Building Java Programs

Chapter 16
Lecture 16-1: References and linked nodes

reading: 16.1
• NP-complete is a complexity class
  • No known polynomial time ($O(n)$, $O(n^5)$…) solutions!
  • Solutions are, for example, $O(2^n)$ – ouch!

**MY HOBBY:**

**EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS**

**CHOTCHKIES RESTAURANT**

<table>
<thead>
<tr>
<th>APPETIZERS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mixed Fruit</td>
<td>2.15</td>
</tr>
<tr>
<td>French Fries</td>
<td>2.75</td>
</tr>
<tr>
<td>Side Salad</td>
<td>3.35</td>
</tr>
<tr>
<td>Hot Wings</td>
<td>3.55</td>
</tr>
<tr>
<td>Mozzarella Sticks</td>
<td>4.20</td>
</tr>
<tr>
<td>Sampler Plate</td>
<td>5.80</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SANDWICHES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Barbecue</td>
<td>6.55</td>
</tr>
</tbody>
</table>

**WE'D LIKE EXACTLY $15.05 WORTH OF APPETIZERS, PLEASE.**

...Exactly? Uhh...

Here, these papers on the knapsack problem might help you out.

Listen, I have six other tables to get to —

—As fast as possible, of course. Want something on traveling salesman?
### Collection efficiency

- Complexity class of various operations on collections:

<table>
<thead>
<tr>
<th>Method</th>
<th>ArrayList</th>
<th>Stack</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>add (or push)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>add(index, value)</td>
<td>O(N)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>indexOf</td>
<td>O(N)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>get</td>
<td>O(1)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>remove</td>
<td>O(N)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>set</td>
<td>O(1)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>size</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

- Which operations are fast, and which are slow?
- Could we build lists differently to optimize other operations?
Recall: stacks and queues

- **stack**: retrieves elements in reverse order as added
- **queue**: retrieves elements in same order as added

![Diagram of a stack and a queue]

- **push**
- **pop, peek**
- **top**
- **bottom**
- **front**
- **back**
- **add**
- **remove, peek**
- **queue**
Array vs. linked structure

- All collections in this course use one of the following:
  - an **array** of all elements
    - examples: ArrayList, Stack, HashSet, HashMap

  - **linked objects** storing a value and references to other(s)
    - examples: LinkedList, TreeSet, TreeMap

- First, we will learn how to create a **linked list**.
- To understand linked lists, we must understand **references**.
Non-contiguous memory

- Array

| 42 | -3 | 17 | 9 |

- Spread in memory

| 42 |   | 9 | -3 |   | 17 |
Arrays vs. linked lists

**Array advantages**
- Random access: can quickly retrieve any value

**Array disadvantages**
- Adding/removing in middle is O(n)
- Expanding requires creating a new array and copying elements

**Linked list advantages**
- Adding/removing in middle is O(1)
- Expanding is O(1) (just add a node)

**Linked list disadvantages**
- Sequential access: can't directly retrieve any value
A swap method?

- Does the following swap method work? Why or why not?

```java
public static void main(String[] args) {
    int a = 7;
    int b = 35;

    // swap a with b
    swap(a, b);
    System.out.println(a + " " + b);
}

public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```
Value semantics

- **value semantics**: Behavior where values are copied when assigned to each other or passed as parameters.
  - When one primitive is assigned to another, its value is copied.
  - Modifying the value of one variable does not affect others.

```java
int x = 5;
int y = x; // x = 5, y = 5
y = 17;    // x = 5, y = 17
x = 8;     // x = 8, y = 17
```
Reference semantics

- **reference semantics**: Behavior where variables actually store the address of an object in memory.
- When one reference variable is assigned to another, the object is *not* copied; both variables refer to the *same object*.

```java
int[] a1 = {4, 5, 2, 12, 14, 14, 9};
int[] a2 = a1;  // refers to same array as a1
a2[0] = 7;
System.out.println(a1[0]);  // 7
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>12</td>
<td>14</td>
<td>14</td>
<td>9</td>
</tr>
</tbody>
</table>
References and objects

• In Java, objects and arrays use reference semantics. Why?
  • *efficiency.* Copying large objects slows down a program.
  • *sharing.* It's useful to share an object's data among methods.

```java
DrawingPanel panel1 = new DrawingPanel(80, 50);
DrawingPanel panel2 = panel1;  // same window
panel2.setBackground(Color.CYAN);
```

![Diagram showing references between panel1 and panel2]
References as fields

- Objects can store references to other objects as fields.
  Example: Homework 2 (HTML Validator)
  - `HtmlValidator` stores a reference to a `Queue`
  - the `Queue` stores many references to `HtmlTag` objects
  - each `HtmlTag` object stores a reference to its element `String`

```java
private Queue<HtmlTag> tags;
...
```

```java
private String element;
...
```

```java
private String element;
...
```
Null references

- **null**: A value that does not refer to any object.
  
  - The elements of an array of objects are initialized to null.
  
    ```java
    String[] words = new String[5];
    ```
  
  - not the same as the empty string ""
  - not the same as the string "null"

- Why does Java have null? What is it used for?
Null references

- Unset reference fields of an object are initialized to null.

```java
public class Student {
    String name;
    int id;
}

Student timmy = new Student();
```

[Diagram showing `timmy` with `null` for `name` and `0` for `id`]
Things you can do w/ null

- **store null in a variable or an array element**
  
  ```java
  String s = null;
  words[2] = null;
  ```

- **print a null reference**
  
  ```java
  System.out.println(timmy.name); // null
  ```

- **ask whether a variable or array element is null**
  
  ```java
  if (timmy.name == null) { ... // true
  ```

- **pass null as a parameter to a method**
  
  - some methods don't like null parameters and throw exceptions

- **return null from a method (often to indicate failure)**
  
  ```java
  return null;
  ```
Dereferencing

- **dereference**: To access data or methods of an object.
  - Done with the dot notation, such as `s.length()`
  - When you use a . after an object variable, Java goes to the memory for that object and looks up the field/method requested.

Student timmy = new Student();
timmy.name = "Timmah";
String s = timmy.name.toUpperCase();
Null pointer exception

- It is illegal to dereference `null` (it causes an exception).
- `null` does not refer to any object; it has no methods or data.

```java
Student timmy = new Student();
String s = timmy.name.toUpperCase();  // ERROR
```

Output:
```
Exception in thread "main"
java.lang.NullPointerException
at Example.main(Example.java:8)
```
References to same type

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

```java
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?
A list node class

public class ListNode {
    int data;
    ListNode next;
}

• Each list node object stores:
  • one piece of integer data
  • a reference to another list node

• ListNode objects can be "linked" into chains to store a list of values:

```
<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td></td>
</tr>
<tr>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>null</td>
</tr>
</tbody>
</table>
```
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 = 17;
        list.next.next.next = null;
        System.out.println(list.data + " " + list.next.data + " " + list.next.next.data);
        // 42 -3 17
    }
}
public class ListNode {
    int data;
    ListNode next;

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

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

• Exercise: Modify the previous client to use these constructors.
Linked node problem 1

- What set of statements turns this picture:

```
list
10 ----> 20
```

- Into this?

```
list
10 ----> 20 ----> 30
```
References vs. objects

\[ \text{variable} = \text{value}; \]

a *variable* (left side of \(=\)) is an arrow \(\text{variable} = a\). (the base of an arrow)

a *value* (right side of \(=\)) is an object \(=\text{value}\). (a box; what an arrow points at)

- For the list at right:
  - \(a\text{.next} = \text{value};\) means to adjust where 1 points
  - \(\text{variable} = a\text{.next};\) means to make \text{variable} point at 2

\[ \begin{array}{c|c}
\text{data} & \text{next} \\
\hline
10 & 1 \\
\hline
20 & \end{array} \]
Reassigning references

- when you say:
  - `a.next = b.next;`

- you are saying:
  - "Make variable `a.next` refer to the same value as `b.next`."  
  - Or, "Make `a.next` point to the same place that `b.next` points."
Linked node problem 2

- What set of statements turns this picture:

  list →

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

  list →

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

- Into this?

  list →

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

  list →

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

  list →

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>
Linked node problem 3

- What set of statements turns this picture:

```
list1 -> | data | next |
       -> | 10   | ---- |
list2 -> | data | next |
       -> | 30   | ---- |
```

- Into this?

```
list1 -> | data | next |
       -> | 10   | ---- |
       -> | 20   | ---- |
       -> | 30   | ---- |
list2 -> | data | next |
       -> | 40   | ---- |
```
Linked node problem 3

- How many ListNode variables?

- Which variables change?

```
list1  |
| 1 |
| 10 | 20 |
| 30 | 40 |

list2  |
| 4 |
| 10 | 20 |
| 40 |
```
Linked node problem 4

- What set of statements turns this picture:

  list \[ \quad \begin{array}{|c|c|} \hline 
  \text{data} & \text{next} \\ \hline 
  10 & \cdots \\ \hline 
  \end{array} \quad \rightarrow \quad \begin{array}{|c|c|} \hline 
  \text{data} & \text{next} \\ \hline 
  990 & \cdots \\ \hline 
  \end{array} \quad \rightarrow \quad \begin{array}{|c|c|} \hline 
  \text{data} & \text{next} \\ \hline 
  1000 & \cdots \\ \hline 
  \end{array} \]

- Into this?

  list \[ \quad \begin{array}{|c|c|} \hline 
  \text{data} & \text{next} \\ \hline 
  10 & \cdots \\ \hline 
  \end{array} \quad \rightarrow \quad \begin{array}{|c|c|} \hline 
  \text{data} & \text{next} \\ \hline 
  990 & \quad \rightarrow \quad \begin{array}{|c|c|} \hline 
  \text{data} & \text{next} \\ \hline 
  1000 & \cdots \\ \hline 
  \end{array} \]