Documentazione Test – `ListAdapterTest`

Corso di Ingegneria del Software

Dal Bianco Luca

Class: `ListAdapterTest`

Questa classe contiene tutti i test per verificare la corretta implementazione di `ListAdapter` secondo le specifiche Java 1.4.2.

testEmptyList

Summary

Verifica che una lista appena creata sia vuota, controllando `size()` e `isEmpty()`.

Test Case Design

Obiettivo: confermare che l'istanza di `ListAdapter` inizializzata con il costruttore rispetti le aspettative di una lista vuota (size = 0 e isEmpty() = true).

Test Description

- Viene istanziata una nuova lista (`list = new ListAdapter();`).
- Si chiama `list.size()` e si confronta il valore restituito con 0.
- Si chiama `list.isEmpty()` e si verifica che restituisca `true`.

Pre-Conditions

- È stata richiamata l'annotazione `@Before`: la lista `list` è una nuova istanza di `ListAdapter`, vuota.

Post-Conditions

- Lo stato della lista rimane invariato (vuota, con size = 0).

Expected Results

- `list.size()` deve restituire 0.
- `list.isEmpty()` deve restituire `true`.

testAddAndGet

Summary

Verifica che il metodo `add(Object)` aggiunga correttamente due elementi e che `get(int)` restituisca i valori inseriti nell'ordine corretto.

Test Case Design

L'obiettivo è valutare:

- che `add("A")` e `add("B")` restituiscano `true`;
- che `size()` aumenti a 2 dopo due inserimenti;
- che `get(0)` restituisca `"A"` e `get(1)` restituisca `"B"`.

Test Description

1. Richiamare `list.add("A")`, verificare che restituisca `true`.
2. Richiamare `list.add("B")`, verificare che restituisca `true`.
3. Richiamare `list.size()` e confrontare con 2.
4. Richiamare `list.get(0)` e confrontare con `"A"`.
5. Richiamare `list.get(1)` e confrontare con `"B"`.

Pre-Conditions

- `list` è una nuova istanza vuota di `ListAdapter`.

Post-Conditions

- `list` contiene due elementi, in ordine: `"A"`, `"B"`.

Expected Results

- Entrambi gli `add(...)` restituiscono `true`.
- `size()` restituisce 2.
- `get(0)` restituisce `"A"`.
- `get(1)` restituisce `"B"`.

testGetNegativeIndex

Summary

Verifica che `get(-1)` su una lista vuota lanci un'`IndexOutOfBoundsException`.

Test Case Design

Obiettivo: assicurarsi che il metodo `get(int)` controlli correttamente l'indice e sollevi `IndexOutOfBoundsExcep`
quando l'indice è negativo.

Test Description

1. Senza inserire alcun elemento (lista vuota), si chiama `list.get(-1)`.
2. Si attende l'eccezione `IndexOutOfBoundsException`.

Pre-Conditions

- `list` è una nuova istanza vuota di `ListAdapter`.

Post-Conditions

- La lista rimane vuota e non viene modificata.

Expected Results

- Viene lanciata `IndexOutOfBoundsException`.

testGetIndexOutOfBounds

Summary

Verifica che `get(1)` su una lista di dimensione 1 sollevi un'`IndexOutOfBoundsException` se l'indice è maggiore o uguale a `size()`.

Test Case Design

Obiettivo: controllare che l'indice pari a `size()` (1) sia considerato fuori intervallo e sollevi l'eccezione.

Test Description

1. Inserire un elemento: `list.add("X")`.
2. Chiamare `list.get(1)`, ovvero indice pari a `size()`, e attendere l'eccezione.

Pre-Conditions

- `list` contiene un solo elemento (`size() == 1`).

Post-Conditions

- La lista rimane con un elemento, invariata.

Expected Results

- Viene lanciata `IndexOutOfBoundsException`.

testAddAtIndex

Summary

Verifica il funzionamento di `add(int, Object)` in posizione iniziale, centrale e finale, controllando che gli elementi si inseriscano nell'ordine corretto.

Test Case Design

Obiettivo: dimostrare che:

- Inserendo "A" in posizione 0 e "C" in posizione 1, si ottiene `["A","C"]`.
- Inserendo "B" in posizione 1, la lista diventa `["A","B","C"]`.
- Gli indici e i valori degli elementi risultano corretti.

Test Description

1. `list.add("A")` (indice 0).
2. `list.add("C")` (indice 1).
3. `list.add(1, "B")`: inserisce "B" tra "A" e "C".
4. Verificare `list.size()` uguale a 3.
5. Verificare `list.get(0) == "A"`, `list.get(1) == "B"`, `list.get(2) == "C"`.

Pre-Conditions

- `list` è inizialmente vuota.

Post-Conditions

- La lista contiene tre elementi in ordine: `"A","B","C"`.

Expected Results

- `size()` ritorna 3.
- `get(0) == "A"`, `get(1) == "B"`, `get(2) == "C"`.

testAddAtNegativeIndex

Summary

Verifica che `add(-1, "X")` sollevi un'`IndexOutOfBoundsException` quando l'indice è negativo.

Test Case Design

Obiettivo: controllare che un indice negativo sia considerato non valido da `add(int, Object)` e generi l'eccezione corrispondente.

Test Description

1. Chiamare direttamente `list.add(-1, "X")` su lista vuota.
2. Attendere l'eccezione `IndexOutOfBoundsException`.

Pre-Conditions

- `list` è una nuova istanza vuota di `ListAdapter`.

Post-Conditions

- La lista rimane vuota e non viene modificata.

Expected Results

- Viene lanciata `IndexOutOfBoundsException`.

testAddAtTooLargeIndex

Summary

Verifica che `add(2, "B")` su una lista di dimensione 1 sollevi un'`IndexOutOfBoundsException` se l'indice è maggiore di `size()`.

Test Case Design

Obiettivo: accertare che `add(int, Object)` consideri invalidi anche gli indici superiori a `size()` e generi l'eccezione corretta.

Test Description

1. Inserire un elemento: `list.add("A")` (ora `size() == 1`).
2. Chiamare `list.add(2, "B")`, dove $2 >$ `size()`.
3. Attendere l'eccezione `IndexOutOfBoundsException`.

Pre-Conditions

- `list` contiene un elemento (`size = 1`).

Post-Conditions

- La lista rimane con un solo elemento, invariata.

Expected Results

- Viene lanciata `IndexOutOfBoundsException`.

testRemoveByIndex

Summary

Verifica che `remove(int)` rimuova correttamente l'elemento all'indice specificato e restituisca il valore precedente.

Test Case Design

Obiettivo: confermare che:

- Con una lista di tre elementi `["A","B","C"]`, `remove(1)` restituisca `"B"`.
- Dopo la rimozione, la lista contenga `"A","C"` e `size()` sia 2.

Test Description

1. Inserire `"A"`, `"B"`, `"C"` nella lista.
2. Chiamare `Object removed = list.remove(1);` dove l'elemento in posizione 1 è `"B"`.
3. Verificare che `removed.equals("B")`.
4. Verificare che `list.size() == 2`.
5. Verificare `list.get(0) == "A"` e `list.get(1) == "C"`.

Pre-Conditions

- `list` contiene tre elementi, in ordine: `"A","B","C"`.

Post-Conditions

- La lista ora contiene due elementi: `"A","C"`.

Expected Results

- Il valore restituito da `remove(1)` è `"B"`.
- `size()` ritorna 2.
- `get(0) == "A"` e `get(1) == "C"`.

testRemoveIndexOutOfBounds

Summary

Verifica che `remove(1)` su una lista di dimensione 1 sollevi un'`IndexOutOfBoundsException` se l'indice non esiste.

Test Case Design

Obiettivo: assicurarsi che `remove(int)` controlli l'indice e generi un'eccezione quando l'indice richiesto non è valido.

Test Description

1. Inserire un elemento: `list.add("X")`.
2. Chiamare `list.remove(1)`, dove 1 >= `size()`, e attendere l'eccezione.

Pre-Conditions

- `list` contiene un solo elemento (`size = 1`).

Post-Conditions

- La lista rimane con un solo elemento, invariata.

Expected Results

- Viene lanciata `IndexOutOfBoundsException`.

testRemoveByObject

Summary

Verifica che `remove(Object)` rimuova la prima occorrenza dell'elemento e restituisca `true`, e che restituisca `false` se l'elemento non è presente.

Test Case Design

Obiettivi:

- Su lista `["A","B","C"]`, `remove("B")` restituisca `true` e rimuova "B".
- Successivamente, `remove("Z")` su elemento non presente restituisca `false`.

Test Description

1. Inserire `"A"`, `"B"`, `"C"`.
2. Chiamare `assertTrue(list.remove("B"))` e verificare che la lista contenga ora `"A","C"` con `size()==2`.
3. Chiamare `assertFalse(list.remove("Z"))` su elemento non presente e verificare che `size()==2`.

Pre-Conditions

- `list` contiene tre elementi: `"A","B","C"`.

Post-Conditions

- Dopo la prima rimozione, `list` contiene `"A","C"`.
- Dopo il secondo tentativo di rimozione, la lista rimane `"A","C"`.

Expected Results

- `remove("B")` restituisce `true`, rimuove "B".
- `remove("Z")` restituisce `false`, lista invariata.

testIndexOfAndLastIndexOf

Summary

Verifica i metodi `indexOf(Object)` e `lastIndexOf(Object)` in presenza di duplicati e per elemento non presente.

Test Case Design

Obiettivo:

- Su lista `["A","B","A","C"]`, `indexOf("A")` restituisce 0, `lastIndexOf("A")` restituisce 2.
- `indexOf("B")` e `lastIndexOf("B")` restituiscono 1.
- `indexOf("Z")` restituisce -1.

Test Description

1. Inserire `"A","B","A","C"`.
2. Verificare `indexOf("A") == 0` e `lastIndexOf("A") == 2`.
3. Verificare `indexOf("B") == 1` e `lastIndexOf("B") == 1`.
4. Verificare `indexOf("Z") == -1`.

Pre-Conditions

- `list` contiene quattro elementi: `"A","B","A","C"`.

Post-Conditions

- La lista rimane invariata (`"A","B","A","C"`).

Expected Results

- `indexOf("A")` = 0, `lastIndexOf("A")` = 2.
- `indexOf("B")` = 1, `lastIndexOf("B")` = 1.
- `indexOf("Z")` = -1.

testContainsAndContainsAll

Summary

Verifica i metodi `contains(Object)` e `containsAll(HCollection)` su elementi presenti e non presenti.

Test Case Design

Obiettivi:

- Con lista `["A","B","C"]`, `contains("B")` = `true`, `contains("Z")` = `false`.
- Con una seconda collezione `other = ["A","C"]`, `containsAll(other)` = `true`. Aggiungendo "Z" a `other`, `containsAll(other)` = `false`.

Test Description

1. Inserire `"A","B","C"` nella lista.
2. Verificare `assertTrue(list.contains("B"))` e `assertFalse(list.contains("Z"))`.
3. Creare `HCollection other = new ListAdapter(); other.add("A"); other.add("C");`
4. Verificare `assertTrue(list.containsAll(other))`.
5. Chiamare `other.add("Z")` e verificare `assertFalse(list.containsAll(other))`.

Pre-Conditions

- `list` contiene `"A","B","C"`.

Post-Conditions

- La lista rimane invariata (`"A","B","C"`).
- La collezione `other` viene modificata aggiungendo "Z".

Expected Results

- `list.contains("B")` = `true`.
- `list.contains("Z")` = `false`.
- `list.containsAll(["A","C"])` = `true`.
- `list.containsAll(["A","C","Z"])` = `false`.

testAddAllRemoveAllRetainAll

Summary

Verifica i metodi `addAll(HCollection)`, `removeAll(HCollection)` e `retainAll(HCollection)` in diversi scenari.

Test Case Design

- *addAll*: con `other = ["X","Y","Z"]`, su lista inizialmente vuota, confermare che tutti e tre gli elementi vengano aggiunti.
- *removeAll*: su lista contenente `["X","Y","Z","A","B"]`, rimuovere `["X","B"]` e verificare che rimangano `"Y","Z","A"`.
- *retainAll*: su lista contenente `["X","Y","Z","A"]`, mantenere solo `["Z","A"]` e verificare che la lista diventi `"Z","A"`.

Test Description

1. Creare `HCollection other = ["X","Y","Z"]`.
2. `assertTrue(list.addAll(other))`; verificare che `list.size() == 3` e gli elementi in posizioni 0,1,2 siano `"X","Y","Z"`.
3. Aggiungere a `list` anche gli elementi `"A","B"` con `list.add("A"); list.add("B");`.
4. Creare `HCollection toRemove = ["X","B"]`.
5. `assertTrue(list.removeAll(toRemove))`; verificare che `"X","B"` non siano più contenuti, ma `"Y","Z","A"` lo siano.
6. Ripristinare la lista a `["X","Y","Z","A"]` con `list.clear()`, quindi `list.add("X"); list.add("Y"); list.add("Z"); list.add("A");`.
7. Creare `HCollection toRetain = ["Z","A"]`.
8. `assertTrue(list.retainAll(toRetain))`; verificare che `list.size() == 2` e che contenga solo `"Z","A"`.

Pre-Conditions

- All'inizio del test, `list` è vuota.
- Successivamente, la lista viene riempita manualmente per ciascun sotto-test.

Post-Conditions

- Dopo `addAll`, la lista contiene `"X","Y","Z"`.
- Dopo `removeAll`, rimangono `"Y","Z","A"`.
- Dopo `retainAll`, rimangono solo `"Z","A"`.

Expected Results

- `addAll` aggiunge correttamente tutti gli elementi e restituisce `true`.
- `removeAll` rimuove `"X","B"` lasciando gli altri.
- `retainAll` mantiene solo gli elementi presenti in `toRetain`.

testClear

Summary

Verifica che il metodo `clear()` rimuova tutti gli elementi dalla lista.

Test Case Design

Obiettivo: confermare che, dopo aver inserito due elementi, `clear()` svuoti completamente la lista (`size() == 0` e `isEmpty() == true`).

Test Description

1. Inserire `"A"`, `"B"`.
2. Verificare che `size() == 2`.
3. Chiamare `list.clear()`.
4. Verificare che `size() == 0` e `isEmpty() == true`.

Pre-Conditions

- `list` contiene inizialmente due elementi.

Post-Conditions

- La lista è vuota (`size() == 0`, `isEmpty() == true`).

Expected Results

- `list.size()` restituisce 0.
- `list.isEmpty()` restituisce `true`.

testEqualsAndHashCode

Summary

Verifica che due liste con gli stessi elementi e nello stesso ordine siano uguali e abbiano lo stesso `hashCode()`, e che il confronto fallisca quando gli elementi differiscono.

Test Case Design

Obiettivi:

- Con due istanze di `ListAdapter` entrambe contenenti `"A","B"`, verificare `equals()` e `hashCode()`.
- Aggiungere un elemento a `list2` e verificare che `equals()` ritorni `false`.

Test Description

1. Creare due liste separate, `list1` e `list2`.
2. Inserire in entrambe `"A"` e `"B"`.
3. Verificare `assertTrue(list1.equals(list2))` e `assertEquals(list1.hashCode(), list2.hashCode(`
4. Eseguire `list2.add("C")`.
5. Verificare `assertFalse(list1.equals(list2))`.

Pre-Conditions

- `list1` e `list2` sono inizialmente vuote.

Post-Conditions

- Dopo la prima parte, le due liste sono identiche (`"A","B"`).
- Dopo l'aggiunta di "C" a `list2`, le due liste sono diverse.

Expected Results

- `list1.equals(list2)` = `true` e stessissimo `hashCode()`.
- Dopo `list2.add("C")`, `list1.equals(list2)` = `false`.

testToArray

Summary

Verifica che `toArray()` restituisca un array di tutti gli elementi nell'ordine corretto.

Test Case Design

Obiettivo: confermare che, con lista `["A","B","C"]`, il metodo `toArray()` restituisca un array esattamente uguale a `{"A","B","C"}`.

Test Description

1. Inserire `"A","B","C"`.
2. Chiamare `Object[] arr = list.toArray();`.
3. Verificare `arr.length == 3`.
4. Verificare `arr[0]=="A"`, `arr[1]=="B"`, `arr[2]=="C"`.

Pre-Conditions

- `list` contiene tre elementi: `"A","B","C"`.

Post-Conditions

- La lista rimane invariata (`"A","B","C"`).

Expected Results

- `toArray()` restituisce un array ordinato `{"A","B","C"}`.

testToArrayWithParameter

Summary

Verifica il metodo `toArray(Object[])` per due casi: array di dimensione maggiore o uguale e array di dimensione minore della lista.

Test Case Design

Obiettivi:

- Caso 1: array passato di dimensione più grande rispetto a `size()`: deve essere usato direttamente e l'elemento in eccesso deve diventare `null`.
- Caso 2: array passato di dimensione minore, deve essere creato e restituito un nuovo array di dimensione esatta.

Test Description

1. Inserire `"X","Y"`.
2. Creare `String[] arr1 = new String[3];`.
3. Chiamare `Object[] result1 = list.toArray(arr1);` e verificare:
   - `result1 == arr1` (stesso riferimento).
   - `arr1[0] == "X"`, `arr1[1] == "Y"`, `arr1[2] == null`.
4. Creare `String[] arr2 = new String[1];`.
5. Chiamare `Object[] result2 = list.toArray(arr2);` e verificare:
   - `result2 != arr2` (nuovo array restituito).
   - `result2[0] == "X"`, `result2[1] == "Y"`.

Pre-Conditions

- `list` contiene due elementi: `"X","Y"`.

Post-Conditions

- La lista rimane invariata (`"X","Y"`).
- L'array `arr1` è stato sovrascritto, l'array `arr2` rimane intatto.

Expected Results

- Per `arr1`, `result1 == arr1`, con `["X","Y",null]`.
- Per `arr2`, `result2` è un nuovo array `["X","Y"]`.

testIterator

Summary

Verifica i metodi base di `HIterator`: `hasNext()` e `next()` su una lista di tre elementi, assicurandosi che l'iterazione proceda correttamente fino alla fine.

Test Case Design

Obiettivo: confermare che l'iteratore restituisca in sequenza `"A","B","C"` e poi segnali la fine della lista con `hasNext() = false`.

Test Description

1. Inserire `"A","B","C"` in `list`.
2. Ottenere `HIterator it = list.iterator();`.
3. Verificare `assertTrue(it.hasNext())` e `assertEquals("A", it.next())`.
4. Verificare `assertTrue(it.hasNext())` e `assertEquals("B", it.next())`.
5. Verificare `assertTrue(it.hasNext())` e `assertEquals("C", it.next())`.
6. Verificare `assertFalse(it.hasNext())`.

Pre-Conditions

- `list` contiene tre elementi: `"A","B","C"`.

Post-Conditions

- L'iteratore è avanzato oltre l'ultimo elemento (`hasNext() == false`).
- La lista rimane invariata (`"A","B","C"`).

Expected Results

- `next()` restituisce `"A","B","C"` in sequenza.
- `hasNext()` restituisce `false` dopo l'ultimo elemento.

testIteratorNextNoSuchElement

Summary

Verifica che `next()` su iteratore avanzato oltre l'ultimo elemento sollevi `NoSuchElementException`.

Test Case Design

Obiettivo: assicurarsi che, dopo aver tratto l'unico elemento da una lista di dimensione 1, un secondo `next()` sollevi l'eccezione corretta.

Test Description

1. Inserire `"X"`.
2. Ottenere l'iteratore: `HIterator it = list.iterator();`.
3. Chiamare `it.next()` (ritorna `"X"`).
4. Chiamare nuovamente `it.next()` e attendere `NoSuchElementException`.

Pre-Conditions

- `list` contiene un solo elemento (`"X"`).

Post-Conditions

- L'iteratore è in uno stato non valido (avanzato oltre l'ultimo elemento).
- La lista rimane invariata (`"X"`).

Expected Results

- Il secondo `next()` solleva `java.util.NoSuchElementException`.

testIteratorRemoveBeforeNext

Summary

Verifica che chiamare `remove()` prima di `next()` su un iteratore sollevi `IllegalStateException`.

Test Case Design

Obiettivo: accertare che il metodo `remove()` controlli lo stato interno e generi un'eccezione se viene invocato senza un precedente `next()` valido.

Test Description

1. Inserire `"Y"`.
2. Ottenere l'iteratore: `HIterator it = list.iterator();`.
3. Chiamare `it.remove()` senza aver chiamato `next()`.
4. Attendere `IllegalStateException`.

Pre-Conditions

- `list` contiene un solo elemento (`"Y"`).
- L'iteratore non ha ancora chiamato `next()` (`lastRet == -1`).

Post-Conditions

- Non ci sono modifiche alla lista.
- L'iteratore rimane in uno stato non valido per la rimozione.

Expected Results

- Viene lanciata `IllegalStateException`.

testIteratorRemove

Summary

Verifica che `remove()` su iteratore, dopo un precedente `next()`, rimuova correttamente l'elemento e aggiorni la lista.

Test Case Design

Obiettivi:

- Con una lista iniziale `["1","2"]`, chiamare `next()` e poi `remove()` per rimuovere il primo elemento.
- Verificare che la lista risultante contenga solo `"2"`.

Test Description

1. Inserire `"1"`, `"2"`.
2. Ottenere l'iteratore: `HIterator it = list.iterator();`.
3. Chiamare `it.next()` per ottenere `"1"`.
4. Chiamare `it.remove()`, che deve rimuovere l'elemento `"1"`.
5. Verificare che `list.size() == 1` e che `list.get(0) == "2"`.

Pre-Conditions

- `list` contiene due elementi: `"1","2"`.

Post-Conditions

- La lista è stata modificata, rimane con un solo elemento `"2"`.

Expected Results

- Dopo `remove()`, `size()` ritorna 1.
- `get(0)` restituisce `"2"`.

testListIteratorForwardBackward

Summary

Verifica i metodi base di `HListIterator`, in particolare `hasNext()`, `next()`, `hasPrevious()`, `previous()`, per scorrere in avanti e poi indietro.

Test Case Design

Obiettivo: confermare che un `HListIterator`:

- Avanzi di due posizioni con due `next()` e restituisca correttamente i valori.
- Ritorni indietro con due `previous()` e restituisca i valori corretti.

Test Description

1. Inserire `"A","B","C"`.
2. Ottenere `HListIterator it = list.listIterator();`.
3. Verificare `assertTrue(it.hasNext())`.
4. Chiamare `assertEquals("A", it.next())`.
5. Chiamare `assertEquals("B", it.next())`.
6. Verificare `assertTrue(it.hasPrevious())`.
7. Chiamare `assertEquals("B", it.previous())`.
8. Chiamare `assertEquals("A", it.previous())`.
9. Verificare `assertFalse(it.hasPrevious())`.

Pre-Conditions

- `list` contiene tre elementi: `"A","B","C"`.
- L'iteratore è posizionato all'inizio (indice 0).

Post-Conditions

- L'iteratore è tornato all'inizio della lista (`hasPrevious() == false`).
- La lista rimane invariata (`"A","B","C"`).

Expected Results

- `next()` restituisce in sequenza `"A","B"`.
- `previous()` restituisce in sequenza `"B","A"`.
- `hasPrevious()` ritorna `false` quando il cursore è all'indice 0.

testListIteratorAddSetRemove

Summary

Verifica i metodi `add(Object)`, `set(Object)` e `remove()` di `HListIterator` in diverse fasi: inserimento da inizio lista, sostituzione di un elemento e rimozione.

Test Case Design

- *add*: con iteratore all'inizio di una lista vuota, `add("X")` deve inserire "X" in posizione 0.
- *set*: con iteratore avanzato dopo aver recuperato l'unico elemento, `set("Y")` deve sostituire "X" con "Y".
- *remove*: dopo un `next()` e un `set()`, `remove()` deve cancellare l'elemento corrente, lasciando la lista vuota.

Test Description

1. Ottenere `HListIterator it = list.listIterator();`.
2. Chiamare `it.add("X")`.
3. Verificare `assertEquals(1, list.size())` e `assertEquals("X", list.get(0))`.
4. Riposizionarsi: `it = list.listIterator();`.
5. Verificare `assertEquals("X", it.next())`.
6. Chiamare `it.set("Y")`.
7. Verificare `assertEquals("Y", list.get(0))`.
8. Chiamare `it.remove()`.
9. Verificare `assertTrue(list.isEmpty())`.

Pre-Conditions

- `list` è inizialmente vuota.

Post-Conditions

- Dopo `add("X")`, la lista contiene `"X"`.
- Dopo `set("Y")`, la lista contiene `"Y"`.
- Dopo `remove()`, la lista è vuota.

Expected Results

- `add("X")` inserisce correttamente l'elemento e `size()` diventa 1.
- `set("Y")` sostituisce "X" con "Y".
- `remove()` rimuove correttamente l'elemento corrente, portando la lista a dimensione 0.

testListIteratorSetBeforeNext

Summary

Verifica che chiamare `set(Object)` su un `HListIterator` prima di aver invocato `next()` o `previous()` sollevi un'`IllegalStateException`.

Test Case Design

Obiettivo: garantire che lo stato interno dell'iteratore tenga traccia di quando `set()` è consentito (solo dopo `next()` o `previous()`).

Test Description

1. Inserire `"A"`.
2. Ottenere `HListIterator it = list.listIterator();`.
3. Chiamare direttamente `it.set("B")`, senza aver precedentemente chiamato `next()`.
4. Attendere `IllegalStateException`.

Pre-Conditions

- `list` contiene un elemento (`"A"`).
- L'iteratore non ha ancora avanzato (`lastRet == -1`).

Post-Conditions

- La lista rimane invariata (`"A"`).
- L'iteratore rimane in uno stato non valido per `set()`.

Expected Results

- Viene lanciata `IllegalStateException`.

testListIteratorNextTooFar

Summary

Verifica che un secondo `next()` su un iteratore avanzato oltre l'ultimo elemento sollevi `NoSuchElementExceptio`

Test Case Design

Obiettivo: confermare che `next()` controlli la posizione e sollevi eccezione quando si cerca di avanzare troppo.

Test Description

1. Inserire `"A"`.
2. Ottenere `HListIterator it = list.listIterator();`.
3. Chiamare `it.next()` (ritorna `"A"`).
4. Chiamare di nuovo `it.next()` e attendere `NoSuchElementException`.

Pre-Conditions

- `list` contiene un solo elemento (`"A"`).
- L'iteratore è all'inizio (indice 0).

Post-Conditions

- L'iteratore è avanzato oltre il singolo elemento.
- La lista rimane invariata (`"A"`).

Expected Results

- Il secondo `next()` solleva `java.util.NoSuchElementException`.

testSubListView

Summary

Verifica che `subList(int, int)` restituisca una vista live: mutazioni sulla sub-list si riflettono nella lista originale e viceversa.

Test Case Design

Obiettivi:

- Creare una lista `["A","B","C","D"]`.
- Ottenere la sub-list `sub = list.subList(1,3)` che corrisponde a `["B","C"]`.
- Rimuovere l'elemento in posizione 0 di `sub`, verificando che "B" venga tolto anche dalla lista originale.
- Modificare un elemento con `list.set(2, "Z")` e verificare che la lista originale cambi, mentre non serve verificare la sub-list perché l'indice 2 è fuori dal suo intervallo.

Test Description

1. Inserire `"A","B","C","D"` nella lista.
2. Ottenere `HList sub = list.subList(1,3);` che contiene `"B","C"`.
3. Verificare `sub.size() == 2`, `sub.get(0) == "B"`, `sub.get(1) == "C"`.
4. Chiamare `Object removed = sub.remove(0);` (rimuove "B").
5. Verificare che `removed.equals("B")`, `sub.size() == 1`, `sub.get(0) == "C"`.
6. Verificare che `list.size() == 3`, `list.get(1) == "C"` e `list.get(2) == "D"`.
7. Chiamare `list.set(2, "Z")` (sostituisce "D" con "Z").
8. Verificare che `list.get(2) == "Z"`.

Pre-Conditions

- `list` contiene quattro elementi: `"A","B","C","D"`.
- `sub` è una sub-list live sugli indici `1` e `2`.

Post-Conditions

- Dopo `sub.remove(0)`, la lista originale è `"A","C","D"`.
- Dopo `list.set(2,"Z")`, la lista originale è `"A","C","Z"`.
- La sub-list è `"C"` dopo la rimozione.

Expected Results

- `sub.remove(0)` restituisce "B" e rimuove "B" da entrambe le liste.
- `list.set(2,"Z")` modifica la lista originale correttamente.

testAddAllAtIndex

Summary

Verifica che `addAll(int, HCollection)` inserisca correttamente una collezione in una posizione specificata.

Test Case Design

Obiettivo: confermare che, con lista iniziale `["A","D"]` e collezione `["B","C"]` su indice 1, la lista diventi `["A","B","C","D"]`.

Test Description

1. Inserire `"A","D"` in `list`.
2. Creare `HCollection toAdd = ["B","C"]`.
3. Chiamare `assertTrue(list.addAll(1, toAdd));`.
4. Verificare `list.size() == 4`.
5. Verificare `list.get(0)=="A"`, `list.get(1)=="B"`, `list.get(2)=="C"`, `list.get(3)=="D"`.

Pre-Conditions

- `list` contiene due elementi: `"A","D"`.
- `toAdd` contiene due elementi: `"B","C"`.

Post-Conditions

- La lista è `"A","B","C","D"`.

Expected Results

- `addAll(1, toAdd)` restituisce `true`.
- La lista assume l'ordine corretto: `"A","B","C","D"`.

testAddAllAtInvalidIndex

Summary

Verifica che `addAll(int, HCollection)` su indice non valido (maggiore di `size()`) sollevi `IndexOutOfBoundsException`.

Test Case Design

Obiettivo: assicurarsi che chiamare `addAll(1, c)` su una lista vuota (`size() == 0`) sollevi l'eccezione appropriata.

Test Description

1. Creare `HCollection c = ["X"]`.
2. Chiamare `list.addAll(1, c);` su lista vuota.
3. Attendere `IndexOutOfBoundsException`.

Pre-Conditions

- `list` è vuota (`size() == 0`).
- `c` contiene almeno un elemento (`"X"`).

Post-Conditions

- La lista rimane vuota.

Expected Results

- Viene lanciata `IndexOutOfBoundsException`.

Fine documentazione di `ListAdapterTest`.