CSSE 220 Day 28

Threads and Animation
Lists, Collections, and Iterators

Check out *ThreadsIntro* project from SVN

Vector Graphics Grading

- On the grading sheet, it says that the Required Features are worth 30 points
  - I scaled it to 50 because I had said that the Required Features would be worth at least 75%
- If I missed something that your program does correctly, please talk with me about it

Questions

Multithreaded programs

- Often we want our program to do multiple (semi) independent tasks at the same time
- Each thread of execution can be assigned to a different processor, or one processor can simulate simultaneous execution through "time slices" (each typically a large fraction of a millisecond)

Time → Slices	1	2	3	4	5	6	7	8	9	10	11	12	13	14
running thread 1	■	■		■				■		■			■	■
running thread 2			■		■	■	■		■		■	■		

Q1-2

A Java Program's Threads

- There is always one default thread; you can create others
- Uses for additional threads:
  - Animation that runs while still allowing user interaction
  - A server (such as a web server) communicates with multiple clients
  - Animate multiple objects (such as the timers in the soon-to-be-seen **CounterThreads** example)
- A thread may suspend execution for (approximately) a specified amount of time by calling

```
Thread.sleep(numberOfMilliseconds);
```

The Emperor's New Threads

- How to create and run a new thread
  1. Define a new class that implements the **Runnable** interface: (it has one method: `public void run();`)
  2. Place the code for the threaded task in the `run` method:
     - ```
       class MyRunnable implements Runnable {
           public void run () {
               // task statements go here
           }
       }
       ```
  3. Create an object of this class:
     - `Runnable r = new MyRunnable();`
  4. Construct a Thread object from this Runnable object:
     - `Thread t = new Thread(r);`
  5. Call the **start** method to start the thread:
     - `t.start();`

Q5
       ```

Threads examples (in your SVN repos.)

- **Greetings** –simple threads, different wait times
- **AnimatedBall** – move balls, stop with click
- **CounterThreads** – multiple independent counters
- **CounterThreadsRadioButtons** – same as above, but with radio buttons

The remaining two are more advanced than we will use in this course, dealing with race conditions and synchronization. Detailed descriptions are in *Big Java* Chapter 20

- **BankAccount**
- **SelectionSorter**

Simple example (1) – greetings Output

One thread prints the **Hello** messages; the other Thread prints the **Goodbye** messages.

Each thread sleeps for a random amount of time after printing each line.

```
Thu Jan 03 16:09:36 EST 2008 Hello, World!
Thu Jan 03 16:09:36 EST 2008 Goodbye, World!
Thu Jan 03 16:09:36 EST 2008 Hello, World!
Thu Jan 03 16:09:36 EST 2008 Goodbye, World!
Thu Jan 03 16:09:36 EST 2008 Goodbye, World!
Thu Jan 03 16:09:36 EST 2008 Hello, World!
Thu Jan 03 16:09:37 EST 2008 Goodbye, World!
Thu Jan 03 16:09:37 EST 2008 Hello, World!
Thu Jan 03 16:09:38 EST 2008 Hello, World!
Thu Jan 03 16:09:38 EST 2008 Goodbye, World!
Thu Jan 03 16:09:38 EST 2008 Goodbye, World!
Thu Jan 03 16:09:38 EST 2008 Hello, World!
Thu Jan 03 16:09:39 EST 2008 Goodbye, World!
Thu Jan 03 16:09:39 EST 2008 Goodbye, World!
Thu Jan 03 16:09:39 EST 2008 Goodbye, World!
Thu Jan 03 16:09:39 EST 2008 Hello, World!
Thu Jan 03 16:09:39 EST 2008 Hello, World!
Thu Jan 03 16:09:39 EST 2008 Goodbye, World!
Thu Jan 03 16:09:40 EST 2008 Hello, World!
Thu Jan 03 16:09:40 EST 2008 Goodbye, World!
. . .
```

This example was adapted from Cay Horstmann's *Big Java 3ed*, Chapter 20

Simple example(2) – GreetingThreadTester

```
public class GreetingThreadTester{

  public static void main(String[] args){

    // Create the two Runnable objects
    GreetingRunnable r1 = new GreetingRunnable("Hello, World!");
    GreetingRunnable r2 = new GreetingRunnable("Goodbye, World!");

    // Create the threads from the Runnable objects
    Thread t1 = new Thread(r1);
    Thread t2 = new Thread(r2);

    // Start the threads running.
    t1.start();
    t2.start();
   }
}
```

We do not call **run()** directly.
Instead we call **start()**, which sets up the thread environment and then calls **run()** for us.

Simple example(3) – a Runnable class

```
import java.util.Date;

public class GreetingRunnable implements Runnable {

   private String greeting;
   private static final int REPETITIONS = 15;
   private static final int DELAY = 1000;

   public GreetingRunnable(String aGreeting) {
      greeting = aGreeting;
   }

   public void run() {
      try {
         for (int i = 1; i <= REPETITIONS; i++){
            Date now = new Date();
            System.out.println(now + " " + greeting);
            Thread.sleep((int)(DELAY*Math.random()));
         }
      } catch (InterruptedException exception){
      }
   }
}
```

If a thread is interrupted while it is sleeping, an InterruptedException is thrown.

Q6

Ball Animation

- A simplified version of the way BallWorlds does animation
- When balls are created, they are given position, velocity, and color
- Our `run()` method tells each of the balls to move, then redraws them
- Clicking the mouse turns movement off/on
- Demonstrate the program

Set up the frame

```
public class AnimatedBallViewer {

   static final int FRAME_WIDTH = 600;
   static final int FRAME_HEIGHT = 500;

   public static void main(String[] args){
      JFrame frame = new JFrame();

      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setTitle("BallAnimation");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      AnimatedBallComponent component = new AnimatedBallComponent();
      frame.add(component);

      frame.setVisible(true);
      new Thread(component).start();
   }
}
```

This class has all of the usual stuff, plus this last line of code that starts the animation.

The Ball class

```
class Ball {
   private double centerX, centerY, velX, velY;
   private Ellipse2D.Double ellipse;
   private Color color;
   private static final double radius = 15;

   public Ball(double cx, double cy, double vx, double vy, Color c){
      this.centerX = cx;
      this.centerY = cy;
      this.velX = vx;
      this.velY = vy;
      this.color = c;
      this.ellipse = new Ellipse2D.Double (
          this.centerX-radius, this.centerY-radius,
          2*radius, 2*radius);
   }

   public void fill (Graphics2D g2) {
      g2.setColor(this.color);
      g2.fill(ellipse);
   }

   public void move (){
      this.ellipse.x += this.velX;
      this.ellipse.y += this.velY;
   }
}
```

Everything here should look familiar, similar to code that you wrote for BallWorlds.

AnimatedBallComponent: Instance Variables and Constructor

```
public class AnimatedBallComponent extends JComponent
                    implements Runnable, MouseListener {

   private ArrayList<Ball> balls = new ArrayList<Ball>();
   private boolean moving = true;
   public static final long DELAY = 30;
   public static final int ITERATIONS = 300;

   public AnimatedBallComponent() {
      super();
      balls.add(new Ball(40, 50, 8, 5, Color.BLUE));
      balls.add(new Ball(500, 400, -3, -6, Color.RED));
      balls.add(new Ball(30, 300, 4, -3, Color.GREEN));
      this.addMouseListener(this);
   }
```

Again, there should be no surprises here!

AnimatedBallComponent: run, paintComponent, mousePressed

```
public void run() {
     for (int i=0; i<ITERATIONS; i++) {
        if (moving){
           for (Ball b:balls)
              b.move();
           this.repaint();
        }
        try {
           Thread.sleep(DELAY);
        } catch (InterruptedException e) {}
     }
  }

  public void paintComponent(Graphics g){
     Graphics2D g2 = (Graphics2D)g;
     for (Ball b:balls)
        b.fill(g2);
  }

  public void mousePressed (MouseEvent arg0) {
     moving = !moving;
  }
```

Each time through the loop (if moving), tell each ball to move, then repaint

Sleep for a while

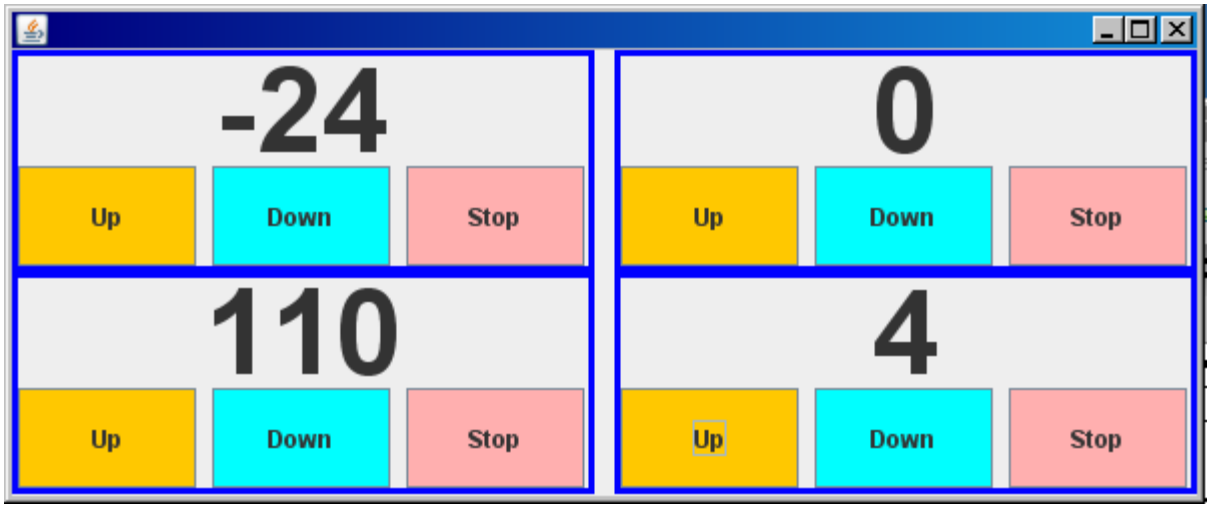
Draw each ball

Toggle "moving" when the mouse is pressed

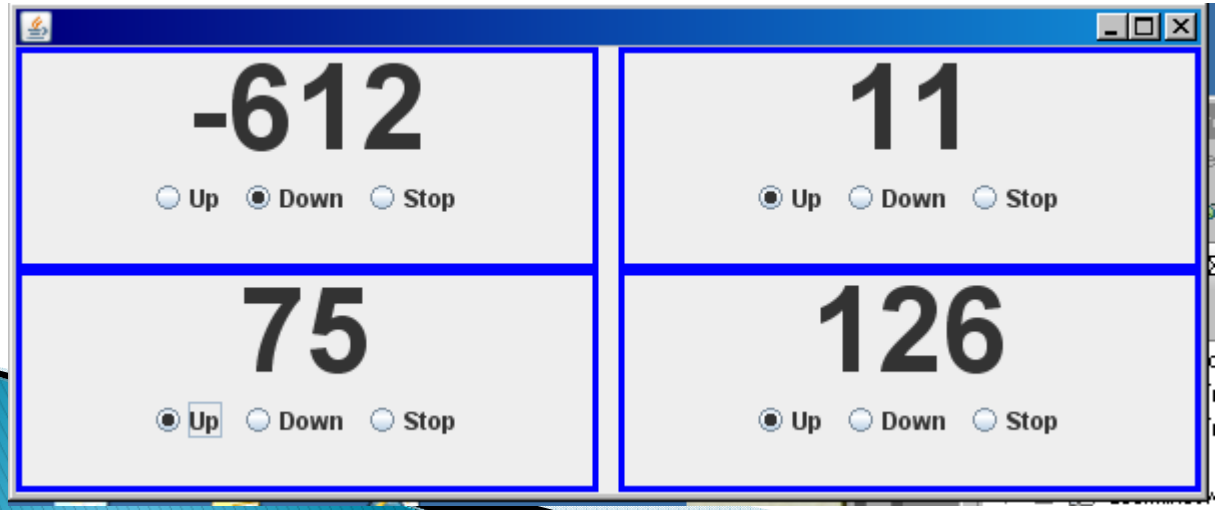
Q7

Another animation: CounterThreads

- With regular buttons



With radio buttons



How many threads does this application appear to have?

CounterThreads setup

```
public class CounterThreads {

  public static void main (String []args) {
    JFrame win = new JFrame();
    Container c = win.getContentPane();
    win.setSize(600, 250);
    c.setLayout(new GridLayout(2, 2, 10, 0));
    c.add(new CounterPane(200));
    c.add(new CounterPane(500));
    c.add(new CounterPane(50)); // this one will count fast!
    c.add(new CounterPane(1000));

    win.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    win.setVisible(true);
   }
}
```

Same old stuff!

CounterPane Basics

```
class CounterPane extends JComponent implements Runnable {

  private int delay;    // sleep time before changing counter
  private int direction = 0; //  current increment of counter
  private JLabel display = new JLabel("0");

  // Constants to define counting directions:
  private static final int COUNT_UP    =   1; // Declaring these
  private static final int COUNT_DOWN   = -1; // constants avoids
  private static final int COUNT_STILL  =  0; // "magic numbers"

  private static final int BORDER_WIDTH =  3;
  private static final int FONT_SIZE    = 60;
```

CounterPane Constructor

```
public CounterPane(int delay) {

    JButton upButton   = new JButton("Up");      // Note that these do
    JButton downButton = new JButton("Down");    // NOT have to be fields
    JButton stopButton = new JButton("Stop");    // of this class.

    this.delay = delay; // milliseconds to sleep

    this.setLayout(new GridLayout(2, 1, 5, 5));
       // top row for display, bottom for buttons.

    JPanel buttonPanel = new JPanel();
    buttonPanel.setLayout(new GridLayout(1, 3, 8, 1));
    display.setHorizontalAlignment(SwingConstants.CENTER);
    display.setFont(new Font(null, Font.BOLD, FONT_SIZE));
       // make the number display big!

    this.add(display);
    this.add(buttonPanel);
    this.setBorder(BorderFactory.createLineBorder(Color.blue,
                                                  BORDER_WIDTH));

    // Any Swing component can have a border.
    this.addButton(buttonPanel, upButton,   Color.orange, COUNT_UP);
    this.addButton(buttonPanel, downButton, Color.cyan,   COUNT_DOWN);
    this.addButton(buttonPanel, stopButton, Color.pink,   COUNT_STILL);

    Thread t = new Thread(this);
    t.start();
```

Put a simple border around the panel. There are also more complex border styles that you can use.

A lot of the repetitive work is done by the calls to **addButton()**.

CounterPane's addButton method

```
// Adds a control button to the panel, and creates an
// ActionListener that sets the count direction.
private void addButton(Container container,
                       JButton button,
                       Color color,
                       final int dir) {
  container.add(button);
  button.setBackground(color);
  button.addActionListener(new ActionListener () {
     public void actionPerformed(ActionEvent e) {
       direction = dir;
      }
  });
}
```

JPanel is a subclass of Container

The value of **dir** will be 1, -1, or 0, to indicate counting up, down, or neither.

- The action listener added here is an anonymous inner class that implements ActionListener.
- Because it is an inner class, its method can access this CounterPane's **dir** instance variable.

Note that each button gets its own ActionListener class, created at runtime. This is Swing's "preferred way" of providing ActionListeners.

CounterPane's run method

- This method is short and simple, because direction is always the amount to be added to the counter (1, –1, or 0).

```
public void run() {
    try {
      do {
        Thread.sleep(delay);
        display.setText(Integer.parseInt(display.getText())
                        + direction  + "");
      } while (true);
    } catch (InterruptedException e) { }
  }
}
```

CounterThreads questions

- Look through the code, discussing it with your partner and/or lab assistants until you think you understand it all. Answer the following questions:

1. How does a CounterPane know whether to count up or down or stay the same?
2. When a counter is not changing, does its thread use less CPU time than one that is changing?
3. Would it be easy to add code to the *main* method that creates a SuperStop button, so that clicking this button stops all counters? Explain.

RadioButton version

```
public CounterPaneRadio(int delay) {

    JRadioButton upButton   = new JRadioButton("Up");
    JRadioButton downButton = new JRadioButton("Down");
    JRadioButton stopButton = new JRadioButton("Stop");

    ButtonGroup group = new ButtonGroup();
    group.add(upButton);
    group.add(downButton);
    group.add(stopButton);
    stopButton.setSelected(true);
```

. . .

And we remove the **`Color`** parameter from **`addButton()`**

Ending a thread

- A thread t ends when its run method terminates.
- Threads used to have a stop method, but it is now deprecated.
- Instead of stopping a thread, you notify it that it should stop itself (return from its run method) by calling t.interrupt();
- The thread can check to see if it has been interrupted by calling this.isInterrupted();
- If so, the thread can decide to clean up and stop itself (or not).
- How does it stop itself?

Lists, Collections, Iterators

List

- A list is an ordered collection where elements may be added anywhere, and any elements may be deleted or replaced.
- **Array List**: Like an array, but growable and shrinkable.
- **Linked List**:

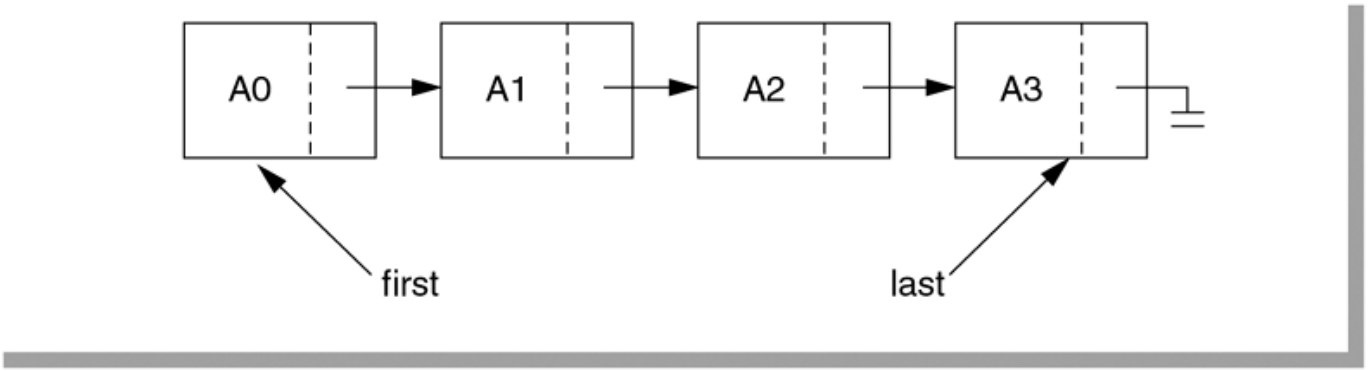


figure 6.19
A simple linked list

- Running time for add, remove, find?

Q12

List Usage Example

```
LinkedList<String> list = new LinkedList<String> ();
list.add("abc");
list.add("xyz");
list.add(1, "ddd");
list.add(2, "jkl");
System.out.println(list);
list.remove("ddd");
System.out.println(list);
list.remove(2);
System.out.println(list);
```

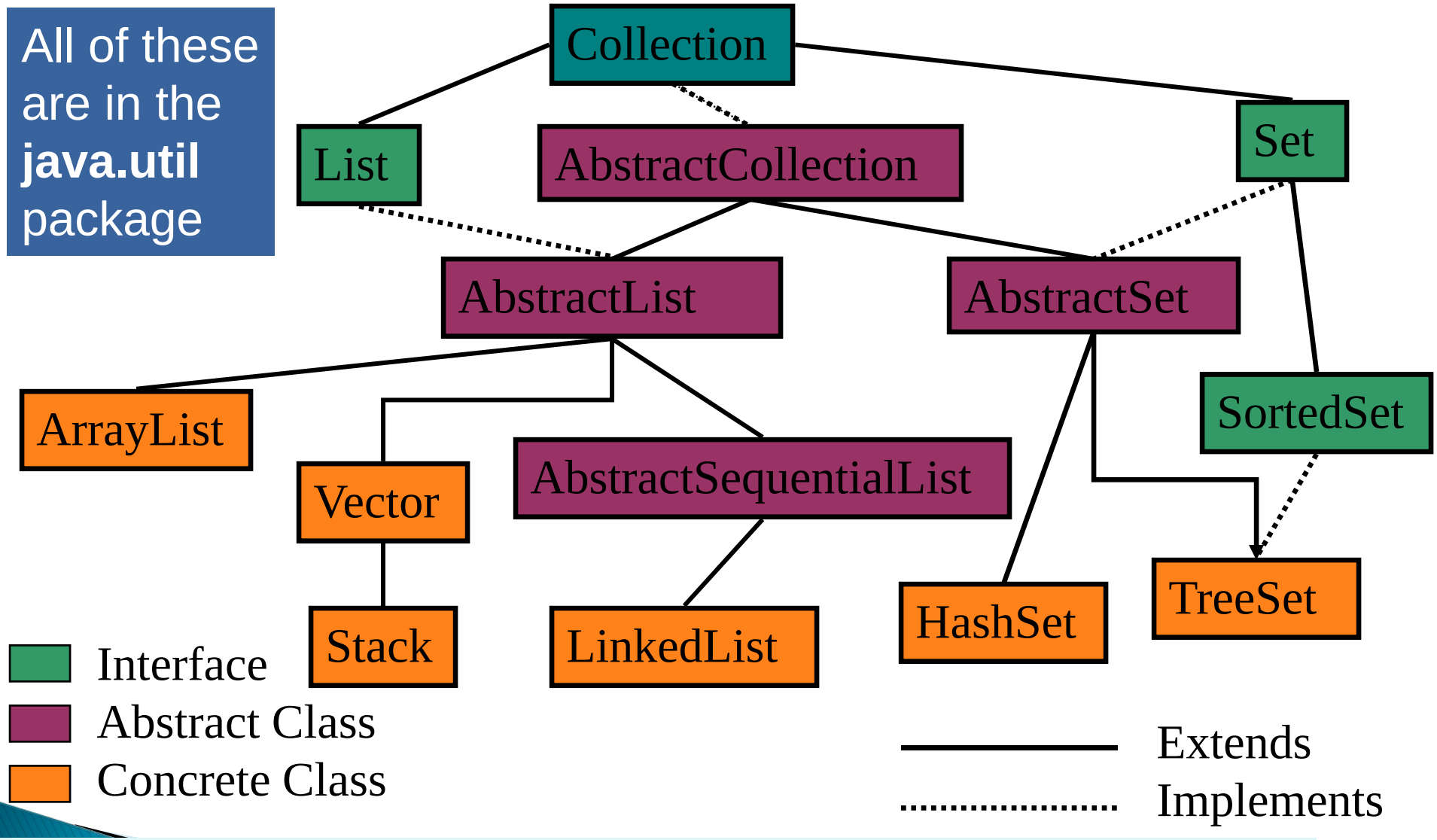
- Output:
- **[abc, ddd, jkl, xyz]**
  [abc, jkl, xyz]
  [abc, jkl]

Java Collections Framework

- From the Source:
  http://java.sun.com/docs/books/tutorial/collections/index.html
  That page and the three pages that you get by clicking Next three times are a very good introduction.
- Collections Framework provides several interfaces and classes to facilitate handling collections of objects.
- Closely related: `java.util.Map` interface.

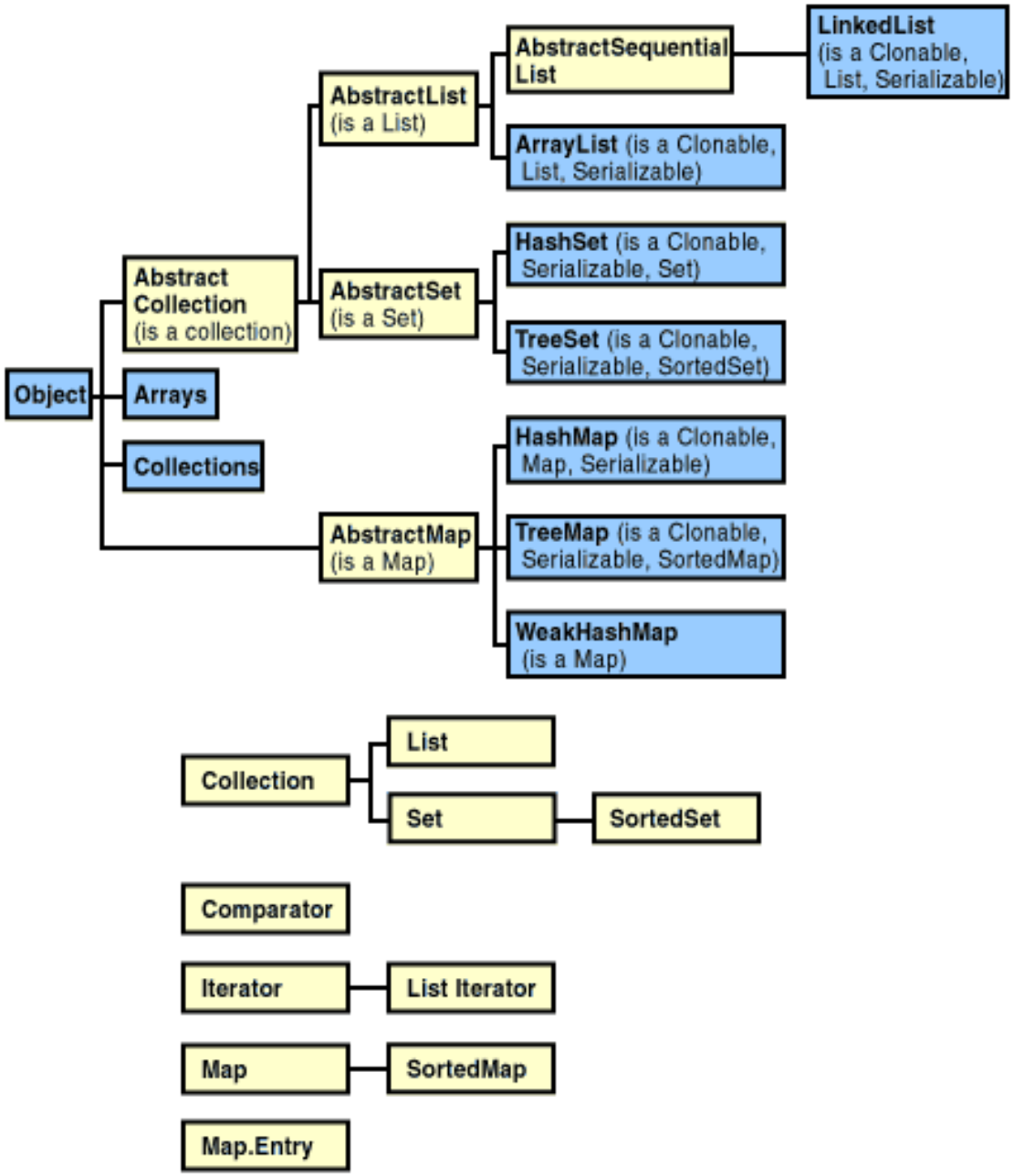
Some `Collection` interfaces and classes

All of these are in the **java.util** package



This is the Java 1.2 picture. Java 1.5 added **Queue**, **PriorityQueue**, and a few other interfaces and classes.

Collections classes and interfaces (classes at top, interfaces at bottom)



Some Methods From the Collection Interface

java.util

Interface Collection<E>

boolean	**add**(E o) Ensures that this collection contains the specified element (optional operation).
boolean	**contains**(Object o) Returns true if this collection contains the specified element.
boolean	**isEmpty**() Returns true if this collection contains no elements.
boolean	**remove**(Object o) Removes a single instance of the specified element from this collection, if it is present (optional operation).
int	**size**() Returns the number of elements in this collection.
Iterator<E>	**iterator**() Returns an iterator over the elements in this collection.

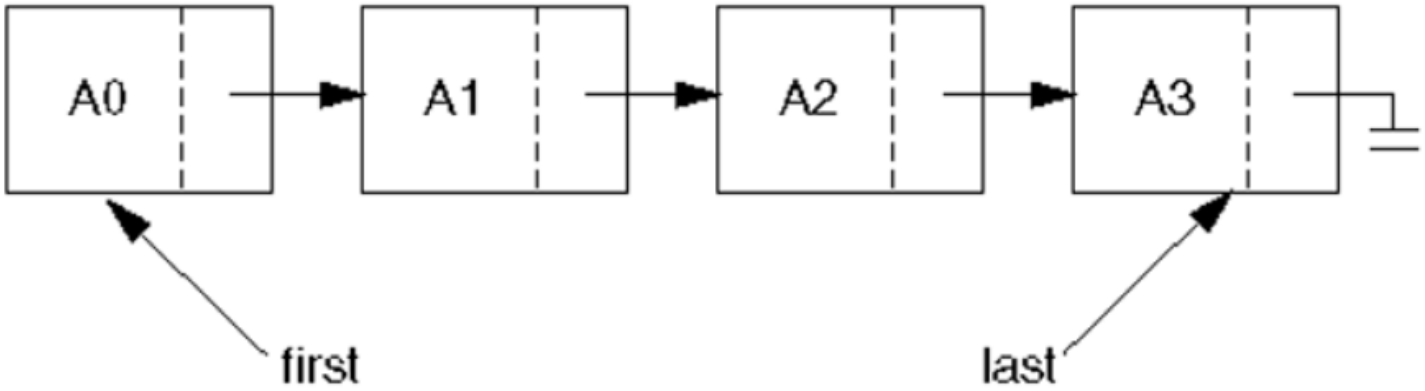
Additional List Interface methods (List extends Collection)

- A List is an ordered collection, items accessible by position. Here, *ordered* does not mean *sorted*.
- interface java.util.List<E>
- User may insert a new item at a specific position.
- Some important List methods:

void	**add**(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
E	**get**(int index) Returns the element at the specified position in this list.
int	**indexOf**(Object o) Returns the index in this list of the first occurrence of the specified element, or -1 if this list does not contain this element.
E	**remove**(int index) Removes the element at the specified position in this list (optional operation).
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element (optional operation).

Break

`LinkedList` implementation of the `List` Interface



- Stores items (non-contiguously) in nodes; each contains a reference to the next node.
- Lookup by index is linear time (worst, average).
- Insertion or removal is constant time once we have found the location.
  - show how to insert A4 after A1.
- If **`Comparable`** list items are kept in sorted order, finding an item still takes linear time.

Consider Part of a `LinkedList` implementation:

```
class ListNode{
 Object element; // contents of this node
 ListNode next;  // link to next node

 ListNode (Object element,
           ListNode next) {
   this.element = element;
   this.next = next;
 }

 ListNode (Object element) {
   this(element, null);
 }
 ListNode () {
   this(null);
 }
}
```

How to implement LinkedList?

- fields
- Constructors
- Methods

Note that the fields of this class have "package" access, so that other classes in the same package can access them directly. **ListNode** objects are used like C structs.

Let's do parts of a `LinkedList` implementation

```
class LinkedList implements List {
  ListNode first;
  ListNode last;
```

Constructors: (a) default (b) single element.

methods:

Attempt these in the order shown here.

`public boolean add(Object x)`

Appends the specified element to the end of this list (returns `true`)

`public int size()` Returns the number of elements in this list.

`public void add(int i, Object x)` adds o at index i.

`throws IndexOutOfBoundsException`

`public boolean contains(Object x)`

Returns true if this list contains the specified element. (2 versions).

`public boolean remove(Object x)`

Removes the first occurrence (in this list) of the specified element.

`public Iterator iterator()` **Can we also write listIterator( ) ?**

Returns an iterator over the elements in this list in proper sequence.

What's an iterator?

- More specifically, what is a `java.util.Iterator`?
  - It's an interface:
  - **`interface java.util.Iterator<E>`**
  - with the following methods:

boolean	**hasNext**() Returns `true` if the iteration has more elements.
E	**next**() Returns the next element in the iteration.
void	**remove**() Removes from the underlying collection the last element returned by the iterator (optional operation).

An extension, `ListIterator`, adds:

boolean	**hasPrevious**() Returns `true` if this list iterator has more elements when traversing the list in the reverse direction.
int	**nextIndex**() Returns the index of the element that would be returned by a subsequent call to `next`.
[redacted]ject	**previous**() Returns the previous element in the list.
int	**previousIndex**() Returns the index of the element that would be returned by a subsequent call to `previous`.
void	**set**(Object o) Replaces the last element returned by `next` or `previous` with the specified element (optional operation).

Work on Linked Lists

- At the board with your team.
- Try to do the simple `add`, then `size`, then the more complex `add`.

Markov work time