Module IV: More on conditionals

Learning Objectives

Introduction

Let's talk a bit about edge detection. The purpose of edge detection is to find areas of high contrast, and is a key algorithm in many image processing applications. Here's a simple algorithm for it:

1. Loop through all the pixels in the picture
    a. Calculate the average color for the current pixel and the pixel at the same x but y+1
    b. Get the difference between the 2 averages
    c. If the absolute value of the difference is greater than some threshold value, turn the current pixel black
    d. Otherwise, turn the current pixel white

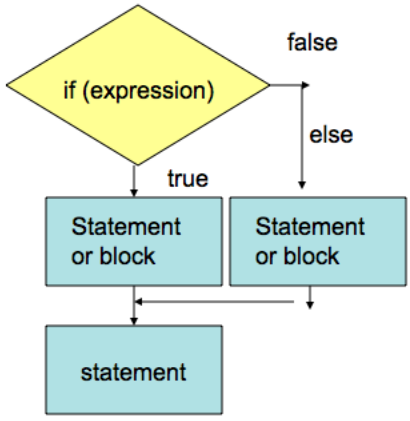
Looks simple enough... But there's a slight problem, recall when we talked about conditionals previously, we learnt that we could execute a statement or block of statements based on a condition being met. But to implement our edge detection algorithm, we also need to run some statement or block of statements if the condition is not met. Here's how we do that.

If and Else

Sometimes you want to do one thing when an expression is true, and a different thing when the expression is false (as with our edge detection algorithm). To do this, we can use the else keyword. The syntax for using the else keyword is as shown:

```
if (expression) {
  // statement or block to execute if expression is true
} else {
  // statement or block to execute if expression is false
}
// next statement
```

The flow chart below also illustrates this concept:



This flow chart should look very similar to the previous one. The difference is that in addition to a special statement or group of statements (blue box at left) to be executed if the conditional is true, there's a special statement or group of statements (blue box at right) to be executed if the conditional is false.

Now that we have a way of executing some code if a condition is true, and some other code if a condition is false, let's take another look at the edge detection algorithm.

Back to edge detection

Let's take a more detailed look at an algorithm for edge detection, and since this is a slightly more involved concept, we'll split it up into different parts. Here's the whole algorithm first:

1. Loop through the rows
    a. Loop through the columns
        a. Get the pixel at the current x and y (top pixel)
        b. get the pixel at the x and y+1 location (bottom pixel)
        c. Get the average of the top pixel color values
        d. Get the average of the bottom pixel color values
        e. If the absolute value of the difference between the averages is over a certain threshold value
            a. Turn the pixel black
            b. Otherwise, turn the pixel white

Now, finding the average pixel color value is easy, you simply sum up all the color values and divide it by 3, like what we did in the grayscale examples. But how do we find the absolute difference between averages? Absolute differences measure the "distance" between 2 numbers, without regard to which number is bigger. For example, the absolute difference between 2 and 4 is 2, as is the absolute difference between 4 and 2. It does not matter which number is bigger, just how far apart they are.

learn by doing

How would we go about finding out the absolute difference between two numbers, **n1** and **n2?**

One possible solution is to say **n2 – n1**.

Under what circumstances would this correctly return the absolute difference between **n1** and **n2**?

Solution:

When n2 is greater than n1, n2 - n1 is greater than 0

The reason n2 - n1 does not work is that it sometimes returns a negative number. We can use conditionals to address this.

```
if (number1 >= number2) {
  difference = number1 - number2;
} else {
  difference = number2 - number1;
}
```

That's a lot of lines of code for a fairly simple job. Luckily, there is an easier solution to the problem. Finding the absolute difference between two numbers is a very common problem (one that many programmers might need to solve). Whenever you think a problem is something that many programmers have encountered, a fast way to find the answer (and the solution most experienced programmers would use if they did not know the solution) is to type a query into Google. Finding the absolute value of two numbers certainly qualifies, so let's try it: search Google for “absolute value java”. If you do, you will quickly discover that there is a function called Math.abs() which returns the absolute value of an integer you pass in to it. The absolute value of an number represents only the magnitude of the number, thus, the Math.abs(3) returns 3, while Math.abs(-4) returns 4. There are many other advanced mathematics utility functions in http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html that you might want to explore.

Now it's time to use Math.abs() to solve our earlier problem. One solution is to type:

```
int diff = n2 - n1;
int mag = Math.abs(diff);
```

However we can solve this problem more simply by remembering that we can pass any expression into **Math.abs()** that evaluates to int. Since **n2-n1** evaluates to **int**, we can say **Math.abs(n2-n1)**.

Now that we know how to find the absolute difference between 2 numbers, let's move on to the next part of our algorithm. In the algorithm, we talk about a threshold value for determining whether to turn a pixel black or white. Let's look at how we can use a conditional. We will compare the magnitude of the difference between the pixels to the threshold.

In our edge detection algorithm, we compare each pixel with the one below it, and use this comparison to decide whether the pixel should be set to black or set to white. Let's look at how we translate this into a conditional. Remember that our pseudocode says:

```
If the absolute value of the difference between the averages is over a certain threshold value
        Turn the pixel black
        Otherwise, turn the pixel white
```

Let's assume that we have two Pixels (the current, or "above" pixel and the next, or "below" pixel),

```
Pixel abovePixel;
Pixel belowPixel;
```

First, let's calculate the average color value for each pixel:

```
int aboveAvg = (abovePixel.getRed() + abovePixel.getBlue() +
        abovePixel.getGreen()) /3;
int belowAvg = (belowPixel.getRed() + belowPixel.getBlue() +
         belowPixel.getGreen()) /3;
```

Finally, we can compute the magnitude of the difference between the average color values of the pixels:

```
int absdiff = Math.abs(topAvg - bottomAvg);
```

We will need to compare this to a threshold in order to determine what color each pixel should be. The threshold value affects how different 2 adjacent pixels need to be, to be considered an "edge". Determining this threshold value usually involves trial and error, and it might be useful to be able to change it on each call to the method. We could set the threshold by hard coding it as a literal into a variable, *or* we could make sure that our function somehow takes a parameter that affects the threshold. Let's pass it in as a parameter. We can call it limit

learn by doing

Now let's figure out what color abovePixel should be. We'll use a conditional for this. It will look something like this. See if you can fill in the values.

```
If ( _____ ) {
   _____[
} else {
   _____;
}
```

Solution

For the algorithm to work correctly, we need to divide pixels into those whose average is larger than the threshold, and those whose average is smaller than the threshold. For this reason, absdiff == threshold would not work, because it would only be true for a small subset of colors whose average is exactly equal to threshold. The answer absdiff > 0 would not work, because it would be true for all pixels. The answer absdiff > threshold works, because pixels that are not very similar

in color will all be greater than threshold, while pixels that are very similar (less different than threshold) will all be less than or equal to threshold.

The correct answer is abovePixel.setColor(Color.BLACK) . The pseudo code says that when the difference between pixels is larger than the threshold, we should turn the top pixel black (Color.BLACK).

The correct answer is abovePixel.setColor(Color.WHITE). The pseudo code says that when the difference between pixels is less than the threshold, we should turn the top pixel white (Color.WHITE).

```
If (absdiff > threshold) {
    abovePixel.setColor(Color.BLACK);
} else {
    abovePixel.setColor(Color.WHITE);
}
```

The next part of the algorithm deals with deciding which pixels to compare. Let's go through a little exercise to see how we should approach this.

learn by doing

If we are comparing each pixel with the one below it, we have to be careful to make sure we don't overrun the bottom of the picture. Assuming that the loop variable representing the current row (call it "y") specifies the first (top) pixel, how would we need to write the loop to ensure that the second (bottom) pixel (taken from the next row) is within the bounds of the picture?

```
for (int y = 0; _____; y++)
```

If we instead compare each pixel to its neighbor to the right, how would we need to change things?

```
for (int y = 0; _____; y++)
```

solution:

```
for (int y = 0; this.getHeight()-1;  y++)
```

Since the loop is written as y the loop will end when y is on the second to last row. This is because the loop is indexed from 0, so the last row is at y == this.getHeight()-1 which is the same as y. When we add in an extra -1 we get to the second to last row.

```
for (int y = 0; this.getHeight();  y++)
```

Since we are no longer comparing across rows, we can now loop to the very last row of t he picture. In fact, we must do this to ensure that our algorithm doesn't miss any pixels. It is the other loop (through the values of x) that will now need to stop one column early.

Now that we've covered the components of the algorithm in detail, let's put our new found knowledge into practice. Given the algorithm and the exercises above, write a method, **edgeDetection** that takes an input **threshold** as a parameter, and runs the edge detection algorithm. The method you write should be able to be tested as follows:

```
Picture p = new Picture("butterfly1.jpg")
p.edgeDetection(10);
p.repaint();
```

learn by doing

Given the algorithm for edge detection, write a method, edgeDetection, that takes an input limit and runs the edge detection algorithm.

Remember to base this on the pseudocode we gave above. You will have nested for loops, with one loop ending a row early, assuming you are comparing the current pixel and the one below it. We've already described how to calculate the average color value for each pixel, and how to take the absolute value of the difference. The previous exercise helped you use an if statement to set the pixel colors. Now you need to put it all together into an implementation of the algorithm. If you name it edgeDetection() you should be able to test it Loop from y = 0 to y = height - 1 Loop from x = 0 to x = image width Get the pixel at the current x and y (top pixel) get the pixel at the x and y+1 location (bottom pixel) Get the average of

the top pixel color values Get the average of the bottom pixel color values If the absolute value of the difference between the averages is over a certain threshold value Turn the pixel black Otherwise, turn the pixel white.

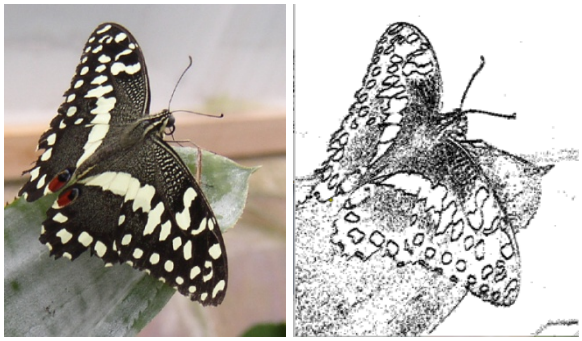
solution

```
public void edgeDetection(int threshold) {
   Pixel abovePixel, belowPixel = null;
   int aboveAvg, belowAvg;

   for (int x = 0; x < this.getWidth(); x++) {
      for (int y = 0; y < this.getHeight() - 1; y++) {
         //get the current pixel
         abovePixel = this.getPixel(x, y);
         // get the pixel below the current pixel
         belowPixel = this.getPixel(x, y + 1);
         // calc the average of the RGB to get the darkness value for both pixels.
         aboveAvg = (abovePixel.getRed() + abovePixel.getBlue() + abovePixel.getGreen()) / 3;
         belowAvg = (belowPixel.getRed() + belowPixel.getBlue() + belowPixel.getGreen()) / 3;

         // if the abs value is greater than 100
         if (Math.abs(aboveAvg - belowAvg) > threshold) {
            abovePixel.setColor(Color.BLACK);
         } else {
            //otherwise set the color of the current to white
            abovePixel.setColor(Color.WHITE);
         }
      }
   }
}
```

If you run it on butterfly.jpg, it will look something like this:



Now, for an additional challenge, let's try a slightly different algorithm for edge detection. This time, instead of comparing with the pixel below, compare with the pixel to the right (x+1). How do you need to change the nested loop? Do you get a different result?

Boolean conditionals

When a simple comparison of two values does not suffice, we can write more complex expressions using *boolean conditionals*. Boolean conditionals can be used to generate complex expressions from several simpler ones.

For easy reference, here is a truth table that shows all the possibilities for two operands and a conditional:

Operand 1	Conditional Name (Java Syntax)	Operand 2	Result
true	AND (&&)	true	**true**
true	AND (&&)	false	**false**
false	AND (&&)	true	**false**
false	AND (&&)	false	**false**
true	OR (||)	true	**true**
true	OR (||)	false	**true**
false	OR (||)	true	**true**
false	OR (||)	false	**false**
true	EXCLUSIVE OR (^)	true	**false**
true	EXCLUSIVE OR (^)	false	**true**
false	EXCLUSIVE OR (^)	true	**true**
false	EXCLUSIVE OR (^)	false	**false**

As seen in the truth table, we can string together expressions and check if multiple items are true by using the AND conditional. In Java, this is represented by the && operator. The syntax for this is as shown:

expression1* && *expression2 ...

Note that **&&** stops as soon as an answer is evident. This is called *short circuiting behavior*. That is, if the first item it evaluates is false, evaluation stops and the entire expression is deemed to be false.

Similarly, we can string together expressions and check if at least one of several things are true by using the OR conditional. In Java, this is represented by the **||** operator. The syntax for this is as shown:

expression1* || *expression2 ...

Like AND, the OR conditional also exhibits short circuiting behavior, and will stop evaluation if the first item is evaluated to be true.

We can check if one and only one of the things are true using the EXCLUSIVE OR (XOR) conditional. In Java, this is represented by the **^** operator. The syntax for this is as shown:

expression1* ^ *expression2 ...

There is one final conditional operator, the NOT operator. The NOT operator behaves quite simply: It changes true to false, and false to true. In Java, the NOT operator is represented by **!**, and its syntax is as shown:

!*expression*

Tip ... Conditional expressions can be combined in the same way that arithmetic operators can be combined -- using parentheses. For example, if you have two variables, int a=2, b=3; you can write something like

((a >= 10) && (a <= 20)) || ((b <= 10) && (b >= 0)))

This conditional returns true if either a is in the range 10-20 or b is in the range 0-10.

Tip... Useful uses for &&

The AND operator is very useful for checking that a value is within a given range. For example, if we want to check if a value lies between 0 and 255 (inclusive), we would usually write it out like so: 0 <= x <= 255.

In Java however, this would be written as

(0 <= x) && (x <= 255)

(x >= 0) && (x <= 255)

Tip... Useful uses for ||

The OR operator is very useful for checking if at least one of several things is true. For example, if want to check that x is greater than 4 or y is less than 6 or z is equal to 7, The code in java would be written as:

(x > 4) || (y < 6) || (z == 7)

Select Statements:

The *switch* statement provides another means to decide which statement to execute next. The *switch* statement evaluates an expression, then attempts to match the result to one of several possible cases. Each *case* contains a value and a list of statements. The flow of control transfers to statement list associated with the first value that matches.

The general structure of a switch statement is as follows:

```
switch ( expression ) {
  case value1:
    statement-list1;
  case value2:
    statement-list2;
  case value3:
    statement-list3;
  case  ...
}
```

Often a *break* statement is used as the last statement in each *case's* statement list. The *break* statement causes control to transfer to the end of the *switch* statement allowing for the code to continue to execute. If a *break* statement is not used, the flow of control will continue into the next case. Sometimes this can be helpful, but usually we only want to execute the statements associated with one case.

A *switch* statement can have an optional *default* case as the last case in the statement. The default case has no associated value and simply uses the reserved word *default.* If the *default* case is present, control will transfer to it if no other case value matches. If there is no *default* case, and no other value matches, control falls through to the statement after the *switch.*

The expression of a *switch* statement must result in an integral data type, like an integer or character; it cannot be a floating point value.

Note that the implicit *boolean* condition in a switch statement is equality - it tries to match the expression with a value. You cannot perform relational checks with a *switch* statement.

Simple example:

```
switch(day) {
  case 1:
    System.out.println("Monday");
    break;
  case 2:
```

```
      System.out.println("Tuesday");
      break;
   case 3:
      System.out.println("Wednesday");
      break;
   case 4:
       System.out.println("Thursday");
      break;
   case 5:
      System.out.println("Friday");
      break;
   default:
      System.out.println("week-end!");
}
```

The switch and the if .. else can be used in many situations NOT ALL interchangeably. IN the above example we could write it as an if.. else as follows:

```
If (day == 1) {
   System.out.println("Monday");
} else if (day == 2) {
   System.out.println("Tuesday");
} else if (day == 3) {
   System.out.println("Wednesday");
} else if (day == 4) {
   System.out.println("Thursday");
} else if (day ==5) {
   System.out.println("Friday");
} else {
   System.out.println("week-end!");
}
```

Summary

In this module, we learned:

- How to use if and else to execute different statements depending on the outcome of an expression
- How to write an edge detection algorithm for images.
- About complex conditionals and how to use them
    - AND (&&) tests for more than one thing being true
    - OR (||) tests for at least one thing being true
    - XOR (^) tests for exactly one thing being true
    - NOT (!) operator changes true to false and false to true