

# Contenido

- Tipo de dato abstracto (TDA): Listas, Pilas y Colas
- Listas, Pilas y Colas con herencia
- El framework de colecciones en Java
  - Tecnologías de almacenamiento
  - Implementaciones de listas provistas por la API: LinkedList y ArrayList
- Tipos Genéricos
- Iteradores

# Tipo de dato abstracto (TDA)

## Listas, Pilas y Colas

Un tipo de dato abstracto (TDA) (en inglés, ADT por Abstract Data Type) es un tipo definido solamente en términos de sus operaciones y de las restricciones que valen entre las operaciones.

- Cada TDA se representa en esta materia con una clase o varias clases abstractas o interface.
- Una o más implementaciones del TDA se brindan en términos de clases concretas.
- Hoy veremos los TDAs Lista (List), Pila (Stack) y Cola (Queue), todas colecciones lineales de objetos.

# TDA Lista (List)

Una Lista es una secuencia lineal de elementos que pueden manipularse libremente, se puede agregar y eliminar elementos en cualquier posición de la misma.

## Operaciones:



- **add(e)**: agrega un elemento e en la última posición.
- **add(pos, e)**: agrega un elemento e en la posición e.
- **get(pos)**: recupera el elemento de la posición pos.
- **indexOf**: retorna el índice de la primera ocurrencia de e.
- **remove(pos)**: elimina el elemento de la posición pos.
- **remove(e)**: elimina el elemento e.
- **contains(e)**: retorna true si e está en la lista, false en caso contrario
- **size()**: Retorna un entero natural que indica cuántos elementos hay en la lista.

# TDA Pila (Stack)

Una pila es una secuencia lineal de objetos actualizada en un extremo llamado tope usando una política LIFO (last-in first-out, el primero en entrar es el último en salir).

## Operaciones:



- **push(e)**: Inserta el elemento e en el tope de la pila.
- **pop()**: Elimina el elemento del tope de la pila y lo entrega como resultado. Si se aplica a una pila vacía, produce una situación de error.
- **isEmpty()**: Retorna verdadero si la pila no contiene elementos y falso en caso contrario.
- **top()**: Retorna el elemento del tope de la pila. Si se aplica a una pila vacía, produce una situación de error.
- **size()**: Retorna un entero natural que indica cuántos elementos hay en la pila.

# TDA Cola (Queue)

Una cola es una secuencia lineal de objetos actualizada en sus extremos llamados frente y rabo siguiendo una política FIFO (first-in first-out, el primero en entrar es el primero en salir) (También se llama FCFS = First-Come First-Served).

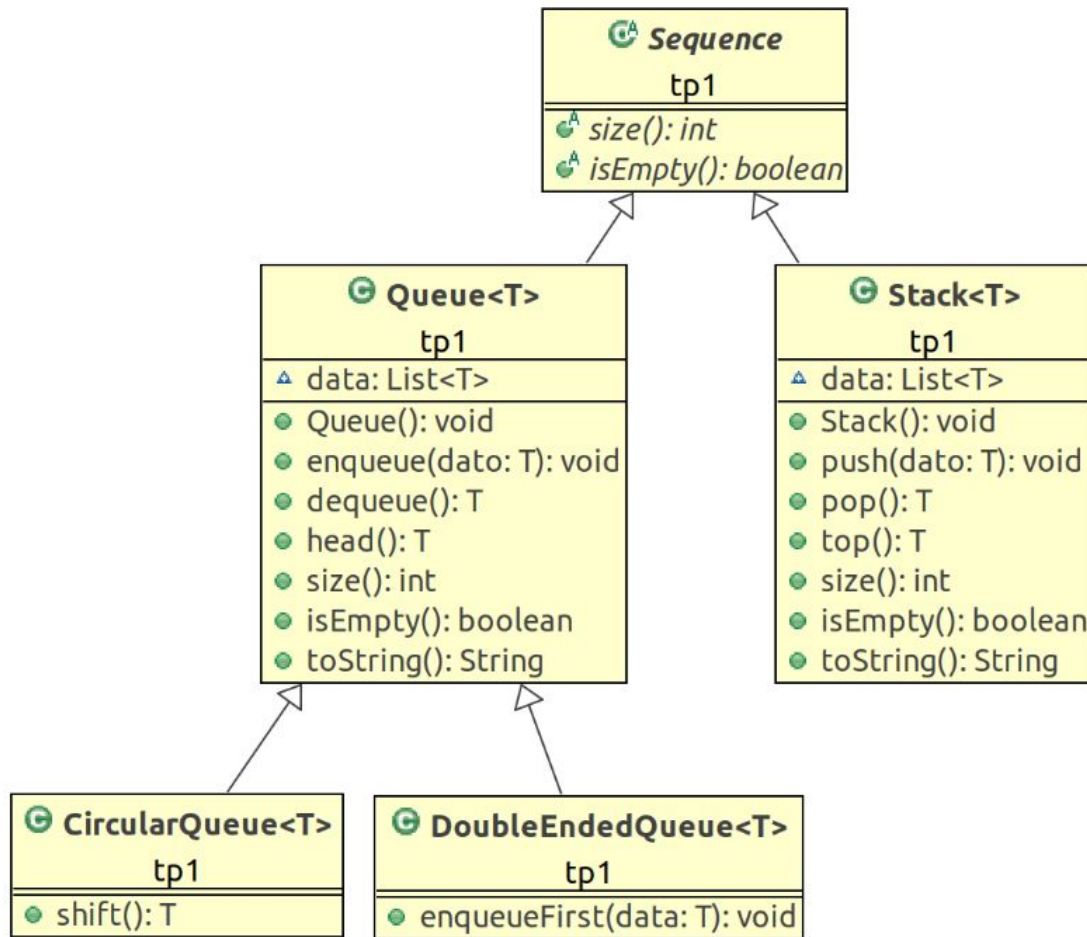
## Operaciones:



- **enqueue(e):** Inserta el elemento en el rabo de la cola
- **dequeue():** Elimina el elemento del frente de la cola y lo retorna. Si la cola está vacía se produce un error.
- **head():** Retorna el elemento del frente de la cola. Si la cola está vacía se produce un error.
- **isEmpty():** Retorna verdadero si la cola no tiene elementos y falso en caso contrario
- **size():** Retorna la cantidad de elementos de la cola.

# Listas, Pilas y Colas

Pensando en la relación de herencia y los TAD analizados podríamos pensar en las siguientes relaciones:



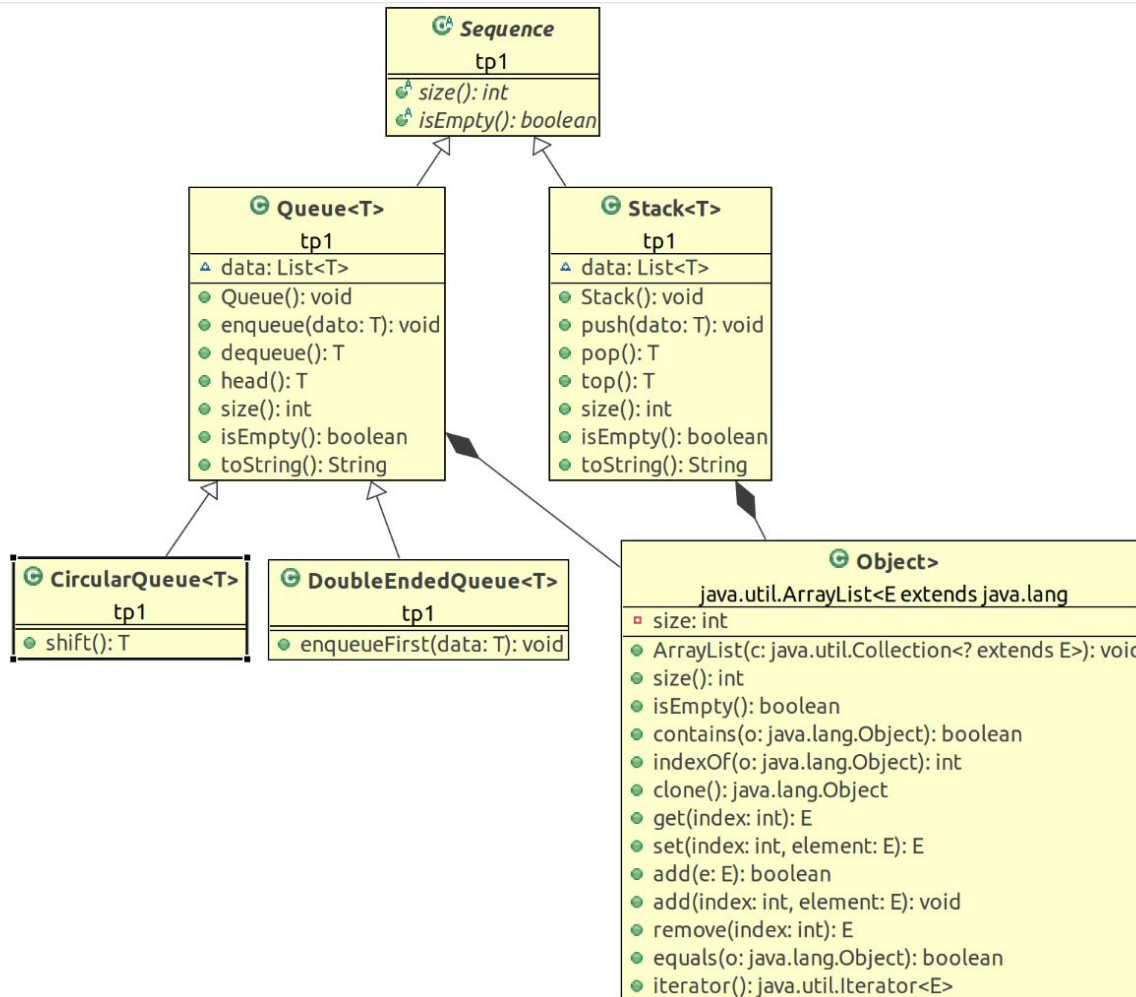
Otra especialización de la cola, haciéndola circular con un nuevo método rotar.

Se agrega esta subclase, para especializar la cola, permitiendo encolar al inicio.

- Podríamos implementar una lista? En qué lugar?
- Qué métodos incluiría?
- Podría la lista ser superclase de Queue y Stack?

# Listas, Pilas y Colas

Pensando en la relación de herencia y los TAD analizados podríamos pensar en las siguientes relaciones:



Qué es ArrayList?

Es una clase, que incluye la API de Java que implementa una lista con un arreglo de tamaño variable.

Está dentro del paquete `java.util` y pertenece al framework de colecciones.

Podríamos usar también `LinkedList`, que implementa una lista con nodos enlazados.

# Colecciones en JAVA

## Estructura

Una colección es simplemente un objeto Java que agrupa otros objetos. El framework de colecciones de Java contiene un conjunto de estructuras de datos y algoritmos para representar y manipular colecciones. Si se aplican correctamente, las estructuras de datos proporcionadas ayudan a reducir el esfuerzo de programación y aumentar el rendimiento.

El *frameworks* de colecciones de JAVA cuentan con:

- Interfaces: son tipos de datos abstractos que representan colecciones y que permiten manejarlas en forma independiente de su implementación.
- Implementaciones: son implementaciones concretas de las interfaces. Son estructuras de datos.
- Algoritmos: son métodos de clase que realizan operaciones útiles (búsquedas y ordenamientos) sobre objetos del framework.



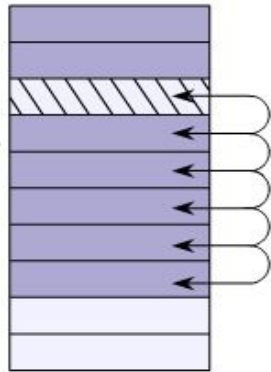
# Colecciones en JAVA

## Tecnologías de almacenamiento

Existen cuatro tecnologías de almacenamiento básicas disponibles para almacenar objetos: arreglo, lista enganchada, árbol y tabla de hash.

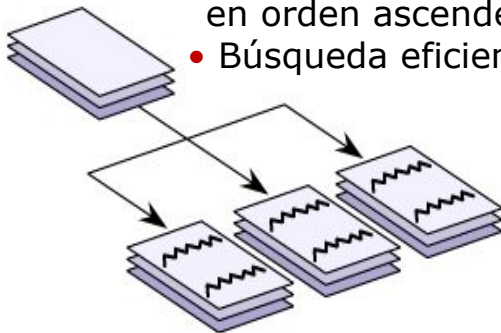
### Arreglo

El acceso es muy eficiente.  
Es ineficiente cuando se agrega/elimina un elemento.  
Los elementos se pueden ordenar.



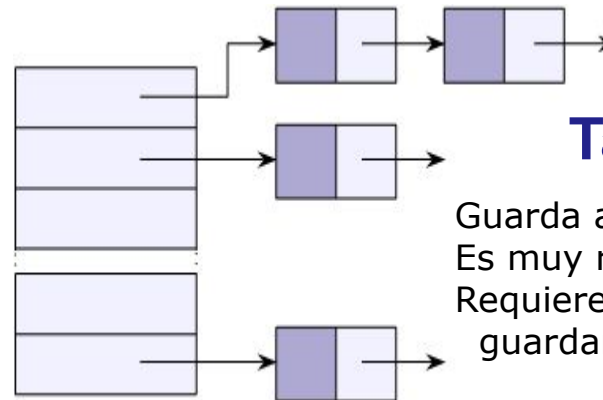
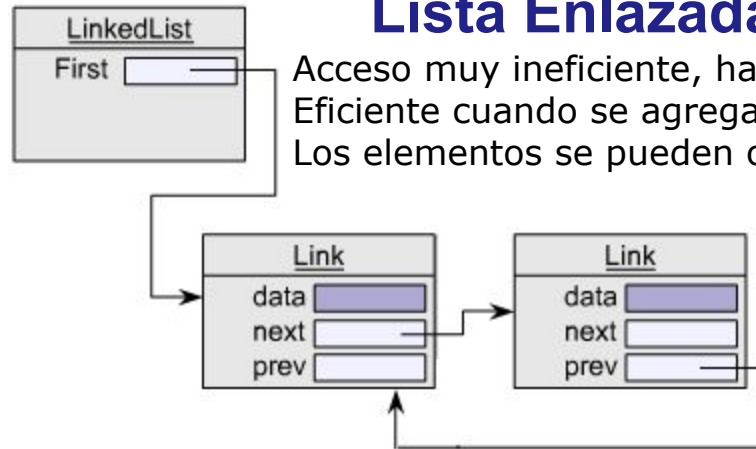
### Arbol

- Almacenamiento de valores en orden ascendente.
- Búsqueda eficiente.



### Lista Enlazada

Acceso muy ineficiente, hay que recorrer la lista.  
Eficiente cuando se agrega/elimina un elemento.  
Los elementos se pueden ordenar.



### Tabla de hash

Guarda asociaciones (clave, valor).  
Es muy rápido, accede por clave  
Requiere memoria adicional para guardar las claves (tabla).

# Colecciones en JAVA

## Implementaciones provistas por la API

La tabla muestra las implementaciones de estructuras de datos más utilizadas que vienen con la plataforma java.

TAD	Tecnologías de almacenamiento			
	Tabla de Hashing	Arreglos tamaño variable	Árbol	Lista Enlazada
Set	HashSet		TreeSet	
List		ArrayList		LinkedList
Queue		ArrayBlockingQueue		LinkedBlockingQueue
Map	HashMap		TreeMap	

En Java existe un paquete llamado `java.util` que provee varias implementaciones de listas, las principales son `ArrayList` y `LinkedList`.

Estas listas permiten almacenar datos en memoria de forma similar a los arrays pero de forma dinámica, es decir, que no es necesario declarar su tamaño, ni este último es fijo tal como pasa con los arreglos.

# Colecciones en JAVA

## Las clases LinkedList y ArrayList

El framework de colecciones tienes muchas implementaciones de estructuras de datos pero solo usaremos dos de ellas `LinkedList` y `ArrayList`:

```
package java.util;

public class LinkedList<E> implements
    List<E>. . . {

    public boolean add(E e) {...}
    public void add(int index, E element){. . .}
    public boolean remove(Object o) {...}
    public E remove(int index) {...}
    public E remove(Object) {...}
    public E get(int index) {...}
    public E set(int index, E element) {
    public int indexOf(Object o) {...}
    public boolean isEmpty(){...}
    public Iterator<E> iterator() {...}
    public int size() {...}
    . . .
}
```

Implementación de una lista con nodos enlazados

```
package java.util;

public class ArrayList<E> implements
    List<E>. . . {

    public boolean add(E e) {...}
    public void add(int index, E element){. . .}
    public boolean remove(Object o) {...}
    public E remove(int index) {...}
    public E remove(Object) {...}
    public E get(int index) {...}
    public E set(int index, E element) {
    public int indexOf(Object o) {...}
    public boolean isEmpty(){...}
    public Iterator<E> iterator() {...}
    public int size() {...}
    . . .
}
```

Implementación de una lista con arreglos

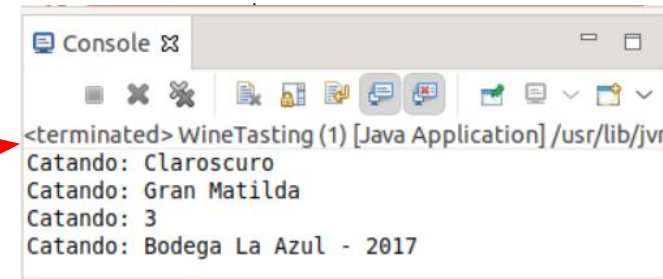
# Colecciones en JAVA

## Las clases LinkedList y ArrayList

**Veamos un ejemplo de uso de las listas de la plataforma Java.**

```
public class TestVinos {  
  
    public static void main(String[] args) {  
        List lista_vinos = new LinkedList();  
        lista_vinos.add(new Vino("Mendoza", "Claroscuro", 2022));  
        lista_vinos.add(new Vino("Mendoza", "Gran Matilda", 2018));  
        lista_vinos.add(3);  
        lista_vinos.add("Bodega Azul - 2017");  
  
        while (!lista_vinos.isEmpty()) {  
            Object vino_actual = lista_vinos.get(0);  
            System.out.println("Catando: " + vino_actual);  
            lista_vinos.remove(0);  
        }  
    }  
}
```

Se pueden guardar  
objetos de cualquier tipo



**Puedo invocar a algún méto de Vino?**

```
System.out.println("Catando: " + vino_actual.getCosecha());  
System.out.println("Catando: " + (Vino)vino_actual.getCosecha());
```

**ERROR:** Para invocar los métodos de instancia de Vino, debo castearlo. Al guardarlo en la lista se perdió el tipo.

**ERROR:** Al castearlo funciona para los objetos Vino, pero qué pasa con los últimos objetos de la lista?

# Colecciones en JAVA

## Las clases LinkedList y ArrayList

**Veamos un ejemplo de uso de una lista de tipo ArrayList.**

```
public class TestVinos {  
  
    public static void main(String[] args) {  
        List lista_vinos = new LinkedList();  
        lista_vinos.add(new Vino("Mendoza", "Claroscuro", 2022));  
        lista_vinos.add(new Vino("Mendoza", "Gran Matilda", 2018));  
        lista_vinos.add(3);  
        lista_vinos.add("Bodega Azul - 2017");  
  
        while (!lista_vinos.isEmpty()) {  
            Object vino_actual = lista_vinos.get(0);  
            System.out.println("Catando: " + vino_actual);  
            lista_vinos.remove(0);  
        }  
    }  
}
```

Al guardar los objetos en la lista de **Object** se pierde el tipo.  
Para recuperar el tipo debemos castear. Luego se puede invocar los método de cada tipo.

Vino wine = (Vino) lista\_vinos.get(0).getHarvest(); **OK**

Vino wine = (Vino) lista\_vinos.get(3).getHarvest(); **ERROR**, no se puede castear un String a Wine

Vino wine = (String) lista\_vinos.get(3).toUpperCase(); **OK**

Vino wine = (String) lista\_vinos.get(0).toUpperCase(); **ERROR**, no se puede castear un Vino a String

# Colecciones en JAVA

## Tipos Genéricos

Para resolver los problemas planteados aparecieron los Tipos de datos Genéricos

- JAVA permite definir clases y métodos genéricos.
- Las clases y métodos genéricos difieren de los "regulares" porque contienen tipos de datos como parámetros formales.
- Los tipos genéricos nos permiten generalizar la implementación de las estructuras de datos.

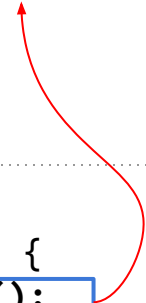
Una clase genérica significa que los elementos en esa clase se pueden generalizar con el parámetro (ejemplo T) para especificar que podemos crear instancias con cualquier tipo como parámetro en lugar de T como: Integer, Character, String o cualquier tipo.

Los genéricos también proporcionan seguridad de tipos en tiempo de compilación que permite a los programadores detectar tipos no válidos en tiempo de compilación.

# Colecciones en JAVA

## Las clases LinkedList y ArrayList

En versiones anteriores a Java 1.8 se debía especificar el tipo de dato en el `new ArrayList<Vino>`



Todas las clases del framework de colecciones son Tipos Genéricos, lo que significa que al instanciarlos podemos pasarle el "tipo" de los elementos que deseamos que contengan.

```
public class TestVinos {
    public static void main(String[] args) {
        List lista_vinos = new ArrayList();
        lista_vinos.add(new Vino("Claroscuro", 2022));
        lista_vinos.add(new Vino("Gran Matilda", 2018));
        lista_vinos.add(3);
        lista_vinos.add("Alma Negra La Azul - 2017");

        error lista_vinos.get(0).getCosecha(); //es Object
        Vivo vino=(Vino)lista_vinos.get(0).getCosecha();
        error Vino vino=(Vino)lista_vinos.get(3).getCosecha();
    }
}
```

### No especificando un tipo

- No especificando tipo, se guardan como Object.
- Object no soporta los métodos de String, ni de Vino, ni de ningún subtipo.
- Se debe saber que tipo se guarda en cada posición o utilizar el operador InstanceOf antes de castear.

```
public class TestVinos {
    public static void main(String[] args) {
        List<Vino> lista_vinos= new ArrayList();
        lista_vinos.add(new Vino("Claroscuro", 2022));
        lista_vinos.add(new Vino("Gran Matilda", 2018));
        lista_vinos.add(new Vino("Catena", 2015));
        int i=1;Vino vino_actual=null;
        while (!lista_vinos.isEmpty()) {
            vino_actual = lista_vinos.get(0);
            System.out.println("Paso "+ i++ +": "
                + vino_actual.getNombre() + "- "
                + vino_actual.getCosecha());
            lista_vinos.remove(0);
        }
    }
}
```

### Especificando Vino como tipo

- Hay chequeo de tipo en compilación: solo se permite agregar objetos de tipo **Vino** (o subclases).
- No es necesario castear para invocar sus métodos

# Colecciones en JAVA

## Implementación de Stack

```
package tp1;
import java.util.*;

public class Stack<T> extends Sequence<T> {
    private List<T> data;

    public Stack() {
        data = new ArrayList<T>();
    }

    public void push(T data) {
        data.add(0, data);
    }

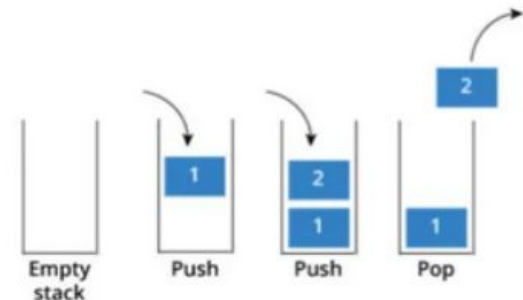
    public T pop() {
        return data.remove(0);
    }

    public T top() {
        return data.get(0);
    }
}
```

```
@Override
public int size() {
    return data.size();
}

@Override
public boolean isEmpty() {
    return data.size()==0;
}

@Override
public String toString() {
    String str = "[";
    for(T d: data)
        str = str + d + ", ";
    str = str.substring(0,str.length()-2)+"]";
    return str;
}
```





# Colecciones en JAVA

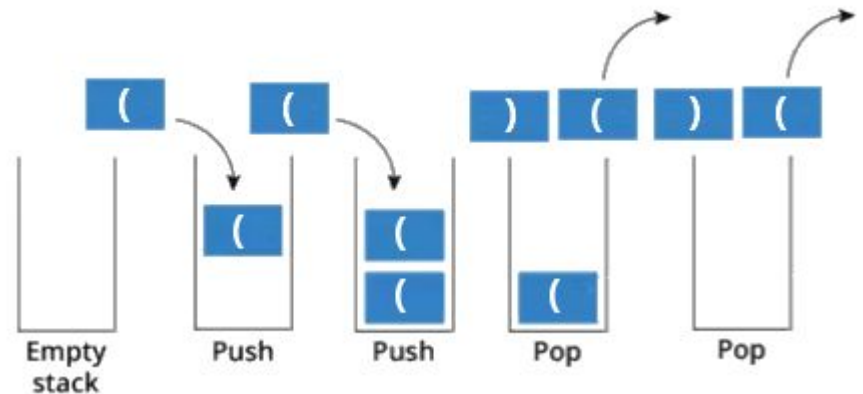
## Implementación de Stack - Ejemplo de uso

```
package tp1;

public class StackTest {

    public static void main(String[] args) {
        String exp = "(() )";
        // suponemos que la expresión sólo contiene paréntesis
        System.out.println("La expresión está balanceada? " + validar(exp));
    }

    private static boolean validar(String expresion) {
        Stack<Character> stack = new Stack<Character>();
        for (int i = 0; i < expresion.length(); i++) {
            char car = expresion.charAt(i);
            if (car == '(')
                stack.push(car);
            else if (stack.isEmpty())
                return false;
            else
                stack.pop();
        }
        return stack.isEmpty();
    }
}
```



# Colecciones en JAVA

## Implementación de Queue

```
package tp1;
import java.util.*;

public class Queue<T> extends Sequence {
    List<T> data;

    public Queue() {
        this.data = new ArrayList<T>();
    }

    public void enqueue(T dato) {
        data.add(dato);
    }

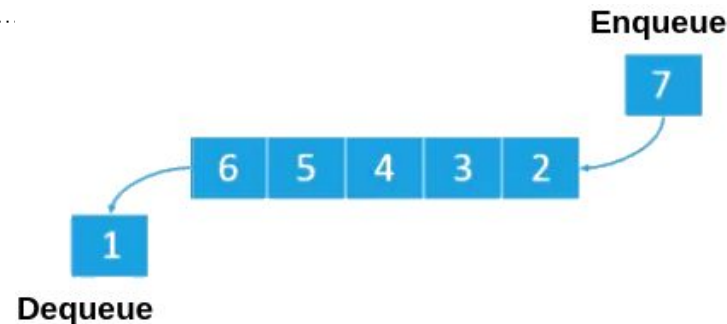
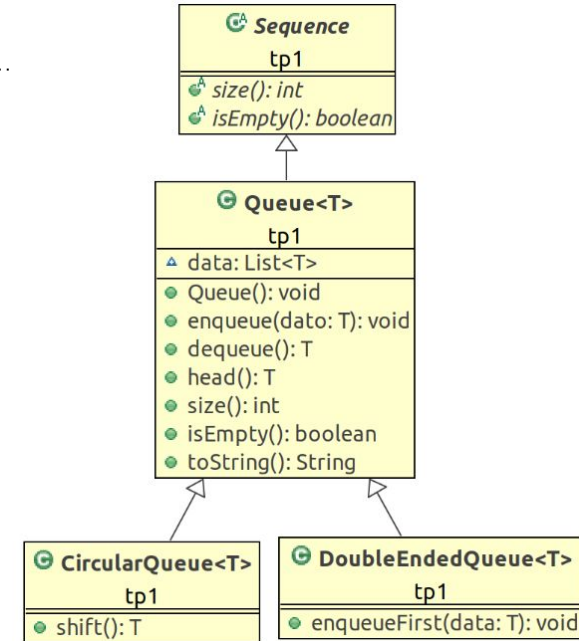
    public T dequeue() {
        return data.remove(0);
    }

    public T head() {
        return data.get(0);
    }
}
```

```
@Override
public int size() {
    return data.size();
}

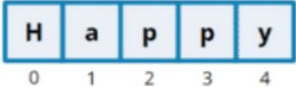
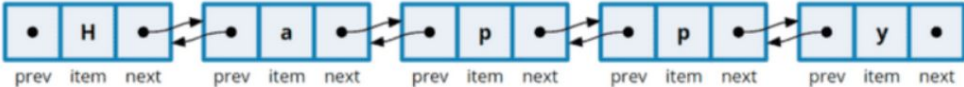
@Override
public boolean isEmpty() {
    return data.size()==0;
}

@Override
public String toString() {
    String str = "[";
    for(T d: data)
        str = str + d +", ";
    str = str.substring(0, str.length()-2)+"]";
    return str;
}
```



# Colecciones en JAVA

## LinkedList vs. ArrayList

ArrayList	LinkedList
<p>Los elementos se almacenan en un arreglo dinámico.</p> 	<p>Los elementos se almacenan en una lista doblemente enlazada.</p> 
<p>Permite el acceso aleatorio ya que los arreglos se basan en índices. Eso significa que acceder a cualquier elemento siempre lleva un tiempo constante <math>O(1)</math>.</p>	<p>Eso significa que acceder a cualquier elemento siempre lleva un tiempo lineal <math>O(n)</math>.</p>
<p>Verificar si existe un elemento específico en la lista dada se ejecuta en tiempo lineal <math>O(n)</math>.</p>	<p>Verificar si existe un elemento específico en la lista dada se ejecuta en tiempo lineal <math>O(n)</math>.</p>
<p>Agregar/Borrar elementos en/de un índice específico, en el peor de los casos, es de <math>O(n)</math>.</p>	<p>Agregar/Borrar elementos en/de en un lugar específico, en el peor de los casos, es de <math>O(n)</math>. Suele ser más rápida porque nunca se necesita cambiar el tamaño de la estructura.</p>
<p>Esta clase es más útil cuando la aplicación requiere acceso a datos y su tamaño no varía demasiado.</p>	<p>Esta clase es más útil cuando se conoce que la aplicación requiere manipulación de datos (muchas inserciones y borrados).</p>

Los principales beneficios de ArrayList y LinkedList es que LinkedList tiene un  $O(1)$  para operaciones de agregar y eliminar al principio y al final de la lista, mientras que ArrayList tiene  $O(1)$  para acceder a elementos por su índice.

Una LinkedList consume un poco más de memoria que una ArrayList ya que cada nodo almacena dos referencias al elemento anterior y al siguiente.

# Colecciones en JAVA

## Métodos para copiar colecciones

Existen varias alternativas para copiar una estructura en otra. También podemos decir clonar en el sentido de que las dos estructuras, luego de la copia, tendrán los mismos datos.

Supongamos que tenemos este objeto ArrayList con nombres de Películas:

```
ArrayList<String> peliculas = new ArrayList<String>()  
peliculas.add("Dias Perfectos");  
peliculas.add("Anatomia de una caida");  
peliculas.add("Los que se quedan");
```

1. Crear, mediante su constructor, un nuevo ArrayList pasando la lista original como argumento al constructor.

```
ArrayList<String> peliculas_copiadas1 = new ArrayList<String>(peliculas);
```

2. Crear un nuevo ArrayList y agregar todos los elementos del original usando el método addAll().

```
ArrayList<String> peliculas_copiadas2 = new ArrayList<>();  
peliculas_copiadas2.addAll(peliculas);
```

3. Clonar el ArrayList usando el método clone()

```
ArrayList<String> peliculas_clonadas = (ArrayList<String>)peliculas.clone();
```

El uso del `clone()` no se recomienda ya que se debe castear.

# Colecciones en JAVA

## Iteradores

**Iterator** es un patrón de diseño de comportamiento que permite el recorrido secuencial por una estructura de datos sin exponer sus detalles internos.

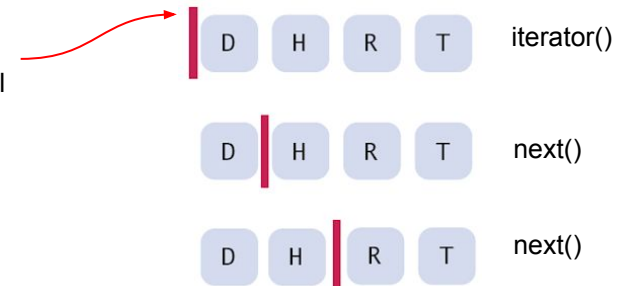
- TDA Iterable: Una colección es iterable si dispone del método `iterator()` para iterar la colección -debe implementar la interface `java.util.Iterable`-.

```
List<Integer> lista = new ArrayList<Integer>();  
Iterator<Integer> it = lista.iterator();
```

- TDA Iterador: La interface `java.util.Iterator` brinda los siguientes métodos:
  - `hasNext()`: Testea si hay elementos para recorrer en el iterador
  - `next()`: Retorna el siguiente elemento del iterador

```
while (it.hasNext()) {  
    System.out.println(it.next());  
}
```

Posición  
inicial del  
iterador



# Colecciones en JAVA

## Iteradores & foreach

```
package tp1.iteradores;
import java.util.*;

public class TestList {
    public static void main(String[] args) {
        List<Integer> lista = new ArrayList();
        lista.add(10);
        lista.add(1);
        lista.add(12);
        Iterator it2 = lista.iterator();
        while (it2.hasNext()) {
            System.out.println(((Integer)it2.next()).intValue());
        }

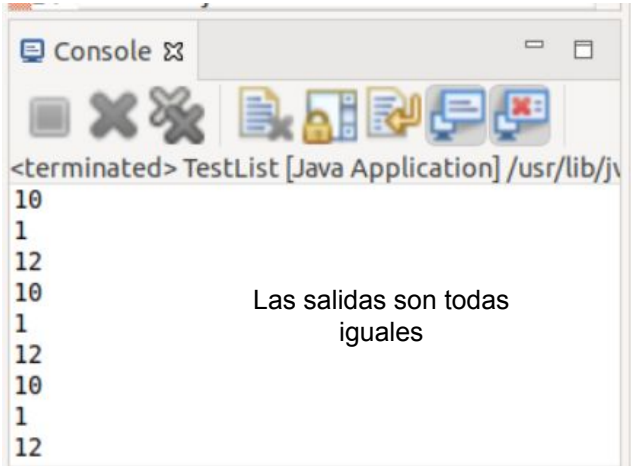
        Iterator<Integer> it1 = lista.iterator();
        while (it1.hasNext()) {
            System.out.println(it1.next().intValue());
        }

        for (Integer i: lista)
            System.out.print(i);
    }
}
```

Devuelve Object

Se puede invocar al intValue() porque el iterador devuelve Integer

El uso del Foreach es el más simple



Console

<terminated> TestList [Java Application] /usr/lib/jv

```
10
1
12
10
1
12
10
1
12
```

Las salidas son todas iguales