CS 367 - Introduction to Data Structures
Week 2, Summer 2017

Homework h1: Due by today (Monday June 26th) 10 pm
Homework h2 posted, complete as soon as possible; due 10pm Sunday, July 2nd
Program 1: Will be released soon. Look for a teammate. Team registration will open soon.

Assignment questions? Discuss within your CS367 Team, post on Piazza, or consult with a TA during scheduled hours.

Report any exam conflicts or McBurney exam accommodations by this Thursday 06/29.

Email your instructor by this Friday, 06/29, if you participate in religious observances that might interfere with course requirements. Include your name, UW ID#, date and explanation.

Last Week

Using lists via the ListADT, implementing ListADT using an array (SimpleArrayList), Java's List interface, Iterators: concept, using, options for implemention, making a class Iterable

This Week

Read: *Exceptions*, *Linked Lists*

Exceptions Review

- throwing
- handling
- execution
- practice with exception handling
- throws and checked vs. unchecked
- defining

Java Primitives vs. References Review, p.14

Chains of Linked Nodes

- Listnode class
- practice with chains of nodes
- LinkedList Class
- LinkedListIterator Class
- Linked List Variations
  - tail reference
  - header node
  - double linking
  - circular linking

Next Week

Read: finish *Linked Lists* and start *Complexity*

Concept, big-O notation, analyzing algorithms practice, analyzing Java code

Exception Throwing – Signaling a Problem

Java Syntax

```
     throw exceptionObject;
```

Example

Exception Handling – Resolving a Problem

Java Syntax

```
try {
   // try block
   code that might cause an exception to be thrown

} catch (ExceptionType1 identifier1) {
   // catch block
   code to handle exception type 1

} catch (ExceptionType2 identifier2) {
   // catch block
   code to handle exception type 2

}
... more catch blocks

finally {
   // finally block - optional
   code always executed when try block is entered
}
```

Example

Exception Execution

Normal Execution

- Start: top of main()

- Execute:

- Skip:

- Switch to Exception Handling Execution

Exception Handling Execution

- Skip:

- Execute:

- Switch back to Normal Execution

Searching for a Matching Catch

1. Locally

2. Remotely

Checking a Match

1. Match Found

2. No Match Found

ExceptionTester

```
public class ExceptionTester {

   public static void main(String[] args) {
      System.out.print("main[");
      try {
         methodA( ); System.out.print("after A,");
         methodE( ); System.out.print("after E,");
      } catch (RedException exc) {
         System.out.print("main-red,");
      } catch (GreenException exc) {
         System.out.print("main-green,");
      } finally {
         System.out.print("main-finally,");
      }
      System.out.println("]main");
   }

   private static void methodA( ) {
      System.out.print("\nA[");
      try {
         methodB( );
         System.out.print("after B,");
      } catch (BlueException exc) {
         System.out.print("A-blue,");
      }
      System.out.println("]A");
   }

   private static void methodB( ) {
      System.out.print("\nB[");
      methodC( );
      System.out.print("after C,");
      try {
         methodD( );
         System.out.print("after D,");
      } catch (YellowException exc) {
         System.out.print("B-yellow,");
         throw new GreenException();
      } catch (RedException exc) {
         System.out.print("B-red,");
      } finally {
         System.out.print("B-finally,");
      }
      System.out.println("]B");
   }
```

What is Output When:

1. no exception is thrown

```
main[
A[
B[
```

2. `methodE` throws a `YellowException`?

```
main[
A[
B[
```

3. `methodC` throws a `GreenException`?

```
main[
A[
B[
```

4. `methodD` throws a `GreenException`?

```
main[
A[
B[
```

What is Output When:

5. **`methodC` throws a `RedException`?**

```
main[
A[
B[
```

6. **`methodD` throws a `RedException`?**

```
main[
A[
B[
```

7. **`methodD` throws a `YellowException`?**

```
main[
A[
B[
```

8. **`methodD` throws a `OrangeException`?**

```
main[
A[
B[
```

What is Output When:

9. `methodC` throws a `YellowException`?

```
main[
A[
B[
```

10. `methodC` throws a `BlueException`?

```
main[
A[
B[
```

11. `methodE` throws a `RedException`?

```
main[
A[
B[
```

`throws` clause – Passing the Buck

Checked Exceptions vs. Unchecked Exceptions

Java Syntax

```
... methodName(parameter list)
        throws ExceptionType1, ExceptionType2, ... {
    ...
}
```

Example

```
public static void main(String[] args) throws IOException { ...
```

Defining a New Exception Class

Checked

```
public class MyException extends ____________________ {


}
```

Unchecked

```
public class MyException extends ____________________ {


}
```

Example (if you want to support an optional message)

```
public class EmptyBagException extends Exception {

    public EmptyBagException() {
        super();
    }

    public EmptyBagException(String msg) {
        super(msg);
    }

}
```

Reference Types: Assignment

References

assume code is in main() Call Stack | Heap

```
ArrayList<String> x, y, z;
x = new ArrayList<String>();
y = x;
z = x;
y = new ArrayList<String>();
z.add("Computer");
y.add("Science");
```

→ **What does each ArrayList contain after the code above executes?**

x's ArrayList has y's ArrayList has z's ArrayList has

→ **What do x, y and z contain?**

Primitive vs. Reference Types: Parameter Passing

Primitives

Call Stack | Heap

Given:

```
void mod1(int x) {
   x = 42;
}
```

Execute code in main():

```
int   x = 11;
int[] y = {11, 22, 33};
mod1(x);
mod1(y[2]);
```

→ **What does variable x and array y in main contain after the code above executes?**

x has y's array has

→ **What happens if we call mod1(y) in main?**

Reference Types: Parameter Passing and Return Values

References

Call Stack | Heap

Given:

```
void mod2(int[] x) {
   x[0] = 21;
}
void mod3(int[] x) {
   x = new int[x.length];
   x[0] = 42;
}
```

Execute code in `main()`:

```
int   x = 11;
int[] y = {11, 22, 33};
mod2(y);
mod3(y);
```

→ **What does variable `x` and array `y` in `main` contain after the code above executes?**

`x` has   `y`'s array has

→ **What happens if we call `mod2(x)` in main?**

References

Call Stack | Heap

Given:

```
int[] mod2(int[] x) {
   x[0] = 21;
   return x;
}

int[] mod3(int[] x) {
   x = new int[x.length];
   x[0] = 42;
   return x;
}
```

Execute code in `main()`:

```
int   x = 11;
int[] y = {11, 22, 33};
mod2(y);
y = mod3(y);
```

→ **What does variable `x` and array `y` in `main` contain after the code above executes?**

`x` has   `y`'s array has

Programmer’s Memory Model for Java

Call Stack

Contains: activation record (stack frame) for each method that is called

Birth: created when program starts
Death: ends when program ends

Note: each method’s activation record starts when method is called and ends when method returns

Heap

Contains: objects (arrays are objects), need a reference to use it.

Memory allocation grows: when new objects are created, and also by initializer lists {1,2,3}, and when primitives are auto-wrapped.

Memory allocation shrinks: when unreferenced objects are garbage collected

Birth: when object is created with “new”
Death: when no references remain that point to the object, they are garbage collected.

Static Data

Contains: literal values like 13, “abc”, “13”, …
class (static) variables and constants

Birth: at program start
Death: at program end

Code

Contains: the program’s instructions
Birth: at program start
Death: at program end

New Data Structure - Chain of Linked Nodes

The Data Structure

Array vs. **Chain of Nodes**

Goal

`Listnode` Class

```
class Listnode<E> {

   private E data;
   private Listnode<E> next;

   public Listnode(E d) {
      this(d, null);
   }

   public Listnode(E d, Listnode<E> n) {
      data = d;
      next = n;
   }

   public E getData()                  { return data; }
   public Listnode<E> getNext()        { return next; }

   public void setData(E d)            { data = d; }
   public void setNext(Listnode<E> n)  { next = n; }
}
```

Practice: Using Listnodes

→ **Draw a memory diagram** corresponding to the given code:

assume code is in main()   Call Stack | Heap

```
Listnode<String> n1 = null;

Listnode<Integer> n2 =
   new Listnode<Integer>(11);

Listnode<String> n3 =
   new Listnode<String>("list",n2);

Listnode<String> n4 =
   New Listnode<String>("" + 11);
```

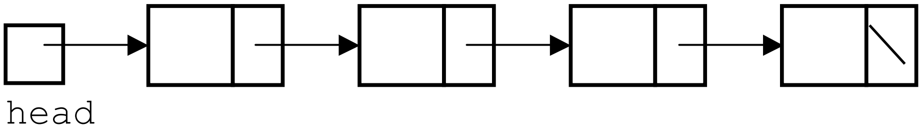
→ **Write the code that results in:**

Recall Chain of Linked Nodes Data Structure

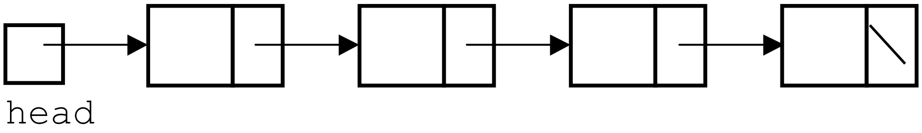
`Listnode` class

```
class Listnode<E> {

   private E data;
   private Listnode<E> next;

   public Listnode(E d)               { . . . }
   public Listnode(E d, Listnode<E> n){ . . . }

   public E getData()                 { return data; }
   public Listnode<E> getNext()       { return next; }
   public void setData(E d)           { data = d; }
   public void setNext(Listnode<E> n) { next = n; }
}
```

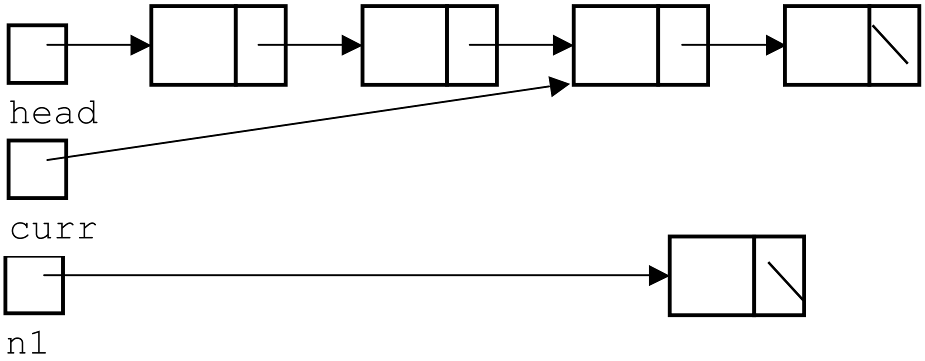
→ **Show how the memory diagrams change** as a result of executing the code beneath each:



```
head.setNext(head.getNext().getNext().getNext().getNext());
```



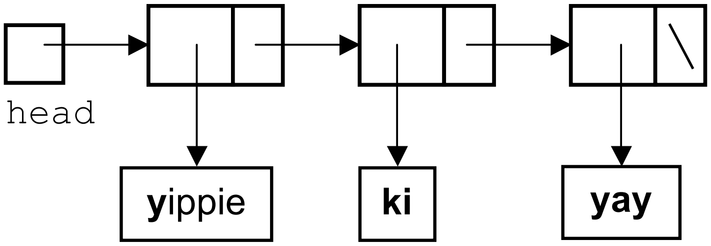
```
head.getNext().getNext().setNext(head);
```



```
n1.setNext(curr.getNext());
curr.setNext(n1);
```

Practice: Making a Chain of Nodes

→ **Create a chain of `Listnodes`** containing the **`String`**s
"yippie", "ki", and "yay" (as shown below) in as few statements as you can.



-

Practice: Traversing a Chain of Nodes

Assume **`head`** points to the first node in a chain of **`Listnode`**s containing **`String`**s.

→ **Write a code fragment** that counts the number of strings in the chain of nodes.

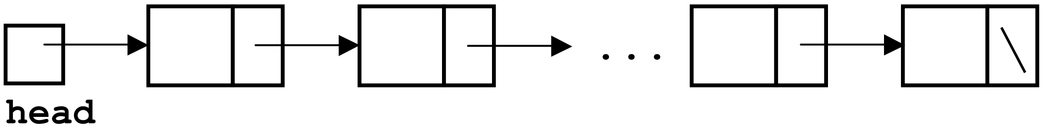
```
int count = 0;
```

-

Practice: Adding a Node at the Chain's End

Assume **`head`** points to the first node in a chain of **`Listnode`**s containing **`String`**s.

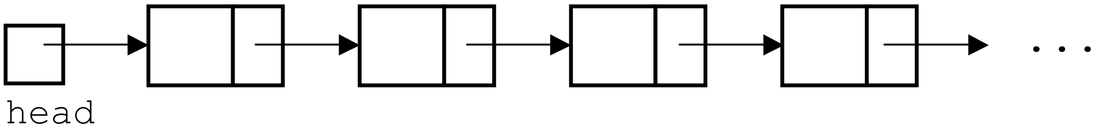
→ **Write a code fragment** that adds a node containing "rear" to the end of the chain of nodes.
You may assume the chain has at least one item.



Practice: Removing a Node from a Chain

Assume **head** points to the first node in a chain of **Listnode**s containing **String**s.

→ **Write a code fragment** that removes the third item from the chain of nodes.
You may assume the chain has at least three items.



→ **How would you generalize your code** so it removes the Nth item from the chain of nodes?

Copyright 2014-2016 Jim Skrentny, 2017 Deb Deppeler, Meena Syamkumar

Practice: Challenge Question

Assume **`head`** points to the first node in a chain of **`Listnode`**s containing **`String`**s.

→ **Write a code fragment** that reverses the order of the nodes in the chain.

Java Visibility Modifiers

public	`public class ArrayList<E>`
private	`private Object[] items`
protected	`protected String name`
package	`class Listnode<E>`

Recall the List ADT

Concept

A List is a general, position-oriented container that stores a contiguous collection of items where duplicates are allowed. It maintains relative ordering and uses zero-based indexing.

Operations

```
void add(E item);
void add(int pos, E item);
E get(int pos);
E remove(int pos);
boolean contains(E item);
int size();
boolean isEmpty();
```

Issues

Null item – detect then signal with `IllegalArgumentException`
Bad position – detect then signal with `IndexOutOfBoundsException`
Empty list – handle as a bad position

`LinkedList` - Implementing `ListADT` using a Chain of Nodes

```
public class LinkedList<E> implements ListADT<E> {

  private Listnode<E> head;
  private int numItems;


  public LinkedList() {




  }

  public void add(E item) {
```

LinkedList (cont.)

```
public class LinkedList<E> implements ListADT<E> {

  private Listnode<E> head;
  private int numItems;

  public LinkedList() { ... }
  public void add(E item) { ... }


  public E get(int pos) {
```

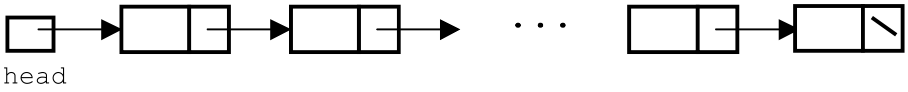
Header Node

Concept

-

empty

non-empty



-
-

Code Example

```
public class LinkedList<E> implements ListADT<E> {

   private Listnode<E> head;
   private int numItems;

   public LinkedList() {
      head = null;
      numItems = 0;
   }

   public void add(E item) {
      if (item == null) throw new IllegalArgumentExceptions();

      Listnode<E> newnode = new Listnode<E>(item);

      //Special Case: empty list
      if (head == null) {
         head = newnode;
      }
      //General Case: non-empty list
      else {
         Listnode<E> curr = head;
         while (curr.getNext() != null)
            curr = curr.getNext();
         curr.setNext(newnode);
      }

      numItems++;
   }
```

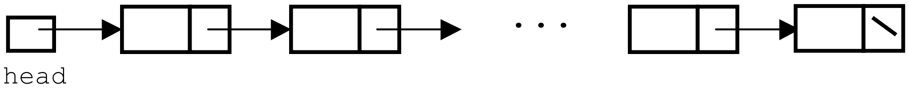
Tail Reference

Concept

-

empty

non-empty

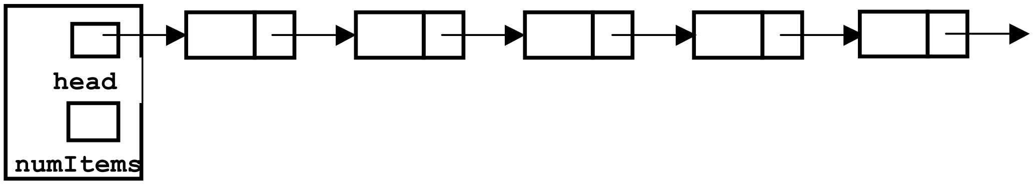


Code Example

```
public class LinkedList<E> implements ListADT<E> {

   private Listnode<E> head;

   private int numItems;

   public LinkedList() {
      head = null;

      numItems = 0;
   }

   public void add(E item) {
      if (item == null) throw new IllegalArgumentExceptions();

      Listnode<E> newnode = new Listnode<E>(item);

      //Special Case: empty list
      if (head == null) {
         head = newnode;

      }
      //General Case: non-empty list
      else {
         Listnode<E> curr = head;
         while (curr.getNext() != null)
            curr = curr.getNext();
         curr.setNext(newnode);
      }

      numItems++;
   }
```

Implementing `LinkedListIterator`

→ **Should an indirect or a direct iterator implementation be used with a `LinkedList`?**



```
import java.util.*;

public class LinkedListIterator<E> implements Iterator<E> {



   LinkedListIterator(                              ) {


   }
   public boolean hasNext() {


   }
   public E next() {
       if (                    ) throw new NoSuchElementException();





   }
   public void remove() {
       throw new UnsupportedOperationException();
   }
}
```

Making `LinkedList` Iterable

```
public class LinkedList<E> implements ListADT<E> {

  private Listnode<E> head;
  private int numItems;


  public LinkedList() { ... }
  public void add(E item) { ... }
  public E get(int pos) { ... }
    .
    .
    .

  public Iterator<E> iterator() {



  }

}
```