

Estructuras de Datos

Iteradores

Adriana Collaguazo Jaramillo, Mg.

La Interface Iterable

Utilidad

Permite el uso de una versión mejorada de lazo **for**

A este tipo de lazo se lo conoce como **for-each**

```
for (Item item: customDataStructure) {  
    // do stuff  
}
```

Recorriendo ArrayList y LinkedList

```
ArrayList al = new ArrayList();  
al.add(3);  
al.add(2);  
al.add(1);  
al.add(4);  
al.add(5);  
al.add(6);  
al.add(6);  
  
Iterator iter1 = al.iterator();  
while(iter1.hasNext()){  
    System.out.println(iter1.next());  
}
```

```
LinkedList ll = new LinkedList();  
ll.add(3);  
ll.add(2);  
ll.add(1);  
ll.add(4);  
ll.add(5);  
ll.add(6);  
ll.add(6);  
  
Iterator iter2 = ll.iterator();  
while(iter2.hasNext()){  
    System.out.println(iter2.next());  
}
```

Recorriendo ArrayList y LinkedList

```
ArrayList al = new ArrayList();  
al.add(3);  
al.add(2);  
al.add(1);  
al.add(4);  
al.add(5);  
al.add(6);  
al.add(6);
```

```
Iterator iter1 = al.iterator();  
while(iter1.hasNext()){  
    System.out.println(iter1.next());  
}
```

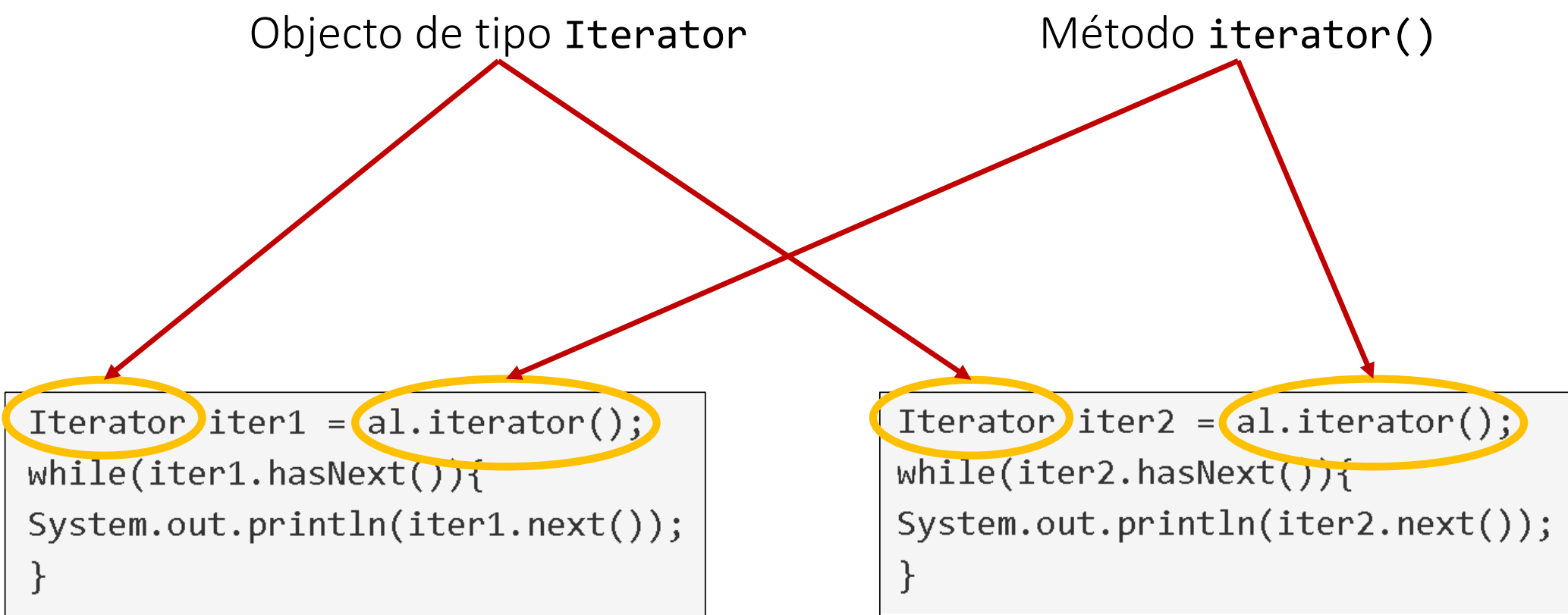
```
LinkedList ll = new LinkedList();  
ll.add(3);  
ll.add(2);  
ll.add(1);  
ll.add(4);  
ll.add(5);  
ll.add(6);  
ll.add(6);
```

```
Iterator iter2 = ll.iterator();  
while(iter2.hasNext()){  
    System.out.println(iter2.next());  
}
```

Recorriendo ArrayList y LinkedList

Objeto de tipo Iterator

Método iterator()



```
Iterator iter1 = al.iterator();  
while(iter1.hasNext()){  
    System.out.println(iter1.next());  
}
```

```
Iterator iter2 = al.iterator();  
while(iter2.hasNext()){  
    System.out.println(iter2.next());  
}
```

¿Cómo Implementarla?

Implementar la interface **Iterable**

Crear una clase que implemente la interface **Iterable**

```
class CustomDataStructure implements Iterable<E> {  
  
    // code for data structure  
    public Iterator<E> iterator() {  
        return new CustomIterator<>(this);  
    }  
  
}
```

Class CustomIterator

```
class CustomIterator<E> implements Iterator<E> {  
  
    // constructor  
    CustomIterator<E>(CustomDataStructure obj) {  
        // initialize cursor  
    }  
  
    // Checks if the next element exists  
    public boolean hasNext() {  
    }  
  
    // moves the cursor/iterator to next element  
    public E next() {  
    }  
  
    // Used to remove an element. Implement only if needed  
    public void remove() {  
        // Default throws UnsupportedOperationException.  
    }  
}
```


El método `iterator`

Retorna una instancia de la interface **Iterator**

Dicha instancia guarda el estado de la iteración (en qué punto del recorrido vamos)

Debe producir un nuevo iterador cada vez que es invocado

No debe retornar el mismo iterador dos veces. De otro modo, hay conflictos

Cómo podemos guardar el estado de la iteración de una lista?

Interface Iterator

El método `iterator()` retorna un objeto de tipo `Iterator`

Un iterador es una es una abstracción que provee acceso a los elementos de la lista

Quien usa el TDA no tiene que instanciar un nodo viajero ni llamar a `getContent`

En Java, `Iterator` es una interface parametrizada por tipo.

```
java.util
```

```
Interface Iterator<E>
```

```
Type Parameters:
```

```
E - the type of elements returned by this iterator
```

Interface Iterator

All Methods	Instance Methods	Abstract Methods	Default Methods
Modifier and Type		Method and Description	
boolean		hasNext()	Returns true if the iteration has more elements.
E		next()	Returns the next element in the iteration.

Toda clase que implementa esta interfaz debe definir dos métodos:

```
boolean hasNext ();
```

```
E next();
```

<https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html>

Qué hacen `next()` y `hasNext()`?

Un iterador debe hacer seguimiento del elemento en el que está actualmente

Debe permitir avanzar de un elemento a otro

Esto se hace en el método **`next()`** que:

1. devuelve el elemento actual y,
2. avanza el cursor al siguiente elemento

Antes de avanzar, verificamos si hay siguiente elemento:

```
while(iterator.hasNext()) { //if next element exists
    E e = iterator.next(); // advance the pointer
}
```

Resumen

`boolean hasNext ();`
`E next();`

`Iterator iterator ();`

```
Iterator iter1 = al.iterator();  
while(iter1.hasNext()){  
    System.out.println(iter1.next());  
}
```

Implementando Iterators

Forma 1: Clase que implemente la interface Iterator

```
public class MyIterator<E> implements Iterator {  
    // interface implementation  
}  
  
    public Iterator<E> iterator() {  
        return new MyIterator<E>();  
    }
```

Forma 2: Creando una clase anónima

```
public Iterator<E> iterator() {  
  
    Iterator<E> it = new Iterator<E>(){  
        // interface implementation  
    }  
  
    return it;  
}
```


La Interface ListIterator

Interfaz ListIterator

Permite implementar iteradores para recorrer la lista en cualquier dirección:

Method Summary

Methods

Modifier and Type	Method and Description
void	add(E e) Inserts the specified element into the list (optional operation).
boolean	hasNext() Returns true if this list iterator has more elements when traversing the list in the forward direction.
boolean	hasPrevious() Returns true if this list iterator has more elements when traversing the list in the reverse direction.
E	next() Returns the next element in the list and advances the cursor position.
int	nextIndex() Returns the index of the element that would be returned by a subsequent call to next() .
E	previous() Returns the previous element in the list and moves the cursor position backwards.
int	previousIndex() Returns the index of the element that would be returned by a subsequent call to previous() .
void	remove() Removes from the list the last element that was returned by next() or previous() (optional operation).
void	set(E e) Replaces the last element returned by next() or previous() with the specified element (optional operation).

Interfaz ListIterator

Permite implementar iteradores para recorrer la lista en cualquier dirección:

Method Summary	
Methods	
Modifier and Type	Method and Description
void	add(E e) Inserts the specified element into the list (optional operation).
boolean	hasNext() Returns true if this list iterator has more elements when traversing the list in the forward direction.
boolean	hasPrevious() Returns true if this list iterator has more elements when traversing the list in the reverse direction.
E	next() Returns the next element in the list and advances the cursor position.
int	nextIndex() Returns the index of the element that would be returned by a subsequent call to next() .
E	previous() Returns the previous element in the list and moves the cursor position backwards.
int	previousIndex() Returns the index of the element that would be returned by a subsequent call to previous() .
void	remove() Removes from the list the last element that was returned by next() or previous() (optional operation).
void	set(E e) Replaces the last element returned by next() or previous() with the specified element (optional operation).