

UNIT 5. ACTIVE PROCESSING

Summary

This is a difficult unit – we finally move away from boring calculation programs and start to have programs that animate and you can interact with. These are called *active* programs in the Processing lingo. In this section, you will...

- ♦ Move away from static programs toward interactive, dynamic programs
- ♦ Learn about code blocks, logical units or groups of programming commands
- ♦ See how a variable, once created, only works in some parts of the program: it has a scope within which it's valid
- ♦ Learn how to get the mouse position
- ♦ Learn some new commands for finding out which of two numbers is larger or smaller

Learning Objectives

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

- ♦ Create an active program that can change while it is running
- ♦ Create animations where the drawing changes as time passes
- ♦ Use the mouse location as input into your program
- ♦ Calculate which of two numbers is larger or smaller

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.



5.1 Introduction

So far, you have been using what processing calls *static sketches* – your programs run straight through once and you see the final result. This is cool, but it's time to start doing something more exciting: *active sketches*!!!!

Active sketches start up, and then keep on running and running and running, repeating the program and making changes, until you press stop. This is exciting as it opens up a whole new world of programming, such as animations and programs that act based on your mouse and keyboard actions. So far, you use Processing just by typing commands in. To switch over to active mode, there are no settings in Processing – instead, it's how you write your program.

First, here is a magic line of computer code that draws a line from the center of the screen to a random location. We don't yet have all the tools necessary to properly learn the `random` command, so for now, please be patient with just typing this in and not fully understanding it. Assuming that our canvas is 500 by 500, here is a complete program:

```
size(500,500);  
line(250,250,random(500),random(500));
```

run this program, and you will see a random line from the center of the screen to some spot. Run it again. And again... and you will get a different line each time. That is each time `random` is called, it generates a random number for the `line` command (in this case, from 0 to 499, excluding 500, but we'll learn that in more detail later). This illustrates the limitation of the static program – it only runs once. To move forward, let's learn about active sketches.

5.2 Active Sketches

To move from static to active sketches, we just need to type in our program a little differently. In an active sketch, you actually specify two chunks of computer code: code that runs just one time at the start of your program, and code that gets repeated. We call these two fundamental regions “blocks” of code. **There is a setup region – this is code that runs once at the beginning of your program.** This is kind of like the static programs you have been creating already. The second region is called **a draw region. This code gets run over and over and over again.** In fact, since computers are so fast, Processing tries to run it 60 times every single second. You can change this speed, but that's beyond the scope of this course.

Active sketches require some new syntax. To mark off these sections of your program – the run once setup code, and the repeating draw code – you need to learn about code blocks. Processing marks off a block of code by sandwiching it between

Zoom!! Almost as fast as I can run!



two curly brackets: { and }. For example:

```
{
  // this is a code block
}
```

Be careful not to use the square brackets [] or the round ones () (and especially not the triangle ones < >) as they mean different things in Processing.




Notice how I indented the code inside the block? This is a good practice to get into as it makes your program easier to read. You'll get a lot of practice with this, especially as we start using blocks all over the place later on. Hint: **The Processing editor has a feature to automatically do your indenting. It is hidden in one of the menus – can you find it?**

You don't usually create a code block just by itself, like in the above example, but it gets attached to something. In this case, we need to give these code blocks names so processing knows which one is which. We simply put the name in front with some special syntax.

```
void setup()
{
  // run once
}

void draw()
{
  // run 60 times every second
}
```

What exactly does this mean? For now, just memorize this syntax. The word void means *empty* or *nothing*, meaning the block doesn't generate any data for others to use. The empty brackets () means that the block doesn't require data when it is started to get going. This is confusing, and you will learn more about it later in the course.



Advanced: You have already seen many commands, like `line`, `ellipse`, etc., that require parameters (data) to run, e.g., screen coordinates. When we create blocks, they can likewise be configured to take data (we'll learn much later in the course). We haven't yet seen any commands that give you data (except `random`, in passing), but blocks can likewise return (result in, and pass along) data to be used elsewhere. The syntax for creating a block in processing enables you to specify parameters and

return data, even though we won't be using it for a while. Here is the general syntax for setting up a block, which we will learn later:

```
resultType blockName(paramType1 paramName1, type2 name2,...)
{
    // block of code
}
```

Our setup and draw blocks do not provide any results (like `line()`) and do not take any parameters, so we can set them up with the `void` return type and no parameters.

Try typing the setup and draw code from the previous page into Processing and make sure it runs. However, since there is no code inside the blocks, nothing should happen but seeing an empty canvas. Make sure there are no errors.

So what exactly is happening here?



- ♦ 1) Processing runs the `setup` block one time. This is exactly like your static Processing programs you have been using throughout the course.
- ♦ 2) Processing runs the `draw` block. It runs every command top to bottom.
- ♦ 3) Processing waits until it's time to draw again, and then runs the `draw` block again. This repeats at about 60 times a second (60 Hertz) until you close the program or until an error is encountered.



Advanced: Why does processing need to wait before drawing again? 60 frames per second (FPS) may seem very fast, but from the perspective of a computer that measures time in billionths of a second, the time between frames is very long. For 60 FPS, there is a delay of about 0.0166 seconds between drawing each frame. On the billionths of a second time scale, that is equivalent to 16 million clock ticks, in which time the computer can get a whole boatload of work done. So, it needs to wait before drawing again, and will probably check your email, do some networking, etc.

Let's rewrite our earlier random line example, from the beginning of the unit, in active Processing. Let's put everything into the run-once setup block.

```
void setup()
{
    size(500,500);
    line(250,250,random(500),random(500));
}
void draw()
{
}
```

This should run as expected. The setup code runs once only – this acts the same as your static sketch did before. The repeating draw block is still empty.

Now, let's see what happens if we move the line command into the draw block. Keep the size command in the setup block because we only want it to run once, at the start of the program.

```
void setup()
{
  size(500,500);
}
void draw()
{
  line(250,250,random(500),random(500));
}
```

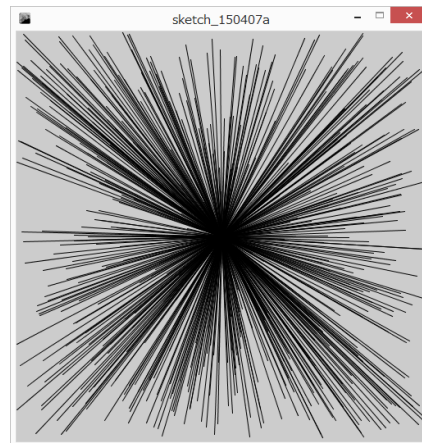
Yay!! We now have an animated program. Let's step through what is happening.

- ♦ 1) Processing runs the `setup` block one time. The `size` command gets run and we have a 500x500 canvas.
- ♦ 2) Processing runs the `draw` block one time. A line is drawn from 250x250 to some random location.
- ♦ 3) Processing waits until it's time to draw again.
- ♦ 4) Processing runs the `draw` block again. A line is drawn from 250x250 to a *new* random location. Repeat.

After just a few seconds, we see something like on the right.

We can now put any code we want into the draw block, and it will play over and over and over again.

NOTE: in active Processing, all commands generally must go inside a block. The only exception is creating variables, which we will get to below.



5.3 Variables and Blocks

Now that we have blocks, variables get a little more complicated – where do you define them?

Let's try the following example: what do you reasonably expect to happen?

```

void setup()
{
  size(500,500);
}
void draw()
{
  int linePosition = 0;
  line(250,0,linePosition,linePosition);
  linePosition = linePosition + 1;
}

```

Let's step through this. The `setup` block is run one time, and the canvas size is set. The `draw` block is run. The variable `linePosition` is created and set to 0. The line is drawn from 250,0 to `linePosition`, `linePosition` which is 0x0. `linePosition` is increased by 1. So far so good. Processing waits until it is time to draw again, and then runs the `draw` block again. Next time, since we changed `linePosition`, will it draw to 1,1? And then 2,2? Run it to see.

If you try this out, it doesn't work – we only get a line from 250,0 → 0,0. Apparently, the value in `linePosition` doesn't increase as we expect. What the heck is going on?



Variables and blocks have some very important properties. **If you define a variable inside a block, it only exists until the block is finished. You can do it, but once the block is over, the variable actually gets destroyed.** This is particularly annoying for the `draw` block, since it runs 60 times a second. Any variable you create in there gets destroyed and re-created EACH TIME, losing its data from before. So each time through in our example, `linePosition` gets created, initialized to 0, used, increased by 1 (to a value of 1), and then destroyed, to be re-created on the next draw. It doesn't hold its value.

What happens if we move the variable declaration to the `setup` block instead? This way, perhaps the variable can be created only once, but used many times?

```

void setup()
{
  size(500,500);
  int linePosition = 0;
}
void draw()
{
  line(250,0,linePosition,linePosition);
}

```

```
linePosition = linePosition + 1;
}
```

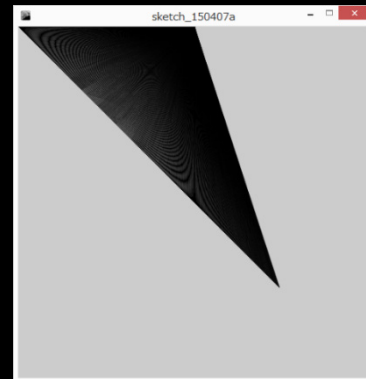


Try to run this. Processing complains and raises an error. It says: Cannot find anything named "linePosition". That's strange, we created it right there! This raises another property of variables and blocks: **A variable created inside one block is only visible from within that block – other blocks cannot see them!** This is called a **scope rule** – it defines the area where a variable is visible. So, the variable we created inside setup cannot be used or seen inside draw, since its scope is limited to setup.

The combination of the two above rules regarding variables and scopes is a little complex, but they will stick with you throughout the course – pay special attention to them. A) variables only exist within the bloc they were created, and get destroyed at the end. B) variables created inside one block is only visible from within that block.

In this case, the solution is to move our variables outside our blocks to the top of our program. These are called **global variables**. This way, they are created outside the blocks. They are created when the program starts. They exist until the program ends – they do not get destroyed. They are also visible to all our blocks:

```
int linePosition = 0;
void setup()
{
  size(500,500);
}
void draw()
{
  line(250,0,linePosition,linePosition);
  linePosition = linePosition + 1;
}
```



Cool! It works! We draw a line from that spot across the whole diagonal. You may notice that instead of being solid black, the region has a funny curvy wavy pattern. This is as a result of *anti-aliasing*, an advanced graphics technique that is way beyond this course! Feel free to ask your instructor about it.

I'm anti-aliases too – people should use their real names!



One more thing. Why does the line build up a black region across the canvas, instead of just moving along? What do you do if you want an actual line moving across – and not leave that

trail behind? Well, just like a real canvas, if you keep drawing, your paint adds up. If you want an animation with something moving, you need to clear the canvas each time. Each time you draw, you need to actually erase the old paint first. We learned this back in chapter 2. The `background` command erases to a color, so you can add this at the beginning of the `draw` block to start over with a fresh clean canvas each time! (try `background(127);`);

Where do you put the background command? If you put background at the end of the drawing, what happens? It erases all the paint *after* you do your drawing but before showing it on screen, so you end up seeing nothing. Put the `background` command at the beginning of the `draw` block to start with a fresh canvas. Here is the updated program

```
int linePosition = 0;
void setup()
{
  size(500,500);
}
void draw()
{
  line(250,0,linePosition,linePosition);
  linePosition = linePosition + 1;
}
```

Now is a good time to re-evaluate the order that things happen in, since we have global variables. **When you press run, this is what happens, in the order specified:**

- ◆ 1) the global variables are created and initialized
- ◆ 2) the setup block is run one time
- ◆ 3) the draw block is run. Wait and repeat, so that it is run at 60 times per second

5.4 Processing Globals – mouse position

In addition to making your own global variables, Processing provides some global variables for you that you can use in your program. One of the best ones (!!)

 is mouse position, which you can access through the following global variables:

```
// current position of the mouse pointer, x and y
mouseX
mouseY
// the previous position of the mouse, last time we drew
pmouseX
```



```
pmouseY
```

Note that these are variables that you can use, and are not commands. What is the difference? Commands do work underneath the hood. You also call commands with parenthesis () at the end of them. These are variables, so you can just read their current value like any other variable.



Note: The mouse position variables store the position of the mouse at the beginning of the draw block. It may have already moved by the time you draw something! This usually doesn't matter, but may explain some strange effects you may see.

You can use these variables at any time. Try the following sample:

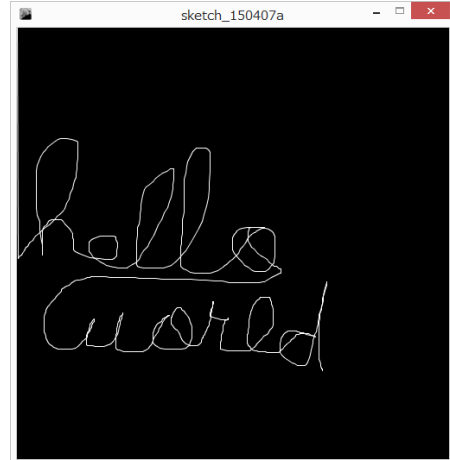
```
void setup()
{
  size(500,500);
}
void draw()
{
  background(0);
  stroke(255);
  line(0,0,mouseX,mouseY);
}
```

Very cool! You can make a line draw to the mouse cursor! Each time it runs, it clears the background to black, sets the color to white, and draws from 0, 0 to the mouse location. Try moving the mouse around while the program is running.

Here is another quick example. Here, if we draw a line from the previous position to the current one, what do we get?

```
void setup()
{
  size(500,500);
}
void draw()
{
  background(0);
  stroke(255);
  line(pmouseX,pmouseY,mouseX,mouseY);
}
```

This one is a lot of fun, you have to really try it – it draws a line behind the mouse, like you're being followed by a tiny worm. What happens if you remove the background command – so that we don't erase the paint each time? Move that command to the setup block so that you only clear the background once. You have a painting program!!



5.5 Example – hold the roof up!

Let's draw a line across the screen that slowly falls to the bottom of the screen. However, you can hold it up with your mouse: that is, as the line falls, if it hits the mouse, it will stop. There are a few pieces to this. First, let's make an active sketch where the line falls.

We need a global variable for the current line position. This has to be global because we need to remember the line position between draws, so it can draw at a new position each time as it falls.

```
int linePosition = 0;
```

Then, in the draw block, let's increase its position by one, and draw a line across the screen at that Y coordinate. Don't forget to erase the canvas

```
void draw()
{
  background(255);
  linePosition = linePosition + 1;
  line(0,linePosition,500,linePosition);
}
```

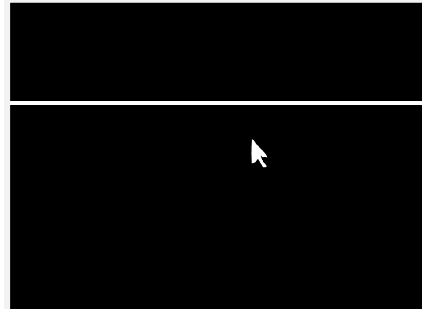
Now we have a line that starts at a y of 0, and increases by 1 each frame, to fall down the screen. Now for the second part: how can we push it back up with the mouse? Well, what we need to do is to compare the mouse y coordinate with the line. If the mouse is above the line – if the y is smaller – then move the line to that point. Luckily we have some helper functions that we can use here. We have

```
max(a,b); // gives you the biggest of two numbers
min(a,b); // gives you the smallest of two numbers
```

This is a little tricky, but using `min`, we can find out which is smaller – the line position or the mouse – and set the line to that.

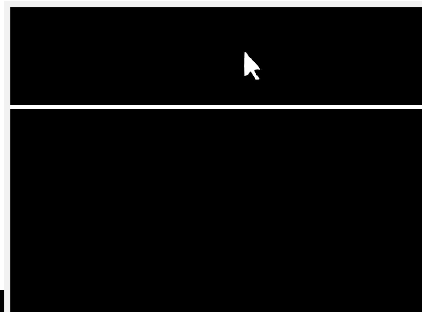
```
linePosition = min(linePosition, mouseY);
```

Think through this for the second. The `min` command takes the `linePosition` and the `mouseY`, and returns whichever is smaller. We have two basic cases, shown in diagrams here. If the `linePosition` is smaller than the `mouseY` (higher up, as in the first image), then `min` will give us the line position. So in this case, the line position actually doesn't change from the `min` command, and the line just keeps falling. As expected.



```
linePosition = min(linePosition, mouseY);  
// if line position is smaller than mouseY,  
// then this essentially becomes  
linePosition = linePosition; // no change!
```

In the other case, if the `mouseY` is smaller than the `linePosition`, then the `min` command returns `mouseY`. So, the line position jumps to the mouse location. In this case, the line cannot fall anymore, because each time, the line gets brought back to the mouse position.



```
linePosition = min(linePosition, mouseY);  
// if mouseY is smaller or equal to line position,  
// then this essentially becomes  
linePosition = mouseY; // move the line to the mouse
```

This solves the exercise! You can now hold up the line. Here is my final code:

```
int linePosition = 0;  
void setup()  
{  
  size(500,500);  
}  
  
void draw()
```

```

{
  background(255);
  linePosition = linePosition + 1;
  linePosition = min(mouseY, linePosition);
  line(0, linePosition, 500, linePosition);
}

```

Now is a good time to look closely at this program. Do you know what order things happen in? Do you remember exactly what happens, step by step? This course will only start to get more complicated, so now is the time to ensure that you understand the mechanics here – you need to know what happens, in what order, piece by piece.

5.6 Example – make the cat face active

Let's try to update our earlier cat face to be an active sketch. One great thing we did was to make the cat nose and whiskers all draw relative to some variable values. Don't remember? Re-visit Unit 3 for a quick refresher. What happens if we link these variables to the mouse coordinates?

First, type up the last cat example again from Section 3.8. Run it to make sure it works.

To make the program global, you need to break the program into the `draw` and `setup` blocks. In this case, keep the variables global. Remember that no commands except for the variable declarations can be outside the blocks. How do you decide what goes into the `setup` and `draw`? Remember: do once, goes in `setup`. Draw each time, goes in `draw`. In this case, the `setup` only needs the canvas size setting.

Next, since we are re-drawing the cat rapidly, make sure to clear the canvas (probably to black?) on each draw. This way, if we move the cat, we won't have the old paint kicking around. Now, run your program to ensure it works.

Finally, let's animate the mouse. Currently, the nose center of the cat is linked to the global variables `noseCenterX` and `noseCenterY`, which are set to static values at the start of the program. To animate the cat, instead update the nose center based on the mouse position. At the beginning of the `draw` block, copy the mouse position into the nose center variables. Now you have an animated cat face! Try running it!

Just in case you hit a snag, here is my final cat-face code. You should try linking the mouse to other things, such as eye position, size, etc. Play around with this.

```

/*****
 * Cat Face! Draw a cat face on the screen
 * author: Teo the dog

```

```

* version: try #awesome
* purpose: to show how a cat can be drawn
*****/

// variables
int headCenterX = 250;
int headCenterY = 250;
int noseSize = 30;
int pupilWidth = 15;
int noseCenterX = headCenterX;
int noseCenterY = headCenterY+50;

void setup()
{
  size(500, 500); // make a 500x500 canvas
}

void draw()
{
  background(0);
  // copy the mouse position into the nose position
  noseCenterX = mouseX;
  noseCenterY = mouseY;

  //draw the head
  ellipse(headCenterX, headCenterY, 300, 300);

  //draw the ears
  triangle(headCenterX+125, headCenterY-170,
    headCenterX+50, headCenterY-100,
    headCenterX+150, headCenterY-50);
  triangle(headCenterX-125, headCenterY-170,
    headCenterX-50, headCenterY-100,
    headCenterX-150, headCenterY-50);

  //draw the eyes
  ellipse(headCenterX-75, headCenterY-25,
    pupilWidth*4, pupilWidth*2); // left eye
  ellipse(headCenterX+75, headCenterY-25,
    pupilWidth, pupilWidth*2);

```

```

ellipse(headCenterX+75, headCenterY-25,
  pupilWidth*4, pupilWidth*2); // right eye
ellipse(headCenterX+75, headCenterY-25,
  pupilWidth, pupilWidth*2);

//whiskers!
line(noseCenterX, noseCenterY,
  noseCenterX-50, noseCenterY-25);
line(noseCenterX, noseCenterY,
  noseCenterX+50, noseCenterY-25);
line(noseCenterX, noseCenterY,
  noseCenterX-60, noseCenterY);
line(noseCenterX, noseCenterY,
  noseCenterX+60, noseCenterY);
line(noseCenterX, noseCenterY,
  noseCenterX-50, noseCenterY+25);
line(noseCenterX, noseCenterY,
  noseCenterX+50, noseCenterY+25);

// draw the nose after whiskers for nice overlap effect
ellipse(noseCenterX, noseCenterY, noseSize, noseSize);
}

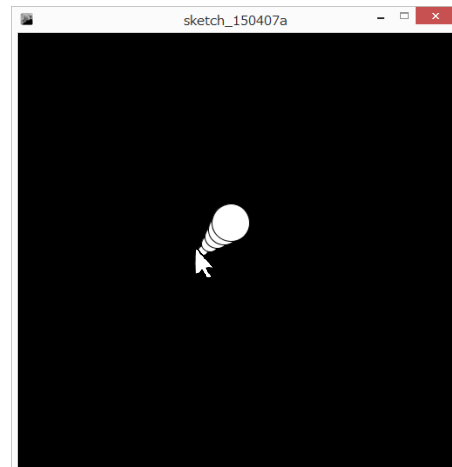
```

5.7 Example – mouse bubbles

Let's make bubbles come out from the mouse cursor. Like in the inset, except animated – they come up out of the mouse and then pop, and another one comes up.

How do you attack this problem? The bubble is size 0 at the mouse pointer and slowly grows to some maximum size (say, 50). As it grows, it moves up (-y) and right (+x).

First, just think about the static cases. This is a common technique for tackling a larger problem. If we can solve the basic problem in a few of the static cases, it makes seeing the animation solution a little easier. Let's draw a bubble of size 0 at the mouse pointer:



```

ellipse(mouseX,mouseY,0,0);

```

clearly this is useless. But, what bubble is next? It's a size 1 bubble, with +1 x and -1 y in comparison to this one.

```
ellipse(mouseX+1,mouseY-1,1,1);
```

What bubble is next? Size 2

```
ellipse(mouseX+2,mouseY-2,2,2);
```

Do you see the pattern? At each step, we add to x the step amount, subtract it from y, and make the ellipse that size. Look at them lined up

```
ellipse(mouseX,mouseY,0,0);  
ellipse(mouseX+1,mouseY-1,1,1);  
ellipse(mouseX+2,mouseY-2,2,2);
```

Why don't we replace those numbers with a variable and see how it looks?

```
int step = 0;  
ellipse(mouseX+step, mouseY-step, step, step);
```

Now, to draw the next bubble, we increase step by one and use the exact same ellipse code.

```
step = step + 1;  
ellipse(mouseX+step,mouseY-step, step, step); // same above
```

If we look at all three, then we can see a pattern

```
int step = 0;  
ellipse(mouseX+step, mouseY-step, step, step);  
step = step + 1;  
ellipse(mouseX+step, mouseY-step, step, step); // same above  
step = step + 1;  
ellipse(mouseX+step, mouseY-step, step, step); // same above
```

This is nice because the ellipse command is identical, and all that we do is change the `step` variable. In this case, we don't want to draw all 50 bubbles at once, we want them to animate. Also, copy-pasting this 50 times is tedious and not a good use of time. So what can we do? Since we know that `draw` runs repeatedly, if we make `step` a global variable, we can increase the step size each time `draw` runs.

This way, each draw gives us one bubble. Then it gets bigger for the next draw, and so on. It also gets further away. Here is how it may work.

```
//globals
int step = 0;
...
// in draw
ellipse(mouseX+step, mouseY-step, step, step);
step = step + 1;
```

Try coding this up. yes! We have the animation! So what's the next problem?

Step just keeps getting bigger and bigger... but we want it to reach size 50 and then start over with a new bubble.

There are several ways to solve this. However, one of the simplest ways is to use a technique that we actually already learned, but unfortunately, is not very obvious at all. We can do this using modulo (the remainder function). Let me explain...

Let's say that you have a number, n , and you divide it by, say, 4. What possible results can you get? What remainders can you get? You should try calculating out n from 1 to 20 and see what remainder you get. Here are the first 8.

n	0	1	2	3	4	5	6	7	8
$n/4$	0	0	0	0	1	1	1	1	2
$n\%4$	0	1	2	3	0	1	2	3	0

Do you notice anything special about the remainder? What possible remainders can we get? If we do $n/4$, can we get remainder 5? Why not?

Remainders of n/m are always in the range from 0 to $m-1$. In this case, if you divide a number by 4, it is always in the range 0...3. Once it hits 4, the division result gets bigger, and the remainder re-starts at zero. This is quite confusing, think on it a minute.

So how does this help us? We can use modulo to make our step size loop. If we do the following:

```
step = step + 1;
step = step % 50; // remainder when /50
```

Then step will go from, 0,1,2,...,49. When it hits 50, the remainder is 0 again. You may have noticed that I can write the above two lines in one single line:


```
step = (step+1)%50; // same as above.
```

And it works! Step gets bigger, but once it hits 50, it starts at 0 again. We're done this example! Here is my final code:

```
int step = 0;

void setup()
{
  size(500, 500);
}
void draw()
{
  background(0);
  ellipse(mouseX+step, mouseY-step, step, step);
  step = (step + 1)%50;
}
```

Make sure you take time to really understand what is going on here. There are only a few lines of code, but this is suddenly a lot more complex than what you have been seeing in previous units. Now is the time to get help with this if you're not following, since things are just going to get harder from here.



Check your Understanding

5.8 Check Your Understanding: Exercises



Exercise 1. Create an active Processing program that draws a circle directly under the mouse as it moves around. Make the circle have a radius of 20, and make it white against a black background. There should be no trails left behind the mouse as it moves around.

- Update this example so that the width of the ellipse is 1 tenth the `mouseX` coordinate, and the height is 1 20th of the `mouseY`.



Exercise 2. Create an active Processing program that visualizes the remainder (modulo or mod) operator (%). You will do this by making a block that follows the mouse, but the block's size will be the remainder when the x coordinate is divided by 100. Specifically:

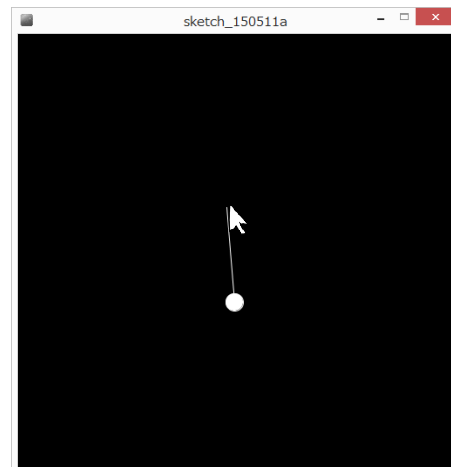
- Create the `setup` and `draw` blocks, and resize the canvas to 500x500 in the `setup` block.
- Create the following global integer constants (use "final" to declare

- them) and set them to reasonable greyscale values: `BG_COLOR`, `BOX_COLOR`
- In the `draw` block, create the following local variables and set them to the current mouse coordinates: `boxX` (set to `mouseX`), `boxY` (set to `mouseY`)
 - Create a local variable `boxSize` and set it temporarily to size 30.
 - Use the `rect` command to draw the box using `boxX`, `boxY`, and `boxSize`. Don't forget to set the fill color.
 - Run your program and make sure that the box follows the mouse around. Notice that the box touches the mouse at the top left corner.
 - Next, make the box centered on the mouse. Offset `boxX` and `boxY` by half the size of the box. That is, subtract `boxSize/2` from `mouseX` and from `mouseY` when setting `boxX` and `boxY`. This will make the box centered on the mouse. Be sure to use the `boxSize` variable and not literal numbers.
 - Run your program – it should now follow the mouse but be centered on the mouse.
 - Now, to visualize mod, set the box size to the remainder when you divide the `mouseX` coordinate by 100. Remember that you can get the remainder by using the `%` operator. The size of the box will now change to values from 0 to 99.
 - Run your program. You should now see what the remainder looks like as you move the mouse from left to right across the x axis – it gets larger until it reaches its maximum size, then resets to size 0, and repeats!
 - Finally, after calculating `boxSize`, but before using it to calculate `boxX` and `boxY`, make sure that the size is never less than 10. Use either the `min` or the `max` function to do this.



Exercise 3. Create a program that simulates a ball hanging from the mouse on an elastic band.

- Create an empty active Processing program with `setup` and `draw` blocks, and a canvas of 500x500.
- Create global integer variables to keep track of the ball's position. Also create variables to set the ball's color and size. The ball can start initially at the center of the canvas.
- Make the `draw` block draw the ball and clear the canvas each frame. Use global variables. Your program should run now although not much

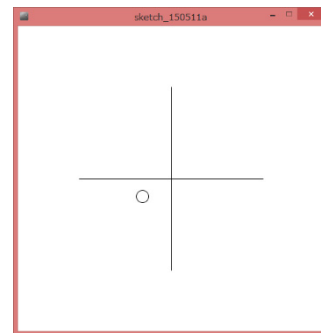


will happen.

- d. In each iteration of the draw block, make the ball move toward the mouse. This is tricky, you do it as follows. Do the X and Y separately.
 - i. Calculate how far the ball is from the mouse in terms of X: create a new local integer variable called `diffX` and set it to `mouseX - ballX`.
 - ii. If we add `diffX` to `ballX` (`ballX = ballX + diffX`) then the ball will move to the mouse. Try the algebra on a piece of paper and you will see it.
 - iii. Instead, add 1/10th of the difference to `ballX`. Do this in one line, by setting `ballX` to `ballX` plus `diffX` divided by 10.
 - iv. Do this for the Y coordinate as well in the same way
- e. Draw a line from the ball to the mouse to simulate an elastic.
- f. Now, add gravity! In each frame, simply add 10 pixels to the Y. The ball will fall, and, will stop falling when it tries to move 10 pixels toward the mouse but is pulled down 10 pixels by gravity.



Exercise 4. Create a program where the mouse acts like a joystick to drive a ball on the screen. In the picture shown you can see a cross hairs, and a ball. This program will work as follows: if the mouse is at the crosshairs, the ball doesn't move. If the mouse is to the right or left, the ball moves right or left. If the mouse is above or below the center, the ball moves up or down. The distance of the mouse from the center determines how fast the ball moves.



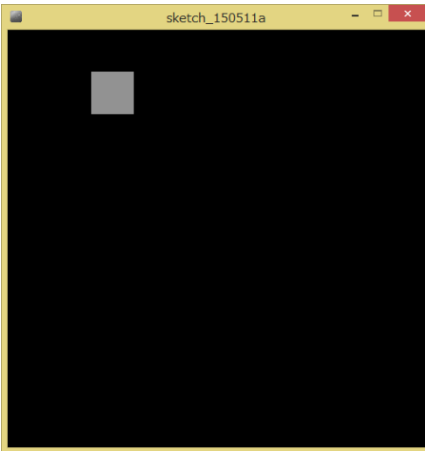
- a. First, draw the crosshairs. The exact dimensions do not matter as long as they center in the middle of the screen.
- b. Create some global variables to remember where the ball is. Start the ball at the center of the screen. Don't forget to draw the ball!
- c. Calculate how much the ball should move in the x and y directions separately and store in the variables `moveX` and `moveY`.
 - i. For x, you use `mouseX-250`. If the mouse is at 250 (the screen center), then we get 0, no movement! If the mouse is left of 250, you get negative movement! If it is to the right, you get positive movement.
 - ii. Do the same for Y.
- d. Add the `moveX` and Y to the ball position.
- e. Run your program, the ball should move. But – Whoa! It flies off the screen! Use `min` and `max` to make sure the ball stays within visible range
 - i. Use one check for the right side of the screen, and one for the left. It is easy to get `min` and `max` backward here, so try an

example on paper.

- ii. Use another set of checks for the top and bottom of the screen
- iii. Can you figure out how to do it so that the whole ball stays on the screen and not just half? The basic solution works off of the ball center, so it goes over the edge...
- f. Finally, the ball moves way too fast. How can you make it move slower? Just use division on the move speeds.



Exercise 5. Let's write a processing program that uses some clever math to make visual effects. This program will have two distinct features: a rectangle will snap to a grid under the mouse (move along the grid as you move the mouse instead of being right under the mouse), and, the color will follow a simple formula that makes it look unpredictable.



- a. Create a clean active Java program, with a canvas sized 500
- b. You will make a rectangle snap to a grid 10 by 10 (100 places). Make a global called `gridSize` and set it to 10. In the draw loop, make a local variable called `cellSize` (how big each cell is) and store the size of each cell. Hint: screen size divided by grid size.
- c. Calculate the top left corner for a rectangle to draw. This corner will have to snap to this grid. That is, it can only be $0,0$, or $cellSize*i$, $cellSize*j$ for some integers i and j . You will find the nearest grid point to the top left of the mouse cursor. There is a trick to it.
- d. Integer division always throws away the fraction part. If you divide the mouse coordinate by the cell size, it will tell you how many cells along the mouse is. Do this separately for x and y and store in `int cellX`, `cellY`. This is not the coordinates on the screen, but rather, the cell number
- e. Convert from the cell numbers to screen coordinates. This is easy. Just multiply the cell number by the cell size. Cell 0 gives you 0, cell 1 gives `cellSize`, and cell 2 gives $2*cellSize$
- f. Draw a rectangle at this cell
- g. Now you should be done the drawing part. Set some simple color for now and debug up to this point before moving along. Next let's work on the color.
- h. Create a global integer called `runningColor` to keep track of the last color we used. Then, each frame calculate the color as follows. This is not something you should develop an intuition for, it's just a silly bit of math to make the color act interestingly as you move the mouse

around.

- i. Take the product of the `mouseX` and `runningColor` and add `mouseY` to it. Then modulo the result by 256, and multiply by -1. Store this final result as the new `runningColor`.
- j. Before setting the current fill and stroke color to running color, make a copy and cap it above zero in case it happened to go negative. Don't store the capped number in `runningColor`, if its negative let it stay that way for next time.

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)