

Diplomado en:

Programación en Java

Guía didáctica N° 4



Formación Virtual

.....educación sin límites

GUÍA DIDÁCTICA N°4

M2-DV59-GU04

MÓDULO 4: ESTRUCTURAS DE DATOS

DIPLOMADO EN PROGRAMACIÓN EN JAVA

© DERECHOS RESERVADOS - POLITÉCNICO DE COLOMBIA, 2018
Medellín, Colombia

Proceso: Gestión Académica

Realización del texto: Diego Palacio, Docente

Revisión del texto: -, Asesor Gramatical

Diseño: Cristian Quintero, Diseñador Gráfico

Editado por el Politécnico de Colombia

ÍNDICE

PRESENTACIÓN.....	4
COMPETENCIAS.....	5
TEMA 1 Estructuras de Datos.....	6
TEMA 2 Datos por Teclado	8
TEMA 3 Listas	17
TEMA 4 Pilas.....	48
TEMA 5 Colas.....	54
TEMA 6 Métodos de Ordenamiento	61
TEMA 7 Recursividad.....	66
ASPECTOS CLAVES	73
REFERENCIAS BIBLIOGRÁFICAS	74

PRESENTACIÓN

La Guía Didáctica N°4 del MÓDULO 4: ESTRUCTURAS DE DATOS, es un material que ha sido desarrollado para el apoyo y orientación del participante en el *Diplomado en Programación en Java*, especialmente, está orientada a la consolidación y/o desarrollo de las habilidades y destrezas necesarias para generar unas adecuadas bases en lo que concierne a la programación en Java.

Como bien conoces, el objetivo principal de este módulo número 4 es introducir al estudiante en todo a lo referente a las estructuras de datos en Java, todos sus componentes, características y conceptos básicos para un adecuado desarrollo idóneo de las bases.

Para ello, se ha organizado esta guía en nueve (9) contenidos temáticos entre ellos: (a) Estructuras de datos, (b) Datos por teclado, (c) Listas, (d) Pilas, (e) Colas, (f) Métodos de ordenamiento e (g) Recursividad.

COMPETENCIAS

Se espera que con los temas abordados en la Guía Didáctica N°4 del MÓDULO 4: ESTRUCTURAS DE DATOS, el estudiante logre la siguiente competencia:



- Comprender las diferentes estructuras de datos para el diseño y almacenamiento de los mismos.

Indicadores de logro:

- Comprende la sintaxis de funcionamiento de las diferentes estructuras de datos en cuanto a: Listas, Pilas y Colas.
- Aplicar correctamente el uso de la recursividad en métodos.
- Conocer los diferentes métodos de ordenamiento y sus funciones.
- Insertar datos por teclados de forma dinámica.

TEMA 1

Estructuras de Datos

Los programadores frecuentemente se encuentran con la necesidad de escribir programas que manipulan una estructura de datos del mismo tipo. Con los conocimientos básicos de programación inmediatamente se piensa que la solución para trabajar con estos datos es utilizar un arreglo, con ventajas y desventajas de estos. Si bien es cierto que así podría ser, no siempre es conveniente utilizar arreglos debido a que en general el acceso a los datos no es por posición, así que se debe buscar otra organización adecuada para los datos terminados de facilidad de manejo de estos por parte del programador y en términos de rapidez de ejecución de las tareas relativas a dicho manejo.

Las estructuras de datos tienen como objetivo facilitar la organización, con el propósito de que la manipulación de ellos sea eficiente. Por eficiencia se entiende la habilidad de encontrar y manipular los datos con el mínimo de recursos tales como tiempo de proceso y espacio en memoria. No es lo mismo hacer un programa para manipular decenas de datos que para miles de ellos.

Conocer y, sobre todo, utilizar las estructuras de datos es esencial para escribir programas que utilicen eficientemente los recursos de la computadora. Existen diversos tipos de estructuras de datos, las hay desde las muy generales y ampliamente utilizadas hasta otras muy especializadas para problemas particulares. La selección de la estructura de datos apropiada permite utilizar la que sea más eficiente para el

problema específico que se desea resolver, con lo cual se optimiza el rendimiento del programa. (fciencias, 2018).

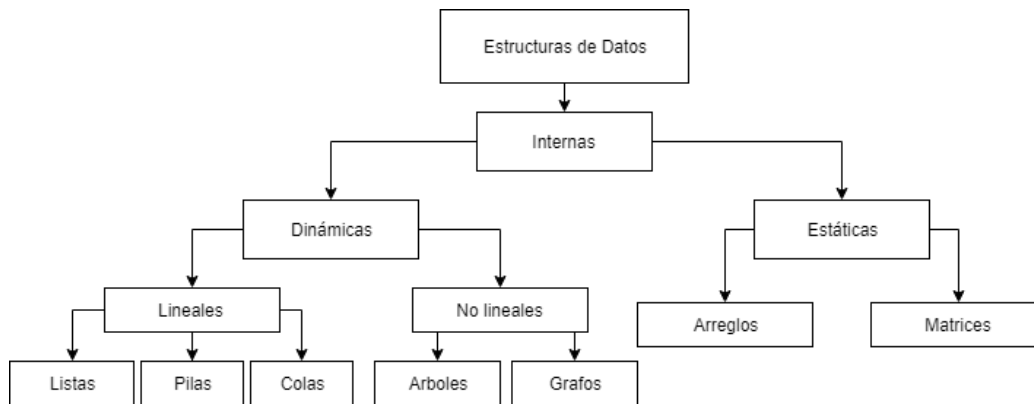


Ilustración 1: Estructuras de datos.

Fuente: Draw.io.

Las principales estructuras que se dictarán en el transcurso de esta guía número 4 son las estructuras dinámicas en las que entran en especial las lineales: Listas, Pilas y Colas, además de todos sus métodos y características que componen estas estructuras.

Hay que tener en cuenta que todos estos tipos de estructuras de datos pueden ser provistos por una librería o creados desde cero. Las estructuras de datos no solo representan la información, también tienen un comportamiento interno, y se rige por ciertas reglas/restricciones dadas por la forma en que está construida internamente.

TEMA 2

Datos por Teclado

La entrada o lectura de datos en Java es uno de los conceptos más importantes y fundamentales al momento de interactuar con los usuarios en los programas, dado que se deja a un lado los datos estáticos definidos previamente y permite el ingreso de datos dinámicos en cada ejecución. La entrada de datos en Java, a diferencia de otros lenguajes es un poco complicada (no demasiado) y existen diferentes formas de hacerlo, unas más complejas que otras. En esta ocasión se tendrán dos maneras sencillas de leer datos para los programas en Java. La primera aplicando las clases **BufferedReader** y **InputStreamReader** ambas de la librería **java.io** y la segunda con la clase **Scanner** de la librería **java.util**.

BufferedReader - InputStreamReader

Son dos clases que permiten la entrada y lectura de datos en Java a partir de la librería **java.io**; con métodos que proporcionan la interacción con los datos.

Para hacer uno de estas clases el proceso a seguir es simple. Véase.

```
package Clase;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Main {

    public static void main(String[] args)
    {

    }

}
```


Ilustración 2: Lectura de datos por teclado - Buffered.

Fuente: Eclipse.

Son dos clases las que se deben importar para el control de los datos, pero Java obliga el uso del IOException para el manejo de excepciones dentro de la clase. Esa es una de las formas de importar el paquete de las clases. Otra pueda ser la siguiente:

```
package Clase;

import java.io.*;

public class Main {

    public static void main(String[] args)
    {

    }

}
```

Ilustración 3: Lectura de datos por teclado - Buffered.

Fuente: Eclipse.

Donde se importa directamente toda la librería gracias al * que hace uso directamente de todas las clases de esta.

```
public class Main {

    public static void main(String[] args) throws IOException
    {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        System.out.print("Por favor ingrese su nombre: ");

        String nombre = br.readLine();

        System.out.print("Bienvenido " + nombre + ". Por favor ingrese su edad: ");

        String entrada = br.readLine();

        int edad = Integer.parseInt(entrada);

        System.out.println("Gracias " + nombre + " tienes " + edad + " años.");
    }

}
```

Ilustración 4: Clase BufferedReader.

Fuente: Eclipse.

Ahora línea por línea en el uso de estas dos clases el funcionamiento es el siguiente:

Lo primero a resaltar es que el administrador de excepciones que java.io proporciona para el uso de `BufferedReader` e `InputStreamReader`.

```
public static void main(String[] args) throws IOException
```

Al hacer uso de estas clases, se debe crear un objeto de las mismas para hacer uso de sus métodos.

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

Para conocer en qué momento solicitar datos por teclado, se debe informar al usuario por medio de un mensaje en pantalla.

```
System.out.print("Por favor ingrese su nombre: ");
```

El método que permite capturar los datos por teclado es `readLine`, pero para acceder al método se debe hacer uso del objeto que se creó y el valor capturado almacenarlo en una variable. Hay que tener en cuenta que por medio de estas clases únicamente se pueden capturar cadenas de texto, si se desean almacenar números, posteriormente se deben castear (Para saber en qué momento terminar la captura de datos, se realiza un salto de línea con la tecla Enter, la variable almacena todo lo anterior a ésta).

```
String nombre = br.readLine();
```

Ya teniendo los datos previamente almacenados, se prueban y se solicita de nuevo información (En este caso números para hacer el casteo).

```
System.out.print("Bienvenido " + nombre + ". Por favor ingrese su edad: ");
```

La captura de datos siempre se realiza la misma forma, con una variable para almacenarse y por medio del objeto creado llamar al método `readLine`.

```
String entrada = br.readLine();
```

El casteo de datos es obligatorio en el trabajo de datos por teclado con las clases `BufferedReader` y `InputStreamReader`. (Guía didáctica 1).

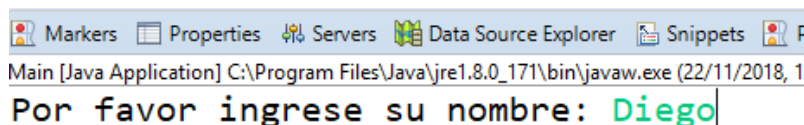
```
int edad = Integer.parseInt(entrada);
```

Finalmente se imprimen los valores recibidos por teclado en las variables asignadas.

```
System.out.println("Gracias " + nombre + " tienes " + edad + " años.");
```

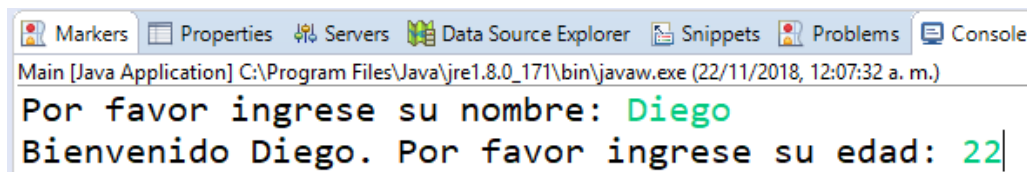
El resultado en tiempo de ejecución a diferencia de cómo se ha trabajado durante el diplomado, es secuencial, la ejecución espera que se ingresen datos para continuar con la siguiente instrucción hasta imprimir todo el código.

- Ejecución del primer bloque.



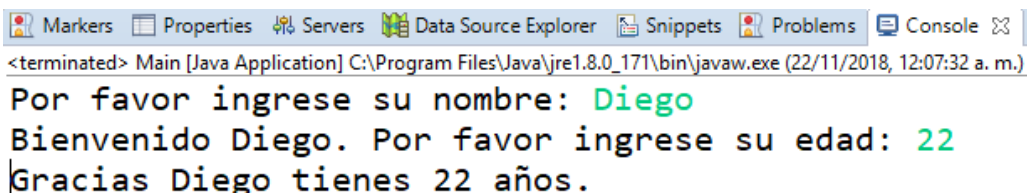
Markers Properties Servers Data Source Explorer Snippets P
Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (22/11/2018, 1.
Por favor ingrese su nombre: **Diego**


- Ejecución del segundo bloque.



Markers Properties Servers Data Source Explorer Snippets Problems Console
Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (22/11/2018, 12:07:32 a. m.)
Por favor ingrese su nombre: **Diego**
Bienvenido Diego. Por favor ingrese su edad: **22**

- Resultado final.



Markers Properties Servers Data Source Explorer Snippets Problems Console 
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (22/11/2018, 12:07:32 a. m.)
Por favor ingrese su nombre: **Diego**
Bienvenido Diego. Por favor ingrese su edad: **22**
Gracias Diego tienes 22 años.

(El texto verde representa los valores ingresados por teclado).

La entrada de datos por teclado por medio de la clase **BufferedReader** y **InputStreamReader** cuenta con la dificultad de no poder diferenciar los tipos de datos en su captura, por lo que el casteo se hace obligatorio cuando se hace necesario trabajar con otros tipos de datos. Representa más trabajos y reprocesos que son innecesarios.

disponibles para el aprendizaje



Para desarrollar las habilidades y destrezas necesarias en cada competencia, es muy importante que tengas acceso a los recursos didácticos adecuados.

Entonces, si quieres ampliar la información que hemos presentado aquí, te sugerimos revisar la siguiente dirección <https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html>

<https://docs.oracle.com/javase/8/docs/api/?java/io/InputStreamReader.html>

La clase **Scanner** por su parte cuenta con métodos independientes para cada tipo. Véase ahora la clase **Scanner** a profundidad.

Scanner

Es una clase permite representar la entrada y lectura de datos en Java a partir de la librería `java.util`; con métodos que proporcionan la interacción con los datos y sus tipos.

Para hacer uso de esta clase el proceso a seguir es simple. Véase su declaración.

```
package Clase;

import java.util.Scanner;

public class Main {

    public static void main(String[] args)
    {

    }

}
```

Ilustración 5: Importar - Scanner.

Fuente: Eclipse.

La clase debe importarse para el control de los datos. Esa es una de las formas de importar el paquete de las clases. Otra pueda ser la siguiente:

```
package Clase;

import java.util.*;

public class Main {

    public static void main(String[] args)
    {

    }

}
```

Ilustración 6: Importar - Scanner.

Fuente: Eclipse.

Donde se importa directamente toda la librería gracias al * que hace uso directamente de todas las clases de esta.

```
package Clase;

import java.util.*;

public class Main {

    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        System.out.print("Por favor ingrese su nombre: ");

        String nombre = sc.nextLine();

        System.out.print("Bienvenido " + nombre + ". Por favor ingrese su edad: ");

        int edad = sc.nextInt();

        System.out.println("Gracias " + nombre + " tienes " + edad + " años.");
    }
}
```

Ilustración 7: Clase Scanner.

Fuente: Eclipse.

Ahora línea por línea en el uso de esta clase, el funcionamiento es el siguiente:

Para el uso de esta clase, al igual de `BufferedReader` y `InputStreamReader`, se debe crear un objeto de la misma para hacer uso de sus métodos.

```
Scanner sc = new Scanner(System.in);
```

Para conocer en qué momento solicitar datos por teclado, se debe informar al usuario por medio de un mensaje en pantalla.

```
System.out.print("Por favor ingrese su nombre: ");
```

El método que permite capturar los datos por teclado es independiente al tipo de dato que se desea almacenar, sí es una cadena `nextLine`, sí es un entero `nextInt`, sí es un double `nextDouble`, así sucesivamente por cada tipo, para acceder al método se debe hacer uso del objeto que se creó y el valor capturado almacenarlo en una variable. (Para saber en qué momento terminar la captura de datos, se realiza un

salto de línea con la tecla Enter, la variable almacena todo lo anterior a ésta).

```
String nombre = sc.nextLine();
```

Ya teniendo los datos previamente almacenados, se prueban y se solicita de nuevo información, en este caso un valor entero.

```
System.out.print("Bienvenido " + nombre + ". Por favor ingrese su edad: ");
```

La captura de datos siempre se realiza la misma forma, con una variable para almacenarse y por medio del objeto creado llamar al método nextInt.

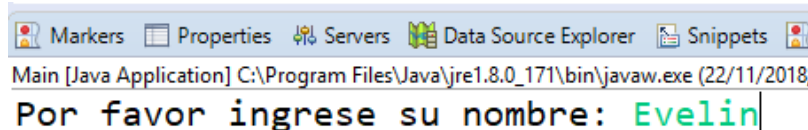
```
int edad = sc.nextInt();
```

Finalmente se imprimen los valores recibidos por teclado en las variables asignadas respectivamente.

```
System.out.println("Gracias " + nombre + " tienes " + edad + " años.");
```

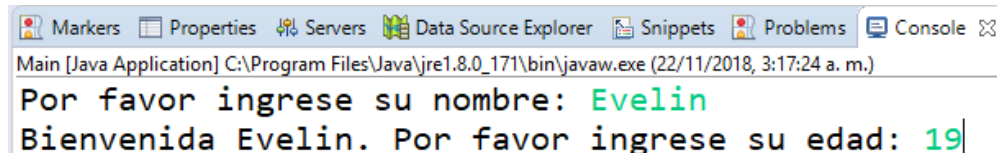
El resultado en tiempo de ejecución a diferencia de cómo se ha trabajado durante el diplomado, es secuencial, la ejecución espera que se ingresen datos para continuar con la siguiente instrucción hasta imprimir todo el código.

- Ejecución del primer bloque.



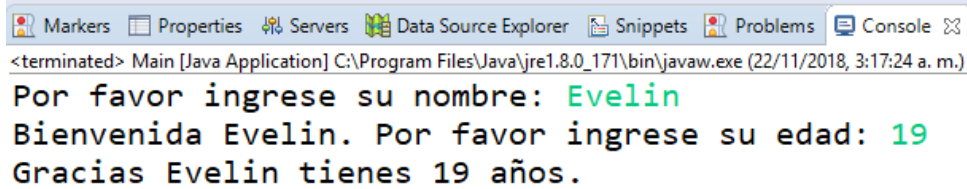
Markers Properties Servers Data Source Explorer Snippets
Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (22/11/2018,
Por favor ingrese su nombre: Evelin

- Ejecución del segundo bloque.



Markers Properties Servers Data Source Explorer Snippets Problems Console
Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (22/11/2018, 3:17:24 a. m.)
Por favor ingrese su nombre: Evelin
Bienvenida Evelin. Por favor ingrese su edad: 19

- Resultado final.



Markers Properties Servers Data Source Explorer Snippets Problems Console
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (22/11/2018, 3:17:24 a. m.)
Por favor ingrese su nombre: **Evelin**
Bienvenida Evelin. Por favor ingrese su edad: **19**
Gracias Evelin tienes 19 años.

(El texto verde representa los valores ingresados por teclado).

A diferencia de las **BufferedReader** y **InputStreamReader**, **Scanner** permite trabajar con otros tipos de datos, evitando reprocesos casteando datos y reduciendo líneas de código.

disponibles para el aprendizaje



Para desarrollar las habilidades y destrezas necesarias en cada competencia, es muy importante que tengas acceso a los recursos didácticos adecuados.

Entonces, si quieres ampliar la información que hemos presentado aquí, te sugerimos revisar la siguiente dirección <https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html>

TEMA 3

Listas

El uso de listas en Java es una forma útil de almacenar y manipular grandes volúmenes de datos, tal como se haría en una matriz o arreglo, pero con una serie de ventajas que hacen de este tipo de variables las preferidas para el procesamiento de grandes cantidades de información.

La implementación de las listas en Java, al igual que otras estructuras de datos se puede realizar de dos formas diferentes:

- Clases e Interfaces de Java.
- Implementación desde cero.

En el núcleo de esta, se centrará en el uso de clases e interfaces de Java para la implementación de estas estructuras.

Las estructuras de datos que hacen uso de listas en Java y se abordaran en esta guía son:

- **List**
- **ArrayList**
- **LinkedList**

Son clases e interfaces que se especializan en el uso e implementación de Listas a partir de estructuras existentes y definidas por Java con todos los métodos definidos e implementados. La

aplicación de éstas no se diferencia la una de las otras, dado que las estructuras, métodos y aplicación son muy similares.

Para conocer y mostrar la aplicación de los respectivos métodos de las estructuras List y ArrayList, se conocerá primero sus definiciones, características y creación. Posteriormente por medio de la interfaz List se aplicará detalladamente los principales métodos y su funcionamiento, dado que los métodos son similares en cada estructura.

La clase LinkedList es un poco más particular, por lo que se describirán sus métodos y aplicaciones de forma individual.

Interfaz List

La interfaz de la Lista de Java, `java.util.List` representa una secuencia ordenada de objetos. Los elementos contenidos en un Java List se pueden insertar, acceder, iterar y eliminar de acuerdo con el orden en que aparecen internamente en el Java List. El orden de los elementos es la razón por la que esta estructura de datos se denomina Lista. (jenkov, 2018).

Algo interesante de las listas en Java es el hecho de que no es necesario establecer un tamaño específico para la List, a diferencia de las tradicionales matrices o arreglos. Las listas son sumamente versátiles y mucho más fáciles de manejar que otros tipos de variables de agrupación de datos.

Algunas características que se deben tener en cuenta frente al uso de List.

- Puede tener un tamaño establecido, aunque su dimensión es dinámica.
- Se recomienda definir un tipo de dato para la List.
- La importación de las clases e interfaces es obligatoria.
- Hay listas que pueden almacenar cualquier tipo de dato en su estructura. Simplemente no se le asigna tipo alguno.
- Los índices empiezan en 0 al igual que en los vectores y matrices.

Declaración de una List

Para el uso de clases e interfaces de este tipo es necesario hacer la respectiva importación de los paquetes de Java que se desean trabajar, List opera con dos en este caso: List – ArrayList (Aunque más adelante se describirá específicamente el uso de ésta).

```
package Clase;

import java.util.ArrayList;
import java.util.List;

public class Main {

    public static void main(String[] args)
    {
        List<String> listaNombres = new ArrayList<String>();
    }
}
```

Ilustración 8: List.

Fuente: Eclipse.

En este caso se declara una List de tipo String, dentro de <> se puede declarar el tipo de dato que se desee. Por ejemplo:

- List de enteros.

```
public static void main(String[] args)
{
    List<Integer> lista = new ArrayList<Integer>();
}
```

- List de Doubles.

```
public static void main(String[] args)
{
    List<Double> lista = new ArrayList<Double>();
}
```

- List de Objetos.

```
public static void main(String[] args)
{
    List<Object> lista = new ArrayList<Object>();
}
```

- List de un tipo de clase.

```
public static void main(String[] args)
{
    List<Usuario> lista = new ArrayList<Usuario>();
}
```

- List de cualquier tipo.

```
public static void main(String[] args)
{
    List lista = new ArrayList();
}
```

- List con tamaño establecido.

```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>(10);
}
```

A partir de la lista creada se puede implementar las funcionalidades a la misma a partir de lo que se conoce como CRUD, siglas de Create – Read – Update – Delete (Crear – Leer – Actualizar – Eliminar).

Agregar Elementos a una List

La inserción de datos en una lista no es más que agregar elementos a esta en base al tipo de dato definido en su declaración.

Para realizar este proceso existen varias formas o métodos que permiten llevar a cabo la inserción de datos en una lista. Véase.

Método Add

Recibe únicamente el dato que se desea guardar, el índice lo asigna la lista en base a los demás datos registrados. Recordar que los índices empiezan el 0.

```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>();

    lista.add("Colombia");
    lista.add("Chile");
    lista.add("Argentina");
    lista.add("Venezuela");
    lista.add("Perú");

}
```

Ilustración 9: List - Add.

Fuente: Eclipse.

El método se puede usar cuantas veces sea necesario, dado que no existe un tamaño predefinido que limite el número de datos a establecer.

Método Add con Índice

Recibe el dato que se desea guarda y el índice que se le desea asignar al dato. Hay que tener presente que no existan datos en posiciones donde se desea acceder.

```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>();

    lista.add(0, "Colombia");
    lista.add(2, "Chile");
    lista.add(3, "Argentina");
    lista.add(5, "Venezuela");
    lista.add(10, "Perú");

}
```

Ilustración 10: List - Add.

Fuente: Eclipse.

Método Add List

Recibe todos los elementos de una lista nueva, ésta nueva debe ser del mismo tipo de la contenedora. Los índices se asignan automáticamente.

```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>();
    lista.add(0, "Colombia");
    lista.add(1, "Chile");
    lista.add(2, "Argentina");
    lista.add(3, "Venezuela");
    lista.add(4, "Perú");

    List<String> listaNueva = new ArrayList<String>();

    listaNueva.add("México");
    listaNueva.add("Panamá");
    listaNueva.add("Ecuador");

    lista.addAll(listaNueva);
}
```

Ilustración 11: List - Add.

Fuente: Eclipse.

Método Add List con Índice

Recibe todos los elementos de una lista nuevo en una posición especificada. Hay que tener presente que no existan datos en posiciones donde se desea acceder.

```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>();
    lista.add(0, "Colombia");
    lista.add(1, "Chile");
    lista.add(2, "Argentina");
    lista.add(3, "Venezuela");
    lista.add(4, "Perú");

    List<String> listaNueva = new ArrayList<String>();

    listaNueva.add("México");
    listaNueva.add("Panamá");
    listaNueva.add("Ecuador");

    lista.addAll(5, listaNueva);
}
```

Ilustración 2: List - Add.

Fuente: Eclipse.

Método Set

Actualiza un índice de la lista a partir de una posición y un valor del mismo tipo determinado. No se pueden actualizar posiciones no existentes

```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>();
    lista.add(0, "Colombia");
    lista.add(1, "Chile");
    lista.add(2, "Argentina");
    lista.add(3, "Venezuela");
    lista.add(4, "Perú");

    lista.set(0, "Costa Rica");
}
```

Ilustración 13: List - Set.

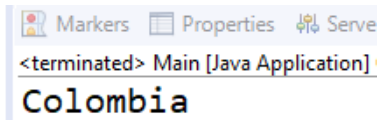
Fuente: Eclipse.

Método Get

Recupera un valor de la lista a partir de una posición determinada. Recordar que el índice debe ser un entero.

```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>();
    lista.add(0, "Colombia");

    System.out.println(lista.get(0));
}
```



Markers Properties Serve
<terminated> Main [Java Application]
Colombia

Ilustración 14: List - Get.

Fuente: Eclipse.

Método Size

Devuelve el tamaño de la lista a partir de los valores existentes dentro de ésta.

```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>();

    lista.add(0, "Colombia");
    lista.add(1, "Ecuador");
    lista.add(2, "Venezuela");
    lista.add(3, "Argentina");

    System.out.println("El tamaño de la lista es: " + lista.size());
}
```



Markers Properties Servers Data Source Explo
<terminated> Main [Java Application] C:\Program Files\Java\jre
El tamaño de la lista es: 4

Ilustración 15: List - Size.

Fuente: Eclipse.

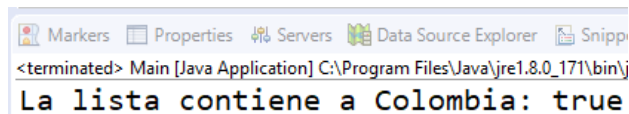
Método Constains

Recibe un valor del mismo tipo y determina si existe o no dentro de la lista. Devuelve true si existe y falso en caso contrario.

```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>();

    lista.add(0, "Colombia");
    lista.add(1, "Ecuador");
    lista.add(2, "Venezuela");
    lista.add(3, "Argentina");

    System.out.println("La lista contiene a Colombia: " + lista.contains("Colombia"));
}
```



Markers Properties Servers Data Source Explorer Snipp
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\j
La lista contiene a Colombia: true

Ilustración 16: List - Constain.

Fuente: Eclipse.

Este método puede ser usado dentro de condicionales por el tipo booleano de retorno.

```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>();

    lista.add(0, "Colombia");
    lista.add(1, "Ecuador");
    lista.add(2, "Venezuela");
    lista.add(3, "Argentina");

    if(lista.contains("Colombia"))
    {
        System.out.println("Existe");
    }
}
```

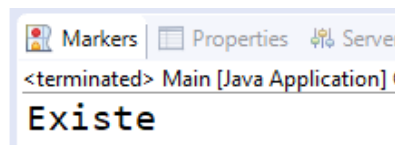


Ilustración 17: List - Contains.

Fuente: Eclipse.

Método Clear

Limpia todos los datos de la lista de forma que esta queda completamente vacía.

```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>();

    lista.add(0, "Colombia");
    lista.add(1, "Ecuador");
    lista.add(2, "Venezuela");
    lista.add(3, "Argentina");

    lista.clear();

    System.out.println("El tamaño de la lista es: " + lista.size());
}
```

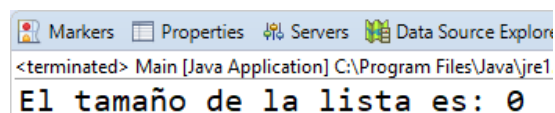


Ilustración 18: List - Clear.

Fuente: Eclipse.

Método isEmpty

Retorna verdadero en caso de que la lista se encuentre vacía o falso en caso contrario.

```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>();

    lista.add(0, "Colombia");
    lista.add(1, "Ecuador");
    lista.add(2, "Venezuela");
    lista.add(3, "Argentina");

    if(lista.isEmpty())
    {
        System.out.println("La lista está vacía");
    }
    else
    {
        System.out.println("La lista no está vacía");
    }
}
```

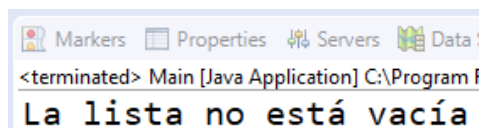


Ilustración 18: List - isEmpty.

Fuente: Eclipse.

Método Remove por Índice

Recibe índice válido dentro de la lista y elimina el dato que se encuentre en ésta.

```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>();

    lista.add(0, "Colombia");
    lista.add(1, "Ecuador");
    lista.add(2, "Venezuela");
    lista.add(3, "Argentina");

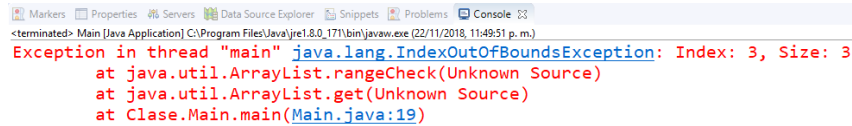
    lista.remove(3);

    System.out.println(lista.get(3));
}
```

Ilustración 19: List - Remove.

Fuente: Eclipse.

Error dado que la posición no se encuentra disponible en la lista.



```
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (22/11/2018, 11:49:51 p. m.)
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 3, Size: 3
    at java.util.ArrayList.rangeCheck(Unknown Source)
    at java.util.ArrayList.get(Unknown Source)
    at Clase.Main.main(Main.java:19)
```

Método Remove por Valor

Elimina la primera aparición del dato especificado en el método.

```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>();

    lista.add(0, "Colombia");
    lista.add(1, "Ecuador");
    lista.add(2, "Venezuela");
    lista.add(3, "Argentina");

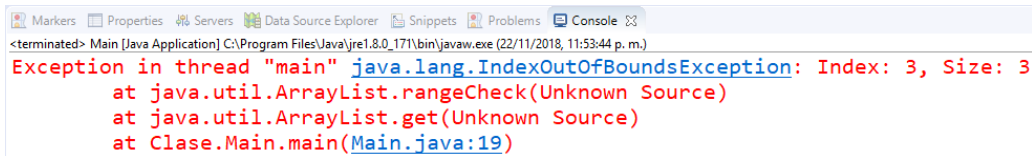
    lista.remove("Argentina");

    System.out.println(lista.get(3));
}
```

Ilustración 20: List - Remove.

Fuente: Eclipse.

Error dado que la posición no se encuentra disponible en la lista.



```
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (22/11/2018, 11:53:44 p. m.)
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 3, Size: 3
    at java.util.ArrayList.rangeCheck(Unknown Source)
    at java.util.ArrayList.get(Unknown Source)
    at Clase.Main.main(Main.java:19)
```

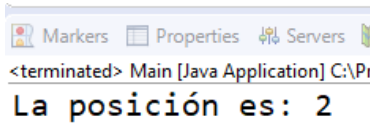
Método IndexOf

Devuelve el índice de un dato especificado en el método en caso de que exista en la lista.

```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>();

    lista.add(0, "Colombia");
    lista.add(1, "Ecuador");
    lista.add(2, "Venezuela");
    lista.add(3, "Argentina");

    System.out.println("La posición es: " + lista.indexOf("Venezuela"));
}
```



Markers Properties Servers
<terminated> Main [Java Application] C:\Pr
La posición es: 2

Ilustración 21: List - IndexOf.

Fuente: Eclipse

Método Iterator

Crea una variable de iteración de listas para recorrer la mismas en el orden en el que se encuentra almacenada.

```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>();

    lista.add(0, "Colombia");
    lista.add(1, "Ecuador");
    lista.add(2, "Venezuela");
    lista.add(3, "Argentina");

    Iterator<String> listaIterable = lista.iterator();

    while(listaIterable.hasNext())
    {
        System.out.println("Valor: " + listaIterable.next());
    }
}
```



Markers Properties Servers Da
<terminated> Main [Java Application] C:\Progr
Valor: Colombia
Valor: Ecuador
Valor: Venezuela
Valor: Argentina

Ilustración 22: List - Iterator.

Fuente: Eclipse.

Se debe importar de igual forma Iterator para correcto funcionamiento.

```
import java.util.Iterator;
```

Estos son unos de los métodos principales del interfaz List, existen muchos otros métodos que derivan de los anteriores y que su implementación no va muy allá de la vista en éstos.

disponibles para el aprendizaje



Para desarrollar las habilidades y destrezas necesarias en cada competencia, es muy importante que tengas acceso a los recursos didácticos adecuados.

Entonces, si quieres ampliar la información que hemos presentado aquí, te sugerimos revisar la siguiente dirección <https://docs.oracle.com/javase/8/docs/api/java/util/List.html#>

Clase ArrayList

La clase de la Lista de Java, `java.util.ArrayList` representa una secuencia ordenada de objetos. Los elementos contenidos en un Java `ArrayList` se pueden insertar, acceder, iterar y eliminar de acuerdo con el orden en que aparecen internamente en el Java `ArrayList`. El orden de los elementos es la razón por la que esta estructura de datos se denomina `ArrayList`.

Realiza la implementación de una matriz de tamaño variable de la interfaz de `List`. Implementa todas las operaciones de `List` opcionales y permite todos los elementos, incluido el valor nulo. Además de implementar la interfaz de `List`, esta clase proporciona métodos para manipular el tamaño de la matriz que se utiliza internamente para almacenar la lista. (Esta clase es aproximadamente equivalente a `Vector`, excepto que no está sincronizada).

Algunas características que se deben tener en cuenta frente al uso de `ArrayList`.

- `ArrayList` se puede inicializa por un tamaño, sin embargo, el tamaño puede aumentar si la colección aumenta o se reduce si los datos se eliminan de la colección.
- `ArrayList` no se puede usar para tipos primitivos, como `int`, `char`, etc. Se deben usar `Wrappers`.
- La importación de las clases e interfaces es obligatoria.
- Hay `ArrayList` que pueden almacenar cualquier tipo de dato en su estructura. Simplemente no se le asigna tipo alguno.

- Los índices empiezan en 0 al igual que en los vectores y matrices.

Declaración de un ArrayList

Para el uso de la clase de este tipo es necesario hacer la respectiva importación de los paquetes de Java que se desean trabajar, ArrayList opera con una única clase: ArrayList.

```
package Clase;

import java.util.ArrayList;

public class Main {

    public static void main(String[] args)
    {
        ArrayList<Integer> lista = new ArrayList<Integer>();
    }
}
```

Ilustración 22: ArrayList.

Fuente: Eclipse.

En este caso se declara una ArrayList de tipo Integer, dentro de <> se puede declarar el tipo de dato que se desee, siempre y cuando no sea primitivo. Por ejemplo:

- ArrayList de cadenas.

```
public static void main(String[] args)
{
    ArrayList<String> lista = new ArrayList<String>();
}
```

- ArrayList de Doubles.

```
public static void main(String[] args)
{
    ArrayList<Double> lista = new ArrayList<Double>();
}
```

- ArrayList de Objetos.

```
public static void main(String[] args)
{
    ArrayList<Object> lista = new ArrayList<Object>();
}
```

- ArrayList de un tipo de clase.

```
public static void main(String[] args)
{
    ArrayList<Usuario> lista = new ArrayList<Usuario>();
}
```

- ArrayList de cualquier tipo.

```
public static void main(String[] args)
{
    ArrayList lista = new ArrayList();
}
```

Dado que los ArrayList tienen características similares a las matrices y vectores, estos pueden recibir un tamaño establecido de entrada.

```
public static void main(String[] args)
{
    ArrayList<Integer> lista = new ArrayList<Integer>(10);
}
```

disponibles para el aprendizaje



Para desarrollar las habilidades y destrezas necesarias en cada competencia, es muy importante que tengas acceso a los recursos didácticos adecuados.

Recuerda que los métodos y la aplicación de éstos es similar a la interfaz List. Para conocer los métodos que se implementan en esta clase, te sugerimos revisar la siguiente dirección <https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html>

Clase LinkedList

La clase de la Lista de Java, `java.util.LinkedList` representa una secuencia ordenada de objetos enlazados. El orden de los elementos es la razón por la que esta estructura de datos se denomina `LinkedList`.

Las listas enlazadas son una estructura de datos lineales donde los elementos no se almacenan en ubicaciones contiguas y cada elemento es un objeto separado con una parte de datos y una parte de dirección. Los elementos están vinculados mediante punteros y direcciones. Cada elemento es conocido como un nodo. Debido al dinamismo y la facilidad de las inserciones y eliminaciones, se prefieren a las matrices. También tiene algunas desventajas, ya que no se puede acceder directamente a los nodos, sino que debemos comenzar desde la cabeza y seguir el enlace para llegar a un nodo al que deseamos acceder. (geeksforgeeks, 2018).

Algunas características que se deben tener en cuenta frente al uso de `LinkedList`.

- No se puede inicializa el tamaño, sin embargo, el tamaño aumenta si la colección aumenta o se reduce si los datos se eliminan de la colección.
- No se puede usar para tipos primitivos, como `int`, `char`, etc. Se deben usar `Wrappers`.
- La importación de las clases e interfaces es obligatoria.
- Hay `LinkedList` que pueden almacenar cualquier tipo de dato en su estructura. Simplemente no se le asigna tipo alguno.
- Los índices empiezan en 0 al igual que en los vectores y matrices.

Declaración de un LinkedList

Para el uso de la clase de este tipo es necesario hacer la respectiva importación de los paquetes de Java que se desean trabajar, `LinkedList` opera con una única clase: `LinkedList`.


```
package Clase;

import java.util.LinkedList;

public class Main {

    public static void main(String[] args)
    {
        LinkedList<String> lista = new LinkedList<String>();
    }
}
```

Ilustración 23: LinkedList.

Fuente: Eclipse.

En este caso se declara un LinkedList de tipo String, dentro de <> se puede declarar el tipo de dato que se desee, siempre y cuando no sea primitivo. Por ejemplo:

- LinkedList de Enteros.

```
public static void main(String[] args)
{
    LinkedList<Integer> lista = new LinkedList<Integer>();
}
```

- LinkedList de Doubles.

```
public static void main(String[] args)
{
    LinkedList<Double> lista = new LinkedList<Double>();
}
```

- LinkedList de Objetos.

```
public static void main(String[] args)
{
    LinkedList<Object> lista = new LinkedList<Object>();
}
```

- LinkedList de un tipo de clase.

```
public static void main(String[] args)
{
    LinkedList<Usuario> lista = new LinkedList<Usuario>();
}
```

- LinkedList de cualquier tipo.

```
public static void main(String[] args)
{
    LinkedList lista = new LinkedList();
}
```

Método Add

Recibe únicamente el dato que se desea guarda, el índice lo asigna la lista en base a los demás datos e índices registrados. Recordar que los índices empiezan el 0.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add("Diego");
    lista.add("Evelin");
    lista.add("Juan");
}
```

Ilustración 24: LinkedList - Add.

Fuente: Eclipse.

El método se puede usar cuantas veces sea necesario, dado que no existe un tamaño predefinido que limite el número de datos a establecer.

Método Add con Índice

Recibe el dato que se desea guarda y el índice que se le desea asignar al dato. Hay que tener presente que no existan datos en posiciones donde se desea acceder.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Diego");
    lista.add(1, "Evelin");
    lista.add(2, "Juan");
}
```

Ilustración 25: LinkedList - Add.

Fuente: Eclipse.

Método Add LinkedList

Recibe todos los elementos de una lista nueva, ésta nueva debe ser del mismo tipo de la contenedora. Los índices se asignan automáticamente.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Diego");
    lista.add(1, "Evelin");
    lista.add(2, "Juan");

    LinkedList<String> listaNueva = new LinkedList<String>();

    listaNueva.add(3, "Stiven");

    lista.addAll(listaNueva);
}
```

Ilustración 26: LinkedList - Add.

Fuente: Eclipse.

Método Add LinkedList con Índice

Recibe todos los elementos de una lista nueva en una posición especificada. Hay que tener presente que no existan datos en posiciones donde se desea acceder.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Diego");
    lista.add(1, "Evelin");
    lista.add(2, "Juan");

    LinkedList<String> listaNueva = new LinkedList<String>();

    listaNueva.add(3, "Stiven");

    lista.addAll(3, listaNueva);
}
```

Ilustración 27: LinkedList - AddAll.

Fuente: Eclipse.

Método AddFirst

Recibe únicamente el dato que se desea guarda, el índice asignado es el 0, así que queda en la primera posición, por ende, las demás posiciones aumentan en 1 en caso de que existan.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add("Stiven");
    lista.add("Fernando");
    lista.addFirst("Diego");
}
```

Ilustración 28: LinkedList - AddFirst.

Fuente: Eclipse.

Método AddLast

Recibe únicamente el dato que se desea guarda, el índice asignado depende de la lista, dado que este debe quedar en la última posición. Si no existe última. Éste dato será el primero, último y único hasta que se ingresen más.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.addLast("Alejandro");
    lista.add("Stiven");
    lista.add("Fernando");
}
```

Ilustración 29: LinkedList - AddLast.

Fuente: Eclipse.

Método Set

Actualiza un índice de la lista a partir de una posición y un valor del mismo tipo determinado. No se pueden actualizar posiciones no existentes

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");

    lista.set(1, "Juan");
}
```

Ilustración 29: LinkedList - Set.

Fuente: Eclipse.

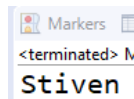
Método Get

Recupera un valor de la lista a partir de una posición determinada.
Recordar que el índice debe ser un entero.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");

    System.out.println(lista.get(0));
}
```



Markers
<terminated> Main
Stiven

Ilustración 30: LinkedList - Get.

Fuente: Eclipse.

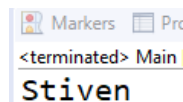
Método GetFirst

Recupera un valor del primer elemento de la lista.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    System.out.println(lista.getFirst());
}
```



Markers
<terminated> Main
Stiven

Ilustración 31: LinkedList - GetFirst.

Fuente: Eclipse.

Método GetLast

Recupera un valor del último elemento de la lista.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    System.out.println(lista.getLast());
}
```

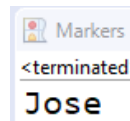


Ilustración 32: LinkedList - GetLast.

Fuente: Eclipse.

Método Size

Devuelve el tamaño de la lista a partir de los valores existentes dentro de ésta.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    System.out.println("El tamaño de la lista es: " + lista.size());
}
```

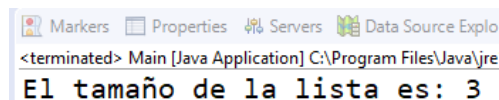


Ilustración 33: LinkedList - Size.

Fuente: Eclipse.

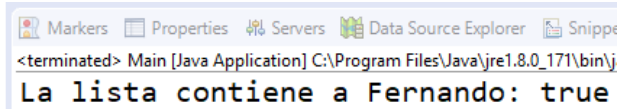
Método Constains

Recibe un valor del mismo tipo y determina si existe o no dentro de la lista. Devuelve true si existe y falso en caso contrario.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    System.out.println("La lista contiene a Fernando: " + lista.contains("Fernando"));
}
```



La lista contiene a Fernando: true

Ilustración 34: LinkedList - Constains.

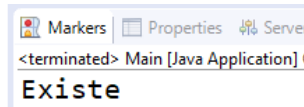
Fuente: Eclipse.

Este método puede ser usado dentro de condicionales por el tipo booleano de retorno.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    if(lista.contains("Fernando"))
    {
        System.out.println("Existe");
    }
}
```



Existe

Ilustración 35: LinkedList - Constains.

Fuente: Eclipse.

Método Clear

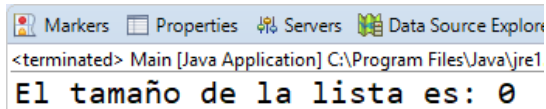
Limpia todos los datos de la lista de forma que esta queda completamente vacía.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    lista.clear();

    System.out.println("El tamaño de la lista es: " + lista.size());
}
```



El tamaño de la lista es: 0

Ilustración 36: LinkedList - Clear.

Fuente: Eclipse.

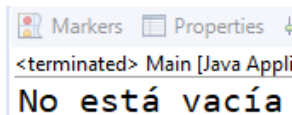
Método isEmpty

Retorna verdadero en caso de que la lista se encuentre vacía o falso en caso contrario.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    if(lista.isEmpty())
    {
        System.out.println("Está vacía");
    }
    else
    {
        System.out.println("No está vacía");
    }
}
```



No está vacía

Ilustración 37: LinkedList - isEmpty.

Fuente: Eclipse.

Método Remove

Recibe índice válido dentro de la lista y elimina el dato que se encuentre en ésta. La lista se ordena de forma que no queda espacios vacíos o huecos.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    lista.remove();
}
```

Ilustración 38: LinkedList - Remove.

Fuente: Eclipse.

Método Remove por Valor

Elimina la primera aparición del dato especificado en el método.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    lista.remove("Fernando");
}
```

Ilustración 39: LinkedList - Remove.

Fuente: Eclipse.

Método Remove por Índice

Elimina el dato que se encuentre en la posición especificada.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    lista.remove(2);
}
```

Ilustración 40: LinkedList - Remove.

Fuente: Eclipse.

Método Offer

Recibe únicamente el dato que se desea guarda, el índice asignado depende de la lista, dado que este debe quedar en la última posición. Si no existe última. Éste dato será el primero, último y único hasta que se ingresen más.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");

    lista.offer("Evelin");

    lista.add(2, "Jose");
}
```

Ilustración 41: LinkedList - Offer.

Fuente: Eclipse.

Método OfferFirst

Recibe únicamente el dato que se desea guarda, el índice asignado es el 0, así que queda en la primera posición, por ende, las demás posiciones aumentan en 1 en caso de que existan.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");

    lista.offerFirst("Mía");

    lista.add(2, "Jose");
}
```

Ilustración 42: LinkedList - OfferFirst.

Fuente: Eclipse.

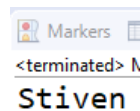
Método Peek

Recupera, pero no elimina el primero elemento de la lista.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    System.out.println(lista.peek());
}
```



Markers
<terminated> N
Stiven

Ilustración 43: LinkedList - Peek.

Fuente: Eclipse.

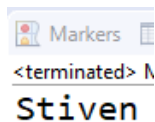
Método PeekFirst

Recupera, pero no elimina el primero elemento de la lista.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    System.out.println(lista.peekFirst());
}
```



Markers
<terminated> N
Stiven

Ilustración 44: LinkedList - PeekFirst.

Fuente: Eclipse.

Método PeekLast

Recupera, pero no elimina el último elemento de la lista.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    System.out.println(lista.peekLast());
}
```

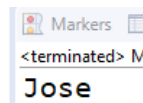


Ilustración 45: LinkedList - PeekLast.

Fuente: Eclipse.

Método Poll

Recupera y elimina el primer elemento de la lista.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    System.out.println(lista.poll());

    System.out.println(lista.get(0));
}
```

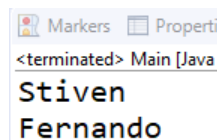


Ilustración 46: LinkedList - Poll.

Fuente: Eclipse.

Método PollFirst

Recupera y elimina el primer elemento de la lista.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    System.out.println(lista.pollFirst());

    System.out.println(lista.get(0));
}
```

Ilustración 47: LinkedList - PollFirst.

Fuente: Eclipse.

Método PollLast

Recupera y elimina el último elemento de la lista.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    System.out.println(lista.pollLast());
}
```

Ilustración 48: LinkedList - PollLast.

Fuente: Eclipse.

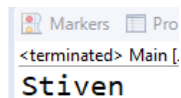
Método Pop

Recupera y no elimina el primer elemento de la lista.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    System.out.println(lista.pop());
}
```



Markers Pro
<terminated> Main [.
Stiven

Ilustración 49: LinkedList - Pop.

Fuente: Eclipse.

Método Push

Recibe únicamente el dato que se desea guarda, el índice asignado es el 0, así que queda en la primera posición, por ende, las demás posiciones aumentan en 1 en caso de que existan.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    lista.push("Andres");
}
```

Ilustración 50: LinkedList - Push.

Fuente: Eclipse

Método LastIterator

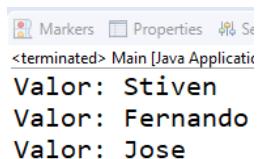
Crea una variable de iteración de listas para recorrer la mismas en el orden en el que se encuentra almacenada.

```
public static void main(String[] args)
{
    LinkedList<String> lista = new LinkedList<String>();

    lista.add(0, "Stiven");
    lista.add(1, "Fernando");
    lista.add(2, "Jose");

    ListIterator<String> listaIterable = lista.listIterator();

    while(listaIterable.hasNext())
    {
        System.out.println("Valor: " + listaIterable.next());
    }
}
```



Markers Properties Set
<terminated> Main [Java Applicati
Valor: Stiven
Valor: Fernando
Valor: Jose

Se debe importar de igual forma LastIterator para el correcto funcionamiento.

```
import java.util.ListIterator;
```

Ilustración 51: LinkedList - LastIterator.

Fuente: Eclipse

Estos son unos de los métodos principales de la clase LinkedList, existen muchos otros métodos que derivan de los anteriores y que su implementación no va muy allá de la vista en éstos.

disponibles para el aprendizaje



Para desarrollar las habilidades y destrezas necesarias en cada competencia, es muy importante que tengas acceso a los recursos didácticos adecuados.

Te sugerimos revisar la siguiente dirección para profundizar los temas acerca de LinkedList:
<https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html>

TEMA 4

Pilas

Una pila es una versión restringida de una lista (Stack en inglés) es una estructura de datos lineal que solo tienen un único punto de acceso fijo por el cual se añaden, eliminan o se consultan elementos. El modo de acceso a los elementos es de tipo LIFO (del inglés Last In First Out, último en entrar, primero en salir). (codigolibre, 2018).

En el lenguaje Java contamos con la clase Stack en la librería java.util.

- Los índices empiezan en 1 con los métodos de la Clase Stack, si se implementan otros, desde el índice 0.
- La única forma de acceder a los elementos es desde la cima de la pila.
- Su administración es muy sencilla ya que tiene pocas operaciones.
- Si la pila está vacía no tiene sentido referirse a una cima ni a un fondo.
- En caso de querer acceder a un elemento que no se encuentre en la cima de la pila se debe realizar un volcado de la pila a una pila auxiliar, una vez realizada la operación con el elemento se vuelve a volcar los elementos de la pila auxiliar a la original.

Declaración de un Pila

Para el uso de la clase de este tipo es necesario hacer la respectiva importación del paquete de Java que se desea trabajar, Stack opera con una única clase: Stack.


```
package Clase;

import java.util.Stack;

public class Main {

    public static void main(String[] args)
    {
        Stack<String> pila = new Stack<String>();
    }
}
```

Ilustración 52: Stack.

Fuente: Eclipse

En este caso se declara una Stack de tipo String, dentro de <> se puede declarar el tipo de dato que se desee, siempre y cuando sea primitivo. Por ejemplo:

- Stack de Enteros.

```
public static void main(String[] args)
{
    Stack<Integer> pila = new Stack<Integer>();
}
```

- Stack de Doubles.

```
public static void main(String[] args)
{
    Stack<Double> pila = new Stack<Double>();
}
```

- Stack de Objetos.

```
public static void main(String[] args)
{
    Stack<Object> pila = new Stack<Object>();
}
```

- Stack de un tipo de clase.

```
public static void main(String[] args)
{
    Stack<Usuario> pila = new Stack<Usuario>();
}
```

- Stack de cualquier tipo.

```
public static void main(String[] args)
{
    Stack pila = new Stack();
}
```

La clase Stack cuenta 5 métodos propios de su clase: **Empty, Peek, Pop, Push y Search**. Y a su vez implementa alguno de los métodos de las otras estructuras: **Add, Remove, Clear, Set, Get**, entre otros.

Método Push

Recibe un dato del tipo de la Pila y este es insertado la parte superior de la Pila.

```
public static void main(String[] args)
{
    Stack<String> pila = new Stack<String>();

    pila.push("Matemáticas");
    pila.push("Español");
    pila.push("Sociales");
}
```

Ilustración 53: Stack - Push.

Fuente: Eclipse

Método Pop

Devuelve y elimina el elemento superior de la pila. Se lanza una excepción 'EmptyStackException' si se llama pop () cuando la pila de invocación está vacía.

```
public static void main(String[] args)
{
    Stack<String> pila = new Stack<String>();

    pila.push("Matemáticas");
    pila.push("Español");
    pila.push("Sociales");

    System.out.println(pila.pop());
    System.out.println(pila.pop());
}
```

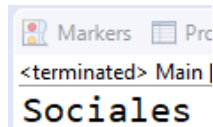


Ilustración 54: Stack - Pop.

Fuente: Eclipse

Método Peek

Devuelve el elemento en la parte superior de la pila, pero no lo elimina.

```
public static void main(String[] args)
{
    Stack<String> pila = new Stack<String>();

    pila.push("Matemáticas");
    pila.push("Español");
    pila.push("Sociales");

    System.out.println(pila.peek());
}
```

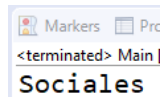


Ilustración 54: Stack - Peek.

Fuente: Eclipse

Método Empty

Retorna verdadero en caso de que la lista se encuentre vacía o falso en caso contrario.

```
public static void main(String[] args)
{
    Stack<String> pila = new Stack<String>();

    pila.push("Matemáticas");
    pila.push("Español");
    pila.push("Sociales");

    System.out.println("¿La pila está vacía?: " + pila.empty());
}
```



Ilustración 55: Stack - Empty.

Fuente: Eclipse

Método Search

Determina si un dato existe en la pila. Si se encuentra el elemento, devuelve la posición del elemento desde la parte superior de la pila. Si no, devuelve -1.

```
public static void main(String[] args)
{
    Stack<String> pila = new Stack<String>();

    pila.push("Matemáticas");
    pila.push("Español");
    pila.push("Sociales");

    System.out.println(pila.search("Español"));
    System.out.println(pila.search("Sociales"));
    System.out.println(pila.search("Ingles"));
}
```

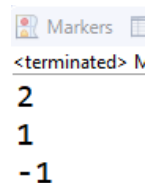


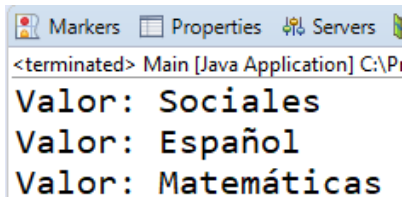
Ilustración 56: Stack - Search.

Fuente: Eclipse

Método Recorrer Pila

Recupera y recorre todos los datos de la Pila a partir de su estructura.

```
public static void main(String[] args) {  
  
    Stack<String> pila = new Stack<String>();  
  
    pila.push("Matemáticas");  
    pila.push("Español");  
    pila.push("Sociales");  
  
    do {  
        System.out.println("Valor: " + pila.peek());  
    }while(pila.pop() != null && !pila.empty());  
}
```



Markers Properties Servers
<terminated> Main [Java Application] C:\P
Valor: Sociales
Valor: Español
Valor: Matemáticas

Ilustración 57: Stack - Recorrido.

Fuente: Eclipse

disponibles para el aprendizaje



Para desarrollar las habilidades y destrezas necesarias en cada competencia, es muy importante que tengas acceso a los recursos didácticos adecuados.

Te sugerimos revisar la siguiente dirección para profundizar los temas acerca de la clase Stack:
<https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html#>

TEMA 5

Colas

Una cola, es una estructura de datos caracterizada por ser una secuencia de elementos en la que la operación de inserción se realiza por un extremo y la operación de extracción se realiza por el otro. También se llama estructura FIFO (First In First Out), debido a que el primer elemento en entrar será también el primero en salir.

Este tipo de estructura de datos abstracta se implementa en lenguajes orientados a objetos mediante clases, en forma de listas enlazadas. (RoverOportunity2012, 2018).

La particularidad de una estructura de datos de cola es el hecho de que sólo puede accederse al primer y al último elemento de la estructura. Así mismo, los elementos sólo se pueden eliminar por el principio y sólo se pueden añadir por el final de la cola.

Ejemplos de colas en la vida real serían: personas comprando en un supermercado, esperando para entrar a ver un partido de béisbol, esperando en el cine para ver una película, una pequeña peluquería, etc. La idea esencial es que son todas líneas de espera.

Teóricamente, la característica de las colas es que tienen una capacidad específica. Por muchos elementos que contengan siempre se puede añadir un elemento más y en caso de estar vacía borrar un elemento sería imposible hasta que no se añada un nuevo elemento. (ecured, 2018).

Algunas características que componen las Colas en java son las siguientes.

- Se recomienda el uso de tipo de dato, en la mayoría de los casos primitivo.

- El tamaño es dinámico.
- Cuenta con métodos de las otras estructuras, pero cuenta 6 métodos propios de su interface.
- La interface que implemente tiene como nombre: queue.
- La clase en la que se apoya es: LinkedList.
- Maneja la filosofía FIFO.

Declaración de una Cola

Para el uso de la interface de este tipo es necesario hacer la respectiva importación del paquete de Java que se desea trabajar, Queue opera con una interface: Queue y una clase de lista: LinkedList.

```
package Clase;

import java.util.LinkedList;
import java.util.Queue;

public class Main {

    public static void main(String[] args) {

        Queue<Integer> cola = new LinkedList<Integer>();
    }
}
```

Ilustración 58: Queue.

Fuente: Eclipse

En este caso se declara una Queue de tipo Integer, dentro de <> se puede declarar el tipo de dato que se desee, siempre y cuando sea primitivo. Por ejemplo:

- Queue de Cadenas.

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

    Queue<String> cola = new LinkedList<String>();
}
```

- Queue de Doubles.

```
public static void main(String[] args) {
    Queue<Double> cola = new LinkedList<Double>();
}
```

- Queue de Objetos.

```
public static void main(String[] args) {
    Queue<Object> cola = new LinkedList<Object>();
}
```

- Queue de un tipo de clase.

```
public static void main(String[] args) {
    Queue<Usuario> cola = new LinkedList<Usuario>();
}
```

- Queue de cualquier tipo.

```
public static void main(String[] args) {
    Queue cola = new LinkedList();
}
```

La interface Queue cuenta 6 métodos propios de su interface: **Add, Element, Offer, Peek, Poll y Remove**. Y a su vez implementa alguno de los métodos de las otras estructuras: **AddAll, Remove, Clear, Size, Iterator**, entre otros.

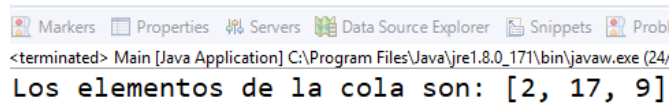
Método Add

Recibe únicamente el dato que se desea guarda, el índice lo asigna la cola en base a los demás datos e índices registrados. Recordar que los índices empiezan el 0.

```
public static void main(String[] args) {
    Queue<Integer> cola = new LinkedList<Integer>();

    cola.add(2);
    cola.add(17);
    cola.add(9);

    System.out.println("Los elementos de la cola son: " + cola);
}
```



Markers Properties Servers Data Source Explorer Snippets Probl
 <terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (24/
 Los elementos de la cola son: [2, 17, 9]

Ilustración 59: Queue - Add.

Fuente: Eclipse

Método Element


Recupera el primer dato de la Cola, pero no elimina a éste.

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

    Queue<Integer> cola = new LinkedList<Integer>();

    cola.add(98);
    cola.add(2);
    cola.add(18);

    System.out.println("El primer elemento es: " + cola.element());
}
```



Markers Properties Servers Data Source Explorer
<terminated> Main [Java Application] C:\Program Files\Ja
El primer elemento es: 98

Ilustración 60: Queue - Element.

Fuente: Eclipse

Método Offer

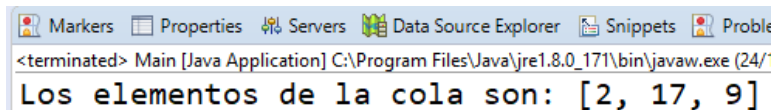
Recibe únicamente el dato que se desea guarda, el índice lo asigna la Cola en base a los demás datos e índices registrados. Recordar que los índices empiezan el 0.

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

    Queue<Integer> cola = new LinkedList<Integer>();

    cola.add(98);
    cola.add(2);
    cola.offer(17);
    cola.offer(9);

    System.out.println("Los elementos de la cola son: " + cola);
}
```



Markers Properties Servers Data Source Explorer Snippets Problem
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (24/'
Los elementos de la cola son: [2, 17, 9]

Ilustración 61: Queue - Offer.

Fuente: Eclipse

Método Peek

Recupera el primer dato presente en la Cola, pero no elimina éste.
o devuelve nulo si esta cola está vacía.

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

    Queue<Integer> cola = new LinkedList<Integer>();

    cola.add(98);
    cola.add(2);
    cola.offer(17);
    cola.offer(9);

    System.out.println("El primer elemento de la cola es: " + cola.peek());
}
```



El primer elemento de la cola es: 98

Ilustración 62: Queue - Peek.

Fuente: Eclipse

Método Poll

Recupera y elimina el primer elemento de la Cola, o devuelve nulo si esta cola está vacía.

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

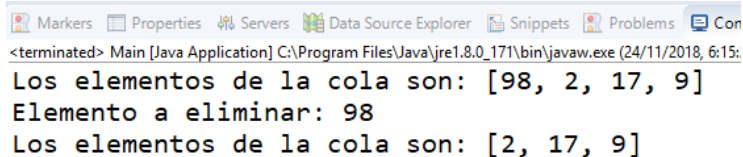
    Queue<Integer> cola = new LinkedList<Integer>();

    cola.add(98);
    cola.add(2);
    cola.offer(17);
    cola.offer(9);

    System.out.println("Los elementos de la cola son: " + cola);

    System.out.println("Elemento a eliminar: " + cola.poll());

    System.out.println("Los elementos de la cola son: " + cola);
}
```



Los elementos de la cola son: [98, 2, 17, 9]
Elemento a eliminar: 98
Los elementos de la cola son: [2, 17, 9]

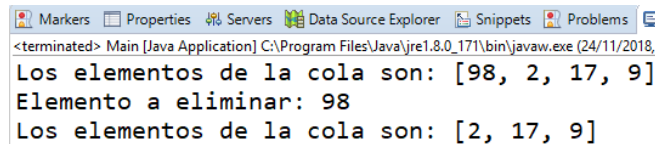
Ilustración 63: Queue - Poll.

Fuente: Eclipse

Método Remove

Recupera y elimina el primer elemento de la Cola.

```
public static void main(String[] args) {  
  
    Queue<Integer> cola = new LinkedList<Integer>();  
  
    cola.add(98);  
    cola.add(2);  
    cola.offer(17);  
    cola.offer(9);  
  
    System.out.println("Los elementos de la cola son: " + cola);  
  
    System.out.println("Elemento a eliminar: " + cola.remove());  
  
    System.out.println("Los elementos de la cola son: " + cola);  
}
```



```
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (24/11/2018,  
Los elementos de la cola son: [98, 2, 17, 9]  
Elemento a eliminar: 98  
Los elementos de la cola son: [2, 17, 9]
```

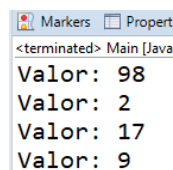
Ilustración 63: Queue - Remove.

Fuente: Eclipse

Método Iterator

Crea una variable de iteración de Cola para recorrer la mismas en el orden en el que se encuentra estructurada.

```
public static void main(String[] args) {  
  
    Queue<Integer> cola = new LinkedList<Integer>();  
  
    cola.add(98);  
    cola.add(2);  
    cola.offer(17);  
    cola.offer(9);  
  
    Iterator<Integer> colaIterable = cola.iterator();  
  
    while(colaIterable.hasNext())  
    {  
        System.out.println("Valor: " + colaIterable.next());  
    }  
}
```



```
<terminated> Main [Java  
Valor: 98  
Valor: 2  
Valor: 17  
Valor: 9
```

Ilustración 64: Queue - Iterator.

Fuente: Eclipse

disponibles para el aprendizaje



Para desarrollar las habilidades y destrezas necesarias en cada competencia, es muy importante que tengas acceso a los recursos didácticos adecuados.

Te sugerimos revisar la siguiente dirección para profundizar los temas acerca de la interface Queue:
<https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html>

TEMA 6

Métodos de Ordenamiento

Los métodos de ordenamiento permiten, como su nombre lo dice, ordenar. En este caso, para ordenar vectores o matrices con valores asignados. Centraremos en los métodos más populares, analizando la cantidad de comparaciones que suceden, el tiempo que demora y revisando el código, escrito en Java, de cada método.

En este informe se conocerá más a fondo cada método distinto de ordenamiento, desde uno simple hasta el más complejo. Se realizarán comparaciones en tiempo de ejecución, pre-requisitos de cada algoritmo, funcionalidad, alcance, etc.

Algunos de los métodos de ordenamiento que se abordarán serán:

- **Burbuja**
- **Inserción**
- **Selección**

Complejidad

Cada método de ordenamiento por definición tiene operaciones y cálculos mínimos y máximos que realiza (complejidad), a continuación, una tabla que indica la cantidad de cálculos que corresponden a cada método de ordenamiento:

Método	Operaciones
Burbuja	$\Omega(n^2)$
Inserción	$\Omega(n^2/4)$
Selección	$\Omega(n^2)$

Método de Ordenamiento Burbuja

Es un sencillo algoritmo de ordenamiento. Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada. Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas".

También es conocido como el método del intercambio directo. Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo uno de los más sencillos de implementar. (wikipedia, 2018). Observe el funcionamiento del método [aquí](#).

Véase la implementación del método ahora.

```
public static void main(String[] args) {  
  
    //Vector a ordenar  
    int vector[] = {9,2,5,7,1,2,0};  
    //Variable auxiliar  
    int temp;  
  
    //Primer ciclo de recorridos  
    for(int i=1;i < vector.length;i++)  
    {  
        //Segundo ciclo de recorridos  
        for (int j=0 ; j < vector.length- 1; j++)  
        {  
            //Comparación de las posiciones y sus valores  
            //Para determinar en mayor de la comparación  
            if (vector[j] > vector[j+1])  
            {  
                //Intercambio de posiciones y valores  
                temp = vector[j];  
                vector[j] = vector[j+1];  
                vector[j+1] = temp;  
            }  
        }  
    }  
}
```

Ilustración 65: Método - Burbuja.

Fuente: Eclipse

El resultado del ordenamiento antes y después de operar el método sería el siguiente:

```
Vector sin ordenar:  
9 - 2 - 5 - 7 - 1 - 2 - 0 -  
Vector ordenado:  
0 - 1 - 2 - 2 - 5 - 7 - 9 -
```

Ilustración 66: Resultado - Burbuja.

Fuente: Eclipse

Método de Ordenamiento Por Inserción

El ordenamiento por inserción es una manera muy natural de ordenar para un ser humano, y puede usarse fácilmente para ordenar un mazo de cartas numeradas en forma arbitraria.

La idea de este algoritmo de ordenación consiste en ir insertando un elemento de la lista o un arreglo en la parte ordenada de la misma, asumiendo que el primer elemento es la parte ordenada, el algoritmo ira comparando un elemento de la parte desordenada de la lista con los elementos de la parte ordenada, insertando el elemento en la posición correcta dentro de la parte ordenada, y así sucesivamente hasta obtener la lista ordenada. (lwh, 2018) Observe el método funcionando [aquí](#).

Véase la implementación del método ahora.

```
public static void main(String[] args) {  
  
    //Vector a ordenar  
    int vector[] = {9,2,5,7,1,2,0};  
  
    //Primer ciclo de recorrido  
    for(int i = 0; i < vector.length; i++)  
    {  
        //Declaración y asignación de la variable auxiliar  
        int aux = vector[i];  
        //Declaración de la variable del ciclo  
        int j;  
        //Segundo ciclo de recorrido  
        for (j = i-1; j >= 0 && vector[j] > aux; j--)  
        {  
            //Intercambio de valores en posiciones  
            vector[j+1] = vector[j];  
        }  
        //Asignación de valores al ciclo  
        vector[j+1] = aux;  
    }  
}
```

Ilustración 67: Método - Inserción.

Fuente: Eclipse

El resultado del ordenamiento antes y después de operar el método sería el siguiente:

```
Vector Sin Ordenar  
9 - 2 - 5 - 7 - 1 - 2 - 0 -  
Vector Ordenado  
0 - 1 - 2 - 2 - 5 - 7 - 9 -
```

Ilustración 68: Resultado - Inserción.

Fuente: Eclipse

Método de Ordenamiento por Selección

Consiste en encontrar el menor de todos los elementos del arreglo o vector e intercambiarlo con el que está en la primera posición. Luego el segundo más pequeño, y así sucesivamente hasta ordenarlo todo. Su implementación requiere $O(n^2)$ comparaciones e intercambios para ordenar una secuencia de elementos. (ecured, 2018)

Este algoritmo mejora ligeramente el algoritmo de la burbuja. En el caso de tener que ordenar un vector de enteros, esta mejora no es muy sustancial, pero cuando hay que ordenar un vector de estructuras más complejas, la operación de intercambiar los elementos sería más costosa en este caso. Observe el método funcionamiento [aquí](#).

Véase la implementación del método ahora.

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

    //Vector a ordenar
    int vector[] = {9,2,5,7,1,2,0};
    //Variable auxiliar
    int temp;

    //Primer ciclo de recorrido
    for(int k = 0; k <= vector.length-1; k++)
    {
        //Almacenado de variable de recorrido
        int p = k;

        //Segundo ciclo de recorrido
        for(int i = k+1; i <= vector.length-1; i++)
        {
            //Comparación de valores y almacenaje de posiciones
            if(vector[i] < vector[p]) p = i;

            //Comparación de posiciones
            if(p != k)
            {
                //Intercambio de valores en posiciones
                temp = vector[p];
                vector[p] = vector[k];
                vector[k] = temp;
            }
        }
    }
}
```

Ilustración 69: Método - Selección.

Fuente: Eclipse

El resultado del ordenamiento antes y después de operar el método sería el siguiente:

```
Vector sin ordenar
9 - 2 - 5 - 7 - 1 - 2 - 0 -
Vector ordenado
0 - 1 - 2 - 2 - 7 - 5 - 9 -
```

Ilustración 70: Resultado - Selección.

Fuente: Eclipse

disponibles para el aprendizaje



Para desarrollar las habilidades y destrezas necesarias en cada competencia, es muy importante que tengas acceso a los recursos didácticos adecuados.

Te sugerimos consultar los siguientes métodos:

- Shell Short
- Merge
- Quick

TEMA 7 Recursividad

Las funciones recursivas son aquellas que se invocan a sí mismas en algún momento de su ejecución.

En análisis de Algoritmos las técnicas recursivas se usan mucho para la solución de problemas. Esta forma en análisis de algoritmos es llamada Divide y Vencerás.

Para poder resolver un problema de forma recursiva es necesario saber alguna solución no recursiva para alguno de los casos más sencillos. *"Usamos la solución más simple para resolver un problema más complejo."*

Así, todo método recursivo debe tener al menos una sentencia que devuelva un resultado (la solución del caso más sencillo) y las sentencias necesarias para acercarse en cada invocación a ese caso.

La recursión permite programar algoritmos aparentemente complicados con un código simple y claro, ahorrando trabajo al programador. A simple vista parece la solución perfecta para muchos

problemas, pero hay que tener en cuenta que en ocasiones ralentizará el programa en exceso. (wikibooks, 2018).

Cuando un método se llama a sí mismo, a las nuevas variables y parámetros locales se les asigna almacenamiento en la pila (stack), y el código del método se ejecuta con estas nuevas variables desde el principio. Una llamada recursiva no hace una nueva copia del método. Solo los argumentos son nuevos. A medida que retorna o devuelve cada llamada recursiva, las viejas variables y parámetros locales se eliminan de la pila, y la ejecución se reanuda en el punto de la llamada dentro del método. Se podría decir que los métodos recursivos se “desplazan” hacia afuera y hacia atrás.



Ilustración 71: Recursividad gráfica

Fuente: geekytheory

Ejercicio de factorización

Calcular el [factorial](#) de un número con recursividad es el típico ejemplo para explicar este método de programación. Recuerda que el factorial de un número es multiplicar dicho número por todos sus anteriores hasta llegar a 1.

Clase Recursividad – Método Factorial:

```
package Clase;

public class Recursividad
{
    public int factorial(int numero)
    {
        if ( numero <= 1 )
        {
            return 1;
        }
        else
        {
            return numero*factorial(numero-1);
        }
    }
}
```

Ilustración 72: Recursividad – Factorial.

Fuente: Eclipse

Clase Main – Ejecución de factorial:

```
package Clase;

public class Main {

    public static void main(String[] args)
    {
        Recursividad recursividad = new Recursividad();

        System.out.println("El factorial de 5 es: " + recursividad.factorial(5));
    }
}
```

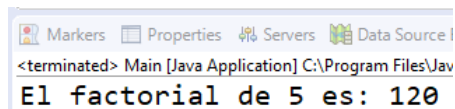


Ilustración 73: Factorial - Ejecución

Fuente: Eclipse

Para evaluar esta expresión, se llama a factorial () con n-1. Este proceso se repite hasta que n es igual a 1 y las llamadas al método comienzan a devolver. Por ejemplo, cuando se calcula el factorial de 2, la primera llamada a factorial () hará que se realice una segunda

llamada con un argumento de 1. Esta llamada devolverá 1, que luego se multiplicará por 2 (el valor original de n). La respuesta es 2.

Algunos casos de uso en la recursividad puedes encontrarlo en el siguiente enlace: <https://www.discoduroderoer.es/ejercicios-propuestos-y-resueltos-de-recursividad-java/>:

La solución iterativa es fácil de entender. Utiliza una variable para acumular los productos y obtener la solución. En la solución recursiva se realizan llamadas al propio método con valores de n cada vez más pequeños para resolver el problema.

Cada vez que se produce una nueva llamada al método se crean en memoria de nuevo las variables y comienza la ejecución del nuevo método.

Para entender el funcionamiento de la recursividad, se puede pensar que cada llamada supone hacerlo a un método diferente, copia del original, que se ejecuta y devuelve el resultado a quien lo llamó.

Un método recursivo debe contener:

Uno o más casos base: casos para los que existe una solución directa.

Una o más llamadas recursivas: casos en los que se llama sí mismo
Caso base: Siempre ha de existir uno o más casos en los que los valores de los parámetros de entrada permitan al método devolver un resultado directo. Estos casos también se conocen como solución trivial del problema.

En el ejemplo del factorial el caso base es la condición:

```
if ( numero <= 1 )  
{  
    return 1;  
}
```

Si $n = 0$ el resultado directo es 1. No se produce llamada recursiva

Llamada recursiva: Si los valores de los parámetros de entrada no cumplen la condición del caso base se llama recursivamente al método. En las llamadas recursivas el valor del parámetro en la llamada se ha de modificar de forma que se aproxime cada vez más hasta alcanzar al valor del caso base.

En el ejemplo del factorial en cada llamada recursiva se utiliza $n-1$

```
return numero*factorial(numero-1);
```

Por lo que en cada llamada el valor de n se acerca más a 0 que es el caso base.

La recursividad es especialmente apropiada cuando el problema a resolver (por ejemplo, cálculo del factorial de un número) o la estructura de datos a procesar (por ejemplo los árboles) tienen una clara definición recursiva.

No se debe utilizar la recursión cuando la iteración ofrece una solución obvia. Cuando el problema se pueda definir mejor de una forma recursiva que iterativa lo resolveremos utilizando recursividad.

Para medir la eficacia de un algoritmo recursivo se tienen en cuenta tres factores:

- Tiempo de ejecución
- Uso de memoria
- Legibilidad y facilidad de comprensión

Las soluciones recursivas suelen ser más lentas que las iterativas por el tiempo empleado en la gestión de las sucesivas llamadas a los métodos. Además, consumen más memoria ya que se deben guardar los contextos de ejecución de cada método que se llama.

A pesar de estos inconvenientes, en ciertos problemas, la recursividad conduce a soluciones que son mucho más fáciles de leer y comprender que su correspondiente solución iterativa. En estos casos una mayor claridad del algoritmo puede compensar el coste en tiempo y en ocupación de memoria.

De todas maneras, numerosos problemas son difíciles de resolver con soluciones iterativas, y sólo la solución recursiva conduce a la resolución del problema (por ejemplo, Torres de Hanoi o recorrido de Árboles).

disponibles para el aprendizaje



Para desarrollar las habilidades y destrezas necesarias en cada competencia, es muy importante que tengas acceso a los recursos didácticos adecuados.

La recursividad presenta una nueva forma de solucionar problemas en Java de formas simples para resolver problemas de gran tamaño, te sugerimos revisar el siguiente vídeo para entender un poco más su uso y aplicación:
https://www.youtube.com/watch?v=zN-p9_51FFg

Recursos disponibles para el aprendizaje



Para desarrollar las habilidades y destrezas necesarias en cada competencia, es muy importante que tengas acceso a los recursos didácticos adecuados.

Entonces, si necesitas reforzar esta información, te sugerimos revisar nuevamente los **Vídeos de Apoyos**, disponibles en el campus virtual, indicados en las lecturas anteriores. Además, recuerda que puede consultar las **Fuentes Documentales** que aparecen en esta guía, particularmente, en el apartado de Referencias Bibliográficas.

ASPECTOS CLAVES

Recuerda tener muy presente los conceptos visto en esta guía número 4 dado que en el transcurso del diplomado se tendrán en cuenta continuamente para su implementación.

Recuerda algunos aspectos abordados en el módulo:

- Comprender la diferencia entre el uso de las dos estructuras para manejar datos por teclado.
- Scanner permite de entrada diferentes tipos de datos.
- BufferedReader permite la lectura únicamente de Strings.
- Hay estructuras de datos que comparten métodos en común.
- Entender que la mayoría de estructuras de datos internamente funcionan como listas.
- Diferenciar el uso y complejidad de los métodos de ordenamiento.
- La definición correcta del caso base de un método recursivo marca la diferencia en su efectividad.
- No olvidar los temas vistos en los módulos pasados.

¡Felicidades! 🍀 Has concluido con la lectura de la Guía Didáctica N°4. Así que ya puedes realizar la Actividad Evaluativa.

REFERENCIAS BIBLIOGRÁFICAS

- wikibooks. (26 de 11 de 2018). Obtenido de wikibooks:
https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_Java/Funciones_recurativas
- codigolibre. (23 de 11 de 2018). *codigolibre*. Obtenido de codigolibre:
<http://codigolibre.weebly.com/blog/pilas-en-java>
- ecured. (24 de 11 de 2018). *ecured*. Obtenido de ecured:
[https://www.ecured.cu/Cola_\(Estructura_de_datos\)](https://www.ecured.cu/Cola_(Estructura_de_datos))
- ecured. (26 de 11 de 2018). *ecured*. Obtenido de ecured:
https://www.ecured.cu/Algoritmo_de_ordenamiento_por_selecci%C3%B3n
- fciencias. (21 de 11 de 2018). *fciencias*. Obtenido de fciencias:
<http://hp.fciencias.unam.mx/~alg/estructurasDeDatos/>
- geeksforgeeks. (23 de 11 de 2018). *geeksforgeeks*. Obtenido de geeksforgeeks:
<https://www.geeksforgeeks.org/linked-list-in-java/>
- jenkov. (22 de 11 de 2018). *jenkov*. Obtenido de jenkov: <http://tutorials.jenkov.com/java-collections/list.html>
- lwh. (26 de 11 de 2018). *lwh*. Obtenido de lwh:
http://lwh.free.fr/pages/algo/tri/tri_insertion_es.html
- RoverOportunity2012. (24 de 11 de 2018). *RoverOportunity2012*. Obtenido de RoverOportunity2012: <https://es.slideshare.net/RoverOportunity2012/java-pilas-ycolas>
- wikipedia. (24 de 11 de 2018). *wikipedia*. Obtenido de wikipedia:
https://es.wikipedia.org/wiki/Ordenamiento_de_burbuja

Esta guía fue elaborada para ser utilizada con fines didácticos como material de consulta de los participantes en el Diplomado Virtual en Programación en Java del Politécnico de Colombia, especialmente, a los técnicos, tecnólogos y profesionales de carreras afines, estudiantes de todas las carreras, empíricos, y público en general con conocimientos básicos en informática que intentan entrar en el mundo de la programación, que se desempeñen o no en las áreas de TIC de cualquier tipo de organización y que deseen obtener las competencias y habilidades necesarias para conocer los fundamentos prácticos del lenguaje de programación Java para la aplicación y desarrollo de algoritmos y aplicaciones, y solo podrá ser reproducida con esos fines. Por lo tanto, se agradece a los usuarios referirla en los escritos donde se utilice la información que aquí se presenta.

Derechos reservados - POLITÉCNICO DE COLOMBIA, 2018
Medellín, Colombia