Java, Pixels, and Rasters

- JAVA- a whirlwind tour
- Pixels and Rasters
- Project 1

- Sign up!
- Recitations start next week

Hearn & Baker - pp. 53-56 (Study)

Object-Oriented Programming

Java supports the Object-Oriented Programming (OOP) approach.

OOP focuses on the design of data-types rather than algorithms called *objects* or *classes*

Why is this distinction important?
Objects are nouns, Algorithms are verbs

Which is an easier task?
identify the actions that apply to a given noun
identify those objects to which a given action might be applied

What is easier?
identifying similar object classes
specifying general-purpose algorithms

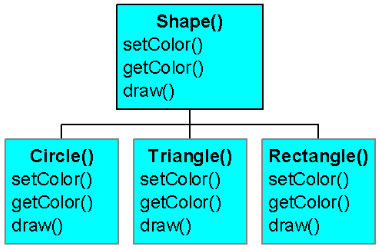
Advantages of OOP

- promotes design abstraction (hiding details)
- promotes code reuse
- consolidates specification and interface-- *encapsulation*

Example

Object-Oriented Design

- Similar object classes
- Common interfaces
- Common usage
- Code reuse-- *inheritance*
- Defers implementation and algorithm decisions



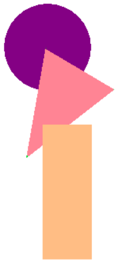

```
draw(object, color, arguments) {
  if (object is a Circle) {
  } else
  if (object is a Triangle) {
  } else
  if (object is a Rectangle) {
  }
}
```

Procedural Design

- Algorithm centered-- forces early implementation and algorithm decisions
- Exposes more details
- Difficult to extend
- Difficult to maintain

Polymorphism

Polymorphism allows algorithms to be specified in terms of the most general (and/or reasonable) object type



```
void processShape(Shape s) {
s.setColor(Color.red);
// ... do stuff
s.draw();
}

Circle c = new Circle(x, y, r);

Point2D v1 = new Point2D(x1, y1);
Point2D v2 = new Point2D(x2, y2);
Point2D v3 = new Point2D(x3, y3);
Triangle t = new Triangle(v1, v2, v3);

Point2D c1 = new Point2D(u1, v1);
Point2D c2 = new Point2D(u2, v2);
Triangle r = new Rectangle(c1, c2);

processShape(c);
processShape(t);
processShape(r);
```

Java Specifics

A Java source program is composed of
white space, *comments*, *declarations*, and *statements*.

White space is a very important part of any Java program

- has no effect on the code's meaning
- enhances readability and communicates structure
- includes, spaces, tabs, and new lines.
- typically, tabs or indention set off blocks of code
- new-lines separate statements

There aren't hard-fast rules on using white space.
Adopt a consistent style!

Lecture 2 Slide 5 6.837 Fall '98

A Java source program is composed of white space, comments, declarations, and statements.

White space is a very important part of any Java program. While it has no effect on the meaning of the code, white space, more than any other factor, contributes to the readability and signals the structure of a code fragment. White space includes, spaces, tabs, and new lines. Typically, tabs or indention is used to set off blocks of code, and new lines are used to separate statements. There are no hard-fast rules how and when to use white space, all I suggest is that you adopt a consistent style.

Comments

Next to white space, the most important parts of any Java program are its comments.

Inline comments:

```
answer = 42; // valid independent of the input
```

Block comments:

```
/*
    ... This next bit of code is so obvious that I hesitated to
    comment it a first. But, rather than risk any point reduction
    in this otherwise flawless program, I have decided to add
    this block comment with a recipe for a Gremolata garnish:

    2 tablespoons of finely chopped parsley
    1 minced clove of garlic
    ½ teaspoon of grated lemon rind

    Sprinkle this mixture on sauce, gravy, or computer
    programs during the last 5 minutes of preparation.
*/
question = "To be, or not to be".
```

Lecture 2 Slide 6 6.837 Fall '98

Next to white space, the most important part of any Java program is comments. Keep in mind the syntax of a language is designed for communicating a program's design to a computer. Comments, on the other hand, are the primary mechanism for communicating a program's design to humans. Java has three types of comments.

Inline comments:

```
answer = 42; // valid independent of the input
```

Block comments:

Special Comments

Java provides a special form of comment for generating automatic documentation, and web pages. It uses a tool called *javadoc*. Within these special comment blocks several special variables, indicated by a leading @, are recognized.

Documentation comments:

```
/**
 *  This interface is sure to make any Java object Cool. It
 *  defines no methods, but it adds a certain amount of positive
 *  karma to any class (not to mention that it yields an edible
 *  combination of mathematical constants).
 *
 *  @see java.hot
 *  @author Leonard McMillan
 *  @version 1.0
 */
public abstract interface Cool {
    public final static int KARMA = Math.PI + Math.E;
}
```

Lecture 2 Slide 7 6.837 Fall '98

Java also provides a special form of comment that is for generating automatic documentation, and web pages, for Java programs using a tool called *javadoc*. Within these comment blocks several special variables, indicated by a leading @, are recognized.

Documentation comments:

```
/**
 *  This interface is sure to make any Java object Cool. It
 *  defines no methods, but it adds a certain amount of positive
 *  karma to any class (not to mention that it yields an edible
 *  combination of mathematical constants).
```


Java Data Types

Variables in Java must be objects, arrays, or one of the primitive data types

Java has 8 primitive data types:

```
(boolean, byte, short, int, long, double, float, char)
```

Notes:

- All integer types are signed (and there is no `unsigned` keyword)
- Unlike C and C++ `char` variables are 16 rather than 8 bits
- Primitive data types are *passed by value*

```
boolean like6837 = true;       // boolean likewise have values {true, false}
byte teeth = 32;                       // bytes go from -128 to 127

if (!like6837) teeth = teeth - 1;

float pi = 3.141592f;          // you need the "f" here
short classes = 0xbad;         // hexadecimal
```

Lecture 2 Slide 8 6.837 Fall '98

All variables in Java must be declared as an object, an array, or a primitive data types.

Java has 8 primitive data types:

```
(boolean, byte, short, int, long, double, float, char)
```

- Primitive types are created by declaration if they are created at all
- Primitive types are passed to methods by value, their actual value of a primitive type can only be modified by the methods of the declaring or derived

Java Arrays

Java arrays fall between a primitive data type and an object

Properties of Java arrays:

- can contain objects, primitive types, or other arrays
- contents are accessed via zero-based positive integers enclosed within the brackets, [, and]
- declaration and creation of arrays are distinct operations
- created with the **`new`** operator
- uninitialized arrays have a value of **`null`**
- have a single instance variable called **`length`**
- checks at run time, that all accesses are between `0` and `length - 1`
- *passed to methods by reference*

```
byte buffer[];                  // array declaration (buffer = null)
buffer = new byte[1024];        // array creation (contents initialized to zeros)
int table[] = new int[10];      // declaration and creation combined

int sqrs[] = {0, 1, 4, 9, 16, 25, 36, 49, 64, 81 };            // with an initializer
buffer[5] = sqrs[sqrs.length-1];        // array references

int triangle[][] = new int[3][];
triangle[0] = new int[3];
triangle[1] = new int[2];
triangle[2] = new int[1];
```


In Java arrays fall somewhere between a primitive data type and an object

- Arrays of any object, primitive type, or arrays can be formed
- Array contents are accessed via zero-based positive integers enclosed within the bracket operators, [, and]
- Arrays are passed to methods by reference, the actual values in any array may be changed any methods that receives it as an argument
- Declaration and creation of arrays are distinct operations in Java
- Like objects, arrays are created with the `new` operator, and uninitialized arrays have a value of `null`
- Arrays have a single instance variable called `length`

Java Objects

Objects are containers around aggregates of primitive types, arrays, and other objects

Properties of Classes

- **All** Java code is contained within objects
- Objects are *passed to methods by reference*
- Definition, declaration, and creation of objects are distinct
- Instances of objects are created with `new` and have a value of **`null`**
- Objects have four forms: the **class**, the **abstract class**, the **inner class**, and the **interface**

```
class Circle {                                      // object definition
    static double pi = 3.141592;                    // class variable
    double radius;                                  // instance variable
    public Circle(double r) {                       // constructor method
        radius = r;
    }
    public double circumference() { return 2*pi*radius; }
    public double area() { return pi*radius*radius; }
}

Circle c;                                   // object declaration
c = new Circle(4.0);                // object creation
double a = c.area();
```


Java Statements

Statements are all that remains!

- Individual statements are terminated by semicolons
- Blocks of statements are grouped using braces, {, and }
- Simplest statement is the *expression*.
  An expression is a string of variables and constants separated by operators.

Java allows the following set of operators within statements:

```
. [] ()
++ -- ! ~ instanceof
* / %
+ -
<< >> >>>
< > <= >=
== !=
&
^
|
&&
||
?:
= *= /= %= += -= <<= >>= >>>= &= ^= |=
```


The final component of a Java program is its statements. Individual statements are terminated by semicolons. Groups of statements and declarations enclosed within braces, { and }, are called **block statements** and they act like an individual statement. Block statements do not require a terminating semicolon. The simplest type of statement is the *expression*. Java allows the following set of operators within statements:

```
. [] ()
++ -- ! ~ instanceof
* / %
+ -
<< >> >>>
< > <= >=
== !=
```


Control Statements

The second type of statement is the *control statement*. The following list illustrates some of the control statements available in Java:

```
if (BooleanExpression) Statement

if (BooleanExpression) Statement else Statement

while (BooleanExpression) Statement

do Statement while (BooleanExpression)

for (Expression; BooleanExpression; Expression) Statement

break <Label>

continue <Label>

return Expression

Label: Statement

switch (IntegerExpression) {
    case IntegerValue:
            Statements
        default:
            Statements
}
```


The second type of statement is the *control statement*. The following list illustrates some of the control statements available in Java:

```
if (BooleanExpression) Statement

if (BooleanExpression) Statement else Statement

while (BooleanExpression) Statement

do Statement while (BooleanExpression)

for (Expression; BooleanExpression; Expression) Statement
```

A First Java Program

We start by writing what is called a Java *applet*. This is a form of a Java program that can be embedded in a HTML document and accessed from the web. An example HTML document that calls our example applet is shown below:

```
<HTML>
<HEAD>
<TITLE>Demo Applet</TITLE>
</HEAD>

<BODY>
<H1>Demo Applet</H1>
<P>My favorite class is 6.837</P>
<HR>
<CENTER>
<APPLET code="Demo.class" width=200 height = 200>
</APPLET>
</CENTER>
<HR>
</BODY>
</HTML>
```

The highlighted lines add the Java code to our document. Notice the *.class* extension. All Java source code files should end with a *.java* extension. The source is here.

We will start by writing what is called a Java applet. This is a form of a Java program that can be imbedded in a HTML document and accessed from the web. An example HTML document that calls the example applet is shown below:

Demo.htm

The highlighted lines add the Java code to our document. Notice the *.class* extension. All Java source code files should end with a *.java* extension. When a Java program is compiled one or more *.class* files are generated. Unlike, C and C++ the number of *.class* files generated by a compile depends on how many objects are defined in the source file. You can get a copy of this HTML file if you like. You might have to change the extension to *.html*.

Java Example

Next, the source of the *demo.java* file is shown.

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

public class Demo extends Applet {
        Image image;
        int count;

        public void init()
        {
            image = getImage(getDocumentBase(), "World.jpg");
            count = 1;
        }

        public void paint(Graphics g)
        {
            g.drawImage(image, 0, 0, this);
            g.setColor(Color.red);
            for (int y = 15; y  1) ? "s" : ""));
            count += 1;
        }
}
```

You can get the source here.

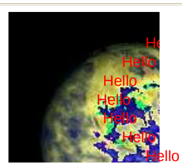
Since all Java code must be contained within an object, an object called *Demo* is defined in this file (here is a copy). This object is derived from a previously defined object called *Applet*. The Applet object encapsulates all of the basically functionality provided to, and available from, a web page. For more details about the methods available to an Applet see pages 233-236 of **Flanagan**. Often objects are extended in order to add new functionality, in the form of methods. In this case we aren't adding any new methods, instead we are *overriding* two existing methods, called *init( )* and *paint( )*, that are inherited from the Applet base class. Thus, the Demo object that we define behaves much like an Applet, the only difference is in the modifications to the overridden classes. You don't even need to know any of the details of how a class works, beyond the interfaces to methods, in order to modify the original code. This is a simple example of the power of object-oriented programming.

The *Demo* class declares two new *instance* variables. The first variable declared is an *Image* object with the name *image*. Note that this declaration does not actually create an *Image* object. Instead the *Image* is created by the first line of the *init( )* method. The second instance variable is a primitive *int* type named

The Applet in Action

Demo Applet

My favorite class is 6.837

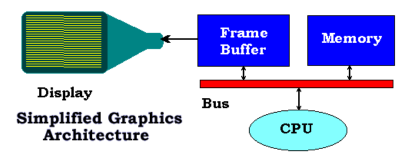


This is all of the Java programming language that we will specifically cover in class.

If you have never used Java before, you might want to consider buying one of the books discussed on the course's home page.

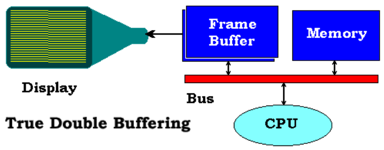
Thus far we have only discussed the Java language. Next time, we will look at Java development environments.

Review of Raster Displays



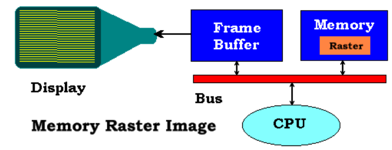
- Display synchronized with CRT sweep
- Special memory for screen update
- Pixels are the discrete elements displayed
- Generally, updates are visible

High-end Graphics Display System



- Adds a second frame buffer
- Swaps during vertical blanking
- Updates are invisible
- Costly

Implementing a Memory Raster Object



- Maintains a copy of the screen (or some part of it) in memory
- Relies on a fast copy
- Updates are *nearly* invisible
- Conceptual model of a physical object

A Java Model of a Memory Raster

We begin with some boilerplate...

- Indicates that we will use some of the classes (objects) in these packages
- Corresponds to a directory structure
- Directory hierarchies might be *zipped* up

```
import java.awt.*;
import java.awt.image.*;

class Raster {
    public int width, height;
    public int pixel[];
```

- Define a new class (extends *Object* by default)
- Declare instance variables
- Permissions (public, protected, private, private protected, *default*) see Flanagan pg. 73

Here is the source for the Raster object

More Raster Object

```
/////////////////////////// Constructors //////////////////////

/**
 *  This constructor, which takes no arguments,
 *  allows for future extension.
 */
public Raster()
{
}

/**
 *  This constructor creates an uninitialized
 *  Raster Object of a given size (w x h).
 */
public Raster(int w, int h)
{
    width = w;
    height = h;
    pixel = new int[w*h];
}
```

- Constructors are called when an instance of this class is created (via *new*)
- Example of Overloading
- You should almost always have a constructor that takes no arguments

Lets talk about Pixels

- Pixels stored as *int*s in the array called *pixel[]*
- Java's standard model of a pixel

Alpha	Red	Green	Blue

The four bytes of a 32-bit int
Alpha of 0 is Transparent
Alpha of 255 is Opaque

```
/**
 *  This constructor creates an Raster initialized
 *  with the contents of an image.
 */
public Raster(Image img)
{
    try {
        PixelGrabber grabber = new PixelGrabber(img, 0, 0, -1, -1, true);
        if (grabber.grabPixels()) {
            width = grabber.getWidth();
            height = grabber.getHeight();
            pixel = (int []) grabber.getPixels();
        }
    } catch (InterruptedException e) {
    }
}
```


Raster Methods

```
///////////////////////// Methods ////////////////////////

/**
 *  Returns the number of pixels in the Raster
 */
public int size( )
{
    return pixel.length;
}

/**
 *  Fills a Raster with a solid color
 */
public void fill(Color c)
{
    int s = size();
    int rgb = c.getRGB();
    for (int i = 0; i
```


More Raster Methods

```
/**
 *  Gets a color from a Raster
 */
public Color getColor(int x, int y)
{
    return new Color(pixel[y*width+x]);
}

/**
 *  Sets a pixel to a given value
 */
public boolean setPixel(int pix, int x, int y)
{
    pixel[y*width+x] = pix;
    return true;
}

/**
 *  Sets a pixel to a given color
 */
public boolean setColor(Color c, int x, int y)
{
    pixel[y*width+x] = c.getRGB();
    return true;
}
}
```


Example Use: Rastest.java

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

public class Rastest extends Applet {
    Raster raster;
    Image output;

    public void init() {
        raster = new Raster(getImage(getDocumentBase(), getParameter("image")));
        output = raster.toImage();
    }

    public void paint(Graphics g) {
        g.drawImage(output, 0, 0, this);
    }

    public void update(Graphics g) {
        paint(g);
    }

    public boolean mouseUp(Event e, int x, int y) {
        int s = raster.size();
        for (int i = 0; i
```

Raster Demonstration

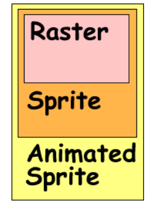
```
<HTML>
<HEAD>
<TITLE>Raster Test Applet</TITLE>
</HEAD>

<BODY>
<H1>Raster Test Applet</H1>
<HR>
<CENTER>
<APPLET code="Rastest.class" width=160 height = 160>
<param name="image" value="girls.jpg">
</APPLET>
</CENTER>
<HR>
</BODY>
</HTML>
```

Raster Test Applet

Try it yourself

- Rastest.java source
- Rastest.html source

Next Time



- More tricks with rasters
- IDEs and RADs
- Project #1