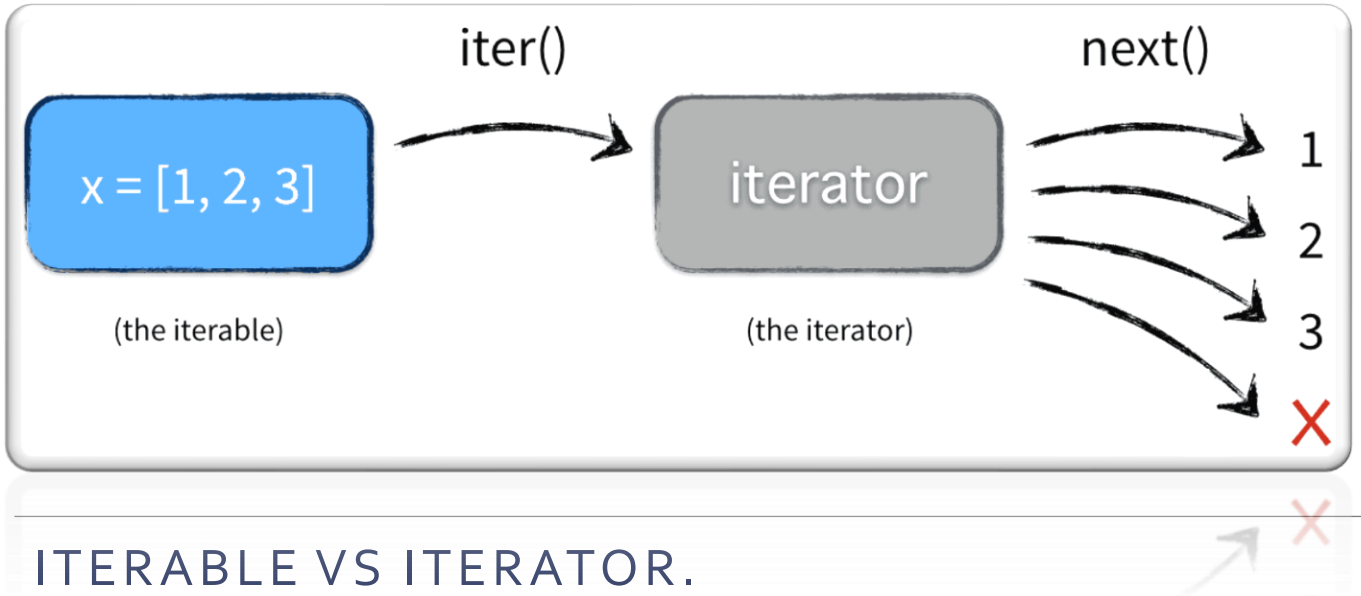
[Estructuras de Datos]



ITERABLE VS ITERATOR.

TDA ITERABLE E IMPLEMENTACIÓN DE ITERADORES.

ITERABLE DE POSICIONES EN POSITIONLIST.

Recursos adicionales

Enlaces externos de interés

- Acceda a información útil mediante enlaces externos a:
  - **código fuente** disponible de forma **online**.
  - **otro material** disponible de forma **online**.

Copyright

ITERABLE VS. ITERATOR

Introducción: ¿qué es un recorrido?

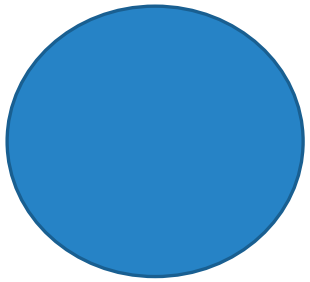
- Los conceptos **iterable** e **iterador** surgen a partir de un concepto más general que es **recorrido**.
- ¿Qué es un **recorrido** en una ED?
  - Es la **visita** de **todas** sus **celdas**, en **algún orden** determinado, de tal forma que cada celda sea **visitada una única vez**.
  - Una ED **puede** ser **recorrida** de **diferentes** maneras: por ejemplo, una lista puede recorrerse en forma creciente o decreciente.

Iterable vs. Iterador

- De lo mencionado al conceptualizar **recorrido**, surge la diferencia.
- Una **ED** que **puede recorrerse**, se dice que es **iterable.**
- El **orden** en la que puede **recorrerse** una **ED iterable**, queda definido a través de un **iterador**.

Iterable → Adjetivo → *Palabra que acompaña al sustantivo para* **expresar una cualidad.**

Iterador[1] → Sustantivo[1] → *Que tiene* **existencia real** *e independiente.*



Este ***círculo*** *es* ***iterable****, lo que implica* ***que permite ser recorrido****.*
Este ***círculo*** *es* ***iterable****, y se lo puede recorrer* ***mediante*** *dos* ***iteradores*** *diferentes:*

1) *Uno que lo recorre en sentido horario.*
2) *Otro que lo recorre en sentido anti-horario.*

Dato de color sobre 1:

Iterador *no es una palabra reconocida por la RAE, sino es una* ***traducción informal*** *y* ***literal*** *que utilizamos en programación para el término inglés* ***Iterator****.*

TDA ITERABLE E IMPLEMENTACIÓN DE ITERADORES

Colecciones Iterables - TDA Iterable

- Una **colección de objetos** se dirá **Iterable**, si el TDA que lo representa **provee métodos** para tal fin.
- En particular, se puede definir un **tipo de dato Iterable** que indique qué métodos lo define.

Interface Iterable<E>
+ iterator(): Iterator<E>

```
public interface Iterable<E> {
  public Iterator<E> iterator();
}
```

En la materia utilizaremos el **TDA Iterable<E>** definido por **Java**, de forma tal de poder utilizar ciertas herramientas tales como la instrucción **for-each**. ¿Recuerda esta consigna del TP Nº 1?

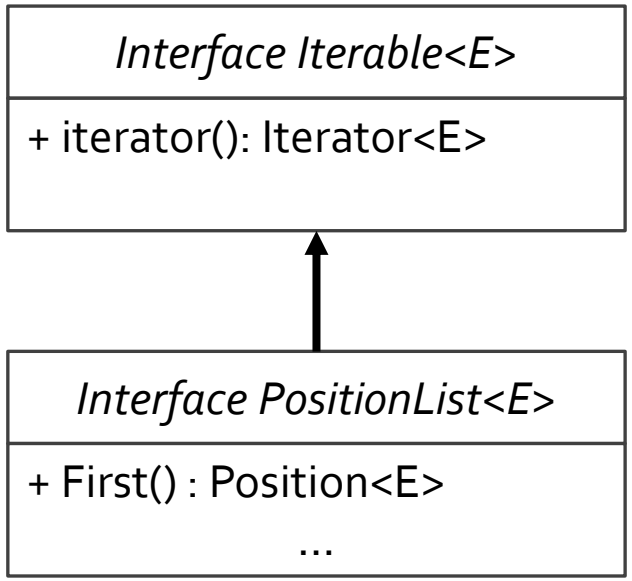
Indique qué hace la siguiente porción de código:

```
int[] arr = {1, 2, 3};
for (int i : arr)
        System.out.println(i);
```

Explique brevemente el concepto de iterador en Java.

TDAs Iterables

- Luego, un TDA será **iterable** si implementa los métodos definidos en la interfaz **Iterable<E>**.
- El TDA **PositionList** define su interfaz de la siguiente manera:



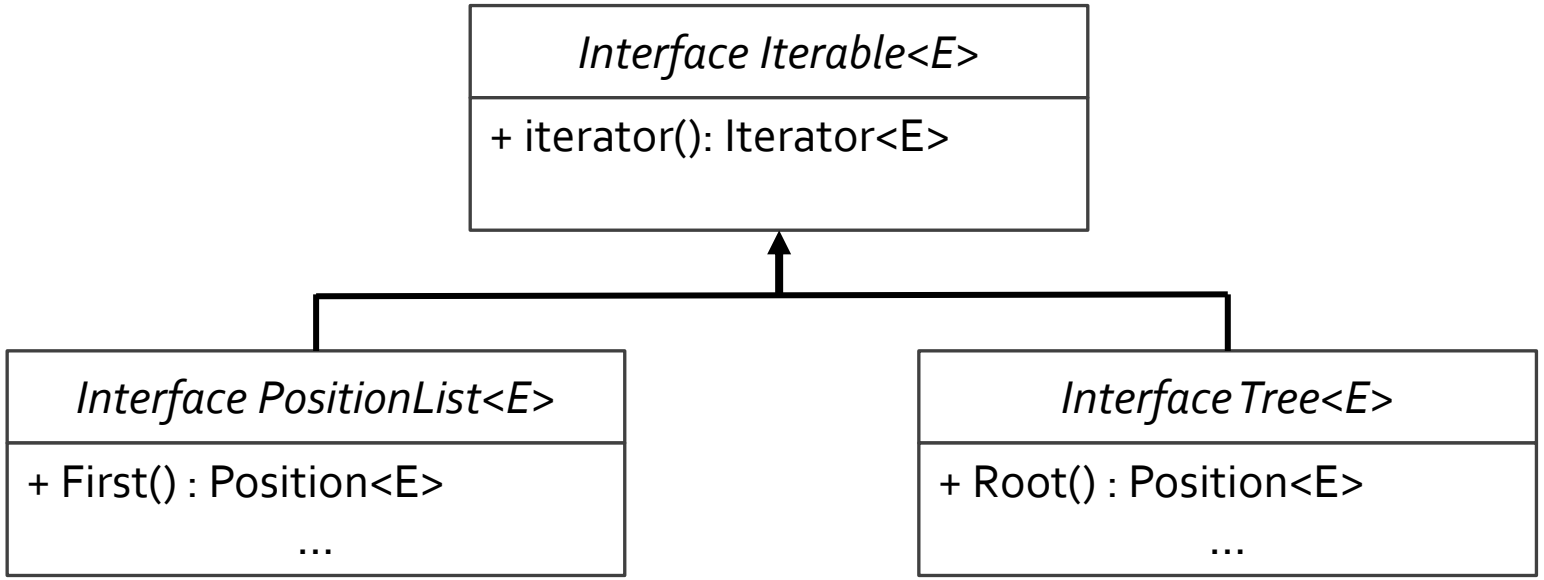

```
public interface Iterable<E> {
  public Iterator<E> iterator();
}
```

```
public interface PositionList<E> extends Iterable<E> {
  public Position<E> first() throws EmptyListException;
  …
}
```

Cualquier clase que implemente **PositionList<E>**, será también **Iterable<E>**, por lo que **estará obligada** a implementar el método **iterator().**

TDAs Iterables

- Luego, un TDA será **iterable** si implementa los métodos definidos en la interfaz **Iterable<E>**.
- La noción de **iterable** la usaremos tanto para el **TDA Lista**, como para otros TDAs, como por ejemplo el **TDA Árbol**.



TDA Iterador

- Una **ED** que **puede recorrerse**, se dice que es **iterable.** El **orden** en la que puede **recorrerse** una **ED iterable**, queda definido a través de un **iterador**.
- ¿Qué es un **iterador**?
  - Es un **patrón de diseño** que **abstrae** el **proceso** de **recorrer** una ED.
  - Se puede abstraer considerando que consiste de:
    - Una **secuencia S**.
    - Un elemento **actual**, denominado **cursor**.
    - Un mecanismo para **pasar** a un **próximo elemento**, haciéndolo a este, el **nuevo actual**.
  - Encapsula los conceptos de **próximo**, **lugar** y **posición** de la **ED iterable**.

TDA Iterador

- **Patrón de diseño** que **abstrae** el **proceso** de **recorrer** una ED.
- Se puede abstraer considerando que consiste de:
  - Una **secuencia** **S**.
  - Un elemento **actual**, denominado **cursor**.
  - Un mecanismo para **pasar** a un **próximo elemento**, haciéndolo a este, el **nuevo actual**.

Interface Iterator<E>
+ hasNext(): boolean + next() : E

```
public interface Iterator<E> {
  public boolean hasNext();
  public E next();
}
```

En la materia utilizaremos el **TDA Iterator<E>** definido por **Java**, de forma tal de poder utilizar ciertas herramientas tales como la instrucción **for-each**.

TDA Iterador

- **Patrón de diseño** que **abstrae** el **proceso** de **recorrer** una ED.
- Se puede abstraer considerando que consiste de:
  - Una **secuencia** **S**.
  - Un elemento **actual**, denominado **cursor**.
  - Un mecanismo para **pasar** a un **próximo elemento**, haciéndolo a este, el **nuevo actual**.

Interface Iterator<E>
+ hasNext(): boolean + next() : E

```
public interface Iterator<E> {
  public boolean hasNext();
  public E next();
}
```

- La **forma** en la que se **recorren** los elementos de la colección, **queda encapsulada** en la operación ***next().***

TDA Iterador :: Implementación

- **Patrón de diseño** que **abstrae** el **proceso** de **recorrer** una ED.
- Se puede abstraer considerando que consiste de:
  - Una **secuencia** **S**.
  - Un elemento **actual**, denominado **cursor**.
  - Un mecanismo para **pasar** a un **próximo elemento**, haciéndolo a este, el **nuevo actual**.
- Existen dos alternativas para implementar la operación de un iterador:
  - Operando **directamente** sobre la **colección a iterar**.
  - Operando **sobre** una **ED auxiliar** que **clona** la **colección a iterar**.

TDA Iterador :: Implementación

- Operando **directamente** sobre la **colección a iterar**.

```
public class IteradorListaSobreEdOriginal<E> implements Iterator<E>{
  private PositionList<E> lista;
  private Position<E> cursor;

  public IteradorListaSobreEdOriginal(PositionList<E> l){
    try {
      lista = l;
      if (lista != null && !lista.isEmpty())
        cursor = l.first();
      else
        cursor = null;
    }catch (EmptyListException e) {
      cursor = null;
    }
  }
  public boolean hasNext(){ // COMPLETAR }
  public E next() throws NoSuchElementExcepcion{ // COMPLETAR }
}
```

*¿Qué valores toma el atributo **cursor** para identificar que existen o no elementos aún por recorrer en la ED?*

*¿Cómo se utiliza el atributo **cursor** en esta operación? ¿Debe ser actualizado? ¿Qué sucede si no hay próximo elemento a recorrer?*

¿Qué **sucede** si una vez creado el iterador la **lista es modificada**?

El **iterador sólo es válido** si la lista **no es modificada**.

TDA Iterador :: Implementación

- Operando sobre una ED auxiliar que clona la colección a iterar.

```
public class IteradorListaSobreEdAuxiliar<E> implements Iterator<E>{
  private Queue<E> cola;

  public IteradorListaSobreEdAuxiliar(PositionList<E> lista) {
    cola = new LinkedQueue<E>();
    try {
      Position<E> cursor = lista.isEmpty() ? null : lista.first();
      while(cursor != null) {
        cola.enqueue(cursor.element());
        cursor = (cursor == lista.last()) ? null : lista.next(cursor);
      }
    }
    catch(EmptyListException e) { cola = new LinkedQueue<E>();}
    catch(InvalidPositionException e1) { cola = new LinkedQueue<E>();}
    catch(BoundaryViolationException e2) { cola = new LinkedQueue<E>();}
  }

  public boolean hasNext(){ // COMPLETAR }
  public E next() throws NoSuchElementExcepcion{ // COMPLETAR }
}
```

¿Qué **sucede** si una vez creado el iterador la **lista es modificada**?

El **iterador seguirá** siendo **válido** pero las modificaciones no serán visibles en este recorrido.

*¿Qué se debe consultar sobre el atributo **cola** para identificar que existen o no elementos aún por recorrer en la ED?*

*¿Cómo se utiliza el atributo **cola** en esta operación? ¿Cómo se actualiza **cola**?*

Instrucción For-each

Indique qué hace la siguiente porción de código:

```
int[] arr = {1, 2, 3};
for (int i : arr)
        System.out.println(i);
```

Explique brevemente el concepto de iterador en Java.

- El funcionamiento de la instrucción for-each, **podría** verse de la siguiente manera:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
arr.add(10); arr.add(20);
for(Integer i : arr)
  System.out.println(i);
```

```
ArrayList<Integer> arr = new ArrayList<Integer>();
arr.add(10); arr.add(20);
Iterator<Integer> it = arr.iterator();
while(it.hasNext()){
  int i = it.next();
  System.out.println(i);
}
```

- **arr** es un tipo de dato **iterable**, por lo tanto, **puede recorrerse**.
- **arr** es un tipo de dato **iterable**, por lo que brinda un **método iterator()**.
- Mediante un tipo de dato **iterator**, **arr** puede **recorrerse** de **principio** a **fin**.

ITERABLE DE POSICIONES EN POSITIONLIST

Iterable de posiciones

- El TDA Lista ofrece un método que retorna un **iterable de posiciones**.
- ¿Qué es un **iterable** de **posiciones**? (repaso)
  - Un **iterable** es una **colección** que **puede recorrerse**, en este caso, los **elementos** de la colección son **posiciones** de **lista**.
- ¿Qué se puede **hacer** con un **iterable**? (repaso)
  - Una **colección iterable** ofrece un **iterador**, que define el **orden** en el que se **recorren** los **elementos** de la **colección**.

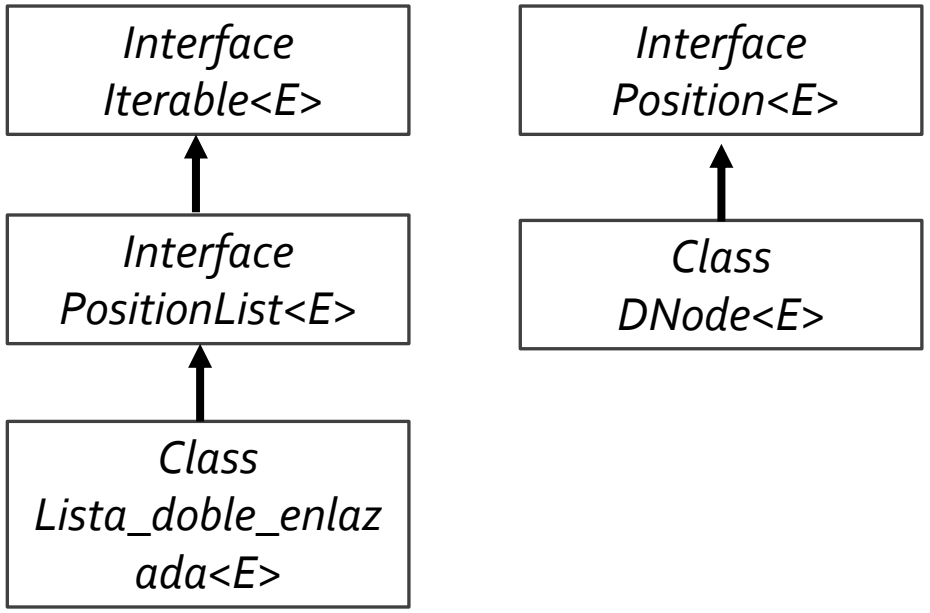
Iterable de **posiciones** → **Colección** de **posiciones** de **lista** que se **puede recorrer**.

Iterable de **posiciones** → Se recorre **mediante** un **iterador** → El **iterador** define el **orden** en el que se **visitan** las **posiciones** de la **colección**.

Iterable de posiciones

- ¿Cómo **generar** un **iterable** de **posiciones**?
  - Se requiere de una **colección** de **posiciones**.
  - Se requiere que la **colección** sea **iterable**.
- ¿Qué **ED** se ha estudiado en la materia que modela una **colección** y es **Iterable**?

```
public class Lista_doble_enlazada<E> implements PositionList<E>{
  protected DNode<E> header, trailer;
  …
  public Iterable<Position<E>> positions() {
    PositionList<Position<E>> iterable;
    iterable = new Lista_doble_enlazada<Position<E>>();
    DNode<E> actual = header.getNext();
    while(actual != trailer) {
      iterable.addLast(actual);
      actual = actual.getNext();
    }
    return iterable;
  }
}
```



Iterable de posiciones: cuándo usarlo

- Los **iterables** de **posiciones** son especialmente útiles cuando se **recorre una lista** y se deben **realizar modificaciones** sobre ella.
- Imagine que dada una **lista de enteros**, se deben **eliminar** todos los elementos **pares** de la misma.

```
public void eliminar_con_iterador(PositionList<Integer> l){
  int elemento;
  Iterator<Integer> it = l.iterator();
  while (it.hasNext()){
    elemento = it.next();
    if (elemento % 2 == 0)
      // Elimina elemento de lista.
  }
}
```

Es una muy mala práctica utilizar un **iterador** de **elementos** en un lista, cuando es necesario contar con las posiciones de los **elementos iterados** para realizar alguna otra operación.

*¿Cómo está definido el método **remove()** en el TDA Lista?*

```
public E remove(Position<E> p) …
```

*Para poder eliminar un elemento, se requiere la posición de lista en la que se encuentra el elemento. Si se utilizan **iteradores** de elementos, será **necesario** recorrer la lista nuevamente para **hallar** la posición a eliminar, lo que es **totalmente ineficiente**.*

*Además, como no se conoce **cómo** es que está **implementado** el **iterador**, al remover un elemento de la lista mientras esta es iterada **puede corromper** el iterador (ver **IteradorListaSobreEdOriginal**).*

Iterable de posiciones: cuándo usarlo

- Los **iterables** de **posiciones** son especialmente útiles cuando se **recorre una lista** y se deben **realizar modificaciones** sobre ella.
- Imagine que dada una **lista de enteros**, se deben **eliminar** todos los elementos **pares** de la misma.

```
public void eliminar_con_recorrido_manual(PositionList<Integer> l){
  int elemento;
  try{
    Position<Integer> actual = l.isEmpty() ? null : l.first();
    Position<Integer> aux;
    while(actual != null && actual != l.last()){
      elemento = actual.element();
      aux = l.next(actual);
      if (elemento % 2 == 0) l.remove(actual);
      actual = aux;
    }
    if (actual != null) && (actual.element() % 2 == 0)
      l.remove(actual);
  }catch(EmptyListException e){
  }catch(InvalidPositionException e1){
  }catch(BoundaryViolationException e2){}
}
```

*Al recorrer la lista "manualmente", se requiere **especial** cuidado al **actualizar el puntero** a la **posición actual**.*
*Es claro que estas **dos líneas** de código **no son intercambiables.** ¿Qué sucedería si primero se **elimina** el elemento de la posición **actual**, y luego se intenta acceder a **l.next(actual)**?*

Iterable de posiciones: cuándo usarlo

- Los **iterables** de **posiciones** son especialmente útiles cuando se **recorre una lista** y se deben **realizar modificaciones** sobre ella.
- Imagine que dada una **lista de enteros**, se deben **eliminar** todos los elementos **pares** de la misma.

```
public static void eliminar_con_positions(PositionList<Integer> l){
  try{
    for(Position<Integer> pos : l.positions()){
      if (pos.element() % 2 == 0)
        l.remove(pos);
    }
  }catch(InvalidPositionException e){}
}
```

Para pensar: ¿qué es más eficiente? ¿recorrer la lista "manualmente" o bien mediante un iterable de posiciones?

Analice esto comparando el ***tiempo de ejecución***, y el ***orden del tiempo de ejecución***.

Fin de la presentación.