ICSI201
Intro. to CS
Spring 14
Last week

- Proj6 wrapup
- Miscellaneous topics
- Summary for final studying
Proj6 Wrapup

• V4: Swap pictArray[0] <-> pictArray[param]
• V5 (obvious bug)

```javascript
this.likeCounts[pick] =
    this.likeCounts[pick] + 1; // Count!
```

• V5 1 **FIND** the **Max**!
  2 **find WHERE** the Max(s) is or are
(1 **MUST BE DONE FIRST**!)
  3 when copying each Picture,
    if( it is a Max )
    {
        paint a pretty border
    }
(1 **MUST** BE DONE FIRST, 2 is smart to
do first so you know you did 1 right.)
1. Parallel arrays:

- life counts
- pictArray

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>0</th>
<th>7</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>[07]</td>
<td>[27]</td>
<td>[37]</td>
<td>[47]</td>
<td>[57]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3531</th>
<th>4258</th>
<th>5352</th>
<th>9921</th>
<th>1038</th>
<th>4180</th>
</tr>
</thead>
<tbody>
<tr>
<td>[07]</td>
<td>[11]</td>
<td>[27]</td>
<td>[37]</td>
<td>[47]</td>
<td>[57]</td>
</tr>
</tbody>
</table>

- parallel principle: Whenever (always!) swapping two popularity counts, also swap the corresponding picture addresses!
**Showoff**

- Your goal!
  - Before swap with first (3)
  - After swap with first (3)

- 3521
- 47324

- Don't copy colors of pixels!
Proj6 Wrapup

- V6 Sort THE ADDRESSES of the Pictures according to popularity... keep the likecount values consistent with the order of the Picture (addrs).
  (Clever idea: Do it all by swapping!)

- (10%) 1 Determine wall width, height from the Picture dimensions
  2 MAKE (new EditablePicture)
  3 copyIn the Pictures, nicely centered.
\begin{itemize}
  \item \( i = 1 \)
  \item \( j = 0 \)
  \item \( C[i] = 7 \)
  \item \( C[j] = 1 \)
  \item \( C[i] < C[j] \) \text{ (YES)}
  \item \( \text{DO SWAP} \)
  \item \( i = 2 \)
  \item \( \text{DO SWAP} \)
  \item \( j = j + 1 \)
  \item \( j = 3 \)
  \item \( C[i] = 7 \)
  \item \( C[j] = 1 \)
  \item \( C[i] < C[j] \) \text{ (NO)}
  \item \( \text{DO NOT SWAP} \)
  \item \( i = 3 \)
  \item \( \text{DO SWAP} \)
  \item \( j = j + 1 \)
  \item \( j = 4 \)
  \item \( C[i] = 7 \)
  \item \( C[j] = 1 \)
  \item \( C[i] < C[j] \) \text{ (NO)}
  \item \( \text{DO NOT SWAP} \)
\end{itemize}
i = i + 1;

7

(1 0 3 6 7)

j = j + 1; j = 3

NO!

3 0 1 6 7

j = j + 1

YES!

6 0 1 3 7

j = j + 1

YES

7 7 0 1 3 6

Use the value of i = 1 to restrict the scan to biggest to only what is after i = 1.
i = 1

1 = i + 1

i = 2

j = j + 1

j = 3

3 0 1 6

j = 5

3 0 1 6

j = j + 1

j = 6

STOP THE INNER LOOP

j < this like count = length

j is False 6
(10% flexible sizes and nice centering)

1. Determine wall width(w), height(h) from ALL the Pictures' dimensions
   (You choose a between-Pict. space amount.)
2. MAKE (new EditablePicture(w,h);)
   (1 must be done FIRST, because Pictures, like arrays, cannot be resized, and the dimensions must be known)
3. copyIn the Pictures, nicely centered.
   (2 must be done FIRST: copyIn cannot be done without a destination.)
(See other scans about centering within last week's slides)
Review (for final) about arrays

- An array is many* variables (which implies an array stores many values). A single variable stores only one value at a time (* “many” could be 0 or 1)

- Each variable HAS THE SAME TYPE.

- Each variable is called an element.

- Each variable (or element) is referred to by 2 things: arrayName[ indexValue ] example:
  
  this.likeCounts[ i ] = this.likeCounts[ i ] + 1;
  
  NON-ARRAY variable example:  N = N + 1;

- (In Java and computer hardware) HOW MANY ELEMENTS (the length) is FIXED when the array is made. ( eg. new int[ 356 ]; )

- We program loops to work on all the elements.
Some review and misc. remarks re. objects

- An object is many* variables (which implies an object stores many values). A single variable stores only one value at a time (* “many” could be 0 or 1)

- The variables CAN HAVE DIFFERENT TYPES (or have the same type).

- Each variable is called a field or instance variable.

- **(STUDY!)** Each field is referred to by 2 things: objectAddress.fieldName examples:

  ```
  this.owner = "Prof. Chaiken";
  neighbor.nHDTVs = neighbor.nHDTVs - 1;
  ```

- **(STUDY!)** Methods (chunks of code) can be defined in an object's blueprint to express beautiful* or elegant* software designs. (* nothing prevents ugly designs.)
More review and important facts TO STUDY

- The Mad Ph.D. video showed objects made from the raw materials of brick, wood, glass, etc.
  - The blueprint or plan is the class.
  - Study how to extend a class like Picture by adding methods of your own creation, to program operations you want the computer to do on or with the objects made according to the class.
    - DEFINING A METHOD: KNOW HOW!
    - MAKING AN OBJECT (with new): KNOW HOW!
    - CALLING A METHOD ON A MADE OBJECT: KNOW HOW!

- Question: What is the raw material used for making a Java, computer object?
  - Hint: It is the same raw material used for arrays and variables.
A BORROWED CHUNK OF COMPUTER MEMORY

new ClassName( ) does the borrowing.

Misc. remark: The loan is returned when your program stops running, or the Java garbage collector figures out that your code can never again use the object's address.
Spr14 201's common basic software design or architecture choice:
One class is defined to be the CONTAINER FOR a Java Application (App) (that's the one class with a main)

You define one or more other classes so you can re-use book Classes as plans PLUS ADD methods you (struggle and learn to) write yourself.
You define one or more other classes so you can re-use book Classes as plans PLUS ADD methods you (struggle and learn to) write yourself.

ALSO

Add a plan to borrow memory like

```java
Picture[ ] pictArray = new Picture[6];
```

to make your App more cool.

PURPOSE (example): acquire and store 6 Pictures so they can be shown off, voted for, be re-arranged and shown off with borders on some.
Add a plan to borrow memory like
`int[] likeCount = new int[6];`
to make your App more cool.
PURPOSE (example): store 6 counts of how many likes each of the 6 Pictures gets, so they can re-arranged by popularity and shown off with borders on the most popular.
You can define the same class to BOTH contain your App's main AND be a blueprint for an object stores and has methods to process organized data.

Example: Lab13 class ArtilleryRange DOES ALL 3 THINGS:
1. It contains main
2. Re-uses the Picture book class software.
3. Adds to a Picture some organized data tables (as arrays) plus a method \( \text{fire( double angle )} \) that preprograms a difference-table computation much like that done by the computing Rosies of World War II, and plots the trajectory on a window.

The difference-table computation SIMULATES a projectile projected and falling under the influence of gravity and air resistance.
Miscellaneous remarks
You can define the same class to BOTH contain your App's main AND be a blueprint for an object stores and has methods to process organized data.

This software design or architecture choice is wise in these situations:

(1) The software is very simple. (Like Lab13)

(2) You decide or are required to maintain a UNIT TESTING App to help test over and over your programming of one class designed to be just a little piece of much more complex software.
instance methods vs static methods
The Math class

• Google it: Java API Math You can get accurate estimates of pi and e from Math.PI and Math.E

• The Math class is a shopping bag holding a lot of methods that are called with numeric parameters, do popular mathematical calculations (for scientists, economists, sometimes business people).

• ALL the methods in Java's Math class are static
  – It makes no sense to force people to call them ON an object.
Instance methods

- Almost all the methods we covered: `forward()` (of Turtle), `show()` (of Picture and of Turtle), `getWidth()`, `getHeight()` and `getPixel` (of Picture), `setColor()` (of Pixel) are instance methods

They are ONLY called ON some object (like the Turtle, Picture, or Pixel) on which calling the method makes sense.
static methods

- FileChooser.pickAFile(), Math.sqrt(), main(), weAv(N1, W1, N2, W2)

static or class methods

They are ONLY called BY THEMSELVES, NEVER ON object (like the Turtle, Picture, or Pixel) on which calling the method makes sense.
class MyApp {
    public static void main(String a[]) {
        EPicture p = new EPicture(640, 480);
        p.red1(320, 240);
        p.explore();
    }
}

class EPicture extends Picture {
    //LOTS OMITTED..You'll write:
    public void red1(int x, int y);
    {
        this.getPixel(x,y)
            .setColor(Color.red);
    }
}
class EPicture extends Picture {
    public void red1(int x, int y) {
        this.getPixel(x,y).setColor(Color.red);
    }
}

SAME thing done with a static method!!

public static void picRed1(Picture theP, int x, int y) {
    theP.getPixel(x,y).setColor(Color.red);
}
class EPicture extends Picture {
    public void red1(int x, int y) {
        this.getPixel(x, y).setColor(Color.red);
    }
}

SAME thing done with a static method!!

public static void picRed1(Picture theP, int x, int y) {
    theP.getPixel(x, y).setColor(Color.red);
}
```java
class EPicture extends Picture {
    public void red1(int x, int y) {
        this.getPixel(x, y).setColor(Color.red);
    }
}

picRed1(p, 320, 240); // Caller code.

// SAME thing done with a static method!!
public static void picRed1(Picture theP, int x, int y) {
    theP.getPixel(x, y).setColor(Color.red);
}
```
What about the method call

\[
\text{Math.abs( } -34.2 \text{ )?}
\]

This looks like calling a static method ON the Math object!

No! Syntactically and superficially it looks like Math refers to an object on which abs( -34.2 ) is called, but Math is the name of a class, not a variable referring to an object.

abs( ) is a static method in the Math class.

It's a static (also called class) method, so calling it on an object is impossible!
The calling card

When the CALLER CALLS a method, he/she WRITES all the important information in computer memory in a space called the STACK FRAME. I call it the calling card.
The calling card
What the caller WRITES:

1. Any and all the PARAMETER VALUES

2. WHERE within the caller's Java code TO CONTINUE EXECUTING AFTER the callee does the return thing.

Sometimes, the callee (the method code) writes the RETURN VALUE on the calling card.
REMARK
The calling cards are stored in the computer like a STACK OF pancakes or dishes in a cafeteria.

Last ones In --- are the First ones Out.
Accounting majors call that LIFO.
In CSI310, you call the calling cards stack frames.
t.drawL(3.0); //Caller code.

class Turtle {
    public void drawL (double scale) {
        this.forward((int)(100*scale));
    }
}

makeTurtleDrawL(t, 3.0);//Caller

SAME thing done with a static method!!

public static void makeTurtleDrawL (Turtle theT, double scale) {
    theT.forward((int)(100*scale));
}
In BOTH the familiar instance way AND the static method way

The caller writes 3 things on the calling card.

Calling Card

(1) where to get at the Turtle Prof. Chaiken's desk
(2) number for how big to draw 3.0
(3) whom to tell when you're done

Java code just after ...3.0);
In BOTH the instance method way AND the static method way

**Calling Card**

1. where to get at the Turtle
   - **Prof. Chaiken's desk**
2. number for how big to draw
   - **3.0**
3. whom to tell when you're done
   - Java code just after ...3.0);

(Java system doesn't let programmers see the 3rd box)
static means: If you want the method to do something involving an object like a Turtle, you must code into the method

    AN EXPLICIT PARAMETER
    (like theT I made up)
referring to it (the object like a Turtle.)

“not-static” means: The method has an IMPLICIT PARAMETER named this
for the above purpose, so you don't have to make one up.
Object Oriented Programming
One of it's ideas.

Suppose a method like drawT( int size ) makes sense ONLY when applied to an object of a particular kind, like a Turtle.
Suppose a method like `drawT(int size)` makes sense ONLY when applied to an object of a particular kind, like a Turtle.

Object Oriented Programming
One of it's ideas.
IDEA: It's BETTER to force people (professional software developers) to (A) code CALLERS
someTurt.drawT(24) instead of
drawT(someTurt, 24)
(B) put `drawT` IN the def. of Turtles.
When a CALLER calls an instance (non-static) method, an object to CALL it ON is:

(A) Always required.
(B) Can be there sometimes, but not always.
(C) Is Never Allowed.
class Turtle {
    public void drawI(int size) {
        this.forward((int)2.7*size);
    }
    public void drawT(int size) {
        this.drawI(size); //A
        drawI(size);     //B
    }
}

Line A and Line B mean exactly the same thing! this refers to the object on which the instance method is called. It is implicit when an instance method of one class (Turtle) calls another instance method of the same class (Turtle).
When a CALLER calls an static (class) method, an object to CALL it ON is:

(A) Always required.
(B) Can be there sometimes, but not always.
(C) Is Never Allowed.
class MyApp {
    public static void main(String a[]))
    {
        World w = new World();
        Turtle t = new Turtle(w);
        t.drawL(3.0);
    }
}

class Turtle {
    //LOTS OMITTED..You'll write:
    public void drawL(double scale)
    {
        this.forward(((int)(100*scale)));
    }
}
```java
t.drawL(3.0); //Caller code.

class Turtle {
    public void
don void drawL
    (double scale)
    {
        this.forward(((int)(100*scale)));
    }
}

makeTurtleDrawL(t, 3.0); //Caller

old fashioned kind of method

static void makeTurtleDrawL
    (Turtle theT, double scale)
{
    theT.forward(((int)(100*scale)));
}
```
t.drawL(3.0); //Caller code.

class Turtle {
    public void
    void drawL
        (double scale)
    {
        this.forward((int)(100*scale));
    }
}

makeTurtleDrawL(t, 3.0); //Caller static method

static void makeTurtleDrawL
    (Turtle theT, double scale)
{
    theT.forward((int)(100*scale));
}