Array vs. linked structure

- All collections in this course use one of the following:
  - an **array** of all elements
    
    | 42 | -3 | 17 | 9 |
    
  - **linked objects** storing a value and references to other(s)
    
    front → 42 → -3 → 17 → 9 → null
    
    - 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

Arrays vs. linked lists

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

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

- Linked list advantages
  - Adding/removing in middle is constant time
  - Expanding is constant time (just add a node)

- Linked list disadvantages
  - Sequential access: can't directly retrieve any value
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
```

```
a1  index  0  1  2  3  4  5  6
a2  value  7  5  2 12 14 14  9
```
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;
}
```
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 panel1 and panel2 with arrows pointing to each other.](image)
References as fields

- Objects can store references to other objects as fields.
  
  Example: Homework 2 (HTML Manager)
  - `HTMLManager` 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
String html
```

```java
String body
```
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];
    ```

  
<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

- 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
public class Student {
    String name;
    int id;
}

Student timmy = new Student();
```

![Diagram](#)
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();
```

```
<table>
<thead>
<tr>
<th>Student</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>'T' 'i' 'm' 'm' 'a' 'h'</td>
</tr>
<tr>
<td>id</td>
<td>0</td>
</tr>
<tr>
<td>public int indexOf(String s) { ... }</td>
<td></td>
</tr>
<tr>
<td>public int length() { ... }</td>
<td></td>
</tr>
<tr>
<td>public String toUpperCase() { ... }</td>
<td></td>
</tr>
</tbody>
</table>
```
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:
List node client example

```java
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
    }
}
```

Data structures:
- `ListNode`
- `data` property
- `next` property

Diagram:
- `list` node
- `data: 42`
- `next` pointer
- `next node`: `data: -3`
- `next.next` node: `data: 17`
- `next.next.next` node: `null`
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:

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

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

• Into this?

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

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

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

\texttt{variable = value;}

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

\begin{itemize}
  \item For the list at right:
    \begin{itemize}
      \item \texttt{a.next = value;}
          means to adjust where \textcolor{blue}{1} points
      \item \texttt{variable = a.next;}
          means to make \textcolor{blue}{variable} point at \textcolor{blue}{2}
    \end{itemize}
\end{itemize}
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     |
10     |      | 20     |
```
Linked node problem 3

- What set of statements turns this picture:

  list1
  +---+---+
  | 10 |   |
  +---+---+
  list2
  +---+---+
  | 30 |   |
  +---+---+

  list1
  +---+---+
  | 10 |   |
  +---+---+
  list2
  +---+---+
  | 40 |   |

- Into this?

  list1
  +---+---+
  | 10 |   |
  +---+---+
  list2
  +---+---+
  | 40 |   |

  list1
  +---+---+
  | 10 |   |
  +---+---+
  list2
  +---+---+
  | 30 |   |
Linked node problem 3

- How many ListNode variables?

<table>
<thead>
<tr>
<th>List</th>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>list1</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>list2</td>
<td>30</td>
<td>40</td>
<td></td>
</tr>
</tbody>
</table>

- Which variables change?

<table>
<thead>
<tr>
<th>List</th>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>list1</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>list2</td>
<td>40</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Linked node problem 4

- What set of statements turns this picture:

```
list → data: 10 next → ... → data: 990 next
```

- Into this?

```
list → data: 10 next → ... → data: 990 next → data: 1000 next
```