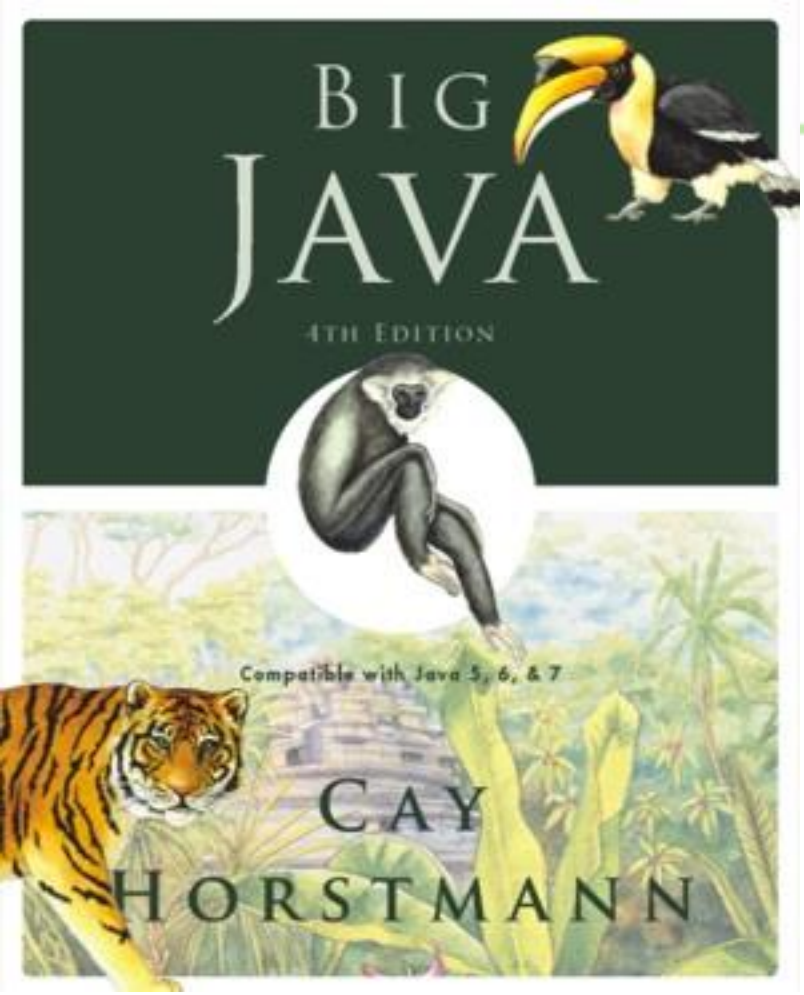


الفصل ١٥ - مقدمة لهياكل البيانات

Chapter 15 – An Introduction to Data Structures

Chapter Goals

- To learn how to use the linked lists provided in the standard library
- To be able to use iterators to traverse linked lists
- To understand the implementation of linked lists
- To distinguish between abstract and concrete data types
- **To know the efficiency of fundamental operations of lists and arrays**
- To become familiar with the stack and queue types

- لمعرفة كيفية استخدام القوائم المرتبطة المنصوص عليها في المكتبة القياسية
- لتكون قادرة على استخدام التكرار الصورة لاجتياز القوائم المرتبطة
- لفهم تنفيذ القوائم المرتبطة
- للتمييز بين أنواع البيانات المجردة وملموسة
- لمعرفة كفاءة العمليات الأساسية للقوائم والمصفوفات
- لتصبح مألوفة مع مكدس وطابور أنواع

Using Linked Lists

باستخدام قوائم مرتبط

- A linked list consists of a number of nodes, each of which has a reference to the next node
- Adding and removing elements in the middle of a linked list is efficient
- Visiting the elements of a linked list in sequential order is efficient
- Random access is not efficient

- وتتكون قائمة مرتبطة بعدد من العقد، كل منها يحتوي على إشارة إلى العقدة التالية
- إضافة وإزالة عناصر في منتصف قائمة مرتبطة غير فعالة
- زيارة عناصر من قائمة مرتبطة في ترتيب تسلسلي غير فعال
- الوصول العشوائي ليست فعالة

Inserting an Element into a Linked List

إدراج عنصر في قائمة مرتبطة

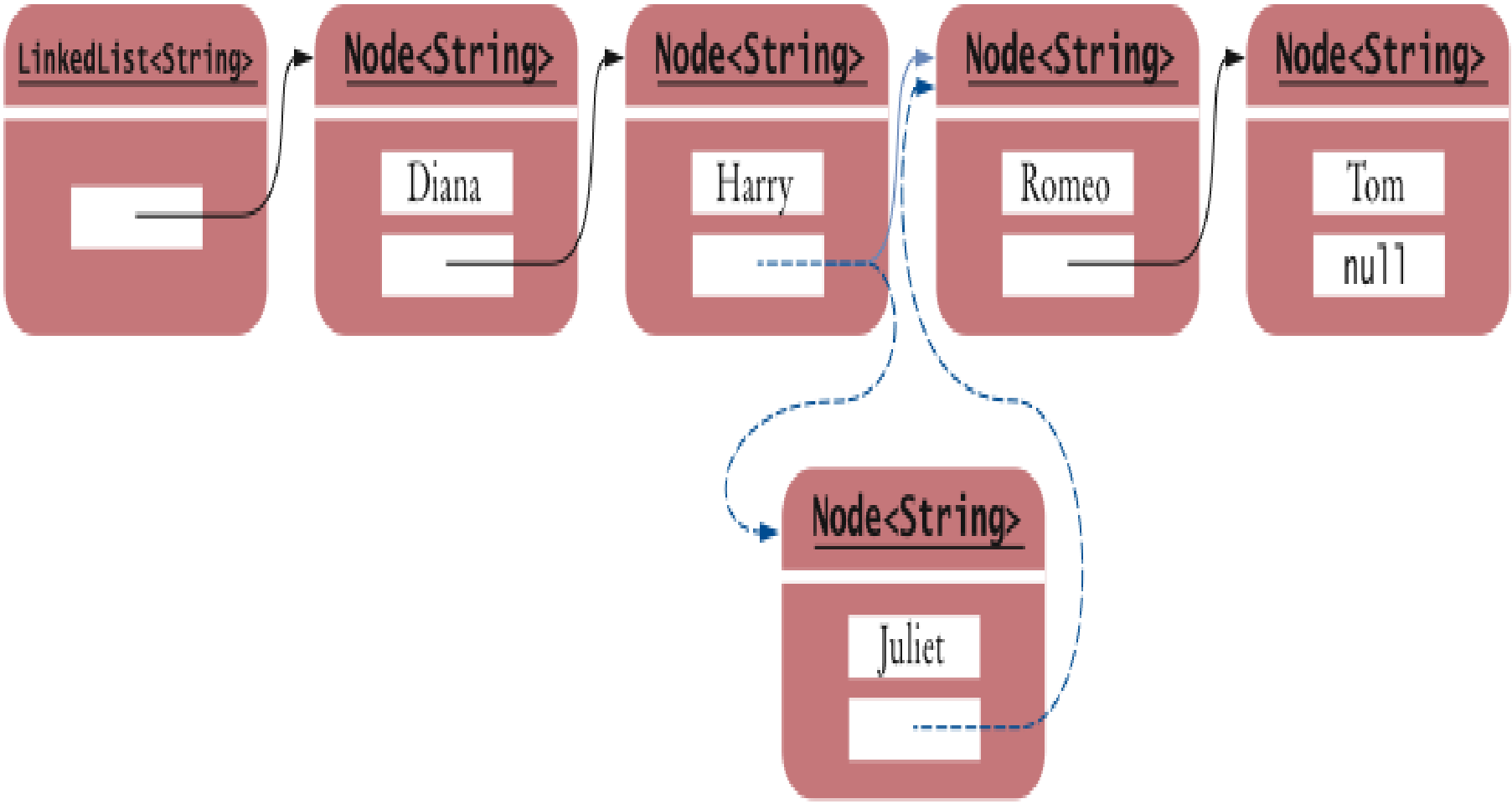


Figure 1 Inserting an Element into a Linked List

Java's `LinkedList` class

- فئة عامة
- تحديد نوع العناصر في أقواس زاوية

- Generic class
  - *Specify type of elements in angle brackets:* `LinkedList<Product>`
- Package: `java.util`

Table 1 LinkedList Methods

`LinkedList<String> lst = new LinkedList<String>();`	An empty list.
`lst.addLast("Harry")`	Adds an element to the end of the list. Same as `add`.
`lst.addFirst("Sally")`	Adds an element to the beginning of the list. `lst` is now `[Sally, Harry]`.
`lst.getFirst()`	Gets the element stored at the beginning of the list; here `"Sally"`.
`lst.getLast()`	Gets the element stored at the end of the list; here `"Harry"`.
`String removed = lst.removeFirst();`	Removes the first element of the list and returns it. `removed` is `"Sally"` and `lst` is `[Harry]`. Use `removeLast` to remove the last element.
`ListIterator<String> iter = lst.listIterator()`	Provides an iterator for visiting all list elements (see Table 2 on page 634).

List Iterator

- `ListIterator` type
- Gives access to elements inside a linked list
- Encapsulates a position anywhere inside the linked list
- Protects the linked list while giving access

- يتيح الوصول إلى العناصر داخل قائمة مرتبطة
- تلخص موقف في أي مكان داخل قائمة مرتبطة
- يحمي القائمة المرتبطة في حين يعطي الوصول

A List Iterator

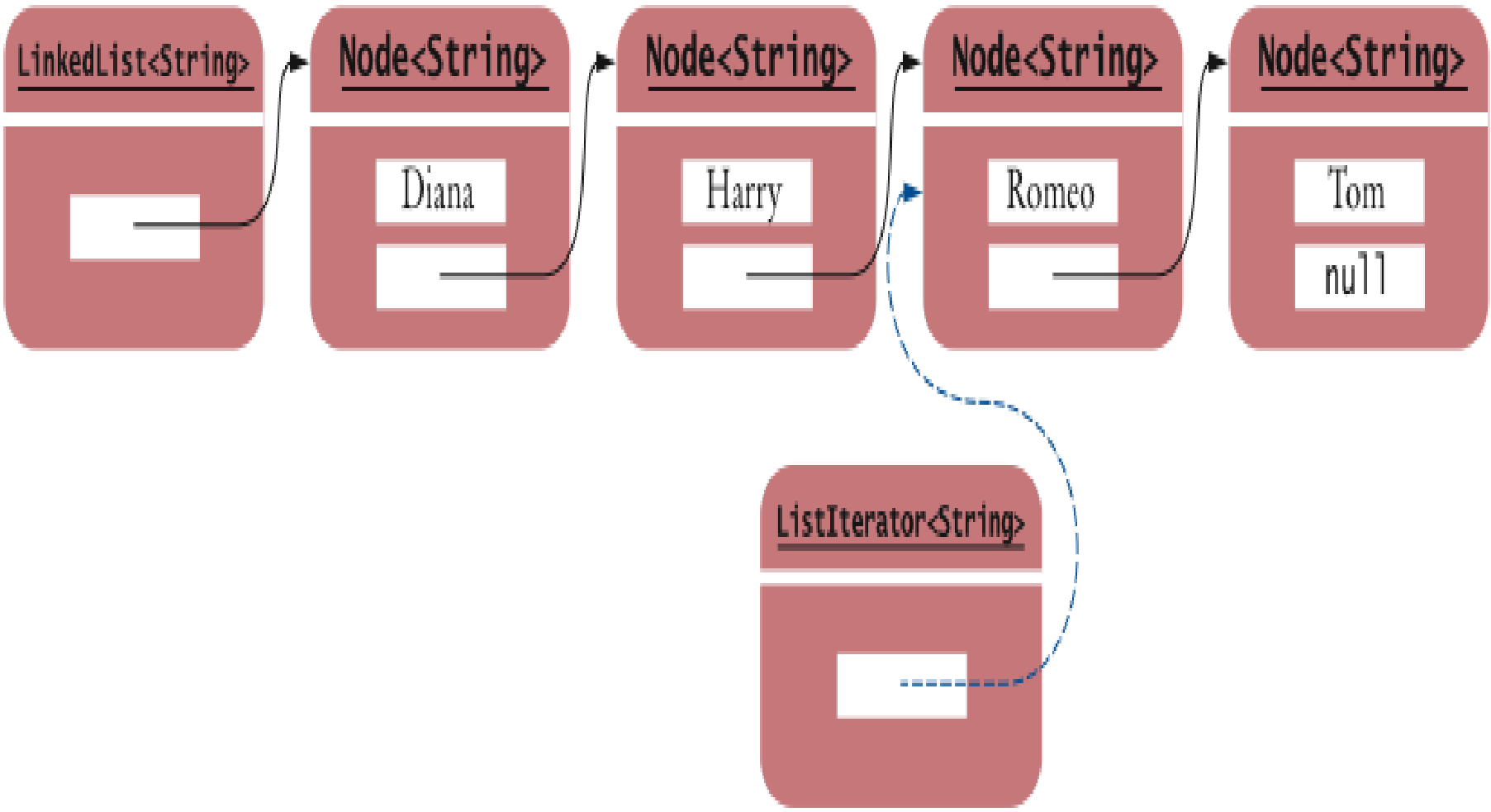


Figure 2 A List Iterator

A Conceptual View of the List Iterator

وجهة نظر المفاهيمي من قائمة ومكرر

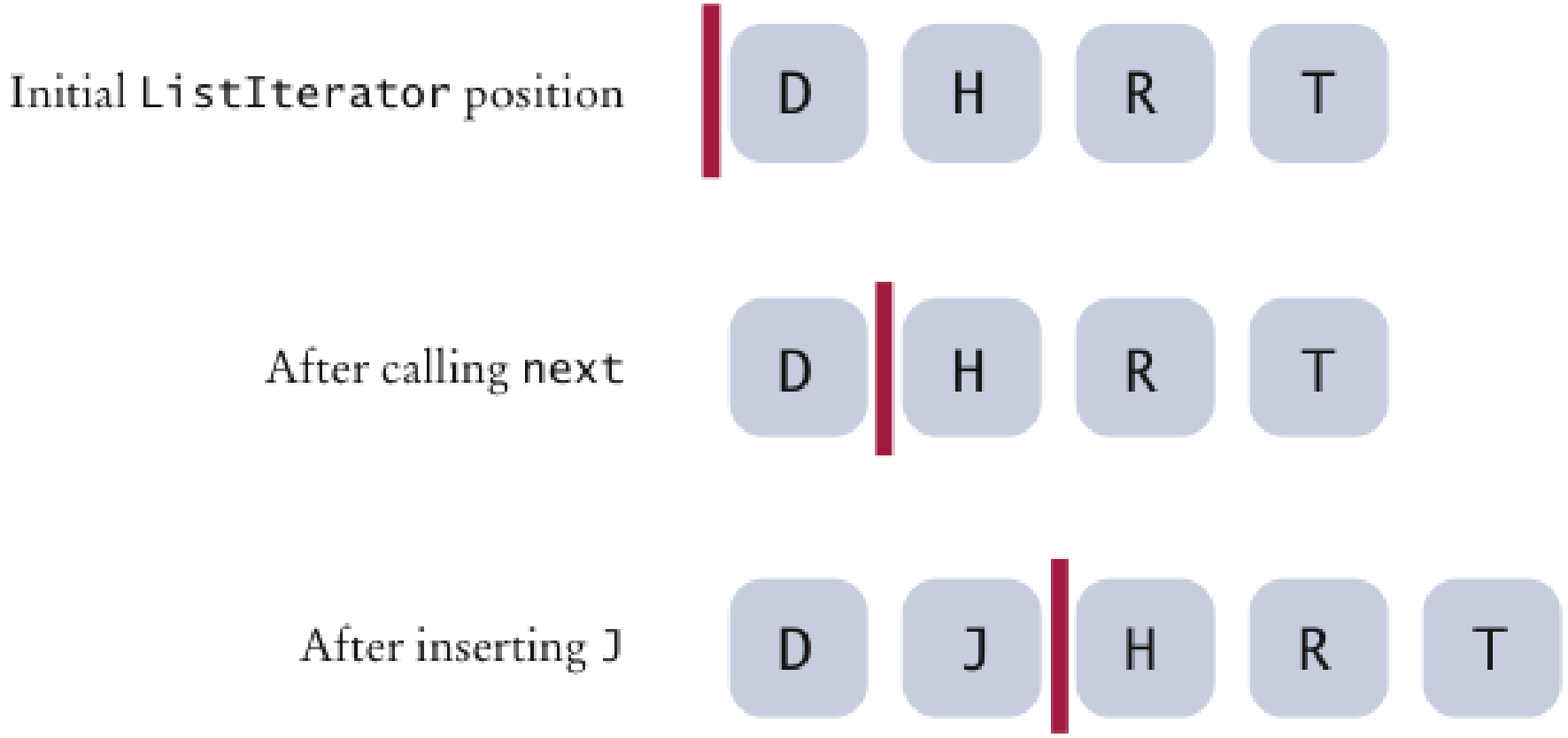
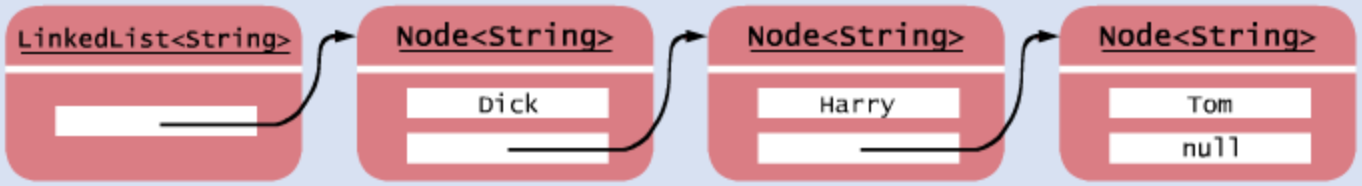


Figure 3 A Conceptual View of the List Iterator

Animation 15.1 — List Iterators



This animation demonstrates list iterators.

List Iterator

قائمة مكرر

- Think of an iterator as pointing between two elements
  - *Analogy: Like the cursor in a word processor points between two characters*
- The `listIterator` method of the `LinkedList` class gets a list iterator

```
LinkedList<String> employeeNames = ...;
ListIterator<String> iterator =
   employeeNames.listIterator();
```

- التفكير في التكرار كما لافتا بين عنصرين
- التشبيه: مثل المؤشر في نقاط معالج النصوص بين حرفين
- طريقة listIteratorالطبقة LinkedListيحصل على قائمة التكرار

List Iterator

- Initially, the iterator points before the first element
- The `next` method moves the iterator:

```
iterator.next();
```

- `next` throws a `NoSuchElementException` if you are already past the end of the list
- `hasNext` returns true if there is a next element:

```
if (iterator.hasNext())
   iterator.next();
```

- في البداية، ونقطة التكرار قبل العنصر الأول
- الطريقة التالية يتحرك التكرار: `iterator.next();`
- يلقي المقبل على NoSuchElementExceptionإذا كنت بالفعل بعد نهاية قائمة
- hasNextيعود الحقيقية إذا كان هناك العنصر التالي:

```
if (iterator.hasNext())
   iterator.next();
```

List Iterator

- The `next` method returns the element that the iterator is passing:

- الطريقة التالية إرجاع العنصر أن التكرار يمر:

```
while iterator.hasNext()
{
   String name = iterator.next();
   Do something with name
}
```

- Shorthand:

- الاختزال:

```
for (String name : employeeNames)
{
   Do something with name
}
```

- وراء الكواليس، وللحلقة يستخدم التكرار لزيارة جميع عناصر القائمة

Behind the scenes, the `for` loop uses an iterator to visit all list elements

List Iterator

- `LinkedList` is a *doubly linked list*
  - *Class stores two links:*
    - *One to the next element, and*
    - *One to the previous element*
- To move the list position backwards, use:
  - *`hasPrevious`*
  - *`previous`*

- LinkedListهي قائمة مرتبطة مضاعف
- مخازن درجتين الروابط:
- واحد إلى العنصر التالي، و
- واحد إلى العنصر السابق
- لنقل موقف القائمة إلى الوراء، استخدام:
  - *`hasPrevious`*
  - *`previous`*

Adding and Removing from a `LinkedList`

- The `add` method:
  - *Adds an object after the iterator*
  - *Moves the iterator position past the new element:*

```
iterator.add("Juliet");
```

LinkedListإضافة وإزالة من

- طريقة إضافة:
- يضيف كائن بعد التكرار
- يتحرك موقف التكرار الماضي العنصر الجديد:

```
iterator.add("Juliet");
```

Adding and Removing from a `LinkedList`

- The `remove` method
  - *Removes and*
  - *Returns the object that was returned by the last call to `next` or `previous`*

```
//Remove all names that fulfill a certain condition
while (iterator.hasNext())
{
   String name = iterator.next();
   if (name fulfills condition)
      iterator.remove();

}
```

- طريقة إزالة
- ويزيل
- يعود الكائن الذي تم إرجاعها بواسطة المكالمة الأخيرة إلى التالي أو السابق `next` or `previous`

Adding and Removing from a `LinkedList`

- Be careful when calling `remove`:
  - *It can be called only once after calling `next` or `previous`:*

```
iterator.next();
iterator.next();
iterator.remove();
iterator.remove();
// Error: You cannot call remove twice.
```

- كن حذرا عند استدعاء إزالة:
- يمكن أن يطلق عليه مرة واحدة فقط بعد استدعاء التالي أو السابق:

  - *You cannot call it immediately after a call to `add`:*

```
iter.add("Fred");
iter.remove(); // Error: Can only call remove after
               // calling next or previous
```

- لا يمكن أن نطلق عليه على الفور بعد دعوة لإضافة:

  - *If you call it improperly, it throws an `IllegalStateException`*

- إذا كنت اسميها غير صحيح، فإنه يطرح IllegalStateException

Methods of the `ListIterator` Interface

Table 2 Methods of the `ListIterator` Interface

`String s = iter.next();`	Assume that `iter` points to the beginning of the list `[Sally]` before calling `next`. After the call, `s` is `"Sally"` and the iterator points to the end.
`iter.hasNext()`	Returns `false` because the iterator is at the end of the collection.

`if (iter.hasPrevious())`
`{`
`  s = iter.previous();`
`}`
	`hasPrevious` returns `true` because the iterator is not at the beginning of the list.

`iter.add("Diana");`	Adds an element before the iterator position. The list is now `[Diana, Sally]`.
`iter.next();` `iter.remove();`	`remove` removes the last element returned by `next` or `previous`. The list is again `[Diana]`.

Sample Program

برنامج عينة

- `ListTester` is a sample program that
  - *Inserts strings into a list*
  - *Iterates through the list, adding and removing elements*
  - *Prints the list*

- ListTesterهو برنامج عينة
- إدراج السلاسل إلى قائمة
- بالتكرار عبر قائمة، إضافة وإزالة عناصر
- يطبع قائمة

ch15/uselist/ListTester.java

```
 1  import java.util.LinkedList;
 2  import java.util.ListIterator;
 3
 4  /**
 5     A program that tests the LinkedList class
 6  */
 7  public class ListTester
 8  {
 9     public static void main(String[] args)
10     {
11        LinkedList<String> staff = new LinkedList<String>();
12        staff.addLast("Diana");
13        staff.addLast("Harry");
14        staff.addLast("Romeo");
15        staff.addLast("Tom");
16
17        // | in the comments indicates the iterator position
18
19        ListIterator<String> iterator = staff.listIterator(); // |DHRT
20        iterator.next(); // D|HRT
21        iterator.next(); // DH|RT
22
```

Continued

ch15/uselist/ListTester.java (cont.)

```
23          // Add more elements after second element
24
25          iterator.add("Juliet"); // DHJ|RT
26          iterator.add("Nina"); // DHJN|RT
27
28          iterator.next(); // DHJNR|T
29
30          // Remove last traversed element
31
32          iterator.remove(); // DHJN|T
33
34          // Print all elements
35
36          for (String name : staff)
37             System.out.print(name + " ");
38          System.out.println();
39          System.out.println("Expected: Diana Harry Juliet Nina
Tom");
40       }
41    }
```

Continued

ch15/uselist/ListTester.java (cont.)

Program Run:

```
Diana Harry Juliet Nina Tom
Expected: Diana Harry Juliet Nina Tom
```

Self Check 15.1

Do linked lists take more storage space than arrays of the same size?

Answer: Yes, for two reasons. You need to store the node references, and each node is a separate object. (There is a fixed overhead to store each object in the virtual machine.)

- هل القوائم المرتبطة تأخذ مساحة تخزين أكبر من صفائف من نفس الحجم؟
- الجواب: نعم، وذلك لسببين. تحتاج إلى تخزين المراجع عقدة، والعقدة هي كل كائن منفصل.
(هناك أحمال ثابت لتخزين كل كائن في الجهاز الظاهري.)

Self Check 15.2

Why don’t we need iterators with arrays?

Answer: An integer index can be used to access any array location.

- لماذا لا نحتاج التكرار الصورة مع المصفوفات؟
- الإجابة: مؤشر عدد صحيح يمكن استخدامها للوصول إلى أي مكان مجموعة.

Implementing Linked Lists

تنفيذ قوائم مرتبط

- Previous section: Java's `LinkedList` class
- Now, we will look at the implementation of a simplified version of this class
- It will show you how the list operations manipulate the links as the list is modified
- To keep it simple, we will implement a singly linked list
  - *Class will supply direct access only to the first list element, not the last one*
- Our list will not use a type parameter
  - *Store raw `Object` values and insert casts when retrieving them*

- المقطع السابق: فئة LinkedListجاوة
- الآن، ونحن سوف ننظر في تنفيذ نسخة مبسطة من هذه الفئة
- وسوف تظهر لك كيف أن عمليات التلاعب قائمة الارتباطات كما تم تعديل لائحة
- أن يبقيه بسيط، سننفذ قائمة مرتبطة منفردة
- والطبقة تزويد الوصول المباشر فقط إلى أول عنصر قائمة، وليس آخر واحد
- قائمتنا لن تستخدم معلمة نوع
- مخزن القيم كائن الخام وإدراج يلقي عند استرجاع لهم

Implementing Linked Lists

- `Node`: Stores an object and a reference to the next node
- Methods of linked list class and iterator class have frequent access to the `Node` instance variables
- To make it easier to use:
  - *We do not make the instance variables private*
  - *We make `Node` a private inner class of `LinkedList`*
  - *It is safe to leave the instance variables public*
    - *None of the list methods returns a `Node` object*

- العقدة: يخزن كائن وإشارة إلى العقدة التالية
- طرق ربط الدرجة القائمة والطبقة التكرار لها بشكل متكرر؟ الوصول إلى المتغيرات المثال عقدة
- لجعله أسهل للاستخدام:
- نحن لا تجعل المتغيرات المثال الخاص
- نحن جعل عقدة فئة الداخلية خاصة من LinkedList
- أنها آمنة لمغادرة المتغيرات المثال الجمهور
- أيا من الأساليب القائمة بإرجاع كائن عقدة

Implementing Linked Lists

```
public class LinkedList
{
   ...
   private class Node
   {
      public Object data;
      public Node next;
   }
}
```

Implementing Linked Lists

- `LinkedList` class
  - *Holds a reference `first` to the first node*
  - *Has a method to get the first element*

- فئة LinkedList
- حاصل على الإشارة أولا إلى العقدة الأولى
- لديه طريقة للحصول على العنصر الأول

Implementing Linked Lists

```
public class LinkedList
{
   private Node first;
   ...
   public LinkedList()
   {
      first = null;
   }
   public Object getFirst()
   {
      if (first == null)
         throw new NoSuchElementException();
      return first.data;
   }
}
```

Adding a New First Element

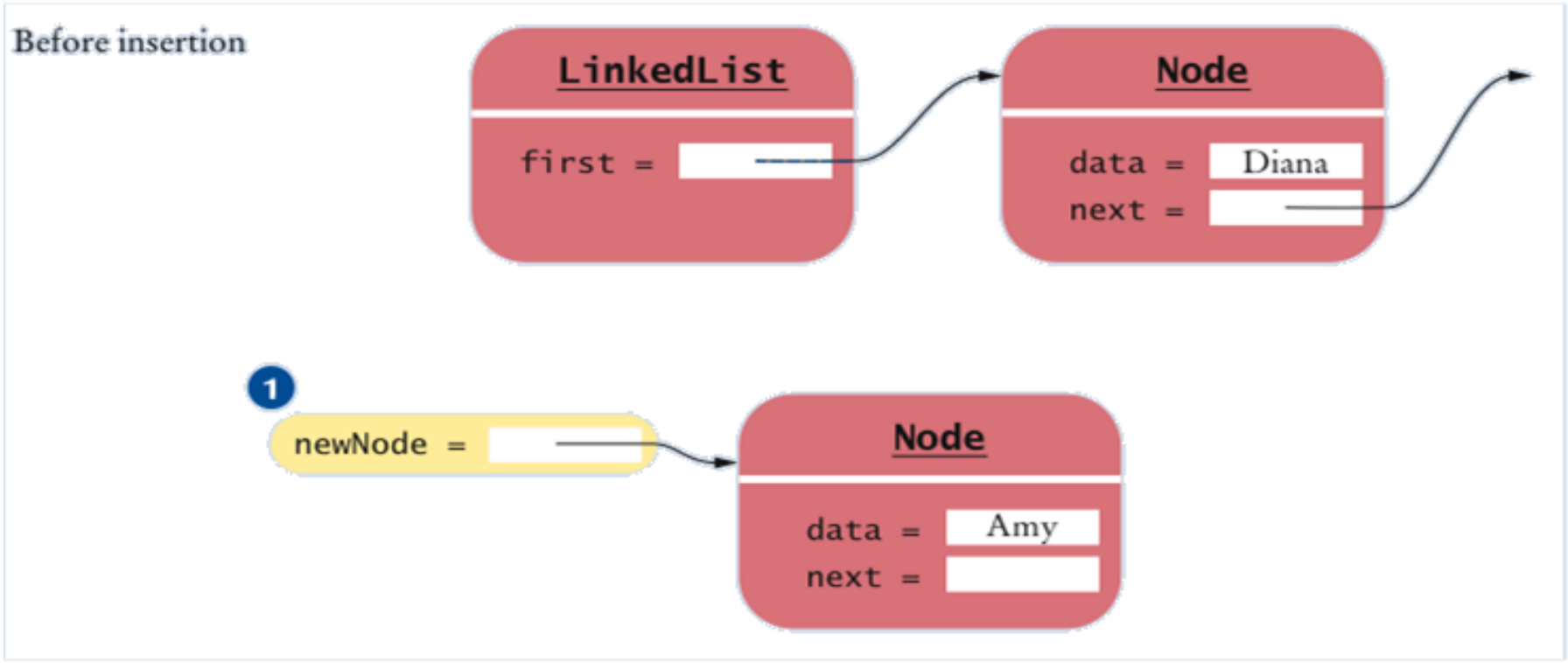
إضافة عنصر جديد أولا

- When a new node is added to the list
  - *It becomes the head of the list*
  - *The old list head becomes its next node*

- عند إضافة عقدة جديدة إلى قائمة
- يصبح رئيس القائمة
- يصبح رأس قائمة القديم عقدة القادمة

Adding a New First Element

```
public void addFirst(Object obj)
{
   Node newNode = new Node(); 1
   newNode.data = obj;
   newNode.next = first;
   first = newNode;
}
```



Adding a New First Element

```
public void addFirst(Object obj)
{
   Node newNode = new Node();
   newNode.data = obj;
   newNode.next = first; 2
   first = newNode; 3
}
```

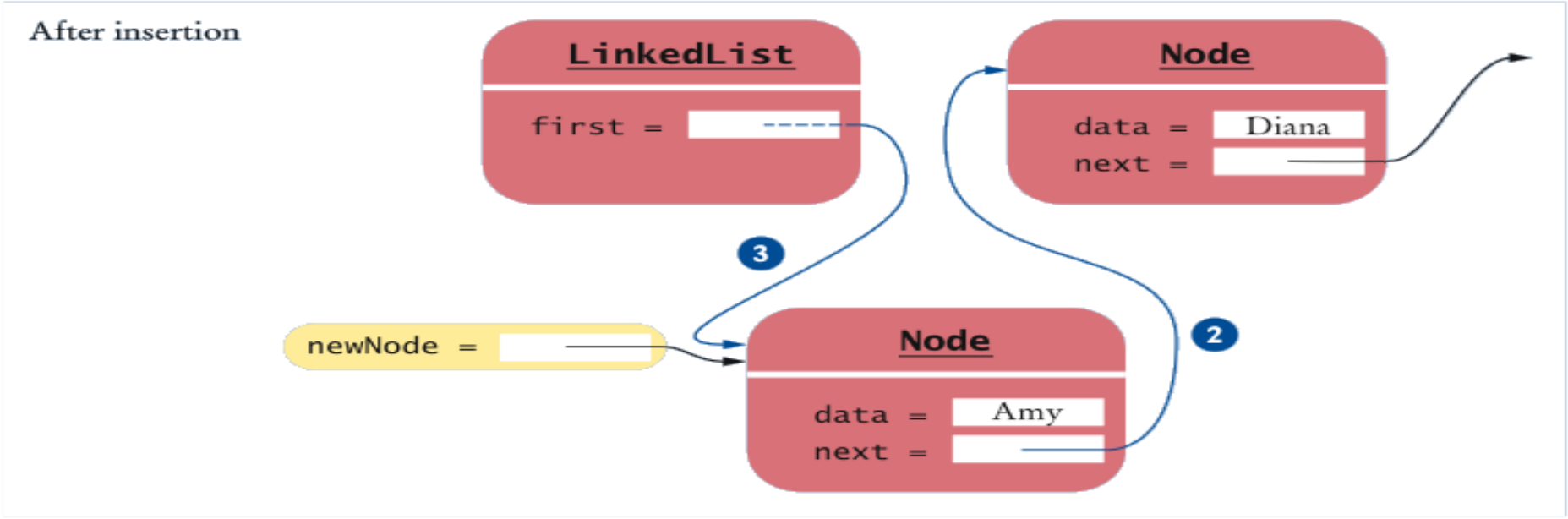


Figure 4 Adding a Node to the Head of a Linked List

Removing the First Element

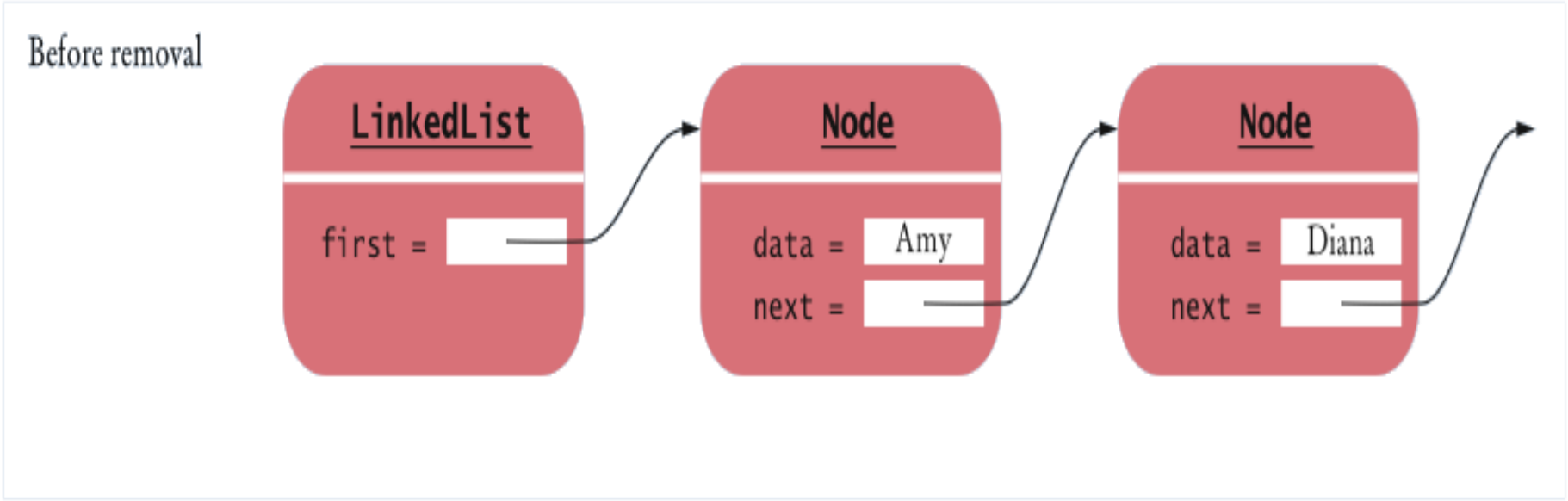
إزالة العنصر الأول

- When the first element is removed
  - *The data of the first node are saved and later returned as the method result*
  - *The successor of the first node becomes the first node of the shorter list*
  - *The old node will be garbage collected when there are no further references to it*

- عند إزالة العنصر الأول
- يتم حفظ البيانات من العقدة الأولى وعاد في وقت لاحق نتيجة طريقة
- خليفة العقدة الأولى يصبح العقدة الأولى من قائمة قصيرة
- سيتم العقدة القديمة التي تم جمعها من القمامة عندما لا يكون هناك إشارات أخرى لذلك

Removing the First Element

```
public Object removeFirst()
{
   if (first == null)
      throw new NoSuchElementException();
   Object obj = first.data;
   first = first.next;
   return obj;
}
```



Removing the First Element

```
public Object removeFirst()
{
   if (first == null)
      throw new NoSuchElementException();
   Object obj = first.data;
   first = first.next; 1
   return obj;
}
```

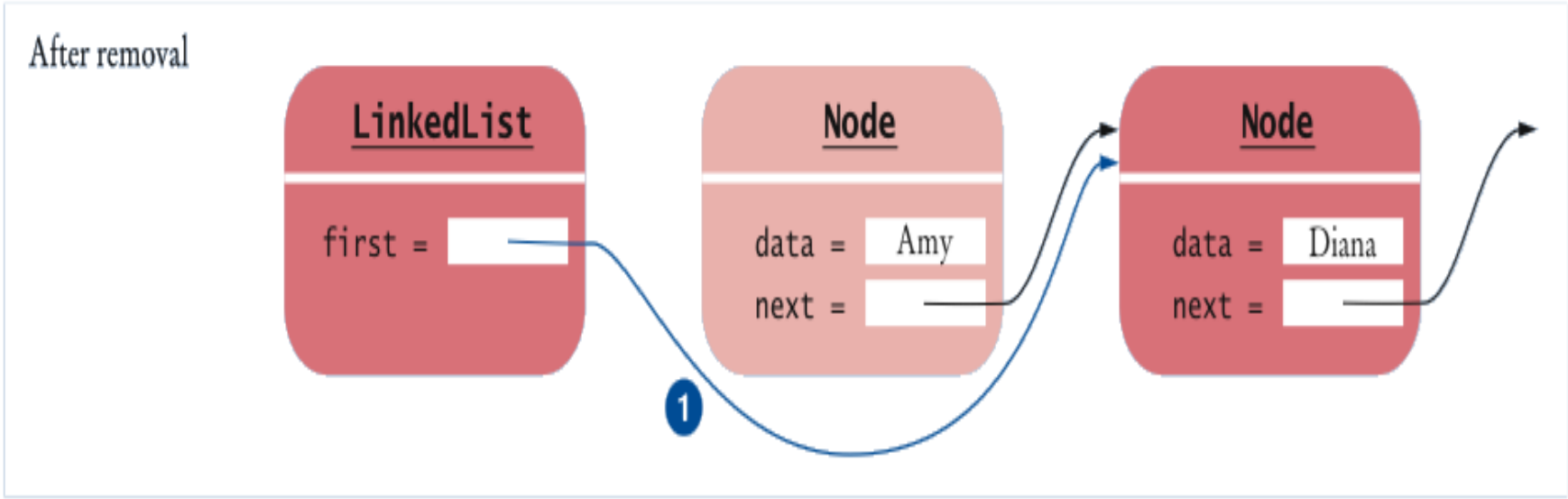


Figure 5 Removing the First Node from a Linked List

Linked List Iterator

- We define `LinkedListIterator`: private inner class of `LinkedList`
- Implements a simplified `ListIterator` interface
- Has access to the `first` field and private `Node` class
- Clients of `LinkedList` don't actually know the name of the iterator class
  - *They only know it is a class that implements the `ListIterator` interface*

- نحدد :LinkedListIteratorفئة الداخلية خاصة من LinkedList
- تنفذ واجهة مبسطة ListIterator
- لديه حق الوصول إلى الحقل الأول والطبقة عقدة خاصة
- عملاء LinkedListلا يعرفون في الواقع اسم الطبقة التكرار
- انهم يعرفون فقط هو الفئة التي تطبق الواجهة ListIterator

LinkedListIterator

- The `LinkListIterator` class:

```
public class LinkedList
{
   ...
   public ListIterator listIterator()
   {
      return new LinkedListIterator();
   }

   private class LinkedListIterator implements
      ListIterator
   {
      private Node position;
      private Node previous;

       ...
```

Continued

`LinkedListIterator` (cont.)

```
        public LinkedListIterator()
        {
            position = null;
            previous = null;
        }
    }
    ...
}
```

The Linked List Iterator's `next` Method

- `position`: Reference to the last visited node
- Also, store a reference to the last reference before that
- `next` method: `position` reference is advanced to `position.next`
- Old position is remembered in `previous`
- If the iterator points before the first element of the list, then the old `position` is `null` and `position` must be set to `first`

- نحدد :LinkedListIteratorفئة الداخلية خاصة من LinkedList
- تنفذ واجهة مبسطة ListIterator
- لديه حق الوصول إلى الحقل الأول والطبقة عقدة خاصة
- عملاء LinkedListلا يعرفون في الواقع اسم الطبقة التكرار
- انهم يعرفون فقط هو الفئة التي تطبق الواجهة ListIterator

The Linked List Iterator's `next` Method

ويرتبط أسلوب قائمة مكرر القادمة

```
public Object next()
{
   if (!hasNext())
      throw new NoSuchElementException();
   previous = position; // Remember for remove
   if (position == null)
      position = first;
   else position = position.next;
   return position.data;
}
```

The Linked List Iterator's `hasNext` Method

وترتبط hasNext
الطريقة قائمة مكرر ل

- The `next` method should only be called when the iterator is not at the end of the list
- The iterator is at the end
  - *if the list is empty* (`first == null`)
  - *if there is no element after the current position* (`position.next == null`)

- الطريقة التالية يجب فقط أن يسمى عند التكرار ليست في نهاية القائمة
- والتكرار في نهاية
- إذا كانت القائمة فارغة (أولا == فارغة)
- إذا كان هناك أي عنصر بعد الوضع الحالي (== position.nextفارغة)

The Linked List Iterator's `hasNext` Method

```
public boolean hasNext()
{
   if (position == null)
      return first != null;
   else
      return position.next != null;
}
```

The Linked List Iterator's `remove` Method

- If the element to be removed is the first element, call `removeFirst`
- Otherwise, the node preceding the element to be removed needs to have its `next` reference updated to skip the removed element
- If the `previous` reference equals `position`:
  - *This call does not immediately follow a call to `next`*
  - *Throw an `IllegalArgumentException`*
- It is illegal to call `remove` twice in a row
  - *`remove` sets the `previous` reference to `position`*

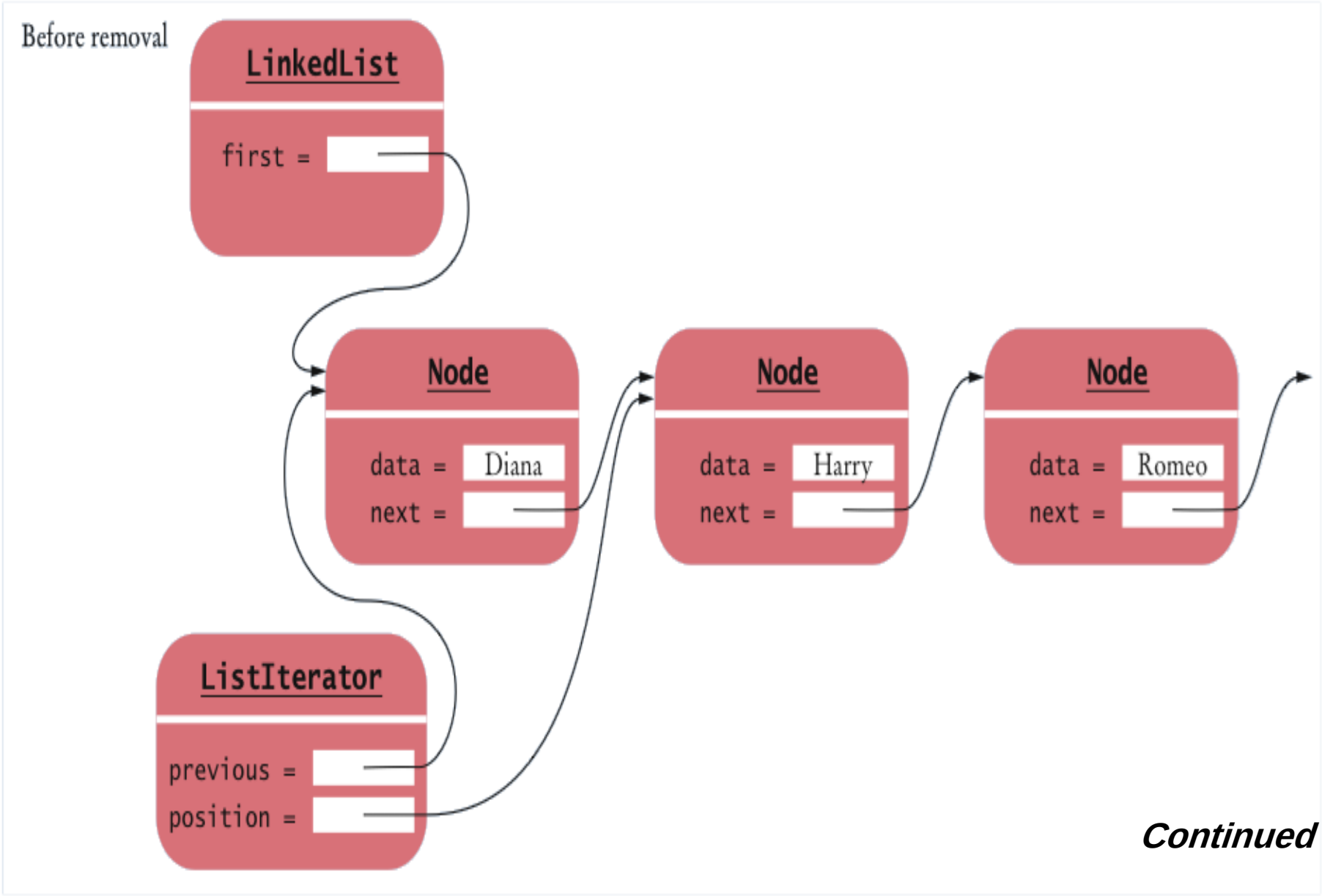
- **وترتبط طريقة حذف قائمة مكرر ل**
- إذا كان العنصر المراد إزالتها هو العنصر الأول، استدعاء removeFirst
- خلاف ذلك، العقدة السابقة العنصر المراد إزالتها احتياجات أن يكون مرجعها المقبلة تجديد لتخطي العنصر إزالة
- إذا كان المرجع السابق يساوي الموقف:
- هذه الدعوة لا يتبع مباشرة مكالمة بجوار
- رمي IllegalArgumentException
- أنه من غير القانوني لاستدعاء إزالة مرتين على التوالي
- إزالة مجموعات الإشارة السابقة إلى وضع

The Linked List Iterator's `remove` Method

```
public void remove()
{
   if (previous == position)
      throw new IllegalStateException();
   if (position == first)
   {
      removeFirst();
   }
   else
   {
      previous.next = position.next;
   }
   position = previous;
}
```

Continued

The Linked List Iterator's `remove` Method (cont.)



Continued

The Linked List Iterator's `remove` Method (cont.)

```
public void remove()
{
   If (previous == position)
      throw new IllegalStateException();
   if (position == first)
   {
      removeFirst();
   }
   else
   {
      previous.next = position.next; (1)
   }
   position = previous; (2)
}
```

Continued

The Linked List Iterator's `remove` Method (cont.)

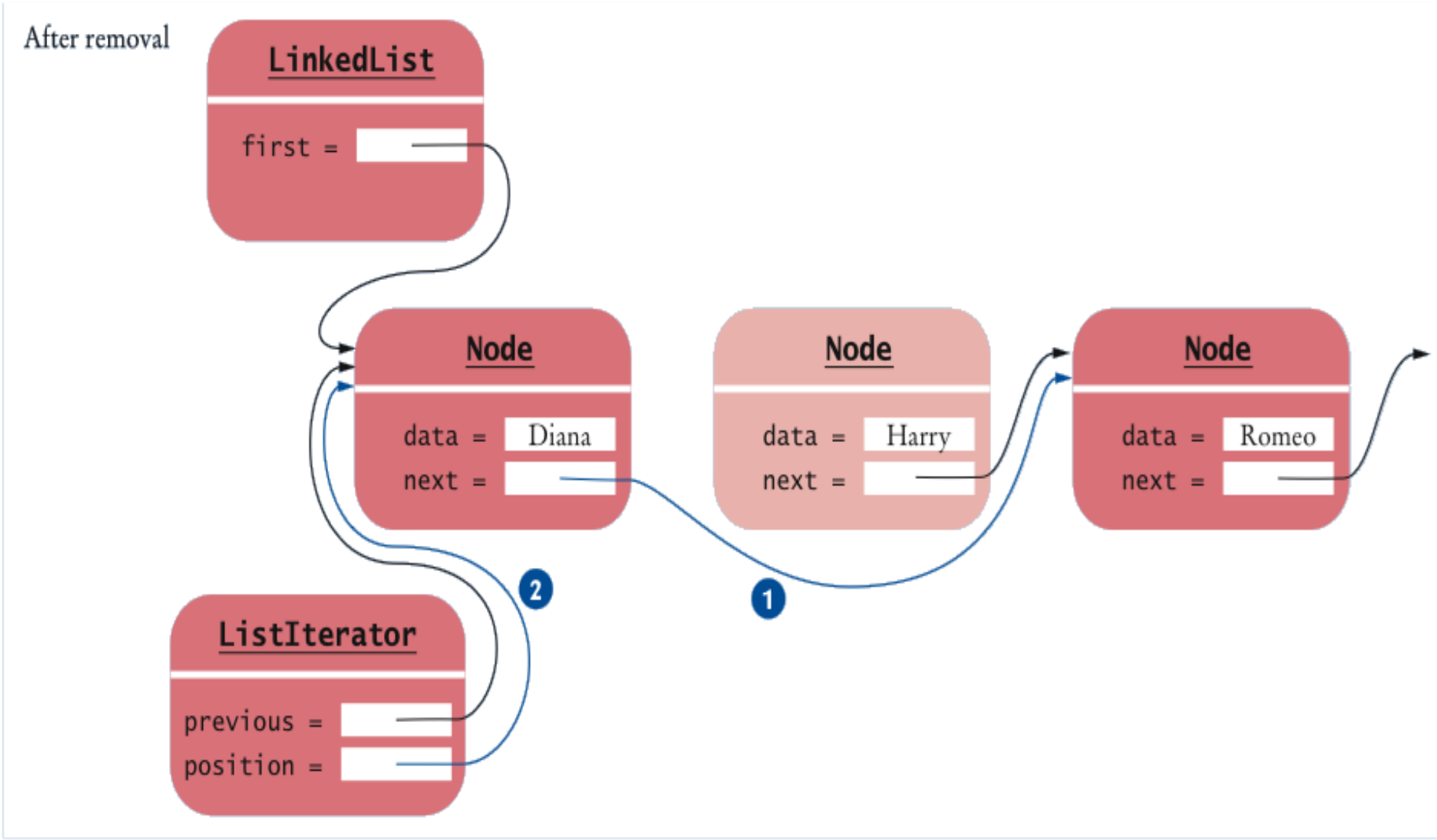


Figure 6 Removing a Node from the Middle of a Linked List

The Linked List Iterator's `set` Method

ويرتبط أسلوب مجموعة قائمة مكرر ل

- Changes the data stored in the previously visited element
- The `set` method

- تغيير البيانات المخزنة في العنصر زار سابقا
- طريقة مجموعة

```
public void set(Object obj)
{
   if (position == null)
      throw new NoSuchElementException();
    position.data = obj;
}
```

The Linked List Iterator's `add` Method

ويرتبط أسلوب إضافة قائمة مكرر ل

- The most complex operation is the addition of a node
- `add` inserts the new node after the current position
- Sets the successor of the new node to the successor of the current position

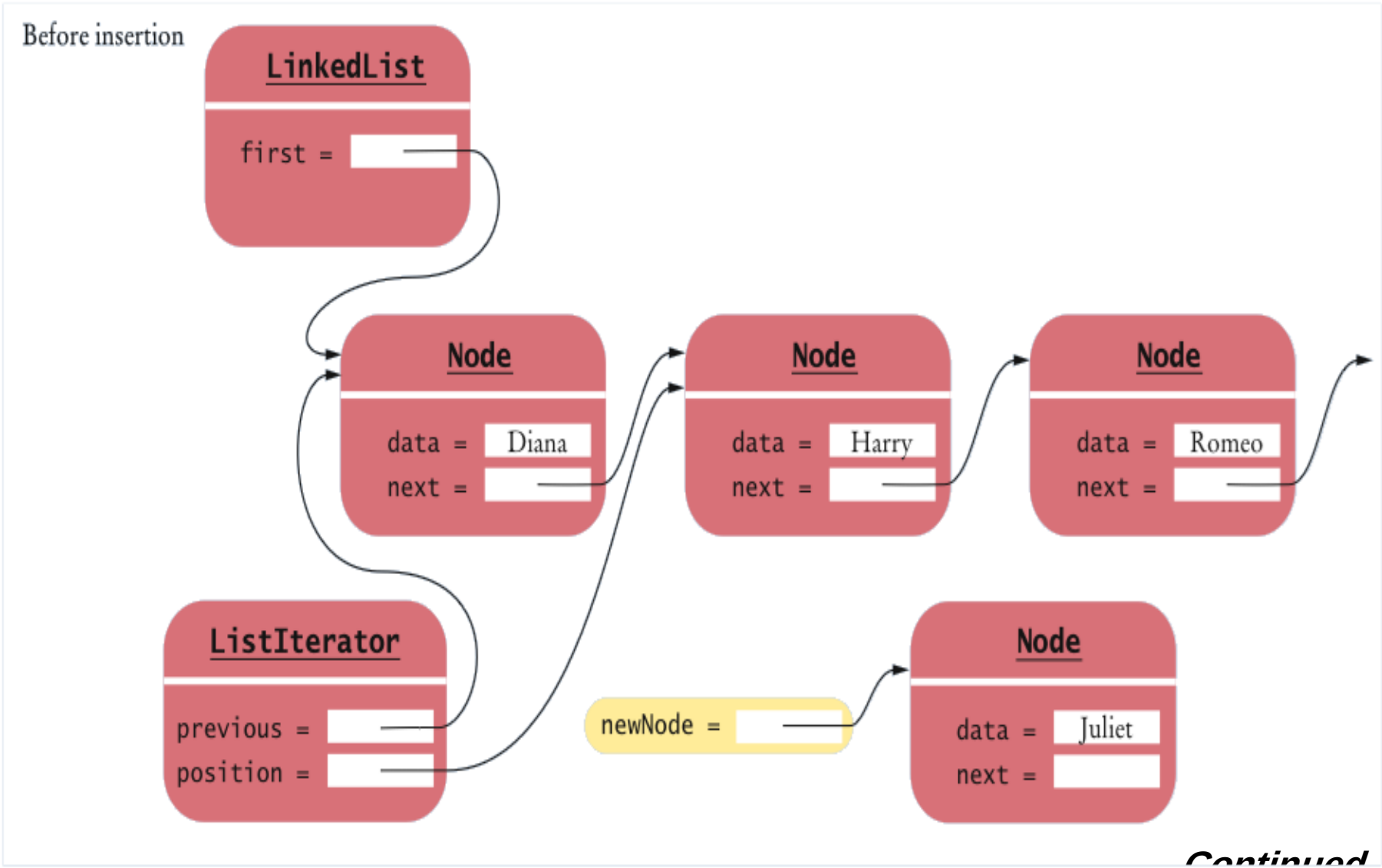
- العملية الأكثر تعقيدا هو إضافة عقدة
- إضافة إدراج عقدة جديدة بعد الوضع الحالي
- يحدد خليفة عقدة جديدة إلى الخلف ل؟ الوضع الحالي

The Linked List Iterator's `add` Method

```
public void add(Object obj)
{
   if (position == null)
   {
      addFirst(obj);
      position = first;
   }
   else
   {
      Node newNode = new Node();
      newNode.data = obj;
      newNode.next = position.next;
      position.next = newNode;
      position = newNode;
   }
   previous = position;
}
```

Continued

The Linked List Iterator's `add` Method (cont.)



Continued

The Linked List Iterator's `add` Method (cont.)

```
public void add(Object obj)
{
   if (position == null)
   {
      addFirst(obj);
      position = first;
   }
   else
   {
      Node newNode = new Node();
      newNode.data = obj;
      newNode.next = position.next;  (1)
      position.next = newNode;  (2)
      position = newNode;  (3)
   }
   previous = position;  (4)
}
```

Continued

The Linked List Iterator's `add` Method (cont.)

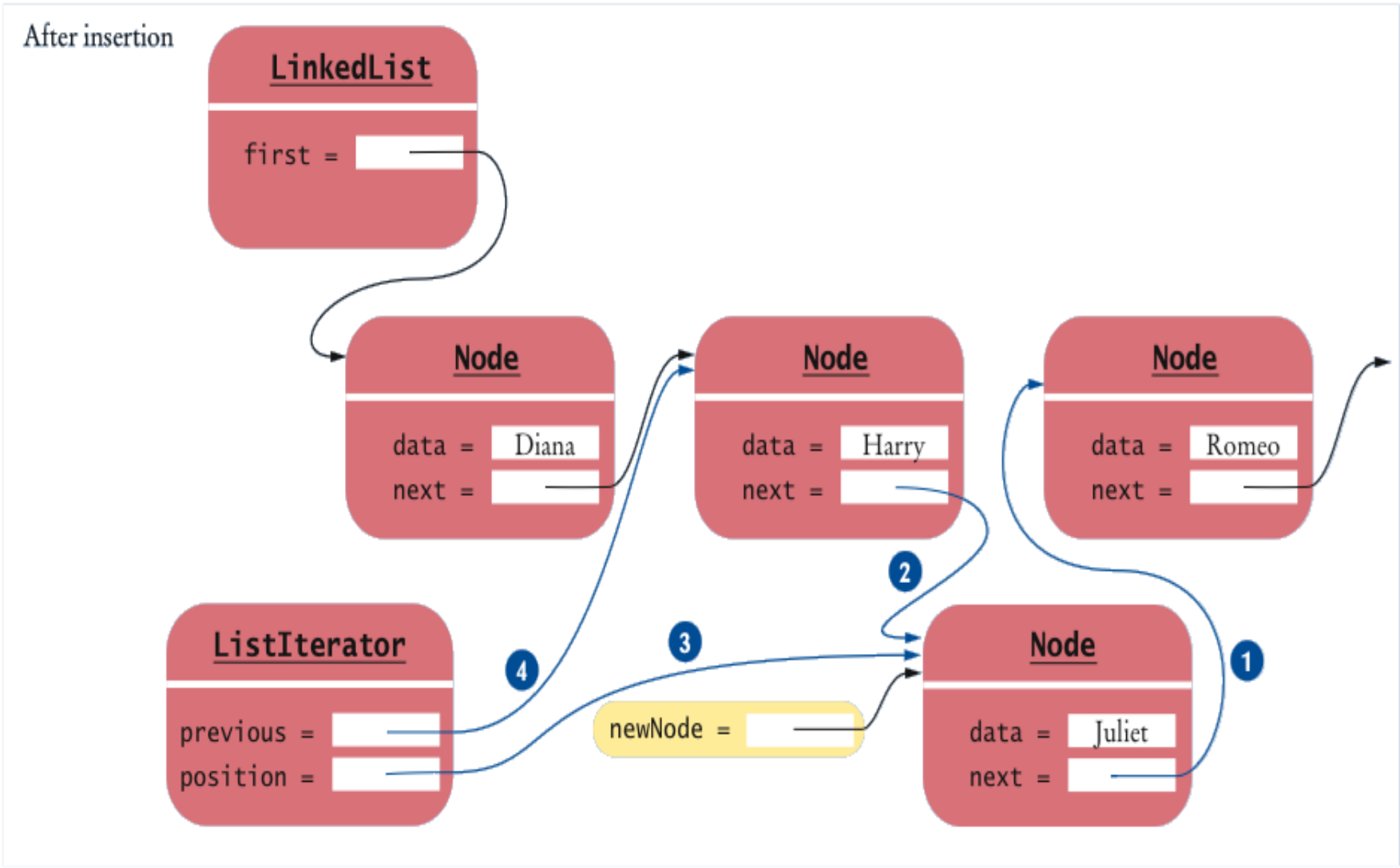


Figure 7 Adding a Node to the Middle of a Linked List

ch15/impllist/LinkedList.java

```
1  import java.util.NoSuchElementException;
2
3  /**
4     A linked list is a sequence of nodes with efficient
5     element insertion and removal. This class
6     contains a subset of the methods of the standard
7     java.util.LinkedList class.
8  */
9  public class LinkedList
10 {
11    private Node first;
12
13    /**
14       Constructs an empty linked list.
15    */
16    public LinkedList()
17    {
18       first = null;
19    }
20
```

Continued

ch15/impllist/LinkedList.java (cont.)

```
21      /**
22         Returns the first element in the linked list.
23         @return the first element in the linked list
24      */
25      public Object getFirst()
26      {
27         if (first == null)
28            throw new NoSuchElementException();
29         return first.data;
30      }
31
32      /**
33         Removes the first element in the linked list.
34         @return the removed element
35      */
36      public Object removeFirst()
37      {
38         if (first == null)
39            throw new NoSuchElementException();
40         Object element = first.data;
41         first = first.next;
42         return element;
43      }
44
```

Continued

ch15/impllist/LinkedList.java (cont.)

```
45      /**
46         Adds an element to the front of the linked list.
47         @param element the element to add
48      */
49      public void addFirst(Object element)
50      {
51         Node newNode = new Node();
52         newNode.data = element;
53         newNode.next = first;
54         first = newNode;
55      }
56
57      /**
58         Returns an iterator for iterating through this list.
59         @return an iterator for iterating through this list
60      */
61      public ListIterator listIterator()
62      {
63         return new LinkedListIterator();
64      }
65
```

Continued

ch15/impllist/LinkedList.java (cont.)

```
66      class Node
67      {
68         public Object data;
69         public Node next;
70      }
71
72      class LinkedListIterator implements ListIterator
73      {
74         private Node position;
75         private Node previous;
76
77         /**
78            Constructs an iterator that points to the front
79            of the linked list.
80         */
81         public LinkedListIterator()
82         {
83            position = null;
84            previous = null;
85         }
86
```

Continued

ch15/impllist/LinkedList.java (cont.)

```
87          /**
88             Moves the iterator past the next element.
89             @return the traversed element
90          */
91          public Object next()
92          {
93             if (!hasNext())
94                throw new NoSuchElementException();
95             previous = position; // Remember for remove
96
97             if (position == null)
98                position = first;
99             else
100               position = position.next;
101
102            return position.data;
103         }
104
```

Continued

ch15/impllist/LinkedList.java (cont.)

```
105          /**
106             Tests if there is an element after the iterator position.
107             @return true if there is an element after the iterator position
108          */
109          public boolean hasNext()
110          {
111             if (position == null)
112                return first != null;
113             else
114                return position.next != null;
115          }
116
```

Continued

ch15/impllist/LinkedList.java (cont.)

```
117         /**
118            Adds an element before the iterator position
119            and moves the iterator past the inserted element.
120            @param element the element to add
121         */
122         public void add(Object element)
123         {
124            if (position == null)
125            {
126               addFirst(element);
127               position = first;
128            }
129            else
130            {
131               Node newNode = new Node();
132               newNode.data = element;
133               newNode.next = position.next;
134               position.next = newNode;
135               position = newNode;
136            }
137            previous = position;
138         }
139
```

Continued

ch15/impllist/LinkedList.java (cont.)

```
140          /**
141             Removes the last traversed element. This method may
142             only be called after a call to the next() method.
143          */
144          public void remove()
145          {
146             if (previous == position)
147                throw new IllegalStateException();
148
149             if (position == first)
150             {
151                removeFirst();
152             }
153             else
154             {
155                previous.next = position.next;
156             }
157             position = previous;
158          }
159
```

Continued

ch15/impllist/LinkedList.java (cont.)

```
160          /**
161             Sets the last traversed element to a different value.
162             @param element the element to set
163          */
164          public void set(Object element)
165          {
166             if (position == null)
167                throw new NoSuchElementException();
168             position.data = element;
169          }
170       }
171    }
```

ch15/impllist/ListIterator.java

```
1  /**
2     A list iterator allows access of a position in a linked list.
3     This interface contains a subset of the methods of the
4     standard java.util.ListIterator interface. The methods for
5     backward traversal are not included.
6  */
7  public interface ListIterator
8  {
9     /**
10       Moves the iterator past the next element.
11       @return the traversed element
12    */
13    Object next();
14
15    /**
16       Tests if there is an element after the iterator position.
17       @return true if there is an element after the iterator position
18    */
19    boolean hasNext();
20
```

Continued

ch15/impllist/ListIterator.java (cont.)

```
21      /**
22         Adds an element before the iterator position
23         and moves the iterator past the inserted element.
24         @param element the element to add
25      */
26      void add(Object element);
27
28      /**
29         Removes the last traversed element. This method may
30         only be called after a call to the next() method.
31      */
32      void remove();
33
34      /**
35         Sets the last traversed element to a different value.
36         @param element the element to set
37      */
38      void set(Object element);
39   }
```

Self Check 15.3

Trace through the `addFirst` method when adding an element to an empty list.

Answer: When the list is empty, `first` is `null`. A new `Node` is allocated. It's `data` instance variable is set to the newly inserted object. It's `next` instance variable is set to `null` because `first` is `null`. The `first` instance variable is set to the new node. The result is a linked list of length 1.

- تتبع خلال طريقة addFirst عند إضافة عنصر إلى قائمة فارغة.
- الإجابة: عندما القائمة فارغة، هي الأولى لاغية. يتم تخصيص عقدة جديدة. تم تعيينه في متغير المثال البيانات إلى الكائن المدرج حديثا. انها تم تعيين المقبل متغير المثال لاغية لأن الأول هو باطل. تم تعيين متغير المقام الأول إلى عقدة جديدة. والنتيجة هي قائمة مرتبطة بطول ١.

Self Check 15.4

Conceptually, an iterator points between elements (see Figure 3). Does the position reference point to the element to the left or to the element to the right? Why does the `add` method have two separate cases?

Answer: It points to the element to the left. You can see that by tracing out the first call to `next`. It leaves position to point to the first node.

- من الناحية النظرية، وهي نقطة التكرار بين عناصر (انظر الشكل ٣). هل النقطة المرجعية القادرة على عنصر إلى اليسار أو إلى العنصر إلى اليمين؟ لماذا أسلوب الإضافة يكون قضيتين منفصلتين؟
- الجواب: إنه يشير إلى عنصر إلى اليسار. يمكنك أن ترى أن البحث عن المفقودين من الاستدعاء الأول إلى التالي. فإنه يترك موقف للإشارة إلى العقدة الأولى.

Self Check 15.5

Why does the `add` method have two separate cases?

Answer: If position is `null`, we must be at the head of the list, and inserting an element requires updating the `first` reference. If we are in the middle of the list, the `first` reference should not be changed.

- لماذا أسلوب الإضافة يكون قضيتين منفصلتين؟
- الجواب: إذا كان الموقف هو باطل، يجب أن نكون على رأس القائمة، وإدخال عنصر يتطلب استكمال أول إشارة. إذا نحن في منتصف القائمة، يجب أن لا يتم تغيير المرجع الأول.

Abstract Data Types

- There are two ways of looking at a linked list
  - *To think of the concrete implementation of such a list*
    - Sequence of node objects with links between them
  - *Think of the abstract concept of the linked list*
    - Ordered sequence of data items that can be traversed with an iterator

- هناك طريقتان للنظر في قائمة مرتبطة
- التفكير في التنفيذ الملموس لهذه القائمة
- سلسلة من عقدة الكائنات مع الروابط بينهما
- التفكير في المفهوم المجرد من القائمة المرتبطة
- تسلسل أمر من عناصر البيانات التي يمكن اجتيازه مع مكرر

Abstract Data Types

أنواع البيانات المجردة

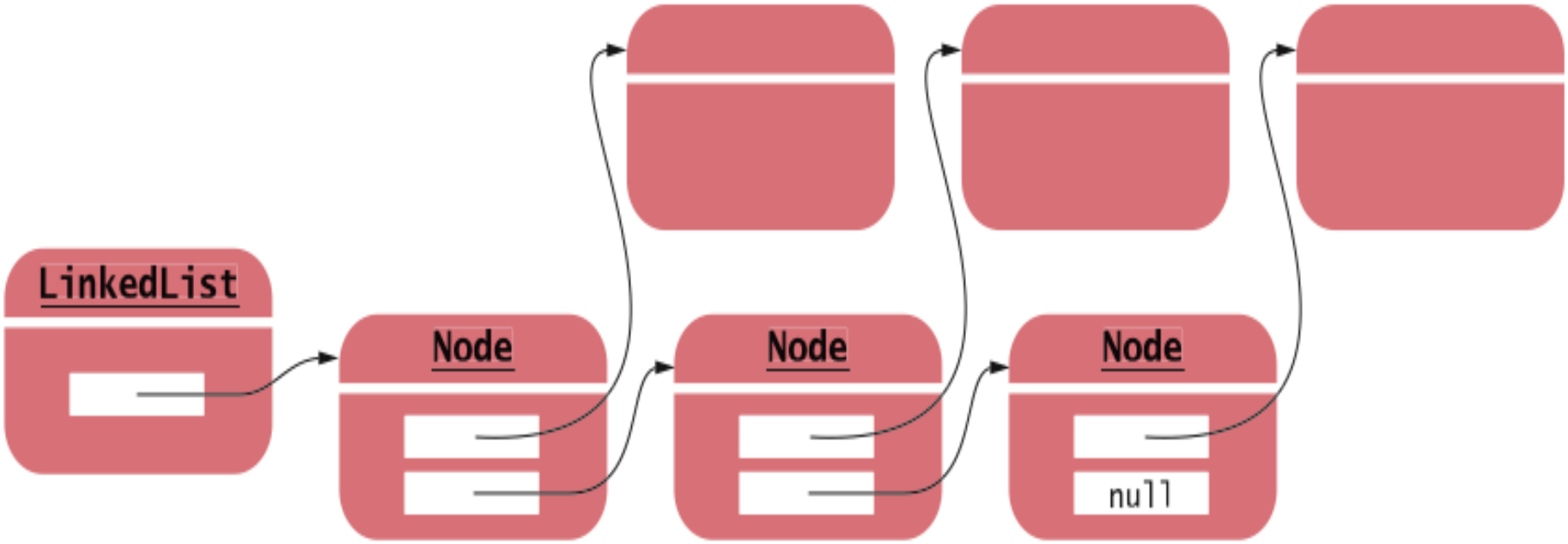


Figure 8 A Concrete View of a Linked List

Figure 9 An Abstract View of a List

Abstract Data Types

أنواع البيانات المجردة

- Define the fundamental operations on the data
- Do not specify an implementation

- تحديد العمليات الأساسية على البيانات
- لم تحدد التنفيذ

Abstract and Concrete Array Type

وتجريدية والخرسانة نوع صفيف

- As with a linked list, there are two ways of looking at an array list
- Concrete implementation: A partially filled array of object references
- We don’t usually think about the concrete implementation when using an array list
  - *We take the abstract point of view*
- Abstract view: Ordered sequence of data items, each of which can be accessed by an integer index

- كما هو الحال مع قائمة مرتبطة، هناك طريقتان للنظر في قائمة مجموعة
- التنفيذ الملموس: هناك مجموعة مملوءة جزئيا من مراجع الكائنات
- نحن لا نعتقد عادة حوالي التنفيذ الملموس عند استخدام قائمة مجموعة
- نأخذ وجهة نظر مجردة
- مجردة رأي: تسلسل أمر من عناصر البيانات، كل واحدة منها يمكن الوصول إليها من قبل مؤشر صحيح

Abstract and Concrete Data Types

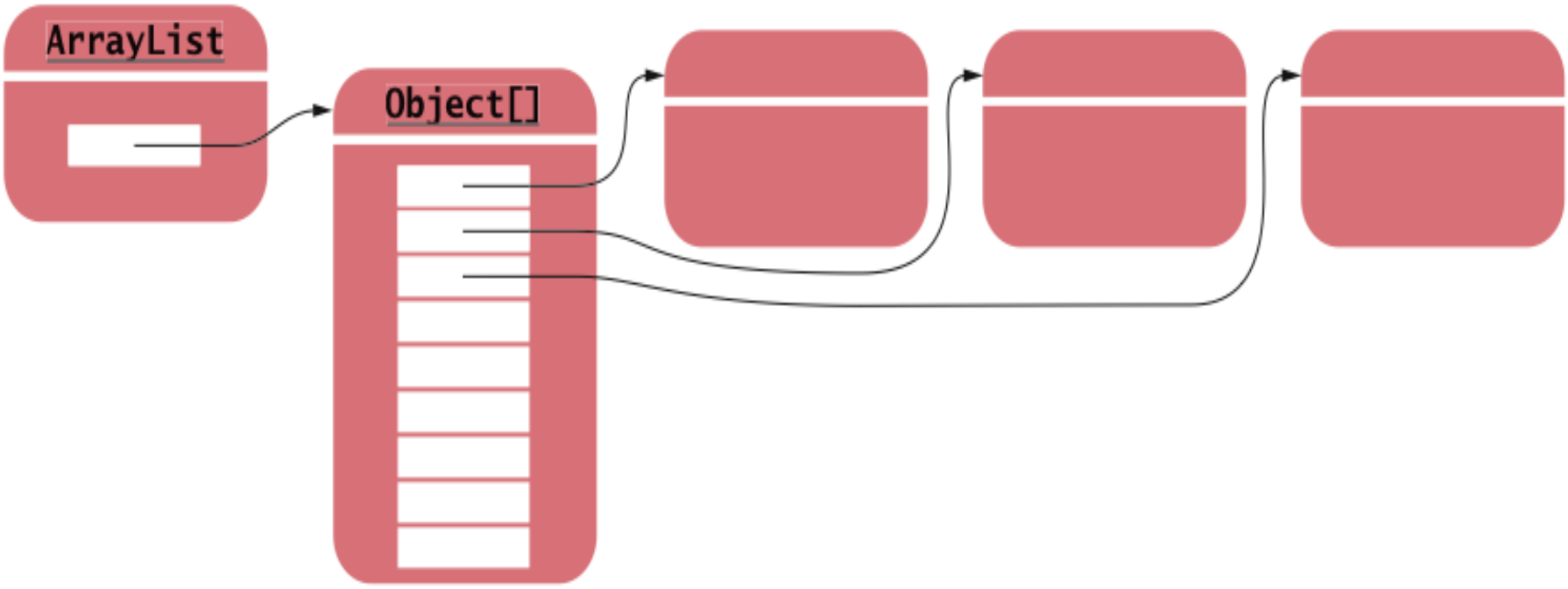


Figure 10 A Concrete View of an Array List

Figure 11 An Abstract View of an Array

Abstract and Concrete Data Types

أنواع البيانات المجردة والملموسة

- Concrete implementations of a linked list and an array list are quite different
- The abstractions seem to be similar at first glance
- To see the difference, consider the public interfaces stripped down to their minimal essentials

- تطبيقات ملموسة من قائمة مرتبطة وقائمة مجموعة مختلفة تماما
- تجريدات ويبدو أن تكون مشابهة للوهلة الأولى
- لمعرفة الفرق، والنظر في واجهات العامة جردت أسفل إلى الحد الأدنى من الضروريات الخاصة

Fundamental Operations on Array List العمليات الأساسية على قائمة صفيف

An array list allows *random access* to all elements:

يسمح على قائمة صفيف الوصول العشوائي لجميع عناصر هي:

```
public class ArrayList
{
   public Object get(int index) {...}
   public void set(int index, Object value) {...)
   ...
}
```

- يسمح قائمة مرتبطة الوصول المتسلسل لعناصرها:

Fundamental Operations on Linked List

A linked list allows *sequential access* to its elements:

```
public class LinkedList
{
   public ListIterator listIterator() {...}
   ...
}

public interface ListIterator
{
   Object next();
   boolean hasNext();
   void add(Object value);
   void remove();
   void set(Object value);
   ...
}
```

Abstract Data Types

أنواع البيانات المجردة

- `ArrayList`: Combines the interfaces of an array and a list
- Both `ArrayList` and `LinkedList` implement an interface called `List`
  - `List` *defines operations for random access and for sequential access*
- Terminology is not in common use outside the Java library
- More traditional terminology: *array* and *list*
- Java library provides concrete implementations `ArrayList` and `LinkedList` for these abstract types
- Java arrays are another implementation of the abstract array type

- ArrayList: يجمع واجهات مجموعة وقائمة
- كلا ArrayList و LinkedList تطبيق واجهة؟ دعا قائمة
- وتحدد قائمة عمليات الوصول العشوائي والوصول التتابعي
- المصطلحات ليست شائعة الاستعمال خارج مكتبة جافا
- المصطلحات الأكثر تقليدية: مجموعة وقائمة
- توفر مكتبة جافا تطبيقات ملموسة ArrayList؟ و LinkedList لهذه الأنواع المجردة
- صفائف جافا هي تنفيذ آخر من مجموعة مجردة؟ اكتب

Efficiency of Operations for Arrays and Lists

كفاءة العمليات للمصفوفات وقوائم

- Adding or removing an element
  - *A fixed number of node references need to be modified to add or remove a node, regardless of the size of the list*
  - *In big-Oh notation: O(1)*
- Adding or removing an element
  - *On average n/2 elements need to be moved*
  - *In big-Oh notation: O(n)*

- إضافة أو إزالة عنصر
- وهناك عدد ثابت من المراجع العقدة إلى تعديل لإضافة أو إزالة عقدة، بغض النظر عن حجم قائمة
- في كبيرة أوه-تدوين: (1) O
- إضافة أو إزالة عنصر
- في المتوسط ن / ٢ عناصر تحتاج إلى نقلها
- في كبيرة أوه-تدوين:) Oن)

Efficiency of Operations for Arrays and Lists

Operation	**Array**	**List**
Random access	$O(1)$	$O(n)$
Linear traversal step	$O(1)$	$O(1)$
Add/remove an element	$O(n)$	$O(1)$

Abstract Data Types

- Abstract list
  - *Ordered sequence of items that can be traversed sequentially*
  - *Allows for insertion and removal of elements at any position*
- Abstract array
  - *Ordered sequence of items with random access via an integer index*

- قائمة مجردة
- تسلسل أمر من الأدوات التي يمكن اجتيازه بالتتابع
- يسمح لإدخال وإزالة العناصر في أي موقف
- مجموعة مجردة
- أمر تسلسل البنود مع وصول عشوائي عن طريق مؤشر صحيح

Self Check 15.6

What is the advantage of viewing a type abstractly?

Answer: You can focus on the essential characteristics of the data type without being distracted by implementation details.

- ما هي ميزة عرض نوع تجريدي؟
- الإجابة: يمكنك التركيز على الخصائص الأساسية من نوع البيانات دون أن يصرف من تفاصيل التنفيذ.

Self Check 15.7

How would you sketch an abstract view of a doubly linked list? A concrete view?

Answer: The abstract view would be like Figure 9, but with arrows in both directions. The concrete view would be like Figure 8, but with references to the previous node added to each node.

- كيف يمكنك رسم وجهة نظر مجردة من قائمة مرتبطة مضاعف؟ وجهة نظر ملموس؟
- الإجابة: إن الرأي المجرد يكون مثل الرقم ٩، ولكن مع السهام في كلا الاتجاهين. ان عرض ملموس يكون مثل الشكل ٨، ولكن مع الإشارة إلى العقدة السابقة تضاف إلى كل عقدة.

Self Check 15.8

How much slower is the binary search algorithm for a linked list compared to the linear search algorithm?

Answer: To locate the middle element takes $n / 2$ steps. To locate the middle of the subinterval to the left or right takes another $n / 4$ steps. The next lookup takes $n / 8$ steps. Thus, we expect almost n steps to locate an element. At this point, you are better off just making a linear search that, on average, takes $n / 2$ steps.

- كم أبطأ هي خوارزمية البحث الثنائي لقائمة مرتبطة مقارنة مع خوارزمية البحث خطية؟
- الجواب: لتحديد العنصر المتوسط يأخذ ن / ٢ الخطوات. لتحديد منتصف subintervalإلى اليسار أو يأخذ حق ن / ٤ خطوات أخرى. وبحث المقبل يأخذ ن / ٨ خطوات. وبالتالي، فإننا نتوقع تقريبا ن الخطوات التالية لتحديد عنصر. في هذه المرحلة، كنت أفضل حالا جعل مجرد البحث الخطي أنه، في المتوسط، ويأخذ ن / ٢ الخطوات.

Stacks and Queues

- Stack: Collection of items with “last in, first out” retrieval
- Queue: Collection of items with “first in, first out” retrieval

- كومة: مجموعة من البنود مع "تستمر في، لأول مرة" استرجاع
- طابور: مجموعة من البنود مع "أولا في، أولا من" استرجاع

Stack

كومة او تكدس

- Allows insertion and removal of elements only at one end
  - *Traditionally called the top of the stack*
- New items are added to the top of the stack
- Items are removed at the top of the stack
- Called *last in, first out* or LIFO order
- Traditionally, addition and removal operations are called `push` and `pop`
- Think of a stack of books

- يسمح الإدراج وإزالة العناصر فقط في نهاية واحدة
- دعا تقليديا الجزء العلوي من كومة
- يتم إضافة عناصر جديدة إلى الجزء العلوي من كومة
- تتم إزالة البنود في الجزء العلوي من كومة
- ودعا في الماضي، لأول مرة أو النظام LIFO
- تقليديا، وتسمى إضافة وإزالة عمليات الشد والبوب
- التفكير في كومة من الكتب

Stack

- Think of a stack of books:

Figure 12
A Stack of Books

Queue

طابور

- Add items to one end of the queue (the tail)
- Remove items from the other end of the queue (the head)
- Queues store items in a *first in, first out* or FIFO fashion
- Items are removed in the same order in which they have been added
- Think of people lining up:
  - *People join the tail of the queue and wait until they have reached the head of the queue*

- إضافة عناصر إلى واحدة من نهاية الطابور (الذيل)
- إزالة العناصر من الطرف الآخر من قائمة الانتظار (الرأس)
- تخزين المواد طوابير في أول في، لأول مرة أو FIFOالأزياء
- تتم إزالة العناصر في نفس الترتيب الذي تم إضافتها
- اعتقد من الناس يصطفون:
- الناس الانضمام ذيل قائمة الانتظار والانتظار حتى وصلت إلى رأس قائمة الانتظار

A Queue

Figure 13 A Queue

Stacks and Queues: Uses in Computer Science

المكدسات وقوائم الانتظار: يستخدم في علوم الحاسب الآلي

- Queue
  - *Event queue of all events, kept by the Java GUI system*
  - *Queue of print jobs*
- Stack
  - *Run-time stack that a processor or virtual machine keeps to organize the variables of nested methods*

- طابور
- طابور الحدث من جميع الأحداث، التي يحتفظ بها نظام جافا واجهة المستخدم الرسومية
- طابور من مهام الطباعة
- كومة
- وقت التشغيل كومة أن المعالج أو الجهاز الظاهري يحتفظ لتنظيم المتغيرات أساليب المتداخلة

Stacks and Queues in the Java Library

- Class `Stack` implements the abstract stack data type and the `push` and `pop` operations
- Methods of `Queue` interface in the standard Java library include:
  - `add` *to add an element to the tail of the queue*
  - `remove` *to remove the head of the queue*
  - `peek` *to get the head element of the queue without removing it*
- `Queue` implementations in the standard library are designed for use with multithreaded programs
- The `LinkedList` class also implements the `Queue` interface, and you can use it when a queue is required:

```
Queue<String> q = new
  LinkedList<String>();
```

- الدرجة المكدس تنفذ نوع البيانات كومة مجردة والشد وعمليات البوب
- طرق واجهة قائمة الانتظار في مكتبة جافا القياسية ما يلي:
- إضافة لإضافة عنصر إلى ذيل قائمة الانتظار
- إزالة لإزالة رأس قائمة الانتظار
- نظرة خاطفة للحصول على عنصر رأس قائمة الانتظار دون إزالته
- تم تصميم تطبيقات قائمة الانتظار في المكتبة القياسية للاستخدام مع برامج مؤشرات
- الطبقة LinkedListأيضا بتطبيق واجهة قائمة الانتظار، ويمكنك استخدامه عند الحاجة طابور:

Working with Queues and Stacks

Table 4 Working with Queues and Stacks

`Queue<Integer> q = new LinkedList<Integer>();`	The `LinkedList` class implements the `Queue` interface.
`q.add(1); q.add(2); q.add(3);`	Adds to the tail of the queue; `q` is now `[1, 2, 3]`.
`int head = q.remove();`	Removes the head of the queue; `head` is set to 1 and `q` is `[2, 3]`.
`head = q.peek();`	Gets the head of the queue without removing it; `head` is set to 2.
`Stack<Integer> s = new Stack<Integer>();`	Constructs an empty stack.
`s.push(1); s.push(2); s.push(3);`	Adds to the top of the stack; `s` is now `[1, 2, 3]`.
`int top = s.pop();`	Removes the top of the stack; `top` is set to 3 and `s` is now `[1, 2]`.
`head = s.peek();`	Gets the top of the stack without removing it; `head` is set to 2.

Self Check 15.9

Draw a sketch of the abstract queue type, similar to Figures 9 and 11.

رسم تخطيطي من نوع طابور مجردة، على غرار أرقام ٩ و ١١.

Answer:

Self Check 15.10

Why wouldn’t you want to use a stack to manage print jobs?

Answer: Stacks use a “last in, first out” discipline. If you are the first one to submit a print job and lots of people add print jobs before the printer has a chance to deal with your job, they get their printouts first, and you have to wait until all other jobs are completed.

- لماذا لا تريد استخدام كومة لإدارة مهام الطباعة؟
- استخدام أكوام من "تستمر في، لأول مرة" الانضباط: الجواب. إذا كنت أول واحد لتقديم وظيفة الطباعة والكثير من الناس إضافة مهام الطباعة قبل الطابعة لديه فرصة للتعامل مع وظيفتك، ويحصلون على المطبوعات لأول مرة، وعليك أن تنتظر حتى يتم الانتهاء من جميع وظائف أخرى.