM y ejerce, desde 1983, como Profesor de dicha facultad en las Divisiones de Ingeniería Eléctrica en distintas asignaturas de la carrera de Ingeniería en Computación y Educación Continua. Desde 1986, es Profesor a tiempo completo y ha ocupado los siguientes cargos: Coordinador de las carreras de Ingeniero en Computación y, de manera temporal, de Ingeniero Eléctrico Electrónico e Ingeniero en Telecomunicaciones, Jefe del Departamento de Ingeniería en Computación, Secretario Académico de las Divisiones de Ingeniería Eléctrica y Estudios de Postgrado y Coordinador de Postgrado en la Secretaría de Postgrado e Investigación de la Facultad de Ingeniería. También ha sido cofundador de dos empresas de computación y asesor en esa misma área de diversas compañías.
Actualmente, es miembro de diferentes comités, evaluador de planes de estudio del área de computación para diferentes organismos nacionales y es responsable de un proyecto de investigación e innovación tecnológica sobre el Diseño de algoritmos para robots. Su productividad, principalmente, está orientada a la docencia.

Contenido

Acceso a los materiales de la Web de apoyo

Para tener acceso al material de la plataforma de contenidos interactivos del libro, siga los siguientes pasos:

1. Ir a la página http://libroweb.alfaomega.com.mx
2. Ir a la sección Catálogo y seleccionar la imagen de la portada del libro, al dar doble clic sobre ella, tendrá acceso al material descargable.

Estimado profesor: Si desea acceder a los contenidos exclusivos para docentes por favor contacte al representante de la editorial que lo suele visitar o envíenos un correo electrónico a webmaster@alfaomega.com.mx

Información del contenido de la página Web

El material marcado con asterisco (*) sólo está disponible para docentes.

Capítulo 1

Encapsulamiento a través de clases y objetos

- Autoevaluación
- Presentaciones*

Capítulo 2

Introducción al lenguaje de programación Java

- Autoevaluación
- Videotutorial:
  - Instalar y utilizar Eclipse para Java
- Presentaciones*

Capítulo 3

Programación orientada a objetos

- Autoevaluación
- Videotutorial:
  - Uso del javadoc
- Presentaciones*

Capítulo 4

Estructuras de datos dinámicas lineales

- Autoevaluación
- Presentaciones*

Capítulo 5

Compresión de archivos mediante el algoritmo de Huffman

- Autoevaluación
- Videotutorial:
  - Algoritmo de Huffman
- Presentaciones*

Capítulo 6

Recursividad

- Autoevaluación
- Presentaciones*

Capítulo 7

Árboles

- Autoevaluación
- Presentaciones*

Capítulo 8

Complejidad algorítmica

- Autoevaluación
- Videotutorial:
  - Algoritmo heapsort, ordenamiento por montículos
- Presentaciones*

Capítulo 9

Algoritmos de ordenamiento

- Autoevaluación
- Presentaciones*

Capítulo 10

Estrategia algorítmica

- Autoevaluación
- Videotutorial:
  - Problema de los billetes por programación dinámica
- Presentaciones*

Capítulo 11

Algoritmos sobre grafos

- Autoevaluación
- Videotutoriales:
  - Algoritmo de Dijkstra por greddy
  - Algoritmo de Dijkstra por dinámica
  - Algoritmo de Prim
  - Algoritmo de Kruskal
- Presentaciones*

Más material

Código fuente de cada capítulo

Hipervínculos de interés

Fe de erratas

	Videotutoriales que complementan la obra.
	Conceptos para recordar: bajo este ícono se encuentran definiciones importantes que refuerzan lo explicado en la página.
	Comentarios o información extra: este ícono ayuda a comprender mejor o ampliar el texto principal.

Prólogo

Siempre da gusto leer un libro de programación escrito por un autor que ha aplicado sus conocimientos en el campo laboral, porque no repite simplemente lo que ha recibido de otros. Más bien recrea las palabras y los ejemplos, pues el código le es familiar como una cotidiana compañía. En el caso del Ing. Pablo Augusto Sznajdleder se tiene además un estilo directo y sencillo, que seguramente agradecerán todas aquellas personas a quienes les gustar ir directamente "al grano".

Y ese estilo nos hará llegar de manera concisa hacia a la conceptualización de la programación orientada a objetos (POO), desde los conocimientos del lenguaje C hasta Java (capítulo 1). Ya puestos en ese contexto, hace un recorrido de Java con ejemplos breves pero ilustrativos, haciendo hincapié en los aspectos que son diferentes del lenguaje C (capítulo 2). Pero la POO no hubiera representado un paso tan grande en el campo de la programación si no es por las posibilidades que incorporó. Cuatro de esas características se desglosan en el capítulo 3: clases, herencia (con todo lo que implica), colecciones y excepciones. Las 134 páginas escritas hasta aquí pueden constituir sin ningún problema el material base para un curso de Programación Orientada a Objetos con Java.

A partir del capítulo 4 se comenzaría lo que para muchas escuelas es el temario de Estructura de Datos: listas, pilas, colas, tablas de dispersión y árboles vistos de manera sucinta, que resalta formas óptimas de su implementación. Posteriormente se analiza a detalle el algoritmo de Huffman para la compresión de archivos, tema que suele pasarse por alto en casi todos los libros a pesar de su enorme importancia (capítulo 5).

El capítulo 6 nos introduce al tema de recursividad (una función que se llama a si misma) a través de su conceptualización básica y algunos ejemplos: la comparación entre algoritmos iterativos y recursivos; la permutación de caracteres de una cadena; la búsqueda binaria; el ordenamiento por selección y la función de Fibonacci. Aquí vale la pena hacer un paréntesis: como dijera el autor, "el lector podría pensar que se trata de un tema complicado. Sin embargo… el desarrollo de una función recursiva como factorial [o cualquier otra] resulta extremadamente fácil ya que solo tenemos que ajustarnos a su definición matemática". El potencial de la recursividad y de los diversos temas del libro es muy grande; los ejemplos pueden sonar, como en todo libro de texto, un poco "artificiales". Pero si se entiende el concepto se puede pasar con relativa facilidad a aplicaciones reales.

El texto continúa con otro tema fundamental tratado en el capítulo 7: árboles. En palabras de Arnold, "una estructura no lineal, recursiva, en la que se observa que desde cada nodo descienden uno o varios nodos de su mismo tipo salvo los últimos, que no tienen descendencia". El autor nos habla de árboles binarios y sus diferentes tipos de recorridos, y posteriormente se refiere a los árboles n-arios (que pueden tener varios hijos). Como ejemplo de aplicación real, aparece el tema del AutoSuggest, la función de autocompletado en el campo de búsqueda de Google.

En algunas escuelas, estos tres capítulos son la base de otro curso: Estructura de Datos, tal vez junto con el número 11, que aborda la temática de los grafos. Con los grafos se puede representar "una red de carreteras que unen diferentes ciudades y

analizar, por ejemplo, cuál es el camino más corto para llegar desde una ciudad hasta otra". Google Maps los emplea en todo momento.

Los capítulos 8 a 10 constituyen la base de otro curso dedicado a los Algoritmos Computaciones. Estamos hablando de lo que sucede conforme aumenta el número de datos, concepto crucial para entender fenómenos como la eficiencia de consultas en base de datos o el mundo de Big Data, aunque normalmente se suele aprender tomando como base los algoritmos de ordenamientos. En el capítulo 8 se abordan estos aspectos, reflejados en lo que se llama la notación O grande (la función que refleja el comportamiento del algoritmo según el número de datos), junto con el análisis de la búsqueda lineal y binaria. Se deja para el capítulo 9 la comparación de diferentes algoritmos de ordenamiento: burbuja (Bubble sort), selección (sort), inserción, ordenamiento rápido (Quicksort), por montículos (Heapsort), Shell, por cajas (Binsort) y de raíz (Radix). Finalmente, en el capítulo 10 se expone la estrategia algorítmica, haciendo hincapié en los "algoritmos voraces", que evalúan cada uno de diversos elementos para decidir si incluirlo o no en la solución. Tema independiente dentro de este mismo capítulo es el de programación dinámica.

En resumen, el estudiante tiene en sus manos un buen libro, conciso, directo, cuidado, que puede servirle para dos o tres cursos, según el plan de estudios que tenga. Le dará muchas bases en los temas de algoritmos computacionales, estructuras de datos y complejidad computacional. Esperamos que estos cimientos contribuyan hacia la meta final en esta área de estudios: la construcción de software de gran calidad.

José Luis López Goytia
PRESIDENTE DE LA ACADEMIA DE PROGRAMACIÓN DE UPIICSA
Ciudad de México, octubre de 2016

Introducción

Programación orientada a objetos y estructura de datos a fondo es un libro pensado para extender los conocimientos que fueron adquiridos a lo largo de los cursos de las asignaturas iniciales de programación; principalmente de Programación estructurada.

Justamente, la idea de comenzar explicando Programación Orientada a Objetos (POO) es proveerle al estudiante una herramienta que le permita encapsular la lógica y la complejidad de aquellos algoritmos; ocultándola para no verla. Y, de este modo, poder dedicar su concentración al análisis, diseño y desarrollo de algoritmos con mayor grado de complejidad.

Para aquellos lectores que adquirieron sus conocimientos básicos programando en C, la obra comienza explicando los conceptos principales de encapsulamiento implementándolos con C++. Pero esto es sólo el comienzo. Pues el curso de programación avanzada que aquí se propone transita por los senderos del lenguaje Java.

¿Por qué Java? Porque hoy en día, y desde hace más de 20 años, Java es el lenguaje de programación con mayor nivel de aceptación en el ámbito profesional. La mayoría de las empresas desarrollan sus aplicaciones en Java. Y, aunque aquí el foco principal estará puesto sobre la lógica algorítmica, cuando implemente estos algoritmos con Java, el lector estará adquiriendo una destreza que le permitirá, posteriormente, incorporarse a trabajar en proyectos de desarrollo que utilicen este lenguaje.

Además, el lenguaje de programación Java provee una muy extensa biblioteca de clases. Gracias a esto estaremos exentos de tener que programar nosotros mismos cuestiones que, a esta altura, resultarían triviales y básicas.

Incluso, en temas que sí abordaremos aquí, como: las estructuras lineales (listas, pilas, colas) y métodos de ordenamiento de colecciones, podremos relajarnos y concentrarnos en entender su lógica de programación; pues la implementación ya nos la provee el lenguaje Java.

Así, la obra se divide en dos partes principales: la primera parte abarca la Programación Orientada a Objetos. Todo este paradigma de programación está implementado en Java. Se explica el lenguaje y se compara su sintaxis y semántica con C/C++; de modo que aquellos lectores que tengan conocimientos previos sobre estos lenguajes puedan notar la gran cantidad de similitudes que existen entre ambos y tomar nota sobre las diferencias.

La segunda parte de la obra explica los algoritmos vinculados a las estructuras de datos más complejas: árboles y grafos. Algoritmos recursivos que permitan visitar todos los nodos de un árbol; diferentes tipos de recorridos sobre árboles y comparaciones entre las versiones recursivas e iterativas.

Se explica la técnica de Complejidad Algorítmica cómo un método analítico para comparar la eficiencia de algoritmos equivalentes; y se utiliza esta técnica para clasificar los diferentes enfoques y/o métodos de ordenamiento. Para llegar a deducir, por ejemplo, que el algoritmo Quicksort es mucho más eficiente que el Bubble sort.

Se estudian patrones que delinean diferentes estrategias de diseño algorítmico: Greedy, Divide & conqueer y Programación dinámica.

Finalmente, se estudian los algoritmos que resuelven los problemas típicos sobre grafos: Dijkstra, Prim, Kruskal.

Encapsulamiento a través de clases y objetos

1

Contenido

Objetivos del capítulo

- Conocer la estructura de las clases.
- Encapsular lógica y complejidad mediante el uso clases y objetos.
- Determinar la necesidad y disponer de tipos de datos genéricos.
- Introducir al alumno en el lenguaje de programación Java, previo paso por C++.

1.1 Introducción

La programación estructurada hace énfasis en los procesos. Los módulos o procesos, que se implementan como funciones en C, reciben sus parámetros y realizan la tarea para la cual fueron diseñados en forma adecuada.

Lo anterior suena razonable hasta que nos comenzamos a formular preguntas como esta: ¿Por qué si quiero usar una cadena de caracteres, tengo que preocuparme por agregarle el '\0' al final, asignar memoria para que la contenga y conocer funciones de biblioteca tales como `strcpy`, `strcat`, `strlen`, etcétera?

Justamente, las clases permiten encapsular toda esa complejidad, propia de la implementación, pero totalmente ajena para el programador.

En este capítulo utilizaremos C++ y Java para estudiar la primera de las premisas de la teoría de objetos: el encapsulamiento.

Comenzaremos codificando en C++, pero con un estilo de programación muy particular, casi idéntico al estilo que se utiliza para programar en Java, ya que aquí haremos la transición entre ambos lenguajes.

Cabe aclarar también que los temas que vamos a exponer se tratarán con cierta ligereza. Esto nos permitirá concentrarnos en los conceptos y tener una primera aproximación a la programación orientada a objetos, tema que analizaremos en detalle en el Capítulo 3.

1.2 Clases y objetos

Cuando hablamos de programación orientada a objetos, hablamos de clases y objetos. Para hacerlo rápido y no dar demasiadas vueltas sobre el tema diremos lo siguiente:

Una clase es un tipo de datos definido por el programador, algo así como un `struct` de C. Por lo tanto, llamamos objeto a las variables cuyo tipo de dato es una clase.

Una clase es una especie de estructura que agrupa datos y funciones. Un objeto es una variable cuyo tipo de datos es una clase.

1.2.1 Las clases

Una clase es un tipo de datos definido por el programador. Una especie de estructura en la que, además de definir campos, definimos las funciones a través de las cuales permitiremos manipular los valores de esos campos.

En lugar de hablar de "campos" hablaremos de "variables de instancia" y en lugar de hablar de "funciones" hablaremos de "métodos". Esto solo es una cuestión semántica ya que funciones y métodos, variables de instancia y campos son exactamente lo mismo.

Para entender de qué estamos hablando, veamos el siguiente programa en donde utilizamos la clase `Cadena` (que analizaremos y desarrollaremos más abajo) para concatenar varias cadenas de caracteres.

```
#include <stdio.h>
#include "Cadena.h"

int main()
{
   // definimos un objeto de tipo Cadena inicializado con "Hola,"
   Cadena* s = new Cadena("Hola,");

   // le concatenamos otra cadena
   s->concatenar(" que tal?");


   // le concatenamos otra cadena
   s->concatenar(" Todo bien?");

   // le concatenamos otra cadena
   s->concatenar(" Me alegro mucho!");

   // mostramos su contenido
   printf("%s\n", s->toString());

   // eliminamos el objeto, ya no lo necesitamos
   delete s;

   return 0;
}
```

Como `Cadena` es una clase, entonces `s` (variable de tipo `Cadena`) es un objeto al que le asignamos un valor inicial y luego le concatenamos varias cadenas más. Al final del programa mostramos el resultado en la consola.

Notemos que en ningún momento nos preocupamos por declarar una variable de tipo `char*` ni por asignar un '\0' ni mucho menos por hacer `malloc`. Simplemente nos limitamos a invocar una serie de métodos sobre el objeto `s`.

Evidentemente, toda la complejidad que en C implica inicializar y concatenar cadenas de caracteres está encapsulada dentro de los métodos de la clase `Cadena` cuyo código analizaremos enseguida.

1.2.2 Miembros de la clase

Como comentamos más arriba, una clase se compone de variables de instancia (o campos) y métodos (o funciones). Al conjunto compuesto por las variables de instancia y los métodos de una clase lo llamamos "miembros de la clase".

> Al conjunto compuesto por las variables de instancia y los métodos de una clase lo llamamos "miembros de la clase".

Veamos la estructura de la clase `Cadena`, que se compone de los siguientes miembros:

Variables miembro de `Cadena` = { `cad` }
Métodos miembro de `Cadena` = { `Cadena`, `~Cadena`, `concatenar`, `toString` }

```
class Cadena
{
   // variable de instancia
   private: char* cad;

   // constructor
   public: Cadena(const char* cadInicial)
   {
       // :
       // aqui va el codigo del constructor
       // :
   }

   // destructor
   public: ~Cadena()
   {
       // :
       // aqui va el codigo del destructor
       // :
   }

   // metodo concatenar
   public: void concatenar(const char* cadConcat)
   {
       // :
       // aqui va el codigo del metodo concatenar
       // :
   }

   // metodo toString
   public: char* toString()
   {
       // :
       // aqui va el codigo del metodo toString
       // :
   }
};
```

El código anterior solo refleja la estructura de la clase `Cadena`. Falta la codificación de todos los métodos que, obviamente, analizaremos más adelante.

Sin embargo, esta estructura nos permite apreciar lo siguiente:

- La variable de instancia `cad` está definida como `private`.
- Los métodos están definidos como `public`.

Como explicamos más arriba, los métodos deben ser los únicos responsables de manipular los valores de los campos. Las variables de instancia se suelen definir como miembros privados (`private`) para limitar su accesibilidad y así evitar que puedan ser manipuladas desde afuera de la clase como, por ejemplo, desde el programa principal.

Por el contrario, los métodos se definen como públicos (`public`), lo que nos permite aplicarlos sobre los objetos de la clase y así interactuar.

Decimos entonces que los miembros privados están encapsulados y los miembros públicos están expuestos y constituyen lo que llamaremos la interfaz de los objetos de la clase.

Las variables de instancia se suelen definir como miembros privados para limitar su accesibilidad y así evitar que puedan ser manipuladas desde afuera de la clase. Los miembros públicos están expuestos y constituyen la interfaz de los objetos de la clase.

1.2.3 Interfaz y encapsulamiento

Para comprender mejor los conceptos de interfaz y encapsulamiento, podemos pensar, por ejemplo, en un teléfono celular. El teléfono tiene los botones numéricos a través de los que podemos marcar un número y luego dos botones: uno verde para establecer la llamada y uno rojo para cortar la comunicación.

Con esto, a cualquier usuario común le resulta muy fácil manejar un teléfono celular ya que, usando adecuadamente este conjunto de botones, puede marcar un número y luego establecer la comunicación deseada.

Obviamente, nadie, salvo un estudiante de ingeniería en comunicaciones, se preocupará por entender el proceso que se origina dentro del teléfono luego de que presionamos el botón verde para establecer la llamada. Simplemente, nos abstraemos del tema ya que ese no es nuestro problema.

Decimos entonces que el conjunto de botones numéricos más los botones rojo y verde constituyen la interfaz del objeto teléfono celular. El usuario (nosotros) interactúa con el objeto a través de su interfaz y no debe abrir la carcasa del teléfono para ver ni mucho menos para tocar su implementación (cables, chips, transistores, etc.).

Justamente, en el teléfono celular, los botones están expuestos y los componentes electrónicos son internos y están protegidos (encapsulados) dentro de la carcasa para que nadie los pueda tocar.

Ahora sí podremos ver el código fuente completo de la clase `Cadena` donde podremos observar que dentro de los métodos accedemos a la variable de instancia `cad` como si esta fuese global.

```
#include <stdio.h>
#include <string.h>

class Cadena
{
   // variables de instancia
   private: char* cad;

   // constructor
   public: Cadena(const char* cadInicial)
   {
      cad=(char*)malloc(sizeof(char)*strlen(cadInicial)+1);
      strcpy(cad,cadInicial);
   }
```

```
   // destructor
   public: ~Cadena()
   {
      free(cad);
   }

   // metodos...
   public: Cadena* concatenar(const char* cadConcat)
   {
      int size=strlen(cad)+strlen(cadConcat)+1;
      char* aux=(char*)malloc(sizeof(char)*size);
      strcpy(aux,cad);
      strcat(aux,cadConcat);
      cad=aux;

      return this;
   }

   public: char* toString()
   {
      return cad;
   }
};
```

1.2.4 Estructura de una clase

Analicemos por partes el código de la clase `Cadena` que, siguiendo los lineamientos de este estilo particular de codificación C++, debe estar ubicada en el archivo `Cadena.h`.

Comenzamos definiendo el nombre de la clase y las variables de instancia.

```
#include <stdio.h>
#include <string.h>

class Cadena
{
   // variables de instancia
   private: char* cad;
```

Como se comentó más arriba, los campos (o variables de instancia) son privados ya que no deben ser accedidos desde afuera de la clase. Por este motivo, anteponemos la palabra `private` a la declaración de la variable `cad`. En cambio, los métodos son públicos porque constituyen la interfaz a través de la cual el programador que use la clase podrá interactuar con los objetos.

1.2.5 El constructor y el destructor

Los objetos se construyen, se usan y luego se destruyen. Para construir un objeto, utilizamos el operador `new` mientras que para destruirlo utilizamos el operador `delete`. Estos operadores reciben como parámetro a los métodos “constructor” y “destructor” respectivamente.

El constructor es un método especial a través del cual podemos asignar valores iniciales a las variables de instancia del objeto. Análogamente, el destructor también es un método especial donde podremos liberar los recursos que el objeto haya obtenido.

```
// constructor
public: Cadena(const char* cadInicial)
{
   cad=(char*)malloc(sizeof(char)*strlen(cadInicial)+1);
   strcpy(cad,cadInicial);
}

// destructor
public: ~Cadena()
{
   free(cad);
}
```

Decimos que el constructor es un método especial con las siguientes características:

1. Solo puede invocarse como argumento del operador `new`.
2. El nombre del constructor debe coincidir con el nombre de la clase.
3. El constructor no tiene valor de retorno.

Decimos que el destructor es un método especial con las siguientes características:

1. Solo puede invocarse como argumento del operador `delete`.
2. El nombre del destructor debe comenzar con el carácter "~" (léase carácter "tilde" o "ñuflo") seguido del nombre de la clase.
3. El destructor no tiene valor de retorno.

En el código del constructor, vemos que recibimos un `char*,` que será el valor inicial que tomará la variable de instancia `cad`. Claro que para asignarlo tenemos que dimensionar la cadena con `malloc` y luego copiar carácter a carácter con `strcpy` que, además, asignará un '\0' al final.

Observemos que toda esta complejidad resulta totalmente transparente para el programa principal, donde simplemente definimos el objeto `s` de la siguiente manera:

```
Cadena* s = new Cadena("Hola,");
```

Cuando ya no necesitemos usar la cadena `s,` tenemos que destruirla. Esto lo haremos invocando al operador `delete` pasándole como argumento el objeto que queremos destruir. El operador `delete` invocará al destructor del objeto donde, volviendo al código, vemos que usamos la función `free` para liberar el espacio de memoria direccionado por `cad`.

1.2.6 Los métodos

Los métodos, incluidos el constructor y el destructor, son los únicos responsables de manipular los valores de las variables de instancia.

Notemos que dentro de los métodos tenemos acceso a las variables de instancia como si estas fuesen variables globales.

```
   // metodos...
   public: void concatenar(const char* cadConcat)
   {
      int size=strlen(cad)+strlen(cadConcat)+1;
      char* aux=(char*)malloc(sizeof(char)*size);
      strcpy(aux,cad);
      strcat(aux,cadConcat);
      cad=aux;
   }

   public: char* toString()
   {
      return cad;
   }
};
```

Observemos toda la complejidad que encapsula el método `concatenar`, donde redimensionamos el espacio de memoria direccionado por `cad` para que pueda contener, además, la cadena que se recibe como parámetro.

Gracias a este método, concatenar una cadena en el programa principal resulta ser algo trivial como esto:

```
s->concatenar(" que tal?");
```

Por último, para tener acceso al valor de la variable `cad` y usarla, por ejemplo, en un `printf`, el método `toString` retorna su valor, que es de tipo `char*`.

```
printf("%s\n", s->toString());
```

1.2.7 Los objetos

Como dijimos más arriba, un objeto es una variable cuyo tipo de dato es una clase. La clase define las variables de instancia y los métodos necesarios para manipular sus valores.

Veamos el siguiente código:

```
Cadena* s1 = new Cadena("Pablo");
Cadena* s2 = new Cadena("Juan");
```

Aquí tenemos dos objetos tipo `Cadena`: `s1` y `s2`, cada uno de los cuales tiene su propio valor en la variable `cad`. Es decir: la variable `cad` de `s1` tiene la dirección de la cadena "Pablo" mientras que la variable `cad` de `s2` tiene la dirección de la cadena "Juan".

Decimos entonces que `s1` y `s2` son instancias de la clase `Cadena` ya que cada una mantiene valores propios e independientes en sus variables de instancia.

Por supuesto, también es correcto referirse a `s1` y `s2` como objetos de tipo `Cadena`.

1.2.8 Instanciar objetos

Si bien los términos "objeto" e "instancia" se usan como sinónimos, existe una sutil diferencia entre ambos: un objeto puede no estar instanciado y, por otro lado, una instancia puede no estar siendo referenciada por un objeto.

Veamos: en la siguiente línea de código declaramos un objeto `s` de tipo `Cadena`, pero lo dejamos sin instanciar (no invocamos a su constructor, no invocamos al operador `new`).

```
// declaramos el objeto s sin instanciarlo
Cadena* s;
```

En la siguiente línea de código instanciamos al objeto `s` declarado más arriba.

```
// instanciamos el objeto
s = new Cadena("Hola");
```

Más adelante, veremos casos en los que instanciamos la clase sin tener objetos que apunten a esas instancias.

1.2.9 Operadores `new`, `delete` y punteros a objetos

El operador `new` permite instanciar objetos. Cuando hacemos:

```
Cadena* s = new Cadena("Hola");
```

usamos el operador `new` para asignar memoria dinámicamente y asignamos a `s` la dirección del espacio de memoria recientemente asignada. La variable `s` es en realidad un puntero, por lo que el operador `new` de C++ es comparable a la función `malloc` de C.

Por otro lado, el operador `delete`, luego de invocar al destructor del objeto, libera la memora direccionada por este.

1.2.10 Sobrecarga de métodos

Sobrecargar un método implica escribirlo dos o más veces dentro de la misma clase con diferentes cantidades y/o tipos de parámetros y diferentes implementaciones.

Por ejemplo, podemos sobrecargar el método `concatenar` en la clase `Cadena` para que, además, permita concatenar valores enteros.

```
// permite concatenar una cadena
public: void concatenar(const char* cadConcat)
{
   int size=strlen(cad)+strlen(cadConcat)+1;
   char* aux=(char*)malloc(sizeof(char)*size);
   strcpy(aux,cad);
   strcat(aux,cadConcat);
   cad=aux;
}
```

```
   // permite concatenar un entero
   public: void concatenar(int n)
   {
      // convertimos el entero a cadena
      char sNum[10];
      itoa(n,sNum,10);

      // invocamos a la otra version de concatenar pasandole
      // la cadena que representa al entero
      concatenar(sNum);
   }
```

Ahora, en un programa podemos concatenar tanto cadenas como números enteros.

```
Cadena* s = new Cadena("Estamos en diciembre de ");
s->concatenar(2011);
```

1.3 Encapsulamiento de estructuras lineales

Las clases son una gran herramienta para encapsular la complejidad de ciertas estructuras de datos: pilas, colas y listas.

Aquí analizaremos cómo encapsular la estructura lineal más simple de todas: la pila que, como ya sabemos, se implementa sobre una lista enlazada de nodos y define dos operaciones: `poner y sacar`.

1.3.1 Análisis de la clase `Pila`

Analicemos entonces la clase `Pila` que debe estar contenida en el archivo `Pila.h` cuya estructura, a grandes rasgos, será la siguiente:

```
class Pila
{
   private Nodo* p;

   public: void poner(int v)
   {
      // aqui va la implementacion del metodo
   }

   public: int sacar()
   {
      // aqui va la implementacion del metodo
   }
}
```

Contando con la clase `Pila`, utilizar una pila de enteros en nuestro programa resultará extremadamente simple:

```
#include "Pila.h"

int main()
{
   Pila* pila = new Pila();
   pila->poner(3);
   pila->poner(2);
   pila->poner(1);

   printf("%d\n", pila->sacar());
   printf("%d\n", pila->sacar());
   printf("%d\n", pila->sacar());

   return 0;
}
```

Veamos la implementación de la clase.

```
#include <stdio.h>
#include <stdlib.h>

// estructura Nodo
typedef struct Nodo
{
   int valor;
   struct Nodo* sig;
}Nodo;

// clase Pila
class Pila
{
   private: Nodo* p;

   public: Pila()
   {
      p = NULL;
   }

   public: void poner(int v)
   {
      Nodo* nuevo = (Nodo*) malloc(sizeof(Nodo));
      nuevo->valor = v;
      nuevo->sig = p;
      p = nuevo;
   }

   public: int sacar()
   {
      Nodo* aux = p;
      int ret = aux->valor;
      p = aux->sig;
      free(aux);
      return ret;
   }
};
```

Observemos que el manejo de punteros dentro de los métodos `poner` y `sacar` es mucho más simple que cuando implementamos estas operaciones como funciones sueltas. Esto se debe a que `p` (puntero al primer nodo de la pila) ahora es una variable de instancia a la que los métodos pueden acceder directamente para modificar su valor. Es decir, no tenemos que pasar punteros por referencia.

1.3.2 Templates y generalizaciones

Hasta aquí hemos venido arrastrando un problema que, aun sin haberlo mencionado como tal, más de un lector habrá detectado.

La clase `Pila`, por ejemplo, nos permite apilar elementos de tipo `int`, pero no funciona con valores de otro tipo de datos. Esto se debe a que el tipo de datos del campo `valor` de la estructura `Nodo` es `int`. Está *hardcodeado*.

```
typedef struct Nodo
{
   int valor;
   struct Nodo* sig;
}Nodo;
```

Si pudiésemos hacer que el tipo de datos del campo `valor` fuera variable, entonces la clase `Pila` servirá también para apilar datos de diferentes tipos siempre y cuando especifiquemos el tipo de datos concreto con el que queremos trabajar.

Cuando los tipos de datos con los que trabaja una clase son variables y pueden especificarse como argumentos, decimos que la clase es un *template*.

Cuando los tipos de datos con los que trabaja una clase son variables y pueden especificarse como argumentos, decimos que la clase es un *template*.

Veamos primero cómo quedaría un programa que utilice instancias de una clase `Pila` genérica (un *template*) para apilar enteros, *doubles* y cadenas.

```
#include <stdlib.h>
#include <stdio.h>
#include "Pila.h"
#include "Cadena.h"

int main()
{
   // una pila de enteros
   Pila<int>* pila1 = new Pila<int>();
   pila1->poner(3);
   pila1->poner(2);
   pila1->poner(1);

   printf("%d\n", pila1->sacar());
   printf("%d\n", pila1->sacar());
   printf("%d\n", pila1->sacar());
```