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!
**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>

- 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 stack and queue operations]

- **push**
- **pop, peek**
- **remove, peek**
- **add**
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>
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);
```
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`

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

private String element;
...
```

```
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 "" or 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
class Student {
    String name;
    int id;
}

Student timmy = new Student();
```

![Diagram showing that `timmy`'s `name` field is `null` and `id` field is `0`.](image-url)
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.

```java
Student timmy = new Student();
timmy.name = "Timmah";
String s = timmy.name.toUpperCase();
```

```
Student
- name
- id

String
- 'T' 'i' 'm' 'm' 'a' 'h'

- public int indexOf(String s) {...}
- public int length() {...}
- public String 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

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

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

- **ListNode**s can be "linked" into chains to store a list of values:
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
    }
}
List node w/ constructor

```java
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 →

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

- Into this?

  list →

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>30</td>
</tr>
</tbody>
</table>
References vs. objects

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

A variable (left side of \( = \)) is an arrow (the base of an arrow)
A value (right side of \( = \)) is an object (a box; what an arrow points at)

For the list at right:

- \( \text{a.next} = \text{value}; \)
  means to adjust where 1 points

- \( \text{variable} = \text{a.next}; \)
  means to make \text{variable} point at 2
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  | data | next |
10    |      |
      |      |
20    |      |
```

- Into this?

```
list  | data | next |
30    |      |
      |      |
10    |      |
      |      |
20    |      |
```
Linked node problem 3

- What set of statements turns this picture:

\[
\text{list1} \rightarrow \begin{array}{c|c} \text{data} & \text{next} \\ \hline 10 & \end{array} \rightarrow \begin{array}{c|c} \text{data} & \text{next} \\ \hline 20 & \end{array} \\
\text{list2} \rightarrow \begin{array}{c|c} \text{data} & \text{next} \\ \hline 30 & \end{array} \rightarrow \begin{array}{c|c} \text{data} & \text{next} \\ \hline 40 & \end{array}
\]

- Into this?

\[
\text{list1} \rightarrow \begin{array}{c|c} \text{data} & \text{next} \\ \hline 10 & \end{array} \rightarrow \begin{array}{c|c} \text{data} & \text{next} \\ \hline 20 & \end{array} \rightarrow \begin{array}{c|c} \text{data} & \text{next} \\ \hline 30 & \end{array} \\
\text{list2} \rightarrow \begin{array}{c|c} \text{data} & \text{next} \\ \hline 40 & \end{array}
\]
Linked node problem 3

- How many ListNode variables?

  - list1: data = 10, next = 2
  - list2: data = 30, next = 5

- Which variables change?

  - list1: data = 10, next = 2
  - list2: data = 40, next = 5
Linked node problem 4

• What set of statements turns this picture:

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

... → | data | next |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>990</td>
<td></td>
</tr>
</tbody>
</table>

• Into this?

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

... → | data | next |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>990</td>
<td></td>
</tr>
</tbody>
</table>

... → | data | next |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td></td>
</tr>
</tbody>
</table>