

UNIT 11. ADVANCED CONDITIONALS

Summary

Now that you have basic conditionals under your belt, we can look at more advanced techniques. In this section, you will...

- ♦ Learn about how to compare data, e.g., if they are equal
- ♦ Learn exception cases including comparing strings and floating point numbers
- ♦ See new syntax for chaining multiple if conditions together
- ♦ See how code blocks are not always needed (but should be used)



Learning Objectives

After finishing this unit, you will be able to ...

- ♦ Compare numerical values, and see if they equal, or if one is larger than the other
- ♦ Compare booleans to see if they are equal
- ♦ Combine boolean and relational operations to complete complex tasks
- ♦ Compare String data, to see if two strings are equal
- ♦ Construct if-else-if chains

How to Proceed

- ♦ Read the unit content.
- ♦ Have a Processing window open while you read, to follow along with the examples.
- ♦ Do the sets of exercises in the **Check your Understanding** sections.
- ♦ Re-check the **Learning Objectives** once done.

11.1 Introduction

Boolean logic is at the heart of computer science. Computer have to look at data, and make decisions based on the circumstances. As such, boolean logic scales up very quickly beyond the simple examples we have seen, and can get very complicated. Think about it – all the software you use on a computer, phone, or tablet, is built using `if` statements. In this unit we look at the next level of difficulty with conditionals.

11.2 Relational Operators

There is only so much you can do with the built in booleans, and with our AND, OR, and NOT operation, so we need to generate new booleans that relate more to our own needs and problems. We do this by using relational operators: operations in Processing that result in a boolean value by comparing two values.

The simplest relational operator is to compare two values to see if they are equal. You can do this by using two equal signs together, for example, `number1 == number2`, gives a boolean result. **Be careful! Use two equal signs stuck together, not one – one equal sign is for setting a variable to a value.**

For example, you can test if two numbers are the same:

```
int a = 5;
int b = 6;
println(a == b);
```

This will print `true` if `a` and `b` have the same value, and `false` if they do not. As we can see, here we will see `false` as our output, since `a` and `b` are not equal.

Just like our logical operators in the previous unit, this takes to values, compares them, and gives a boolean result.

For example, we can detect if the mouse is on the diagonal of the canvas, by seeing if its `x` and `y` coordinates are the same.

```
// brackets are not required but help readability
boolean onDiagonal = (mouseX==mouseY); // true if X equals Y
```

Let's make a drawing program to automatically draw a grey circles when the mouse hits the diagonal.

```
if (onDiagonal)
{
  stroke(127)
}
```

```
    ellipse(mouseX,mouseY,10,10);
}
```

So, in this program, each time it draws, it compares the mouse X and mouse Y. If they are the same, then `onDiagonal` is true, and false otherwise. Then, if they are the same, we draw a circle.

There are other relational operators, too:

<code>!=</code>	<code>a != b</code>	true if not equal
<code><</code>	<code>a < b</code>	true if a is less than b
<code><=</code>	<code>a <= b</code>	true if a is less than or equal to b
<code>></code>	<code>a > b</code>	true if a is greater than b
<code>>=</code>	<code>a >= b</code>	true if a is greater than or equal to b

Be careful! The order of the `<=` and `>=` is important, for example, you cannot do `=>` and `=<`.

In all of these cases, you can compare values and you will get a boolean result based on those values. In addition to storing the result into a variable, you can also use the boolean operators directly within an `if` statement. For example, above we had the following code.

```
boolean onDiagonal = (mouseX==mouseY); // true if X equals Y
if (onDiagonal)
{
    // ...
}
```

This can be simplified to:

```
if (mouseX == mouseY)
{
    // ...
}
```

Which not only saves a variable and is shorter, but is also quite intuitive to read.



Advanced: You may see people using characters with the less than or greater than comparisons. The reason this works is because it looks at the numerical representation underneath (review in Unit 8.7). Some people use this to see if a character is a letter or number, for example, seeing if a character is within the range of A and Z, leveraging the structure of the ASCII table. This is generally considered

bad practice, since it is not Unicode-safe, and introduces a whole slew of issues on international platforms (e.g., the web, or non-English machines). Don't do it.

When using the `==` operation, one thing you will see people doing is to compare boolean variables to `true` or `false`, for example:

```
if (onDiagonal == true)
```

Or


```
if (onDiagonal == false)
```

Logically, this works – you are comparing a boolean to `true` or `false`. This is not an error per se, but is generally frowned upon, since it is redundant. If you think about it, the *definition of an if statement* is to test if a boolean is true. By adding the `== true` in there, it's akin to saying “if a is true is true”, which sounds silly. Experienced programmers notice this right away, and you shouldn't do it.

It can also introduce errors in annoying ways. Consider the following buggy code. Can you see the bug?

```
boolean b = true;
if (b = true)
{
    println("true!");
}
```

Here, the programmer used one `=` and not two `==`. So, instead of comparing `b` to `true` and giving a boolean result, the code *assigns* `b` to `true` (one `=` is used), and then uses the result, `b`, in the test. This if statement a) sets `b` to `true`, and `b` always evaluates to `true`, and runs. Nasty bug.

 **You can avoid these bugs by** not doing the `== true` and `== false` style. Instead, test if a boolean is true with `if (b)`, and test if it is false by using negation, `if (!b)`.

There are some caveats to the relational operators, unfortunately. These relational operators usually work as you would expect. For example, they are very robust with integers, can be used with characters and booleans, but there are important exceptions that we need to cover. These are so important, that they get their own subsections.

11.3 Comparing Strings

If you remember, Strings are objects, and so have annoying quirks. For relational

operators, the greater and less than variants clearly do not make sense for strings. However, how do you compare if two strings store the same text?

So far, we compare two pieces of data by using two equal signs: `==`. Unfortunately, we cannot use this technique with strings. While it will seem to work (that is, it runs and does not crash), it does not do what you think: using `==` on Objects tells you if they are the same object in memory, an advanced concept we do not yet cover. Since we didn't learn objects yet, this gets very confusing and you shouldn't use `==` on strings. Expect bugs if you do.

We need to learn a new technique that lets us compare strings. If you remember, since Strings are objects, we can do extra things to the variable using methods. That is, we can run functions on the variable itself. We already learned some methods, e.g., to check how long a string is.

In this case, we have a method to check if a string has the same data as another string. This is the `.equals` method, which gives you a boolean. It is used as follows:

```
stringVariable.equals(otherString)
```

For example:

```
String s = "Jim";  
if (s.equals("Jim")) // compare s to "Jim"  
{  
    println("yay!!!");  
}
```

In the above example, the `s` variable data is compared to the literal string `"Jim"`. They are equal, so it gives a `true`, and the `if` statement block executes.

You can also compare two string variables this way:

```
String s1 = "cat";  
String s2 = "dog";  
println(s1.equals(s2));
```

Be careful, though, since string comparison is *case sensitive*. The following example gives you `false`:

```
String s1 = "jim";  
String s2 = "Jim";  
println(s1.equals(s2));
```

We won't use string comparisons much in this course, but you should be aware of it. It is definitely testable material.

11.4 Comparing Floating Point Numbers

As we learned in Unit 7, computers cannot store perfect real numbers. Instead, they store approximations only. For the most part, this is okay, and we do not worry about it. However, when comparing floating point numbers, we need to be aware that we may have rounding errors based on how computers store the numbers.

Luckily, the way the numbers are stored is consistent, so the greater than and less than operators are robust for floating point numbers.

Unfortunately, due to precision error and rounding error, we cannot trust the `==` and `!=` operators on floats. Pretty much **you should never use `==` and `!=` on floats.**

Consider the following example:

```
if (0.7 == 0.1+0.6)
{
    println("they are the same!");
}
```

What is the output? Nothing, unfortunately. The `if` statement evaluates to false due to floating point rounding error. You can take a closer look at this by using a few `println` statements to put the results to console.

So how then do you tell if two floating point numbers are the same? The answer lies in realizing that floating points are approximations at best. In the real world, if you are cutting a piece of wood to be 1m long, how precise do you need it? Do you care if it is 1.1m long? probably. Do you care if it is 1.001m long? (1 m and 1mm). Probably not. In this example, you have a tolerance for how close you are before you consider it to be 1m long, probably +/- 2 or 3 mm.

We do the same with floating point numbers. You can take the difference of two, and see how close they are. You can then look at this closeness and make a judgment call based on your application.

11.5 Example: Tiered coloring

Let's modify our drawing program from the last unit (with appropriate active processing template code):

```
if (mousePressed) ;
{
    line (pmouseX, pmouseY, mouseX, mouseY) ;
}
```

```
}
```

so that the drawing color changes depending on where you are drawing. Let's make five bands – if you draw less than x of 100, use color 50. Between 100 and 199, color 100. 200-299, color 150, 300-399 color 200, and 400-500 color 255.

For simplicity, I am starting from the base case where you draw only if the mouse is pressed, and clear the screen on a key press.

One way to approach this is to start with the first case:

```
if(mouseX < 100)
{
    stroke(50);
}
```

Make sure to place this before the line code to ensure that the stroke is properly set before drawing (and not after!);

For the next condition, we could simply add this condition following the first one:

```
if (mouseX < 200)
{
    stroke(100);
}
```

But there is a problem here! (try running it). This condition overrides the first case. For example, the coordinate 50 is true for both the above tests, and the color is always 100, and not 50.

One solution is to only test for < 200 if we know it's not <100. The result will be a number from 100-199 (not < 100, but < 200). That is, if it's NOT < 100, then test. We can do this using our `else` block and nesting an `if` inside there.

```
if (mouseX < 100)
{
    stroke(50);
}
else // not < 100, so >= 100
{
    if (mouseX < 200) // >= 100 but < 200
    {
        stroke(100);
    }
}
```

```
}
```

Let's now look at the next case – 200-299. We follow the same logic as above. If our test for <200 fails, then it must be >=200, so we add an else block to THAT if statement. This gets confusing fast.

```
if (mouseX < 100)
{
  stroke(50);
}
else // not < 100, so >= 100
{
  if (mouseX < 200) // >= 100 and < 200
  {
    stroke(100);
  }
  else // not < 200, so >= 200
  {
    if (mouseX < 300) // >= 200 and < 300
    {
      stroke(150);
    }
  }
}
```

Counting those brackets sucks



Whew! And all these ifs are nested inside the draw block. We have two more to go! This is getting very messy very fast (and those closing brackets and opening brackets are really confusing. There must be a better way!

In fact, there is. Because this if-else-if pattern is so common, we have special syntax for it.

11.6 If-else-if chains

When you have a series of ifs,elses, and then nested ifs, you can use the following cleaner syntax that doesn't nest ridiculously.

```
if (condition)
{
}
else if (condition) // only if above condition was false
{
}
```



```
else if (condition) // only check if all above are false
{
} // you can have as many of as you like
else // only run if ALL the above conditions
are false
{
}
```

Much nicer!!!



Keep in mind that each `else if` is only checked if all the above ones were false. At any point, if any `if` test is true, then we run that block and no other conditions are checked – because we don't have an `else`!

The final `else`, called a terminal `else`, is only run if all the conditions above are false. **The terminal `else` is optional, you do not need it.**

Let's rewrite the above partial solution using the new syntax:

```
if (mouseX < 100)
{
    stroke(50);
}
else if (mouseX < 200) // >= 100 and < 200
{
    stroke(100);
}
else if (mouseX < 300) // >= 200 and < 300
{
    stroke(150);
}
```

This is much cleaner, and easier to read. Now we can easily add the next conditions. Finally, we can end with the final `else` for the last column, from 400-500:

```
if (mouseX < 100)
{
    stroke(50);
}
else if (mouseX < 200) // >= 100 and < 200
{
    stroke(100);
}
```

```

else if (mouseX < 300) // >= 200 and < 300
{
  stroke(150);
}
else if (mouseX < 400) // >= 300 and < 400
{
  stroke(200);
}
else // must be >= 400
{
  stroke(255);
}

```

Great! It works! Here is my final code for this program:

```

void setup()
{
  size(500, 500);
  background(0);
}

void draw()
{
  stroke(255);

  // only draw when pressing the mouse button
  if (mousePressed)
  {
    if (mouseX < 100)
    {
      stroke(50);
    }
    else if (mouseX < 200) // >= 100 and < 200
    {
      stroke(100);
    }
    else if (mouseX < 300) // >= 200 and < 300
    {
      stroke(150);
    }
  }
}

```



```

    }
    else if (mouseX < 400) // >= 300 and < 400
    {
        stroke(200);
    }
    else // must be >= 400
    {
        stroke(255);
    }

    line(pmouseX, pmouseY, mouseX, mouseY);
}

// erase the screen if any key is pressed
if (keyPressed)
{
    background(0);
}
}

```

Here is a variant of the above if-else-if chain that has a bad bug: can you see it?

```

if (mouseX < 500)
{
    stroke(255);
}
else if (mouseX < 400)
{
    stroke(200);
}
else if (mouseX < 300)
{
    stroke(150);
}
else if (mouseX < 200)
{
    stroke(100);
}
else

```

```
{  
    stroke(50);  
}
```

The first test will always be true, since the mouse is always less than 500 on the X axis. Due to a concept called *short circuiting*, once a value is true, no other condition is checked. There is no searching for a best fit.



Remember: the computer works its way from top to bottom. If a condition is false, the next one is checked, until the end. If all conditions are false, the terminal `else` block runs, if it exists. From top to bottom, if any condition is true, then that block runs, and the rest of the chain is skipped.

11.7 Example: Click on a Button

Let's do a basic button. First, let's simply draw a square to act as our button. Set whatever background and drawing colors you like.

```
rect(100,100,50,50);
```

Let's make the rectangle change color if the mouse is over it. That is, we need some boolean tests to determine if the mouse is intersecting the rectangle, and based on that test, change the draw color. At this point it is a good idea to re-factor the code using good variable names that specify the button location and size. Declare them at the top of your program and initialize them, so that you can type:

```
rect(buttonX, buttonY, buttonSizeX, buttonSizeY);
```

These variables help us test the mouse position against the button position.

To solve this problem, let's tackle one piece at a time. Let's focus on X first. To hit the button, the `mouseX` has to be bigger than or equal to the button's left side. Let's setup the basic initial test, and the color change:

```
if (mouseX >= buttonX) // to the right of button left edge  
{  
    fill(127);  
}  
else  
{  
    fill(255);  
}
```

Try running the code. The button changes if we are to the right of the button left edge. Of course, this also includes beyond the button right edge, which isn't right. So, let's add another condition. In regards to the X, we are on top of the button if we are \geq the left edge, AND, we are \leq the right edge. How do we calculate the right edge? `buttonX+buttonSizeX`:

```
if (mouseX >= buttonX && mouseX <= buttonX+buttonSizeX)
```

So this works with respect to the X, now we need to also add the Y conditions. You can simply just keep on adding your `&&`s to have more AND conditions.

```
if (mouseX >= buttonX && mouseX <= buttonX+buttonSizeX &&
    mouseY >= buttonY && mouseY <= buttonY+buttonSizeY)
```

Now, the button changes color if we mouse over it! This is the calculation your computer performs every time a button changes color or highlights when the mouse goes over it. Try to add one more condition – only change color IF the mouse button is pressed in addition to the other conditions. Again, you can just chain these.

```
if (mouseX >= buttonX && mouseX <= buttonX+buttonSizeX &&
    mouseY >= buttonY && mouseY <= buttonY+buttonSizeY &&
    mousePressed) // all four edges
```

We now have a button that we can press!

This logic gets confusing fast, particularly when you start mixing AND and OR and different relational operators. One thing you can do is to add brackets to help clarify it visually, although it is not necessary.

```
if ((mouseX >= buttonX) && (mouseX <= buttonX+buttonSizeX) &&
    (mouseY >= buttonY) && (mouseY <= buttonY+buttonSizeY) &&
    mousePressed) // all four edges
```

This small change can really make it easier to see.

Here is my whole program:

```
int buttonX = 100;
int buttonY = 100;
int buttonSizeX = 50;
int buttonSizeY = 50;
```

```

void setup()
{
  size(500, 500);
  background(0);
}
void draw()
{
  if((mouseX >= buttonX) && (mouseX <= buttonX+buttonSizeX) &&
      (mouseY >= buttonY) && (mouseY <= buttonY+buttonSizeY) &&
      mousePressed) // all four edges
  {
    fill(50);
  }
  else
  {
    fill(255);
  }

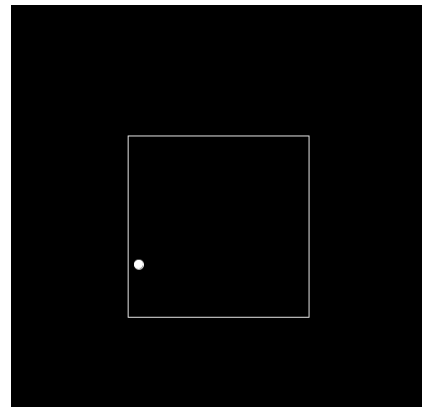
  rect(100, 100, 50, 50);
}

```

11.8 Example: Rebounding Ball

Let's do a simple model of a ball bouncing around inside a box. Let's draw a box in the middle of the screen, and have a ball moving inside it. When the ball hits an edge, make it bounce.

First, let's make a ball start at the center of the screen, and give it an initial speed. We can set its speed in the x and y directions independently. Changing these numbers changes the angle it moves at. Here are my initial variables:



```

int ballX = 250; // start at the center of the screen
int ballY = 250;
int ballSpeedX = 2;
int ballSpeedY = 1;
int ballSize = 10;

```

In the draw loop, we need to draw the ball, and make the ball move by the speed amount each frame. Keep in mind that a negative speed just means that the ball

moves in a different direction.

```
// draw the ball
ellipse(ballX, ballY, ballSize, ballSize);

// move the ball
ballX += ballSpeedX;
ballY += ballSpeedY;
```

Now, we need to draw the box that the ball bounces around in. I will setup some variables to define the box, and then draw the box in the draw loop:

```
int boxLeft = 150;
int boxTop = 150;
int boxSize = 200;
int boxRight = boxLeft + boxSize;
int boxBottom = boxTop + boxSize;
...
// draw the box
stroke(255);
line(boxLeft, boxTop, boxRight, boxTop);
line(boxRight, boxTop, boxRight, boxBottom);
line(boxRight, boxBottom, boxLeft, boxBottom);
line(boxLeft, boxBottom, boxLeft, boxTop);
```

Now you should see a ball that starts in the center of the screen and moves inside a box. Of course, it just goes through the box!! We need to make the ball bounce.

If you remember your physics, there are real formulas to calculate how a ball bounces, but that is much too complex for us here. All that we need to do to simulate a reasonable bounce is: if the ball hits the side walls, change the direction of the X movement. If it hits the top or bottom walls, change the direction of the Y movement. We can change these directions simply by multiplying by -1.

```
// check the bounds
// right and left
if (ballX < boxLeft || ballX > boxRight)
{
    ballSpeedX *= -1;
}
```

```
// top and bottom
if (ballY < boxTop || ballY > boxBottom)
{
    ballSpeedY *= -1;
}
```

That's it! We now have a bouncing ball inside a box. I encourage you to play with this example, such as doing things on a key or mouse press, moving the ball around, or changing the speed.

11.9 Logic Practice and Short Circuiting

Logic can nest and get complex very quickly! Here are some examples. Look at the following expression:

```
!( ! (a<b) || c)
```

Fast: what are the data types of a, b, c? Can you tell?

a and b must be numerical, since you are comparing if one is less than another. c must be a boolean since it is being used in an OR operation.

What is the result if a=3, b=1, and c=true?

Let's work through it

```
!( ! (a<b) || c)
!( ! (3<1) || T)
!( ! (F) || T)
!( T || T)
!( T)
F
```

In general you want to avoid such annoying and complex statements in your programs. If you find yourself making these kinds of messy expressions, you can fix it with better variable names, and doing some of the calculation first into a variable to help the reader understand what is happening.

In the above example, do you see a shortcut to get the final F answer without doing all the calculations? In fact once you know that c==true, you do not need to do the rest of the test at all. If you have one true value in an OR (||) statement, then the OR is true, so we know that !(a<b) || c is true if c is true, regardless of the values of a and b. This is general idea is called short circuiting and you should be aware of it. In short,

Short Circuit! NEED INPUT!



the computer can detect a short circuit, and some commands may never be run. For example

```
if (myVariable && someCommand())
```

In this case, if `myVariable` is false, then the whole and statement is necessarily false, and the computer may not run `someCommand`. That command may have some important side effects (like drawing something) that you expect to happen (although this is a terrible idea and you should never do that!). At this point, you will not likely run into this problem, but it is important to be aware of.



Advanced: You can force your computer to not short circuit a logical operation. There are special commands for this. In Processing, instead of `&&` and `||` for and and or, you can use the single versions, `&` and `|`. These behave the same on boolean values as the regular and and or, without short circuiting. Most people are not aware of these, and they are not commonly used.

11.10 Alternate If Statement Syntax

We learned that the `if` statement requires a code block to run, in the case that your test is true:

```
if (test)
{ // code block
}
else
{ // else block
}
```

Actually, these blocks are not always necessary. In the special case where you only have one command to run if a statement is true or false, then you can avoid the block and just put your single command:

```
if (test)
  oneCommand();
else
  otherCommand();
```

Why would you lie to us??



Although this may seem like a great way to save space, it generally is a bad idea because it can easily lead to problems. Consider the following example:

```
if (shouldMugJim)
    putOnBlackMask();
    practiceToughVoice();
    mugJim();
continueProgram();
```

Even though the indentation is setup to look like all three commands will be run if the if test is true, because code blocks aren't used, only the first command (`putOnBlackMask()`) is run. The other two run anyway, regardless of the outcome of the test. That is, if `shouldMugJim` is false, you still practice your tough voice and mug Jim, without putting a mask on. To make things worse, when you do if-else-if chains, you can mix and match, as follows:

```
if (booleanStatement)
{
    commandA1;
    commandA2;
}
else if (booleanStatement)
    command B;
else
    command C;
```

This creates even more opportunities for bugs. For example, can you spot the bug in the following code?

```
if (temperature < -50 || temperature > 50) {
// invalid temperature
    severity = -1;
} else { // temperature is valid
    if (temperature < -20)
        severity = 4;
    } else if (temperature < -10) {
        severity = 3;
    } else if (temperature < 0) {
        severity = 2;
    } else
        severity = 1;
}
```

In this case, there is a missing open bracket after the test for `temperature < -20`, so that `if` test does not use a code block. When the closing bracket is encountered after `severity = 4`, it is closing the `else` block for `temperature` is valid. Then, we are trying to follow a final terminal `else` with a new `else if`, which doesn't make any sense. This won't compile. Think through this one.

ALWAYS using explicit code blocks avoids this oversight risk:

```
if (shouldMugJim)
{
    putOnBlackMask();
    practiceToughVoice();
    mugJim();
}
continueProgram();
```



Check your Understanding

11.11 Check your Understanding: Exercises



Exercise 1. Make a basic drawing program as in this chapter, with a black background and white drawing ink, which draws while the user is pressing a mouse button.

- Update the program so that it only draws if the mouse is in the top left quadrant of the screen.
- Change the program so that it only draws if the mouse is in the top right quadrant of the screen.
- Change the program so that it only draws if the mouse is NOT in the bottom right quadrant (so it draws in the other three).



Exercise 2. Update the rebounding ball example (11.8) so that the ball bounces when the *edge* of the ball hits. Currently, it bounces when the center hits a wall.



Exercise 3. Make a simple drawing program that has four buttons on the top of the screen.

- If the first button is pressed, clear the canvas
- Second button: set the color to white
- Third button: set the color to grey
- Fourth button: set the color to black (erase)

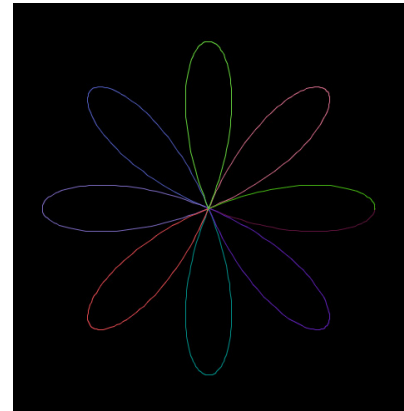


Exercise 4. Update the simple button clicking example in 11.7, to turn the button into a toggle button. If you click it once, it stays down. If you click again, it comes up. You will need global variables to remember the state, and, a technique (as in the previous unit) to ensure that only one change happens when the mouse button is pressed (and that it doesn't rapidly toggle).



Exercise 5. Update Exercise 5 in section 7.8, the rose program. (hopefully, you saved your exercise from earlier!). Now, make it so that the rose has a random color on each petal.

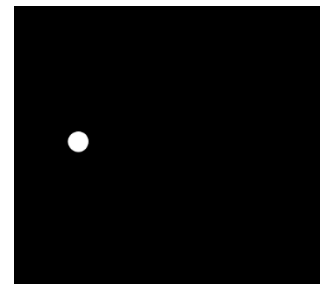
- Determine whether the current point is farther from the center than the previous one. Use a **boolean** variable to store this information. That tells you whether the line is moving away from the middle, or toward it.
- When do you change the color? When the line is now moving away from the center, but it wasn't last time. You will need a **boolean** expression to figure this out.
- Now you need to keep more information from the last frame: How far away the last point was, and whether it was moving toward the center, or away from it. Use global variables again, one of them a **boolean** variable.
- To pick a random color you can use `stroke(random(256), random(256), random(256));`



Exercise 6. This unit introduced a problem of a ball bouncing around *inside* a square (11.8). Instead, make the ball bounce around outside the square. That is, it bounces off the walls of the canvas (like inside a square), but additionally, it stays outside of a square placed in the middle of the canvas. This is deceptively challenging, and a lot harder than it looks.



Exercise 7. Mke a version of the game “Whack-a-Mole”, where you have to hit targets with your mouse before they disappear. This will be challenging, with quite a bit of logic, and many things to keep track of, using global variables. The game works as follows. A “mole” gets generated at a random location. It starts as a circle of size 0 and then grows at some speed until a maximum size is reached (use global named constants, of course). Then it starts shrinking. When it reaches size 0, a new mole starts to grow at a new random location. But if the user clicks on a mole before it disappears, the mole is instantly “killed”, the player’s score increases by one, and a new mole begins somewhere else. Hint: Use the



Quick! Click on the circle before it disappears!

distance from the mouse to the center of the mole to detect a “hit”. After this much of the game is working, you can make it fancier. Make the moles different colors or sizes or speeds. Give more points for smaller or faster ones. Make the moles go faster the longer the game lasts. Add a “game over”. Print the score in the window (look up the “text” command). Have fun with it!

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

(page intentionally left blank)