Java Music Systems
Nick Didkovsky, nick@didkovsky.com
Class #2 — Java Class interactions and inheritance, graphics, exceptions

• Inheritance
Object Oriented Programming languages like Java feature hierarchical class inheritance. A new class which extends an existing class inherits all the properties of that class (ie. its class variables and methods). If you are writing an Applet, for example, called MyFirstApplet, you simply declare it as:

```java
public class MyFirstApplet extends java.applet.Applet
```

The Java word “extends” means “inherits from”. So, MyFirstApplet will inherit all the properties of the Applet class, including its security model, its ability to access media via URL’s, its ability to be run in a web browser, etc. This functionality has already been written into Applet, so you, the programmer, do not have to reinvent that wheel. MyFirstApplet is called a subclass of Applet. Applet is the superclass of MyFirstApplet.

• Adding methods to subclasses
A class that inherits from another can add methods and class variables of its own that never existed before. For example, MyFirstApplet might have the following method added to it:

```java
public void greeting() {
    System.out.println("Pleased to meet you");
}
```

This greeting() method was not built into the Applet class by Sun Microsystems. It is a new method that the programmer decided to add. Now MyFirstApplet has more functionality than Applet.

• Overriding methods
Java also allows you to "override" methods. Overriding means redefining a method inherited from the superclass to do something different in the subclass. The init() and start() methods of Applet are good examples. Applet is built so that it will execute the init() method when it is first loaded into a browser. Once loaded, start() is fired. The stock init() and start() methods do nothing. By defining them as below, you take advantage of overriding them to do something useful for you.

```java
public void init() {
    setBackground(Color.pink);
    repaint();
}

public void start() {
    AudioClip sound = getAudioClip(getCodeBase(), "screech.au");
    sound.play(); // play an 8bit sound clip natively by Java
}
```

• Constructors
An object is a particular instance of a class. When the program calls new on a class, an object is created. When an object is instantiated with new, a unique method called a constructor is executed. A constructor method is specified to have the exact same name as the class it serves. For example, this method would be the constructor of a class named Stegosauria.

```java
public Stegosauria () {
    hasSpikes = true; // class variable inherited from Thyreophoria
}
```
When your program calls `Stegosauria myStegosaurus = new Stegosauria ();`, the Java virtual machine allocates memory for a new object of type `Stegosauria`, then runs the constructor declared above. The constructor can do anything, including print out messages notifying you that a new object is coming to life. Typically it is used to do some initial setup that is needed to get the object into shape and ready to go.

An object can have any number of constructors, each with a different “method signature” (see overloading below). Depending on how `new` is called, one of these constructors will be executed. For example:

```java
class Stegosauria {
    public Stegosauria (int numSpikes) {
        hasSpikes = true;
        numberOfSpikes = numSpikes;  // might be 14 for Kentrosaurus
    }
}
```

Now the programmer has a choice creating a new `Stegosaurus` with the default number of spikes by calling:

`Stegosauria myStegosaurus = new Stegosauria ();`

…or specify the number directly in the constructor by calling:

`Stegosauria myKentrosaurus = new Stegosauria (14);`

Note that the constructor method that actually executes is chosen by the method signature used above (ie, either passing an int parameter or not).

Method signatures cannot differ simply by the parameter names. Java cannot tell the difference between

```java
public Stegosauria (int numSpikes) {
    //...and...
    public Stegosauria (int numPlates)
```

…since these have the same method signature. There would be no way of knowing whether `new Stegosauria(10)` referred to the number of spikes or the number of plates. This is not a shortcoming of Java since a human wouldn't know what 10 referred to either.

* Accessor and Mutator methods
Java classes have "class variables" which hold data unique to an object. If a class has a variable of type int (ie a whole number), that variable can hold a value that is different from object to object.

Changing and retrieving the values of class variables is best done with methods, rather than by accessing the variable directly. This is in the spirit of hiding the implementation of a class from the programmer. The idea is to program to an object’s interface rather than its implementation. Then, if the implementation of data retrieval changes, well-written code won’t break. Accessors and mutators, then, are the interfaces to class variables in Java programs.

For example:

```java
public class Stegosauria extends Thyreophora {
    private long age;  // how many years ago this particular specimen lived
    private int numberOfSpikes;

    // mutator method, changes the value of a class variable
```
```java
public void setAge(long years) {
    age = years;
}

// accessor method, allows you to retrieve age
public long getAge() {
    return age;
}
}
```

In the example above, the programmer who instantiates an object of type `Stegosauria`, can only access its age through the `getAge()` method, and can only change it through `setAge()`. The programmer cannot directly change the age variable hiding under the hood. Why is this a good thing? A year after this Java implementation is published, there may be a new way to estimate the age of a `Stegosauria` fossil. Getting an age may involve logging into a database, running some fossil data through a query, and retrieving an extremely accurate age estimate. As long as the code you’ve written simply calls `getAge()`, it will not care whether `Stegosauria` logs into a database (new version) or simply returns a local variable (old version).

**Overloading**

Overloading is the technique whereby the same method name can be reused, but its action differs based on the parameter list it is passed, called the “method signature”. In JMSL’s `JMSLRandom` class, for example, the `choose()` method is heavily overloaded.

Calling `JMSLRandom.choose()` returns a random double in the range $[0.0 .. 1.0)$.
Calling `JMSLRandom.choose(10)` returns a random int in the range $[0 .. 10)$.
Calling `JMSLRandom.choose(10, 20)` returns a random int in the range $[10 .. 20)$.

Overloading methods is easy: just declare and implement each method with a unique parameter list.

For example:

```java
/** @return area of a square */
public int getArea(int side) {
    return side * side;
}

/** @return area of a rectangle */
public int getArea(int length, int width) {
    return length * width;
}
```

**Static class variables and methods**

A variable or a method that is declared `static` does not require an instance of an object to be accessed. For example, the `JMSLRandom` class does not require that you first create an object of type `JMSLRandom` before you can use `choose()`. The `choose()` method is declared `static`, so you can simply call `JMSLRandom.choose()`.

A class variable that is declared `static` contains the same value for every active object of that type. If any object changes this value, all objects will be affected. To use a horrific example, the number of customers served that is displayed under McD*nalds’s golden arches, can be thought of as a static variable. The number is the same for all instances of McD*nalds. The store manager, and list of employees, location, etc, are of course non-static, as they belong to a particular instance
of the McD*nacls class. But if "10 gazillion served" changes to "11 gazillion served", this value changes for all instances of McD*nacls.

Static variables are more than a convenience saving you time by not having to declaring an object first. They are a way of managing changes that can be accessed over a population of objects. An int variable called fooCount, for example, would be a good use of static variable which is incremented in a class's constructor. Then, each object's name could be unique by appending that int onto a String prefix.

Example:

```java
package javamus;

public class Foo {
    private static int fooCount;
    private String name;

    /** every time this constructor is called, the
    object instance will be named Foo_#, and # is incremented. */
    public Foo() {
        name = "Foo_" + fooCount++;
    }

    public String getName() {
        return name;
    }

    public static void main(String args[]) {
        Foo[] fooList = new Foo[10];
        for (int i=0; i<fooList.length; i++) {
            fooList[i] = new Foo();
        }
        for (int i=0; i<fooList.length; i++) {
            System.out.println(fooList[i].getName());
        }
    }
}
```

Graphics in Java

Java’s Graphics class has a large catalog of methods with which you can program graphic drawing, such as setColor(), drawLine(), and fillRect(). Look up this class in the Java docs and you will see the wealth of choices. More recent versions of Java have Graphics2D which provides more sophisticated control over image manipulation and rendering, such as antialiased drawing. We will restrict ourselves to the simpler Graphics model here, to get you up and running.

A useful class for drawing is java.awt.Canvas. Canvas is simply a Component with a rectangular surface. You can add() a Canvas to a Layout like any other Component such as Button, Label, TextField, etc. Use setSize(w, h) to set the dimensions of the canvas you need.
Drawing directly into a Graphics object
This approach is good if you simply want to draw directly into an AWT Canvas or a Swing JPanel. Build your GUI layout as usual. When you are ready to draw, call getGraphics() on your drawing surface and draw to that Graphics object, like so:

```java
package didkovsky.javamusic;

import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

public class DrawOnJPanel extends JApplet {
  JPanel canvas;
  JButton button;

  public void start() {
    setLayout(new FlowLayout());
    canvas = new JPanel();
    canvas.setPreferredSize(new Dimension(200, 200));
    add(canvas);
    add(button = new JButton("click"));
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ev) {
        Graphics g = canvas.getGraphics();
        int x1 = (int) (Math.random() * 200);
        int y1 = (int) (Math.random() * 200);
        int x2 = (int) (Math.random() * 200);
        int y2 = (int) (Math.random() * 200);
        g.drawLine(x1, y1, x2, y2);
      }
    });
  }
}
```

The code excerpt above draws a new line segment between randomly chosen endpoints every time the button is clicked.

Of course you can draw algorithmically by performing calculations and passing the results of those calculations to these routines, such as:

```java
... Graphics g = myCanvas.getGraphics();
g.setColor(Color.blue);
for (int i=0; i<10; i++) {
  g.drawLine(0, 0, i, 200);
}
...
```

What does the above do?

Following is a complete working example of an Applet that draws randomly colored rectangles into a JPanel.
package didkovsky.javamusic;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Title: CrazyRectanglesApplet
 * 
 * Graphics example. Draw randomly colored and randomly sized rectangles at random
 * locations on a
 * Canvas.
 * 
 * Copyright: Copyright (c) Nick Didkovsky, 2003, updated to Swing Sep 2012
 * 
 * Company: Punos Music
 * 
 * @author Nick Didkovsky, didkovn@mail.rockefeller.edu
 * @version 1.0
 */

public class CrazyRectanglesApplet extends JApplet implements ActionListener {

    JPanel myCanvas;
    JButton goButton;
    JButton clearButton;

    public static final int CANVAS_WIDTH = 500;
    public static final int CANVAS_HEIGHT = 400;

    public void init() {
        setLayout(new BorderLayout());
        myCanvas = new JPanel();
        myCanvas.setBackground(Color.yellow);
        myCanvas.setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
        add(BorderLayout.SOUTH, myCanvas);
        JPanel topPanel = new JPanel();
        topPanel.setLayout(new FlowLayout());
        goButton = new JButton("GO");
        topPanel.add(goButton);
        goButton.addActionListener(this);
        clearButton = new JButton("CLEAR");
        topPanel.add(clearButton);
        clearButton.addActionListener(this);
        add(BorderLayout.NORTH, topPanel);
    }

    public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();
        if (source == goButton) {
            drawRectangles();
        } else if (source == clearButton) {
            Graphics g = myCanvas.getGraphics();
            g.setColor(Color.yellow);
            g.fillRect(0, 0, CANVAS_WIDTH, CANVAS_HEIGHT);
        }
    }

    public void drawRectangles() {
        Graphics g = myCanvas.getGraphics();
        for (int i = 0; i < 100; i++) {
            int xPosition = (int) (Math.random() * CANVAS_WIDTH);
            int yPosition = (int) (Math.random() * CANVAS_HEIGHT);
            g.setColor(Color.red);
            g.fillRect(xPosition, yPosition, 100, 100);
        }
    }
}
int yPosition = (int) (Math.random() * CANVAS_HEIGHT);
int randomWidth = (int) (Math.random() * 100);
int randomHeight = (int) (Math.random() * 100);
int red = (int) (Math.random() * 256);
int green = (int) (Math.random() * 256);
int blue = (int) (Math.random() * 256);
System.out.println(red + " , " + green + " , " + blue);
Color myColor = new Color(red, green, blue);
g.setColor(myColor);
g.fillRect(xPosition, yPosition, randomWidth, randomHeight);
}
/**
 * Here's a trick to check your applets in a main() method. Instantiate an applet and
 * put it in
 * a JFrame. Don't forget to call init() and start() on your applet explicitly since
 * there's no
 * browser to do this automatically.
 */
public static void main(String args[]) {
    JFrame f = new JFrame("Click to close");
    CrazyRectanglesApplet applet = new CrazyRectanglesApplet();
    applet.init();
    f.add(applet);
    f.pack();
    f.setVisible(true);
    f.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
    applet.start();
}

Using an offscreen image to generate smooth animation


Sometimes you want to animate changes to an image over time, and want it to be animated
smoothly. You could redraw every frame by drawing directly into the Canvas's Graphics object,
but that would result in jerky animation since every drawing command would be seen happening
in real time.

The solution is to do your drawing into the Graphics object of an offscreen Image. When that
drawing it done, 'paste' that offscreen image all at once into the JPanel. This action is typically
done in a paintComponent(Graphics g) method, which is called whenever you want a Swing
JComponent to be repainted. Your code never calls paintComponent() directly. Instead, it called
repaint() which calls paintComponent() behind the scenes.

The following example illustrates. A background image is drawn and a solid colored ball travels
across it.

First we present a subclass of JPanel which provides an offscreen drawing context available to
other classes.

package didkovsky.javamusic;
import java.awt.*;
import javax.swing.JPanel;

public class DrawingPanel extends JPanel {

    Image offscreenImage;
    Graphics offscreenGraphics;
    int width;
    int height;

    public DrawingPanel(int w, int h) {
        super();
        this.width = w;
        this.height = h;
        setPreferredSize(new Dimension(width, height));
    }

    /** Get the offscreen graphics context. Draw into it and call repaint() */
    public Graphics getOffscreenGraphics() {
        return offscreenGraphics;
    }

    public void paintComponent(Graphics g) {
        if (offscreenImage == null || offscreenGraphics == null) {
            offscreenImage = this.createImage(width, height);
            offscreenGraphics = offscreenImage.getGraphics();
            System.out.println("created image and offscreen graphics");
        }
        if (offscreenImage != null) {
            g.drawImage(offscreenImage, 0, 0, width, height, this);
        } else {
            System.out.println("no image to draw ");
        }
    }
}

Now we present the JApplet itself which, 15 times per second, draws a background image and blue oval travelling over it. A checkbox is included. When that checkbox is unchecked, the graphics context into which the applet draws is the graphics of the canvas itself. When checked, the graphics is that of the offscreen image. In this mode, when the drawing is done, it calls repaint() on the canvas.

Note that since this applet does something over time, we want it to pause between tasks in a loop. A good way to do this in Java is using Threads. The applet implements the Runnable interface, which requires that all timing related activity happens in a run() method. Threads are the basis for scheduling events over time in Java, which later, you will see are at the heart of JMSL.
public class AnimationApplet extends JApplet implements ActionListener, Runnable {

    DrawingPanel myCanvas;
    boolean keepGoing;
    JButton goButton;
    JButton stopButton;
    public static final int CANVAS_WIDTH = 500;
    public static final int CANVAS_HEIGHT = 400;
    public static final int BALL_WIDTH = 30;
    public static final int BALL_HEIGHT = 20;

    Thread runner;

    /** When unchecked, draw directly to the Canvas graphics. When checked, draw to offscreen graphics for flicker-free animation */
    Checkbox smoothAnimationCheckbox;

    public void init() {
        setLayout(new BorderLayout);
        myCanvas = new DrawingPanel(CANVAS_WIDTH, CANVAS_HEIGHT);
        add(BorderLayout.CENTER, myCanvas);

        JPanel topPanel = new JPanel();
        topPanel.setLayout(new FlowLayout());
        goButton = new JButton("GO");
        topPanel.add(goButton);
        goButton.addActionListener(this);
        stopButton = new JButton("STOP");
        topPanel.add(stopButton);
        stopButton.addActionListener(this);

        topPanel.add(smoothAnimationCheckbox = new Checkbox("Smooth Animation"));

        add(BorderLayout.NORTH, topPanel);
    }

    /** Override applet start() */
    public void start() {
        keepGoing = false;
    }

    /** Override applet stop() */
    public void stop() {
        stopAnimation();
    }

    public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();
        if (source == goButton) {
            startAnimation();
        }
        if (source == stopButton) {
            stopAnimation();
        }
    }
}
private void drawBackground(Graphics g) {
    if (g != null) {
        g.setColor(Color.white);
        g.fillRect(0, 0, CANVAS_WIDTH, CANVAS_HEIGHT);
        int steps = 50;
        int stepSize = CANVAS_WIDTH / steps;
        for (int i = 0; i < steps; i++) {
            if (i % 2 == 0) {
                g.setColor(Color.yellow);
            } else {
                g.setColor(Color.red);
            }
            g.drawLine(i * stepSize, 0, i * stepSize, CANVAS_HEIGHT);
        }
    }
}

private void drawCharacter(Graphics g, int x, int y) {
    if (g != null) {
        g.setColor(Color.blue);
        g.fillArc(x, y, BALL_WIDTH, BALL_HEIGHT, 0, 360);
    } else {
        System.out.println("NULL, cannot animate character");
    }
}

private void startAnimation() {
    if (runner == null) {
        runner = new Thread(this);
        keepGoing = true;
        runner.start();
    }
}

private void stopAnimation() {
    keepGoing = false;
    if (runner != null) {
        try {
            runner.join(1000);
            runner = null;
        } catch (InterruptedException e) {} 
    }
}

public void run() {
    int x = CANVAS_WIDTH / 2;
    int y = CANVAS_HEIGHT / 2;
    int xStep = 2;
    int yStep = 2;
    int fps = 15;

    // will be either canvas graphics or offscreen graphics
    Graphics g;
    while (keepGoing) {
        g = myCanvas.getOffscreenGraphics();
    } else {
}
g = myCanvas.getGraphics();
}
drawBackground(g);
drawCharacter(g, x, y);

if (smoothAnimationCheckbox.getState()) {
    myCanvas.repaint();
}

pause(1000 / fps);

// update x, y and bounce off the walls if reach boundary
x += xStep;
y += yStep;
if (x == 0 || x == CANVAS_WIDTH-BALL_WIDTH) {
    xStep *= -1;
}
if (y == 0 || y == CANVAS_HEIGHT-BALL_HEIGHT) {
    yStep *= -1;
}
}

private void pause(int millis) {
    try {
        Thread.sleep(millis);
    }
    catch (InterruptedException e) {} 
}

/** Here's a trick to check your applets in a main() method.
* Instantiate an applet and put it in a Frame.
* Don't forget to call init() and start() on your applet
* explicitly since there's no browser to do this
* automatically.
*/
public static void main(String args[]) {
    JFrame f = new JFrame("Click to close");
    AnimationApplet applet = new AnimationApplet();
    applet.init();
    f.add(applet);
    f.pack();
    f.setVisible(true);
    f.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
    applet.start();
}
Handling errors with Java's Exceptions
Java provides a nice way to catch errors which might happen at runtime. If you execute some code that might cause an error (like trying to open a file that does not exist), you bracket all that code in a try { } block, and follow it with a catch(Exception e) {} block.

Java's Exception class has many subclasses, such as IOException, ArrayIndexOutOfBoundsException, and NumberFormatException. It is very hip to catch the exception you specifically expect in your code rather than catching Exception in general.

Example follows:

```java
import java.net.*;
import java.io.*;
import java.awt.*;

/** Open a file on server and echo contents in Applet
 * @author Nick Didkovsky
 */
public class ReadURL extends java.applet.Applet {
    TextArea textArea;
    public void init() {
        textArea = new TextArea("Contents of file on server:\n", 20, 40);
        add(textArea);
    }

    // opens a file on server and echoes contents to console
    public void start() {
        System.out.println("starting up...");
        try {
            URL url = new URL(getDocumentBase(), "/..//..//cgi-bin/hub1");
            DataInputStream in = new DataInputStream(url.openStream());
            readAndEcho(in);
            in.close();
        } catch (MalformedURLException e1) {
            System.out.println("Bad URL in ReadWriteURL's start(): " + e1);
        } catch (IOException e2) {
            System.out.println("Error reading file in ReadWriteURL's start() " + e2);
        }
    }

    void readAndEcho(DataInputStream in) {
        try {
            while (true) {
                byte b = in.readByte();
                textArea.appendText((char)b+"");
            }
        } catch (EOFException e1) {
            System.out.println("That's all folks!");
        } catch (IOException e2) {
            System.out.println("Error in ReadWriteURL's readAndEcho() " + e2);
        }
    }
}
```
**Extra Credit Homework:**

Create an applet that does some random drawing under user-specified constraints of your own choosing. The user should be able to use TextFields to enter constraints and a button to initiate the drawing. For example, one TextField might be used to specify the minimum X value of a randomly drawn line, while another might be used to specify the maximum value. Clicking “go” might generate 20 lines with endpoints randomly chosen within these constraints.

To generate a random number between [0..1) use Math.random()

Scale this range up to choose random x, y endpoints between [ 0..100 ) :

```java
int x1 = (int)(Math.random() * 100);
int y1 = (int)(Math.random() * 100);
int x2 = (int)(Math.random() * 100);
int y2 = (int)(Math.random() * 100);
g.drawLine(x1, y1, x2, y2);
```