cs3101-003 Java: lecture #5

- news:
  - homework #4 due today
  - homework #5 out today
- today's topics:
  - applets, networks, html
  - graphics, drawing, handling images
  - graphical user interfaces (GUIs)
  - event handling
  - interfaces

networks (1).

- two or more computers connected to each other
- networked computers can share information
- and resources, e.g.:
  - printer
  - file server
- example: CUNIX system
- connections:
  - *point-to-point* — computers are directly connected to each other
    - message speed is fast
    - adding computers is expensive
  - single communication line
    - message speed can be slow
    - adding computers is cheap

networks (2).

- *network address*
  - uniquely identifies each computer on the network
- *packet*
  - long messages are split into pieces
  - each piece is a packet, sent individually along the network
  - improves message speed
- *local-area network* (LAN)
  - designed to span short distances
- *wide-area network* (WAN)
  - designed to span longer distances
  - connects multiple LANs

networks (3).

- the *Internet*
  - developed in the 1970s as ARPANET
  - the ultimate WAN — a network of networks
- *protocol*
  - set of rules governing communication
  - TCP/IP (transmission control protocol / internet protocol)
- IP address = network address on the Internet
  - numeric, e.g., 204.192.116.2
  - also have text equivalents called *Internet addresses*, which are comprised of local computer names (i.e., name of computer on LAN) plus domain names (i.e., name of LAN on WAN)
  - domain names are controlled by the Internet Naming Authority
- *domain name system* (DNS)
  - translates between IP address and Internet address

networks (4).

- the *World Wide Web* (WWW)
- provides standard method of interfacing to the Internet from the user level
- uses *hypertext*
  - non-linear method of organizing information
  - refers only to textual information
- *hypermedia*
  - refers to non-textual information, such as sound, video and graphics
- *browser*
  - user program that provides method of viewing WWW documents
  - early browsers included: Archie, Gopher
  - *Mosiac*
    - first graphical WWW browser
    - released in 1993
    - became Netscape

networks (5).

- *HyperText Markup Language* (HTML)
  - standard format for WWW documents
- *Uniform Resource Locator* (URL)
  - unique document address on the WWW
- *HypterText Transfer Protocol* (http)
  - protocol used for transfering HTML documents
  - provides *one-way* transfer from *server* to *client*
- other protocols include: ftp, telnet
  - these provide *two-way* transfer between *server* and *client*
- Java
  - grew out of the above
  - allows *two-way* transfer
  - text, graphics, sound

networks (6).

- *client-server* architecture
- comes from operating system design
- methodology by which tasks are divided onto different processors according to functionality
- programs can be divided into:
  - computation portion
  - drawing or output portion
- each portion can be executed on a different CPU
- X windows
  - windowing system used under UNIX
  - with X windows, the drawing is done on the *client*, although the execution may be happening on a different physical machine, the *server*

applets (1).

- Java programs can run as *applications* or *applets*
- *application:*
  - executed using the *java* command
  - server and client can be the same machine or different machines
  - client invokes JVM which interprets classes and runs them
- *applet:*
  - must be executed using a browser, like Netscape, or the *appletviewer* command
  - server sends applet to the client, in the form of class files; applet invokes JVM which interprets classes and runs them on the client
  - there are two parts:
    - an HTML file used to invoke the applet
    - Java class file(s) that contain the applet code

applets (2).

- file name = hi.html

```
<html>
<title>
sample applet page
</title>

the applet will be shown below...

<applet code="hi.class" width=400 height=400>
</applet>

</html>
```

applets (3).

- file name = hi.java

```
import java.awt.*;
import java.applet.Applet;

public class hi extends Applet {

  public void paint( Graphics g ) {
    g.drawString( "hi",10,10 );
  } // end of paint()

} // end of class hi
```

applets (4).

- `java.awt` package
  - *Abstract Windowing Toolkit* (AWT)
  - classes that support graphical user interfaces (GUI)
  - includes `java.awt.Component` method:
    - `public void paint()`
- `java.applet.Applet` class
  - `public void init()`
  - `public void start()`
  - `public void stop()`

frames (1).

- used when you want to write an *application* that has graphics
- relationship between `Frame` and `Applet`:

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Panel
                    |
                    +--java.applet.Applet

java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Window
                    |
                    +--java.awt.Frame
```

frames (2).

```
import java.awt.*;

public class hiho extends Frame {

  mycanvas c;

  public hiho( String title ) {
    super( title );
  } // end of hiho constructor

  public static void main( String[] args ) {
    hiho h = new hiho( "myframe!" );
    h.c = new mycanvas();
    h.add( h.c );
    h.setSize( 100,100 );
    h.setBackground( Color.red );
    h.show();
  } // end of main()

} // end of class hiho
```

frames (3).

```
import java.awt.*;

public class mycanvas extends Canvas {

  public Dimension getMinimumSize() {
    return( new Dimension( 90,90 ));
  } // end of getMinimumSize()

  public void paint( Graphics g ) {
    setBackground( Color.blue );
    g.drawString( "hiho",10,10 );
  } // end of paint()

} // end of class mycanvas
```

graphics (1).

- `java.awt.Graphics` class
- X-windows coordinate system
- drawing primitives:
  - lines
  - Strings
  - rectangles
  - ovals
  - arcs
- color

graphics (2).

- simple methods from the `java.awt.Graphics` class
- `void drawLine( int x1, int y1, int x2, int y2 );`
  - draws a line connecting (x1,y1) and (x2,y2);
- `void drawString( String str, int x, int y );`
  - draws the text in “str”, with its lower left corner at (x,y)

graphics (3).

```
import java.awt.*;
import java.applet.Applet;

public class hi2 extends Applet {

  public void paint ( Graphics g ) {
    g.drawString( "hello world!",10,10 );
    g.drawLine( 0,400, 400,0 );
  } // end of paint()

} // end of class hi2()
```

graphics (4).

- bounding rectangles
  - coordinates of origin (upper left corner)
  - extent (width and height)
- arcs
  - measured in degrees
  - starting from 0° (along positive X-axis)
  - extent (total angle of arc)

graphics (5).

- methods from the `java.awt.Graphics` class for drawing outlines of shapes
- `void drawRect( int x, int y, int width, int height );`
  - draws a rectangle with its upper left corner at (x,y), extending the specified "width" and "height"
- `void drawOval( int x, int y, int width, int height );`
  - draws an oval circumscribed in the bounding rectangle with its upper left corner at (x,y), extending the specified "width" and "height"
- `void drawArc( int x, int y, int width, int height, int startAngle, int arcAngle );`
  - draws an arc whose oval is circumscribed in the bounding rectangle with its upper left corner at (x,y), extending the specified "width" and "height", where the arc starts at the "startAngle", measured in degrees (where 0°) is horizontal along the positive x-axis), extending for "arcAngle" degrees

graphics (6).

```
import java.awt.*;
import java.applet.Applet;

public class hi3 extends Applet {

  public void paint ( Graphics g ) {
    g.drawRect( 10,300,25,25 );
    g.drawOval( 10,250,25,25 );
    g.drawArc( 10,200,25,25,45,90 );
  } // end of paint()

} // end of class hi3()
```

graphics (7).

- methods from the `java.awt.Graphics` class for drawing filled shapes
- `void fillRect( int x, int y, int width, int height );`
  - draws a filled rectangle with its upper left corner at (x,y), extending the specified "width" and "height"
- `void fillOval( int x, int y, int width, int height );`
  - draws a filled oval circumscribed in the bounding rectangle with its upper left corner at (x,y), extending the specified "width" and "height"
- `void fillArc( int x, int y, int width, int height, int startAngle, int arcAngle );`
  - draws a filled arc whose oval is circumscribed in the bounding rectangle with its upper left corner at (x,y), extending the specified "width" and "height", where the arc starts at the "startAngle", measured in degrees (where 0°) is horizontal along the positive x-axis), extending for "arcAngle" degrees

graphics (8).

- methods from the `java.awt.Graphics` class for drawing polygons
- `void drawPolygon(int[] xPoints, int[] yPoints, int nPoints);`
  - draws a closed polygon defined by arrays of x and y coordinates
- `void drawPolygon( Polygon p );`
  - draws the outline of a polygon defined by the specified `Polygon` object
- `void drawPolyline( int[] xPoints, int[] yPoints, int nPoints );`
  - draws a sequence of connected lines defined by arrays of x and y coordinates
- the first two have counterparts for drawing filled polygons:
  - `void fillPolygon(int[] xPoints, int[] yPoints, int nPoints);`
  - `void fillPolygon( Polygon p );`

graphics (9).

- `java.awt.Color` class
- color is defined using the “RGB” methodology
- “Red”, “Green”, “Blue”
- each is an integer between 0 and 255, where 0 means no color and 255 means maximum color
- so white is: red=255 green=255 blue=255 or the ordered triple (255,255,255)
  - and black is: red=0 green=0 blue=0
  - and red is: red=255 green=0 blue=0
  - and green is: red=0 green=255 blue=0
  - and blue is: red=0 green=0 blue=255
- make up your own colors...

graphics (10).

- even more methods from the `java.awt.Graphics` class
  - `void setColor( Color color );`
    - sets the foreground (pen) color to the specified color
  - `void fillRect( int x, int y, int width, int height );`
    - draws a filled rectangle with its upper left corner at (x,y), extending the specified "width" and "height"
  - `void fillOval( int x, int y, int width, int height );`
    - draws a filled oval circumscribed in the bounding rectangle with its upper left corner at (x,y), extending the specified "width" and "height"
  - `void fillArc( int x, int y, int width, int height, int startAngle, int arcAngle );`
    - draws a filled arc whose oval is circumscribed in the bounding rectangle with its upper left corner at (x,y), extending the specified "width" and "height", where the arc starts at the "startAngle", measured in degrees (where 0°) is horizontal along the positive x-axis), extending for "arcAngle" degrees

graphics examples.

- `snowman.java`
- `bullseye.java`
- `snowman2.java`, `man.java` (with helper class)
- `snowman3.java` (animated)
- `snowman4.java` (another way of doing animation)

GUIs (1).

- GUI = Graphical User Interface
- `java.awt` classes:
  - `Component`
  - `Container`
  - `LayoutManager`
  - `Event`
- `java.awt.event` classes:
  - `ActionListener`
  - `ItemListener`
  - `KeyListener`
  - `MouseListener`
  - `MouseMotionListener`

GUIs (2).

- *components*
- a component is a building block of any GUI
- here are some examples:
  - `Label`
  - `TextField`, `TextArea`
  - `Button`
  - `Checkbox`, `CheckboxGroup`
  - `Choice`
  - `List`

GUIs (3).

- *containers*
- a container is a special component that can hold other components
- here are some examples:
  - `Applet`
  - `Frame`
  - `Panel`

GUIs (4).

- *layout managers*
- a layout manager describes where the components are laid out within a given container
- you need to "set" the layout manager for each container
- you can "nest" containers (and their layour managers)
- `BorderLayout` — simplest layout manager
- looks like this:

north		
west	center	east
south		

- `GridBagLayout` — more complex layout manager; but gives you the most control

GUIs (5).

- *listeners*
- are interfaces
- you need to `implement` the appropriate listener(s), depending on what events you want to handle
- then you need to override each method in the interface
- e.g., for a `KeyListener`, you need:
  - `keyPressed()`
  - `keyTyped()`
  - `keyReleased()`
- the body of a method can be empty, if you don't want to do anything when a given event occurs

events (1).

- an *event* represents some action on the part of the user
- user-generated events are entered either through the *mouse* or the *keyboard*
- examples:
  - mouse pressed
  - mouse released
  - mouse clicked
  - mouse entered
  - mouse exited
  - mouse moved
  - mouse dragged

listeners (1).

- a *listener* is a part of a program that captures these events for processing in the program
- frequently, a *listener interface* is created
- for example, `java.awt.event.MouseListener`:
  - `void mousePressed( MouseEvent evt );`
  - `void mouseReleased( MouseEvent evt );`
  - `void mouseClicked( MouseEvent evt );`
  - `void mouseEntered( MouseEvent evt );`
  - `void mouseExited( MouseEvent evt );`
- what is a `MouseEvent`?
  - `Point getPoint();`
  - `int getX();`
  - `int getY();`
  - `int getClickCount();`

listeners (2).

- another example, `java.awt.event.KeyListener`:
  - `void keyPressed( KeyEvent evt );`
  - `void keyReleased( KeyEvent evt );`
  - `void keyTyped( KeyEvent evt );`
- what is a `KeyEvent`?
  - `char getKeyCode();`

GUI examples.

- MouseListener examples:
  - `Dots.java`
  - `Dots2.java`
  - `Dots3.java`
- KeyListener examples:
  - `Dots4.java`
  - `Dots5.java`
- `gui.java`, `snowflake.java`

interfaces (1).

- an *interface* is a group of *abstract methods* that are defined by all classes that implement the interface
- an *abstract method* is one that does not have an implementation, i.e., there is no body of code for the method
- *polymorphism* means "having many forms"
  - lets us use different implementations of a single interface
  - *binding* happens when a particular implementation is locked to an interface
  - this can happen at compile time or at run time
  - an example of a run-time or *dynamic binding* is:
    `((Philosopher)current).pontificate();`
    from the example to follow

interfaces (2).

```
public interface Speaker {

  public void speak();

  public void announce( String str );

} // end of Speaker interface
```

interfaces (3).

```
public class Philosopher implements Speaker {
  private String philosophy;

  public Philosopher ( String thoughts ) {
    philosophy = thoughts;
  } // end of Philosopher constructor

  public void speak () {
    System.out.println( philosophy );
  } // end of Philosopher method speak

  public void announce ( String announcement ) {
    System.out.println( announcement );
  } // end of Philosopher method announce

  public void pontificate() {
    for ( int i=0; i<5; i++ )
      System.out.println( philosophy );
  } // end of Philosopher method pontificate

} // end of Philosopher class
```

interfaces (4).

```
public class Dog implements Speaker {

  public void speak() {
    System.out.println( "woof" );
  } // end of Dog method speak

  public void announce( String arf ) {
    System.out.println( "woof: " + arf );
  } // end of Dog method announce

} // end of class Dog
```

interfaces (5).

```
public class Talking {

  public static void main( String[] args ) {

    Speaker current;

    current = new Dog();
    current.speak();

    current = new Philosopher( "I think, therefore I am." );
    current.speak();

    ((Philosopher)current).pontificate();

  } // end of main()

} // end of Talking class
```

exercise.

- look at the examples in the class web page:
  `http://www.cs.columbia.edu/~cs3101/examples/gallery.html`
- copy the code for one of the snowman examples
- copy the code for the gui example
- replace the snowflake in the gui with the snowman
- change the components of the gui to do something interesting with the snowman

advanced graphics (1): fonts.

- fonts in Java are defined using the `java.awt.Font` class
- you can see which fonts (by name) are available in your system by using the `java.awt.GraphicsEnvironment.getAllFonts()` method
- you can get information about the point size of the font, whether it is italic, bold or plain
- you will most likely want to get the size of a string that might be drawn with the current font. first you need to create a `FontMetrics` object, then you can call the `FontMetrics.stringWidth( String str )` method to find the width

advanced graphics (2): fonts, continued.

- useful font properties available in `FontMetrics`:
  - **ascent** — the distance from the font's baseline to the top of an alphanumeric character use `int FontMetrics.getMaxAscent()`
  - **descent** — the distance from the font's baseline to the bottom of an alphanumeric character with descenders use `int FontMetrics.getMaxDescent()`
  - **height** — the distance between the baseline of adjacent lines of text; the sum of the leading + ascent + descent.
  - **leading** — aka interline spacing; the logical amount of space to be reserved between the descent of one line of text and the ascent of the next line
  - **advance** — the distance from the leftmost point to the rightmost point on the string's baseline use `int FontMetrics.charWidth( char ch )` to get the advance of the char argument

advanced graphics (3): images.

- you can load images from a URL and draw them
- load them using `java.applet.getImage( URL url )` for an applet or `java.awt.Toolkit.getImage( URL url )` for an application
- draw them using `Graphics.drawImage()` — there are a number of versions of this method
- note that an `Image` is a Java object unto itself, defined in the `java.awt` package
- also note that `URL` is a Java object that must be instantiated prior to using either of the `getImage()` methods

advanced graphics (4): animation.

- computer animation is kind of like an old-fashioned flip book
- you need to draw the object(s) being animated repeatedly, in each new location
- each time, you calculate the new position of the object(s) and redraw
- you can either redraw the entire scene
- or you can only redraw the object(s) that are moving
- but in the second case, you need to “erase” the object first, then move it to its new location and redraw
- the erasing part can be tricky if the background is not solid

advanced graphics (5): GridBagLayout.

- `GridBagLayout`
- you place components in the container in “rows” and “columns”
- you can specify the number of rows and columns
- you can specify the spacing between each row and/or column
- you can specify how a component is placed within its row/column, if it is smaller than the space allocated
- note that the height of an entire row is uniform, even if the components in each column are of different heights
- and the same for the width of a column
- all these are specified using a `GridBagConstraints` object

advanced graphics (6): GridBagLayout, continued.

```
GridBagConstraints( int gridx, int gridy, int gridwidth,
  int gridheight, double weightx, double weighty,
  int anchor, int fill, Insets insets,
  int ipadx, int ipady );
```

- `gridx`, `gridy` specify the location of the component, starting from (0,0)
- `gridwidth`, `gridheight` specify how many columns/rows the component occupies
- `weightx`, `weighty` specify how to distribute extra horizontal and vertical space
- `anchor` specifies where to place a component when it is smaller than its display area (e.g., CENTER, NORTH, NORTHEAST, ...)
- `fill` specifies whether to resize a component if it is smaller than its display area (e.g., NONE, HORIZONTAL, VERTICAL, BOTH)
- `insets` specifies minimum amount of space between a component and the edges of its display area (external padding)
- `ipadx`, `ipady` specifies how much space to add to the minimum width and height of the component (internal padding)