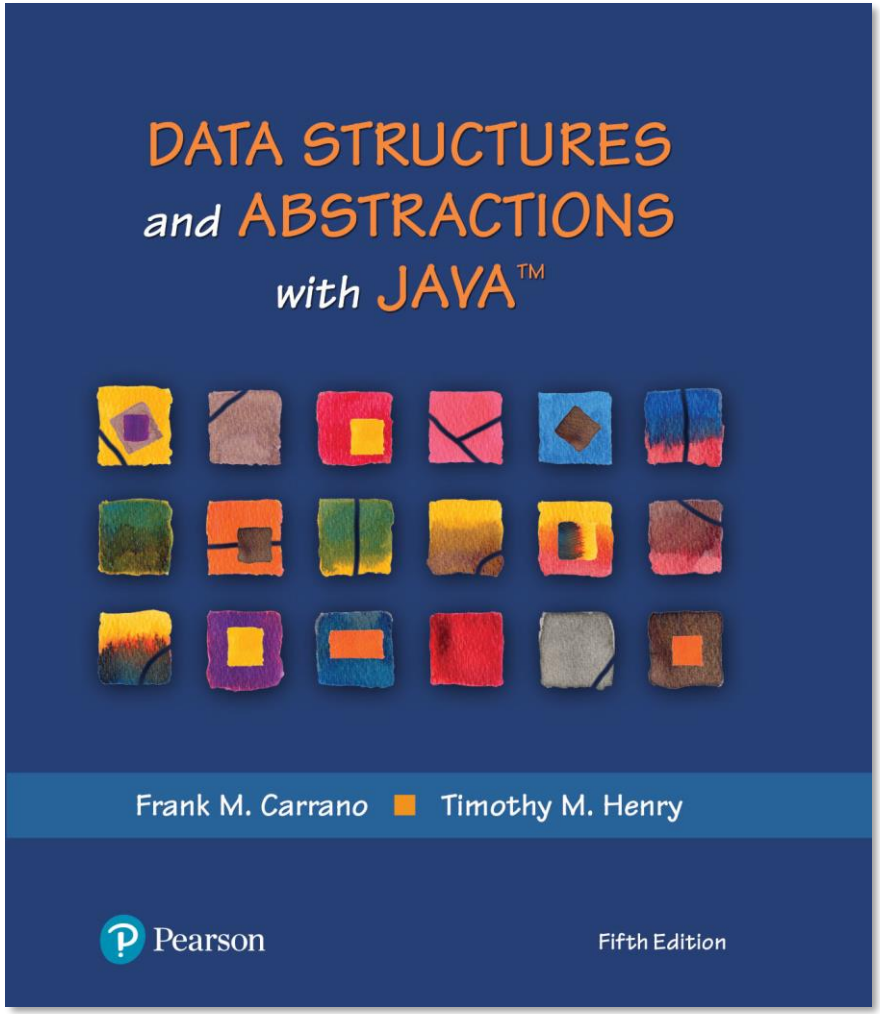
Data Structures and Abstractions with Java™

5th Edition



Chapter 18

Inheritance and Lists

Inheritance to Implement a Sorted List

```
/**  A class that implements the ADT sorted list by extending LList.
   Duplicate entries are allowed.  */
public class SortedList<T extends Comparable<? super T>>
          extends LList<T> implements SortedListInterface<T>
{
   public void add(T newEntry)
   {
      int newPosition = Math.abs(getPosition(newEntry));
      super.add(newPosition, newEntry);
   } // end add

/* < Implementations of remove(anEntry) and getPosition(anEntry) go here. >
   . . . */
} // end SortedList
```

If `SortedList` inherited methods from `LList`, we would not have to implement them again.

Inheritance to Implement a Sorted List

```
/** Adds newEntry to the list at position newPosition. */
public void add(int newPosition, T newEntry);
/** Replaces the entry at givenPosition with newEntry. */
public T replace(int givenPosition, T newEntry);
```

Although `SortedList` conveniently inherits methods such as `isEmpty` from `LList`, it also inherits two methods that a client can use to destroy the order of a sorted list.

Inheritance to Implement a Sorted List

- Possible ways to avoid the pitfall
  - Use **`SortedListInterface`** in the declaration of the sorted list.
  - Implement the list's add and replace methods within the class **`SortedList`**
  - Implement the list's add and replace methods within the class **`SortedList`** and have them throw an exception when invoked
  - We will subclass AList instead using revised ListInterface

Designing a Base Class (Part 1)

```
/** A class that implements the ADT list by using a chain of
 linked nodes that has a head reference.  */
public class LList<T> implements ListInterface<T>
{
      private Node firstNode;          // Reference to first node
      private int  numberOfEntries;

      public LList()
      {
      initializeDataFields();
      } // end default constructor

  public int clear()
  {
    initializeDataFields();
  } // end clear

/*  < Implementations of the public methods add, remove,
        replace, getEntry, contains,
    getLength, isEmpty, and toArray go here. >
  . . . */
```

LISTING 18-1 Relevant aspects of the class `LList`

Designing a Base Class (Part 2)

```
  // Initializes the class's data fields to indicate an empty list.
   private void initializeDataFields()
   {
     firstNode = null;
     numberOfEntries = 0;
   } // end initializeDataFields

   // Returns a reference to the node at a given position.
   private Node getNodeAt(int givenPosition)
   {
//   . . .
   } // end getNodeAt

   private class Node
   {
     private T data;
     private Node next;
//   . . .

   } // end Node
} // end LList
```

LISTING 18-1 Relevant aspects of the class `LList`

Designing a Base Class

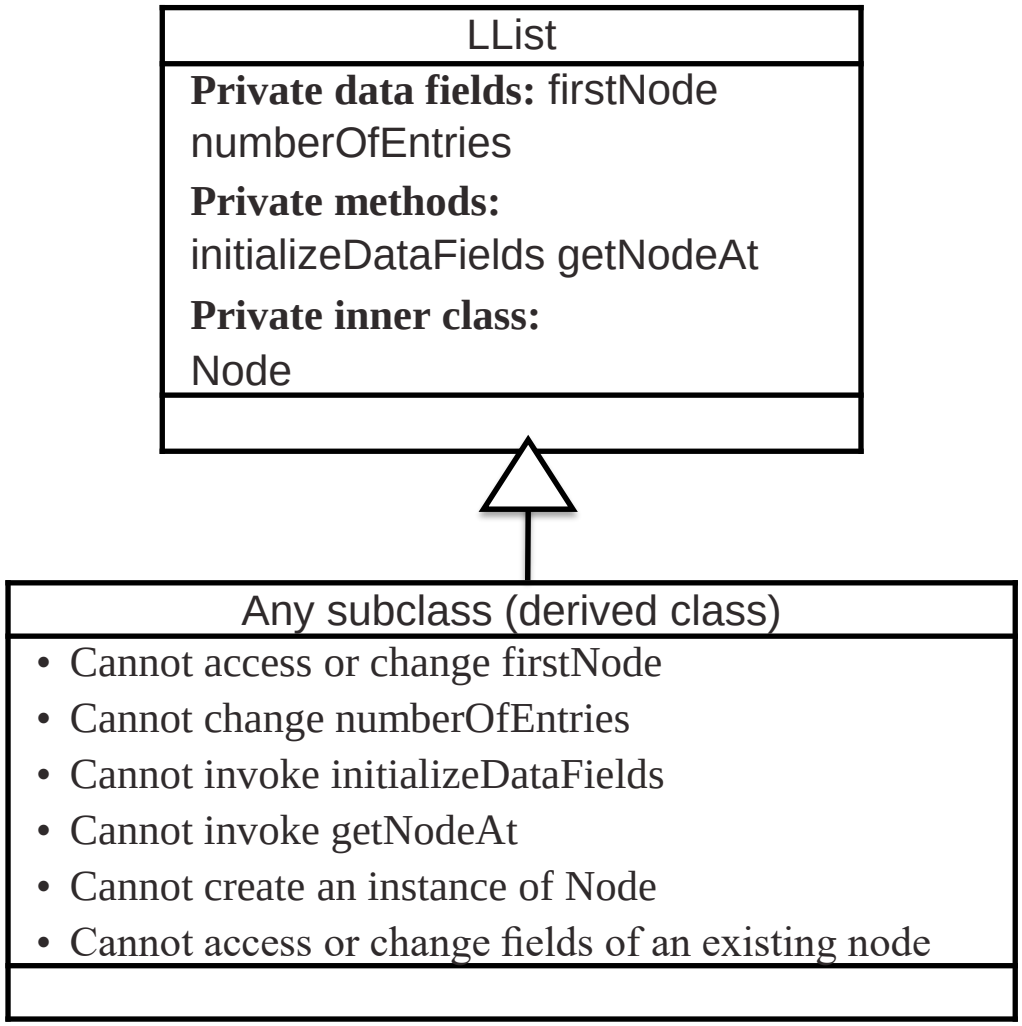


FIGURE 18-1 A derived class of the class `LList` cannot access or change anything that is private within `LList`

Designing a Base Class

```
protected final Node getFirstNode()
{
  return firstNode;
} // end getFirstNode
```

Protected method `getFirstNode`, enabling the subclass to access the head reference `firstNode`

Designing a Base Class

```
/** Adds a node to the beginning of a chain. */
protected final void addFirstNode(Node theNode)

/** Adds a node to a chain after a given node. */
protected final void addAfterNode(Node nodeBefore, Node theNode)

/** Removes a chain's first node. */
protected final T removeFirstNode()

/** Removes the node after a given one. */
protected final T removeAfterNode(Node nodeBefore)
```

Protected methods to `add` and `remove` nodes, changing `firstNode` and `numberOfEntries` as necessary

Designing a Base Class

```
/** Adds a node to the beginning of a chain. */
protected final void addFirstNode(Node theNode)
{
  // Assertion: theNode != null
  theNode.setNextNode(firstNode);
  firstNode = theNode;
  numberOfEntries++;
} // end addFirstNode
```

Definition of `addFirstNode`.

Designing a Base Class

```
public T remove(int givenPosition)
{
  T result = null;

  if ((givenPosition >= 1) && (givenPosition <= getLength()))
  {
    // Assertion: The list is not empty
    if (givenPosition == 1)       // Case 1: Remove first entry
      result = removeFirstNode();
    else                          // Case 2: givenPosition > 1
    {
      Node nodeBefore = getNodeAt(givenPosition - 1);
      result = removeAfterNode(nodeBefore);
    } // end if
    return result;                // Return removed entry
  }
  else
    throw new IndexOutOfBoundsException(
                          "Illegal position given to remove operation.");
} // end remove
```

Revision of `LList`'s method `remove`

Designing a Base Class

```
protected final T getData()
protected final void setData(T newData)
protected final Node getNextNode()
private final void setNextNode(Node nextNode)
```

Node has these four methods.

Designing a Base Class

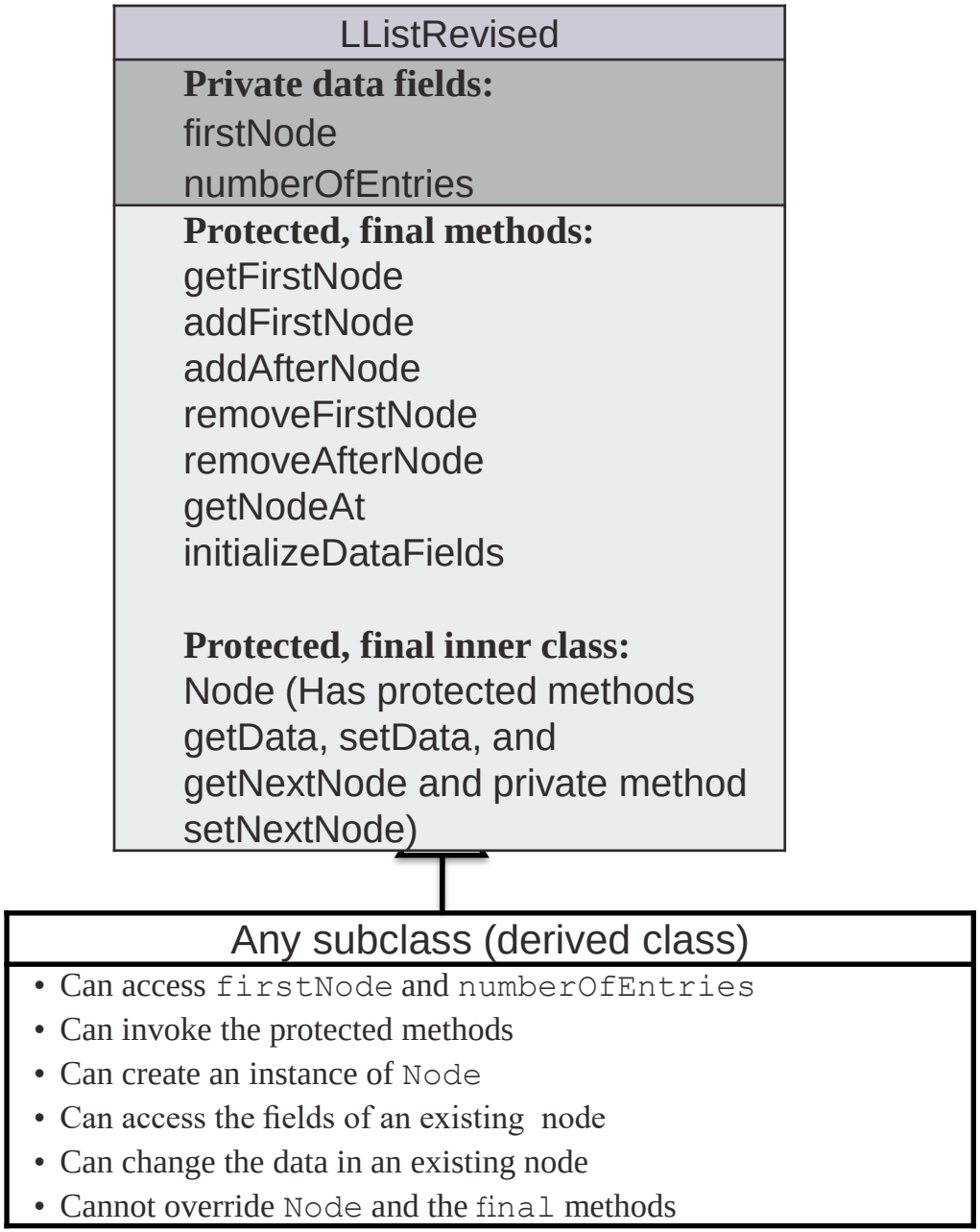


FIGURE 18-2 Access available to a class derived from the class `LListRevised`

Designing a Base Class

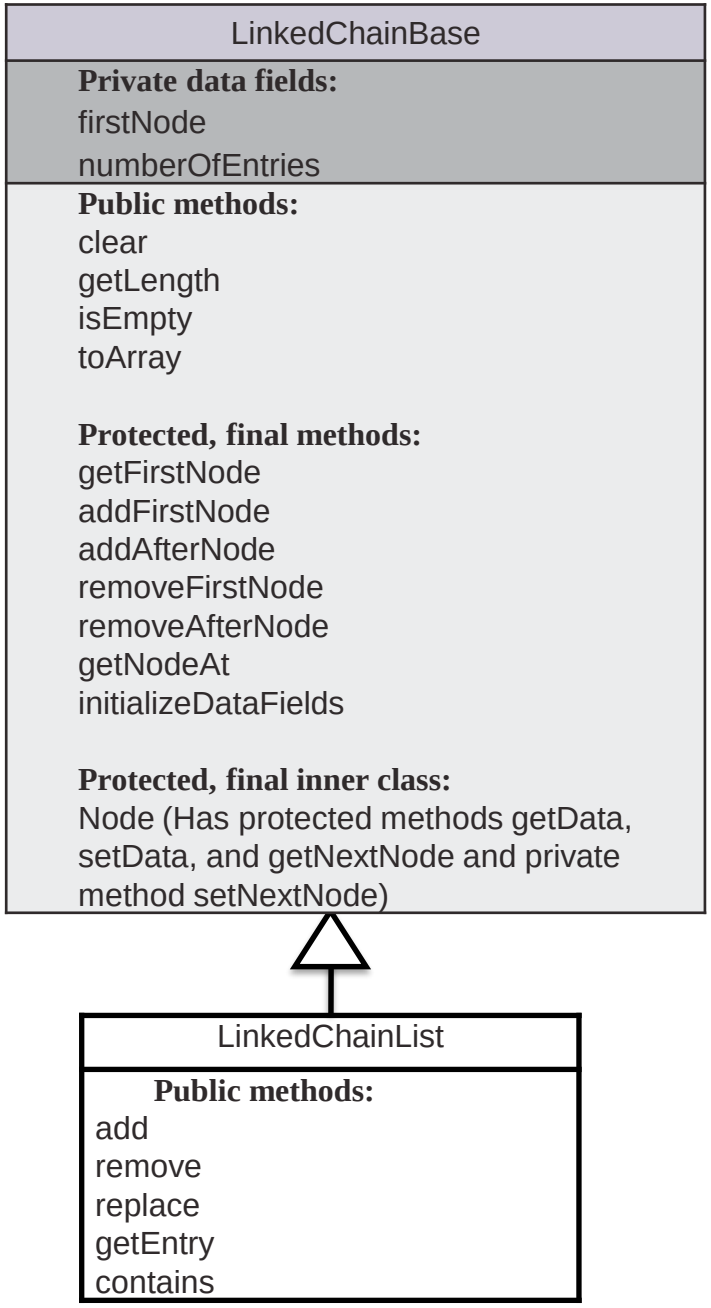


FIGURE 18-3 The separation of linked-chain operations and list operations

Creating an Abstract Base Class

```
/**  An abstract base class for use in implementing the ADT list
   by using a chain of nodes. All methods are implemented, but
   since the class is abstract, no instances can be created.  */
public abstract class LinkedChainBase<T>
{
        private Node firstNode; // Reference to first node
        private int  numberOfEntries;

        public LinkedChainBase()
        {
        initializeDataFields();
        } // end default constructor

/* < Implementations of the public methods clear, getLength, isEmpty, and toArray go here. >
   . . .

  < Implementations of the protected, final methods getFirstNode, addFirstNode,
    addAfterNode, removeFirstNode, removeAfterNode, getNodeAt, and
            initializeDataFields go here. >

        protected final class Node
        {
/*    < Implementations of the protected methods getData, setData, and getNextNode go here. >

     < Implementation of the private method setNextNode goes here. >     . . . */
        } // end Node
} // end LinkedChainBase
```

LISTING 18-2 The abstract base class `LinkedChainBase`

Efficient Implementation of a Sorted List

```
public class LinkedChainSortedList<T extends Comparable<? super T>>
        extends LinkedChainBase<T>
        implements SortedListInterface<T>
```

We want our class to extend `LinkedChainBase`

Efficient Implementation of a Sorted List

```
public void add(T newEntry)
{
  Node theNode = new Node(newEntry);
  Node nodeBefore = getNodeBefore(newEntry);

  if (nodeBefore == null) // No need to call isEmpty
    addFirstNode(theNode);
  else
    addAfterNode(nodeBefore, theNode);
} // end add
```

Details of a previous add method now hidden within the protected methods `addFirstNode` and `addAfterNode` of `LinkedChainBase`

Efficient Implementation of a Sorted List

```
private Node getNodeBefore(T anEntry)
{
  Node currentNode = getFirstNode();
  Node nodeBefore = null;

  while ((currentNode != null) &&
         (anEntry.compareTo(currentNode.getData()) > 0))
  {
     nodeBefore = currentNode;
     currentNode = currentNode.getNextNode();
  } // end while

  return nodeBefore;
} // end getNodeBefore
```

The private method `getNodeBefore`

End

Chapter 18