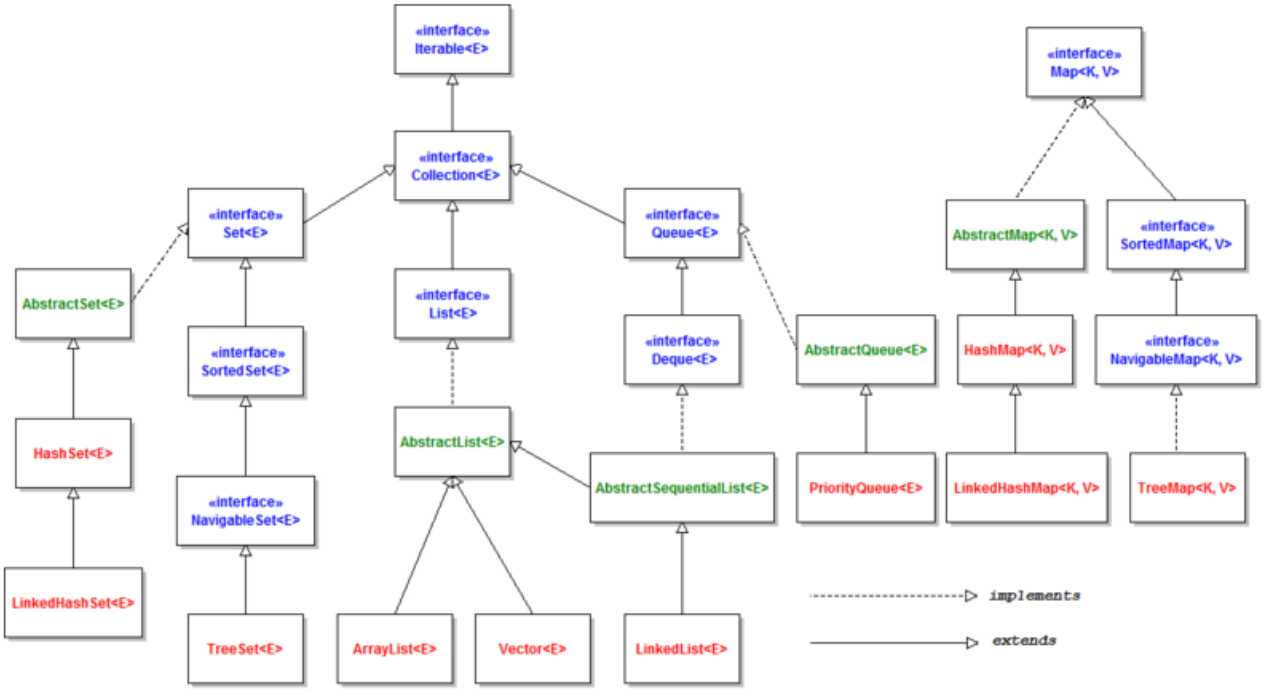
Recordemos el diagrama del Frame Collections. No es un gráfico exhaustivo pues le faltan detalles pero nos llega sobre todo para fijarnos en los grandes interfaces que viene en color azul. Observa que tanto interfaces como clases son genéricas



Class diagram of Java Collections framework

Hay muchos tipos de listas, pero muy simplificadamente distinguimos entre:

- Listas genéricas que valen para resolver todo tipo de problemas de listas
- Listas más específicas, para resolver problemas más concretos de forma más clara:
  - Pilas
  - Colas
  - otras

La clave para distinguir un tipo de lista de otro es observar las operaciones que se hacen con ellas:

- En una pila se inserta y elimina por cabeza (LIFO)
- La cola es FIFO
- Una lista usa todo tipo de operaciones, por ejemplo inserciones y eliminaciones al principio, al final y por el medio. Hay muchos problemas que deben resolverse con listas.

En este boletín veremos cosas de la jerarquía asociada al interface List que a groso modo se ocupa de las listas genéricas, de la pila y del interface Queue que se ocupa de las colas.

EL INTERFACE LIST

CUESTIONES DE ALMACENAMIENTO

Las dos implementaciones principales del interface List son LinkedList y ArrayList. Para elegir entre una u otra, debemos tener una idea aproximada de cómo se almacenan en memoria y sus consecuencias de rendimiento según el contexto de uso.

Almacenamiento de una LinkedList

Una LinkedList se basa en el uso de una lista enlazada de nodos. Si no tienes claro que es una lista enlazada de nodos revisa los ejemplos de lista enlazada basada en nodos "casera".

Almacenamiento de un ArrayList

Un ArrayList se almacena en un Array. Si no tienes claro esto, revisa los ejemplos de "ArrayList casero".

Ultra resumen:

- PARA ACCESO DIRECTO MEJOR ARRAYLIST: Para acceder a un elemento i de la lista: ArrayList permite acceso directo por índice en cambio linked list necesita internamente hacer un barrido secuencial de 0 a i-1 para llegar a i
- SI HAY MUCHOS BORRADOS E INSERCIONES MEJOR LINKEDLIST: ArrayList crea nuevos arrays subyacentes, linkedlist simplemente ajusta los enlaces de los nodos.

Ejemplo LinkedList

```
import java.util.LinkedList;

public class App{
   public static void main(String[] args){
      LinkedList<Integer> ll=new LinkedList<>(); // Declaración y creación del LinkedList de enteros.
      ll.add(1); // Añade un elemento al final de la lista.
      ll.add(3); // Añade otro elemento al final de la lista.
      System.out.println("Después de añadir 1 y 3:" +ll);
      ll.add(1,2); // Añade en la posición 1 el elemento 2.
      System.out.println("Después de añadir en la posición 1 el elemento 2:" +ll);
      ll.add(ll.get(1)+ll.get(2)); // Suma los valores contenidos en la posición 1 y 2, y lo agrega al final.
      System.out.println("Suma los valores contenidos en la posición 1 y 2, y lo agrega al final.:" +ll);
      ll.remove(0); // Elimina el primer elemento de la lista.
      System.out.println("Eliminado el primer elemento de la lista anterior:" +ll);
   }
}
```

Ejercicio U7_B4_1: Haz el ejercicio anterior con ArrayList. Y opina:

- ¿LinkedList mejor o peor que ArrayList?
- A la vista de lo fácil que es trabajar con ArrayList, ¿crees que merece la pena usar arrays tradicionales?

Ejercicio U7_B4_2: Repite el ejercicio anterior utilizando referencias de tipo List y opina ¿Mejor o peor?

Ejercicio:Consulta en el api el interface List y comprueba que evidentemente todos los métodos de List están en ArrayList y LinkedList.

INICIALIZACIÓN RÁPIDA DE ArrayList y LinkedList

Recuerda que podemos inicializar rápidamente un array con la sintaxis de {}. Por ejemplo:

int[] miArrayDeInt={1,2,3};

Y que si queremos inicializar un array en un lugar que no queremos usar variable referencia, por ejemplo en un return, escribimos

new int[]{1,2,3};

Podemos también inicializar de forma rápida una lista con Arrays.asList()

Si consultas el API observas que su cabecera es

public static <T> List<T> asList(T... a)

Es decir es un método genérico de tipo T, que devuelve una List<T>, el parámetro también incluye el tipo T lo que quiere decir que sólo admite objetos, no tipos primitivos. Recuerda también que la sintaxis de parámetros variables *T… a* implica que que puede pasar un número indeterminado de argumentos, de tipo T, de dos formas:

- Como una lista de variables de tipo T, pero ojo podría ponerse valores primitivos si el compilador puede aplicar autoboxing.
- Como un array de tipo T

Observa que la diferencia entre un parámetro T … a y otro T[] está simplemente en las posibilidades sintácticas de cómo llamar a ese método, con parámetro T[] obligatoriamente tengo que pasar un argumento de tipo T[], en cambio, con parámetro T … a puedo pasar como argumento tanto un array T[] como valores T "sueltos" separados por comas.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

class App {
    public static void main(String[] args) {
        // el tipo T de asList se infiere como Integer y se admite ints por autoboxing
        List<Integer> listaNumeros1 = Arrays.asList(1, 2, 3);
        System.out.println("listaNumeros1:" + listaNumeros1);

        // fíjate en el siguiente error
        // la indicar un array de int no se puede aplicar autoboxing
        // ya un int[] no puedo convertirse por autoboxing a Integer[]
        // List<Integer> listaNumeros2=Arrays.asList(new int[]{1,2,3});

        // ahora OK ya que se infiere T como Integer
        List<Integer> listaNumeros3 = Arrays.asList(new Integer[] { 4, 5, 6 });
        System.out.println("listaNumeros3:" + listaNumeros1);

        List<Character> listaCar = Arrays.asList(new Character[] { 'a', 'b', 'c' });
        System.out.println("listaCar:" + listaCar);

        List<String> listString = Arrays.asList(new String[] { "hola", "adios", "chao" });
        System.out.println("listaString:" + listString);
```

```
        // Si me interesa, puedo crear un arrayList o LinkedList a partir de List
        // consulta en API constructor de ArrayList/LinkedList
        ArrayList<Integer> arrayListNumeros = new ArrayList<>(listaNumeros1);
        System.out.println(arrayListNumeros);

        LinkedList<Integer> linkedListNumeros = new LinkedList<>(listaNumeros1);
        System.out.println(linkedListNumeros);

        // ERROR hijo(arrayList) no puede referenciar a padre(List)
        // ArrayList<Integer> unArrayListNumeros3 = Arrays.asList(1,2,3);

    }
}
```

¿La lista que devuelve asList() de qué clase concreta es?.

Es una variante de ArrayList que obliga a tamaño fijo, observa el siguiente ejemplo en el que descubrimos el nombre de dicha clase y que efectivamente es de tamaño fijo y da error add()

```
import java.util.Arrays;
import java.util.List;
class App {
    public static void main(String[] args) {
        //el tipo T de asList se deduce como Integer por autoboxing
        List<Integer> lista = Arrays.asList(1, 2, 3);
        System.out.println(lista.getClass().getName());
       //se pueden modificar los valores de la lista, pero no el tamaño
        lista.set(0,99);
        //lista.add(6);//salta excepción no se puede añadir ni borrar
        System.out.println("lista: " + lista);
    }
}
```

En el siguiente ejemplo vemos un resumen de posibilidades de inicialización en el que se incluye la novedad desde jdk 9 del uso del método factoría *of()* que devuelve una lista inmutable.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
class App {
    public static void main(String[] args) {
        //asList devuelve lista de tamaño fijo pero se puede modificar
        List<Integer> lista1 = Arrays.asList(1, 2, 3);
        lista1.set(0,999);
        System.out.println("lista1: "+lista1);
       //el constructor de ArrayList convierte cualquier tipo de lista en Arraylist
       //ahora puedo aumentar la lista
        List<Integer> lista2 = new ArrayList<Integer>(Arrays.asList(1,2,3));
        lista2.add(6);
        System.out.println("lista2: " + lista2);

        //con of se obtiene una lista inmutable
        //no se puede cambiar de tamaño ni de valores
        List<Integer> lista3 = List.of(1,2,3);
        lista3.set(0,99);//salta excepción
        lista3.add(7);//salta excepción
```

```
    }
}
```

Ejercicio U7_B4_3: Escribir el siguiente código con una ArrayList. Piensa en el juego 3 en Raya y en un método hayRaya() que mira si una jugada en (fila,columna) provoca completar una raya.
Este ejercicio es un tostón, cuando te canses échale un vistazo a la solución. Su objetivo es observar r lo incómodo y poco visual que resulta utilizar ArrayList para tableros.

```
public class App {

    static boolean hayRaya(char[][] tabla, int fila, int columna, char jugador) {
        //suponemos  que  tabla[fila][columna] se corresponde a una celda libre

        boolean hay = false;
        tabla[fila][columna] = jugador;
        //miramos si hay raya en fila
        if (tabla[fila][0] == tabla[fila][1] && tabla[fila][1] == tabla[fila][2] && tabla[fila][2] == jugador) {
            hay = true;
        } //miramos si raya en  columna
          else if (tabla[0][columna] == tabla[1][columna] && tabla[1][columna] == tabla[2][columna] && tabla[2][columna] ==
jugador) {
            hay = true;
        } //miramos si hay raya en diagonal, sólo dos  2 casos .
        else if (tabla[0][0] == tabla[1][1] && tabla[1][1] == tabla[2][2] && tabla[2][2] == jugador) {
            hay = true;
        } else if (tabla[0][2] == tabla[1][1] && tabla[1][1] == tabla[2][0] && tabla[2][0] == jugador) {
            hay = true;
        } else {
            hay = false;
        }
        //volvemos a dejar como libre la celda, este método investiga pero no cambia nada
        tabla[fila][columna] = '_';

        return hay;
    }

    public static void main(String[] args) {

        char[][] tabla = {
            {'X', '_', 'X'},
            {'O', '_', 'O'},
            {'X', '_', '_'}
        };
        //suponemos que la  tabla a comprobar no tiene ya raya
        System.out.println(hayRaya(tabla, 0, 1, 'X') ? "SI" : "NO");
        System.out.println(hayRaya(tabla, 0, 1, 'O') ? "SI" : "NO");



    }
}
```

Ejercicio U7_B4_4: Teclado estropeado id 144 (acepta el reto)
Resuélvelo con lista y piensa qué incomodidades ocurrirían si lo resolvemos con array. Por categorías se encuentra en la categoría de listas que sorprendentemente sólo tiene dos problemas.

OJO CON LA RELACIÓN REFERENCIA/OBJETO AL TRABAJAR CON COLECCIONES

Cuando se trabaja con colecciones podemos llevarnos sorpresas con la famosa relación referencia/objeto a la que le dimos tantas y tantas vueltas.

Para entender el Ejemplo1 y el Ejemplo2 "dibuja" la creación y modificación de objetos

Ejemplo1:

```
import java.util.LinkedList;
public class App {


    public static void main(String[] args) {
        LinkedList<Integer> l= new LinkedList<>();
        Integer i= new Integer(3);
        l.add(i);
        i++;
        System.out.println("el valor Integer que consta en la lista: "+ l.get(0));
        System.out.println("el valor de la referencia Integer i es: "+ i);
    }
}
```

En el código anterior, puntos clave:

- Integer i= new Integer(3);

La referencia de este objeto se almacena en la variable "i".
A continuación también se pasa esta referencia a un elemento de la lista con

- l.add(i);

Ejecutada esta instrucción el objeto Integer creado anteriormente está referenciado por dos referencias, la variable i y por un elemento de la lista.

- i++;

Autoboxing, provoca que se cree un nuevo objeto Integer y la referencia i se desliga del objeto al que apunta la referencia Integer de la lista

En el ejemplo anterior vimos que tras hacer i++, el Integer de la lista no cambió de valor y debemos tener claro por qué.

Si dudas de la explicación anterior, el siguiente código que imprime los valores de referencias puede ayudarte

```
import java.util.LinkedList;
public class App {

    public static void main(String[] args) {
        LinkedList<Integer> l= new LinkedList<>();
        Integer i= 3;
        System.out.println(System.identityHashCode(i));
        l.add(i);
        //hacer l.add(i) es similar a j=i, es decir, copiar el valor de una variable en otra
        Integer j;
        j=i;//no hay unboxing ya que no es int/Integer es Integer/Integer. Simplemente el valor de i se copia en j
        System.out.println(System.identityHashCode(j));
        j=i;
        System.out.println(System.identityHashCode(l.get(0)));
```

```
        i++;
        System.out.println("el valor Integer que consta en la lista: "+ l.get(0));
        System.out.println("el valor de la referencia Integer i es: "+ i);
        System.out.println(System.identityHashCode(i));
        System.out.println(System.identityHashCode(l.get(0)));
    }
}
```

Ejemplo2:

Ahora hay un objeto Envolvente que encapsula uno Integer.

```
import java.util.LinkedList;

class Envolvente{
    public Integer num;
    Envolvente (int num) { this.num=num;}//autoboxing
}
public class App {
    public static void main(String[] args) {

        LinkedList<Envolvente> lista=new LinkedList<>();
        Envolvente entero1=new Envolvente(3);
        lista.add(entero1);
        entero1.num++;
        System.out.println("el valor entero1.num es: "+ entero1.num);
        System.out.println("el valor Integer que consta en la lista: "+ lista.get(0).num);
    }
}
```

Ahora si cambiamos el entero encapsulado desde la referencia externa a la lista también modificamos la lista

MÁS SINTAXIS DE GENÉRICOS: Wildcards

Míra esto por encima y vuelvelo a mirar si ves que te hace falta más adelante. Después de echarla un vistazo rápido avanza hasta pilas y colas en este documento.

Esta nueva sintaxis no hace falta dominarla al 100%, se puede manejar intuitivamente en muchas situaciones, pero le echaremos un vistazo para que no nos resulte muy extraña ya que la observamos en algunos parámetros de métodos de utilidad del framework collections.

El comodín(wildcard)? para indicar cualquier tipo.

El símbolo *?* en la sintaxis de genéricos representa no a un tipo, sino a un conjunto de tipos. Normalmente suele verse este comodín al trabajar con colecciones. Usamos ejemplo de https://docs.oracle.com/javase/tutorial/java/generics/unboundedWildcards.html

Observa el error en el siguiente ejemplo:

```
import java.util.Arrays;
import java.util.List;
public class App {
    public static void printList(List<Object> list) {
    for (Object elem : list)
        System.out.println(elem + " ");
        System.out.println();
    }

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

```
    List<Object> l1=Arrays.asList(1,2,3);
    printList(l1);

    List<Integer> l2=Arrays.asList(1,2,3);
    printList(l2);
  }
}
```

Una variable Object puede referenciar a un Integer ya que Object es superclase, pero, una var List<object> no puede referenciar a un objeto List<Integer> ya que no es cierto que List<Integer> extends List<Object>

La solución es indicar tipo ? en lugar de Object

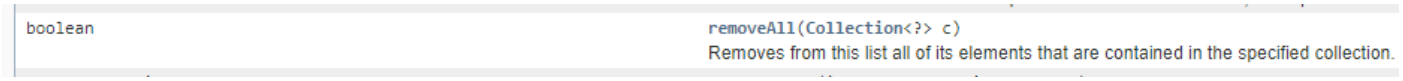
```
import java.util.Arrays;
import java.util.List;
public class App {
  public static void printList(List<?> list) {
  for (Object elem : list)
     System.out.println(elem + " ");
     System.out.println();
  }

  public static void main(String args[]) {
    List<Object> l1=Arrays.asList(1,2,3);
    printList(l1);

    List<Integer> l2=Arrays.asList(1,2,3);
    printList(l2);

    List<String> l3=Arrays.asList("one","two","Three");
    printList(l3);
  }
}
```

En muchos métodos del api se observa este recurso por ejemplo observa el método removeAll de ArrayList



Ahora resolvemos el problema anterior declarando el método como genérico, sin usar wildcard

```
import java.util.Arrays;
import java.util.List;
public class App {
  public static <T> void printList(List<T> list) {
  for (T elem : list)
     System.out.println(elem + " ");
     System.out.println();
  }

  public static void main(String args[]) {
    List<Object> l1=Arrays.asList(1,2,3);
    printList(l1);

    List<Integer> l2=Arrays.asList(1,2,3);
    printList(l2);

    List<String> l3=Arrays.asList("one","two","Three");
    printList(l3);
  }
}
```

Bounded wildcard vs Type parameter (comodín vs letra genérica)

¿Qué solución es mejor?

Si realmente no es necesario el manejo de una letra o parámetro genérico(Type parameter) , en Oracle consideran mejor estilo usar la versión con wildcard(bounded wildcard). Supongo que la consideran mejor porque es más sencilla.
En el ejemplo anterior el método printList necesita manejar sólo el toString() de un objeto lo que le permite trabajar con Object y por polimorfismo es innecesario trabajar con un tipo concreto que es lo que ocurriría si usamos la versión con letra genérica T ya que dicha T se sustituirá por un tipo concreto en algún momento de la ejecución.

El comodín *? extends base* y genéricos

Ya vimos el uso de *T extends base* en el boletín de 4. Genéricos. Significaba cualquier tipo que extienda al tipo base.
? extends base también significa "cualquier tipo que extienda a base" pero se usa en sitios diferentes. *T extends base* se usa en la declaración de clases y métodos genéricos, y *? extends base* se usa en la declaración de variables y parámetros normalmente de tipo colección (variables list, map, etc…)

Veamos un ejemplo primero que nos conduce a la necesidad de la nueva sintaxis

```
import java.util.ArrayList;
public class Unidad5 {
    public static void main(String args[]) {
        Number n;
        n=new Integer(2);
        n=new Double(2.2);
        ArrayList<Number> aln1=new ArrayList<Integer>();
        ArrayList<Number> aln2=new ArrayList<Double>();
        ArrayList<? extends Number> aln3=new ArrayList<Integer>();
        ArrayList<? extends Number> aln4=new ArrayList<Double>();
    }
}
```

Sabemos que Number es una clase abstracta y que tiene como subclases en el JDK las siguientes subclases:

Direct Known Subclasses:

AtomicInteger, AtomicLong, BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, Short

---

Por lo tanto, es correcto:
n=new Integer();
n=new Double();

Ya que number es superclase de Integer y de Double() y hay compatibilidad de tipos

Pero observa que en cambio genera error
ArrayList<Number> aln1=new ArrayList<Integer> ya que ArrayList<Number> no es en absoluto superclase de ArrayList<Integer> y tampoco son clases del mismo tipo

Ahora con ? extends base
Fíjate en el siguiente ejemplo, que nos permite sumar elementos de una lista de diferentes tipos de objetos siempre y cuando pertenezcan a clases que extienden a Number. Nos aprovechamos que todos los subtipos de Number tienen un método doubleValue()

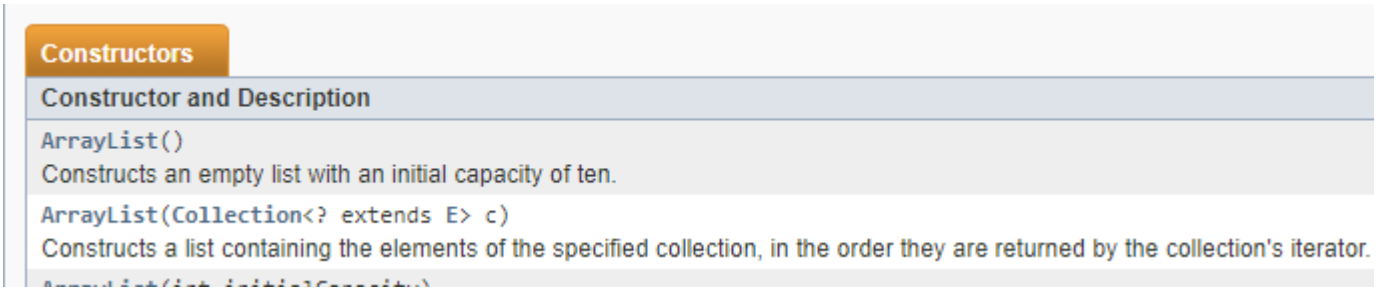
```
import java.util.Arrays;
import java.util.List;
public class App {
    public static double sumOfList(List<? extends Number> list) {
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
    public static void main(String args[]) {
      List<Integer> li = Arrays.asList(1, 2, 3);
      System.out.println("sum = " + sumOfList(li));

      List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
      System.out.println("sum = " + sumOfList(ld));

      List<BigInteger> lbi = Arrays.asList(new BigInteger("111"),new BigInteger("222"));
      System.out.println("sum = " + sumOfList(lbi));
    }
}
```

El ejemplo anterior está extraído de
https://docs.oracle.com/javase/tutorial/java/generics/upperBounded.html
Al que se le ha añadido una lista de BigInteger() para observar mejor su generalidad

En el api también observamos esta sintaxis por ejemplo en el constructor de ArrayList



Constructors

Constructor and Description
`ArrayList()` Constructs an empty list with an initial capacity of ten.
`ArrayList(Collection<? extends E> c)` Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

MANEJAR PILAS Y COLAS

En boletines anteriores de "estructuras dinámicas" hicimos implementaciones "a pelo" con arrays y nodos enlazados. Evidentemente, ya existe código escrito al respecto en el JDK.

Usar LinkedList para funcionar como una pila

linkedList es una lista genérica pero usando los métodos apropiados podemos trabajar como si fuera una pila o una cola .

Ejercicio U7_B4_5: En el boletín de estructuras dinámicas escribiste una clase Pila "a pelo" usando nodos enlazados. Escribe ahora una clase Pila pero utilizando subyacentemente LinKedList

```
import java.util.LinkedList;
class Pila<T>{
   LinkedList<T> listaDatos= new LinkedList<>();
   //escribir push(), pop() y esVacia()
}
public class App {

   public static void main(String args[]) {
      Pila<Character> miPila= new Pila<>();
```

```
        miPila.push('a');miPila.push('b');miPila.push('c');
        while(!miPila.esVacia()){
            System.out.print(miPila.pop()+" ");
        }
    }
}
```

LA CLASE STACK

Ejercicio U7_B4_6: Ahora en lugar de escribir una clase Pila repetimos el ejercicio anterior usando directamente la clase del JDK Stack.

¿Ventaja de Stack sobre linkedlist?. Si consultas el API observas que es una clase más concreta que consta de sólo 5 operaciones(métodos) que nos obligan a respetar la filosofía de Pila, es decir, genera una solución más legible y elegante. Observa por ejemplo que la clase Stack no me permite insertar un elemento por el medio de la pila. Si necesitara insertar por el medio de la pila, entonces no necesito una pila si no una lista genérica como la que implementa ArrayList/LinkedList

Ejercicio U7_B4_7: Acepta el reto 141.

Recuerda el ejercicio de paréntesis balanceados que hicimos con la pila casera. Se pide una solución en la que también **incluya OBLIGATORIAMENTE un método paréntesis Balanceados** similar al que hicimos pero que tendras que retocar para que incluya los nuevos símbolos de este enunciado. También es obligatorio para nosotros utilizar la clase Stack.

EL INTERFACE QUEUE

Las colas tienen una serie de complejidades y variantes que no tienen las pilas y por eso tienen un interface específico.
¿Cual es la diferencia entre `Queue<Integer> cola=new LinkedList<>()` y `LinkedList<Integer> cola=new LinkedList<>();` ?
Como explicamos entre Stack vs LinkedList, el interface Queue es más simple y concreto. Si el problema se debe resolver con una cola, usar este interface me obliga a no desviarme de la filosofía FIFO ya que tiene sólo 6 métodos y realmente son 3 repetidos con la variante de que o devuelven un valor especial o generan una excepción. Si consultas el API

Summary of Queue methods

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Observa en el API de Queue que en el apartado All Known Implementing Classes, que ArrayList no está en esta lista. Seguramente porque para la operativa habitual de una cola, LinkedList es más eficiente . Por lo tanto esto da error en tiempo de compilación

```
Queue<Integer> cola=new ArrayList<>();
```

Observa también que para trabajar con una cola no es apropiado trabajar con el interface List ya que no tiene un método específico para borrar por el principio de la cola.

Ejercicio U7_B4_8: se escribió una cola "casera" *MiCola* para resolver el siguiente main().

```
class App{

    public static void main(String[] args) {
        MiCola<Integer> mc1 = new MiCola<>();
        mc1.encolar(1);
```

```
        mc1.encolar(2);
        mc1.encolar(3);
        while (!mc1.esVacia()) {
            System.out.print(mc1.desencolar()+ " ");
        }
    }
}
```

SE PIDE: Escribir el main con LinkedList sin escribir ninguna clase(resolver directamente en main)

Ejercicio U7_B4_9: Lo mismo que en el ejercicio anterior utilizando el interface Queue

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

Ejercicio U7_B4_10: hacer el recorrido en árbol en anchura que hicimos con cola casera Ahora con un objeto ***Queue***

Ejercicio U7_B4_11: Acepta el reto, al mundial con transatlántico 473
Obligatorio con interface Queue

El siguiente pseudocódigo te puede inspirar, pero hazlo como quieras
PSEUDOCÓDIGO(para 1 caso)

Mientras la cola tiene un tamaño mayor que 1
Hacer tanda tiros hasta perder el último balón

aquí se llega con un sólo balón en la cola y se imprime el número del balón

Y podemos especificar un poco más que es hacer una tanda de tiros en función de una cola
Hacer tanda tiros hasta perder el último balón =>quitar un balón de principio de la cola y volver a meterlo al final de la cola menos el último balón de la tanda que se saca pero no se mete ya que se va al mar

RECORRER SECUENCIALMENTE LAS COLECCIONES VISTAS
Tres formas:

1. con los métodos propios de la clase concreta
2. con for-each
3. con un objeto iterador. Lo veremos más adelante

El recorrido de principio a fin de una colección es una operación muy habitual y merece unas observaciones.

Recorrer una lista

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
public class App {
    public static void main(String args[]) {

        List<Integer> l = new ArrayList(Arrays.asList(1, 2, 3, 4, 5, 6, 7,8,9));
        for (int i = 0; i < l.size(); i++) {
            System.out.print(l.get(i) + " ");
        }
        System.out.println("");
        for(int i:l){
            System.out.print(i+ " ");
        }
```

```
    }
}
```

Pero, si al mismo tiempo que recorremos *añadimos o borramos,* estamos modificando el tamaño de la lista y por tanto el valor de size(). En el siguiente código queremos borrar el elemento 3 y 4 pero no es así:

Observa en el ejemplo de abajo como inesperadamente no borra el valor 14 y borra el 15.

Cuando get(i)==13 i vale 3. Pero justo despues de remove(i), ahora el valor 14 se corresponde con i==3 con lo que remove(i+1) borra el valor 15

El for mejorado también también produce efectos inesperados

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class App {
    public static void main(String args[]) {
        List<Integer> l = new ArrayList(Arrays.asList(11, 12, 13, 14, 15, 16, 17,18,19));
        for (int i = 0; i < l.size(); i++) {
            if (l.get(i) == 13) {
                l.remove(i);l.remove(i+1);
            }
            System.out.print(l.get(i) + " ");
        }

    }

}
```

También hay problemas la añadir elementos como en el ejemplo siguiente y en muchos otros

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;


public class App {
    public static void main(String args[]) {
        List<Integer> l = new ArrayList(Arrays.asList(11, 12, 13, 14, 15, 16, 17,18,19));
        for (int i = 0; i < l.size(); i++) {
            if (l.get(i) == 13) {
                l.add(l.get(i));
            }
            System.out.print(l.get(i) + " ");
        }
     }
}
```

SOLUCIÓN: Utilizar iteradores (o meter código adicional que corrija el efecto). Los iteradores se verán en otro boletín

RECORRER PILA: NO ES UNA OPERACIÓN TÍPICA

Una pila se accede por su cabeza, no se recorre, por eso no hay métodos con índices en la implementación de una pila.

Lo más parecido a recorrerla sería "vaciarla" de forma que voy quitando todos los elementos por la cabeza hasta vaciarla. Por lo tanto existe el método size() pero es heredado no propio y se prefiere utilizar isEmpty()

Para simular una inicialización rápida usamos el método addAll(), pero realmente este método no es propio de la Pila si no heredado.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Stack;

public class App {
    public static void main(String args[]) {
        List<Integer> l = new ArrayList(Arrays.asList(11, 12, 13, 14, 15, 16, 17,18,19));
        Stack<Integer> p = new Stack<>();
        p.addAll(l);
        while(!p.isEmpty()){
            System.out.print(p.pop()+" ");
        }

    }

}
```

Ejemplo de problemas al borrar mientras se recorre

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Stack;

public class App {
    public static void main(String args[]) {
        List<Integer> l = new ArrayList(Arrays.asList(11, 12, 13, 14, 15, 16, 17,18,19));
        Stack<Integer> p = new Stack<>();
        p.addAll(l);
        //forma clasica y apropiada de trabajar con pila
        while(!p.isEmpty()){
            System.out.print(p.pop()+" ");
        }
        System.out.println("");
        p = new Stack<>();
        p.addAll(l);
```

```
        //trabajar con p.size() da problema ya que pop() varía el size
        for(int i=0;i<p.size();i++){
            System.out.print(p.pop()+" ");
        }

    }
}
```

RECORRER UNA COLA: NO ES UNA OPERACIÓN TÍPICA

Similar a una pila, lo que queremos de una cola es introducir por el final y sacar por el principio.

Si nos empeñamos en recorrer secuencialmente una cola y usamos poll(), podemos valernos de que poll() devuelve null si la cola está vacía

Observa de nuevo que cuando usamos size() en el for al borrar con poll(), hay problemas

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

public class App {
    public static void main(String args[]) {
        List<Integer> l = new ArrayList<>(Arrays.asList(11, 12, 13, 14, 15, 16, 17,18,19));
        Queue<Integer> c = new LinkedList<>();
        c.addAll(l);
       //forma clásica de vaciar una cola
        while(!c.isEmpty()){
            System.out.print(c.poll()+" ");
        }
        System.out.println("");
        c = new LinkedList<>();
        c.addAll(l);
       //otra forma  de vaciar una cola más fea
        Integer i= c.poll();
        while(i!=null){
            System.out.print(i+" ");
            i= c.poll();
        }

        System.out.println("");
        c = new LinkedList<>();
        c.addAll(l);
       // forma  de vaciar incorrecta
        for(int j=0;j<c.size();j++){
            int x=c.poll();
            System.out.print(x+" ");
```

```
    }

  }
}
```