**CMSC 131: Chapter 12 (Supplement)**

**Interfaces and the Picture Library**

### Fundamentals of Digital Images

**Digital Image**: A digital picture (also called a digital image) is stored as a two-dimensional array of small regions called pixels.

**Pixel**: ("picture element") contains a representation of a color.

**Color**: There are many different ways to represent a color. The most common way for computer monitors is based on the three (additive) primary colors: red, green, and blue, also called RGB.

**Color components**: Each RGB color component is typically stored a single integer ranging from 0 to 255.

### Fundamentals of Digital Images

**Color Rescaling**: Because the discrete range [0..255] is messy to work with computationally, it is common to scale each integer color component to a double ranging from 0.0 to 1.0:

\[
\text{double doubleRed} = \left( \frac{\text{intRed}}{255.0} \right);
\]

**Color values**: 0.0 = darkest; 1.0 = brightest.

**Some Common colors**:

<table>
<thead>
<tr>
<th>R</th>
<th>G</th>
<th>B</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>White</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Red</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Green</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Blue</td>
</tr>
</tbody>
</table>

**Mixtures**: (0.5, 0.5, 0.5) = Gray; (0.23, 1.00, 0.51) = Lime green.
Fundamentals of Digital Images

Picture Structure: Pixels are arranged in a 2-dimensional grid.

Width and height: are measured in terms of the number of pixels in this grid, horizontally and vertically.

Pixel index: Each pixel is specified by giving its index, which indicates its row and vertical column numbers in this grid. Pixel indices start with (0, 0) in the upper left corner.

Row indices: increase as we move down.

Column indices: increase as we move to the right.

cmsc131PictureLib

cmsc131PictureLib: This package contains a number of classes for creating, accessing, and displaying images. Major elements:

Picture: An interface for specifying the content of a picture.

PictureUtil: Contains a number of utilities, most important the method show( p ), for displaying a Picture p.

PictureColor: Represents the color of a picture pixel. It stores the RGB color components as doubles in the range 0.0 to 1.0.

Constructor: is given the RGB values (as doubles) as its arguments:

PictureColor myPixel = new PictureColor( 0.23, 1.0, 0.512 );

Accessors: getRed( ), getGreen( ), and getBlue( ), return the respective RGB color components for the pixel (as doubles):

double r = myPixel.getRed( );       // set r = 0.23
System.out.println( myPixel.getBlue( ) ); // outputs: 0.512
Creating a Picture

**Picture:** To create a pixel you need to specify three things: its width, its height, and the color of each pixel.

**Picture interface:** Specifies the three public methods that any Picture class must implement:

- `int getWidth( )`: returns the width (w) of the picture in pixels.
- `int getHeight( )`: returns the height (h) of the picture in pixels.
- `PictureColor getColor( int x, int y )`: returns the color (as a PictureColor object) of the pixel at column x and row y. (Remember that the (0,0) corresponds to the upper left corner.)

**Creating a Picture**

**Step 1:** You create a class that implements the Picture interface. This means that it contains the methods `getWidth( )`, `getHeight( )` and `getColor( )`.

**Step 2:** You create a new object p of this type.

**Step 3:** Call `PictureUtil.show( p )`. Your picture will appear!
**How it Works**

_Huh? How can we define an entire image by just implementing three little methods (getHeight, getWidth, getColor)?_

You might ask, “Where is the 2-dimensional grid of pixels?”

**Answer:** You do not need to; it is generated by PictureUtil.show().

**PictureUtil.show(p):**

- invokes `p.getWidth()` and `p.getHeight()` to determine the size of the new image;
- asks Java to create an image of this size. (This creates the 2-dimensional grid.)
- fills in the pixel colors of this image, row-by-row and column-by-column. To get the color of each pixel, it calls `p.getColor(x, y)`, which returns the desired color.
- when all the pixels have been filled in, it asks Java to display it.

**Examples**

To illustrate this, consider a few examples of Picture classes. Remember, each one needs to define: `getWidth`, `getHeight`, and `getColor`.

- **RedSquare**: Draws a solid red square.
- **FrenchFlag**: Draws a French flag.
- **Inverse**: Complements the colors of all pixels (as seen in HW3).
- **FlipLeftRight**: Produces a mirror image of a picture, by flipping it left to right.
RedSquare

RedSquare: Has a fixed width (150) and height (150), and every pixel is defined to be red. It has no instance data, and no need for a constructor.

We could create our own red color using:

```
    PictureColor red = new PictureColor( 1, 0, 0 );
```

But the PictureColor class defines a number of common colors:

```
    PictureColor.RED
    PictureColor.GREEN
    PictureColor.BLUE

    (...and also WHITE, BLACK, CYAN, MAGENTA, YELLOW, and GRAY).
```

RedSquare

Defining it:

```
    public class RedSquare implements Picture {
        public int getWidth( ) { return 150; } 
        public int getHeight( ) { return 150; } 
        public PictureColor getColor( int x, int y ) { 
            return PictureColor.RED;
        }
    }
```

Displaying it: (from your Driver main method):

```
    RedSquare redSquare = new RedSquare( );
    PictureUtil.show( redSquare );
```
FrenchFlag

FrenchFlag:

- **Constructor**: takes the width (in pixels) as an argument, and saves it in the instance variable `width`.

- **Height**: is set to 3/4 (75%) of the width, and is saved.

- **getWidth, getHeight**: access the width or height instance variables.

- **getColor(x, y)**: returns
  
  - blue in the leftmost third (that is, for `x < width/3`),
  - white in the center third (for `x` from `width/3` to `2*width/3`), and
  - red in the rightmost third.

```java
public class FrenchFlag implements Picture {
    private int width;
    private int height;

    public FrenchFlag(int wid) { // constructor: height = 3/4 * width
        width = wid;
        height = (3 * width) / 4;
    }

    public int getWidth() { return width; }
    public int getHeight() { return height; }

    public PictureColor getColor(int x, int y) {
        if (x < width/3) return PictureColor.BLUE;
        else if (x < 2*width/3) return PictureColor.WHITE;
        else return PictureColor.RED;
    }
}
```
FrenchFlag

How does it work?

public PictureColor getColor( int x, int y ) {
    if ( x < width/3 ) return PictureColor.BLUE;
    else if ( x < 2*width/3 ) return PictureColor.WHITE;
    else return PictureColor.RED;
}

As PictureUtil.show( ) fills in pixel colors starting with row y=0 and generates x = 0, 1, 2, ..., 149:

g getColor( 0, 0 ) → (0 < 50): yes → BLUE
getColor( 1, 0 ) → (1 < 50): yes → BLUE
...  
getColor(50, 0) → (50 < 50): no!
      (50 < 100): yes → WHITE
...  
getColor(100, 0) → (100 < 50): no!
      (100 < 100): no!
else → RED
**Inverse**

**Inverse**: Is given a base image in its constructor and creates an image of the same size, but each pixel is replaced with its **color complement**.

This is done by replacing each color component $c$, which ranges from 0.0 up to 1.0, with $(1.0 - c)$.

**Example**: The yellowish tones of the cat’s fur is close to the RGB color (1.0, 1.0, 0.0), and is mapped to (0.0, 0.0, 1.0), which is blue.

```java
public class Inverse implements Picture {
    private Picture base;
    public Inverse( Picture basePicture ) { base = basePicture; }
    public int getWidth( ) { return base.getWidth( ); }
    public int getHeight( ) { return base.getHeight( ); }
    public PictureColor getColor( int x, int y ) {
        PictureColor color = base.getColor( x, y );
        return new PictureColor(
            1.0 - color.getRed( ),
            1.0 - color.getGreen( ),
            1.0 - color.getBlue( ) );
    }
}
```

... Image cuteKitty = new Image( imageName );
Inverse psychoKitty = new Inverse( cuteKitty );
PictureUtil.show( psychoKitty );
...
**FlipLeftRight**

*FlipLeftRight*: transforms an image by creating a *mirror image* (flipped left to right).

**Constructor**: is given a base image, and saves it.

**getWidth, getHeight**: access the base image's width or height.

**getColor**: To generate the color of a given pixel \((x, y)\), access the pixel of the same row \((y)\) in the base image, but reverse the column index \((x)\) by subtracting \(x\) from \(width-1\).

```
public class FlipLeftRight implements Picture {
    private Picture base;

    public FlipLeftRight( Picture basePicture ) { base = basePicture; }

    public int getWidth( ) { return base.getWidth( ); }
    public int getHeight( ) { return base.getHeight( ); }

    public PictureColor getColor( int x, int y ) {
        int flipX = base.getWidth( ) - 1 - x;
        return base.getColor( flipX, y );
    }
}
```

```
Image cuteKitty = new Image( imageName );
FlipLeftRight sinisterKitty = new FlipLeftRight( cuteKitty );
PictureUtil.show( sinisterKitty );
```
### Avoiding “Off-By-1” Errors

**Off-by-1**: Being “off by 1” is a very common programming error.

**Why did we use width-1?**

\[
\text{flipX} = \text{width} - 1 - x; \quad (\text{rather than}) \quad \text{flipX} = \text{width} - x;
\]

Being off by 1 is very common, but easy to avoid. Just consider the two extreme cases of the largest and smallest possible values of \(x\). We know that the column indices range from \([0..\text{width}-1]\).

<table>
<thead>
<tr>
<th>(x)</th>
<th>(\text{width} - x)</th>
<th>(\text{width} - 1 - x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>min:</td>
<td>0</td>
<td>(\text{width}) (NO)</td>
</tr>
<tr>
<td></td>
<td>(\text{width} - 1)</td>
<td>(0) (YES)</td>
</tr>
<tr>
<td>max:</td>
<td>(\text{width} - 1)</td>
<td>1 (NO)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 (YES)</td>
</tr>
</tbody>
</table>

If calculations work correctly in the extreme cases, they usually work correctly for the cases in between.