MATH CLASS

- Java has a class that contains a suite of static methods to provide mathematical functionality within Java. There are trigonometric and assorted other mathematical functions provided.
- There are also a few predefined final static variables in class Math, such as Math.PI.

Ex:

```
/*     FILE: MathTestDrive.java     */

// Illustration of the class Math.

// Trig sine function is provided by the method sin( )
// Constant pi is provided by PI

public class MathTestDrive{
  public static void main(String args[ ])
  {
    double start, end, current, step, value;

    /* Set initial values */
    start = 0.0;
    end = 2 * Math.PI;
    step = 0.01;

    /* Loop to compute and display values */
    for(current = start; current <= end; current += step){
      value = Math.sin(current);
      System.out.println(value);
    }
  }
}


/*    OUTPUT: MathTestDrive.java

        0.0
        0.009999833334166664
        0.01999866669333308
        ...
        0.9999417202299663
        0.9999996829318346
        0.9999576464987401
        ...
        0.0015926529165099209
        -0.008407247367125526
        ...
        -0.9999971463877179
        ...
        -0.003185301793227696

*/
```


PRIMITIVE TYPE WRAPPER CLASSES

- The primitive data types do not represent Java classes and objects. Java is an object-oriented programming language. Java has a set of classes that correspond to each of the primitive data types. In this way objects can be created that correspond to each of the primitive data types. These classes create a “Wrapper” around a value of a particular primitive data type.
- The Wrapper classes allow primitive type data to be treated like other Java objects.
- The Wrapper classes allow associated methods to be called for objects of the class.
- The Wrapper classes allow functionality for a category of data to be defined for that particular class of data.

- Note: Wrapper class is a general concept and is not specific only to primitive data types. Any class that creates a wrapper around an object or element of another type or class is considered a “wrapper.”

Ex:

```
/*     FILE: Wrapper1.java     */

// Illustration of the Integer wrapper class.

public class Wrapper1
{
  public static void main(String args[ ])
  {
    int x[ ];
    Integer x2[ ];

    x = new int[5];
    for(int i=0; i<x.length; i++)  // Load up some int integers
      x[i] = i*10;

    x2 = new Integer[x.length];
    for(int i=0; i<x.length; i++)  // Load up some Integer integers
      x2[i] = new Integer(x[i]);   // ... from int integers

    array_display(x);
    array_display(x2);
  }

  static void array_display(int a[ ])
  {
    for(int i = 0; i < a.length; i++){
      System.out.print( "a[" + i + "] = " + a[i]+"  ");
    }
    System.out.println( );
  }

  static void array_display(Integer a[ ])
  {
    for(int i = 0; i < a.length; i++){
      System.out.print( "A[" + i + "] = " + a[i]+"  ");
    }
    System.out.println( );
  }
}
```

cont…

AUTOBOXING

- Java makes automatic conversions to and from the Primitive Wrapper classes in most contexts. This "automatic" conversion is called autoboxing.
- Autoboxing is the "wrapping" and unwrapping of the primitive-type element within each Primitive Wrapper class object.

Ex:

```
/*     FILE: Wrapper5.java     */

// Illustration of the Character wrapper class.

// Autoboxing - Java performs automatic conversions between
// ...Primitive-type wrapper classes and the corresponding
// ...primitive datatypes in most contexts.

public class Wrapper5
{
  public static void main(String args[ ])
  {
    Character c = new Character('A');
    char c2 = c;

    if(Character.isUpperCase(c))
      System.out.println(c + ":c is upper case alpha.");
    if(Character.isUpperCase(c2))
      System.out.println(c2 + ":c2 is upper case alpha.");

    c2 = 68;
    c2++;

    System.out.println("c2: " + c2);

    c = 68;
//    c = c + 1;
    c++;

    System.out.println("c: " + c);
  }
}


/*    OUTPUT: Wrapper5.java

       A:c is upper case alpha.
       A:c2 is upper case alpha.
       c2: E
       c: E

*/
```


FORMATTING NUMERIC VALUES

- The default format of the string representation of a numeric value may not be the representation desired. Several "formatting" classes exist to allow a programmer to take control of the formatting of numeric data.
- The DecimalFormat class is used as an illustration below.

Ex:

```
/*     FILE: FormatDemo.java     */

// Illustration of the DecimalFormat class for controlling
// ... the number of decimal places on output.

import java.text.DecimalFormat;
public class FormatDemo
{
   public static void main(String args[ ])
   {
      double x,y,z;
      DecimalFormat decimal2 = new DecimalFormat("0.00");
      x = 1.0/3.0;
      y = 2.0/3.0;

      System.out.println("x = " + x);
      System.out.println("y = " + y);

      System.out.println("x = " + decimal2.format(x));
      System.out.println("y = " + decimal2.format(y));
   }
}


/*    OUTPUT: FormatDemo.java

        x = 0.3333333333333333
        y = 0.6666666666666666
        x = 0.33
        y = 0.67

*/
```

BIG NUMBERS

- The primitive integer and floating-point types have limited precision. They can store large values but they do have limits.
- Java provides a set of classes to provide arbitrary-precision signed floating-point and integer numbers. These classes are BigDecimal and BigInteger, respectively.
- These classes are only of interest in situations where very, very large or very, very exact values are required.

Ex:

```
/*     FILE: BigNumbers.java     */

// Arbitrarily large numbers, BigInteger & BigDecimal

public class BigNumbers{
  public static void main( String args[ ] )
  {
    int x = 2111222333;

    System.out.println( "x = " + x);

    x *= 2;

    System.out.println( "Now x = " + x);
  }
}


/*    OUTPUT: BigNumbers.java

       x = 2111222333
       Now x = -72522630

*/
```


STATIC IMPORT

- Static instance variable names can be imported to simplify your code .
  - Note: The class whose names you are statically importing must be in a package.
- Just like a normal import statement, this is only a convenience for simplifying names.

```
/*     FILE: MathTestDrive2.java     */

// Illustration of the class Math.

// Trig sine function is provided by the method sin( )
// Constant pi is provided by PI

import static java.lang.Math.*;

public class MathTestDrive2{
  public static void main(String args[ ])
  {
    double start, end, current, step, value;

    /* Set initial values */
    start = 0.0;
    end = 2 * PI;
    step = 0.01;

    /* Loop to compute and display values */
    for(current = start; current <= end; current += step){
      value = sin(current);
      System.out.println(value);
    }
  }
}


/*    OUTPUT: MathTestDrive2.java

       0.0
       0.009999833334166664
       0.01999866669333308
       ...
       0.9999417202299663
       0.9999996829318346
       0.9999576464987401
       ...
       0.0015926529165099209
       -0.008407247367125526
       ...
       -0.9999971463877179
       ...
       -0.003185301793227696

*/
```

EXCEPTIONS

- Exceptions are unexpected run-time error conditions.
- Java provides an exception handling mechanism that allows the programmer to attempt to deal with these.
- The exception handling mechanism is separate from the parameter passing/return value mechanism of messaging or method calling. This allows exception handling to be kept separate from the method invocation mechanism and not be interlaced with it through specialized parameters or return values.
- The base class for all exceptions is Throwable. Any subclass of Throwable can be thrown as an exception.
- Throwable has two subclasses, Error and Exception. Error is a “system” class of exception that usually is beyond the control of the programmer. Class Exception and it’s subclasses are the focus of exception handling for the programmer.
- Class RunTimeException is a subclass of Exception. These are exceptions that are of a very general nature. Java does not require a method to identify that it throws these types of exceptions.
- For other classes of Exceptions the Java compiler will force the method that generates the exception to state that it can throw that type of exception. This statement is made with a throws clause on the first line of the method definition.

INNER CLASSES

- Java allows classes to contain other class definitions. These "contained" classes are referred to as "inner" classes since they are contained "in" another class.
- Inner classes allow:
  - class definitions to be localized and hidden
  - classes to be created on an "as-needed" basis
  - class definitions to be made without name concerns, global or named at all
- Class definitions contained in other classes are referred to as local classes.
- Class definitions that are not named are referred to as anonymous classes.

Ex:

```
/*     FILE: ./shapes5/Triangle.java     */

package shapes5;
import shapes4.Shape;
import shapes4.Point;
import java.awt.Graphics2D;

/**
    Model a Triangle that is-a Shape with 2 Point vertices

    An inner class defined to produce an object to compute area
    of the Triangle.
*/
public class Triangle extends Shape{
    private Point v2;
    private Point v3;

    public Triangle( )
    { }

    public Triangle( Point p_p1, Point p_p2, Point p_p3)
    {
      setTriangle(p_p1, p_p2, p_p3);
    }

    public Triangle(int px1, int py1, int px2, int py2,
                    int px3, int py3)
    {
      setTriangle(new Point(px1, py1), new Point(px2, py2), new Point(px3, py3));
    }

    public void setTriangle( Point p1, Point p2, Point p3)
    {
      setX(p1.getX( ));
      setY(p1.getY( ));
      v2 = new Point(p2);
      v3 = new Point(p3);
    }

    public Point getVertex1( )
    {
      return new Point(getX( ),getY( ));
    }
```

cont…

APPLICATIONS WITH WINDOWS

- A window in Java using Swing is an object of class JFrame. JFrame is derived from other Java classes.
- To make a window-based application extend class JFrame.
- The original 'windowing' classes in java were part of the "Abstract Windowing Toolkit" or AWT.
- Swing was an addition to Java that gave 'windowing' control to java itself.
- Windows are found in Graphical User Interfaces so many window routines are found in the AWT. The Java Swing routines are found in javax.swing.
- Since windows exist in an event-driven environment they must respond to events that affect them.

Ex:

```
/*     FILE: MyJWindow.java     */

// Creates an application with a Swing window by extending JFrame
import javax.swing.JFrame;

public class MyJWindow extends JFrame
{

  public static void main(String args[ ])
  {
    MyJWindow theWindow = new MyJWindow("my Window");

    theWindow.setSize(500,500);
    theWindow.setVisible(true);
  }
  public MyJWindow(String s)
  {
    super(s);
  }
}
```

cont…

PLACING A JPANEL IN A JFRAME

- A JFrame can be populated with other GUI objects.
- One useful object is the JPanel. It can be thought of as a very primitive subwindow, or subregion of a window.
- Drawing and adding of additional components can take place on the JPanel.

Ex:

```
/*     FILE: JFrameDraw.java     */

// Drawing in a JFrame

/*

   Getting information about the "non-frame" portion of the
   window.

*/

import javax.swing.JFrame;
import java.awt.Graphics;

public class JFrameDraw extends JFrame
{

  public static void main(String args[ ])
  {
    JFrameDraw theWindow = new JFrameDraw("JFrame Draw");

    theWindow.setDefaultCloseOperation(EXIT_ON_CLOSE);

    theWindow.setSize(400,200);
    theWindow.setVisible(true);
  }

  public JFrameDraw(String s)
  {
    super(s);
  }

  public void paint(Graphics g)
  {
    super.paint(g);

    int height, width, centerX, centerY, length;

    height = getContentPane( ).getHeight( );
    width = getContentPane( ).getWidth( );

    System.out.println("Content width = " + width);
    System.out.println("Content height = " + height);

    height = getHeight( );
    width = getWidth( );

    System.out.println("Frame width = " + width);

    System.out.println("Frame height = " + height);

  }
}
```

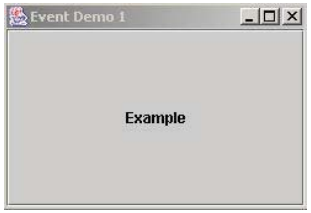
cont…

COMPONENT & EVENT BASICS

- Each component in the AWT gets notified of the events that impact it.
- For some third party to get informed of the event it must tell the component to put it on a "listener" list for notification of the event.
- To be a listener an object must implement a particular listener interface. (The compiler validates that the object implements the correct interface.)
- The component can communicate with the listener object through the interface when the "listened for" event occurs. (This will happen regardless of the listening objects actual class since an interface establishes "type.")

Ex:

```
/*     FILE: Event1.java     */

// A JFrame with a button.

import javax.swing.JFrame;
import javax.swing.JButton;

class Event1 extends JFrame
{
  //The Button
  private JButton myButton;

  public static void main (String[ ] args)
  {
    JFrame f = new Event1( );
    f.setDefaultCloseOperation(EXIT_ON_CLOSE);
    f.setSize(250,170);
    f.setVisible(true);
  }

  public Event1( )
  {
    super("Event Demo 1");

    //create JButton
    myButton = new JButton("Example");

    //add JButton to window
    getContentPane( ).add(myButton);
  }
}
```



WINDOW EVENTS

- Windows, just like any other graphical component,get notified of the events that impact it.

Ex:

```
/*     FILE: WindowEvents1.java     */

// Creates an application with a Window using JFrame
import javax.swing.JFrame;

public class WindowEvents1 extends JFrame
{

   public static void main(String args[ ])
   {
     WindowEvents1 theWindow = new WindowEvents1("Window events 1");
     theWindow.setDefaultCloseOperation(EXIT_ON_CLOSE);

     theWindow.setSize(500,500);
     theWindow.setVisible(true);
   }

   public WindowEvents1(String s)
   {
     super(s);
   }
}
```

COMBINING DRAWING AND MOUSE EVENTS

- Mouse events will be monitored and used to generate drawing within a JPanel.
- A JPanel is treated like a sub-Window within a Jframe.

Ex:

```
/*     FILE: TicTacToe.java     */

// TicTacToe starter in a Swing JFrame

/*  MouseEvents and drawing are combined to produce a TicTacToe
    game, at least the start.

    Just indicates which cell is picked.                        */

import java.awt.event.MouseEvent;
import java.awt.event.MouseAdapter;
import javax.swing.JFrame;
import javax.swing.JPanel;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.Line2D;
import java.awt.geom.Ellipse2D;

public class TicTacToe extends JFrame
{

   public static void main(String args[ ])
   {
      TicTacToe theWindow = new TicTacToe("TicTacToe Window");

      theWindow.setDefaultCloseOperation(EXIT_ON_CLOSE);

      theWindow.getContentPane( ).add(new DrawingPanel( ));

      theWindow.setSize(400,400);
      theWindow.setVisible(true);
   }

   public TicTacToe(String s)
   {
      super(s);
   }
}
```

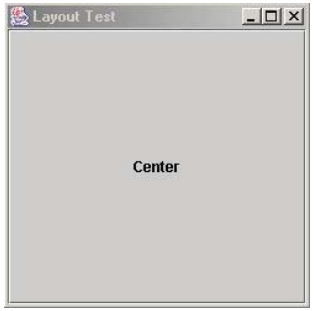
cont…

LAYOUT MANAGERS

- Layout managers control how the components in a container are “layed out” when they are displayed.
- Java provides several different layout managers and a programmer can write their own.

Ex:

```
/*     FILE: LayoutExample.java     */

/*
    The default Layout Manager for a JFrame is a BorderLayout.

    By default, a component added into a BorderLayout is placed
    in the Center region of the North/South/East/West/Center
    regions of the layout.

    By default the Center region expands to fill any adjacent,
    unused regions.

    By default a component is expanded to fill its region.
*/
import javax.swing.JFrame;
import javax.swing.JButton;

public class LayoutExample
{
  public static void main(String[ ] args)
  {
    JFrame frame = new JFrame("Layout Test");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    frame.getContentPane( ).add(new JButton("Center"));

    frame.setSize(250,250);
    frame.setVisible(true);
  }
}
```



MORE COMPONENTS

- The following examples introduce some more graphical Java components.

Ex:

```
/*     FILE: TicTacToe6.java     */

// TicTacToe game in a Swing JFrame

/*
    Use JButtons for cells. Each button/cell reacts to
    clicks and allows a label to be displayed through
    features of the JButton class.  Simplifies work for
    this application.

*/

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;
import javax.swing.JLabel;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.Line2D;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.BorderLayout;
import java.awt.GridLayout;

public class TicTacToe6 extends JFrame
{

   public static void main(String args[ ])
   {
      TicTacToe6 theWindow = new TicTacToe6("TicTacToe Window");

      theWindow.setDefaultCloseOperation(EXIT_ON_CLOSE);

      JLabel turn = new JLabel( );

      TicTacToePanel ticTacToe = new TicTacToePanel(turn);

      theWindow.getContentPane( ).add(ticTacToe ,BorderLayout.CENTER);
      theWindow.getContentPane( ).add(turn ,BorderLayout.SOUTH);

      theWindow.setSize(400,400);
      theWindow.setVisible(true);
   }

   public TicTacToe6(String s)
   {
      super(s);
   }
}

class TicTacToePanel extends JPanel
{

   private boolean x;
   private JLabel turn;
   private int winner[ ];
   private JButton board[ ][ ];
```

cont…

FILE I/O - BASICS

- File I/O is accomplished by using a set of Stream and Stream Control objects
- There are two major types of I/O streams:
  - Input/Output streams

  - byte based I/O routines
  - Reader/Writer streams

  - character based, text I/O routines

Ex:

```
/*     FILE: WriteText.java     */

// This program writes some strings to a text file.
// It passes on any exceptions.
import java.io.IOException;
import java.io.BufferedWriter;
import java.io.FileWriter;

public class WriteText{

  public static void main(String args[ ]) throws IOException
  {
     BufferedWriter output = null;
     int count;

     // Open the file
     output = new BufferedWriter(new FileWriter("WriteText.out"));
     for (count = 1; count < 11; count++) {
               output.write("Line of output " + count + ".");
               output.newLine( );
     }
     output.close( );
     System.exit(0);
  }
}

/*   OUTPUT: WriteText.out

       Line of output 1.
       Line of output 2.
       Line of output 3.
       Line of output 4.
       Line of output 5.
       Line of output 6.
       Line of output 7.
       Line of output 8.
       Line of output 9.
       Line of output 10.

*/
```

SERIALIZATION OF OBJECTS

- Objects can be written to a file/stream. This technique is called “serialization” in Java.
- The objects can then be reconstructed by reading them back in.

Ex:

```
/*     FILE: Serialize1.java     */

import java.io.*;
import shapesSerial.*;

class Serialize1{

  public static void main(String arg[ ])
  {
    Point p;
    Triangle t;
    Circle c;

    p = new Point(2,5);
    c = new Circle(p,5);
    t = new Triangle(1,2,3,4,5,6);

    ObjectOutputStream Ooutput = null;
    String fileName = "Serialize1.bin";

    // Open the file
    try {
      Ooutput = new ObjectOutputStream(
                    new BufferedOutputStream(
                         new FileOutputStream(fileName)));
    }
    catch (IOException e) {
      System.err.println("Unable to open file " + fileName +
                         "\n" + e.toString( ));
      System.exit(1);
    }

    try{
      Ooutput.writeObject(t);
      Ooutput.writeObject(c);
      Ooutput.writeObject(p);
    }
    catch (IOException io) {
      System.err.println("Error during write to file " +
                         fileName + "\n" + io.toString( ));
      System.exit(1);
    }
    try {
      Ooutput.close( );
      System.exit(0);
    }
    catch (IOException io) {
      System.err.println("Error closing file " +
                         fileName + "\n" + io.toString( ));
      System.exit(1);
    }
  }
}
```


THREADS

- Threads are separate lines of control within a single program.
- There are two ways to create threads within a Java program:
  - Extend class Thread
  - Implement the interface Runnable
- Each thread is an instantiated object of class Thread.
- These separate threads of control can then be started, stopped, paused and interrupted.

Ex:

```
/*     FILE: Thread1.java     */

// Using threads.  Several threads get started and each
// gets an interleaved opportunity to run.

public class Thread1 {

   public static void main(String args[])
   {
      Thread_extended thread1 = new Thread_extended(1);
      thread1.start( );

      Thread_extended thread2 = new Thread_extended(2);
      thread2.start( );

      Thread_extended thread3 = new Thread_extended(3);
      thread3.start( );

      Thread_extended thread4 = new Thread_extended(4);
      thread4.start( );
   }
}

class Thread_extended extends Thread {
   private int id;

   public Thread_extended(int i)
   {
     id = i;
   }

   public void run( )
   {
     for(int i=0; i<1000; i++){
       System.out.println("Thread " + id + " running.");
     }
     return;
   }
}
```

cont…

NETWORK APPLICATIONS

- Java has classes and packages that allow Java applications to communicate across a network in a client/server scenario using the TCP or the UDP protocols.
- A computer system with one network card has one network connection. To allow multiprocessing over the network many separate transfers/conversations can take place over this one connection. Each conversation is uniquely identified by a port number.
- Conceptually each port can be thought of as a separate connection. But in reality it is only an ID number that allows the computer system to route network traffic to the correct application. Each connection through a port is treated as a "socket" that two applications can "plug-in to" to connect across the network and then communicate through.
- With a TCP connection, a server application starts up and listens on its designated port for a client to connect to it. When the server and client connect, their conversation is transferred to another port so that the server can continue listening on its original port for other clients.
- With a datagram connection the server just listens. When it receives a message it decides what to do. There is no two-way link between the two. If the server so chooses, it can get the source of the datagram from the datagram and can then respond. Also the client just sends. Since there may be no confirmation, and there is no real "conversation" between the two, the client can send and then be done.

REPOSITORIES & ITERATORS

- Java provides several classes of "repositories" for storing objects. These classes allow objects to be collected and processed as a group or set.
- Often these types of repositories are referred to generically as "collections" or "containers". However these names are already used by Java so the generic term used in these notes will be "repository".
- Objects can be easily added to the repository and removed from the repository.
- Objects can be accessed individually.
- The entire repository can be treated as a single entity.
- Arrays are a "well-known" example of a repository.
- Iterators are a class that allows an entire repository to be processed and controls the processing so each object in the repository gets processed.

JAVA ARCHIVER - JAR

- Java recognizes files stored in an archive format produced by the java archiver. It is in actuality the zip file format.
- Jar files have two main advantages:
  - compression
  - Many files can be packaged as a single file
- The compression and the packaging of multiple files in a single file improves applet performance since less is transferred over the network and the multiple files don't require multiple transfers.
- Jar files can be listed along with the applet tags in an HTML document and also placed in the CLASSPATH variable that Java uses to search for .class files.
- Jar files can be used with applications also.

Ex:

```
/*     FILE: Shapes6JAppletX.html     */

<html>
<applet code="Shapes6JAppletX.class"
        archive=Shapes6JAppletX.jar
        width=275 height=125>
</html>
```