NC STATE UNIVERSITY

Linked Nodes

Reading
Zybooks: 14.1 & 14.2

CSC 216: Software Development Fundamentals © NC State CSC 216 Faculty

2

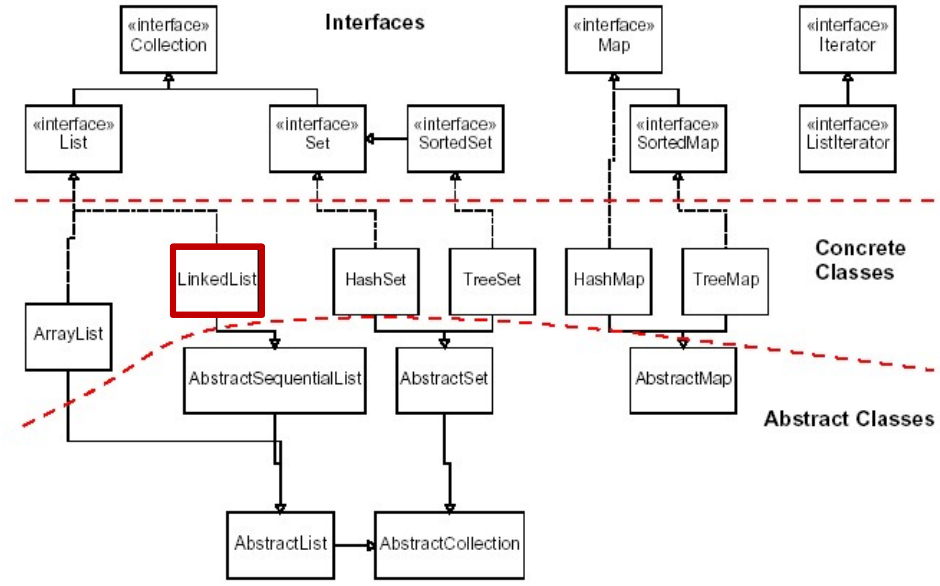
NC STATE UNIVERSITY

Collections

- Data structures – stores elements in a manner that makes it easy for a client to work with the elements
- Specific collections are specialized for particular types of operations
  - Ordering
  - Duplicates
  - Add/remove
- Common set of operations: add, remove, get, contains, size

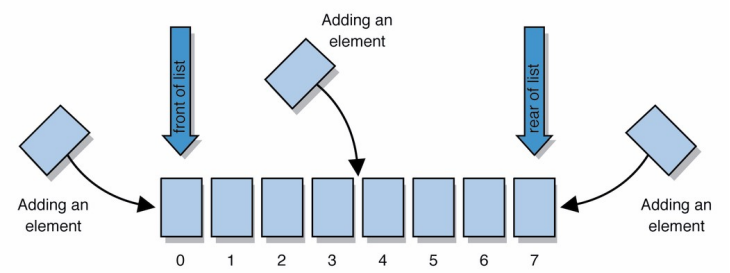
CSC 216: Software Development Fundamentals © NC State CSC 216 Faculty 3

3



Lists

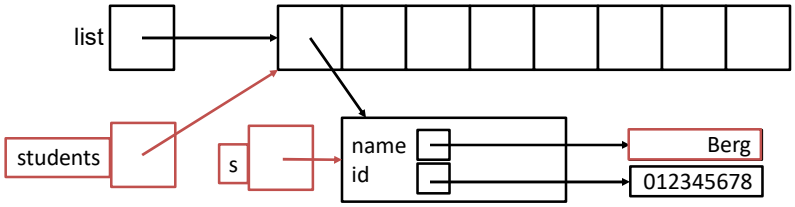
- List: an ordered collection of elements
  - Accessible by a zero-based index
  - Size: number of elements in the list
  - Elements may be added to *an empty list*, at the *front*, *middle*, and *back*



NC STATE UNIVERSITY

Reference Semantics

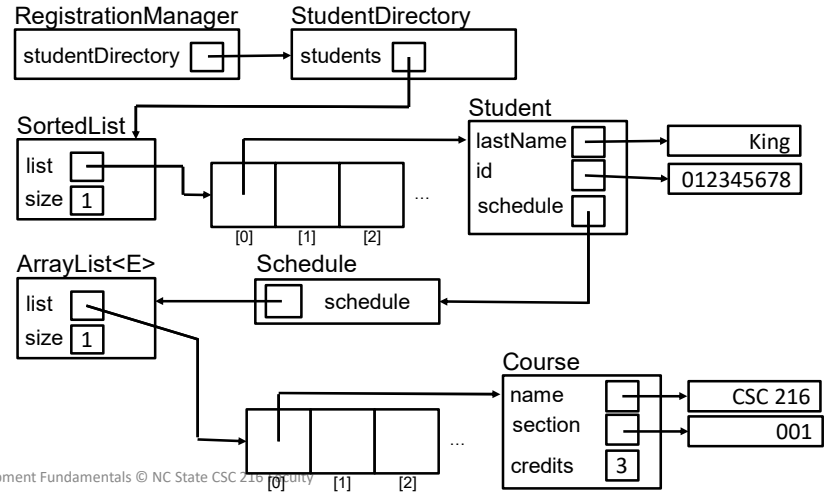
- Objects and arrays use reference semantics
  - *Efficiency*: we don't have to copy the object when passing to a method or returning from a method
    - We copy the reference!
  - *Sharing*: many references can point to or share the same object. Changes made by one are seen by the others.



NC STATE UNIVERSITY

References in Composition

- Objects can store references to other objects as fields.



Composition of the Same Type

- What would happen if we had a class that declared one of its own type as a field?

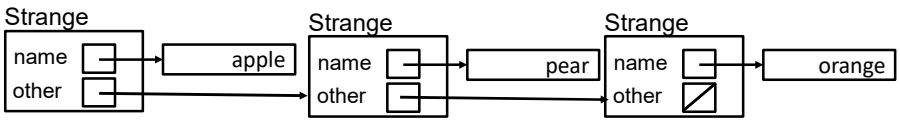
```
public class Strange {
    private String name;
    private Strange other;
}
```

  - Will this compile?
    - If so, what is the behavior of the `other` field? What can it do?
    - If not, why not? What is the error and the reasoning behind it?

Composition of the Same Type

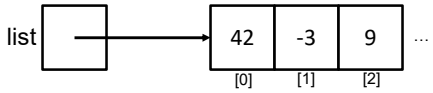
- What would happen if we had a class that declared one of its own type as a field?

```
public class Strange {
    private String name;
    private Strange other;
}
```

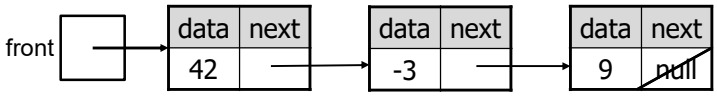


Linear Data Structures

- We can implement lists with two underlying data structures:
  - an **array** of all elements
    - `ArrayList`, `Stack`, `HashSet`, `HashMap`



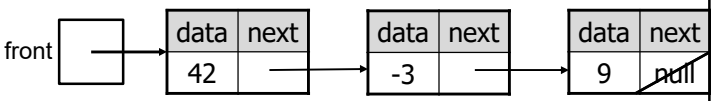
  - a set of **linked objects**, each storing one element, and one or more reference(s) to other element(s)
    - `LinkedList`, `TreeSet`, `TreeMap`



A `ListNode` Class

- Each **`ListNode`** object stores:
  - one piece of integer data
  - a reference to another list node

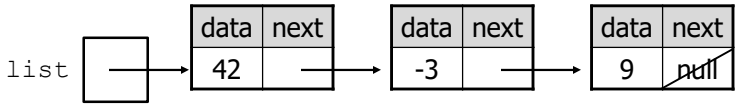
```
public class ListNode {
    int data;
    ListNode next;
}
```



- **`ListNode`**s can be "linked" into chains to store a list of values:

ListNode Client Example

```
public class ConstructList1 {
    public static void main(String[] args) {
        ListNode list = new ListNode();
        list.data = 42;
        list.next = new ListNode();
        list.next.data = -3;
        list.next.next = new ListNode();
        list.next.next.data = 9;
        list.next.next.next = null;
        System.out.println(list.data + " " + list.next.data
                           + " " + list.next.next.data);
        // 42 -3 9
    }
}
```

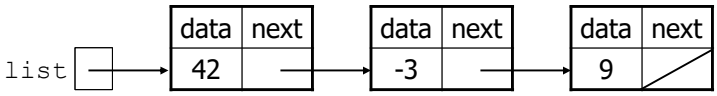


ListNode with Constructor

```
public class ListNode {
    int data;
    ListNode next;

    public ListNode(int data) {
        this(data, null);
    }

    public ListNode(int data, ListNode next) {
        this.data = data;
        this.next = next;
    }
}
```

ListNode Client Example Revisited

```
public class ConstructList1 {
    public static void main(String[] args) {
        ListNode list = new ListNode(42);
        list.next = new ListNode(-3);
        list.next.next = new ListNode(9);
        System.out.println(list.data + " " + list.next.data + " " + list.next.next.data);
        // 42 -3 9
    }
}

public class ConstructList2 {
    public static void main(String[] args) {
        ListNode list = new ListNode(9);
        list = new ListNode(-3, list);
        list = new ListNode(42, list);
        System.out.println(list.data + " " + list.next.data + " " + list.next.next.data);
        // 42 -3 9
    }
}
```

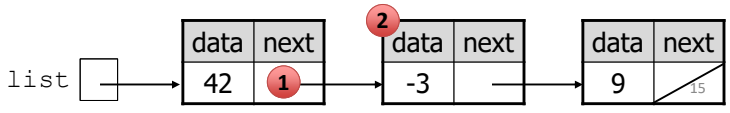


References, Objects, and Assignment

- Left-hand side of = is a ***variable***
  - A reference to an object – where the arrow starts
- Right-hand side of = is a ***value***
  - An object – what the arrow points to

1 **`list.next = value; //updates 1`**

`variable = list.next; //variable points object 2` 2



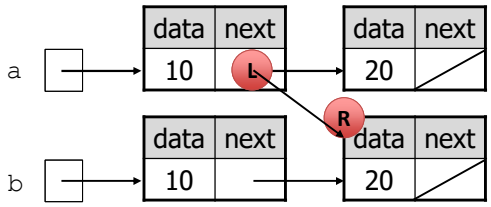
NC STATE UNIVERSITY

Manipulating References

- Consider the statement

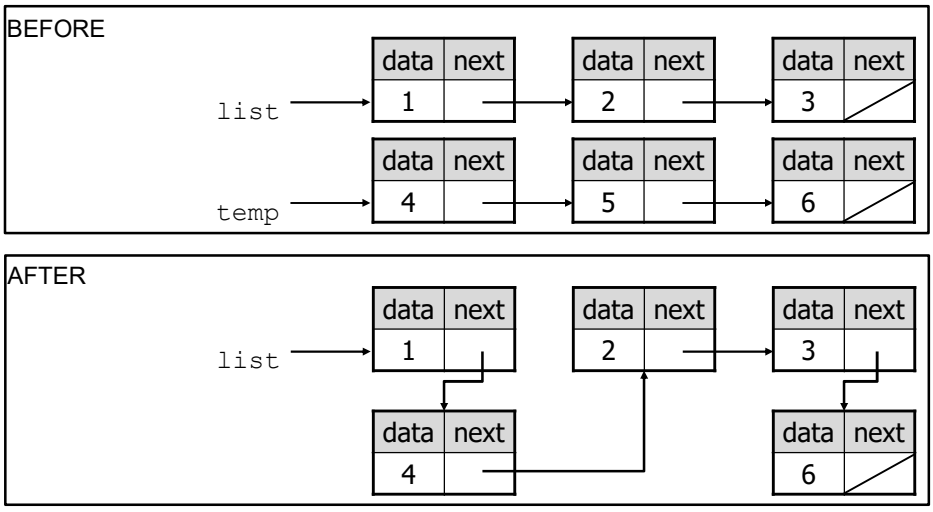
```
a.next = b.next;
```

- Meaning
  - Make the *variable* ***a.next***, point to the same *value* as ***b.next***
  - Make ***a.next*** point to the same thing ***b.next*** points to

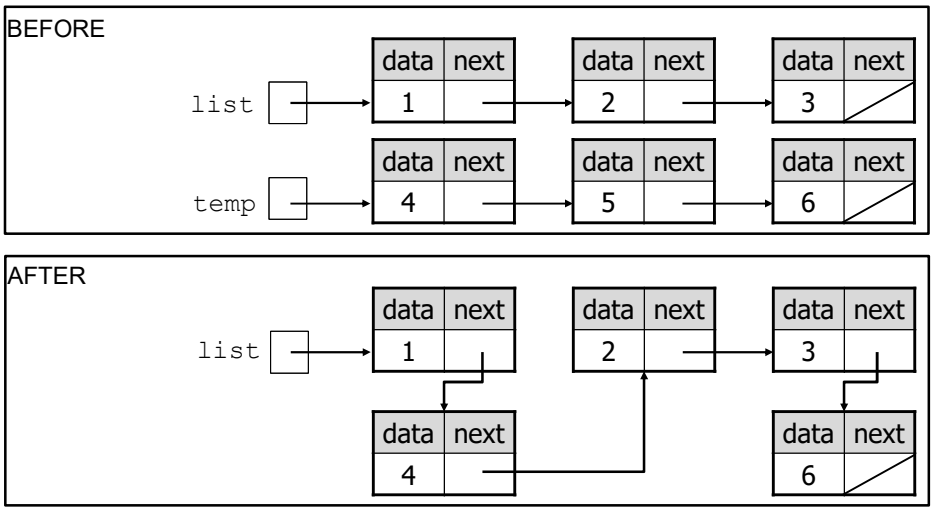


NC STATE UNIVERSITY

Manipulating References Exercise

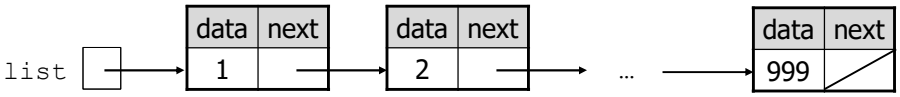


Manipulating References Exercise



Traversing `ListNode`s

- Suppose that we have a long list of **`ListNode`**s…

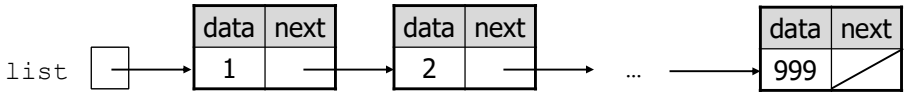


- If we don't know how long the list is, how can we print all the values in the list?
- Algorithm:
  - Start at the front of the list
  - While there are more nodes
    - Print the contents of the node
    - Move to the next node

NC STATE UNIVERSITY

What's Wrong with this Traversal?

```
while (list != null) {
    System.out.println(list.data);
    list = list.next;  //move to next node
}
```

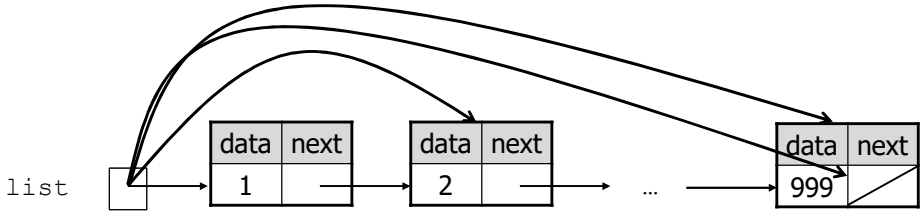


20

NC STATE UNIVERSITY

What's Wrong with this Traversal?

```
while (list != null) {
    System.out.println(list.data);
    list = list.next;  //move to next node
}
```

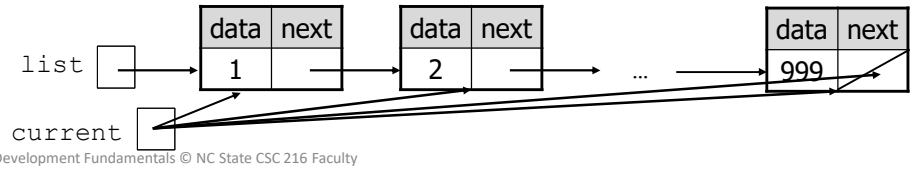


21

Correct Traversal

- Don't change the reference that represents the list
- Create a new reference, just for traversal!
  - Typically called `current`

```
ListNode current = list;
while (current != null) {
    System.out.println(current.data);
    current = current.next; //move to next
}
```



Linked list vs. array

- Algorithm to print list values:

```
ListNode front = ...;

ListNode current = list;
while (current != null) {
    System.out.println(current.data);
    current = current.next;
}
```

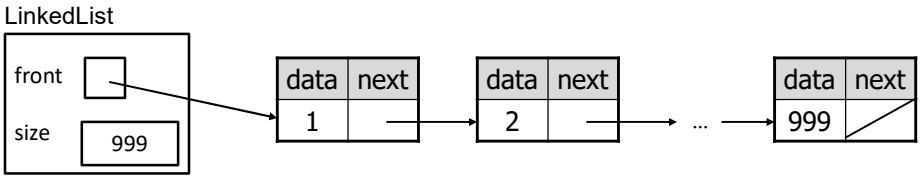
- Similar to array code:

```
int[] a = ...;

int i = 0;
while (i < a.length) {
    System.out.println(a[i]);
    i++;
}
```

Initialization | Continuation Test | Update

`LinkedList` Class

- Let's write a collection library class named **`LinkedList`**.
  - Has standard list methods
    - **`add`**, **`add`**, **`get`**, **`indexOf`**, **`remove`**, **`size`**, **`toString`**
  - The list is internally implemented as linked nodes
    - The **`LinkedList`** has a reference to its **`front`** as a field
    - **`null`** represents the end of the list; a **`null`** front signifies an empty list

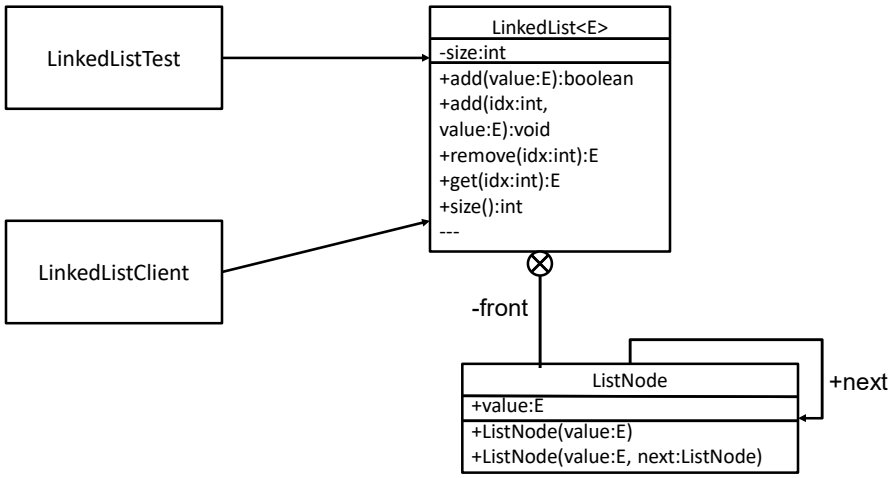


`ListNode`s in `LinkedList`s

- **`LinkedList`**s are composed of a series of connected **`ListNode`**s
- However, the client isn't interested in the **`ListNode`**s
  - The client is only interested in the data!

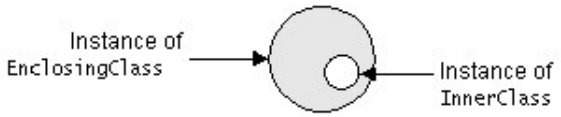
- **`ListNode`**s are typically inner classes of **`LinkedList`**s
  - The **`LinkedList`** controls the lifecycle of **`ListNode`**s
  - The **`ListNode`**s wrap the data in the **`LinkedList`**
  - The **`LinkedList`** encapsulates the **`ListNode`**'s fields

`LinkedList` Class Diagram



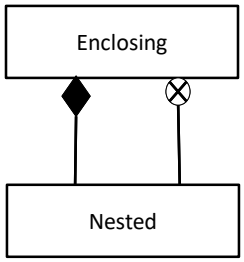
Inner Classes

- **Inner Class**: A class defined inside of another class.
  - can be created as `static` or non-static
  - May also be called a *nested class*
- Usefulness:
  - Inner classes (and their public fields) are hidden from other classes (encapsulated)
  - Inner objects can access/modify the fields of the outer object (if the inner class is not static)



Inner class syntax

```
// outer (enclosing) class
public class Enclosing {
    ...

    // inner (nested) class
    private class Nested {
        ...
    }
}
```



- Only this outer class/object can see the inner class or make objects of it.
- Each inner object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.
  - If necessary, can refer to outer object as **OuterClassName**`.this`

Adding Inner Classes

- Inner classes can be declared in a method or within an entire enclosing class
- You're telling code inspectors that the class is only of interest to the enclosing body.
  - Composition relationship where the enclosing class controls the lifecycle of the contained

```
public class LinkedList<E> {
   private ListNode front;
   private class ListNode {
      public E data; //Uses outer class generic type
      public ListNode next; //Public inner class fields can be
                            //accessed by outer class
   }
}
```

NC STATE UNIVERSITY

Generics and inner classes

```
public class Foo<T> {
    private class Inner<T> {}    // incorrect

    private class Inner {}       // correct
}
```

- If an outer class declares a type parameter, inner classes can also use that type parameter.
- Inner class should NOT redeclare the type parameter. (If you do, it will create a second type parameter with the same name.)

30

NC STATE UNIVERSITY

LinkedList

```
public class LinkedList<E> {
   private ListNode front;
   private int size;

   //Standard list methods go here – next class!

   private class ListNode {
      public E data;
      public ListNode next;
      public ListNode(E data) { this(data, null); }
      public ListNode(E data, ListNode next) {
         this.data = data;
         this.next = next;
      }
   }
}
```


31