Elementos de programación y Estructuras de Datos

Elaborado por: Helder Octavio Fernández Guzman

helder.fernandez@gmail.com

Si encuentra algún problema o error en el texto, puede consultar la sección 1.5 por posibles soluciones o contactar a helder.fernandez@gmail.com

hecho con LaTeX!

Índice general

Acerca de este Texto

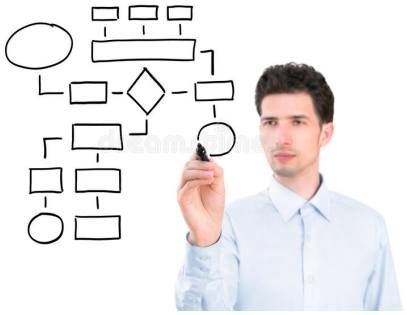
Este texto plantea en sus contenidos técnicas de recursividad, conceptos de administración de la memoria durante la ejecución de un programa, características avanzadas de programación orientada a objetos, y estructuras de datos lineales y no lineales.

La totalidad del contenido de este texto y código fuente son de acceso publico y no tienen fines de lucro, esta pensado para todos aquellos estudiantes que hacen grata esta tarea de ensenar.

CAPITULO 1

Capitulo 1 - Introducción

1. Introducción



¡Bienvenido! este material de apoyo está elaborado de tal forma que se busca que la experiencia del curso sea lo más grata posible. El contenido está organizado en **8** capítulos en el mismo orden en el que se desarrolla el contenido de la materia, pensado principalmente para obtener un aprendizaje muy significativo y que se desarrolla de forma incremental en cuanto al contenido planteado. En general, podemos asumir que todo lo visto en un capítulo previo es necesario para comprender los conceptos del siguiente capítulo.

1.1. ¿Por que este curso?

Un curso introductorio de programación provee las bases para entender cómo programar. Pero, programar de forma más elaborada no solo implica conocer la sintaxis y las bases de la tecnología usada (en este caso JAVA), implica el conocer metodologías de programación pensadas para descomponer y resolver problemas complejos de forma eficiente y correcta.

Comprender detalles avanzados detrás del lenguaje de programación permite representar, organizar y gestionar información compleja de maneras mucho más óptimas y eficientes.

1.2. ¿Por que JAVA?

JAVA es una plataforma informática de programación creada originalmente por Sun Microsystems en 1995[1], sobre esta plataforma que ya pronto llegará a cumplir 30 años y que ha evolucionado bastante durante todo su tiempo de vida, se pueden crear servicios y aplicaciones con los requerimientos de los últimos avances en tecnología de software.

JAVA continúa siendo una de las plataformas más interesantes para programar, pues puede funcionar en diferentes sistemas operativos y hardware que van desde servidores, computadoras personales, dispositivos móviles, micro-ordenadores (como Raspberry) e incluso provee librerías para poder interactuar con plataformas de hardware como Arduino.

Por lo mencionado, JAVA es un recurso muy interesante para desarrollar temas avanzados de programación, me animo a decir que muchos de los conceptos que ha introducido JAVA al

[1]Si bien JAVA originalmente fue tecnología desarrollada por Sun Microsystems, pero esta empresa fue comprada por Oracle en 2010

mundo de la programación han sido replicados o emulados por lenguajes y tecnologías más recientes; entonces, al comprender estos conceptos, los va a poder también asimilar y usar en otros lenguajes de programación.

1.3. ¿Donde obtener JAVA?

Para empezar a programar necesito el software, ¿¿dónde lo obtengo?

Para descargar JAVA: https://www.java.com/es/download/
Para descargar NetBeans (Editor de JAVA): https://netbeans.apache.org/download/index.html
Para descargar Eclipse (otro editor de JAVA): https://www.eclipse.org/downloads/packages/

1.4. Organización del documento

1. **Introducción** En el capítulo 1, se plantean y justifican las razones de la materia y los contenidos planteados, referencias a recursos necesarios en la materia: lenguaje, editores y el repositorio de código fuente usado en el texto.

2. **Conceptos Fundamentales.** En el capítulo 2, se realiza un repaso de alto nivel de conceptos vistos en la materia de Introducción a la programación, pero que son muy relevantes para asimilar los nuevos conceptos propuestos en este texto y materia.

3. **Complejidad Computacional.** En el capítulo 3, se plantean conceptos relacionados a tiempos de ejecución y los espacios de memoria usados durante la ejecución de un programa, para entender cómo hacer un uso óptimo de los mismos. Se abarcan conceptos como complejidad temporal, administración de la memoria, recolector de basura, palabra reservada *static, this y super*, variables, referencias y tipos de operaciones.

4. **Elementos de Programación.** En el capítulo 4, se plantea la recursividad como una alternativa algorítmica que sirve para resolver un conjunto de problemas de manera muy abstracta. En un nivel más avanzado se estudia la recursividad de ejecución no lineal para trabajar en permutaciones y combinaciones de manera más eficiente (usando funciones de poda) mecanismo por el cual la técnica obtiene su nombre: *Backtracking*.

5. **Programación Orientada a Objetos.** En el capítulo 5, se revisan conceptos clave de JAVA: herencia y polimorfismo desde un enfoque de su aplicación y uso en la industria del software.

6. **Programación Genérica.** En el capítulo 6, se revisan mecanismos de reutilización mediante la generalización en JAVA, los cuales están basados en la parametrización de tipos de datos.

7. **Estructuras de Datos.** En el capítulo 7, se describen los conceptos de Estructuras de datos lineales. Y se proveen implementaciones base en JAVA usando referencias.

8. **Estructuras de Datos No lineales.** En el capítulo 8, se describen los conceptos de Estructuras de datos No Lineales. Y se proveen implementaciones base en JAVA usando referencias.

1.5. Errores en el Texto y nuevos aportes

Si crees que algo debe ser corregido en este texto, quieres participar en la edición de las nuevas versiones del mismo o apoyar en la generación de código para los Laboratorios Virtuales, puedes contactarme al correo: helder.fernandez@gmail.com. Toda retroalimentación y aporte nuevo al contenido de este texto son bienvenidos, serás mencionado en la siguiente versión del texto.

2

CAPITULO

Capitulo 2 - Conceptos Fundamentales

2. Conceptos fundamentales

En este capítulo, a manera de repaso, vamos a presentar algunos conceptos que se estudian durante el curso introductorio a programación. Hacemos explícito el contenido para no dar por hecho que los conceptos planteados son de dominio del estudiante programador.

2.1. Sistema de Tipos en JAVA

Podemos ver al Sistema de Tipos como un mecanismo para representar la información a ser manipulada o procesada en el lenguaje de programación, una de las razones: permite al compilador validar la corrección del programa a nivel sintáctico, pues al ser JAVA un lenguaje fuertemente tipado, las transacciones o transferencias de datos e información entre llamadas a funciones y operaciones de asignación y lectura deben ser entre variables y valores del mismo tipo.

La otra razón a saber es un tema técnico, el programa va a procesar la información de una u otra manera, entonces el poder identificar de antemano el tipo de dato permite representarlo o almacenarlo de la forma más eficiente de acuerdo al tipo de cálculo que se hará con esta información.

En JAVA vamos a identificar 2 grupos de Tipos de Datos, los **primitivos** y los **complejos**, a continuación, desarrollamos cada uno de ellos.

2.1.1. Tipos Primitivos

JAVA cuenta con ocho tipos primitivos, que cubren todo tipo de números (reales o decimales y enteros) cada uno con una extensión o magnitud máxima, también cubre los valores lógicos (falso y verdadero) e incluye caracteres. En la mayoría de los casos, los datos de tipo primitivo son gestionados y existen en el **Stack** durante la ejecución del programa. Esto, técnicamente hablando, significa que tienen siempre tamaño fijo y existen mientras una función existe y es evaluada.

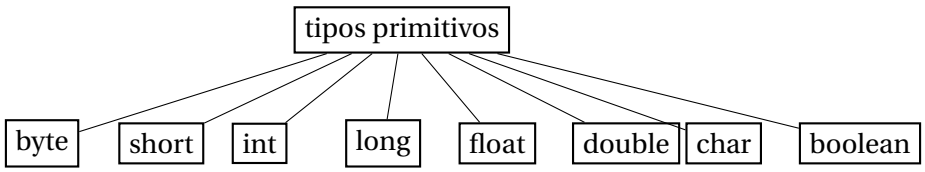


Figura 2.1: tipos primitivos

2.1.2. Tipos Complejos u Objetos

Son datos que son gestionados de forma diferente a los datos de tipo primitivo, dada su naturaleza se localizan en el **HEAP** durante la ejecución del programa y la manera de gestionarlos es a través de su dirección de memoria. Los tipos Objeto se pueden agrupar en varios conjuntos de acuerdo al patrón de comportamiento que tienen o a su uso.

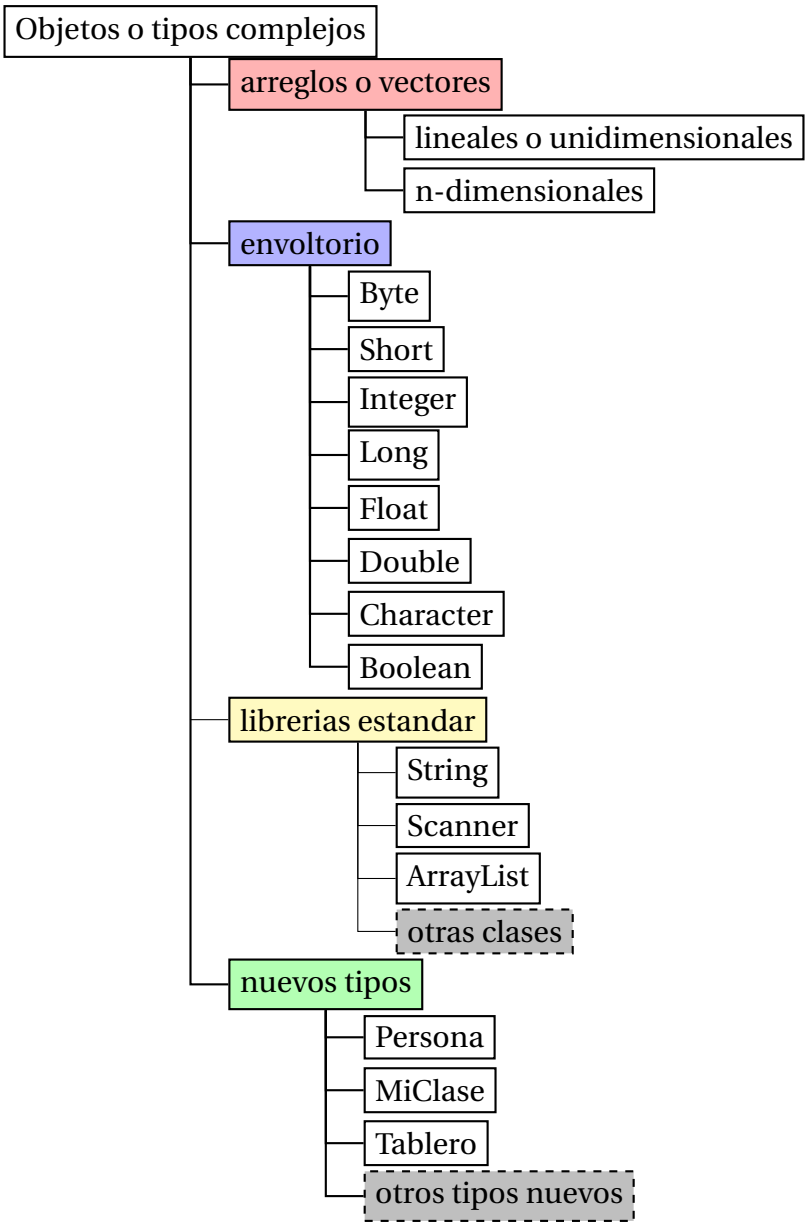


Figura 2.2: tipos no primitivos

2.2. Arreglos en JAVA

Un arreglo o vector en JAVA es una estructura de datos que permite almacenar una colección de elementos del mismo tipo de manera consecutiva en memoria. Cada elemento en el arreglo tiene un índice que lo identifica.

Los arreglos son almacenados en la memoria de forma consecutiva, lo cual significa que todos los elementos del arreglo son almacenados juntos en una ubicación contigua de la memoria.

Cada elemento en un arreglo tiene un índice que se utiliza para acceder a su posición en la memoria. El índice del primer elemento es siempre 0, y el índice del último elemento es siempre el tamaño del arreglo menos 1.

Cuando se declara un arreglo en JAVA, se reserva suficiente espacio en la memoria para contener todos los elementos del arreglo. El tamaño del arreglo se especifica al crear el arreglo, y una vez que se ha reservado el espacio en la memoria, no se puede cambiar.

Cada elemento en el arreglo es un lugar de memoria separado, y para acceder a un elemento en particular, se utiliza su índice. Cuando se accede a un elemento en el arreglo, JAVA utiliza la aritmética de punteros para calcular la ubicación en memoria del elemento.

Tener en cuenta que si se intenta acceder a un elemento fuera de los límites del arreglo (un elemento que no existe), JAVA lanza una excepción de índice fuera de los límites: **ArrayIndexOutOfBoundsException**. Por lo tanto, es importante asegurarse en el programa de que se esté accediendo a los elementos del arreglo dentro de sus límites válidos.

A continuación detallamos las operaciones más frecuentes realizadas con arreglos:

2.2.1. Declaración de un arreglo en JAVA

Para declarar un arreglo en JAVA, se utiliza la siguiente sintaxis:

```
tipo[] nombreArreglo;
```

Por ejemplo, para declarar un arreglo de enteros, se usa:

```
int[] numeros;
```

2.2.2. Creación de un arreglo en JAVA

Para crear un arreglo en JAVA, se utiliza la siguiente sintaxis:

```
nombreArreglo = new tipo[tamano];
```

Por ejemplo, para crear un arreglo de enteros con 5 elementos, se usa:

```
numeros = new int[5];
```

2.2.3. Inicialización de un arreglo en JAVA

Para inicializar un arreglo en JAVA, puede hacerse en la misma línea de creación o utilizando un bucle **for**, por ejemplo:

```
// Inicializacion en la misma linea de creacion
int[] numeros = new int[] {1, 2, 3, 4, 5};

// Inicializacion utilizando un bucle for
int[] numeros = new int[5];
for (int i = 0; i < 5; i++) {
    numeros[i] = i + 1;
}
```

2.2.4. Acceso a elementos de un arreglo en JAVA

Para acceder a un elemento en particular de un arreglo en JAVA, se utiliza su índice entre corchetes. Por ejemplo:

```
int[] numeros = new int][]{1, 2, 3, 4, 5};
System.out.println(numeros[0]); // Imprime el primer elemento (1)
System.out.println(numeros[4]); // Imprime el ultimo elemento (5)
```

2.2.5. Longitud de un arreglo en JAVA

Para obtener la longitud de un arreglo en JAVA, se utiliza la propiedad **length**. Por ejemplo:

```
int[] numeros = new int[]{1, 2, 3, 4, 5};
System.out.println(numeros.length); // Imprime 5
```

2.3. Tipos Envoltorio

Los tipos envoltorio son clases que encapsulan tipos primitivos, como int, float, double, etc., como objetos. Las clases envoltorio se utilizan para proporcionar un conjunto de métodos de utilidad para trabajar con tipos primitivos como objetos.

Los tipos envoltorio se utilizan comúnmente en JAVA para trabajar con colecciones de objetos, como listas y mapas. También se utilizan para trabajar con bibliotecas y marcos de trabajo que requieren objetos en lugar de valores primitivos. Los tipos envoltorio también pueden ser útiles en situaciones donde se requiere **nulabilidad** , ya que **los tipos primitivos no pueden ser nulos**.

Hay ocho tipos de envoltorio en JAVA:

- Byte
- Short
- Integer
- Long
- Float

- Double
- Character
- Boolean

Cada uno de estos tipos envoltorio tiene un nombre de clase correspondiente.

2.3.1. Autoboxing y unboxing

JAVA admite autoboxing y unboxing, que son técnicas para convertir automáticamente entre tipos primitivos y sus correspondientes tipos envoltorio.

Autoboxing es la conversión automática de un tipo primitivo en su tipo envoltorio equivalente.

Unboxing es la conversión automática de un objeto envoltorio en su tipo primitivo equivalente.

Por ejemplo, con autoboxing, se puede hacer lo siguiente:

```
1 int i = 10;
2 Integer integer = i; // Autoboxing
```

Con unboxing, se puede hacer lo siguiente:

```
1 Integer integer = 10;
2 int i = integer; // Unboxing
```

2.3.2. Algunos métodos útiles de los tipos envoltorio

Cada tipo de envoltorio tiene un conjunto de métodos de utilidad para trabajar con valores primitivos y objetos de envoltorio. Estos métodos incluyen:

- *valueOf()*: para crear un objeto envoltorio a partir de un valor primitivo.
- *toString()*: para convertir un objeto envoltorio a una cadena de caracteres.
- *compareTo()*: para comparar dos objetos envoltorio.
- *equals()*: para determinar si dos objetos envoltorio son iguales.
- *parseInt()*: para convertir una cadena de caracteres en un valor entero.
- *doubleValue()*: para obtener el valor primitivo de un objeto envoltorio.

2.4. Algunas Clases útiles de JAVA

En esta sección se describen algunos de los métodos más interesantes de 4 clases que pueden ser muy útiles a la hora de construir tus programas; para un análisis detallado se recomienda visitar la Documentación[1] de JAVA, donde encontrarás más librerías, así como todos sus métodos y funciones.

[1] https://docs.oracle.com/javase/8/docs/

2.4.1. Clase Scanner

Esta clase se usa, entre otras cosas, para que el usuario pueda introducir datos por el teclado. Para usarla hay que importarla desde la librería **java.util**. A continuación se muestra un ejemplo de código con su uso (se omite la declaración de la clase y del método main):

```
1 import java.util.Scanner;
2 ...
3 Scanner s = new Scanner(System.in);
4 int b=0;
5 while (!s.hasNextInt())
6 s.next();
7 b= s.nextInt();
8 System.out.println(b);
```

Por defecto, los datos se separan mediante un espacio. Se puede cambiar el carácter separador, por ejemplo, para cambiarlo por **Enter** usamos el método:

```
1 s.useDelimiter(System.getProperty("line.separator"));
```

Se pueden usar otros delimitadores e incluso combinaciones de ellos por medio de expresiones regulares. También por defecto, si el teclado del ordenador está en español, se usa la coma en lugar del punto para los números decimales. Si queremos que el usuario introduzca los datos usando el punto, utilizamos (hay que importar) la clase **Locale** de **java.util**:

```
1 s.useLocale(Locale.ENGLISH);
```

Algunos métodos interesantes de esta clase son:

- *int nextInt()*: Devuelve un valor de tipo int que el usuario debe introducir por teclado. Hay un método para cada tipo primitivo, excepto para *char*. Ej: *nextDouble()*, *nextBoolean()* ...
- *String next()* Devuelve el siguiente dato introducido por el usuario en formato de String.
- *String nextLine()* Devuelve todo lo que ha introducido el usuario, independientemente de cual sea el separador.
- *boolean hasNext()* Devuelve true si hay algún dato listo para ser leído.
- *boolean hasNextInt()* Devuelve true si lo siguiente que va a leer es un valor de tipo **int** (si lo siguiente que ha introducido el usuario es un int). Existe un método similar para cada tipo primitivo, excepto **char**. Ej. *hasNextDouble()*, *hasNextBoolean()* ...

2.4.2. Clase String

String, además de comportarse como un tipo de dato, es una clase, por lo que tiene métodos que se pueden utilizar para hacer operaciones con cadenas. Para llamar a los métodos se pone <variable de tipo String>.método.

- *char charAt(int index)* Devuelve el carácter que esta en la posición index.
- *int compareTo(String anotherString)* Devuelve 0 si ambas cadenas son iguales, un valor negativo si la cadena es anterior alfabéticamente que el argumento y un valor positivo si es mayor. Si son diferentes devuelve la diferencia de código ASCII entre las dos primeras letras en que se diferencian. Si solo se diferencian en que son de distinta longitud lo que devuelve es la diferencia en la longitud.
- *int compareToIgnoreCase(String str)* Igual al anterior ignorando la diferencia entre mayúsculas y minúsculas.
- *boolean contains(charSequence s)* Devuelve true si la cadena contiene a la subcadena.
- *boolean endsWith(String suffix)* Devuelve true si la cadena acaba de esa forma. Hay otro equivalente si la cadena empieza de esa forma (startsWith).
- *boolean equals (String str)* boolean equalsIgnoreCase(String str) Devuelve verdadero si las dos cadenas son iguales, en el segundo caso ignorando mayúsculas y minúsculas.
- *int indexOf(String str), int indexOf(String str, int ind)* Devuelve un entero con la posición en la que aparece el carácter o la subcadena por primera vez (-1 si no existe). La segunda versión empieza a buscar desde un lugar determinado. También hay 2 versiones equivalentes buscando de atrás hacia delante (lastIndexOf)
- *int length()* Devuelve la longitud de la cadena.
- *String replace (String , String)* Reemplaza todas las ocurrencias de una subcadena por otra. También se puede usar *replaceAll* o *replaceFirst*. La primera se comporta igual pero ademas de recibir una cadena puede recibir una expresión regular. La segunda, también puede recibir expresiones regulares, y solo cambia la primera ocurrencia de la subcadena.
- *String[ ] split(String regex)* Devuelve un arreglo de String resultado de partir la cadena usando como separador el argumento (cadena o expresión regular).

2.4.3. Clase Math

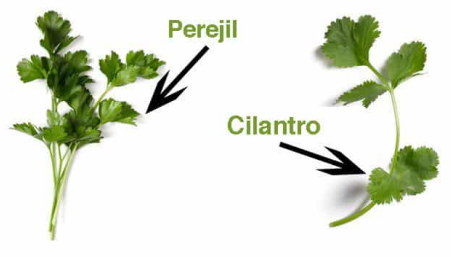
Clase especial que contiene funciones y constantes matemáticas. Se usa indicando el nombre de la clase y llamando al método específico: *Math.<método>*, o indicando el nombre de la clase e indicando el atributo: *Math.E* y *Math.PI*.

Entre los métodos y atributos podemos encontrar los siguientes:

- *static int abs(int a)* Devuelve el valor absoluto del numero pasado como parámetro. También se puede usar con cualquier otro tipo numérico (si le damos un double devolverá un double, etc.)
- *static long round(double a)* Redondea el numero pasado como parámetro. Si el numero es double devuelve long, si es float devuelve int.

- *static double ceil(double a)* Trunca el numero hacia arriba (Math.ceil(3.2) devuelve 4.0).
- *static double floor(double a)* Trunca el numero.
- *static double sin(double a)* Devuelve el seno del angulo a (a debe estar en radianes). También: *cos, tan, asin, acos, atan, sinh, cosh, tanh.*
- *static int max (int a, int b)* Devuelve el máximo de los dos números. También hay versiones para los otros tipos numéricos. También existe min (int a, int b).
- *static double log(double a)* Devuelve el logaritmo neperiano de a, para el logaritmo decimal se usa log10(double a).
- *static double pow(double a, double b)* Eleva a a b.
- *static double exp(double a)* Eleva el numero **e** a a.
- *static double sqrt(double a)* Raíz cuadrada de a.
- *static double cbrt(double a)* Raíz cubica de a.
- *static double random()* Devuelve un numero aleatorio entre 0.0 (incluido) y 1.0 (no incluido)

2.5. Funciones, métodos y procedimientos



En Java, el término **método** se utiliza para referirse a funciones o procedimientos que están asociados con una clase o un objeto. Sin embargo, cuando hablamos de **método estático** y **función**, es importante aclarar que en Java no existe el concepto de **función** como tal, ya que todo en Java está definido dentro de una clase. A continuación, describimos las diferencias clave entre un método estático, un método de instancia y un procedimiento:

2.5.1. Método estático

Un método estático es un método que pertenece a la clase en lugar de a una instancia específica de la clase. Se declara usando la palabra clave static. Puede ser invocado directamente usando el nombre de la clase, sin necesidad de crear un objeto de la clase.

Un método estático solo puede acceder a otros miembros estáticos de la clase (variables estáticas o métodos estáticos). No puede acceder a variables de instancia o métodos de instancia directamente, ya que no tiene una referencia a un objeto específico (this no está disponible). Se utiliza para operaciones que no dependen del estado de un objeto específico, como utilidades o funciones matemáticas.

```
1 public class Matemáticas {
2     public static int sumar(int a, int b) {
3         return a + b;
```

```
    }
}

// Uso del método estático:
int resultado = Matemáticas.sumar(5, 3);
// no se necesita crear una instancia de la clase
```

2.5.2. Método de instancia (no estático)

Un método de instancia es un método que pertenece a un objeto específico de una clase. No se declara con static. Para invocarlo, primero se debe crear una instancia de la clase (un objeto) y luego llamar al método usando esa instancia.

Un método de instancia puede acceder tanto a variables de instancia como a métodos de instancia, ya que opera en el contexto de un objeto específico. También puede acceder a miembros estáticos. Se utiliza para operaciones que dependen del estado de un objeto específico.

```
public class Coche {
    private int velocidad;

    public void acelerar(int incremento) {
        velocidad += incremento;
    }

    public int getVelocidad() {
        return velocidad;
    }
}

// Uso del método de instancia
Coche miCoche = new Coche();
miCoche.acelerar(20); // se llamar al método a partir
                      // de la instancia
System.out.println(miCoche.getVelocidad()); // Imprime 20
```

2.5.3. Métodos con retorno void (procedimientos)

Un método con retorno **void** es un método que no devuelve ningún valor. Su propósito es ejecutar una acción o una serie de acciones sin necesidad de producir un resultado que deba ser capturado o utilizado.Se utilizan para realizar tareas como modificar el estado de un objeto, imprimir mensajes, o ejecutar operaciones que no requieren devolver un valor.
Se declara usando la palabra clave void en lugar de un tipo de dato.

```
public class Impresora {
    // Método void que no retorna ningún valor
    public void imprimirMensaje(String mensaje) {
        System.out.println("Mensaje: " + mensaje);
    }
}

```

```
// Uso del método void
Impresora impresora = new Impresora();
impresora.imprimirMensaje("Hola, mundo!");
// No se espera un valor de retorno
```

2.5.4. Ejemplo combinado

```
public class CuentaBancaria {
    private double saldo;
    // Constructor
    public CuentaBancaria(double saldoInicial) {
        this.saldo = saldoInicial;
    }
    // Método de instancia void para depositar dinero
    public void depositar(double monto) {
        if (monto > 0) {
            saldo += monto;
            System.out.println("Depósito exitoso.");
            System.out.println("Nuevo saldo: " + saldo);
        } else {
            System.out.println("Monto inválido.");
        }
    }
    // Método de instancia void para retirar dinero
    public void retirar(double monto) {
        if (monto > 0 && monto <= saldo) {
            saldo -= monto;
            System.out.println("Retiro exitoso.");
            System.out.println("Nuevo saldo: " + saldo);
        } else {
            System.out.println("Fondos insuficientes");
            System.out.println("o monto inválido.");
        }
    }
    // Método de instancia con retorno para obtener el saldo actual
    public double obtenerSaldo() {
        return saldo;
    }
    // Método estático para calcular el interés
    public static double calcularInteres(double saldo, double tasa) {
        return saldo * (tasa / 100);
    }
    // Método estático void para mostrar información general
    public static void mostrarInformacion() {
        System.out.println("Bienvenido al sistema de");
        System.out.println("cuentas bancarias.");
        System.out.println("Tasa de interés anual: 5%");
    }
}

public class Main {
```

```
45      public static void main(String[] args) {
46          // Uso de método estático void
47          CuentaBancaria.mostrarInformacion(); // No necesita instancia
48          // Crear una cuenta bancaria
49          CuentaBancaria cuenta = new CuentaBancaria(1000);
50          // Uso de métodos de instancia void
51          cuenta.depositar(500); // solo ejecuta una acción
52          cuenta.retirar(200);   // solo ejecuta una acción
53          // Uso de método de instancia con retorno
54          double saldoActual = cuenta.obtenerSaldo();
55          System.out.println("Saldo actual: " + saldoActual);
56          // Uso de método estático con retorno
57          double interes = CuentaBancaria.calcularInteres(saldoActual, 5);
58          System.out.println("Interés generado: " + interes);
59      }
60  }
```

En resumen, en Java no existen **funciones** independientes, solo métodos estáticos y de instancia. Los métodos estáticos pertenecen a la clase y no dependen de un objeto, mientras que los métodos de instancia operan en el contexto de un objeto específico. Los métodos void son aquellos que no devuelven ningún valor y se utilizan para ejecutar acciones. Por otro lado, los métodos con retorno devuelven un valor específico y se utilizan cuando necesitas obtener un resultado. Ambos tipos de métodos son fundamentales en la programación orientada a objetos y se complementan para crear programas modulares y eficientes.

Los modificadores que son parte de la firma pueden incluir public, private, protected, static, final, abstract, synchronized, native, entre otros, y justamente permiten identificar de cuál de las definiciones estamos hablando. La lista de parámetros incluye el tipo y nombre de cada parámetro separados por comas.

CAPITULO 3

Capitulo 3 - Complejidad Computacional

3. Complejidad Computacional, eficiencia y uso de recursos

En el diseño y análisis de algoritmos, la **complejidad computacional** juega un papel fundamental para evaluar su eficiencia y el uso de recursos. La eficiencia de un algoritmo se mide principalmente en términos de su **tiempo de ejecución** y el **uso o consumo de memoria**, dos aspectos críticos que determinan su escalabilidad y aplicabilidad en problemas de gran escala.

Mientras que el tiempo de ejecución se refiere al número de operaciones que un algoritmo realiza en función del tamaño de la entrada, la administración de la memoria se enfoca en la cantidad de espacio que el algoritmo requiere para procesar dicha entrada. Ambos factores están intrínsecamente relacionados: un algoritmo eficiente no solo debe ser rápido, sino que también debe gestionar de manera óptima los recursos de memoria disponibles. En este contexto, la notación **Big-O** se utiliza para describir el comportamiento asintótico de un algoritmo, permitiendo comparar su rendimiento en escenarios donde el tamaño de la entrada tiende a infinito. Comprender estos conceptos es esencial para diseñar soluciones que no solo resuelvan problemas de manera correcta, sino que también lo hagan de manera eficiente y sostenible.

La complejidad computacional es una medida de los recursos que un algoritmo o programa consume, principalmente en términos de tiempo (cuántas operaciones realiza) o espacio (cuánta memoria utiliza). El objetivo es entender cómo es el rendimiento escalando con el tamaño de la entrada.

3.1. Complejidad Temporal

En forma breve, la **Complejidad Temporal** se refiere al tiempo que tarda un algoritmo en ejecutarse, en función del tamaño de la entrada.

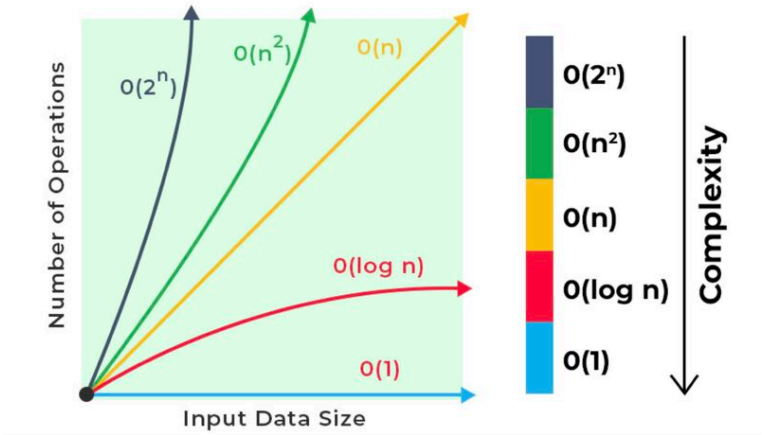


Figura 3.1: Big-O representación visual

3.1.1. Notación Big-O

La notación Big-O es la forma más común de expresar la complejidad de un algoritmo. Describe el comportamiento asintótico de un algoritmo, es decir, cómo crece el tiempo de ejecución o el uso de memoria cuando el tamaño de la entrada (n) tiende a infinito.
Ejemplos de órdenes de complejidad en JAVA:

- **Complejidad constante**:

$$\mathcal{O}(1)$$

Ejemplo: Acceder a un elemento de un arreglo. No importa cuán grande sea el arreglo, siempre se accede a un elemento en una sola operación.

```java
public int getFirstElement(int[] arreglo) {
    // Siempre toma una operación,
    // independientemente del tamaño del arreglo.
    return arreglo[0];
}
```

- **Complejidad logarítmica**:

$$\mathcal{O}(\log n)$$

Ejemplo: Búsqueda binaria en un arreglo ordenado. En cada iteración, el algoritmo divide el espacio de búsqueda a la mitad. Si el arreglo tiene n elementos, el número máximo de iteraciones es $log_2 n$.

```java
public int binarySearch(int[] arreglo, int target) {
    int left = 0;
    int right = arreglo.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arreglo[mid] == target) {
            return mid;
        } else if (arreglo[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}
```

- **Complejidad lineal**:

$$\mathcal{O}(n)$$

Ejemplo: Recorrer todos los elementos de un arreglo. El iterador for recorre todos los elementos del arreglo una vez. Si el arreglo tiene n elementos, entonces se realizan n operaciones.

```java
public int sumArray(int[] arreglo) {
    int sum = 0;
    // Se recorre el arreglo una vez.
```

```
    for (int i = 0; i < arreglo.length; i++) {
        sum += arreglo[i];
    }
    return sum;
}
```

- **Complejidad cuadrática**:

$$\mathcal{O}(n^2)$$

Ejemplo: Algoritmos de ordenamiento ineficientes como Bubble Sort o Insertion Sort. Para cada elemento del arreglo (iterador externo), se recorre nuevamente todo el arreglo (iterador interno). Si el arreglo tiene n elementos, se realizan $n \times n = n^2$ operaciones.

```
public void printPairs(int[] arreglo) {
    // iterador externo.
    for (int i = 0; i < arreglo.length; i++) {
        // iterador interno.
        for (int j = 0; j < arreglo.length; j++) {
            System.out.println(arreglo[i] + ", " + arreglo[j]);
        }
    }
}
```

- **Complejidad cúbica**:

$$\mathcal{O}(n^3)$$

Ejemplo: Multiplicación de matrices utilizando el método clásico. El código tiene tres bucles anidados, cada uno de los cuales se ejecuta n veces. La complejidad total es $n \times nxn = n^3$, por lo que es $O(n^3)$.

```
public void cubicComplexity(int n) {
    for (int i = 0; i < n; i++) { // O(n)
        for (int j = 0; j < n; j++) { // O(n)
            for (int k = 0; k < n; k++) { // O(n)
                System.out.println(i + ", " + j + ", " + k);
            }
        }
    }
}
```

- **Complejidad exponencial**:

$$\mathcal{O}(2^n)$$

Ejemplo: Calcular el n-ésimo número de Fibonacci de manera recursiva. Cada llamada genera dos nuevas llamadas, lo que resulta en un árbol de recursión con 2^n nodos. La complejidad es $O(2^n)$.

```
public int exponentialComplexity(int n) {
    if (n <= 1) {
        return n;
    }
    // Llamadas recursivas
```

```
    return exponentialComplexity(n - 1)
           + exponentialComplexity(n - 2);
}
```

- **Complejidad factorial**:

$$\mathcal{O}(n!)$$

Ejemplo: Este código genera todas las permutaciones de un arreglo. El número de permutaciones de n elementos es $n!$, por lo que la complejidad es $O(n!)$.

```
public void factorialComplexity(int[] arreglo, int start) {
    if (start == arreglo.length - 1) {
        System.out.println(Arrays.toString(arreglo));
    } else {
        for (int i = start; i < arreglo.length; i++) {
            // Intercambiar elementos
            int temp = arreglo[start];
            arreglo[start] = arreglo[i];
            arreglo[i] = temp;

            // Llamada recursiva
            factorialComplexity(arreglo, start + 1);

            // Retroceder (backtracking)
            temp = arreglo[start];
            arreglo[start] = arreglo[i];
            arreglo[i] = temp;
        }
    }
}
```

- **Complejidad lineal-logarítmica**:

$$\mathcal{O}(n \log n)$$

Ejemplo: Algoritmo de ordenamiento eficiente Heap sort.

1. Construcción del heap: La construcción del heap toma $O(n)$. Aunque heapify es $O(log n)$ se llama $n/2$ veces, y el costo total amortizado es$O(n)$, es decir el mayor.
2. Extracción de elementos: Se extrae el máximo elemento (la raíz del heap) n veces. Cada extracción implica reorganizar el heap, lo que toma $O(log n)$. Por lo tanto, este paso toma $O(n log n)$.
3. Complejidad total: La construcción del heap $O(n)$ mas la extracción de elementos $O(n log n)$ resulta en una complejidad total de $O(n log n)$.

```
public class HeapSort {
    // Función principal que implementa Heap Sort
    public void heapSort(int[] arreglo) {
        int n = arreglo.length;
        // Construye el heap (reorganiza el arreglo)
```

```
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arreglo, n, i);
        }
        // Extrae elementos del heap uno por uno
        for (int i = n - 1; i > 0; i--) {
            // Mueve la raíz actual al final
            int temp = arreglo[0];
            arreglo[0] = arreglo[i];
            arreglo[i] = temp;
            // Llama a heapify en el heap reducido
            heapify(arreglo, i, 0);
        }
    }
    // Función para convertir un subárbol en un heap
    private void heapify(int[] arreglo, int n, int i) {
        int largest = i; // Inicializa el más grande como la raíz
        int left = 2 * i + 1; // Índice del hijo izquierdo
        int right = 2 * i + 2; // Índice del hijo derecho
        // Si el hijo izquierdo es más grande que la raíz
        if (left < n && arreglo[left] > arreglo[largest]) {
            largest = left;
        }
        // Si el hijo derecho es más grande que el más grande hasta ahora
        if (right < n && arreglo[right] > arreglo[largest]) {
            largest = right;
        }
        // Si el más grande no es la raíz
        if (largest != i) {
            // Intercambia la raíz con el más grande
            int temp = arreglo[i];
            arreglo[i] = arreglo[largest];
            arreglo[largest] = temp;
            // Recursivamente heapify el subárbol afectado
            heapify(arreglo, n, largest);
        }
    }
    // Función para imprimir el arreglo
    public void printArray(int[] arreglo) {
        for (int value : arreglo) {
            System.out.print(value + " ");
        }
        System.out.println();
    }
    // Método principal para probar el algoritmo
    public static void main(String[] args) {
        int[] arreglo = {38, 27, 43, 3, 9, 82, 10};
        HeapSort heapSort = new HeapSort();
        System.out.println("Arreglo original:");
        heapSort.printArray(arreglo);
        heapSort.heapSort(arreglo);
        System.out.println("Arreglo ordenado:");
```

```
57              heapSort.printArray(arreglo);
58          }
59      }
60  }
```

3.1.2. Pasos para calcular la complejidad de un algoritmo

1. **Identificar el tamaño de la entrada (n):** Determinar qué representa n en el algoritmo. Por ejemplo, en un algoritmo que procesa un arreglo, n podría ser el número de elementos del arreglo.
2. **Contar las operaciones básicas:** Identificar las operaciones que más contribuyen al tiempo de ejecución (por ejemplo, bucles, recursiones, operaciones matemáticas).
3. **Expresar la complejidad en términos de n:** Simplificar la expresión eliminando constantes y términos de menor orden.
4. **Usar la notación Big-O:** Expresar la complejidad en términos de Big-O, enfocado en el término dominante.

3.2. Complejidad espacial

La **complejidad espacial** se refiere a la cantidad de memoria que un algoritmo requiere para resolver un problema en función del tamaño de la entrada. A diferencia de la complejidad temporal, que mide el tiempo de ejecución, la complejidad espacial se enfoca en el uso de recursos de memoria, como variables, estructuras de datos y llamadas recursivas. Este aspecto es particularmente importante en entornos con limitaciones de memoria, como sistemas embebidos o aplicaciones de gran escala.

Factores que influyen en la complejidad espacial

- **Variables y estructuras de datos**: El espacio utilizado por variables individuales, arreglos, matrices, listas, pilas, colas, etc.
- **Llamadas recursivas**: Cada llamada recursiva consume espacio en la pila de ejecución, lo que puede llevar a un uso significativo de memoria.
- **Memoria auxiliar**: Espacio adicional utilizado por el algoritmo para almacenar resultados intermedios o datos temporales.

Ejemplos de complejidad espacial

- **Complejidad constante** ($\mathcal{O}(1)$): Algoritmos que utilizan una cantidad fija de memoria, independientemente del tamaño de la entrada. Por ejemplo, calcular la suma de dos números.
- **Complejidad lineal** ($\mathcal{O}(n)$): Algoritmos que requieren memoria proporcional al tamaño de la entrada. Por ejemplo, almacenar un arreglo de n elementos.
- **Complejidad cuadrática** ($\mathcal{O}(n^2)$): Algoritmos que utilizan memoria proporcional al cuadrado del tamaño de la entrada. Por ejemplo, almacenar una matriz de $n \times n$.

Importancia de la complejidad espacial

La optimización del uso de memoria es crucial en aplicaciones donde los recursos son limitados. Un algoritmo con baja complejidad espacial no solo es más eficiente, sino también más escalable y sostenible. Por ejemplo, en aplicaciones de big data o dispositivos móviles, la gestión eficiente de la memoria puede marcar la diferencia entre un sistema viable y uno inviable.

3.3. Administración de la memoria

Durante la ejecución de un programa, uno de los recursos indispensables es la memoria. Cualquier programa medianamente complejo debe considerar la cantidad de información que debe manejar, desde las variables globales, los objetos a ser instanciados, la cantidad de funciones a ejecutar así como el orden de llamadas entre funciones o métodos.

3.3.1. Áreas de la memoria durante la ejecución

Entonces de manera general podemos plantear una organización de la memoria asignada a la ejecución de un programa, la cual podría considerarse la forma más estándar conocida y que permite la gestión de todo el caos que involucra realizar la ejecución de un programa, y es la siguiente:

1. **Código fuente**. Donde se almacena una copia de todo el programa y sus dependencias (lenguaje de maquina JAVA).
2. **Datos estáticos**. Concepto heredado de lenguajes de generaciones anteriores, permite almacenar elementos comunes al programa, datos globales.
3. **STACK** o **pila de ejecución**. Que permite administrar la ejecución entre llamadas a funciones y métodos aislando cada función e identificando a la función activa en un determinado instante de tiempo.
4. **HEAP** o **memoria dinámica o compartida**. Donde se crean los datos de tamaños no predecibles o irregulares y el lugar que es monitorizado por el Recolector de Basura.

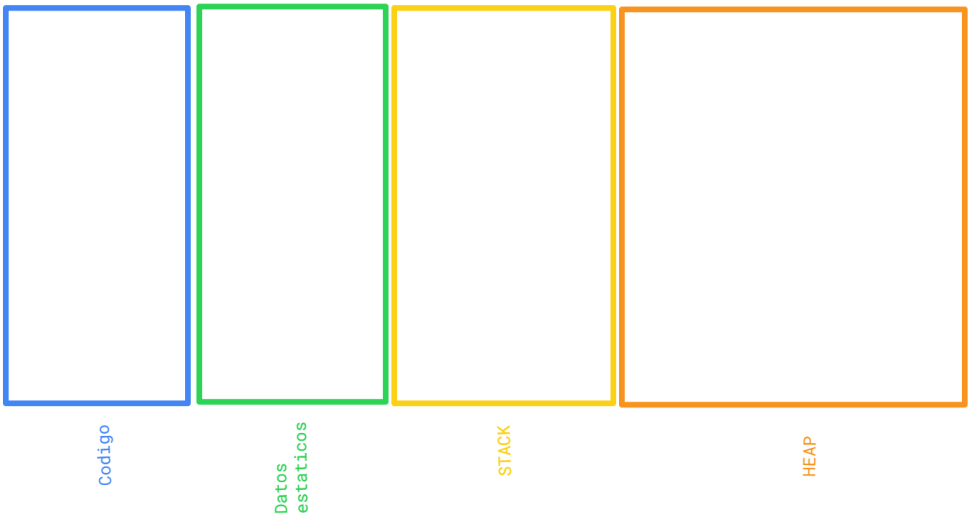


Figura 3.2: áreas de la memoria durante la ejecución

3.3.1.1. Operaciones con referencias o enlaces

Las referencias son fundamentales en JAVA para la manipulación de objetos y son la base del manejo de la memoria en el lenguaje. Las operaciones con referencias o enlaces se refieren a la manipulación de objetos mediante las referencias a ellos. Las referencias en Java son esencialmente *punteros* que apuntan a la dirección de memoria donde se encuentra almacenado un objeto.

Enumeramos algunas operaciones comunes que se realizan con referencias en JAVA:

1. **Declaración de referencias:** Para declarar una referencia a un objeto, se utiliza el tipo de dato de la clase seguido por el nombre de la variable.

```
// Declaracion de referencia
MiClase objeto;
```

2. **Creación de objetos:** Para crear o instanciar un objeto y asignarlo a una referencia, se utiliza el operador **new**.

```
// Creacion de objeto
objeto = new MiClase();
```

3. **Acceso a miembros de objetos:** Una vez que se tiene una referencia a un objeto, se pueden acceder a sus miembros (atributos y métodos) utilizando el operador de acceso punto **(.)**.

```
// Acceso a miembros de objetos
int valor = objeto.getValor();
objeto.setValor(10);
```

4. **Paso de referencias como parámetros:** En JAVA, cuando se pasa un objeto como parámetro a un método, en realidad se esta pasando una referencia al objeto. Esto significa que cualquier cambio que se haga en el objeto dentro del método afecta al objeto original.

```
// Paso de referencias como parametros
modificarObjeto(objeto1);
// el metodo que recibe la referencia
void modificarObjeto(MiClase obj) {
    obj.setValor(20);
}
```

5. **Comparación de referencias:** En JAVA, el operador == usando dos referencias en esencia verifica si dos referencias apuntan a la misma dirección de memoria.

```
// Comparacion de referencias
if (objeto1 == objeto2) {
    // Las referencias apuntan a la misma direccion
    // es decir: al mismo objeto
}
```

6. **Clonación de objetos:** JAVA proporciona el método clone() para crear una copia superficial de un objeto, es decir, una nueva referencia que apunta a un objeto que tiene los mismos valores de los campos que el objeto original.

```
// Clonacion de objetos
MiClase copia = (MiClase) objeto.clone();
```

3.3.2. palabras reservadas this y super

En JAVA, la palabra reservada *super* se usa para acceder a los métodos de la clase principal, mientras que *this* se usa para acceder a los métodos de la clase actual.

3.3.2.1. this

Es una palabra reservada en JAVA, es decir, no podemos usarla como identificador. Se utiliza para referirse a la instancia de la clase actual, así como a los miembros estáticos. Se puede utilizar en varios contextos, como se indica a continuación:

- Para referirse a la variable de instancia de la clase actual.
- Para invocar o iniciar el constructor de la clase actual.
- Se puede pasar como un argumento en la llamada al método.
- Se puede pasar como argumento en la llamada al constructor.
- Se puede utilizar para devolver la instancia de la clase actual.

```
class Ejemplo {
    // instancia de variable
    int a = 10;

    // variable estatica
    static int b = 20;

    void inicializar()
    {
            this.a = 100;
            System.out.println(a);
```

```
12
13              this.b = 600;
14              System.out.println(b);
15      }
16
17      public static void main(String[] args)
18      {
19              new Ejemplo().inicializar();
20      }
21  }
```

3.3.2.2. super

La palabra reservada *super* se refiere a la superclase de la clase en la que se está utilizando actualmente.

1. Se usa para referirse a la instancia de la superclase, así como a los miembros estáticos.

2. También se usa para invocar el método o constructor de la superclase.

```
1  class Padre {
2    // variable de instancia
3    int a = 10;
4    //variable estatica
5    static int b = 20;
6  }
7
8  class Base extends Padre {
9      void imprimir()
10     {
11         System.out.println(super.a);
12         System.out.println(super.b);
13     }
14
15     public static void main(String[] args)
16     {
17         new Base().imprimir();
18     }
19 }
```

El uso más común de *super* es el de eliminar la confusión entre las superclases y subclases que tienen métodos con el mismo nombre. Se puede usar en varios contextos:

- Para hacer referencia a la variable de instancia de la clase principal inmediata.
- Para referirse al método de la clase padre inmediata.
- Para hacer referencia al constructor de la clase principal inmediata.

3.3.3. Recolector de Basura

En JAVA, el recolector de basura se encarga de liberar la memoria de los objetos que ya no son utilizados por la aplicación. Para que el recolector de basura funcione adecuadamente, los

objetos deben ser marcados como "**no utilizados**"por la aplicación. Esto se hace mediante la asignación de valores nulos a las variables que hacen referencia a los objetos o mediante la eliminación de las referencias a los objetos.

Para hacer uso del recolector de basura en JAVA, puedes utilizar el método gc de la clase System: *System.gc()*, que invoca el recolector de basura. Sin embargo, el uso de System.gc() no garantiza que el recolector de basura se ejecute de inmediato. A continuación se muestra un ejemplo básico de su uso:

```
public class EjemploRecolector {
    public static void main(String[] args) {
        EjemploRecolector obj = new EjemploRecolector();
        obj = null; // la referencia al objeto se pierde
        System.gc(); // se invoca el recolector de basura
    }
    @Override
    protected void finalize() throws Throwable {
        System.out.println("Objeto eliminado");
    }
}
```

En este ejemplo, se instancia un objeto de tipo EjemploRecolector y se asigna a la variable *obj*. Luego, se establece la variable *obj* en nulo, lo que significa que ya no hay referencias al objeto creado. Finalmente, se invoca el método System.gc() para solicitar la liberación de memoria utilizada por el objeto. Cuando el recolector de basura libera la memoria del objeto, se llama al método finalize() del objeto, lo que se indica en la consola de salida.

El uso excesivo de System.gc() puede afectar el rendimiento de la aplicación. Además, no es posible determinar exactamente cuándo se ejecutará el recolector de basura. Por lo tanto, se recomienda no depender del recolector de basura para liberar memoria en tiempo real y, en su lugar, liberar explícitamente los objetos cuando ya no sean necesarios en la aplicación.

3.3.3.1. Implementación

La implementación del recolector de basura podría tener los siguientes componentes:

1. *Referencias a objetos*: La implementación del recolector de basura debe mantener un registro de todas las referencias a objetos en la aplicación. Esto se puede lograr mediante el seguimiento de las referencias de objetos en las variables de instancia y en las colecciones como ArrayList y HashMap.
2. *Clasificación de objetos*: El recolector de basura debe ser capaz de distinguir entre los objetos que están siendo utilizados y los que ya no son necesarios. Los objetos que no están siendo utilizados deben ser identificados y marcados para su eliminación.
3. *Algoritmo de eliminación de objetos*: El recolector de basura debe tener un algoritmo para eliminar los objetos marcados para su eliminación. Esto se puede lograr mediante la eliminación de las referencias a los objetos o mediante la liberación de la memoria utilizada por los objetos.

4. *Mecanismos de optimización*: Para mejorar el rendimiento, la implementación del recolector de basura puede utilizar técnicas como la eliminación diferida, la compresión de espacio libre y la generación de diferentes tipos de objetos.
5. Interacción con la JVM: La implementación del recolector de basura debe interactuar con la JVM de JAVA para obtener información sobre los objetos en la aplicación y para liberar la memoria utilizada por los objetos marcados para su eliminación.

La implementación del recolector de basura debe ser capaz de monitorizar todas las referencias de objetos en la aplicación, distinguir entre los objetos que están siendo utilizados y los que no, eliminar los objetos no utilizados y optimizar el rendimiento de la aplicación al utilizar técnicas de eliminación diferida, compresión de espacio libre y generación de diferentes tipos de objetos.

3.3.4. Miembros estáticos

La palabra reservada *static* en JAVA se utiliza principalmente para la gestión de la memoria en casos muy específicos.

3.3.4.1. Características

1. *Asignación de memoria compartida:* A las variables y métodos estáticos se les asigna espacio de memoria solo una vez durante la ejecución del programa. Este espacio de memoria se comparte entre todas las instancias de la clase, lo que hace que los miembros estáticos sean útiles para mantener el estado global o la funcionalidad compartida.
2. *Accesible sin creación de instancias de objetos:* se puede acceder a los miembros estáticos sin necesidad de crear una instancia de la clase. Esto los hace útiles para proporcionar funciones de utilidad y constantes que se pueden usar en todo el programa.
3. *Asociado con la clase, no con los objetos:* los miembros estáticos están asociados con la clase, no con los objetos individuales. Esto significa que los cambios en un miembro estático se reflejan en todas las instancias de la clase y que puede acceder a los miembros estáticos utilizando el nombre de la clase en lugar de una referencia de objeto.
4. *No se puede acceder a miembros no estáticos:* los métodos y variables estáticos no pueden acceder a miembros no estáticos de una clase, ya que no están asociados con ninguna instancia particular de la clase.
5. *Se pueden sobrecargar, pero no anular:* los métodos estáticos se pueden sobrecargar, lo que significa que puede definir varios métodos con el mismo nombre pero con diferentes parámetros. Sin embargo, no se pueden anular, ya que están asociados con la clase en lugar de con una instancia particular de la clase.

Cuando un miembro es declarado estático, se accede a él antes de que se instancie cualquier objeto de su clase y sin referencia a ningún objeto. La palabra reservada *static* es un modificador de no-acceso en JAVA que se aplica en los siguientes escenarios:

- Bloques estáticos.
- Variables estáticas.

- Métodos estáticos.
- Clases estáticas.

3.3.4.2. Bloques estáticos

Para inicializar variables estáticas, se puede declarar un bloque estático que se ejecuta exactamente una vez, cuando la clase se carga por primera vez. El siguiente ejemplo en JAVA demuestra el uso de bloques estáticos.

```
class Ejemplo {
  // variables estaticas
  static int a; //declarado
  static int b = 4; //declarado e inicializado

  // bloque estatico
  static {
    System.out.println("bloque estatico inicializado");
    a = b * 8; //inicializado
  }

  public static void main(String[] args) {
      System.out.println("valor de a : "+ a);
      System.out.println("valor de b : "+ b);
    }
}
```

3.3.4.3. Variables estáticas

Cuando una variable se declara como estática, se crea una sola copia de la variable y se comparte entre todos los objetos en el nivel de clase. Las variables estáticas son, esencialmente, variables globales. Todas las instancias de la clase comparten la misma variable estática.
Puntos importantes para las variables estáticas:

- Podemos crear variables estáticas solo a nivel de clase.
- El bloque estático y las variables estáticas se ejecutan en el orden en que están presentes en un programa.

El siguiente programa en JAVA muestra que el bloque estático y las variables estáticas se ejecutan en el orden en que están declarados en un programa:

```
class Ejemplo {
  // variable estatica
  static int x = inicializar();
  // bloque estatico
  static {
    System.out.println("evaluando el bloque estatico");
  }
  // funcion (estatica)
  private static int inicializar() {
    System.out.println("evaluando inicializar");
```

```
    return 8;
  }
  // funcion main
  public static void main(String[] args)
  {
    System.out.println("evaluando la funcion principal!");
      System.out.println("valor de x : " + x);
    }
}
```

3.3.4.4. Métodos estáticos (funciones)

Cuando un método se declara con la palabra reservada *static*, es conocido como método estático o función. El ejemplo más común de un método estático es el método main(). Como se discutió anteriormente, se puede acceder a cualquier miembro estático antes de que se cree cualquier objeto de su clase y sin referencia a ningún objeto. Los métodos declarados como estáticos tienen varias restricciones:

- Solo pueden llamar directamente a otros métodos estáticos.
- Solo pueden acceder directamente a datos estáticos.
- No pueden referirse a **this** o **super** de ninguna manera.

A continuación se muestra el programa JAVA donde se demuestran las restricciones de los métodos estáticos.

```
class Ejemplo {
    // variable estatica
    static int a = 10;
    // instanciar la variable
    int b = 20;
    // metodo estatico
    static void metodo1()
    {
            a = 20;
            System.out.println("from m1");
            // no se puede acceder al miembro no estatico
            b = 10; // error
            // no se puede acceder al metodo estatico
            // metodo2() de la clase Ejemplo
            metodo2(); // error
            // no se puede usar super desde un metodo estatico
            System.out.println(super.a); // error
    }
    protected void metodo2(){
            System.out.println("from m2");
    }
}
```

3.3.4.5. Clases estáticas

Una clase puede hacerse estática solo si es una clase anidada. No podemos declarar una clase

de nivel superior con un modificador estático, pero podemos declarar clases anidadas como estáticas. Estos tipos de clases se denominan clases estáticas anidadas. La clase estática anidada no necesita una referencia de clase externa. En este caso, una clase estática no puede acceder a miembros no estáticos de la clase externa.

```
1  public class Ejemplo {
2    private static String str = "valor comun para los objetos";
3    // Clase estatica
4    static class ClaseAnidada {
5      // non-static method
6      public void disp(){
7      System.out.println(str);
8      }
9    }
10   public static void main(String args[])
11   {
12     Ejemplo.ClaseAnidada obj
13       = new Ejemplo.ClaseAnidada();
14     obj.disp();
15   }
16 }
```

3.3.4.6. Ventajas del uso de miembros estáticos

- *Eficiencia de la memoria:* a los miembros estáticos se les asigna memoria solo una vez durante la ejecución del programa, lo que puede resultar en un ahorro significativo de memoria para programas grandes.
- *Rendimiento mejorado:* debido a que los miembros estáticos están asociados con la clase en lugar de instancias individuales, se puede acceder a ellos de manera mas rápida y eficiente que a los miembros no estáticos.
- *Accesibilidad global:* se puede acceder a los miembros estáticos desde cualquier parte del programa, independientemente de si se ha creado una instancia de la clase.
- Encapsulación de métodos de utilidad: los métodos estáticos se pueden usar para encapsular funciones de utilidad que no requieren ninguna información de estado de un objeto. Esto puede mejorar la organización del código y facilitar la reutilización de funciones de utilidad en varias clases.
- Constantes: las variables finales estáticas se pueden usar para definir constantes que se comparten en todo el programa.
- Funcionalidad de nivel de clase: los métodos estáticos se pueden usar para definir la funcionalidad de nivel de clase que no requiere ninguna información de estado de un objeto, como métodos de fabrica o funciones auxiliares.

En general, la palabra clave estática es una herramienta poderosa que puede ayudar a mejorar la eficiencia y la organización de los programas en JAVA.

4

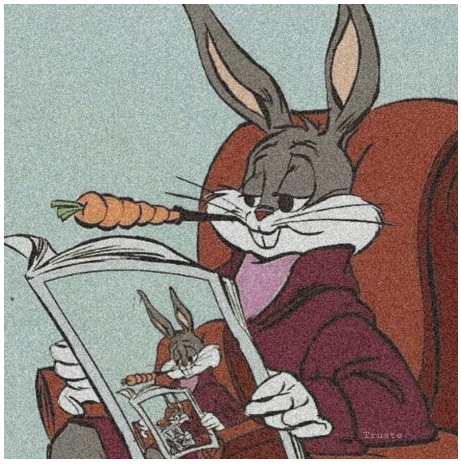
CAPITULO

Capitulo 4 - Elementos de programación

4. La recursividad en la solución de problemas

Muchos de los problemas de programación tienen entre sus soluciones elementos del lenguaje que permiten procesar o ejecutar un conjunto de instrucciones de forma repetitiva, esto es, usando iteradores; entonces, cuando conocemos los usos y ventajas de estos iteradores (for, while, etc.) se modelan o representan los algoritmos para hacer uso de los mismos.

Existe un grupo de problemas para los cuales, si bien su solución o representación puede ser usando iteradores, estos se pueden resolver de forma recursiva si es que sus características así lo permiten. En este capítulo estudiamos una técnica de solución de problemas que justamente se enfoca en descomponer el problema en subproblemas apoyándose en el uso de funciones y no así en el uso de iteradores.



En el contexto de la programación, se denomina recursividad al proceso en el que una función se llama a sí misma directa o indirectamente; esta función es llamada entonces función recursiva. Usando un enfoque recursivo, muchos problemas se pueden resolver con cierta facilidad. Como ejemplos de este tipo de problemas tenemos: Las Torres de Hanoi (TOH), Recorridos en árboles inorden/preorden/postorden, Búsqueda en profundidad, Permutaciones, etc.

Una función recursiva resuelve un problema particular llamando a una copia de sí misma y resolviendo sub-problemas más pequeños del problema original. Es importante proporcionar un escenario conocido como caso base para terminar este proceso de recursión.

La recursividad es una técnica muy interesante, pues nos puede ayudar a reducir la longitud de nuestro código y hacerlo más legible. Su escritura involucra cierto grado de abstracción y puede llegar a tener ciertas ventajas sobre las técnicas de iteración.

4.1. Comprendiendo la recursividad a partir de una interpretación matemática

Consideremos un problema en el que se requiere una función para determinar la sumatoria de los primeros **n** números naturales, el enfoque inicial puede ser el de simplemente agregar los números comenzando desde 1 hasta llegar a n. Entonces la función puede verse de esta

manera:

Enfoque inicial - Simplemente agregando uno por uno todos los valores:

$$f(n) = 1 + 2 + 3 + ... + n$$

Planteamos representar este problema de una forma más concreta, en este caso empezamos desglosando diferentes escenarios concretos para la función descrita previamente, en los cuales buscamos representar distintos valores de **n**:

$$f(5) = 1 + 2 + 3 + 4 + 5 \tag{4.1}$$
$$f(4) = 1 + 2 + 3 + 4 \tag{4.2}$$
$$f(3) = 1 + 2 + 3 \tag{4.3}$$
$$f(2) = 1 + 2 \tag{4.4}$$
$$f(1) = 1 \tag{4.5}$$

Podemos tomar cada una de las expresiones y replantearlas sin perder la equivalencia de las funciones de la siguiente manera:

$$f(5) = f(4) + 5 \tag{4.6}$$
$$f(4) = f(3) + 4 \tag{4.7}$$
$$f(3) = f(2) + 3 \tag{4.8}$$
$$f(2) = f(1) + 2 \tag{4.9}$$
$$f(1) = 1 \tag{4.10}$$

La mayoría de los casos puede expresarse en términos de una expresión menor (problema de menor tamaño), excepto el último caso; justamente este caso es el candidato ideal para ser un caso base, pues ya no se puede expresar como una descomposición del problema en problemas más pequeños.

En base a estas expresiones, planteamos una función, de tal manera que describe todos los escenarios donde hay descomposición. Y adicionalmente incluimos el caso especial (que no se puede descomponer).

Enfoque recursivo: construcción recursiva de la función:

$$f(n) = 1 \qquad n = 1 \tag{4.11}$$
$$f(n) = n + f(n-1) \qquad n > 1 \tag{4.12}$$

Hay una diferencia simple entre el enfoque inicial y el enfoque recursivo y es que en el enfoque recursivo la función *f(n)* se llama a sí misma dentro de su definición, este fenómeno se llama recursión, y la función que contiene la recursión se llama función recursiva. La recursividad puede ser una gran herramienta en la mano de los programadores para codificar algunos problemas de una manera abstracta, mucho más fácil y eficiente.

4.2. Recursividad Lineal

4.2.1. Enfoque recursivo presentado en JAVA

El enfoque recursivo descrito previamente4.1 puede ser expresado usando el lenguaje JAVA de forma directa:

```
1  public static int f(int n)
2  {
3      if(n == 1)
4          return 1;
5      else
6          return n + f(n-1);
7  }
8
9  public static void main(String[] args)
10 {
11     int n = f(5);
12     System.out.println("la sumatoria es:" + n);
13 }
```

Durante la compilación, se verifica el código de la función **f**, inicialmente la firma: se determina que recibe un valor de tipo entero y retorna un valor de tipo entero; posteriormente, en su implementación: en la línea 6 se observa la llamada a la función f (la misma), para ese momento ya se da por válida la definición.

Durante la ejecución se realizan diferentes evaluaciones de la función **f**: se crea una instancia de ejecución (contexto) para cada llamada hasta que la última llamada deriva en el caso base; todas las instancias de ejecución se van clausurando o resolviendo de tal forma que es posible retornar un valor a la llamada inicial.

4.2.2. Usando funciones recursivas para evaluar arreglos

Cuando uno se imagina un arreglo o vector, automáticamente asocia un iterador como por ejemplo **for** para tener un mecanismo de visita a los elementos, como se puede ver en el ejemplo a continuación:

```
1 int[]arreglo = new int[]{1,2,3,4};
2 for(int i=0; i<arreglo.length;i++)
3 {
4    System.out.print(arreglo[i]);//evaluar elemento i
5 }
```

Comprendiendo el funcionamiento de las funciones recursivas en JAVA, también sería posible acceder a los elementos del arreglo de forma recursiva de la siguiente manera:

Considerando como parámetro adicional de la función el número de índice que toca acceder, cada instancia de ejecución de la función recursiva evaluará un elemento del arreglo, y realizará la llamada a la misma definición pero para evaluar el siguiente elemento, de esta forma

se lograría visitar todos los elementos del arreglo hasta que no queda alguno pendiente, en cuyo caso el valor del índice será mayor al rango de elementos (y tomaremos ese caso como base):

```
public static void main(String[] args)
{
    int[]arreglo = new int[]{1,2,3,4};
    int suma = sumar(arreglo,0); //empezar por el elemento 0
}

private static int sumar(int[]arreglo,int i)
{
   if(i>=arreglo.length) //el indice ya esta fuera del rango
       return 0;         //no quedan mas elementos por evaluar
   else
       return arreglo[i] + // elemento a evaluar
           sumar(arreglo,i+1); //siguiente elemento a evaluar
}
```

4.2.3. Usando funciones recursivas para evaluar cadenas

Comprendiendo el comportamiento de las cadenas (String)2.4.2 en JAVA y que tienen una implementación similar a los vectores o arreglos, entonces podemos implementar funciones recursivas en JAVA que accedan a los elementos de una cadena de forma recursiva. Para ello planteamos 2 variantes:

4.2.3.1. Usando solamente una instancia de cadena

Considerando como parámetro adicional de la función el número de índice que toca acceder, cada instancia de ejecución de la función recursiva evaluará un elemento de la cadena, y realizará la llamada a la misma definición pero para evaluar el siguiente elemento, de esta forma se lograría visitar todos los elementos de la cadena hasta que no queda alguno pendiente, en cuyo caso el valor del índice será mayor al rango de elementos (y tomaremos ese caso como base):

```
public static void main(String[] args)
{
    String texto = "hola desde la funcion main";
    imprimirVertical(texto,0); //empezar por el elemento 0
}

private static void imprimirVertical(String texto,int i)
{
   if(i>=texto.length()) //el indice ya esta fuera del rango
       return;          //no quedan mas elementos por evaluar
   else
   {
       System.out.println(texto.charAt(i)); // elemento a evaluar
       imprimirVertical(texto,i+1); //siguiente elemento a evaluar
   }
}
```

4.2.3.2. Usando diferentes instancias de la cadena (subcadenas)

En este escenario, cada instancia de ejecución de la función recursiva evaluará el primer elemento de la cadena, y realizará la llamada a la misma definición pero pasa como información una subcadena que ya no contiene al elemento evaluado; de esta forma se lograría evaluar todos los elementos de la cadena hasta que no queda alguno pendiente, en cuyo caso el valor de la subcadena será **cadena vacía** (y tomaremos ese caso como base):

```
1  public static void main(String[] args)
2  {
3    String texto = "hola desde la funcion main";
4    imprimirVertical(texto); //empezando con el texto completo
5  }
6
7  private static void imprimirVertical(String texto)
8  {
9   if(texto==null || "".equals(texto))
10    return;                    //no quedan mas elementos
11  else
12  {
13    System.out.println(texto.charAt(0));  //elemento a evaluar en 0
14    imprimirVertical(texto.subString(1)); //seguir evaluando
15                                          //los elementos faltantes
16  }
17 }
```

4.3. Recursividad No Lineal

Comprendiendo el comportamiento de la ejecución de una función recursiva lineal, observamos que la recursividad da lugar a momentos de ejecución en los que se tiene una parte de la solución final. Entonces surge una pregunta: ¿qué pasa si a partir de una parte de la solución (o solución parcial) queremos intentar construir la solución final por otro camino?.

4.3.1. Permutaciones

Una permutación es un arreglo de todos o parte de un conjunto de objetos, considerando el orden en que se colocan. Por ejemplo, las permutaciones de los elementos A, B, C son:

- ABC
- ACB
- BAC
- BCA
- CAB
- CBA

4.3.1.1. Permutaciones usando un arreglo

```
1  import java.util.Arrays;
2  public class PermutacionesArreglo {
3      // Método principal para generar permutaciones
4      public static void permutar(int[] arreglo, int inicio) {
5          if (inicio == arreglo.length - 1) {
6              System.out.println(Arrays.toString(arreglo));
7          } else {
8              for (int i = inicio; i < arreglo.length; i++) {
9                  // Intercambiamos el elemento actual con el
10                 // elemento en la posición 'inicio'
11                 intercambiar(arreglo, inicio, i);
12                 // Llamada recursiva para permutar el resto del
13                 // arreglo
14                 permutar(arreglo, inicio + 1);
15                 // Deshacemos el intercambio para volver al estado
16                 // original
17                 intercambiar(arreglo, inicio, i);
18             }
19         }
20     }
```

```
21      // Método auxiliar para intercambiar dos elementos en un arreglo
22      private static void intercambiar(int[] arreglo, int i, int j) {
23          int temp = arreglo[i];
24          arreglo[i] = arreglo[j];
25          arreglo[j] = temp;
26      }
27      public static void main(String[] args) {
28          int[] arreglo = {1, 2, 3};
29          permutar(arreglo, 0);
30      }
31  }
```

4.3.1.2. Permutaciones usando un String

```
1   public class PermutacionesString {
2
3       // Método principal para generar permutaciones
4       public static void permutar(String str, int inicio, int fin) {
5           if (inicio == fin) {
6               System.out.println(str);
7           } else {
8               for (int i = inicio; i <= fin; i++) {
9                   str = intercambiar(str, inicio, i);
10                  permutar(str, inicio + 1, fin);
11                  str = intercambiar(str, inicio, i);
12              }
13          }
14      }
15
16      // Método auxiliar para intercambiar dos caracteres en un String
17      private static String intercambiar(String str, int i, int j) {
18          char[] charArray = str.toCharArray();
19          char temp = charArray[i];
20          charArray[i] = charArray[j];
21          charArray[j] = temp;
22          return String.valueOf(charArray);
23      }
24
25      public static void main(String[] args) {
26          String str = "ABC";
27          permutar(str, 0, str.length() - 1);
28      }
29  }
```

4.3.2. Backtracking

En su forma básica, la idea de Backtracking se asemeja a un recorrido en profundidad dentro de un grafo dirigido, es decir, un recorrido de forma NO lineal. El grafo en cuestión suele ser representado como un árbol, o por lo menos queda claro que no contiene ciclos. Sea cual sea

su estructura, existe solo implícitamente y lo usamos para entender el comportamiento de la memoria durante la ejecución de las llamadas recursivas.

El objetivo del recorrido es encontrar soluciones para algún problema. Esto se consigue haciendo permutaciones o combinaciones que permiten ir construyendo soluciones parciales a medida que progresa el recorrido; estas soluciones parciales limitan las regiones en las que se puede encontrar una solución completa. El recorrido tiene éxito si, procediendo de esta forma, se puede definir por completo una solución (una combinación de todos los elementos que representa una solución al problema). En este caso, el algoritmo puede, o bien detenerse (si lo único que se necesita es una solución del problema) o bien seguir buscando soluciones alternativas (si deseamos examinarlas todas).

Por otra parte, el recorrido no tiene éxito si en alguna etapa la solución parcial construida hasta el momento no se puede completar, es decir, no vale la pena continuar combinando sobre esa solución parcial. En tal caso, el recorrido vuelve atrás exactamente igual que en un recorrido en profundidad, eliminando sobre la marcha los elementos que se hubieran añadido en cada fase. A esto se le denomina **poda**, considerando que estamos representando el recorrido como un árbol. Cuando vuelve a un nodo que tiene uno o más vecinos sin explorar, prosigue el recorrido intentando otra combinación en búsqueda de una solución.

4.3.3. Descripción de la Técnica

Frecuentemente, se implementa este tipo de algoritmos usando funciones recursivas y es el caso que aquí planteamos. Así, en cada llamada a la función, se toma una variable y se le asignan todos los valores posibles, llamando a su vez a la función para cada uno de los nuevos estados. *La diferencia con la búsqueda en profundidad es que se suelen diseñar funciones de cota, de forma que no se generen algunos estados si no van a conducir a ninguna solución, o a una solución peor de la que ya se tiene. De esta forma se ahorra espacio en memoria y tiempo de ejecución.*

Entonces: La técnica Backtracking es un método de búsqueda exhaustiva de soluciones sobre grafos dirigidos acíclicos (árboles), el cual se acelera mediante poda de ramas poco prometedoras. Esto es:

1. Representar todas las posibilidades en un árbol.
2. Buscar la solución por el árbol (de una determinada manera).
3. Evitar zonas del árbol por no contener soluciones **(poda)**. A partir de un estado parcial de la solución se decide no continuar.
4. La solución del problema se representa en una **n-tupla** X1, X2, ... ,Xn de elementos. (no llenando necesariamente todas las componentes).
5. Para cada Xi se escoge desde un conjunto de candidatos.
6. A cada posible **n-tupla** se le llama estado.
7. Se trata de buscar estados solución del problema. Que son en esencia una permutación o combinación de los elementos que representan una solución al problema.
8. Se puede dejar de buscar estados solución cuando:

a) se consiga un estado solución
b) se consigan todos los estados solución
c) se ha recorrido todo el árbol y no se ha encontrado solución alguna.

4.3.3.1. Diseño del Algoritmo de Backtracking

Para diseñar un algoritmo de Backtracking usando recursividad, podemos seguir los siguientes pasos:

1. Buscar una representación del tipo *X1, X2, . . . , Xn* para las soluciones del problema.
2. Identificar las restricciones implícitas y explicitas del problema.
3. Establecer la organización del árbol de ejecución o recorrido que define los diferentes estados en los que se encuentra una solución parcial.
4. Definir una función solución para determinar si una tupla es solución.
5. Definir una función de poda *Bk X1, X2, . . . , Xk* para eliminar ramas del árbol que puedan derivar en soluciones poco deseables o no deseadas.

4.3.3.2. Implementación

Una manera de implementar un algoritmo de Backtracking previamente diseñado usando recursividad sería la siguiente:

1. Definir tres casos en términos de código:

   a) (caso base) Validar solución parcial: Estamos en el camino equivocado –>este camino no nos puede llevar a la solución final.
   b) (caso base) Validar solución final: Encontramos y aceptamos una solución final para el problema.
   c) (caso recursivo) Paso: Estamos en algún punto entre la A a la B, continuamos con el siguiente paso.

2. Llevar los tres casos a una estructura de código recursiva similar a la siguiente:

```
static boolean functionBacktracking(vectorEntrada, vectorSolucion)
{
  // Caso Base 1: podar
  if (!esSolucionParcial)
  {
    return false;
  }
  // Caso Base 2: verificar y aceptar la solucion (completa)
  else if (esSolucionFinal)
  {
    print(vectorSolucion);
    return true;
  }
  else
  {
    // Paso: navegar sobre las posibles soluciones
    for (n in vectorEntrada)
    {
```

```
19        vectorSolucion.push(n);
20        vectorEntrada.remove(n);
21        test = functionBacktracking(vectorEntrada, vectorSolucion);
22        if (test == true)
23        {
24          return true;
25        }
26        else
27        {
28          vectorEntrada.push(n);
29          vectorSolucion.remove(n);
30        }
31        // No hay soluciones
32        return false;
33      }
34    }
35  }
```

5

CAPITULO

Capitulo 5 - Programación Orientada a Objetos

5. Programación Orientada a Objetos

El paradigma orientado a objetos es un enfoque de programación que se basa en el uso de objetos, los cuales son entidades que contienen datos y métodos (funciones) que operan en esos datos. Estos objetos se utilizan para modelar y representar objetos del mundo real o conceptos abstractos en un programa de computadora.

El paradigma orientado a objetos se basa en cuatro principios fundamentales: encapsulamiento, herencia, polimorfismo y abstracción. En conjunto, estos principios permiten la creación de programas modulares, escalables y fáciles de mantener, y son la base de muchos lenguajes de programación populares como JAVA, Python, C++, Ruby y muchos otros.

5.1. Los 4 Pilares de la programación orientada a objetos

La programación orientada a objetos (POO) se basa en cuatro pilares fundamentales que definen su enfoque y estructura. Estos pilares son:

1. **Abstracción:** La abstracción es el proceso de identificar las características esenciales de un objeto o entidad y eliminar los detalles irrelevantes. En POO, se crea una clase para representar un objeto o entidad, y esta clase encapsula sus propiedades (atributos) y comportamientos (métodos) relevantes. La abstracción permite enfocarse en lo que es importante y ocultar la complejidad interna del objeto.

2. **Encapsulamiento:** El encapsulamiento se refiere a la agrupación de datos y funciones relacionadas en una sola unidad llamada clase. Una clase define un conjunto de propiedades y métodos que operan sobre esos datos. El encapsulamiento protege los datos internos de una clase y proporciona métodos para acceder y modificar esos datos de manera controlada. Esto promueve la seguridad y la integridad de los datos, al tiempo que permite la reutilización de código y la simplificación de la implementación.

3. **Herencia:** La herencia es un mecanismo que permite crear nuevas clases basadas en clases existentes. Una clase derivada o subclase hereda las propiedades y métodos de su clase base o superclase, lo que le permite reutilizar el código y extender su funcionalidad. La herencia permite establecer jerarquías y relaciones entre las clases, lo que facilita la organización y el mantenimiento del código.

4. **Polimorfismo:** El polimorfismo se refiere a la capacidad de objetos de diferentes clases para responder al mismo mensaje o realizar una acción similar de formas diferentes. Esto significa que un mismo método puede tener diferentes comportamientos dependiendo del tipo de objeto que lo este ejecutando. El polimorfismo permite escribir código mas genérico y flexible, ya que se pueden tratar distintos objetos de manera uniforme a través de interfaces comunes.

5.2. Herencia

En JAVA, la herencia es un pilar importante de la POO (programación orientada a objetos). Es el mecanismo en JAVA mediante el cual se permite a una clase heredar las características (campos y métodos) de otra clase. En JAVA, herencia significa crear nuevas clases basadas en las existentes; es un mecanismo fundamental para lograr la reutilización de código. Permite que una clase herede los atributos y métodos de otra clase, lo que significa que una clase puede aprovechar y extender el comportamiento de una clase padre.

5.2.1. Ventajas de la herencia en JAVA

1. **Reutilización del código:** La herencia en JAVA promueve la reutilización de código, ya que evita la duplicación de código al permitir que las clases hijas hereden el comportamiento común de las clases padre. Esto facilita la creación y mantenimiento de aplicaciones, ya que los cambios en la clase padre se reflejan automáticamente en todas las clases hijas que la heredan.
2. **Sobre-escritura de métodos:** la sobre-escritura de métodos solo se puede lograr mediante herencia. Es una de las formas en que JAVA logra el polimorfismo en tiempo de ejecución.
3. **Abstracción:** El concepto de abstracción en el que no tenemos que proporcionar todos los detalles se logra mediante la herencia. La abstracción solo muestra la funcionalidad al usuario.
4. **Jerarquía de clases:** la herencia permite la creación de una jerarquía de clases, que se puede utilizar para modelar objetos del mundo real y sus relaciones.
5. **Polimorfismo:** la herencia permite el polimorfismo, que es la capacidad de un objeto de adoptar múltiples formas. Las subclases pueden anular los métodos de la superclase, lo que les permite cambiar su comportamiento de diferentes maneras.

5.2.2. Desventajas de la herencia en JAVA

1. **Complejidad:** la herencia puede hacer que el código sea mas complejo y mas difícil de entender. Esto es especialmente cierto si la jerarquía de herencia es profunda o si se utilizan herencias múltiples.
2. **Acoplamiento estrecho:** la herencia crea un acoplamiento estrecho entre la superclase y la subclase, lo que dificulta realizar cambios en la superclase sin afectar a la subclase.

5.2.3. Terminología utilizada en la herencia de JAVA

1. **Clase:** Una clase es un conjunto de objetos que comparten características/comportamiento y propiedades/atributos comunes. La clase no es una entidad del mundo real. Es simplemente una plantilla, plano, prototipo o diseño a partir del cual se crean los objetos.
2. **Superclase/clase principal:** la clase cuyas características se heredan se conoce como superclase (o clase base o clase principal).
3. **Subclase/clase secundaria:** la clase que hereda de la superclase se conoce como subclase (o clase derivada, clase extendida o clase secundaria). La subclase puede agregar sus propios campos y métodos (ademas de ya tener los campos y métodos **heredados** de la superclase).
4. **Re-usabilidad:** La herencia admite el concepto de reutilización", es decir, cuando queremos crear una nueva clase y ya existe una clase que incluye parte del código que queremos, podemos derivar nuestra nueva clase a partir de la clase existente. Al hacer esto, estamos reutilizando los campos y métodos de la clase existente.

5.2.4. Como utilizar la herencia en JAVA

Para establecer una relación de herencia entre dos clases en JAVA, se utiliza la palabra clave ëxtends̈. La clase hija hereda todos los miembros (atributos y métodos no privados) de la clase padre y puede agregar nuevos miembros o anular los existentes.

5.2.5. Consideraciones importantes de la herencia en JAVA

1. **Superclase predeterminada:** excepto la clase **Object** , que no tiene superclase, cada clase tiene una y solo una superclase directa (herencia única). En ausencia de cualquier otra superclase explicita, cada clase es implícitamente una subclase de la clase Object.
2. **La superclase solo puede ser una:** una superclase puede tener cualquier numero de subclases. Pero una subclase solo puede tener una superclase. Esto se debe a que JAVA no admite herencias múltiples con clases. Aunque con interfaces, JAVA admite múltiples herencias.
3. **Herencia de constructores:** una subclase hereda todos los miembros (campos, métodos y clases anidadas) de su superclase. Los constructores no son miembros, por lo que las subclases no los heredan, pero el constructor de la superclase se puede invocar desde la subclase.
4. **Herencia de miembros privados:** una subclase no hereda los miembros privados de su clase principal. Sin embargo, si la superclase tiene métodos públicos o protegidos (como captadores y definidores) para acceder a sus campos privados, la subclase también puede utilizarlos.

5.3. Polimorfismo

La palabra polimorfismo significa tener muchas formas. En palabras simples, podemos definir el polimorfismo de JAVA como la capacidad de un mensaje de mostrarse en más de una forma.

En este artículo, aprenderemos qué es el polimorfismo y sus tipos.

Ilustración de la vida real del polimorfismo en JAVA : una persona al mismo tiempo puede tener diferentes características. Como un hombre es al mismo tiempo padre, ingeniero y delegado de su OTB. Entonces, la misma persona posee diferentes comportamientos en diferentes situaciones. Esto se llama polimorfismo.

5.3.1. Que es el polimorfismo en JAVA?

El polimorfismo se considera una de las características importantes de la programación orientada a objetos. El polimorfismo nos permite realizar una misma acción de diferentes formas. En otras palabras, el polimorfismo le permite definir una interfaz y tener múltiples implementaciones. La palabra ”poli”significa muchas y ”morfos”significa formas, por lo que significa muchas formas.

5.3.2. Tipos de polimorfismo de JAVA

En JAVA, el polimorfismo se divide principalmente en dos tipos:

1. Polimorfismo en tiempo de compilación.
2. Polimorfismo en tiempo de ejecución.

5.3.2.1. Polimorfismo en tiempo de compilación en JAVA

También se le conoce como polimorfismo estático. Este tipo de polimorfismo se logra mediante sobrecarga de funciones o sobrecarga de operadores (JAVA no permite la sobrecarga de operadores al programador).

Cuando hay varias funciones con el mismo nombre pero con diferentes parámetros, se dice que estas funciones están sobrecargadas . Las funciones pueden sobrecargarse mediante cambios en el número de argumentos y/o un cambio en el tipo de argumentos.

```
1  class ClaseA {
2    static int Multiplicar(int a, int b)
3    {
4      return a * b;
5    }
6
7    static double Multiplicar(double a, double b)
8    {
9      return a * b;
10   }
11
12     static int Multiplicar(int a, int b, int c)
13     {
14         return a * b * c;
15     }
16 }
17
```

```
18 class Main {
19   public static void main(String[] args)
20   {
21     System.out.println(ClaseA.Multiplicar(2, 4));
22     System.out.println(ClaseA.Multiplicar(5.5, 6.3));
23         System.out.println(ClaseA.Multiply(2, 7, 3));
24   }
25 }
```

5.3.2.2. Polimorfismo en tiempo de ejecución en JAVA

También se conoce como resolución dinámica de método. Es un proceso en el que una llamada de función al método anulado se resuelve en tiempo de ejecución. Este tipo de polimorfismo se logra mediante la anulación o sobreescritura de método. La anulación de métodos, por otro lado, ocurre cuando una clase derivada tiene una definición para una de las funciones miembro de la clase base. Se dice que esa función base está anulada o sobreescrita.

El polimorfismo en tiempo de ejecución en JAVA se puede lograr de dos maneras: a través del polimorfismo de clases y del polimorfismo de interfaces. Tanto el polimorfismo de clases como el polimorfismo de interfaces son conceptos clave en JAVA para lograr flexibilidad y extensibilidad en el código, permitiendo tratar a objetos de diferentes clases como si fueran del mismo tipo.

5.3.2.3. El polimorfismo de clases

en JAVA es un concepto fundamental de la programación orientada a objetos que permite tratar objetos de diferentes clases de manera uniforme. El polimorfismo se basa en la capacidad de una clase más general (llamada clase base o superclase) de ser utilizada para referenciar objetos de clases más específicas (llamadas subclases) que heredan de ella.

El polimorfismo en JAVA se logra mediante el uso de herencia y la implementación de métodos que tienen la misma firma (nombre y parámetros) en la clase base y en las subclases. A través del polimorfismo, un objeto de una subclase puede ser tratado como un objeto de la clase base, lo que permite la flexibilidad y extensibilidad del código.

El polimorfismo de clases en JAVA permite escribir código más genérico y reutilizable, ya que se puede tratar a objetos de diferentes clases de manera uniforme, siempre y cuando compartan una relación de herencia común. Esto facilita la extensión del código en el futuro, ya que nuevas subclases pueden agregarse sin necesidad de modificar el código existente.

5.3.2.4. Polimorfismo de interfaces

El polimorfismo de interfaces en JAVA permite que una clase implemente una interfaz y que múltiples clases diferentes implementen esa misma interfaz. Esto permite tratar a objetos de diferentes clases como si fueran del mismo tipo cuando implementan la misma interfaz.

El polimorfismo de interfaces permite escribir código más flexible y desacoplado, ya que las clases pueden implementar múltiples interfaces y ser tratadas como cualquiera de ellas. Esto facilita la creación de código reutilizable y extensible, ya que se pueden agregar nuevas implementaciones de interfaces sin afectar el código existente que depende de esas interfaces.

5.3.3. Ventajas del polimorfismo en JAVA

1. Aumenta la reutilización del código al permitir que objetos de diferentes clases sean tratados como objetos de una clase común.
2. Mejora la legibilidad y el mantenimiento del código al reducir la cantidad de código que debe escribirse y mantenerse.
3. Admite enlace dinámico, lo que permite llamar al método correcto en tiempo de ejecución, según la clase real del objeto.
4. Permite tratar objetos como un solo tipo, lo que facilita la escritura de código genérico que pueda manejar objetos de diferentes tipos.

5.3.4. Desventajas del polimorfismo en JAVA

1. Puede hacer que sea mas difícil comprender el comportamiento de un objeto, especialmente si el código es complejo.
2. Esto puede provocar problemas de rendimiento, ya que el comportamiento polimorfico puede requerir cálculos adicionales en tiempo de ejecución.

6

CAPITULO

Capitulo 6 - Programación Genérica

6. Programación Genérica

Para poder comprender lo que es la programación genérica en JAVA, es importante iniciar por estudiar la clase Object, y entonces plantear situaciones donde se observa que se podría reutilizar código a partir de la parametrización de los tipos de datos.

6.1. La Clase Object

La clase Object está presente en el paquete **java.lang**. Cada clase en Java se deriva directa o indirectamente de la clase Object. Si una clase no hereda de ninguna otra clase, entonces es una clase hija directa de Object y si hereda de otra clase, entonces hereda indirectamente de Object. Por lo tanto, los métodos de la clase Object están disponibles para todas las clases Java. Por tanto, la clase Object actúa como raíz de la jerarquía de herencia en cualquier programa Java.

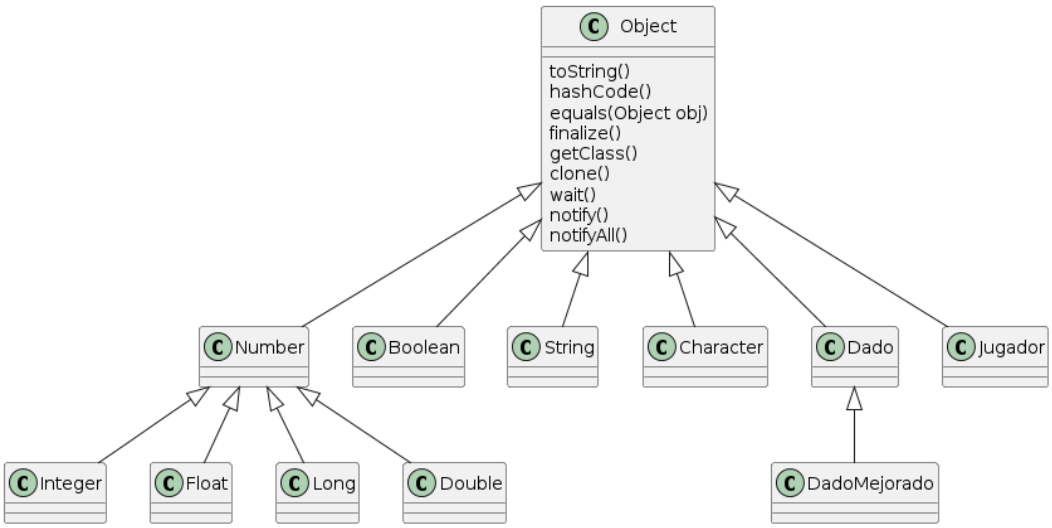


Figura 6.1: Object y la jerarquía de clases

La clase Object proporciona los siguientes métodos:

1. `toString()`: Este método devuelve una representación de cadena del objeto. Por defecto, devuelve una cadena que contiene el nombre de la clase del objeto y su dirección de memoria.
2. `equals(Object obj)`: Este método compara si el objeto actual es igual al objeto pasado como argumento. Por defecto, este método compara las direcciones de memoria de los objetos, pero es común sobrescribir el mismo en las clases derivadas para realizar una comparación significativa basada en el contenido de los objetos.
3. `hashCode()`: Devuelve un código hash único para el objeto. Este método es útil cuando se trabaja con colecciones basadas en hash, como HashMap, HashSet, etc.
4. `getClass()`: Devuelve el objeto Class que representa la clase del objeto. Esto puede ser útil para obtener información sobre la clase del objeto en tiempo de ejecución.

5. `notify()`, `notifyAll()`, `wait()`: Estos métodos son utilizados para la comunicación entre hilos. `notify()` despierta un hilo en espera que este esperando en el objeto actual, `notifyAll()` despierta todos los hilos en espera y `wait()` hace que el hilo actual espere hasta que otro hilo lo notifique.
6. `finalize()`: Este método es llamado por el recolector de basura antes de liberar la memoria ocupada por el objeto. Es utilizado para realizar operaciones de limpieza o liberación de recursos antes de que el objeto sea destruido.

6.2. Genéricas

Como pudimos observar en la sección anterior, la clase **Object** es la superclase de todas las demás clases, entonces las referencias de tipo Object podrían usarse para referirse a cualquier objeto de cualquier tipo. Esta característica genera un problema de validación y control del tipo de dato respecto a los objetos. Las genéricas ayudan a gestionar este problema de seguridad sesgando los objetos hacia tipos concretos.

Podemos entender a las **genéricas** como una manera de tener tipos parametrizados en el programa, es decir, permitir que el tipo (entero, cadena, etc., y tipos definidos por el usuario) sea un parámetro para los métodos, clases e interfaces. Utilizando genéricas, es posible crear clases que trabajen con diferentes tipos de datos. Una entidad como clase, interfaz o método que opera en un tipo parametrizado es una entidad genérica.

6.2.1. Tipos de genéricas en JAVA

- **Clases genéricas:** una clase genérica se implementa exactamente como una clase que no lo es, con la única diferencia de que la primera contiene una sección de parámetros de tipo. Puede haber mas de un parámetro de tipo, separados por una coma. Las clases que aceptan uno o mas parámetros se conocen como clases parametrizadas o tipos parametrizados.
- **Métodos genéricos:** Un método genérico tiene parámetros de Tipo que se citan por el tipo actual. Esto permite utilizar el método genérico de una manera mas general. El compilador se ocupa del tipo de seguridad que permite a los programadores codificar fácilmente, ya que no tienen que realizar largas conversiones de tipos individuales.

6.2.2. Clases genéricas

Usamos <>para especificar tipos de parámetros en la creación de clases genéricas. Para crear objetos de una clase genérica, usamos la siguiente sintaxis.

```
1 class MiClase<T> {
2         T atribA;
3         MiClase(T atribA) { this.atribA = atribA; }
4         public T getA() { return this.atribA; }
5 }
6 class Main {
7   public static void main(String[] args)
```

```
8     {
9       MiClase<Integer> c1 = new MiClase<Integer>(15);
10      System.out.println(c1.getA());
11
12      MiClase<String> c2 = new MiClase<String>("Hola!");
13      System.out.println(c2.getObject());
14    }
15  }
```

6.2.3. Funciones genéricas

También podemos escribir funciones genéricas que se pueden llamar con diferentes tipos de argumentos según el tipo de argumentos pasados al método genérico. El compilador maneja cada método.

```
1  class Test {
2    static <T> void mostrarGenerico(T element)
3    {
4      System.out.println(element.getClass().getName()
5              + " = " + element);
6    }
7    public static void main(String[] args)
8    {
9      mostrarGenerico(11);
10     mostrarGenerico("Hola desde main!");
11     mostrarGenerico(1.0);
12   }
13 }
```

6.2.4. Ventajas de las genéricas

Los programas que usan genéricas tienen muchos beneficios sobre el código no genérico.

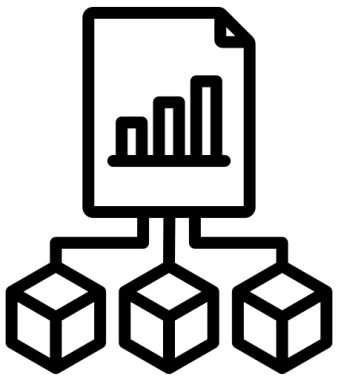
1. **Reutilización de código:** podemos escribir un método/clase/interfaz una vez y usarlo para cualquier tipo que queramos.
2. **Seguridad de tipos:** las genéricas detectan errores que aparecen en tiempo de compilación. Es preferible identificar los problemas en código en tiempo de compilación en lugar de en tiempo de ejecución.
3. **Conversión de tipo:** La conversión un tipo de dato en cada operación de recuperación desde un conjunto de objetos o lista se puede tornar compleja. Si ya sabemos que nuestra lista solo contiene datos de tipo cadena, no necesitamos convertir cada vez.
4. **Implementación de algoritmos genéricos:** mediante el uso de genéricas, podemos implementar algoritmos que funcionan para diferentes tipos de objetos.

7

CAPITULO

Capitulo 7 - Estructuras de Datos

7. Estructuras de datos - TAD y estructuras lineales



7.1. Tipos de Datos Abstractos

Abstraer consiste en ignorar los detalles de la manera particular en que está compuesta una cosa, y representarla solamente con su visión general. Entonces, bajo esa definición, podemos ver a un Tipo de Dato Abstracto (TDA) como un conjunto de valores que pueden tomar los datos de ese tipo, junto a las operaciones que los manipulan.

Un **TDA** es un modelo matemático de una estructura de datos que especifica los tipos de datos almacenados, las operaciones definidas sobre esos datos y los tipos de parámetros de esas operaciones, y no así su representación interna. Esencialmente, un TDA proporciona una interfaz bien definida que especifica qué operaciones se pueden realizar en los datos y cómo se comportan esas operaciones, pero oculta los detalles de cómo se implementan los datos internamente. Esto permite la separación clara entre la estructura de datos y su manipulación, lo que promueve la modularidad, la reutilización del código y la abstracción en el diseño de programas y sistemas.

$$TDA = Valores(tipodedato) + operaciones \tag{7.1}$$

Un **TDA** es materializado por una estructura de datos concreta, en JAVA, esto es, es implementada por una clase. Una clase define los datos que serán almacenados y las operaciones soportadas por los objetos que son instancia de la clase. Como sabemos, al contrario de las interfaces, las clases especifican cómo las operaciones son ejecutadas (implementación).

7.2. Estructura de Datos

Una estructura de datos es una forma de organizar o gestionar un conjunto de datos elementales (un dato elemental es la mínima información que se tiene en el sistema) con el objetivo de facilitar la manipulación de estos datos como un todo y/o individualmente.

Una estructura de datos define la organización e interrelacionamiento de estos datos, y un conjunto de operaciones que se pueden realizar sobre los mismos. Las operaciones básicas son:

- Dar de Alta, adicionar un nuevo valor a la estructura.
- Dar de Baja, borrar un valor de la estructura.
- Búsqueda, buscar un determinado valor en la estructura para realizar una operación con este valor, la búsqueda puede ser por ejemplo de forma SECUENCIAL o BINARIA (si es que los datos estén ordenados).

Otras operaciones que se pueden realizar son:

- Ordenamiento, de los valores pertenecientes a la estructura si es que estos tienen la posibilidad de ser comparados entre ellos (es decir, tienen un criterio de orden: mayor - menor que ...).
- Fusionamiento, dadas dos estructuras originar una nueva ordenada y que contenga todos los valores de ambas listas.

Cada estructura de datos tiene ventajas y desventajas en relación a la simplicidad y eficiencia para la realización de cada operación. Por ello, la elección de la estructura de datos apropiada para cada problema va a depender de factores como la frecuencia y el orden en que se realiza cada operación sobre los datos.

A continuación intentamos listar las estructuras de datos más utilizadas (no excluyentes) en programación :

7.3. Tipos de acceso a los datos

Cuando almacenamos los datos en una estructura, la manera en que accedemos a estos datos puede determinar la eficiencia y pertinencia de la estructura respecto del problema en el que se va a utilizar. Vamos a identificar el acceso a los datos de dos formas: puede ser directo o secuencial.

- **Secuencial.** Para acceder a un objeto se debe acceder a los objetos almacenados previamente en la estructura. El acceso secuencial exige un *recorrido* elemento a elemento, es decir, es necesaria una exploración secuencial comenzando desde el primer elemento disponible en la estructura.
- **Directo o Aleatorio.** Se accede directamente al objeto, sin tener que recorrer los elementos anteriores de forma necesaria. El acceso directo permite procesar o acceder a un elemento determinado haciendo una referencia directamente a su posición en el soporte de almacenamiento.

7.4. Tipos de estructuras de datos

Para fines de su estudio, vamos a agrupar a las estructuras de datos en dos grupos por la forma de organización interna de los valores almacenados: estructuras de datos lineales y estructuras

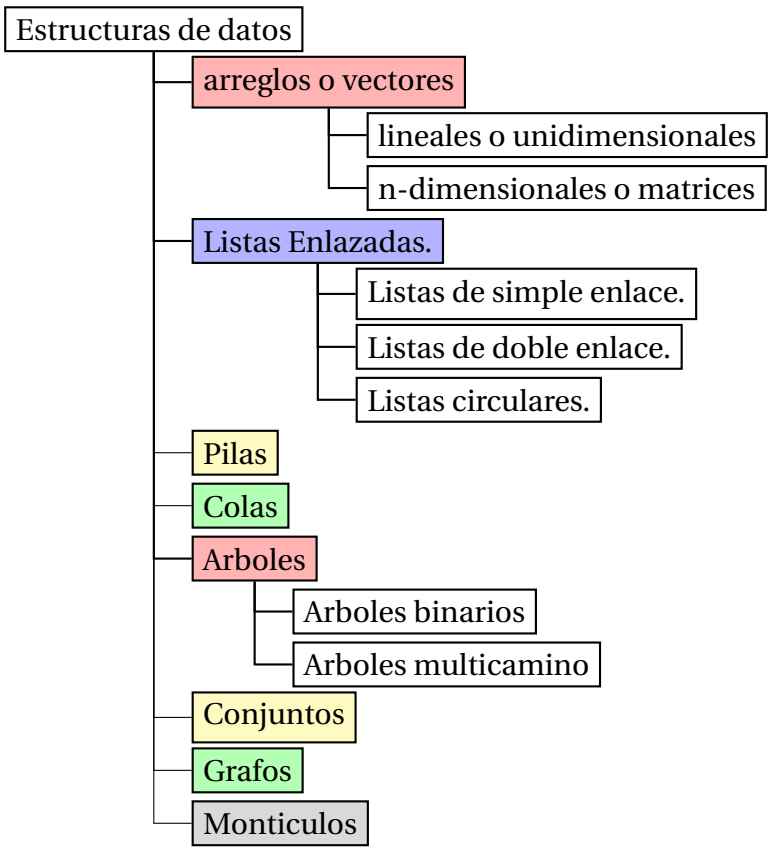


Figura 7.1: clasificación de estructuras de datos

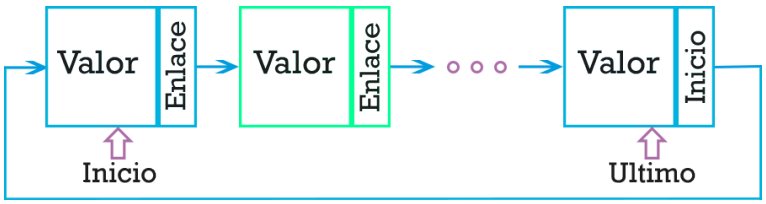
de datos no lineales.

- **Estructuras de Datos Lineales.** Organizar y Almacenan los datos en la memoria de manera secuencial, es decir, los elementos están dispuestos en una secuencia o linea y existe la noción de anterior y posterior. En estas estructuras, cada elemento tiene un sucesor y un predecesor, excepto el primero y el ultimo, que tienen solo uno de los dos. Algunos ejemplos comunes de estructuras de datos lineales incluyen listas, pilas, colas y arreglos (arrays).

- **Estructuras de Datos No Lineales.** Las estructuras de datos no lineales son aquellas en las que los elementos no están organizados de manera secuencial o jerárquica. A diferencia de las estructuras de datos lineales, en las estructuras no lineales los elementos pueden estar interconectados de diversas formas. Entre las estructuras de datos no lineales mas comunes se encuentran los arboles y los grafos. Los arboles se caracterizan por tener un nodo raíz del que se desprenden varios nodos secundarios, formando una estructura jerárquica. Por otro lado, los grafos son conjuntos de nodos interconectados mediante aristas, donde cada nodo puede estar conectado con múltiples nodos. Estas estructuras no lineales son fundamentales en la ciencia de la computación y se utilizan en una amplia gama de aplicaciones, desde bases de datos hasta algoritmos de búsqueda y recorrido.

7.5. Estructuras de Datos Lineales

Estas estructuras de datos se utilizan para representar situaciones en las que los datos tienen un orden específico y en las que la inserción y eliminación de elementos sigue un patrón secuencial. La elección de una estructura de datos lineal sobre otra generalmente depende de los requisitos específicos del problema que se está resolviendo y de las operaciones que se realizarán con los datos.



7.5.1. Listas Enlazadas

Definición Las listas son probablemente la segunda estructura de almacenamiento de propósito general más comúnmente utilizadas, después de los *arreglos* o *vectores*.

Considerando que los arreglos o vectores también son estructuras de datos, mismos que se estudiaron en 2.2, vamos a decir que los mismos presentan ciertas desventajas a la hora de trabajar con muchos datos:

1. En un arreglo desordenado, la búsqueda es lenta.
2. En un arreglo ordenado, la inserción es lenta.
3. En ambos casos, la eliminación es lenta.
4. El tamaño de un arreglo no puede cambiar después que este se ha instanciado.

Una **Lista enlazada** aparece como una alternativa interesante, pues es un mecanismo versátil y muy conveniente para su uso en muchos tipos de estructuras de datos de propósito general. También puede reemplazar a los arreglos como base de la implementación de otras estructuras de almacenamiento, como las pilas y colas.

La ventaja más evidente de utilizar listas enlazadas es que permiten optimizar el uso de la memoria, pues eliminamos la existencia de tener localidades o espacios vacíos:

La desventaja más grande de las Listas enlazadas es que deben ser recorridas desde su inicio para localizar un dato particular. Es decir, no hay forma de acceder al i-ésimo dato de la lista, como lo haríamos en un arreglo.

Algunas estructuras similares a las listas, pero más complejas son, por ejemplo: las listas doblemente enlazadas, las listas circulares, por nombrar algunas.

7.5.2. Lista NO ordenada

Desde el punto de vista de programación, una lista es una estructura de datos dinámica que contiene una colección de elementos homogéneos, con una relación lineal entre ellos. Una re-

lación lineal significa que cada elemento (a excepción del primero) tiene un precedente y cada elemento (a excepción del último) tiene un sucesor.

Visualmente, una lista se puede conceptualizar de la siguiente forma: raíz de la Lista

Lógicamente, una lista es tan solo un objeto de una clase Lista que se conforma de una raíz (posición de inicio) de lista y un conjunto de métodos para operarla.

Se pueden implementar listas estáticas utilizando arreglos de N elementos donde se almacenen los datos. Recordemos la desventaja de esto.

7.5.3. Lista ordenada

Una lista ordenada es una extensión de las listas normales con la cualidad de que todos sus datos se encuentran en orden. Como, al igual que las listas, se debe poder insertar datos, hay dos formas de garantizar el orden de los mismos:

- Crear un método de inserción ordenada, es decir que el dato se acomoda bajo el criterio de orden al momento de la inserción.
- Crear un método de ordenamiento de la lista, en este caso se provee un método que ordena los elementos que ya existen en la lista bajo algún criterio de orden.

7.5.4. Implementación

Los distintos métodos de la lista deben asegurarnos de poder insertar, eliminar u obtener un dato en cualquier posición de la lista.
Como hemos mencionado, una lista se compone de:

- Un raíz de la lista, que es un objeto de Tipo Nodo.
- Un conjunto de instrucciones que controlen su estructura.

La clase Nodo, debe contener al menos un espacio de almacenamiento de datos y una referencia a otro Nodo no necesariamente instanciado. La clase Nodo debe ser nuestra clase base, que incorporará los constructores necesarios para inicializar nuestros datos. El objetivo es una estructura de clase que nos represente:

```
1  public class Nodo<T> {
2      private T valor;
3      private Nodo sig;
4
5      public Nodo(T valor){
6          this.valor = valor;
7          this.sig = null;
8      }
9
10     public T getValor() {
11         return valor;
```

```
    }

    public void setValor(T valor) {
        this.valor = valor;
    }

    public Nodo getSig() {
        return sig;
    }

    public void setSig(Nodo sig) {
        this.sig = sig;
    }

    @Override
    public String toString(){
        return "(" + this.valor + ")->" + this.sig;
    }
}
```

La clase Lista, contiene:

- La referencia a un objeto de tipo Nodo que será la primera referencia de nuestra lista.
- Un grupo de constructores para inicializar la estructura.
- Un grupo de operaciones para el control de la lista, tales como:
  - insertar un nuevo Nodo
  - buscar un Nodo
  - eliminar un Nodo
  - obtener un Nodo en alguna posición
  - mostrar la lista

```
public class ListaEnlazada<T> {
    private Nodo raiz;
    public ListaEnlazada(){
        this.raiz = null;
    }

    public void insertar(T valor){
        if(raiz==null){
            Nodo nuevo = new Nodo(valor);
            raiz = nuevo;
        }
        else{            Nodo aux = raiz;
            while(aux.getSig()!=null){
                aux = aux.getSig();
            }
            Nodo nuevo = new Nodo(valor);
            aux.setSig(nuevo);
        }
    }

```

```
21      public boolean buscar(T valor){
22          Nodo aux = raiz;
23          while(aux!=null){
24              T valorNodo = (T) aux.getvalor();
25              if(valorNodo.equals(valor))
26                  return true;
27              aux = aux.getSig();
28          }
29          return false;
30      }
31
32      @Override
33      public String toString(){
34          return "raiz->" + this.raiz;
35      }
36  }
```

Funciones adicionales, se pueden construir un grupo de funciones adicionales que nos proporcionen información sobre la lista, tales como:

1. tamaño de la lista
2. lista vacía
3. fin de la lista
4. insertar en una lista ordenada
5. ordenar una lista desordenada
6. buscar un elemento de la lista

```
1       public int tamano(){
2           int total = 0;
3           Nodo aux = raiz;
4           while(aux!=null){
5               total++;
6               aux = aux.getSig();
7           }
8           return total;
9       }
```

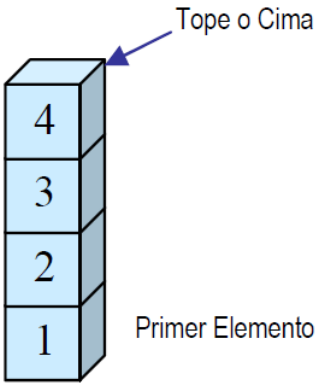
7.5.5. Pilas

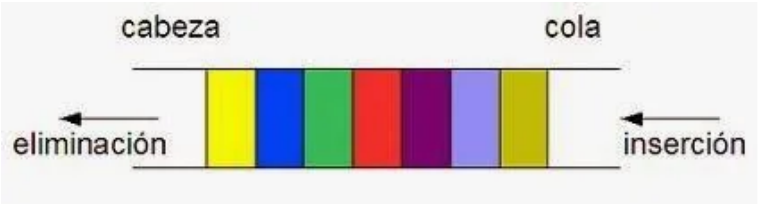
Una pila es una particularización de las listas enlazadas y se puede definir como una estructura de datos en la cual el último elemento que se agregó será el primero en ser eliminado. Es una estructura LIFO (Last In First Out – Último en llegar es el primero en salir).

Las pilas cuentan con 2 operaciones principales, conocidas como **PUSH**, insertar un elemento en la pila, y **POP**, obtener el elemento del tope de la pila.

7.5.6. Colas

Una cola es una estructura de datos similar a una lista con restricciones especiales. Los elementos son agregados en la parte posterior de la cola y son eliminados por el frente. Esta estructura





es llamada FIFO (First In, First Out: el primero que llega, es el primero en salir).

las colas cuentan con 2 operaciones principales, conocidas como **Enqueue(Agregar)**, agregar un elemento al final de la cola, y **Dequeue (Eliminar)**, que elimina y devuelve el elemento que está al inicio de la cola.

8

CAPITULO

Capitulo 8 - Estructuras de Datos No Lineales

8. Estructuras de datos No Lineales

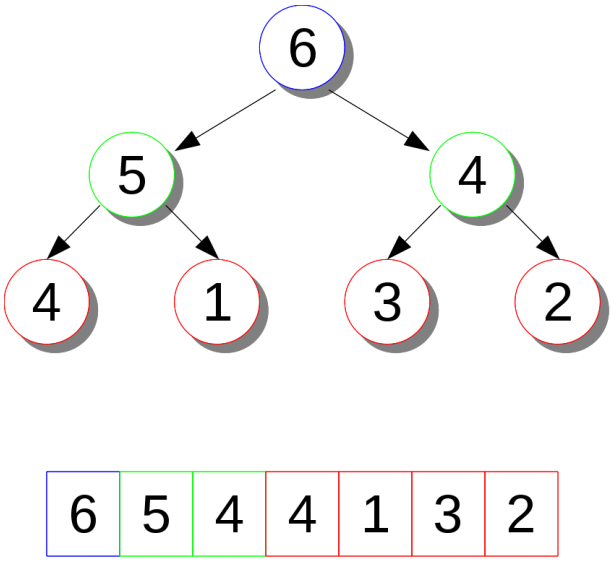
Las estructuras de datos no lineales son fundamentales en programación, pues son utilizadas para modelar una amplia variedad de problemas y escenarios, desde representar relaciones de parentesco en genealogía hasta la organización de datos en bases de datos distribuidas.

Las estructuras de datos no lineales son organizaciones de datos donde los elementos no están dispuestos en una secuencia lineal como en un arreglo o una lista. En lugar de ello, los datos se organizan de manera jerárquica o en redes, lo que permite representar relaciones complejas entre sus elementos.

Algunos ejemplos comunes de estructuras de datos no lineales incluyen:

1. **arboles:**, cada elemento tiene uno o mas elementos hijos, y estos a su vez pueden tener sus propios hijos. Ejemplos incluyen arboles binarios, arboles de búsqueda binaria, arboles AVL, etc.
2. **Grafos:** los elementos (llamados nodos o vértices) están conectados por arcos o bordes. Los grafos pueden ser dirigidos o no dirigidos, dependiendo de si las conexiones tienen una dirección especifica.
3. **Heaps:**, son estructuras de datos especiales basadas en arboles que cumplen ciertas propiedades de orden, como mencionar a los heaps binarios o heaps de Fibonacci.

8.1. Arboles: Conceptos y Ejemplos



Un árbol es una estructura de datos no lineal que organiza elementos jerárquicamente en nodos en la que cada nodo puede apuntar a uno o varios nodos. Cada nodo puede tener hijos, y un único nodo (la raíz) es el punto de inicio. También podemos tener una definición recursiva de esta estructura: un árbol es una estructura compuesta por un dato y varios árboles.

En relación con los nodos, debemos definir algunos conceptos como:

- **Nodo**: Elemento individual de un árbol que contiene datos y referencias a sus hijos.
- **Nodo hijo**: cualquiera de los nodos apuntados por uno de los nodos del árbol.
- **Nodo padre**: nodo que contiene un puntero o referencia al nodo actual.

En cuanto a la posición dentro del árbol, tenemos:

1. **Nodo Raíz**: El nodo principal del árbol y es el único que no tiene padre. Este es el nodo que usaremos para acceder al árbol.
2. **Nodo Hoja**: Nodos que no tiene hijos.
3. **Nodo rama**: estos son los nodos que no pertenecen a ninguna de las dos categorías anteriores

Otras definiciones básicas:

- **Orden**: es el número potencial de hijos que puede tener cada elemento del árbol. De este modo, diremos que un árbol en el que cada nodo puede apuntar a otros dos, es de orden dos, si puede apuntar a tres, es de orden tres, etc.
- **Grado o aridad**: es el número de hijos que tiene el elemento con más hijos dentro del árbol.
- **Nivel**: se define para cada elemento del árbol como la distancia a la raíz, medida en nodos. El nivel de la raíz es cero, y el de sus hijos uno, así sucesivamente.
- **Altura**: la altura de un árbol se define como el nivel del nodo de mayor nivel. Como cada nodo de un árbol puede considerarse a su vez como la raíz de un árbol, también podemos hablar de altura de ramas.
- **Árbol Completo**: Todos los niveles están llenos excepto, quizás, el último, y sus nodos están alineados a la izquierda.
- **Árbol Balanceado**: Un árbol en el que la diferencia de alturas entre sub-árboles es como máximo 1 para todos los nodos.

8.2. Arboles binarios

Los árboles de orden dos o aridad 2 son bastante especiales; estos árboles se conocen también como árboles binarios.

Un árbol binario requiere de una estructura de NODO que permita almacenar el dato correspondiente, además de una referencia al hijo izquierdo y una más al hijo derecho. El árbol tendrá una referencia al nodo raíz, a partir del cual se podrá acceder al resto de los nodos de la estructura.

8.3. Representación de un árbol binario en JAVA

Definición de la clase Nodo de árbol.

```
1 public class Nodo {
2     private int dato;
3     private Nodo izq;
4     private Nodo der;
```

```
5
6       public Nodo(int dato){
7           this.dato = dato;
8       }
9
10      public int getDato() {
11          return dato;
12      }
13
14      public void setDato(int dato) {
15          this.dato = dato;
16      }
17
18      public Nodo getIzq() {
19          return izq;
20      }
21
22      public void setIzq(Nodo izq) {
23          this.izq = izq;
24      }
25
26      public Nodo getDer() {
27          return der;
28      }
29
30      public void setDer(Nodo der) {
31          this.der = der;
32      }
33  }
```

Definición de la clase Árbol.

```
1   class ArbolBinario {
2       Nodo raiz;
3
4       public ArbolBinario() {
5           this.raiz = null;
6       }
7
8       public void guardar(int dato) {
9           if(raiz == null)
10              raiz = new Nodo(dato);
11          else
12              guardarRecursivo(raiz, dato);
13      }
14      ...
15  }
```

8.4. Operaciones básicas con arboles

Salvo que trabajemos con árboles especiales, las inserciones serán siempre en referencias de nodos hoja o en referencias libres de nodos rama. Con estas estructuras no es tan fácil genera-

lizar, ya que existen muchas variedades de árboles.

Tenemos las siguientes operaciones básicas:

- Añadir o insertar elementos.
- Recorrer el árbol.
- Buscar o localizar elementos.
- Borrar elementos.

Destacar también que los algoritmos de inserción y borrado dependen en gran medida del tipo de árbol que estemos implementando.

8.4.1. Recorrido

El modo evidente de moverse a través de las ramas de un árbol es siguiendo las referencias, del modo similar al que se realizaba a través de las listas.Esos recorridos dependen en gran medida del tipo y propósito del árbol, pero hay ciertos recorridos que usaremos frecuentemente. Se trata de aquellos recorridos que incluyen todo el árbol.

Definiremos tres formas de recorrer un árbol completo, y las tres se suelen implementar mediante recursividad. En los tres casos se sigue siempre a partir de cada nodo todas las ramas una por una. Lo que diferencia estos distintos métodos de recorrer el árbol no es el sistema de hacerlo, sino el momento que elegimos para procesar el valor de cada nodo con relación a los recorridos de cada una de las ramas.

Pre-orden:
En este tipo de recorrido, el valor del nodo se procesa antes de recorrer las ramas:

```
1 public void preOrden(Nodo tmp) {
2
3 }
```

In-orden:
En este tipo de recorrido, el valor del nodo se procesa después de recorrer la primera rama y antes de recorrer la última.

```
1 public void inOrden(Nodo tmp) {
2
3 }
```

Post-orden:
En este tipo de recorrido, el valor del nodo se procesa después de recorrer todas las ramas:

```
1 public void postOrden(Nodo tmp) {
2
3 }
```

8.5. Arboles ordenados

Un árbol ordenado, en general, es aquel a partir del cual se puede obtener una secuencia ordenada siguiendo uno de los recorridos posibles del árbol: in-orden, pre-orden o post-orden. En estos árboles es importante que la secuencia se mantenga ordenada, aunque se añadan o se eliminen nodos.
Existen varios tipos de árboles ordenados:

- **árboles binarios de búsqueda (ABB):** son árboles de orden 2 que mantienen una secuencia ordenada si se recorren en in-orden.
- **árboles AVL:** son árboles binarios de búsqueda equilibrados, es decir, los niveles de cada rama para cualquier nodo no difieren en más de 1.
- **árboles perfectamente equilibrados:** son árboles binarios de búsqueda en los que el número de nodos de cada rama para cualquier nodo no difieren en más de 1. Son por lo tanto árboles AVL también.
- **árboles 2-3:** son árboles de orden 3, que contienen dos claves en cada nodo y que están también equilibrados. También generan secuencias ordenadas al recorrerlos en in-orden.
- **árboles-B:** caso general de árboles 2-3, que para un orden M, contienen M-1 claves.

CAPITULO 9

Bibliografía

[1] Mark Allen Weiss, *Data Structures and Algorithm Analysis in Java*, 3rd Edition, Pearson, 2013. Un libro detallado que abarca conceptos avanzados de estructuras de datos, incluidos árboles y grafos, con implementaciones prácticas en Java.

[2] Robert Sedgewick y Kevin Wayne, *Algorithms*, 4th Edition, Addison-Wesley, 2011. Incluye explicaciones claras y ejemplos de algoritmos clásicos que utilizan recursividad y backtracking, además de una introducción a grafos.

[3] Donald E. Knuth, *The Art of Computer Programming*, Volumen 1-3, Addison-Wesley, 1997. Una obra clásica que cubre técnicas fundamentales de programación, incluidos árboles y estructuras de datos avanzadas.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein, *Introduction to Algorithms*, 3rd Edition, MIT Press, 2009. Este libro cubre una amplia gama de algoritmos, con capítulos específicos sobre árboles, grafos y técnicas como recursividad y backtracking.

[5] Robert Lafore, *Data Structures and Algorithms in Java*, 2nd Edition, Sams Publishing, 2002. Un enfoque práctico para aprender estructuras de datos, como árboles y grafos,

con código fuente en Java.

[6] Alfred V. Aho, John E. Hopcroft y Jeffrey D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983. Proporciona una sólida base teórica en estructuras de datos, incluidas aplicaciones de árboles y grafos.

[7] Gilles Brassard y Paul Bratley, *Fundamentals of Algorithmics*, Prentice Hall, 1996. Explica conceptos avanzados de algoritmos, incluyendo recursividad y estrategias de búsqueda como el backtracking.

[8] Michael T. Goodrich y Roberto Tamassia, *Data Structures and Algorithms in Java*, 4th Edition, Wiley, 2006. Proporciona una excelente cobertura de estructuras de datos, árboles y algoritmos de grafos, con ejemplos en Java.