# Conditions and Selections

MODULAR IV

## Conditionals

- If you've ever entered a contest, there are usually terms and conditions.
- This usually means that you could only win a prize if you met those conditions. These conditions are expressions that evaluate to either true or false.
- Example Conditionals
  - These are a few conditions that might be found on contest entries or other places:
    - Must be 18 or older to enter
    - Must be a legal resident of the Canada
    - Must own a dog

## Objectives

- How to conditionally execute a statement or block of statements
- How to remove red-eye from a picture
- How to use conditionals with 2 possibilities.
- How to do simple edge detection.
- How to use 'and', 'or', 'exclusive or' and 'not' in a conditional.

## Conditionals

- All of these conditions can be expressed in the form of a yes/no question, like "Is [age] 18 or older?", or "Is [person] a legal resident of the Canada?".
- Once it is in this question form, it's easy to translate it to Java.
- The question "Is [age] 18 or older?" can be translated to the Java expression age >= 18, where age is a variable representing how old a person is, and >= is asking whether that number is greater than 18.

## Introduction

- In the previous modules, we learnt how to manipulate images. However, we usually manipulated the whole image at once (e.g. removing the blue from all pixels or negating a whole image). But what happens if you only want to modify a specific part of an image?
- A good example of this is red-eye removal. Red-eye happens frequently in pictures, when a camera flash is reflected off a subject's eyes, and makes photos look... creepy.
- We could always write an algorithm that removes all the red from a whole image, since that would definitely remove the red-eye. But if we remove the red, we could also change the color of the persons clothes, which we do not want to do.

## Notation: == versus =

- In Java, == and = have very different meanings.
- = is called the assignment operator and is used to assign values to variables.
- == on the other hand, is a comparison operator.
- == compares 2 expressions, and returns true if both expressions equate to the same value.

## if

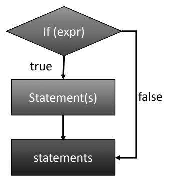
- Sometimes, we want a statement, or block of statements to execute only if some expression is true. To do this, we can use the if statement in Java.
- The syntax for the if statement is like this:

```
if (condition) {
   // statement or block to execute
}
// next statement
```

## Red-eye removal algorithm

- At a high level, the algorithm should look for pixels that are close to red in the area around the eyes. More specifically:
  1. We only want to change the pixels that are "close to" red
  2. We can find the distance between the current color and our definition of red
     a) Change the color of the current pixel only if the current color is within some distance to the desired color.

## if



- If the conditional *expr* in the diamond evaluates to true, then the *statement(s)* inside the if statement executes.
- Once that is completed, the next statement executes.
- If the conditional expression is false, then the statement(s) is skipped, and the next statement executes.

## Red-eye removal algorithm

- Is that algorithm detailed enough to write a method to remove red-eye?

  Not really!
- Assuming we have the coordinates for the area around the eyes (x, y, w, h):
- Loop from x to w
  - Loop from y to h
    - Get the pixel at the current x and y
    - Get the distance between the pixel color and red
    - If the distance is less than some value (say 167), change the color to some passed in new color

## recall

```
int x = 2;
if (x > 1)
   System.out.println("X is > 1");
System.out.println("X is " + x);
x = 0;
if (x > 1)
   System.out.println("X is > 1");
System.out.println("X is " + x);
```

- We execute only *one* statement after the *if* statement.
- If we want to execute more than one, we must create a block of statements using the { }

---

- Now, that algorithm is more developed and can be translated fairly easily into code.
- But what is this color distance thing?
- Colors don't have distances, do they?
- Well, they do, kind of.
- If we take each color as a point in a 3-dimensional space, we can calculate the distance between 2 colors fairly easily.

- The distance between 2 points is computed using the pythagorean theorem, like so:

  distance = square_root($(x1-x2)^2 + (y1-y2)^2$)

- So, to compute the distance between 2 colors, we modify the equation slightly and come up with

  distance = square_root(
  $(red1-red2)^2 + (green1-green2)^2 + (blue1-blue2)^2$ )

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

- If all that math has got you down, don't fret.
- Some programmer somewhere decided that lots of people might need to calculate this, and wrote a method for it in the Pixel class. ;)
- So you can now use the following code:

```
double dist = pixelObj.colorDistance(color1);
```

- Looks simple enough...
- 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.

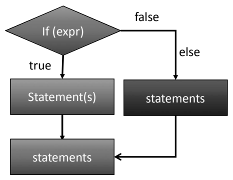
```
public void removeRedEye( int startX, int startY,
                          int endX, int endY,
                          Color newColor)  {

Pixel pixelObj = null;
   // loop through the pixels defined by startX, startY, endX and endY
   for (int x = startX; x < endX; x++)    {
      for (int y = startY; y < endY; y++)  {
         //get the current pixel
         pixelObj = getPixel(x, y);
         //if the color is near red, then change it
         if (pixelObj.colorDistance(Color.red) < 167)   {
            pixelObj.setColor(newColor);
         }
      } // end for
   }  // end for
} // end method
```

## 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
```



## learn by doing

- How would we go about finding out the absolute difference between two numbers, **n1** and **n2?**
- Under what circumstances would this correctly return the absolute difference between **n1** and **n2**?

## Edge detection Algorithm

- Let's take a more detailed look at an algorithm for edge detection, this is a slightly more involved concept:.

```
Loop through the rows
    Loop through the columns
        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

- One possible solution is to say n2 – n1.
- 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;
}
```

- Now, finding the average pixel color value is easy, you simply sum up all the color values and divide it by 3.
- 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.

- We can simplify this solution by using the functionality of the Math library.
- Math.abs() which returns the absolute value of an integer.
- Now, this being a common problem, Math.abs() has a variety of functionality such as passing in two values to get the absolute value between ( ex: Math.abs(4,2) )

## Examples

- Math.abs(3) returns 3
- Math.abs(-4) returns 4
- Math.abs(4, 2) returns 2

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

## Edge detection

- 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.

## Edge detection

- 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
```

## Edge detection

- 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.
- absdiff == threshold would not work, because it would only be true for a small subset of colors whose average is exactly equal to threshold.
- absdiff > 0 would not work, because it would be true for all pixels.
- 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.

## Edge detection

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

## Edge detection

- Therefore;

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

## Edge detection

- 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.
- Since we are not comparing across rows, we can loop to the very last row of the picture.
- In fact, we must do this to ensure that our algorithm doesn't miss any pixels.

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

## Edge detection

- The other loop (through the values of x) that will now need to stop one column early.
- Now, taking everything we have seen, can we complete the method *edgeDetection* that takes in an input *threshold* as a parameter?

## 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 ave 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 (Math.abs(aboveAvg - belowAvg) > threshold) {
              abovePixel.setColor(Color.BLACK);
           } else {
              abovePixel.setColor(Color.WHITE);
           }
        }
     }
  }
```

## Boolean conditionals

- && (AND) :: expression1 && expression2 ...
  - && stops as soon as an answer is evident. This is called short circuiting behavior. That is, if the first item evaluates to false, evaluation stops and the entire expression is deemed to be false.
- || (OR) :: 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.
- ^ (EXCLUSIVE OR or XOR) :: expression1 ^ expression2 ...
  - one and only one of the things are true
- ! (NOT) :: !expression
  - It changes true to false, and false to true.

## Boolean conditionals

| **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** |

## Tips

- 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.

## Multiple *if* statements

- If we are doing different things based on a set of ranges, we can use multiple if statements.
- For example, if we want to do one action when (0 <= x <= 5), another set of actions when (5 < x <= 10) and yet another set of actions when (10 < x), this is best done with multiple if statements.
- The table below shows pseudocode and how it compares with the actual syntax for using multiple if statements.

## Tips

- 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)

## Multiple *if* statements

| Pseudocode | Actual Code |
|---|---|
| if x lies between 0 and 5 inclusive | if (0 <= x && x <= 5) { |
| *do action 1* | action_1(); |
| | } |
| | |
| if x lies between 5 (non-inclusive) and 10 inclusive | if (5 < x && x <= 10) { |
| *do action 2* | action_2(); |
| | } |
| | |
| if x is greater than 10 | if (10 < x) { |
| *do action 3* | action_3(); |
| | } |

## Tips

- 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)

## Multiple *if* statements

- Notation... Inclusive and exclusive ranges
  - If a range of values is defined as "between 5 and 10, inclusive", what this means is that any value between 5 and 10, including 5 and 10 is valid.

    In Java this would be written as (5 <= x && x <= 10)
  - if a range of values is defined as "between 5 and 10, exclusive", this means that any value between 5 and 10, but not including 5 and 10 is valid (i.e. 6, 7, 8, 9 are valid).

    In Java this would be written as (5 < x && x < 10)
  - add an equal (=) sign if it is inclusive, and not add it if it is exclusive.

## Multiple *if* statements

- the code above works great, but if our ranges are *disjoint*, that is, they do not overlap, then we spend extra time going through all the if statements, even if we already found our range and executed the correct statements.
- The ranges that we talked about above are disjoint, and so, if x was 4, then we wouldn't need to check if x was greater than 5 (the second range) or greater than 10 (the third range).
- Since the ranges are disjoint, if we executed the statements in the first range, we would not need to care about the rest of the ranges, as we know that there is no way one value can fall between two or more disjoint ranges.

## Multiple *if* statements

- What we want is to be able to check each conditional, and if it evaluates to true, execute the statements in that block and not evaluate any more conditionals relating to that range..
- The syntax for an else if statement is as follows:

```
if (expression1)
   statement1
else if (expression2)
   statement2
else if (expression3)
   statement3
   …
else
   code that does not fit any other range
```

## Multiple *if* statements

| Pseudocode | Actual Code |
|---|---|
| if x lies between 0 and 5 inclusive | if (0 <= x && x <= 5) { |
| *do action 1* | action_1(); |
| if x lies between 5 (non-inclusive) and 10 inclusive | } else if (x <= 10) { |
| *do action 2* | action_2(); |
| if x is greater than 10 | } else { |
| *do action 3* | action_3(); |
| | } |

## Multiple *if* statements

- If you look at the actual code, you'll notice an interesting side effect of using else if.
- Where before, we needed to check that x was greater than 5 AND less or equal to 10 (if (5 < x && x <= 10)), now we can leave out the part that checks if x is greater than 5, as if x was NOT greater than 5, then the code in the first if block would have executed.
- Similarly, we can leave out the conditional completely for the case where x is greater than 10, and replace that with an else statement, since if x was NOT greater than 10, one of the previous two statement blocks would have executed.

## Did I get this?

Given the following code fragment, what is printed on screen?

```
int x = 6;

if (x < 5)
   System.out.println("A");
else if (x < 10)
   System.out.println("B");
else
   System.out.println("C");
```

```
int x = 6;

if (x < 10)
   System.out.println("A");
if (x < 20)
   System.out.println("B");
else
   System.out.println("C");
```

## solution

- The first *if* block checks if x is less than 5, of which x is not.
- Then the second block executes, if x is less than 10, which it is.
- Thus, B is printed to the screen and the selection statement is halted.

- The first *if* block checks if x is less than 10, which it is. Thus 'A' is printed to the screen.
- Subsequently, since there isn't an *else if* block, the second *if* block is executed when x is less than 20, thus printing 'B' to the screen.
- The overall output is 'AB' and the selection statement is halts.

## switch 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.

## *switch* Statements

- 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.*

## *switch* Statements

- 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  ...
}
```

## *switch* Statements

- 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.

## *switch* Statements

- 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.

## *switch* Statements

```
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!”);
}
```

## *switch* vs *if .. else* but not always

```
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 the if statement to conditionally execute a statement, or block of statements
  - How to apply conditional statements to do useful things, like remove red-eye from pictures
  - The syntax for an *if*, *if .. else* and *switch* statement:
  - 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