



DIPLOMADO VIRTUAL EN
PROGRAMACIÓN EN JAVA
Guía didáctica 4: Estructuras de datos



Formación Virtual

.....educación sin límites



Competencia específica

Se espera que con los temas abordados en la guía didáctica del módulo 4: Estructuras de datos, el estudiante logre la siguiente competencia específica:

- Describir las diferentes estructuras de datos para su diseño y almacenamiento.



Contenidos temáticos

Los contenidos temáticos para desarrollar en la guía didáctica del módulo 4: Estructuras de datos, son:



Ilustración 1: caracterización de la guía didáctica.

Fuente: autor

Tema 1: Introducción

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. (López, 2011).

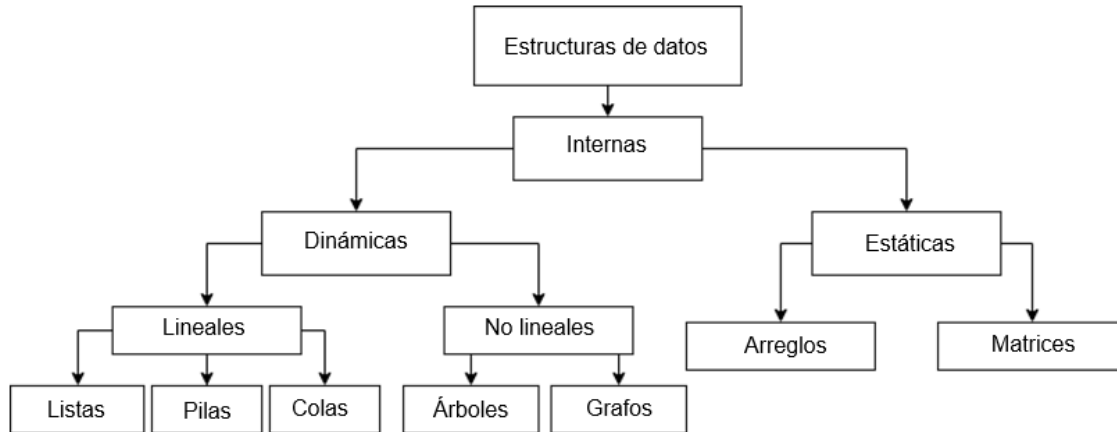


Ilustración 2: estructuras de datos.

Fuente: autor, a partir de Draw.io.

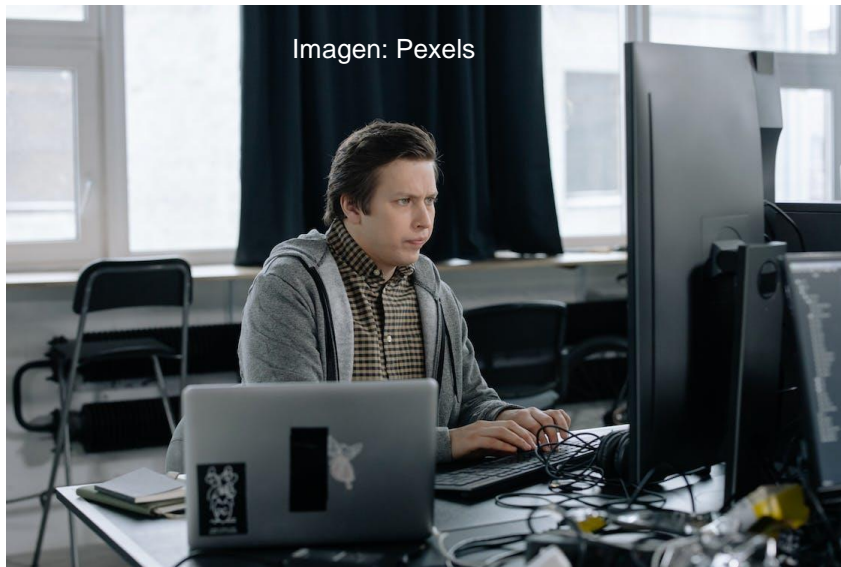
Las principales estructuras que abordaremos en esta guía número 4 son las **estructuras dinámicas**, entre las que encuentran en especial las lineales: **listas**, **pilas** y **colas**, además de todos los métodos y características que las componen.

Es necesario que consideremos 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 rigen por ciertas reglas/restricciones dadas por la forma en que están construidas internamente.

Tema 2: 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 abordarán en esta guía son:

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

Estas 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, en cuya aplicación 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 conocerán primero sus definiciones, características y creación. Posteriormente, por medio de la interfaz *List*, se aplicarán de manera detallada 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 según el orden en que aparecen internamente en el Java *List*. El orden de los elementos es el motivo por el cual esta estructura de datos se denomina «lista».

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* y *ArrayList* (aunque más adelante se describirá específicamente el uso de esta).

```
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>();
    }
}
```

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 pueden implementar las funcionalidades desde lo que se conoce como CRUD (siglas de *create* – *read* – *update* – *delete*; en español: crear – leer – actualizar – eliminar).

Agregar elementos a una *List*

La inserción de datos en una lista no es más que la acción de agregar elementos a esta basándose en el 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. Veamos.

Método *add*: recibe únicamente el dato que se desea guardar, el índice lo asigna la lista con base en los demás datos registrados. Recordemos que los índices empiezan en 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");
}
```

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

Método *add* con índice: recibe el dato que se desea guardar y el índice que se le desea asignar al dato. Hay que tener presente que no existan datos en posiciones donde se desee 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");

}
```

Método *add list*: recibe todos los elementos de una lista nueva, esta 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);

}
```

Método *add list 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 desee 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);
}
```

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.

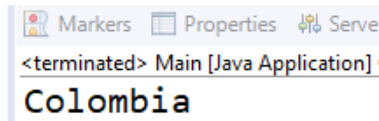
```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>();
    lista.add(0, "Colombia");
    lista.add(1, "Chile");
    lista.add(2, "Argentina");
    lista.add(3, "Venezuela");
    lista.add(4, "Perú");

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

Método get; recupera un valor de la lista a partir de una posición determinada. Recordemos 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

Método *size*: devuelve el tamaño de la lista a partir de los valores existentes dentro de esta.

```
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());
}
```



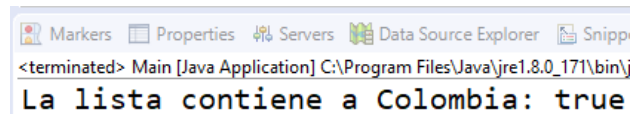
Markers Properties Data Source Explo
<terminated> Main [Java Application] C:\Program Files\Java\jre
El tamaño de la lista es: 4

Método *contains*: 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"));
}
```



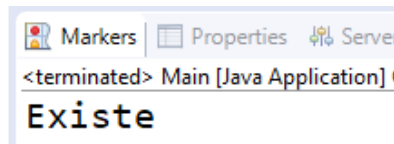
Markers Properties Data Source Explorer Snipp
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\j
La lista contiene a Colombia: true

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");
    }
}
```



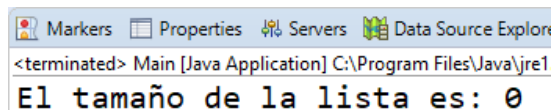
Método *clear*: limpia todos los datos de la lista de forma que queda completamente vacía.

```
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());
}
```



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");
    }
}
```



La lista no está vacía

Método *remove* por índice: recibe índice válido dentro de la lista y elimina el dato que se encuentre en esta.

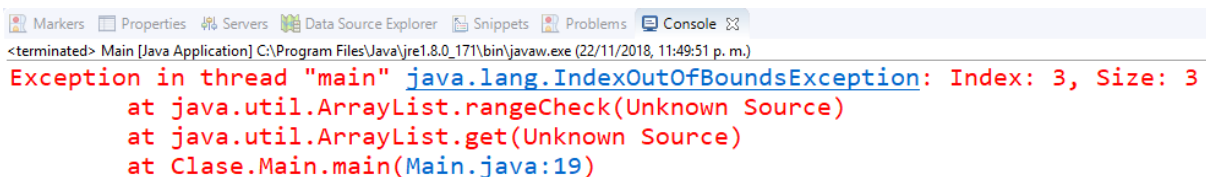
```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>();

    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));
}
```

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



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.

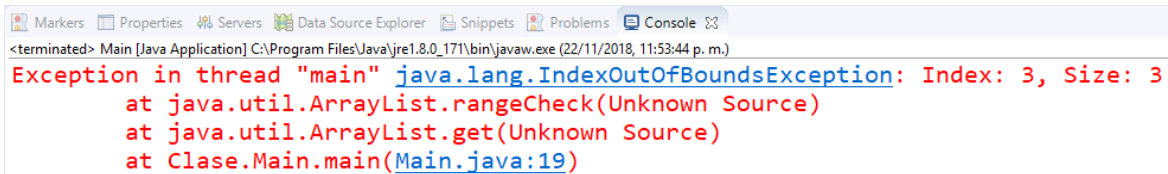
```
public static void main(String[] args)
{
    List<String> lista = new ArrayList<String>();

    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));
}
```

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



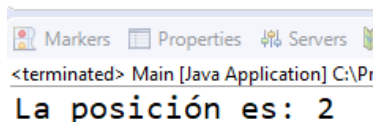
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 3, Size: 3
at java.util.ArrayList.rangeCheck(Unknown Source)
at java.util.ArrayList.get(Unknown Source)
at Clase.Main.main(Main.java:19)

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"));
}
```



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

Método *iterator*: crea una variable de iteración de listas para recorrer la mismas en el orden en el que se encuentran almacenadas.


```
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());
    }
}
```



```
<terminated> Main [Java Application] C:\Progra
Valor: Colombia
Valor: Ecuador
Valor: Venezuela
Valor: Argentina
```

Se debe importar de igual forma *iterator* para un correcto funcionamiento.

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

Estos son unos de los métodos principales del interfaz *List*. Existen muchos otros métodos que se derivan de los anteriores y cuya implementación no dista mucho de la vista en estos.

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 inicializar 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.
- Existen *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>();
    }
}
```

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 a los vectores, estos pueden recibir un tamaño establecido de entrada.

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

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 a la facilidad de las inserciones y eliminaciones, se prefieren a las matrices. También tienen algunas desventajas, pues no podemos 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.

Algunas características que se deben considerar frente al uso de *LinkedList*:

- No se puede inicializar 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>();
    }
}
```

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 guardar, el índice lo asigna la lista con base en los demás datos e índices registrados. Recordemos que los índices empiezan en 0.

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

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

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

Método *add* con índice: recibe el dato que se desea guardar y el índice que se le desea asignar al dato. Hay que tener presente que no existan datos en posiciones donde se desee 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");
}
```

Método *add LinkedList*: recibe todos los elementos de una lista nueva, esta 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);
}
```

Método *add LinkedList* con índice: recibe todos los elementos de una lista nueva en una posición especificada. Hay que tener en cuenta que no existan datos en posiciones donde se desee 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);
}
```

Método *AddFirst*: recibe únicamente el dato que se desea guardar, 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");
}
```

Método *AddLast*: recibe únicamente el dato que se desea guardar, el índice asignado depende de la lista, dado que este debe quedar en la última posición. Si no existe última, este 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");
}
```

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");

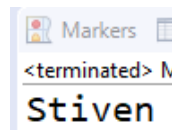
    lista.set(1, "Juan");
}
```

Método *get*: recupera un valor de la lista a partir de una posición determinada. Recordemos 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));
}
```



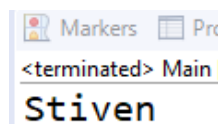
Markers
<terminated> Main
Stiven

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 Proc
<terminated> Main
Stiven

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());
}
```

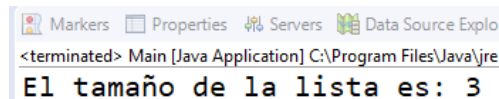


Método *size*: devuelve el tamaño de la lista a partir de los valores existentes dentro de esta.

```
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());
}
```

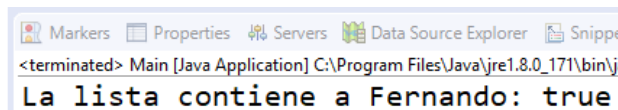


Método *contains*: recibe un valor del mismo tipo y determina si existe o no dentro de la lista. Devuelve *true* si existe, y falso en caso contrario.

```
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"));
}
```

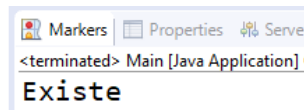


Este método puede ser usado dentro de condicionales por el tipo booleano de retorno.

```
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");
    }
}
```



Markers Properties Servers
<terminated> Main [Java Application] !
Existe

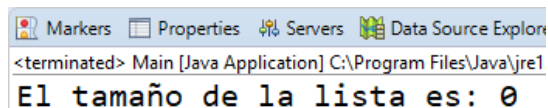
Método *clear*: limpia todos los datos de la lista de forma que esta queda completamente vacía.

```
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());
}
```



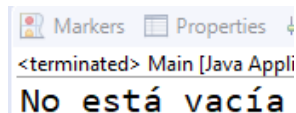
Markers Properties Servers Data Source Explorer
<terminated> Main [Java Application] C:\Program Files\Java\jre1
El tamaño de la lista es: 0

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");
    }
}
```



Markers Properties
<terminated> Main [Java Appli
No está vacía

Método *remove*: recibe índice válido dentro de la lista y elimina el dato que se encuentre en esta. La lista se ordena de forma que no quedan 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();
}
```

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");
}
```

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);
}
```

Método *offer*: recibe únicamente el dato que se desea guardar, el índice asignado depende de la lista, dado que este debe quedar en la última posición. Si no existe última, este 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");
}
```

Método *OfferFirst*: recibe únicamente el dato que se desea guardar, 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("Mia");

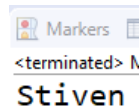
    lista.add(2, "Jose");
}
```

Método *peek*: recupera pero 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.peek());
}
```



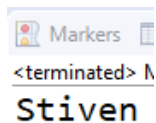
Markers
<terminated> M
Stiven

Método *PeekFirst*: recupera pero 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.peekFirst());
}
```



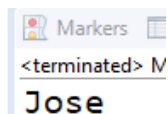
Markers
<terminated> M
Stiven

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());
}
```



Markers
<terminated> M
Jose

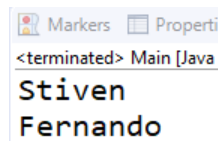
Método *poll*: recupera y elimina el primer elemento de la lista.

```
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));
}
```



Markers Property
<terminated> Main [Java]
Stiven
Fernando

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));
}
```

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());
}
```

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());
}
```



Método *push*: recibe únicamente el dato que se desea guardar, 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");
}
```

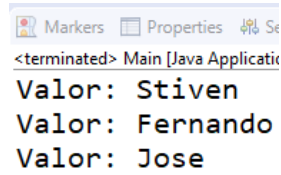
Método *LastIterator*: crea una variable de iteración de listas para recorrerlas en el orden en el que se encuentran almacenadas.

```
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());
    }
}
```



```
<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;
```

Estos son unos de los métodos principales de la clase *LinkedList*, existen muchos otros que se derivan de los anteriores y cuya implementación no va más allá de la vista en estos.

Tema 3: Pilas

Una pila es una versión restringida de una lista (*stack*, en inglés) es una estructura de datos lineal que solo tiene 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) (Código Libre, 2015).

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 vuelven a volcar los elementos de la pila auxiliar a la original (Código Libre, 2015)

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>();
    }
}
```

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 con cinco 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");
}
```

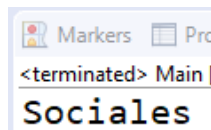
Método *pop*

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());
}
```



Markers Proc
<terminated> Main |
Sociales

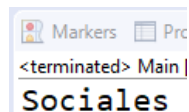
Método *peek*

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());
}
```



Markers Proc
<terminated> Main |
Sociales

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());
}
```



Markers Properties Servers Data Source Explorer
<terminated> Main [Java Application] C:\Program Files\Java\jre
¿La pila está vacía?: false

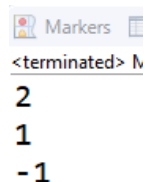
Método *search*

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"));
}
```

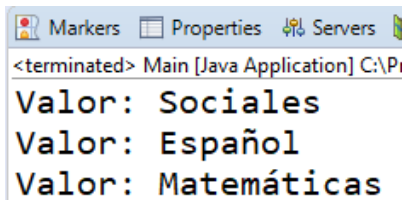


Markers
<terminated> N
2
1
-1

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

Tema 4: Colas

Una cola es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción *push* se realiza por un extremo y la operación de extracción *pop* 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. (Ecu Red, 2012)

«Este tipo de estructura de datos abstracta se implementa en lenguajes orientados a objetos mediante clases, en forma de listas enlazadas». (Cruz, 2012)

La particularidad de una estructura de datos de cola es el hecho de que solo puede accederse al primer y al último elemento de la estructura. Así mismo, los elementos solo se pueden eliminar por el principio y solo 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 añade un nuevo elemento. (Ecu Red, 2012)

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 tiene seis métodos propios de su interfaz.
- La interfaz 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 interfaz de este tipo, es necesario hacer la respectiva importación del paquete de Java que se desea trabajar, *queue* opera con una interfaz: *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>();
    }
}
```

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 interfaz *queue* cuenta con seis métodos propios de su interfaz: ***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 guardar, el índice lo asigna la cola partiendo de los demás datos e índices registrados. Recordemos que los índices empiezan en 0.

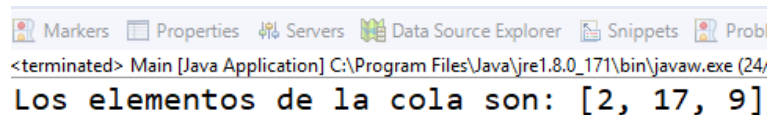
```
public static void main(String[] args) {

    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]

Método *element*

Recupera el primer dato de la cola, pero no lo elimina.

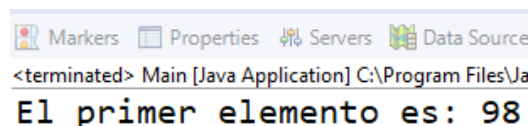
```
public static void main(String[] args) {

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

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

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

}
```

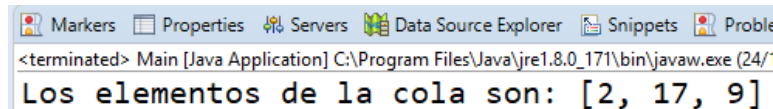


Markers Properties Servers Data Source
<terminated> Main [Java Application] C:\Program Files\Ja
El primer elemento es: 98

Método *offer*

Recibe únicamente el dato que se desea guardar, el índice lo asigna la cola a partir de los demás datos e índices registrados. Recordemos que los índices empiezan en 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]

Método *peek*

Recupera el primer dato presente en la cola, pero no lo elimina, 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());  
}
```



Markers Properties Servers Data Source Explorer Snippets Problem
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.
El primer elemento de la cola es: 98

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) {

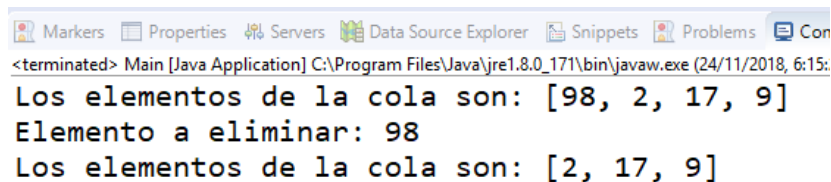
    Queue<Integer> cola = new LinkedList<Integer>();

    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);
}
```



```
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (24/11/2018, 6:15:
Los elementos de la cola son: [98, 2, 17, 9]
Elemento a eliminar: 98
Los elementos de la cola son: [2, 17, 9]
```

Método *remove*

Recupera y elimina el primer elemento de la cola.

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

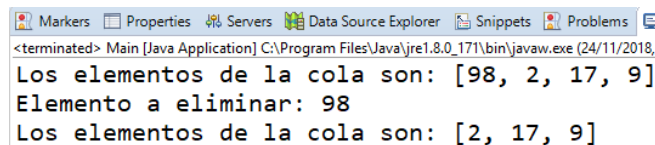
    Queue<Integer> cola = new LinkedList<Integer>();

    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);
}
```

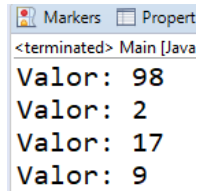


```
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (24/11/2018,
Los elementos de la cola son: [98, 2, 17, 9]
Elemento a eliminar: 98
Los elementos de la cola son: [2, 17, 9]
```

Método *iterator*

Crea una variable de iteración de cola para recorrer la misma 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());  
    }  
}
```



Markers Properties
<terminated> Main [Java]
Valor: 98
Valor: 2
Valor: 17
Valor: 9

Tema 5: 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. (Wikilibros, 2019).



Imagen: Pixabay

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.

Ejercicio de factorial

Calcular el [factorial](#) (Wikipedia, 2023b) de un número con recursividad es el típico ejemplo para explicar este método de programación. Recordemos que el factorial de un número consiste en multiplicar dicho número por todos sus anteriores hasta llegar a 1.

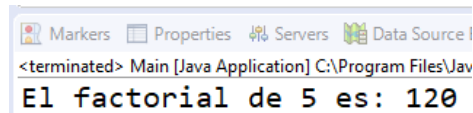
Clase recursividad – método factorial:

```
package Clase;

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

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));  
    }  
}
```



Markers Properties Servers Data Source I
<terminated> Main [Java Application] C:\Program Files\Jav
El factorial de 5 es: 120

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 se encuentran en el siguiente enlace: <https://www.discoduroderoer.es/ejercicios-propuestos-y-resueltos-de-recursividad-java/> (Disco Duro de Roer, 2017).

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 para resolver (por ejemplo, cálculo del factorial de un número) o la estructura de datos por 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 solo la solución recursiva conduce a la resolución del problema (por ejemplo, torres de Hanoi). (Puntocom No Es un Lenguaje, 2012)

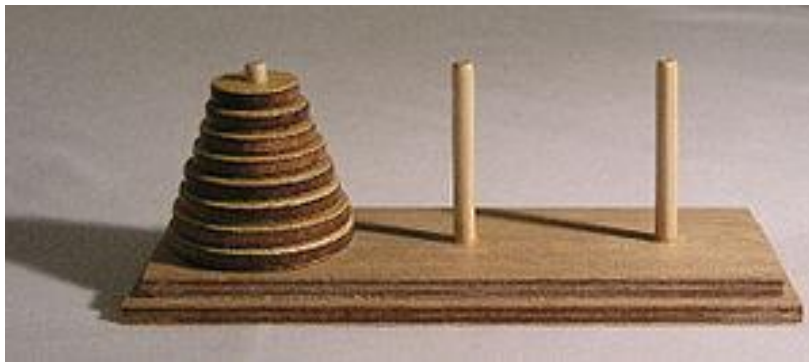


Imagen: Wikipedia

Tema 6: Métodos de ordenamiento

Los métodos de ordenamiento facilitan la acción de ordenar. En este caso, sirven para ordenar vectores o matrices con valores asignados. Nos centraremos en los métodos más populares, analizando la cantidad de comparaciones que suceden, el tiempo que demora, revisando el código escrito en Java, de cada método.

Para conocer más a fondo cada método distinto de ordenamiento, desde uno simple hasta el más complejo, se realizarán comparaciones en tiempo de ejecución, prerequisites de cada algoritmo, funcionalidad, alcance, etc.

Algunos de los métodos de ordenamiento que abordaremos son:

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

Complejidad

Cada método de ordenamiento, por definición, tiene operaciones y cálculos mínimos y máximos que realiza, esto se conoce como 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	$\Omega(n^2)$

Tabla 1: cantidad de cálculos correspondientes a cada método de ordenamiento.
Fuente: autor.

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, 2023a).

Para observar el funcionamiento del método, clic [aquí](#) (Wikipedia, 2023a).

Veamos 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;  
            }  
        }  
    }  
}
```

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 3: resultado del ordenamiento antes y después de operar el método.
Fuente: autor.

Método de ordenamiento por inserción

El ordenamiento por inserción es una forma bastante natural de poner objetos en orden 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 esta, asumiendo que el primer elemento es la parte ordenada; el algoritmo irá 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 Free, s.f.).

Para observar el método funcionando, clic [aquí](#). (Wikimedia, s.f.)

Veamos 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;  
    }  
}
```

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 4: resultado del ordenamiento antes y después de operar el método.
Fuente: autor.

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.

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. (EcuRed, 2019)

Para observar el método funcionamiento, clic [aquí](#). (Wikimedia, s.f.).

Veamos 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;  
            }  
        }  
    }  
}
```

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 5: resultado del ordenamiento antes y después de operar el método.
Fuente: autor.

Tema 7: Datos por teclado

La entrada o lectura de datos en Java es uno de los conceptos más importantes al momento de interactuar con los usuarios en los programas, dado que se dejan a un lado los datos estáticos definidos previamente y se 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 veremos 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 es simple. Veamos.

```
package Clase;

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

public class Main {

    public static void main(String[] args)
    {

    }

}
```

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 puede ser la siguiente:

```
package Clase;

import java.io.*;

public class Main {

    public static void main(String[] args)
    {

    }

}
```

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.");

    }

}
```

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

Lo primero por 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 utilizar estas clases, se debe crear un objeto de estas 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.

Es necesario 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, y la variable almacena todo lo anterior a esta).

```
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* e *InputStreamReader* (guía didáctica 1).

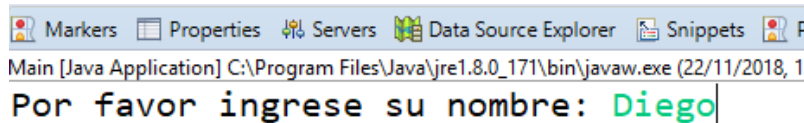
```
int edad = Integer.parseInt(entrada);
```

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

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

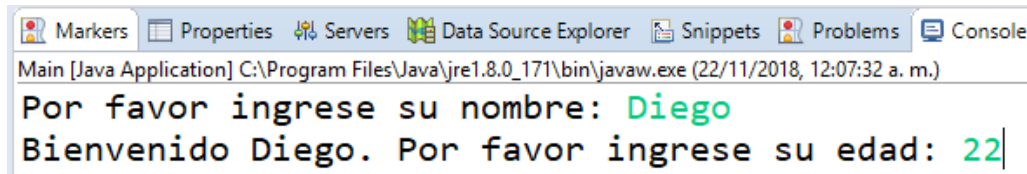
El resultado en tiempo de ejecución, a diferencia de como se ha trabajado durante el diplomado, es secuencial; la ejecución espera que se ingresen datos para continuar con la siguiente instrucción hasta imprimir todo el código.

- Ejecución del primer bloque.



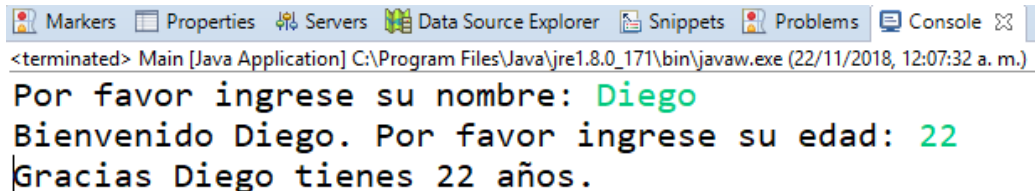
Markers Properties Servers Data Source Explorer Snippets P
Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (22/11/2018, 1.
Por favor ingrese su nombre: **Diego**

- 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* e *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 es necesario trabajar con otros tipos de datos. Representa más trabajo y reprocesos que son innecesarios.

La clase **scanner**, por su parte, cuenta con métodos independientes para cada tipo. Veamos ahora la clase **scanner** a profundidad.

Scanner

Es una clase que 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 es simple. Veamos su declaración.

```
package Clase;

import java.util.Scanner;

public class Main {

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

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)
    {
        }
    }
}
```

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.");
    }
}
```


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

Para utilizar esta clase, al igual de *BufferedReader* e *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, si es una cadena *nextLine*, si es un entero *nextInt*, si 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 esta).

```
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*.

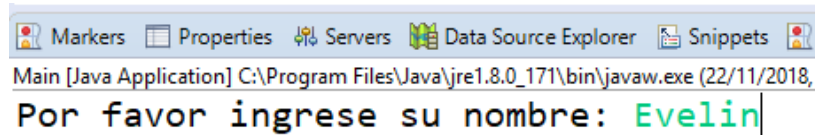
```
int edad = sc.nextInt();
```

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

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

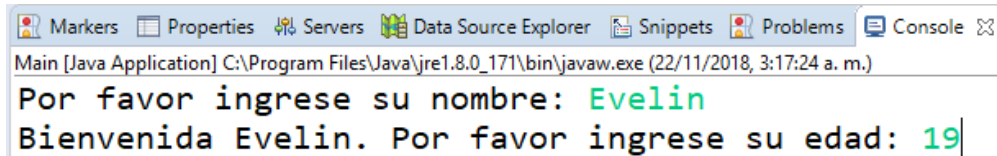
El resultado en tiempo de ejecución, a diferencia de como se ha trabajado durante el diplomado, es secuencial; la ejecución espera que se ingresen datos para continuar con la siguiente instrucción hasta imprimir todo el código.

- Ejecución del primer bloque.



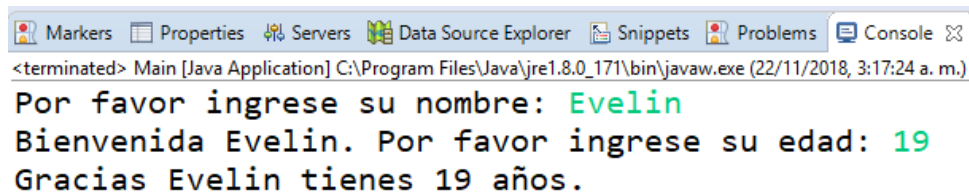
Markers Properties Servers Data Source Explorer Snippets
Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (22/11/2018,
Por favor ingrese su nombre: Evelin

- 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* e *InputStreamReader*, **scanner** permite trabajar con otros tipos de datos, evitando reprocesos, casteando datos y reduciendo líneas de código.



Referencias bibliográficas

- Eclipse Foundation. (2023). *Eclipse*. Eclipse Foundation. <https://www.eclipse.org/>
- Código Libre. (2015, 8 de noviembre). Pilas en Java. *Código Libre*.
<http://codigolibre.weebly.com/blog/pilas-en-java>
- Cruz, M. (2012, 31 de octubre). Colas en C++. Martín Cruz.
<https://blog.martincruz.me/2012/10/colas-en-c.html>
- Disco Duro de Roer. (2017, 29 de diciembre). Ejercicios propuestos y resueltos de recursividad Java. *Disco Duro de Roer*.
<https://www.discoduroderoer.es/ejercicios-propuestos-y-resueltos-de-recursividad-java/>
- EcuRed. (2012, 1 de agosto). Cola (Estrcutura de datos). En *EcuRed*.
[https://www.ecured.cu/Cola_\(Estructura_de_datos\)](https://www.ecured.cu/Cola_(Estructura_de_datos))
- EcuRed. (2019, 10 de julio). Algoritmo de ordenamiento por selección. En *EcuRed*.
https://www.ecured.cu/Algoritmo_de_ordenamiento_por_selecci%C3%B3n
- López, A. (2011). *Estructura de datos con Java: un enfoque práctico*. Facultad de Ciencias UNAM. <http://hp.fciencias.unam.mx/~alg/estructurasDeDatos/>
- Puntocom No Es un Lenguaje. (2012, abril). Recursividad en Java. Programación Java. *BlogSpot*. <https://tinyurl.com/yrrz7hcc>
- Wikilibros. (2019, 5 de abril). Programación en Java/Funciones recursivas. En *Wikibooks*. <https://tinyurl.com/yeysredp>
- Wikimedia. (s.f.). *Simulación del algoritmo* [animación]. Wikimedia. [Selection-Sort-Animation.gif \(100x371\) \(wikimedia.org\)](#)
- Wikipedia. (2023a, 20 de septiembre). Ordenamiento de burbuja. En *Wikipedia*.
https://es.wikipedia.org/wiki/Ordenamiento_de_burbuja
- Wikipedia. (2023b, 26 de octubre). Factorial. En *Wikipedia*.
<https://es.wikipedia.org/wiki/Factorial>



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, 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.

GUÍA DIDÁCTICA 4

M2-DV59-GU04

MÓDULO 4: ESTRUCTURAS DE DATOS

© DERECHOS RESERVADOS - POLITÉCNICO DE COLOMBIA, 2023
Medellín, Colombia

Proceso: Gestión Académica Virtual
Realización del texto: Diego Palacio, docente
Revisión del texto: Comité de Revisión
Diseño: Comunicaciones

Editado por el Politécnico de Colombia