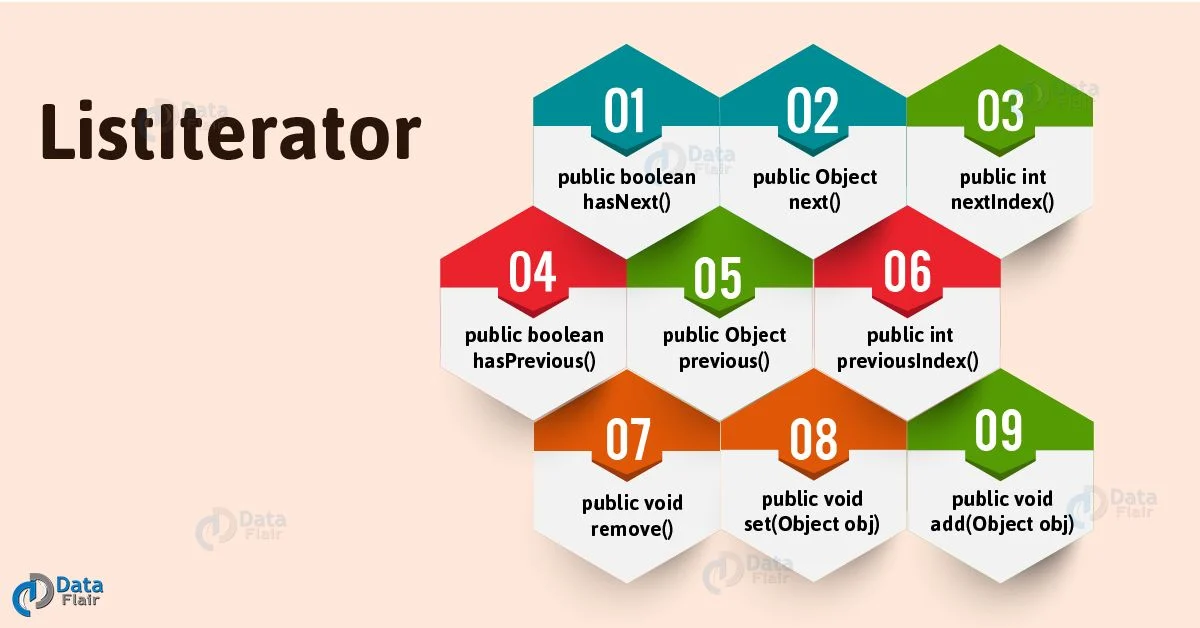
TRABAJO API - PROGRAMACIÓN:

LA INTERFAZ LIST-ITERATOR



***ÍNDICE***

* ***// RESUMEN DE LA INTERFAZ ITERATOR***

**/\* 1 -** Explicación del concepto **(Pág 2)**

**2 -** Sintaxis **(Pág 2)**

**3 -** Métodos **(Pág 3)\*/**

* ***// LA INTERFAZ LIST-ITERATOR***

**/\* 1 -** Diferencias con Iterator **(Pág 4)**

**2 -** Funcionamiento **(Pág 4 - Pág 7)**

**3 -** Métodos **(Pág 7 - Pág 16)**

**4 -** Excepciones **(Pág 16 - Pág 18)\*/**

* ***// CONCLUSIÓN***

***RESUMEN DE LA INTERFAZ ITERATOR***

**1 *EXPLICACIÓN DEL CONCEPTO***

Para empezar a hablar acerca de este tema primero necesitamos aclarar brevemente los conceptos básicos, ya que su entendimiento es fundamental para que no se nos atragante lo que se va a explicar a continuación, así que allá vamos:

***¿QUÉ ES UN ITERADOR?***

**En programación, un iterador se refiere al objeto que le permite al programador recorrer un contenedor, (una colección de elementos) particularmente listas.**

En java, disponemos de la interfaz *“Iterator”* para recorrer todo tipo de colecciones que contengan una lista de elementos, como List, LinkedList o Set, entre otras.

Eso sí, Iterator sólo puede recorrer una colección en un sólo sentido y no puede modificar los datos de dicha colección.

Aunque lo bueno de que el recorrido sea unidireccional es que permite recorrer colecciones de elementos que carezcan de un índice, ya que en Java no todas las colecciones de objetos tienen un índice entero asociado a cada objeto (la interfaz Set es un ejemplo).

**2 *SINTAXIS***

La sintaxis de un Iterador es la siguiente:

***Iterator it = nombreDeLaColeccion.iterator ();***

En la que “it” es el nombre del objeto que estamos declarando y “nombreDeLaColeccion” el nombre de la colección que queremos recorrer.

Algo que cabe aclarar es que la sintaxis es así de extraña porque LA INTERFAZ ITERATOR CARECE DE CONSTRUCTOR por lo que no se puede utilizar la sentencia de tipo **“new Iterator<TipoARecorrer> ( );”** ya que no resulta válida.

**3 *MÉTODOS***

La interfaz Iterator solamente tiene estos métodos:

| **INTERFAZ ITERATOR** | | |
| --- | --- | --- |
| **Devuelve** | **Nombre del método** | **Uso del método** |
| E (Un objeto) | **Next()** | Obtiene el siguiente objeto de la colección. Si se ha llegado al final de la colección y se intenta seguir, da lugar a una excepción de tipo: **NoSuchElementException**. |
| boolean | **hasNext()** | Indica si hay un elemento siguiente (y así evita la excepción). |
| void | **Remove()** | Elimina el último elemento devuelto por next(). |

También, aunque no aparece en la tabla de arriba, Iterator tiene en realidad un cuarto método llamado **forEachRemaining(Consumer<? super E> action)**,de tipo void. Fue añadido en Java 8 y lo que hace es básicamente realizar una acción dada para cada elemento restante de la lista hasta que se hayan procesado todos ellos o la acción genere una excepción.

En concreto, este método toma como parámetro una expresión lambda, así que todavía no podemos explicarlo ya que las lambdas no las hemos visto. Aún así, veamos alguna de sus utilidades:

Un uso común para este método, por ejemplo, sería imprimir la lista entera con el iterador. Para ello, bastaría con utilizar un forEachRemaining() que recorra la lista y vaya imprimiendo sus objetos.

Para que nos hagamos una idea, tendría el mismo comportamiento que este bucle:

***while*** *(it.hasNext()) {*

*System.out.println(it.next());*

*}*

¡Bien! Ahora que entendemos el Iterator, pasemos a explicar el ListIterator.

***LA INTERFAZ LIST-ITERATOR***

La interfaz ListIterator hereda directamente de Iterator y prácticamente tiene su misma utilidad aunque con más funcionalidades.

La sintaxis de ListIterator es prácticamente igual a la de su madre:

***ListIterator listIt = nombreDeLaColeccion.listIterator ();***

También se puede escribir de esta forma si se quiere especificar el tipo de objeto con el que va a trabajar el ListIterator:

***ListIterator <TipoObjeto> listIt = nombreDeLaColeccion.listIterator ();***

**1 *DIFERENCIAS CON ITERATOR***

***ES BIDIRECCIONAL Y PUEDE MODIFICAR ELEMENTOS***

La principal característica que diferencia al ListIterator del Iterator es que mientras que este último sólo puede recorrer la lista en una dirección y no puede modificar los objetos de dicha lista, ListIterator puede recorrer la lista bidireccionalmente, modificar la lista durante la iteración y obtener la posición actual del iterador en la lista. ¡Es mucho más versátil!

***SÓLO FUNCIONA CON LISTAS***

Otra peculiaridad que debemos saber acerca de ListIterator es que solamente puede utilizarse para recorrer listas como List, ArrayList o LinkedList, no es apto para colecciones con otro tipo de estructura que difiera de una lista común, como lo son HashSet o TreeSet.

Esto ocurre porque no tendría sentido recorrer, modificar e intercambiar elementos de lugar si el contenedor que los engloba no los guarda en un orden específico y/o carece además de índice.

**2 *FUNCIONAMIENTO***

Ha llegado la hora de explicar el funcionamiento del ListIterator.

Para empezar, vamos a crear una lista tipo ArrayList y la vamos a rellenar con héroes griegos, porque todo se entiende mejor con temáticas divertidas, y que mejor que con leyendas helenas.

Creamos una clase Principal (para probar todo aquello que se vaya haciendo) y una clase Heroe con los siguientes atributos (además de instaurar el constructor, los getters y setters y el To string):

***public*** *String nombre;*

***public*** *String arma;*

***public******int*** *numHazanhas;*

***public******int*** *numHijos;*

Perfecto, ahora necesitamos rellenar la lista, y para ello le añadimos 4 héroes de los más famosos con el método add() de la colección List.

*lista.add(****new*** *Heroe("Heracles", "Clava", 12, 9));*

*lista.add(****new*** *Heroe("Aquiles", "Xifos", 3, 2));*

*lista.add(****new*** *Heroe("Jasón", "Lanza", 4, 7));*

*lista.add(****new*** *Heroe("Odiseo", "Espada", 24, 1));*

Tras ello, creamos el objeto ListIterator que va a recorrer la lista.

Una vez hecho todo esto, la clase Principal quedaría de esta forma:

***public******class*** *Principal {*

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

*//* ***TODO*** *Auto-generated method stub*

*List <Heroe> lista =* ***new*** *ArrayList <>();*

*lista.add(****new*** *Heroe("Heracles", "Clava", 12, 9));*

*lista.add(****new*** *Heroe("Aquiles", "Xifos", 3, 2));*

*lista.add(****new*** *Heroe("Jasón", "Lanza", 4, 7));*

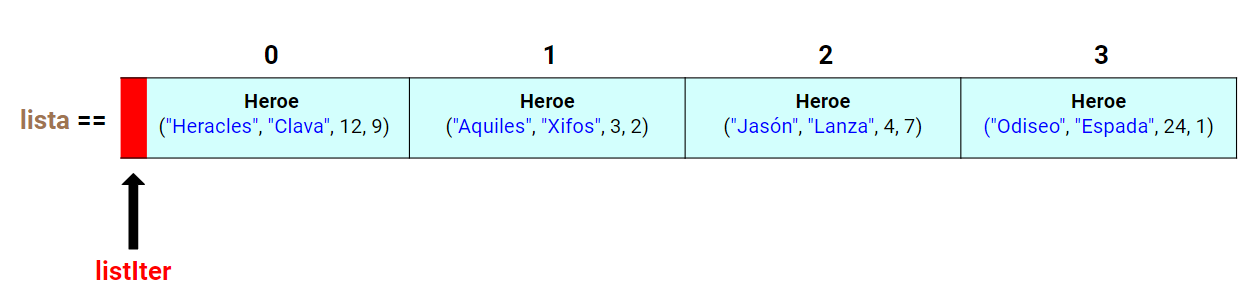
*lista.add(****new*** *Heroe("Odiseo", "Espada", 24, 1));*

*ListIterator listIter = lista.listIterator();*

*}*

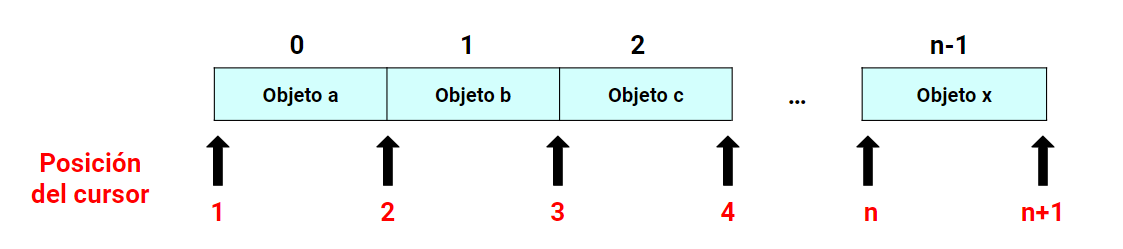
*}*

Ahora, para entender de forma sencilla el funcionamiento del ListIterator lo ilustraremos con una serie de imágenes. El ejemplo que hemos creado se vería de esta forma:



La serie de cuadros azules es la lista *“lista”* y cada cuadro representa una posición de esta que contiene el objeto que se le haya asignado. Los números situados encima de la lista son el índice de la posición en la que se encuentra cada elemento, y la barra roja es el cursor del ListIterator *“ListIter”*.

Un ListIterator no tiene ningún elemento actual. La posición de su cursor siempre se encuentra ENTRE elementos. **Un iterador para una lista de longitud n tiene n+1 posibles posiciones del cursor.**



Para dejarlo más claro y volviendo a nuestro ejemplo, vamos a hacer que el puntero del ListIterator avance una posición hacia adelante utilizando el método next().

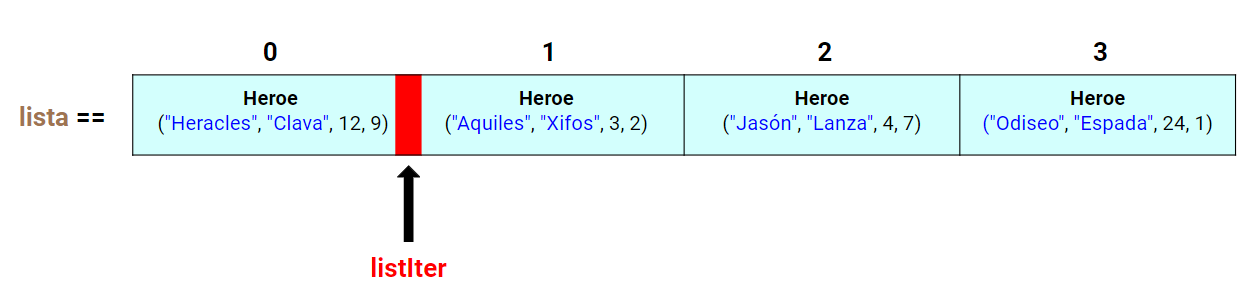
Introducimos la siguiente sentencia:

*System.out.println(listIter.next());*

El programa nos devuelve lo siguiente:

*Heroe [nombre=Heracles, arma=Clava, numHazanhas=12, numHijos=9]*

Y ahora tenemos esto:



Lo que ha ocurrido es que el método next() ha devuelto el elemento que el cursor tenía delante, y al hacerlo, ha avanzado una posición (hacia delante).

¡Perfecto! Ahora que hemos comprendido el funcionamiento del ListIterator es hora de echarle un vistazo a sus métodos ¡Vamos a ello!

**3 *MÉTODOS***

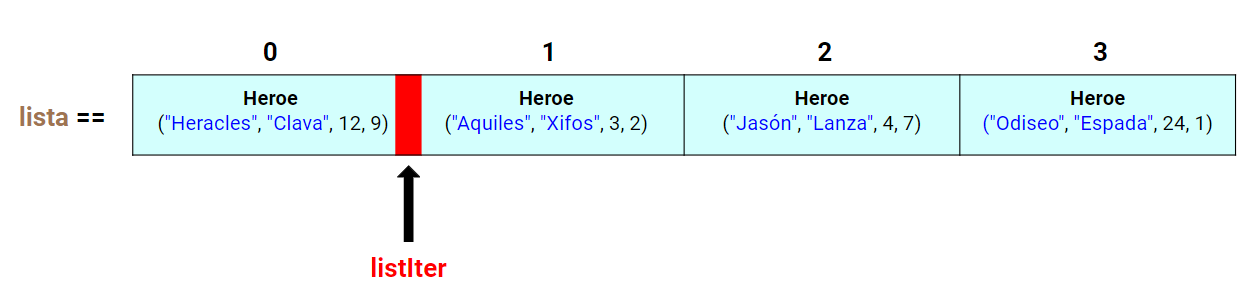
**add (E e) Método void.**

Agrega un elemento especificado en la lista.

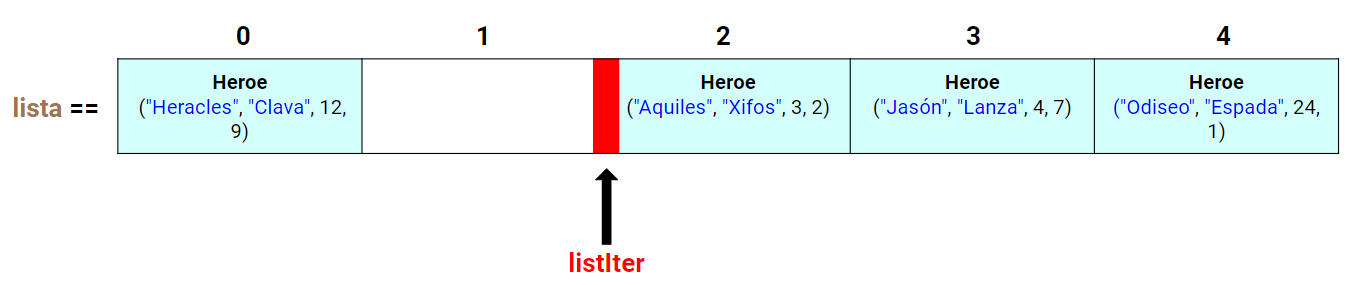
Supongamos que el cursor del iterador está otra vez en el principio de la lista y queremos incluir en nuestra lista a otro héroe más: Perseo, entre Heracles y Aquiles, y para ello debemos utilizar este método.

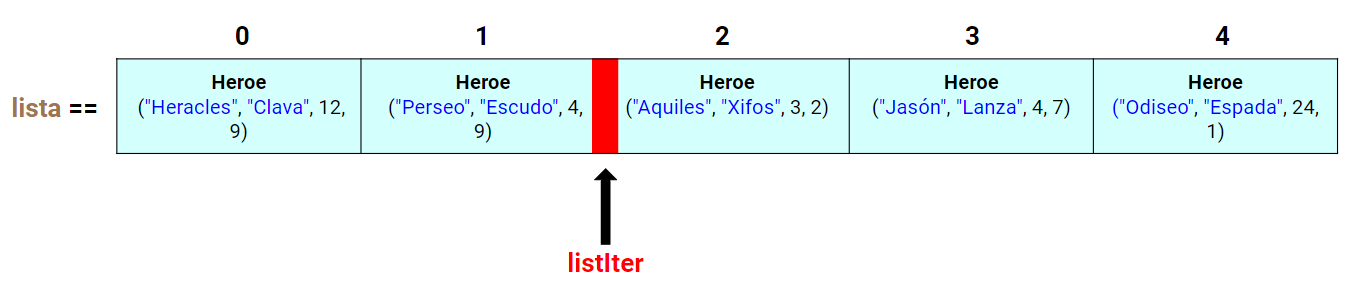
Lo único que habría que hacer sería colocar el puntero entre Heracles y Aquiles con método next() y seguido incluir a Perseo con add(), así:

*listIter.next();*



*listIter.add(****new*** *Heroe("Perseo", "Escudo", 4, 9));* ***//****¡OJO A LO QUE OCURRE!*



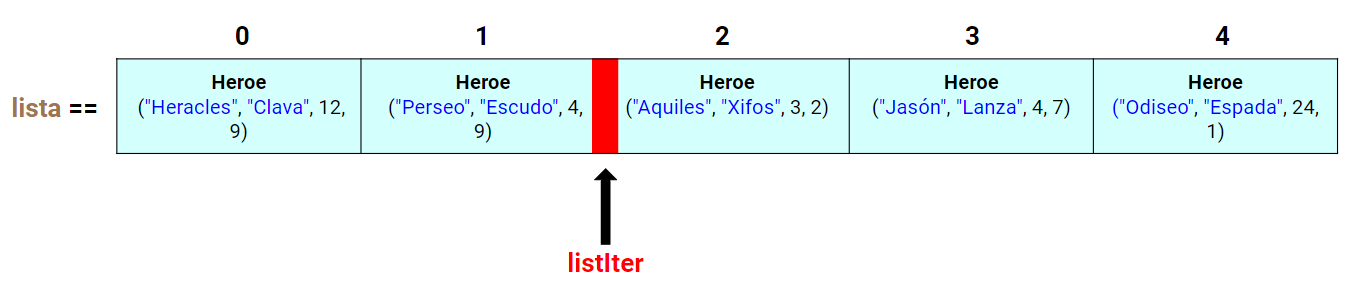


Cuando se utiliza el add() en una lista, el iterador crea un espacio en blanco en el que añade la información que le ha sido proporcionada. Cabe destacar el hecho de que el elemento que va a ser añadido siempre aparecerá en la posición trasera al cursor del ListIterator, y por tanto, dicho puntero permanecerá delante del objeto que se acaba de agregar. También, el índice de la lista se adapta automáticamente a la adición de elementos.

**hasNext() Devuelve boolean.**

Devuelve *“true”* si el iterador de lista tiene más elementos delante al recorrer dicha lista.

Esta es nuestra lista actual:



Introducimos la sentencia:

*System.****out****.println(listIter.hasNext());*

El programa nos devuelve:

*true*

Esto ocurre porque delante del cursor del iterador hay un elemento, en este caso, el héroe Aquiles.

Si no hubiese un elemento delante del puntero del iterador, el programa devolvería *“false”*.

**hasPrevious() Devuelve boolean.**

Devuelve *“true”* si el iterador de lista tiene más elementos detrás al recorrer dicha lista en la dirección inversa.

Es idéntico al método hasNext() sólo que con el elemento que se encuentra detrás del cursor del iterador.

Suponiendo que el cursor se encuentra entre Heracles y Perseo:

*System.****out****.println(listIter.hasPrevious());*

El programa devuelve:

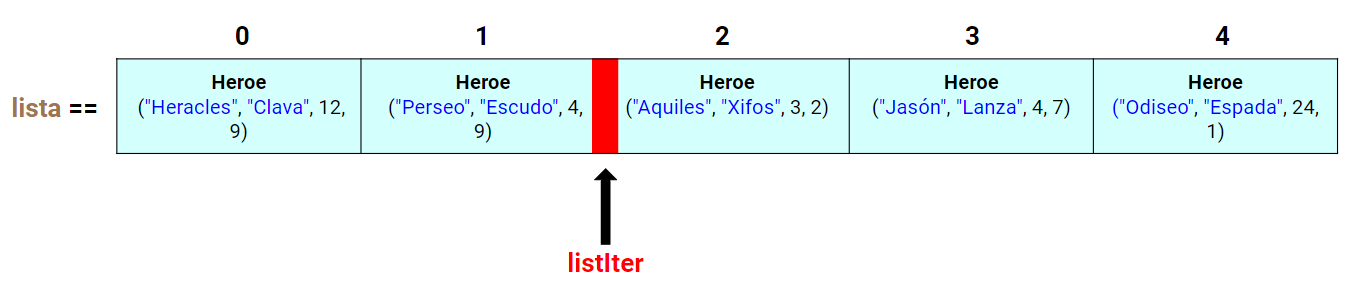
*true*

Si no hubiese un elemento detrás del puntero del iterador, el programa devolvería *“false”*.

**next() Devuelve E.**

Obtiene el siguiente objeto de la colección y avanza el cursor del iterador una posición hacia delante.

Esta es nuestra lista actual:



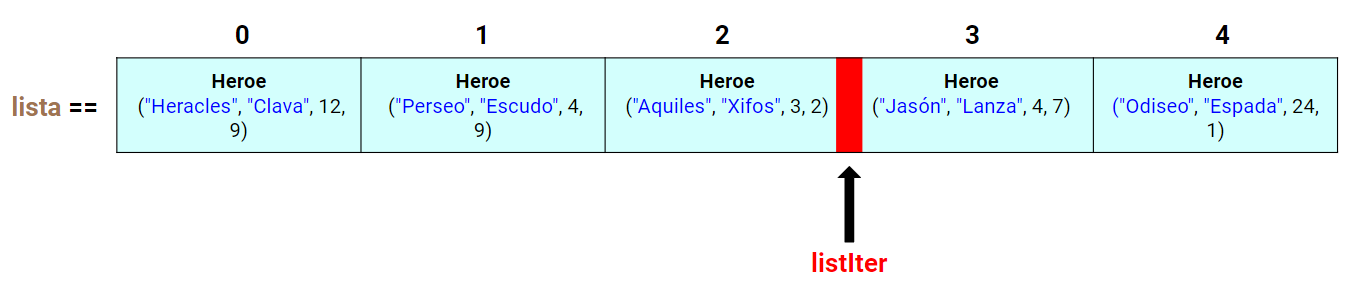
Introducimos la siguiente sentencia:

System.***out***.println(listIter.Next());

El programa devuelve:

Heroe [nombre=Aquiles, arma=Xifos, numHazanhas=3, numHijos=2]

Ahora tenemos:



**nextIndex() Devuelve int.**

Devuelve el índice del elemento que sería devuelto por una llamada posterior a next().

Actualmente, el cursor del iterador se encuentra entre Aquiles y Jasón:

*System.****out****.println(listIter.nextIndex());*

El programa devuelve:

*3*

Devuelve *“3”* ya que el elemento que se encuentra delante del indicador es aquel que tiene índice *“3”* en la posición de la lista.

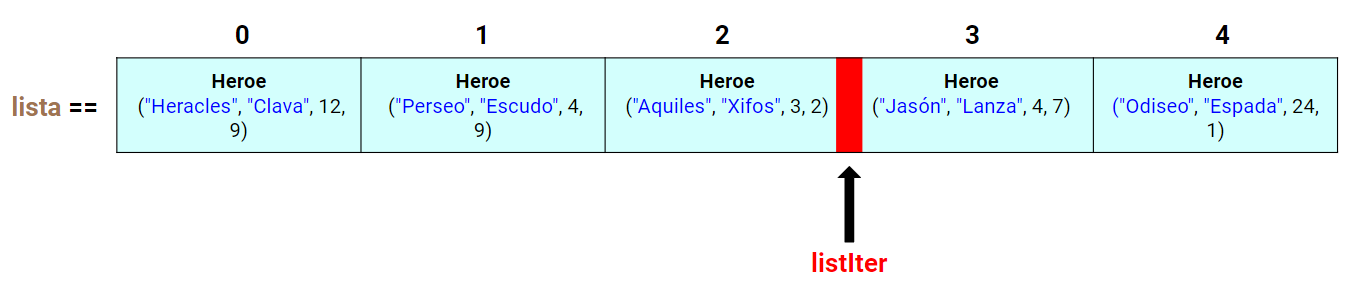
Si el puntero se encontrara entre Heracles y Perseo, por ejemplo, el programa devolvería *“1”*, ya que Perseo se sitúa en dicha posición.

**/\***Como curiosidad, si el cursor del ListIterator se encuentra al final de la lista y utilizamos este método, el programa devolverá *“-1”*.**\*/**

**previous() Devuelve E.**

Obtiene el anterior objeto de la colección y avanza el cursor del iterador una posición hacia atrás.

Esta es nuestra lista:



*System.****out****.println(listIter.previous());*

El programa devuelve:

*Heroe [nombre=Aquiles, arma=Xifos, numHazanhas=3, numHijos=2]*

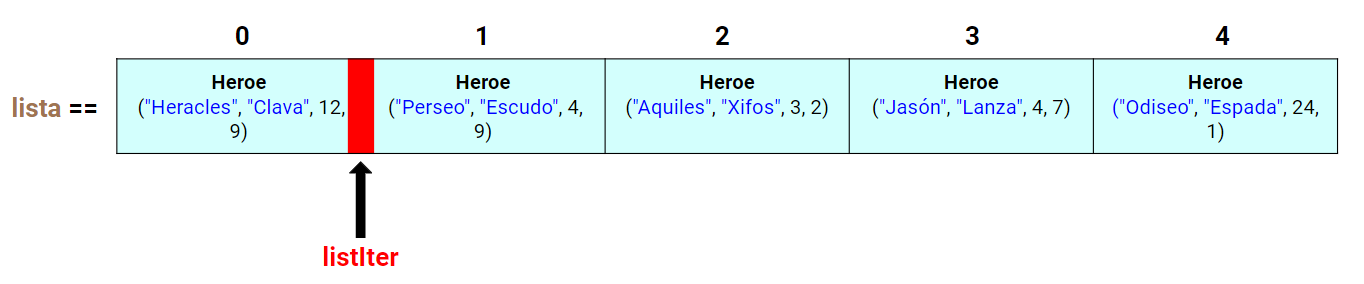
Y lo utilizamos otra vez:

*System.****out****.println(listIter.previous());*

El programa devuelve:

*Heroe [nombre=Perseo, arma=Escudo, numHazanhas=4, numHijos=9]*

Y así ha quedado lo nuestro:



Es básicamente como el método next() pero al revés.

**previousIndex() Devuelve int.**

Devuelve el índice del elemento que sería devuelto por una llamada posterior a previous().

Actualmente, el cursor se encuentra entre Heracles y Perseo, así que para tener más margen de maniobra, vamos a escribir la siguiente sentencia 3 veces: *listIter.Next();*

Ahora el puntero se encuentra entre Jasón y Odiseo.

Escribimos:

*System.****out****.println(listIter.previousIndex());*

El programa devuelve:

*3*

Este método funciona exactamente igual que nextIndex() sólo que devuelve el índice de la posición en la que se encuentra el elemento trasero al cursor del iterador.

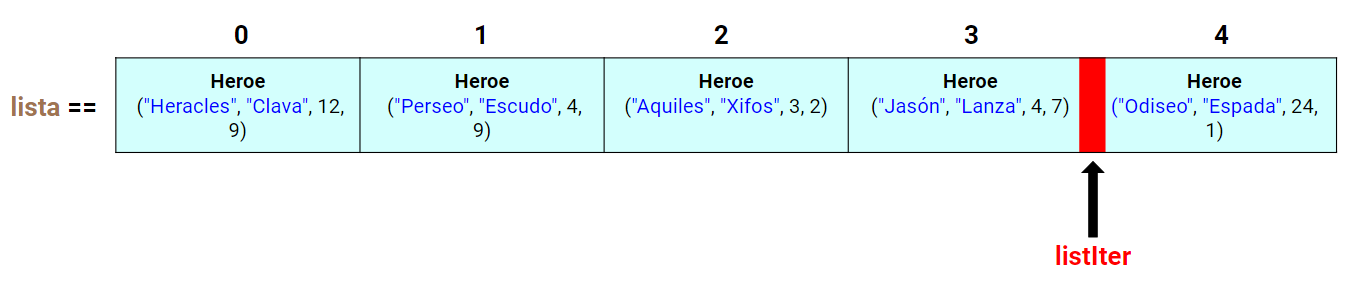
**/\***Al igual que con nextIndex(), si el cursor del ListIterator se encuentra al principio de la lista y utilizamos este método, el programa devolverá “-1”.**\*/**

**remove() Método void.**

Elimina el último elemento devuelto por next() o previous().

Recalco la parte de *“EL ÚLTIMO ELEMENTO DEVUELTO POR next() O previous()”*, ya que si no llamamos a ninguno de esos 2 métodos antes de utilizar este, no funcionara y dará error (que explicaremos más adelante en otro apartado).

Así es nuestra lista:



Vamos a probar este método con un previous().

Introducimos la siguiente sentencia:

*System.****out****.println(listIter.previous());*

El programa devuelve:

*Heroe [nombre=Jasón, arma=Lanza, numHazanhas=4, numHijos=7]*

Y si seguido llamamos al método remove() e imprimimos la lista:

*listIter.remove();*

***for*** *(Heroe h : lista) {*

*System.****out****.println(h);*

*}*

El programa devuelve:

*[Heroe [nombre=****Heracles****, arma=Clava, numHazanhas=12, numHijos=9]*

*[Heroe [nombre=****Perseo****, arma=Escudo, numHazanhas=4, numHijos=9]*

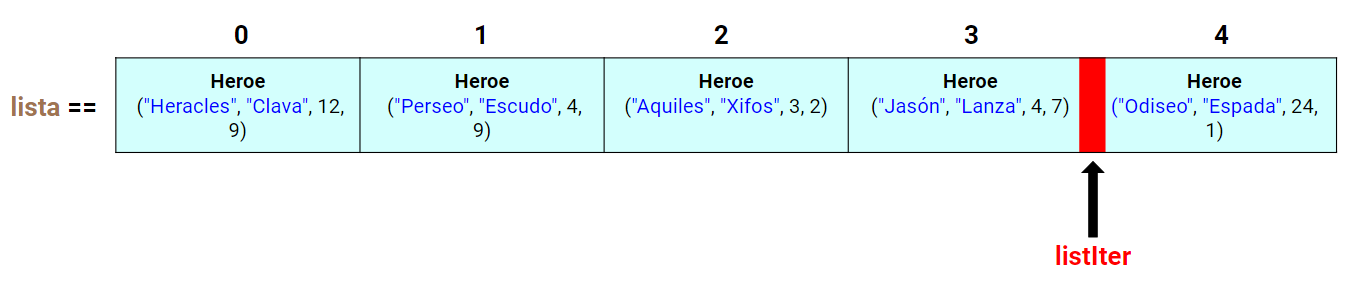
*[Heroe [nombre=****Aquiles****, arma=Xifos, numHazanhas=3, numHijos=2]*

*[Heroe [nombre=****Odiseo****, arma=Espada, numHazanhas=24, numHijos=1]*

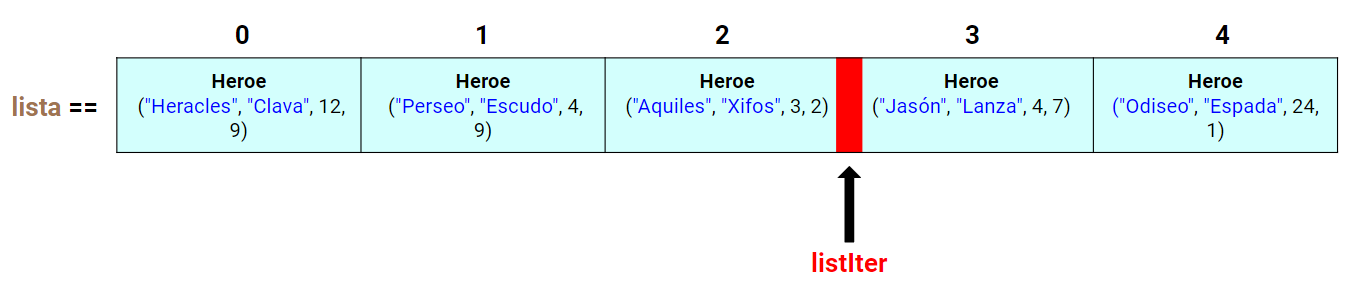
Si nos fijamos ¡Jasón no está en la lista! Ha sido eliminado porque ha sido el último elemento devuelto por previous().

El iterador ha realizado dicha operación de esta forma:

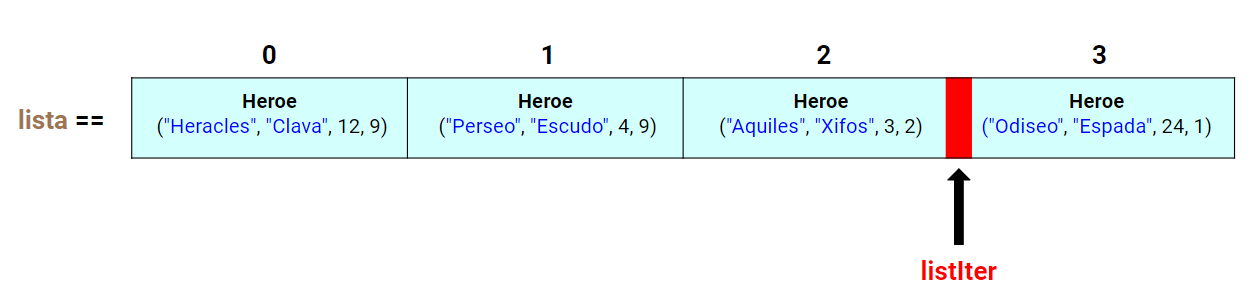
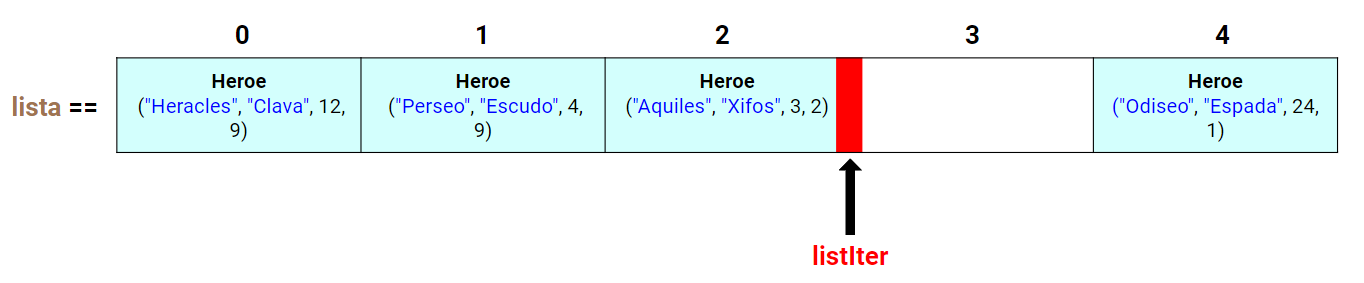
Lista inicial:



*System.****out****.println(listIter.previous());*



*listIter.remove();*



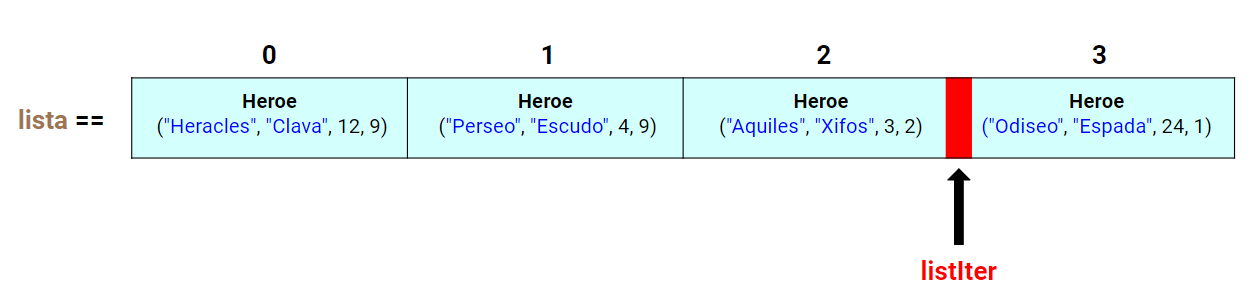
Además, también podemos darnos cuenta de que aparte de borrar el objeto, los índices de la lista situados por delante del cursor del iterador se han adecuado a la eliminación del objeto.

Con next() funciona de la misma forma, si volvemos al caso inicial y después del previous() llamamos al método next() 2 veces y seguido sentenciamos el método remove(), El héroe que habría sido borrado hubiera sido Odiseo.

**set(E e) Método void.**

Reemplaza el último elemento devuelto por next() o previous() con un elemento especificado.

Esta es nuestra lista:



Y nos damos cuenta de que en lugar de Odiseo, debíamos haber añadido a otro héroe: Diómedes. Para cambiar la información e incluir la nueva utilizamos este método.

Como queremos sustituir los datos de Odiseo, debemos hacer que sea el último objeto devuelto por previous() o next() (en este caso next()) para poder utilizar el método set():

*System.****out****.println(listIter.next());*

*listIter.set(****new*** *Heroe("Diómedes", "Xifos Espartano", 4, 0));*

Ahora imprimimos la lista:

***for*** *(Heroe h : lista) {*

*System.****out****.println(h);*

*}*

Y el programa devuelve:

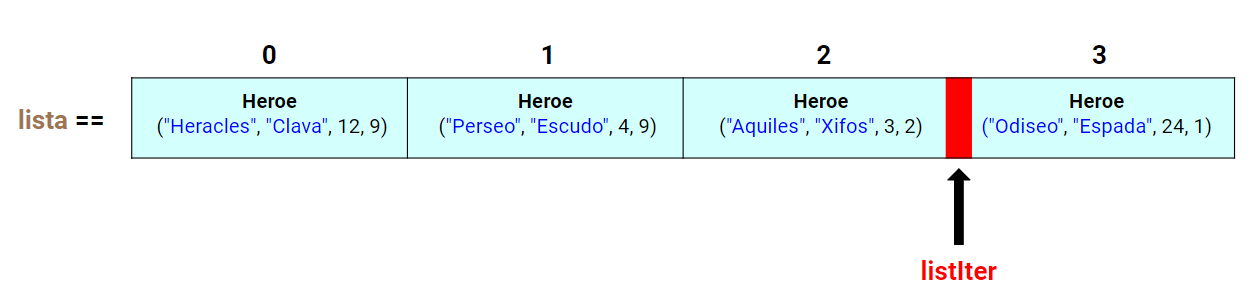
*[Heroe [nombre=Heracles, arma=Clava, numHazanhas=12, numHijos=9]*

*[Heroe [nombre=Perseo, arma=Escudo, numHazanhas=4, numHijos=9]*

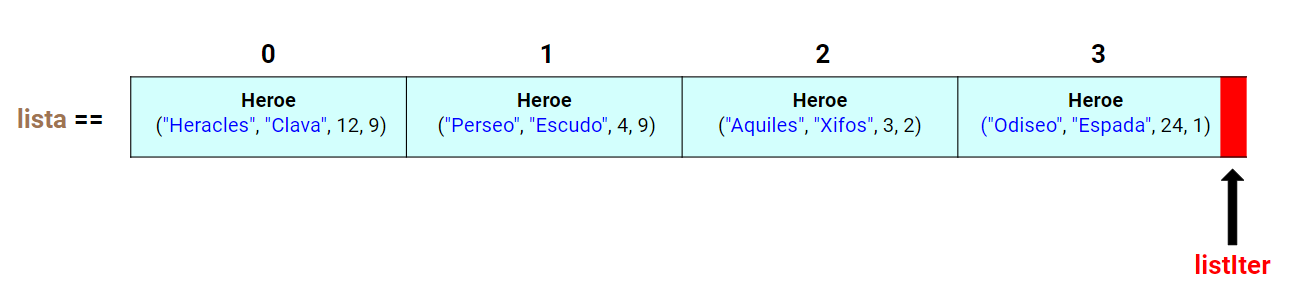
*[Heroe [nombre=Aquiles, arma=Xifos, numHazanhas=3, numHijos=2]*

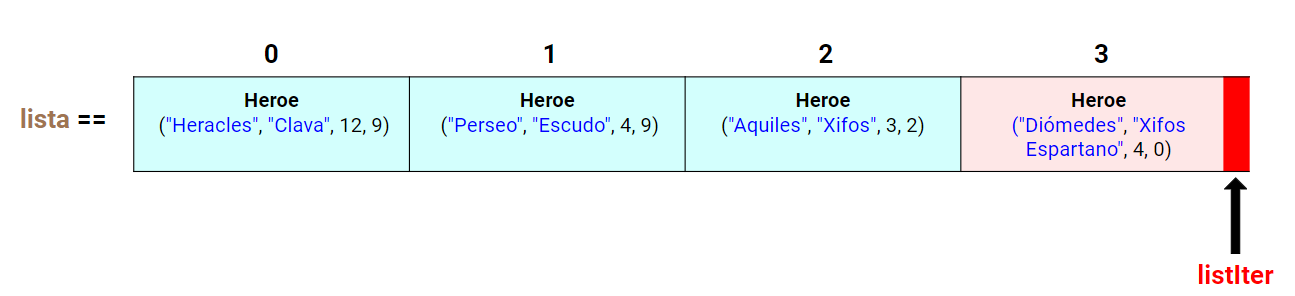
*[Heroe [nombre=Diómedes, arma=Xifos Espartano, numHazanas=4, numHijos=0]*

Como podemos observar, Odiseo ha sido sustituido por Diómedes con éxito ya que el iterador ha detectado cuál ha sido el último objeto devuelto por next() y lo ha sustituido con set(). Esto se vería de la siguiente forma:



*System.****out****.println(listIter.next());*

*listIter.set(****new*** *Heroe("Diómedes", "Xifos Espartano", 4, 0));*



**forEachRemaining(Consumer<? super E> action) Método void.**

Realiza la acción dada para cada elemento restante de la lista hasta que se hayan procesado todos ellos o la acción genere una excepción.

Este método es exactamente igual en Iterator que en ListIterator.

¡Tremendo! Ahora que sabemos acerca de todos los métodos de esta interfaz, es hora de pasar con las excepciones, ya que en este mundo no todo es color de rosas y menos con el ListIterator, que si de excepciones hablamos… Pocas precisamente no tiene.

**4 *EXCEPCIONES***

Con el ListIterator hay que ser cuidadoso a la hora de usarlo, ya que a pesar de ser una interfaz muy versátil con métodos muy útiles, tiene un montón de excepciones que nos pueden salir por donde menos nos lo esperemos, literalmente.

Al igual que en el apartado anterior, vamos a explicar las causas de las excepciones haciendo referencia al método con el que se pueden desencadenar.

**add(E e)**

**Devuelve: ConcurrentModificationException** si se añade un elemento a la lista con este método y previamente se había añadido otro objeto a dicha lista con el método add() de la colección List.

Esto ocurre porque las clases de Collection son *“fail-fast”,* lo que quiere decir que no pueden ser editadas por cualquier otro método (que no sea de un iterador) si sobre una de ellas está actuando un iterador.

**Devuelve: UnsupportedOperationException** si se utiliza este método para editar una lista inmodificable (Esta se puede instaurar con el método .asList() de la clase Arrays, el cual impide modificar una lista).

**Devuelve: ClassCastException** si la clase del elemento especificado impide que se agregue a la lista.

**next()**

**Devuelve: NoSuchElementException** si la iteración no tiene ningún elemento siguiente.

**previous()**

**Devuelve: NoSuchElementException** si la iteración no tiene ningún elemento anterior.

**remove()**

**Devuelve: IllegalStateException** si antes de utilizar el método no se ha llamado a next() o previous(), o se ha llamado a un remove() o add() .

**Devuelve: UnsupportedOperationException** si se utiliza este método para reformar una lista inmodificable.

**set(E e)**

**Devuelve: UnsupportedOperationException** si se utiliza este método para reformar una lista inmodificable.

**Devuelve: IllegalStateException** si antes de utilizar el método no se ha llamado a next() o previous(), o se ha llamado a un remove() o add() .

**Devuelve: IllegalArgumentException** si alguna característica del elemento que vayamos a meter impide que dicho objeto se agregue a la lista.

**Devuelve: ClassCastException** si la clase del elemento especificado impide que se agregue a la lista.

***CONCLUSIÓN***

Y oficialmente, queda explicado el ListIterator. Espero que la lectura se haya hecho amena y haya servido para aprender un poquito más acerca del apasionante mundo de la programación.

¡Hasta la próxima y gracias por su atención!

***TRABAJO REALIZADO POR: LUCAS FALLA URTIAGA.***

***ALUMNO DE 1ºDAM (2023 - 2024).***