# Homework # 1
# due Tuesday, September 10, 10:00 PM

In this assignment, you will implement an ADT for a raster picture system that uses pixels as picture elements.

## 1 ADT: Pixel

For this homework, you are creating a data type to represent a pixel. A pixel has an x- and y-coordinate, as well as a color. The coordinates of a pixel may not be changed, although its color can be.

The implementation of this ADT makes use of Java's `Point` and `Color` classes. Make sure to look them up and understand how to use them. Especially understand the `rgb` values used by `Color`.

Each pixel has two coordinates: `x` increases as you go to the right, and `y` increases as you go down. We allow a pixel to have coordinates of any integer value.

For `Pixel`, you must implement the following methods:

`Pixel(Point,Color)` Main constructor.

`Pixel(int,int,Color)` Alternate constructor.

`Pixel(Point)` Alternate constructor, using a default color.

`Pixel(int,int)` Alternate constructor, using a default color.

`toString()` Return a `String` representation of the pixel. The string should take the form `<x,y,color>` where "color" is the `String` representation of a `Color` object. Note that the `String` representation of `Color` is not always consistent across different systems, so the tests make no assumptions about it.

`equals(Object)` Return true if the argument is another `Pixel` with the same coordinates and same color. Otherwise, return false. Make use of `instanceof`.

`hashCode()` Return a combination of the coordinates and color subject to the constraints described below.

`loc()` Return a new `Point` that describes the coordinates of the pixel.

`color()` Return the `Color` object of the pixel.

`setColor(Color)` Set the color of the pixel.

`invert()` Invert the color of the pixel, such that the new rgb values are 255 minus the old rgb values.

The constructors and the `setColor` method will have to check for illegal arguments and throw the appropriate exception. Note that the pixel must have a non-null location and a non-null color.

You are also given some methods that are used for drawing a pixel. Make sure you look at them.

## 2 Raster

A raster is an object that holds Pixels in a 2d-array. Each pixel should be stored in the array according to its x- and y-coordinates. A still picture can be represented by a raster of pixels.

For `Raster`, you must implement the following methods:

`Raster(int,int)` Main constructor.

`toString()` Return a `String` representation of the raster. The string should take the form "`x by y raster: ...`" where x and y are the dimensions of the raster, and the string is followed by the string repesentations of all the pixels in the raster, in column order, separated by spaces, e.g. `1 by 2 raster: <0,0,red> <0,1,blue>`.

`equals(Object)` Return true if the argument is another raster with the same dimensions and each pixel in each raster is pairwise equal. Otherwise, return false.

`addPixel(Pixel)` Add the pixel to the raster. This should fail if the pixel is illegal for this raster. Also, this should return false if adding the Pixel did not cause a change, and true otherwise.

`getPixel(int,int)` Return the pixel at this location. This should fail if the coordinates are out-of-bounds.

`draw()` Draw the raster. This can be accomplished by having each pixel draw itself.

# 3 hashCode

A hash code is an integer that can be used to represent a particular object. Your hashCode function should return an integer for your pixel based on its particular data. Two Pixels that are equal should always produce the same hash code. Pixels that are different can, and often should, produce different hash codes. However, we can't ensure this; there are not enough different ints to identify all of the possible Pixels uniquely. Therefore, there must be some different Pixels that have the same hashCode.

A Pixel can have any int as x- and y-coordinates, and any Color that Java permits. It's impossible to create a hashCode that would produce a unique code for all possible pixels. Instead, we will design a hashCode method that WILL create differing hash codes under certain circumstances: either 1) The two Pixels have coordinates in the range of [0,255] and their coordinates do not match, or 2) The two Pixels have rgb values in the [0,255] range, and their colors differ substantially in at least one of the color components (r, g, or b). Note that our design is bad, because it has the undesirable property that observably similar Pixels have a similar hash code.

Because our hashCode returns an int, we only have 32 bits to work with. We assign 8 bits to the x coordinate, and 8 to the y, so we can represent the range of [0,255]. This leaves us with 16 bits for the colors. As we assume rgb values are represented as ints in the range [0,255], we don't have enough bits left to represent these fully. Instead, we decide to ignore the least significant 3 bits from each, so we can use the 5 more significant bits in the hashCode. We can get something like this:

```
hashCode = 0rrrrrgggggbbbbbxxxxxxxxyyyyyyyy.
```

For example, if we have

```
x = 00000011,

y = 00000010,

r = 11111111,

g = 00111111,

b = 00000000,
```

we would produce the hashCode

```
0|11111|00111|00000|00000011|00000010, or

01111100111000000000001100000010.
```

This can be accomplised using shift left <<, shift right >>, and bitwise and ˆ operations.

If we use this design, we chop off some of the fine color data. Two similar pixels with a small difference in color might produce the same hashCode. But a large difference in color or a difference in position should produce a different hash code, assuming all values are from [0,255]. IMPORTANT: Our Pixel objects' positions are not actually constrained to the range [0,255]. The hashCode should still function for Pixels out of this range, but for this assignment there are no requirements at all for when such `Pixels` have or do not have the same hash code.

You are not required to use this design, but your hash code must pass the unit tests, so you should design it with these goals in mind.

The `Raster` class also has a `hashCode` method that is given. The `deepHashCode` method used here makes use of the `hashCodes` of the contents of the array, namely the results of the `Pixel hashCode` method that you wrote.

## 4 Required Coding Conventions

In the code that you write for CS 351, you must follow some guidelines:

1. Classes and public methods must have appropriate "Java-doc" comments.
2. Fields should be private.
3. Methods should start with lowercase characters and use "camelCase" to handle multiple words.
4. Methods that override those in a super class (notably `toString()` and `clone()`) should be annotated `@Override`.
5. Code should be appropriately indented with tabs and/or spaces.

## 5 Test Programs

We provide two JUnit test suites `TestPixel.java` and `TestRaster` as well as a sample driver program `Demo.java`. **Do not modify them.** Ensure that your project passes all tests and that the driver program displays something reasonable before submission.

## 6 Files

We use "git" to access and turn in programs. Follow the instructions given in the lab to clone your homework repository onto your own or lab computer. Make sure you always push changes back (commit is insufficient) before switching computers and before the deadline.

Your task is to write the following files:

```
src/edu/uwm/cs351/Pixel.java

src/edu/uwm/cs351/Raster.java
```

These files must exist in your homework1.git repository before the deadline (10:00 PM, Monday). Do not forget to write documentation for all classes, constructors and non-standard public methods.