sit with anyone that looks friendly

CMPS 12B/M
Introduction to Data Structures

- Instructor: Nathan Whitehead

Topics for Midterm 10/31/2014

- Basics
  - Programming constructs
    - Loops
    - Variables
    - Break
    - Functions
  - Java classes and objects
    - Declaring class
    - Constructors
    - Methods
    - Public vs. private

Topics

- Tools
  - Makefiles
    - Rules
    - Variables
    - Targets
  - Git
    - Committing changes
- Big O
  - Identifying O(1), O(n), O(log n), O(n^2), O(n log n)
  - Simplifying formulas into big O
  - Which function class dominates for big n

Topics

- Arrays
  - Unordered versus Ordered
  - *Insert*, *Delete*, *Find*
  - Duplicates
- Sorting
  - Bubble sort, Selection sort, Insertion sort
  - Stability
  - Merge sort
  - Choosing a sort (comparisons, swaps/moves)

Topics

- Stacks, Queues, Priority Queue
  - Simple array implementation of stack
  - Circular buffer implementation
  - Operations
  - Ordered Array implementation of priority queue
  - Postfix arithmetic relationship to stacks
- Linked Lists
  - Nodes/Links class
  - insertFirst, deleteFirst, find, deleteLast, delete
  - Double ended lists

Class and Variable Review

- Java Review part 1

Custom Class

```
class MyClass {
   public int value;
}
```

defining a custom class

```
MyClass o;
```

a variable with type of the custom class

```
o = new MyClass();
```

creating a new instance of the class

```
o.value = 5;
```

accessing public variables

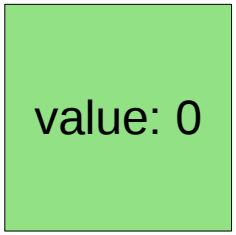
Names versus Values

```
class MyClass {
   public int value;
}


MyClass p, q;

p = new MyClass();
p.value = 42;

q = p;

q.value = 17;

What is p.value?
```

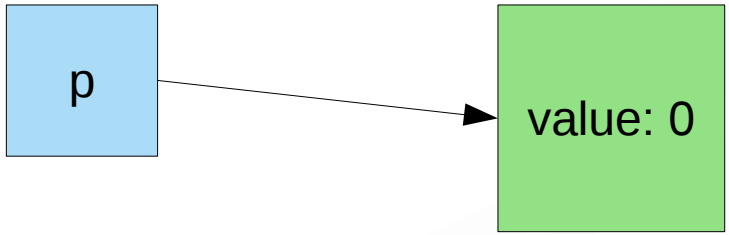
names versus values

Walkthrough

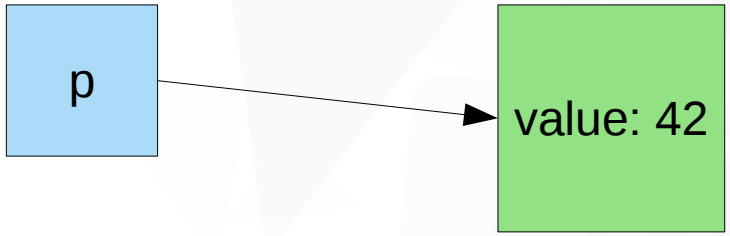
```
new MyClass()
```



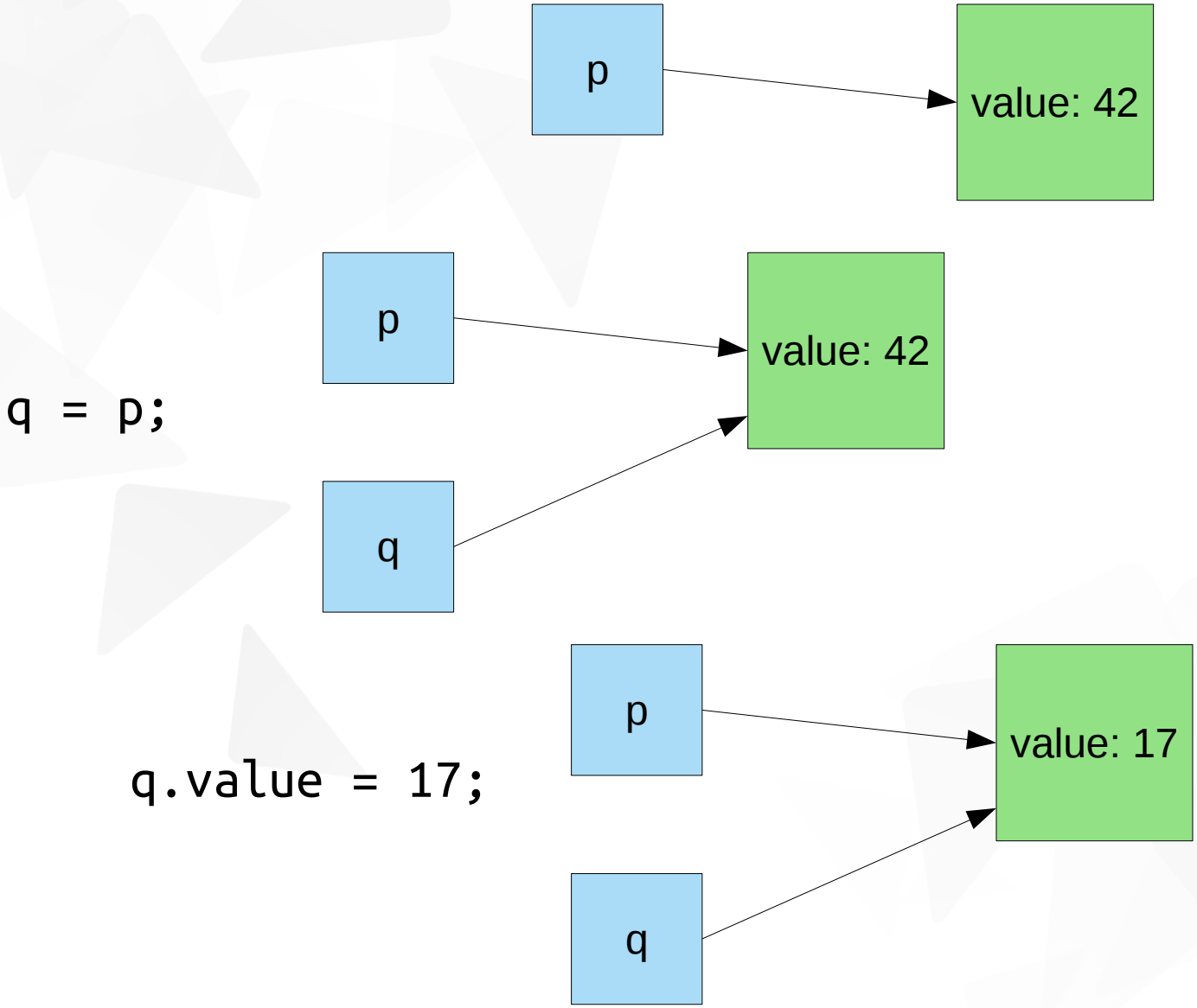

```
p = new MyClass()
```




```
p.value = 42
```



Walkthrough



What happens?

```
class Pair {
   public int a;
   public String b;
}
```

```
Pair p, q, r;

p = new Pair();
p.a = 5;
p.b = “Time”;

q = p;

r = new Pair();
r.a = 12;
r.b = “Space”;

p.a = r.a;
p.b = r.b;

What is q.b?
```

Constructors

```
class Pair {
   private int damage;
   private String name;

   public Pair(int v) {
      damage = v * 10;
      name = “Phasers”;
   }
}
```

defining a constructor

```
 Pair w = new Pair(12);
```

creating a new instance of the class

```
 What is w.damage ?
```

Java Exceptions

- Three errors walk into a bar. The barman says, “Normally I’d throw you all out, but tonight I’ll make an exception.”
  – *@iamdevloper*

Exceptions are about control flow

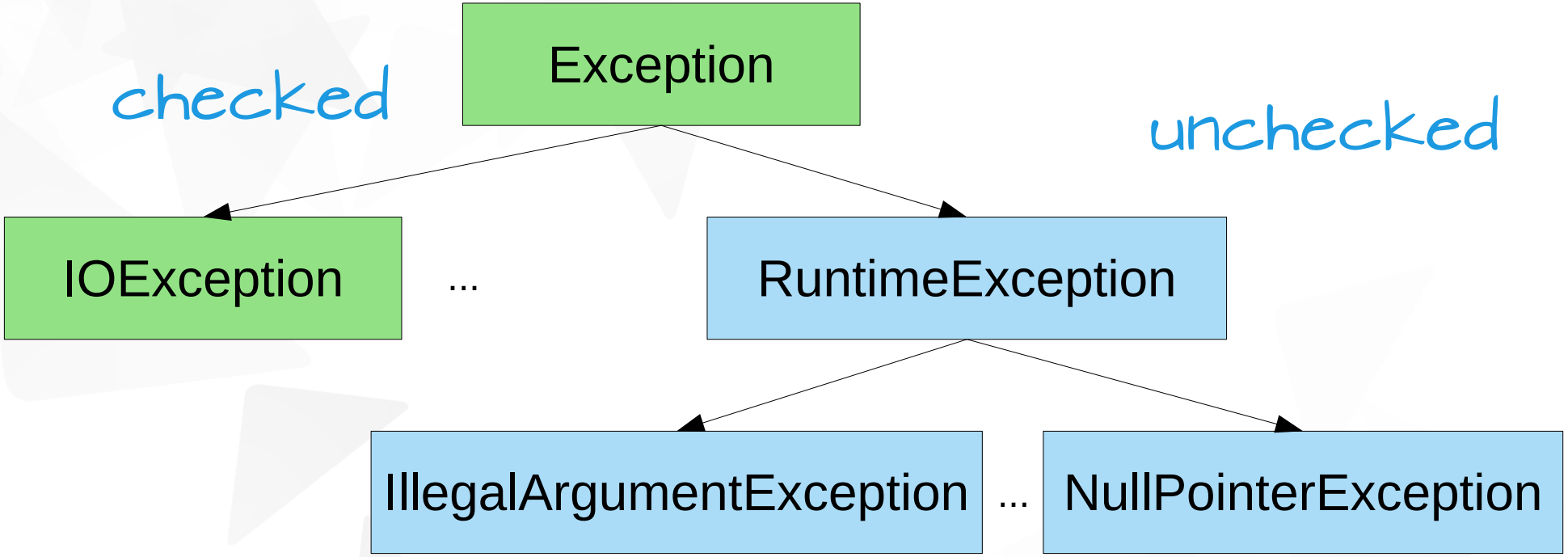
- Exceptions change the flow of control
  - What do you do when errors happen?
  - Can't keep going on like normal...
  - Transfer control somewhere else
- Default exception handler
  - Stop the program
  - Print out the exception
  - Show a stack trace for debugging

Generating Exceptions

- Generating exceptions
  - Try to call methods on a null object
  - Try to access variables on a null object
  - Go into infinite recursion
  - Divide by 0
  - Lots of ways to screw up...
- Or intentionally throw an exception
  - throw <*ExceptionObject*>

```
throw new Exception();
throw new RuntimeException();
throw new IllegalArgumentException();
```

Checked vs. Unchecked Exceptions



- Checked
  - “Invalid conditions outside immediate control of program”
- Unchecked
  - “Conditions that reflect errors in program logic”

Checked Exceptions

```
public int pop() throws Exception {
   if (error()) {
      throw new Exception();
   }
   return arr[top--];
}

public void silly() throws Exception {
   int i = pop();
}
```

unchecked

don't need this

Unit Testing

- “The best TDD can do, is assure that code does what the programmer thinks it should do. That is pretty good BTW.”

  – *James Grenning*

Unit Testing

- Testing is ***critical*** to creating software
  - "If it's not tested, it's broken."
- Unit testing
  - Small tests of individual components
  - Make sure each piece works before combining
  - Find problems earlier
  - Easier to make changes
    - Cleanup
    - Refactor
    - Reimplement
    - Optimize

Test Driven Development

- Write a failing test
- Write code to make test pass
- Refactor to cleanup code

- Observations
  - You end up writing more tests
  - More tests correlates with higher productivity
  - It takes longer
  - Easier to fix later

“Folk wisdom in software development teaches that interfaces shouldn't be unduly influenced by implementations. Writing a test first is a concrete way to achieve this separation.”

— *Kent Beck*

http://en.wikipedia.org/wiki/Test-driven_development

JUnit Tests

```
@Test
public void exprTest() {
    Calc c = new Calc();
    c.push(2.0);
    c.push(3.0);
    c.multiply();
    c.push(4.0);
    c.push(5.0);
    c.multiply();
    c.add();
    assertEquals(26.0, c.pop(), EPSILON);
}
```

Linked Lists

- "You can either have software quality or you can have pointer arithmetic, but you cannot have both at the same time."

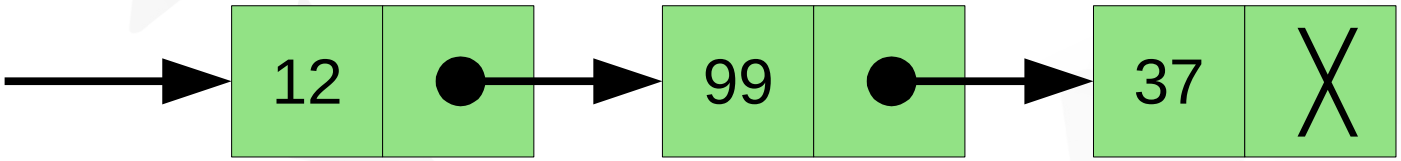
  Bertrand Meyer

What is a linked list?

- Linked lists are versatile data structure
  - 2nd most common after arrays/vectors
- Chain elements together with arrows going from one to the next
  - Can rearrange arrows to insert/delete anywhere in list
    - No shifting required 😄
  - Give up the ability to directly index into list ☹

Singly Linked List

- Three element list of integers
- Singly-linked
  - Each node has a value and a next node
  - Or next node is *null*, represented by X



Node

```
class Node {
    public int value;
    public Node next;
}
```

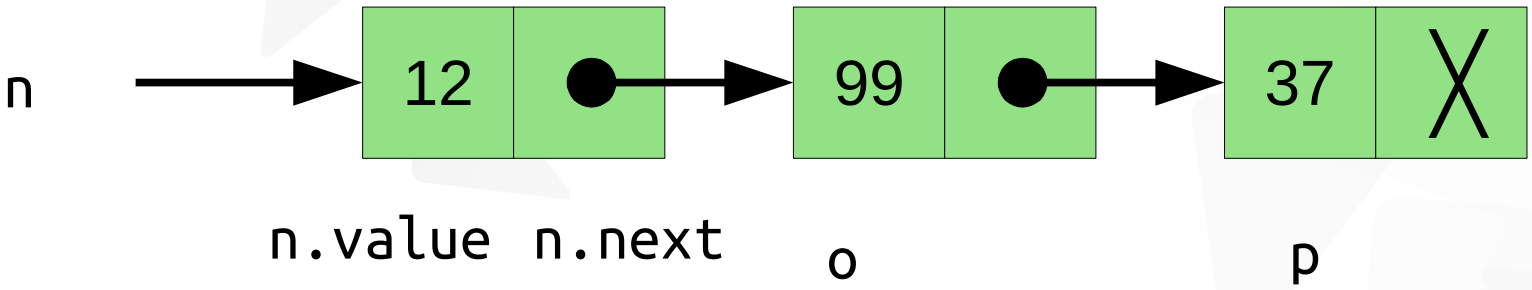
kind of like recursion

defining a class in terms of itself

Building a List

```
class Node {
   public int value;
   public Node next;
}
```

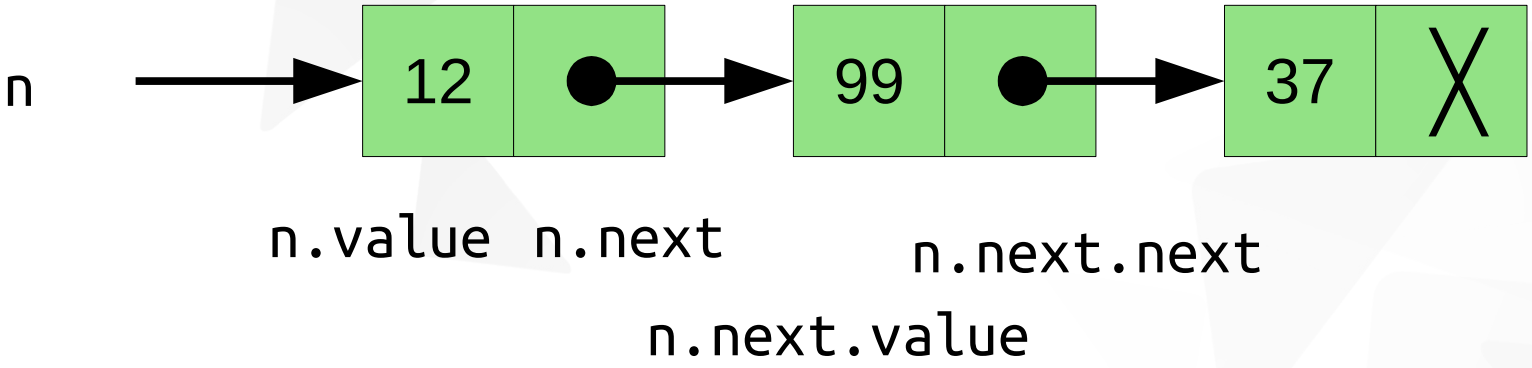
```
Node p = new Node();
p.value = 37;
p.next = null;
Node o = new Node();
o.value = 99;
o.next = p;
Node n = new Node();
n.value = 12;
n.next = o;
```



Building a List

```
class Node {
   public int value;
   public Node next;
}
```

```
Node n = new Node();
n.value = 12;
n.next = new Node();
n.next.value = 99;
n.next.next = new Node();
n.next.next.value = 37;
n.next.next.next = null;
```



Linked List Class

```
class Node {
   public int value;
   public Node next;
}
```

```
class LinkList {
   private Node first;

   public LinkList() {
      first = null;
   }

   public void insertFirst(int value) {
      ...
   }
   .
   .
   .
}
```

Linked List Class - insertFirst

```
class Node {
    public int value;
    public Node next;
}
```

```
class LinkList {
    private Node first;

    public LinkList() {
        first = null;
    }

    public void insertFirst(int value) {
        ...
    }
    .
    .
    .
}
```

implement insertFirst

1. Draw node diagram for each line of test
2. Implement insertFirst

test

```
LinkList lst = new LinkList();
lst.insertFirst(37);
lst.insertFirst(99);
lst.insertFirst(12);
```

Linked List Class - deleteFirst

```
class Node {
    public int value;
    public Node next;
}
```

```
class LinkList {
    private Node first;

    public LinkList() {
        first = null;
    }

    public void deleteFirst() {
        ...
    }
    .
    .
    .
}
```

implement deleteFirst

test

1. Draw node diagram for each line of test
2. Implement deleteFirst

```
LinkList lst = new LinkList();
lst.insertFirst(12);
lst.insertFirst(5);
lst.insertFirst(8);
lst.deleteFirst();
lst.deleteFirst();
```

Linked List Class - find

```
class Node {
    public int value;
    public Node next;
}
```

```
class LinkList {
    private Node first;

    public LinkList() {
        first = null;
    }

    public Node find(int key) {
        ...
    }
    .
    .
    .
}
```

implement find

test

1.. Implement find

2. Check against test

```
LinkList lst = new LinkList();
lst.insertFirst(17);
lst.insertFirst(90);
lst.insertFirst(22);
Node n90 = lst.find(90);
Node n17 = lst.find(17);
Node n23 = lst.find(23);
```

Linked List Class - insertLast

```
class Node {
    public int value;
    public Node next;
}
```

```
class LinkList {
    private Node first;

    public LinkList() {
        first = null;
    }

    public void insertLast(int value) {
        ...
    }
    .
    .
    .
}
```

implement insertLast

1. Draw node diagram for each line of test
2. Implement insertLast

test

```
LinkList lst = new LinkList();
lst.insertFirst(100);
lst.insertLast(216);
lst.insertFirst(8);
lst.insertLast(3);
```

Methods of Linked Lists

- *isEmpty* - Check if list is empty
- *insertFirst* - Insert new value at start of list
- *insertLast* - Insert new value at end of list
- *deleteFirst* - Delete value at start of list
- *find* - Find a given value in the list
- *delete* - Delete a value somewhere in the list

Singly-Linked List Efficiency

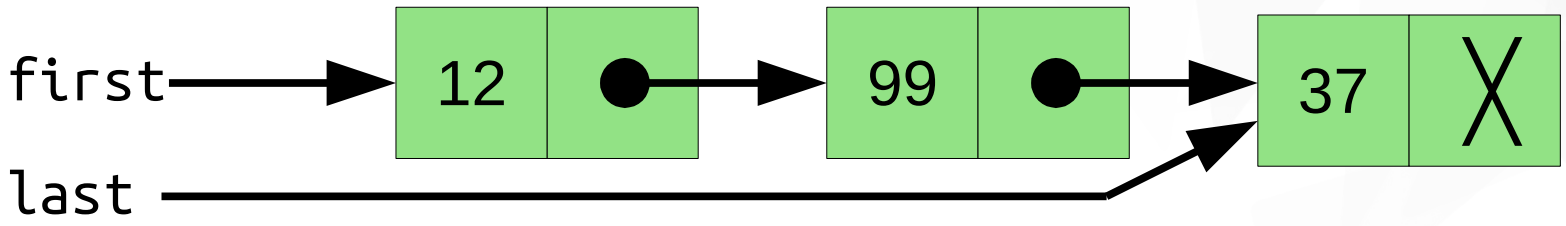
- Insert/delete at start
  - Fast
  - O(1)
- Insert/delete at end
  - Slow
  - O(*n*)
- Find element
  - Linear scan
  - O(*n*)
- Advantage versus arrays
  - Grows and shrinks as needed
  - No need to double size of array when out of room

hmm, maybe I can make a good stack out of this...

Double Ended Lists

- *insertLast* seems very inefficient
  - Idea: keep track of first and last node

```
class DoubleEndedLinkList {
    private Node first, last;

    public LinkList() {
        first = null;
        last = null;
    }

    public void insertLast(int value) {
        Node n = new Node();
        n.value = value;
        if (first == null) {
            first = n;
        } else {
            last.next = n;
        }
        last = n;
    }
}
```



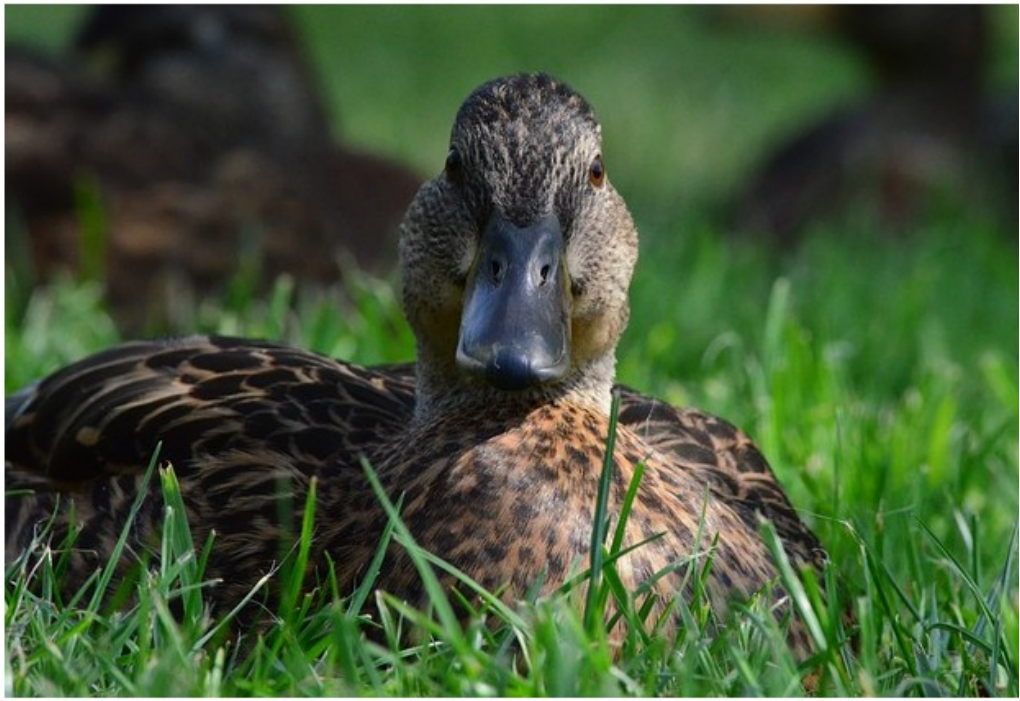
Efficiency of Double Ended Linked List

- Insert/delete at start
  - Fast
  - O(1)
- Insert at end
  - Fast
  - O(1)
- Find element
  - Linear scan
  - O(*n*)

hmm, maybe I can make a good queue out of this...

Abstract Data Types

- ADT
  - Way of looking at data structures
  - Only care about interface, abstract away implementation
- If it walks like a duck and quacks like a duck, it is a duck
  - If it has *push* and *pop,*
  - and they behave correctly,
  - it is a stack!

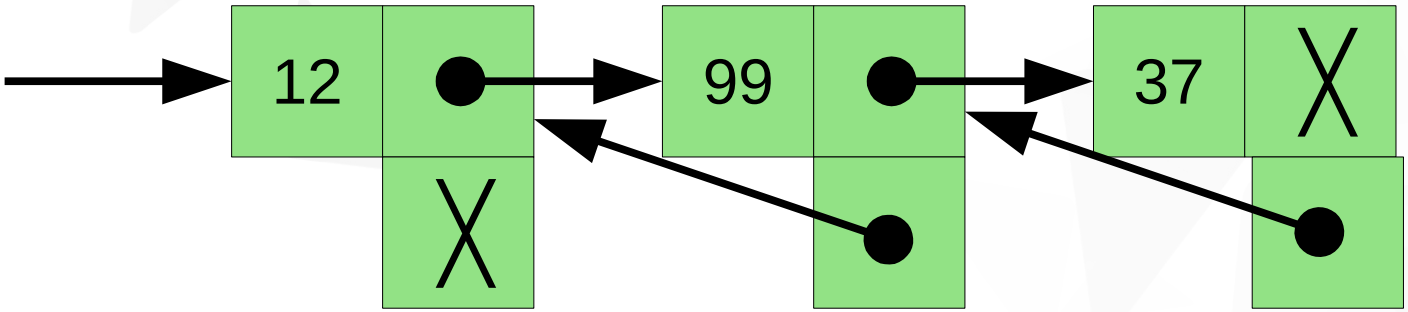


Alternate Implementations

- How to implement a stack
  - Use an array (Lab 3)
  - Or use a singly-linked list
    - Examples/Chap05/LinkStack/linkStack.java
- How to implement a queue
  - Use a circular array
  - Or use a double-ended linked list
    - Examples/Chap05/LinkQueue/linkQueue.java

Doubly-Linked Lists

- Wouldn't it be nice to be able to go forwards and backwards any time we wanted?



Doubly-Linked List Node

```
class Node {
   public int value;
   public Node next;
   public Node prev;
}
```

Doubly-Linked List

- Can go forward and backward from any node
- Operations need to do more bookkeeping
- Given reference to a node, can finally delete that node itself!
  - Couldn't do this before with singly-linked
  - Previous node has next, needs to be updated during delete

The End