Questions during Class?
Raise hand or send here

sli.do   #cse123

BEFORE WE START

**Talk to your neighbors:**

Did you see / light any fireworks last night? If not, what did you do with your break?

Music: [123 24su Lecture Tunes](#)

**Instructor:** Joe Spaniac

**TAs:**
Andras Daniel
Eric Nicole
Sahej Trien
Zach
Lecture Outline

• Announcements

• Contiguous vs. Non-contiguous memory

• Reference Semantics
  - Trains cont.

• Linked Nodes
  - ListNode class
  - Iterating over ListNode
Announcements

• Resubmission Period 1 due tonight (7/5) at 11:59pm
  - Submit the assignment again, mark your new attempt final, fill out the linked google form
• Creative Project 2 is out, due Wednesday (7/10) at 11:59pm
  - Generally regarded as a fun one!
• Resubmission Period 2 opening tonight, due next Friday (7/12)
  - Assignments available: C1, P1
• Check-in 2 next Thursday (7/11)
  - Taken in quiz section, should help in preparation for the quiz.
  - Reminder: you only need to attend 2 of these for the grade, but they should be beneficial regardless.
Lecture Outline

• Announcements

• Reference Semantics
  - Trains cont.

• Contiguous vs. Non-contiguous memory

• Linked Nodes
  - ListNode class
  - Iterating over ListNode
Reference Semantics

- In Java, variables are treated two different ways:

<table>
<thead>
<tr>
<th>Value Semantics</th>
<th>Reference Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive types (int, double, boolean) + Strings</td>
<td>Object types (int[], Scanner, ArrayList)</td>
</tr>
<tr>
<td>Values stored locally</td>
<td>Values stored in memory, reference stored locally</td>
</tr>
<tr>
<td>Initialization copies value (many copies of value)</td>
<td>Initialization copies reference (only one value)</td>
</tr>
</tbody>
</table>

```
int x = 10;
int y = x;
y++;     // x remains unchanged
```

```
int[] x = new int[10];
int[] y = x;
y[0]++;    // x[0] changed
```

- We often draw “reference diagrams” to keep track of everything

![Diagram](https://example.com/diagram.png)
Reference Semantics

• In Java, variables are treated two different ways:

<table>
<thead>
<tr>
<th>Value Semantics</th>
<th>Reference Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive types (int, double, boolean) + Strings</td>
<td>Object types (int[], Scanner, ArrayList)</td>
</tr>
<tr>
<td>Values stored locally</td>
<td>Values stored in memory, reference stored locally</td>
</tr>
<tr>
<td>Initialization copies value (many copies of value)</td>
<td>Initialization copies reference (only one value)</td>
</tr>
</tbody>
</table>

int x = 10;
int y = x;
y++;     // x remains unchanged

int[] x = new int[10];
int[] y = x;
y[0]++;    // x[0] changed

• We often draw “reference diagrams” to keep track of everything

\[
x \xrightarrow{10} y \xrightarrow{10}
\]
Reference Semantics

• In Java, variables are treated two different ways:

<table>
<thead>
<tr>
<th>Value Semantics</th>
<th>Reference Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive types (int, double, boolean) + Strings</td>
<td>Object types (int[], Scanner, ArrayList)</td>
</tr>
<tr>
<td>Values stored locally</td>
<td>Values stored in memory, reference stored locally</td>
</tr>
<tr>
<td>Initialization copies value (many copies of value)</td>
<td>Initialization copies reference (only one value)</td>
</tr>
</tbody>
</table>

```
int x = 10;
int y = x;
y++;     // x remains unchanged
```

```
int[] x = new int[10];
int[] y = x;
y[0]++;    // x[0] changed
```

• We often draw “reference diagrams” to keep track of everything

```
x 10
y 11
```
Reference Semantics

• In Java, variables are treated two different ways:

<table>
<thead>
<tr>
<th>Value Semantics</th>
<th>Reference Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive types (int, double, boolean) + Strings</td>
<td>Object types (int[], Scanner, ArrayList)</td>
</tr>
<tr>
<td>Values stored locally</td>
<td>Values stored in memory, reference stored locally</td>
</tr>
<tr>
<td>Initialization copies value (many copies of value)</td>
<td>Initialization copies reference (only one value)</td>
</tr>
</tbody>
</table>

```
int x = 10;
int y = x;
y++; // x remains unchanged
```

```
int[] x = new int[10];
int[] y = x;
y[0]++; // x[0] changed
```

• We often draw “reference diagrams” to keep track of everything

![Reference Diagram](image)
Reference Semantics

• In Java, variables are treated two different ways:

<table>
<thead>
<tr>
<th>Value Semantics</th>
<th>Reference Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive types (int, double, boolean) + Strings</td>
<td>Object types (int[], Scanner, ArrayList)</td>
</tr>
<tr>
<td>Values stored locally</td>
<td>Values stored in memory, reference stored locally</td>
</tr>
<tr>
<td>Initialization copies value (many copies of value)</td>
<td>Initialization copies reference (only one value)</td>
</tr>
</tbody>
</table>

```java
int x = 10;
int y = x;
y++;     // x remains unchanged
```

```java
int[] x = new int[10];
int[] y = x;
y[0]++;    // x[0] changed
```

• We often draw “reference diagrams” to keep track of everything

![Reference Diagram](image)
Reference Semantics

• In Java, variables are treated two different ways:

<table>
<thead>
<tr>
<th>Value Semantics</th>
<th>Reference Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive types (int, double, boolean) + Strings</td>
<td>Object types (int[], Scanner, ArrayList)</td>
</tr>
<tr>
<td>Values stored locally</td>
<td>Values stored in memory, reference stored locally</td>
</tr>
<tr>
<td>Initialization copies value (many copies of value)</td>
<td>Initialization copies reference (only one value)</td>
</tr>
</tbody>
</table>

```
int x = 10;
int y = x;
y++;     // x remains unchanged
```

```
int[] x = new int[10];
int[] y = x;
y[0]++;    // x[0] changed
```

• We often draw “reference diagrams” to keep track of everything
Lecture Outline

• Announcements

• Reference Semantics
  - Trains cont.

• Contiguous vs. Non-contiguous memory

• Linked Nodes
  - ListNode class
  - Iterating over ListNode
Lecture Outline

• Announcements

• Reference Semantics
  - Trains cont.

• *Contiguous vs. Non-contiguous memory*

• Linked Nodes
  - ListNode class
  - Iterating over ListNode
Contiguous vs. Non-contiguous

• Computer memory = one really, really big array.
Contiguous vs. Non-contiguous

- Computer memory = one really, *really* big array.
  - `int[] arr = new int[10];`

```
  85  47  -51  44  -38  35  -58  79  27  -14  -9  -36  11  5
 13  -17  -85  -99  -20  -33  54  38  -66  8  9  53  71  39
 36  24  27  90  -32  72  -73  11  -85  29  40  80  -77  -79
-90  -64  29  -27  91  64  28  -97  44  59  26  -35  34  21
-68  76  -1  -6  -52  77  21  37  80  69  -34  8  -79  -77
 1  -46  -26  99  -24  -98  25  -79  92  -18  14  57  22  20
-76  -5  -86  -64  66  -78  47  -66  69  18  -74  -53  41  -86
-31  -9  90  -53  46  55  85  37  52  58  70  -13  59  79
 17  20  91  -55  -74  0  -96  -69  -36  90  45  -60  -95  21
```

Memory
Contiguous vs. Non-contiguous

- Computer memory = one really, really big array.
  - `int[] arr = new int[10];`

We call this “contiguous” memory
Contiguous vs. Non-contiguous

- Computer memory = one really, really big array.
  - `EngineCar engine = new EngineCar("Empire Builder", 10, new SleeperCar(10));`

<table>
<thead>
<tr>
<th>engine</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="https://via.placeholder.com/150" alt="Table" /></td>
</tr>
</tbody>
</table>
Contiguous vs. Non-contiguous

- Computer memory = one really, really big array.
  - EngineCar engine = new EngineCar("Empire Builder", 10, new SleeperCar(10));
Contiguous vs. Non-contiguous

• Computer memory = one really, really big array.
  
  - EngineCar engine = new EngineCar("Empire Builder", 10, new SleeperCar(10));
Contiguous vs. Non-contiguous

- Computer memory = one really, really big array.
  - EngineCar engine = new EngineCar("Empire Builder", 10, new SleeperCar(10));

We call this “non-contiguous” memory
Contiguous vs. Non-contiguous

• Computer memory = one really, really big array.

• Contiguous memory = impossible to resize directly
  - Surrounding stuff in memory (we can’t just overwrite)
  - Best we can manage is get more space and copy

• Non-contiguous memory = easy to resize
  - Just get some more memory and link it to the rest

• Is it possible to create a non-contiguous List implementation?
  - Could make the resizing / shifting problems easier...
Lecture Outline

• Announcements

• Reference Semantics
  - Trains cont.

• Contiguous vs. Non-contiguous memory

• Linked Nodes
  - ListNode class
  - Iterating over ListNode
Linked Nodes

• We want to chain together ints “non-contiguously”
  - A bunch of train cars where each is responsible for a single integer

• Accomplish this with nodes we link together
  - Each node stores an int (data) and an reference to the next node (next)
ListNode

• Java class representing a “node”

• Two fields to store discussed state:
  - Fields are public?! We’ll come back to this

• Why can ListNode be a field in the ListNode class?

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

- General pattern iteration code will follow:

```java
ListNode curr = front;
while (curr != null) {
    // Do something
    curr = curr.next;
}
```