Elementos de programación y Estructuras de Datos

Texto Guía

Elaborado por: Helder Octavio Fernandez Guzman

helder.fernandez@gmail.com

Si encuentra algún problema o error en el texto, puede consultar la sección 1.6 por posibles soluciones o contactarme a helder.fernandez@gmail.com

Happy LaTeX-ing!

Índice general

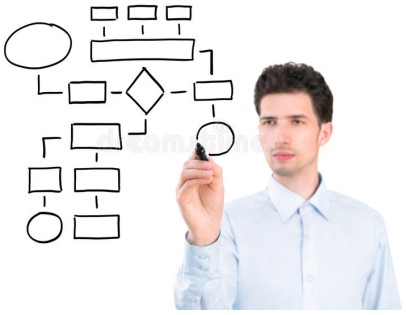
Este texto plantea en sus contenidos técnicas de recursividad, administración de la memoria durante la ejecución de un programa, características de la POO, y estructuras de Datos.

La totalidad del contenido de este texto y código fuente son de acceso público y no tienen fines de lucro, está desarrollado con mucho cariño para todos aquellos estudiantes que hacen grata esta tarea de enseñar.

CAPITULO 1

Capítulo 1 - Introduccion

1. Introducción



¡Bienvenido! este material de apoyo esta elaborado de tal forma que se busca que la experiencia del curso sea lo mas 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 es desarrollado 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 éste curso?

Un curso introductorio de programación provee las bases para entender como programar. Pero, programar de forma más elaborada no sólo 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 ésta plataforma que ya pronto llegara 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 ultimos avances en tecnologia 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, microordenadores (como raspberry) e incluso provee librerías para poder interactuar con plataformas de hardware como Arduino.

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

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

1.3. ¿Dónde obtener JAVA?

Para empezar a programar necesito el software, ¿donde 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

(reorganizar) Se ha organizado el documento en 7 capítulos en los cuales te vas a encontrar lo siguiente:

Capítulo 1: Introducción Se plantean y justifican las razones de la materia y los contenidos planteados, referencias a recursos necesarios en la materia: lenguaje, editores y código fuente.

Capítulo 2: Conceptos Fundamentales. Un repaso de alto nivel de conceptos que se aprenden en la materia de Introducción a la programación, pero que son muy relevantes para plantear nuevos conceptos descritos en el capítulo 3.

Capítulo 3: Elementos de Programación. Se plantea la recursividad como una alternativa algorítmica que sirve para resolver un conjunto de problemas de manera muy abstracta. En un nivel mas avanzado se estudia la recursividad de ejecución no lineal para trabajar en permutaciones y combinaciones de manera mas eficiente (usando funciones de poda) mecanismo por el cuál la técnica obtiene su nombre: *Backtracking*.

Capítulo 4: Administración de la Memoria. Se plantean conceptos muy relacionados a los espacios de memoria usados durante la ejecución de un programa para entender como hacer un uso óptimo de los mismos. Se abarcan conceptos como Recolector de basura, palabra reservada *static*, this y super, variables, referencias y tipos de operaciones.

Capítulo 5: Programacion Orientada a Objetos Conceptos OO clave de JAVA: Herencia, Poliformismo, Genéricas.

Capítulo 6: Estructuras de Datos Lineales variables y referenciasvariables y referenciasvariables

y referenciasvariables y referenciasvariables y referenciasvariables y referenciasvariables y referenciasvariables y referencias.

Capítulo 7: Estructuras de Datos No Lineales variables y referenciasvariables y referenciasvariables y referenciasvariables y referenciasvariables y referenciasvariables y referenciasvariables y referenciasvariables y referencias

1.5. Código Fuente del curso

Si te interesa obtener el código fuente que se usa en el curso lo puedes encontrar en :
`https://AquiEnGITParaHelder/`

1.6. Errores en el Texto y el código

Si crees que algo debe ser corregido puedes contactarme al correo: helder.fernandez@gmail.com. Toda la retroalimentación es bienvenida y como agradecimiento serás mencionado en la siguiente versión del texto.

2

CAPITULO

Capítulo 2 - Conceptos Fundamentales

2. Conceptos fundamentales

En este capítulo a manera de repaso vamos a presentar algunos conceptos que seguramente conoces y has visto durante el curso introductorio a programación, hacemos explícito el contenido para no dar por hecho que los conceptos planteados son de dominio de un programador.

2.1. Sistema de Tipos en JAVA

Podemos ver al Sistema de Tipos como una mecanismo para representar la información a ser manipulada o procesada en el lenguaje de programación, una de las razones:

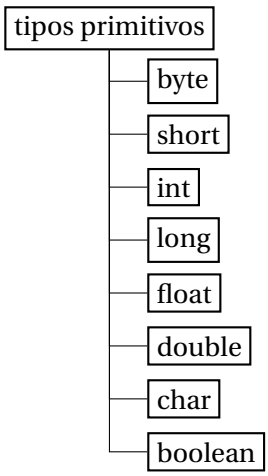
Permite al compilador validar la correctitud del programa a nivel sintáctico pues al ser JAVA un lenguaje fuertemente tipado las transacciones o transferencia de datos e información entre llamadas a funciones y operaciones de asignación y lectura debe ser entre variables y valores del mismo tipo.

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 mas eficiente de acuerdo al tipo de cálculo que se hará con ésta información.

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

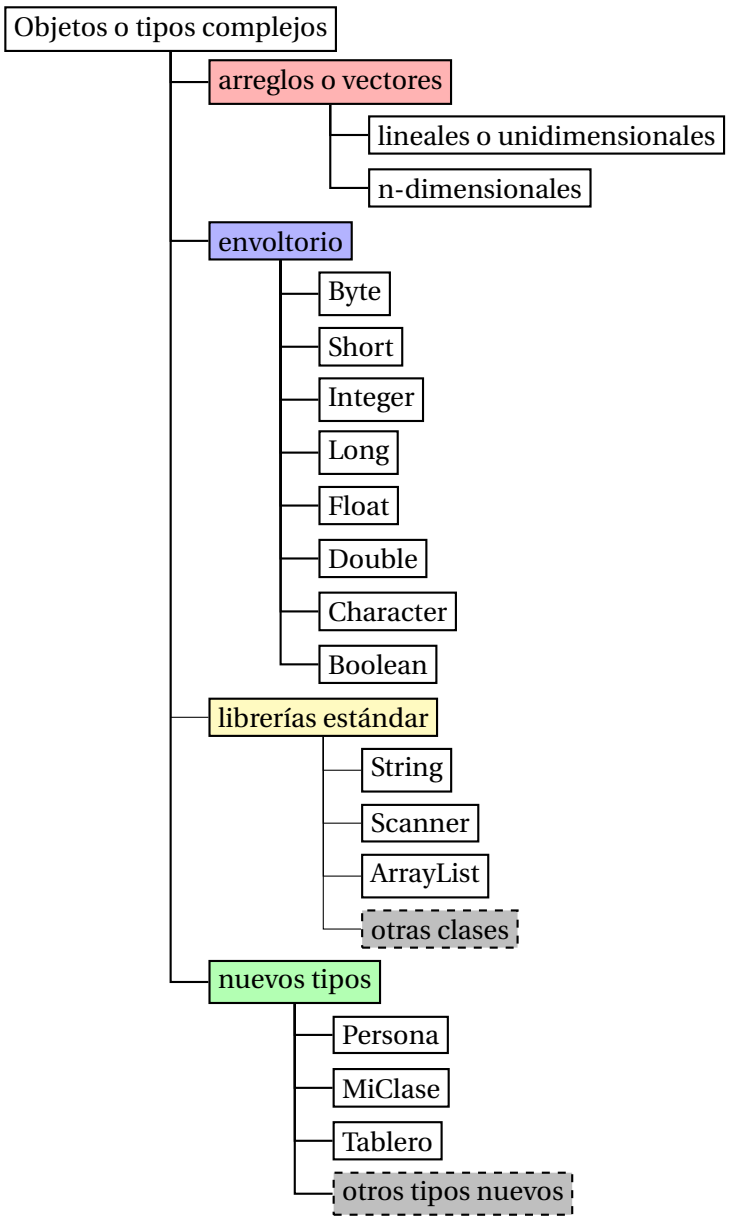
2.1.1. Tipos Primitivos

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 mayoria 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.



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.



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 cuál 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:

```java
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:

```java
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 **nullabilidad**, ya que **los tipos primitivos no pueden ser nulos**.

Hay ocho tipos 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:

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

Con unboxing, se puede hacer lo siguiente:

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

2.3.2. Algunos métodos útiles de los tipos envoltorio

Cada tipo envoltorio tiene un conjunto de métodos de utilidad para trabajar con valores primitivos y objetos 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 comentan 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 encontraras mas librerías así como todos sus métodos y funciones.

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):

```
import java.util.Scanner;
...
Scanner s = new Scanner(System.in);
int b=0;
while (!s.hasNextInt())
s.next();
b= s.nextInt();
System.out.println(b);
```

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

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 cuál 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>.metodo.

- *char charAt(int index)* Devuelve el carácter que está 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 además 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 metodo específico: *Math.<metodo>*, 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 número 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 número pasado como parámetro. Si el número es double devuelve long, si es float devuelve int.

- *static double ceil(double a)* Trunca el número hacia arriba (Math.ceil(3.2) devuelve 4.0).

- *static double floor(double a)* Trunca el número.

- *static double sin(double a)* Devuelve el seno del ángulo 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.

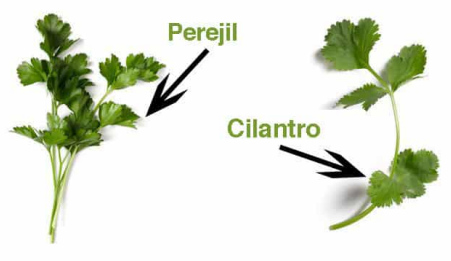
- *static double exp(double a)* Eleva el número **e** a a.

- *static double sqrt(double a)* Raíz cuadrada de a.

- *static double cbrt(double a)* Raíz cúbica de a.

- *static double random()* Devuelve un número aleatorio entre 0.0 (incluido) y 1.0 (no incluido)

2.5. Funciones, métodos y procedimientos



Los procedimientos, funciones y métodos son bloques de código que se pueden utilizar para ejecutar tareas específicas en un programa JAVA. Cada uno se utiliza de forma ligeramente diferente, pero todos pueden ayudar a mejorar la eficiencia y modularidad de un programa. Las funciones, métodos y procedimientos son una parte fundamental de la programación orientada a objetos.

2.5.1. Procedimientos

Un procedimiento es un bloque de código que se ejecuta sin devolver un valor. En JAVA, un procedimiento se define utilizando la palabra clave **void**. Por ejemplo:

```
1 public static void miProcedimiento() {
2     // Bloque de codigo
3 }
```

2.5.2. Funciones

Una función es un bloque de código que se ejecuta y devuelve un valor. En JAVA, se define una función con la palabra clave que indica el tipo de valor que devuelve la función. Por ejemplo:

```
1 public static int miFuncion() {
2     // Bloque de codigo
3     return 0;
4 }
```

2.5.3. Métodos

Un método es una función que está asociada con un objeto o una clase y que se invoca a través de una instancia o referencia a la clase.

En JAVA, por ejemplo, los métodos se definen dentro de una clase y pueden acceder a los atributos de una clase y pueden ser sobrescritos o heredados, lo que no es posible con las funciones.

```
1
2 public class Persona {
3     private String nombre;
```

```
4       public Persona(String nombre) {
5           this.nombre = nombre;
6       }
7
8       public void saludar() {
9           System.out.println("Hola, mi nombre es " + this.nombre);
10      }
11  }
12
13  // En el mismo archivo, se puede usar
14  // la definicion clase Persona desde la funcion main
15  // instanciando un objeto e invocando el metodo saludar
16
17  public class Main {
18      public static void main(String[] args) {
19          Persona persona1 = new Persona("Juan");
20          persona1.saludar();
21
22          Persona persona2 = new Persona("Maria");
23          persona2.saludar();
24      }
25  }
```

En resumen, una función es un término más general que se aplica a cualquier subrutina que realice una tarea y devuelva un valor, mientras que un método es un término más específico que se refiere a una función asociada con un objeto o una clase.

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

3

CAPITULO

Capítulo 3 - Programacion Orientada a Objetos

3. 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.

3.1. 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 el reuso 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 esté ejecutando. El polimorfismo permite escribir código más genérico y flexible, ya que se pueden tratar distintos objetos de manera uniforme a través de interfaces comunes.

3.2. 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.

3.2.1. 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. **Sobreescritura de métodos:** la sobreescritura 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.

3.2.2. Desventajas de la herencia en JAVA

1. **Complejidad:** la herencia puede hacer que el código sea más complejo y más 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.

3.2.3. 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 (además de ya tener los campos y métodos **heredados** de la superclase).

4. **Reusabilidad:** 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.

3.2.4. Cómo utilizar la herencia en JAVA

Para establecer una relación de herencia entre dos clases en JAVA, se utiliza la palabra clave ëxtends:̈ 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.

3.2.5. Consideraciones importantes de la herencia en JAVA

1. **Superclase predeterminada:** excepto la clase **Object** , que no tiene superclase, cada clase tiene una y sólo una superclase directa (herencia única). En ausencia de cualquier otra superclase explícita, cada clase es implícitamente una subclase de la clase Object.

2. **La superclase sólo puede ser una:** una superclase puede tener cualquier número de subclases. Pero una subclase sólo 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.

3.3. 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 su tipo.

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.

3.3.1. Qué 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.

3.3.2. 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

3.3.2.1. 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  }
```

3.3.2.2. Polimorfismo en tiempo de ejecución en Java

También se conoce como resolución dinamica 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 sobre-escritura 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 ejecucion 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.
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.

3.3.2.3. 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.

3.3.3. 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.

3.3.4. Desventajas del polimorfismo en Java

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

4

CAPITULO

Capítulo 4 - Elementos de programacion

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 solucion o representacion puede ser usando iteradores, estos se pueden resolver de forma recursiva si es que sus características asi lo permiten. En este Apartado estudiamos una tecnica de solución de problemas que justamente se enfoca en descomponer el problema en subproblemas apoyandose en el uso de funciones y no asi en el uso de iteradores, a continuacion se describe esta técnica.

4. La recursividad

En el contexto de la programación, se denomina recursividad al proceso en el que una función se llama a sí misma directa o indirectamente, ésta 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 Arboles Inorden/Preorden/Postorden, Búsqueda en profundidad, Permutaciones, etc.

Una función recursiva resuelve un problema particular llamando a una copia de sí misma y resolviendo subproblemas 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 mas 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 éste caso es el candidato ideal para ser un caso base, pues ya no se puede expresar como una descomposición del problema en problemas mas pequeños.

En base a estas expresiones planteamos una función, de tal manera que describe a 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 si misma dentro de su definición, éste 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 linea 6 observa la llamada a la función f (la misma), para ese momento ya se da por valida 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 se la ultima 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:

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

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 la misma definición pero para evaluar el siguiente elemento, de ésta forma se lograría visitar todos los elementos del arreglo hasta que no queda alguno pendiente, en cuyo caso el valor del índice sera mayor al rango de elementos (y tomaremos ese caso como base):

```
{
    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 sólamente 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 la misma definición pero para evaluar el siguiente elemento, de ésta forma se lograría visitar todos los elementos de la cadena hasta que no queda alguno pendiente, en cuyo caso el valor del índice sera 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
```

```
5   }
6
7   private static void imprimirVertical(String texto,int i)
8   {
9       if(i>=texto.length()) //el indice ya esta fuera del rango
10          return;           //no quedan mas elementos por evaluar
11      else
12      {
13          System.out.println(texto.charAt(i)); // elemento a evaluar
14          imprimirVertical(texto,i+1); //siguiente elemento a evaluar
15      }
16  }
```

4.2.3.2. Usando diferentes instancias de la cadena (sub cadenas)

En este escenario, cada instancia de ejecución de la función recursiva evaluará el primer elemento de la cadena, y realizará la llamada la misma definición pero pasa como información una subcadena que ya no contiene al elemento evaluado, de ésta 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, que pasa si a partir de una parte de la solución (o solución parcial) queremos intentar construir la solución final por otro camino, ahí es donde surge la técnica del Backtracking la cual permite usar la recursividad pero con otro objetivo.

4.3.1. 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 sólo 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.

4.3.2. 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.

4.3.2.1. Diseño del Algoritmo de Backtracking

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

1. Buscar una representación del tipo $X1, X2, \ldots, Xn$ para las soluciones del problema.
2. Identificar las restricciones implícitas y explícitas 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 $Bk\,X1, X2, \ldots, Xk$ para eliminar ramas del árbol que puedan derivar en soluciones poco deseables o no deseadas.

4.3.2.2. 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:

```
1  static boolean functionBacktracking(vectorEntrada, vectorSolucion)
2  {
3    // Caso Base 1: podar
4    if (!esSolucionParcial)
5    {
6      return false;
7    }
8    // Caso Base 2: Aceptar la solucion (completa)
9    else if (esSolucionFinal)
10   {
11     print(vectorSolucion);
12     return true;
13   }
14   else
15   {
16     // Paso: navegar sobre las posibles soluciones
17     for (n in vectorEntrada)
18     {
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

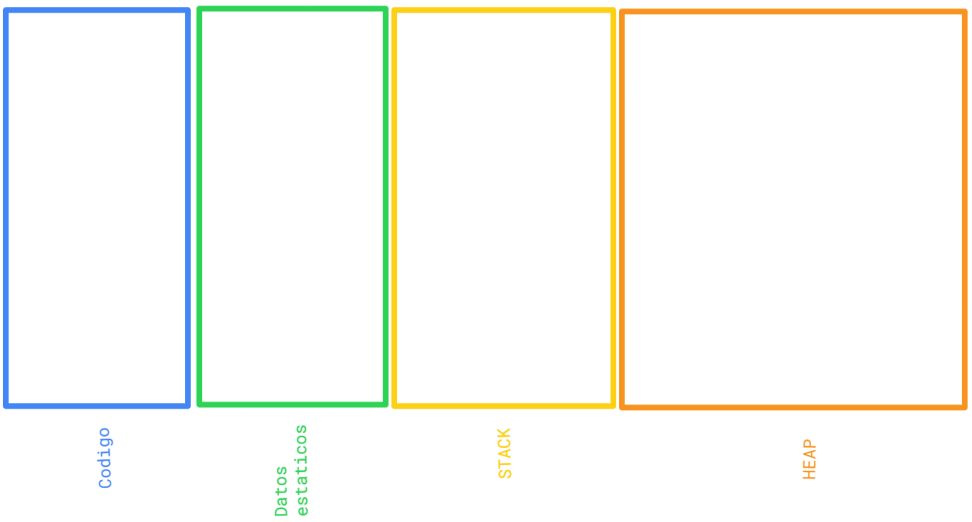
CAPITULO

Capítulo 5 - Administración de la memoria

5. 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.

5.1. Áreas de la memoria durante la ejecución



Entonces de manera general podemos plantear una organización de la memora asignada a la ejecución de un programa la cual podría considerarse la forma mas 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 metodos aislando cada funcion e identificando a la funcion activa en un determinado instante de tiempo.

4. HEAP o memoria dinámica. Donde se crean los datos de tamaños no predecibles o irregulares y el lugar que es monitoreado por el Recolector de Basura.

5.2. Gestión de memoria dinámica

5.2.1. 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 metodo 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:

```
1  public class EjemploRecolector {
2      public static void main(String[] args) {
3          EjemploRecolector obj = new EjemploRecolector();
4          obj = null; // la referencia al objeto se pierde
5          System.gc(); // se invoca el recolector de basura
6      }
7
8      @Override
9      protected void finalize() throws Throwable {
10         System.out.println("Objeto eliminado");
11     }
12 }
```

En este ejemplo, se crea un objeto 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.

5.2.2. 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 monitorear 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.

5.3. Miembros estáticos

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

5.3.1. 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

5.3.2. 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.

```
1  class Ejemplo {
2    // variables estaticas
3    static int a; //declarado
4    static int b = 4; //declarado e inicializado
5
6    // bloque estatico
7    static {
8      System.out.println("bloque estatico inicializado");
9      a = b * 8; //inicializado
10   }
11
12   public static void main(String[] args) {
13       System.out.println("valor de a : "+ a);
```

```
14          System.out.println("valor de b : "+ b);
15      }
16  }
```

5.3.3. 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:

```
1  class Ejemplo {
2    // variable estatica
3    static int x = inicializar();
4    // bloque estatico
5    static {
6      System.out.println("evaluando el bloque estatico");
7    }
8    // funcion\metodo estatico
9    private static int inicializar() {
10     System.out.println("evaluando inicializar");
11     return 8;
12   }
13   // metodo estatico main
14   public static void main(String[] args)
15   {
16     System.out.println("evaluando la funcion principal!");
17       System.out.println("valor de x : " + x);
18     }
19 }
```

5.3.4. Métodos estáticos

Cuando un método se declara con la palabra reservada *static*, es conocido como método estático. 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 esto o super de ninguna manera.

A continuación se muestra el programa JAVA para demostrar 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");
    }

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

5.3.5. 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.

```
public class Ejemplo {
  private static String str = "valor comun para los objetos";
  // Clase estatica
  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  }
```

5.3.6. 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 más 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 fábrica 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.

5.4. 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.

5.4.1. 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 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 clase actual

```
1  class Ejemplo {
2      // instancia de variable
3      int a = 10;
4
5      // variable estatica
6      static int b = 20;
7
8      void inicializar()
9      {
10             this.a = 100;
11             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 }
```

5.4.2. 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    // instance variable
3    int a = 10;
4
5    // static variable
```

```
6     static int b = 20;
7  }
8
9  class Base extends Padre {
10     void imprimir()
11     {
12         System.out.println(super.a);
13
14         System.out.println(super.b);
15     }
16
17     public static void main(String[] args)
18     {
19         new Base().imprimir();
20     }
21 }
```

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 clase principal inmediata.
- Para referirse al método de clase padre inmediato.
- Para hacer referencia al constructor de la clase principal inmediata.

5.5. operaciones con referencias o enlaces

TBD

6

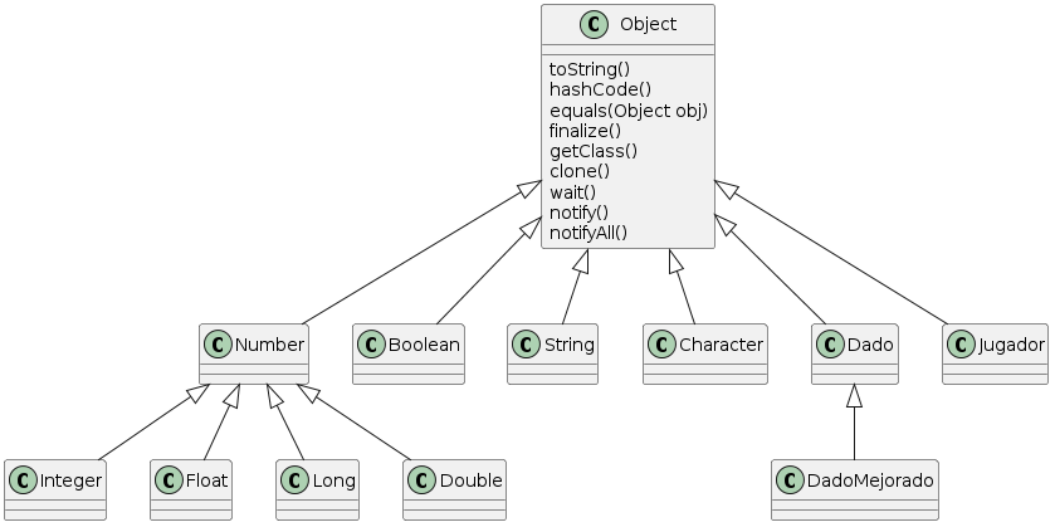
CAPITULO

Capítulo 6 - Programación Genérica

6. Programación Genérica

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.



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 sobrescribirlo 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 esté 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 de 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) sean 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 más de unparámetro de tipo, separados por una coma. Las clases que aceptan uno o más parámetros se conocen como clases parametrizadas o tipos parametrizados.

- **Métodos genéricas:** Un método genérico tiene parámetros de Tipo que se citan por el tipo actual. Esto permite utilizar el método genérico de una manera más 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
7  class Main {
8    public static void main(String[] args)
9    {
10     MiClase<Integer> c1 = new MiClase<Integer>(15);
```

```
11      System.out.println(c1.getA());
12
13      MiClase<String> c2 = new MiClase<String>("Hola!");
14      System.out.println(c2.getObject());
15    }
16  }
```

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
8    public static void main(String[] args)
9    {
10     mostrarGenerico(11);
11
12     mostrarGenerico("Hola desde main!");
13
14     mostrarGenerico(1.0);
15   }
16 }
```

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 compilació. Es preferible identificar los problemas en código en tiempo de compilación en lugar de en tiempo de ejecución.

3. **Conversión de tipo:** La conversion 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 Lineales

7. Estructuras de datos

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 estructuras de datos que especifica los tipos de datos almacenados, las operaciones definidas sobre esos datos y los tipos de parámetros de esas operaciones.

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

Un **TDA** define lo que cada operación debe hacer, más no como. En un lenguaje de programación como JAVA un TDA puede ser expresado como una interface, es decir una lista de declaraciones de métodos.

Un **TDA** es materializado por una estructura de datos concreta, en JAVA, 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. Al contrario de las interfaces, las clases especifican como las operaciones son ejecutadas (implementación).

7.2. Estructura de Datos

En programación, una estructura de datos es una forma de organizar 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, y un conjunto de operaciones que se pueden realizar sobre él. Las operaciones básicas son:

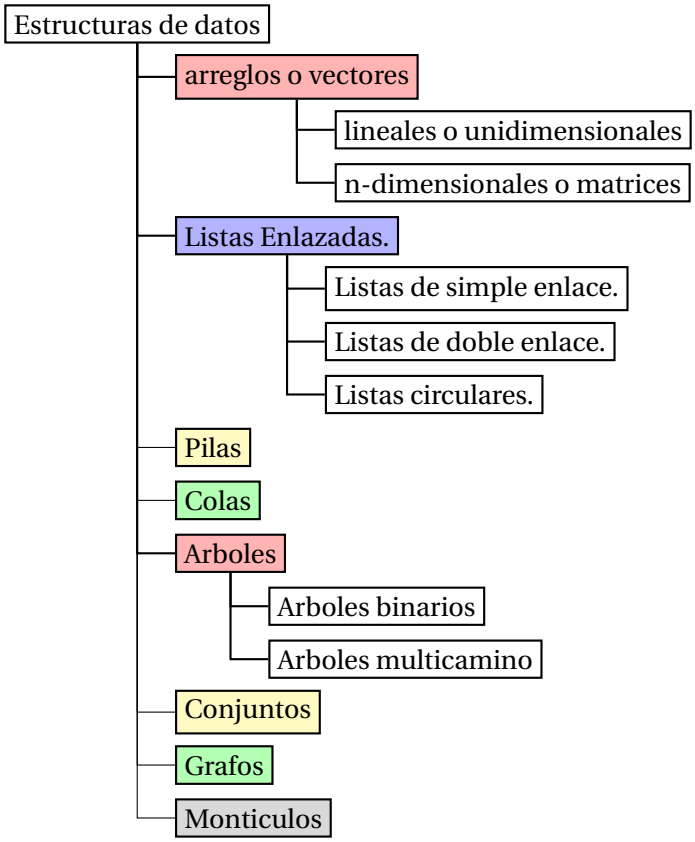
- Alta, adicionar un nuevo valor a la estructura.
- Baja, borrar un valor de la estructura.
- Búsqueda, encontrar un determinado valor en la estructura para se realizar una operación con este valor, en forma SECUENCIAL o BINARIO (siempre y cuando los datos estén ordenados).

Otras operaciones que se pueden realizar son:

- Ordenamiento, de los elementos pertenecientes a la estructura.
- Apareo, dadas dos estructuras originar una nueva ordenada y que contenga a las apareadas.

Cada estructura ofrece ventajas y desventajas en relación a la simplicidad y eficiencia para la realización de cada operación. De esta forma, la elección de la estructura de datos apropiada para cada problema depende de factores como la frecuencia y el orden en el que se realiza cada operación sobre los datos.

Algunas estructuras de datos utilizadas en programación son:



7.3. Acceso directo y secuencial a los datos

- Secuencial. Para acceder a un objeto se debe acceder a los objetos almacenados previamente en el archivo. El acceso secuencial exige elemento a elemento, es necesario una exploración secuencial comenzando desde el primer elemento.

- Directo o Aleatorio. Se accede directamente al objeto, sin recorrer los anteriores. El acceso directo permite procesar o acceder a un elemento determinado haciendo una referencia directamente por su posición en el soporte de almacenamiento.

7.4. Listas enlazadas

Definición Las listas enlazadas, son probablemente, la segunda estructura de almacenamiento de propósito general más comúnmente utilizadas, después de los arreglos.
Los "arreglos.º vectores presentan ciertas desventajas:

- en un arreglo desordenado, la búsqueda es lenta
- en un arreglo ordenado, la inserción es lenta
- en ambos casos, la eliminación es lenta
- el tamaño de un arreglo no puede cambiar después que se creó

Una lista enlazada 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 para otras estructuras de almacenamiento como pilas y colas.
La ventaja más evidente de utilizar estructuras enlazadas, es que permite optimizar el uso de la memoria, pues no desperdiciamos el espacio de localidades vacías: La desventaja más grande de las estructuras 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 listas más complejas son las listas doblemente enlazadas o las listas circulares, por nombrar algunas.

7.5. Listas desordenadas

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 relación 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: Tope de la Lista

Lógicamente, una lista es tan solo un objeto de una clase Lista que se conforma de un tope (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.
Listas ordenadas 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

- crear un método de ordenamiento de la lista

Operaciones sobre listas Los distintos métodos de la clase Lista deben asegurarnos poder insertar, eliminar u obtener un dato en cualquier posición de la lista. Como se mencionó, una lista se compone de - un tope de la lista, que es un objeto de la clase NodoDeLista - un conjunto de instrucciones que controlen su estructura La clase NodoDeLista debe contener al menos un espacio de almacenamiento de datos y un objeto de la clase NodoDeLista no necesariamente instanciado. La clase NodoDeLista debe ser nuestra clase base, que incorporará los constructores necesarios para inicializar nuestros datos. El objetivo es una estructura de clase que nos represente:
La clase lista contiene:

- un objeto de tipo NodoDeLista que será la primera referencia de nuestra lista
- un grupo de constructores para inicializarla
- un grupo de operaciones para el control de la lista, tales como:
    - insertar un nuevo NodoDeLista
    - buscar un NodoDeLista
    - eliminar un NodoDeLista
    - obtener un NodoDeLista en alguna posición
    - mostrar la lista

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

7.6. Pilas

Una pila, es una particularización de las listas y se puede definir como una estructura en la cual los elementos son agregados y eliminados en el tope de la lista. Es una estructura LIFO (Last In First Out – Ultimo en llegar es el primero en salir).
Las pilas cuentan con 2 operaciones principales, conocidas como PUSH, insertar un elemento en el tope de la pila, y POP, leer el elemento del tope de la pila. Veamos como funcionan estos métodos: PUSH 1.- Crear un NodoDeLista nuevo Tope de la pila 2.- Hacer el siguiente de nuevo igual al tope Tope de la pila 3.- Hacer el tope de la pila igual al nuevo nodo Tope de la pila Se

ha agregado el dato a la pila!! POP 1.- Crear un nodo temporal que se instancia al tope de la pila Tope aux 2.- Hacer el tope de la pila igual al siguiente del temporal Tope 3.- Regresar el valor del nodo temporal antes que desaparezca Tope Implementación de pilas Para trabajar con pilas, utilizaremos la misma base con la que trabajamos para las listas. Dispondremos de 2 clases: Pila y NodoDePila. Con la primera crearemos a la pila y las operaciones sobre ella y con la segunda instanciaremos a cada uno de los nodos de la pila.

7.7. Colas

Definición 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). Considere la siguiente figura:
A nivel de lógico, podemos representar a una cola de la siguiente manera: último primero Construcción de colas Implementar colas involucra el uso clases similares a una lista. Las operaciones sobre esta estructura son idénticas: insertar y eliminar, con las consideraciones pertinentes. Insertar Insertar un dato en una cola es muy similar a hacerlo en una pila o una lista, la diferencia es que tendremos que hacerlo por el final de la cola. A este proceso se le llama encolar. 1.- Cola inicial último primero 2.- Creamos el nuevo nodo último primero 3.- Hacemos que último apunte al nuevo nodo último primero Eliminar Para extraer un dato de la cola, es necesario modificar el valor del nodo primero, de manera que deje de apuntar al primer elemento de la cola. De la misma manera que con las pilas, se extrae el valor antes de que se pierda. Se hace que el nodo primero ahora apunte al nuevo elemento (anterior de la cola). 1.- último primero 2.- último último 3.- último primero

8

CAPITULO

Capitulo 8 - Estructuras de Datos NO Lineales

8. Estructuras de datos No Lineales

8.1. 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.

CAPITULO 9

Bibliografía

CAPITULO 9

Bibliografía

[1] Academic and Research Computing. *Text Formatting with LaTeX A Tutorial*. NY, April 2007.
http://www.rpi.edu/dept/arc/docs/latex/latex-intro.pdf

[2] Leslie Lamport. *LaTeX for Beginners*. fifth edition, Document Reference: 3722-2014, March 2014.
http://www.docs.is.ed.ac.uk/skills/documents/3722/3722-2014.pdf

[3] Tobias Oetiker, Hubert Partl, Irene Hyna, and Elisabeth Schlegl. *The Not So Short Introduction to LaTeX*. Version 6.3, March 1994.
https://tobi.oetiker.ch/lshort/lshort.pdf

[4] Helmut Kopka and Patrick W. Daly. *A Guide to LaTeX and Electronic Publishing*. Addison-Wesley, fourth edition, May 2003.
https://www2.mps.mpg.de/homes/daly/GTL/gtl_20030512.pdf

[5] Philip Hirschhorn. *Using the exam document class*. Wellesley College, second edition, MA, November 2017.
http://www-math.mit.edu/~psh/exam/examdoc.pdf

[6] *When should I use \input vs \include.*
https://tex.stackexchange.com/questions/246/when-should-i-use-input-vs-include

[7] Overleaf. *Inserting Images.*
https://www.overleaf.com/learn/latex/Inserting_Images

[8] Rice University. *LaTeX Mathematical Symbols.*
https://www.caam.rice.edu/~heinken/latex/symbols.pdf

[9] Overleaf. *Integrals, sum and limits.*
https://www.overleaf.com/learn/latex/Integrals,_sums_and_limits

[10] Boston University. *LaTeX Command Summary.* December 1994.
https://www.bu.edu/math/files/2013/08/LongTeX1.pdf

[11] Overleaf. *Subscripts and superscripts.*
https://www.overleaf.com/learn/latex/Subscripts_and_superscripts

[12] Disquis. *LaTeX Color.* Addison-Wesley, second edition, Reading, MA, 1994.
http://latexcolor.com/

[13] David Woods. *Useful LaTeX Commands.*
https://www.scss.tcd.ie/~dwoods/1617/CS1LL2/HT/wk1/commands.pdf

[14] Overleaf. *Spacing in Math Mode.*
https://www.overleaf.com/learn/latex/Spacing_in_math_mode

[15] Overleaf. *Fractions and Binomials.*
https://www.overleaf.com/learn/latex/Fractions_and_Binomials

[16] Overleaf. *Environments.*
https://www.overleaf.com/learn/latex/Environments

[17] Overleaf. *Margin Notes.*
https://www.overleaf.com/learn/latex/Margin_notes

[18] Art of Problem Solving. *LaTeX:Commands*
https://artofproblemsolving.com/wiki/index.php/LaTeX:Commands

[19] Latex-Project.
https://www.latex-project.org/about/

[20] Tex Stack Exchange. customizing part style with Tikz.
https://tex.stackexchange.com/questions/159551/customizing-part-style-with-tikz

[21] Tex Stack Exchange. How to change chapter/section style in tufte-book?.
https://tex.stackexchange.com/questions/83057/how-to-change-chapter-section-style-in-t
noredirect=1&lq=1