UNIT 14. USER-DEFINED FUNCTIONS PART 2

Summary
You will see in this unit how to use the full power of user defined functions. You will…

- see how to pass information into functions, and information back from calculations done in a function.
- see common pitfalls associated with functions
- learn how to avoid using changing global variables in functions when possible

Learning Objectives
After finishing this unit, you will be able to …

- create a function to take one or more parameters
- create a function that returns data
- have one function call another function
- apply top-down programming to simplify and solve complex problem

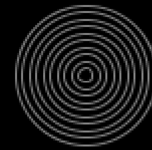
How to Proceed
- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

 © James Young, 2016

14.1 **Introduction**

So far, we learned basic user-defined functions as a technique to help us manage our programs as they get larger – we can break a program down into manageable chunks, and work on one chunk at a time. However, these have been very limited. For example, they all rely on global variables to share information, which gets confusing and difficult to keep track of. You also may have noticed that it is hard to make functions that you can easily re-use, as they may require specific data that changes, and you end up writing multiple similar functions.

Here is an example that illustrates the limitation we currently have with functions. Here is a function that draws a spider web, approximated just by drawing a tight series of circles:



```
float webX = 200;
float webY = 200;
int webRings = 10;
int webColor = 128;
int webSpacing = 7;
void drawSpiderWeb()
{
  fill(0);
  stroke(webColor);
  for (int i = webRings-1; i >=0; i--)
  {
    float ringSize = (i+1)*webSpacing;
    ellipse(webX, webY, ringSize, ringSize);
  }
}
void draw()
{
  background(0);
  stroke(255);
  drawSpiderWeb();
}
```

What if we want to draw several spider webs? For example, draw them to the left and the right of the mouse? With the current solution, we need to change all the global variables, and re-call the function, each time. This is not only tedious, it is error prone, as we may forget what we changed the global variables to. There is a better way – we can send information directly to the function, through a special, secret hidden channel!

 © James Young, 2016

14.2 **Parameters – sending data to functions**

It is great to re-use code, but what if we want slight variation on the code? As in the above example, what if we want to do essentially the same thing, but with small changes to some of the variables involved. For example, if we want a spider web at a different location, or with different spacing, but don't want to rely on changing global variables

We need a new technique: we need to add parameters to our function.

Luckily, you have been using commands and parameters for the whole class, so part of this will look familiar. Every time you use the `line` command, for example, you pass it four parameters that the command then uses to create your line. Similarly, we want to be able to pass information to our own functions. For example, instead of calling the following command in our draw block:

```
drawSpiderWeb();
```

We want to send it some information. For the first case, let's just try sending it the x coordinate of where to draw the spider web:

```
drawSpiderWeb(100);
```

If you try to run this now, it won't work, because Processing will complain that the function cannot take any parameters. We need to fix that by modifying the function.

To create a function with parameters, you modify the function header, and add the parameters inside the brackets after the command name. Starting with our simple case of only accepting one parameter, the x coordinate of the web, here is the syntax to make a function take a parameter:

```
void functionName(parameterType parameterName)
{
  //… code
}
```

In this case, we want the function `drawSpiderWeb` to take an integer parameter specifying the x coordinate of the spider web to draw. So, we re-write the function as follows:

```
void drawSpiderWeb(int x)
{
  ///…
}
```

 © James Young, 2016

Now comes a tricky part. We have now a) created a function that accepts one integer parameter, called `x`, and b) modified the function call to provide an integer (100 in this case). But, once the data is passed into the function, how do we use the data?

When the function is called, for example:

```
drawSpiderWeb(100);
```

Two things happen. First, inside `drawSpiderWeb`, a new local variable is created called `x`, and second, that variable gets set to the parameter value. So, internally we get the following hidden code:

```
int x = 100;
```

Now, you can use `x` as a local variable, just like any other.

Update your program so that the function uses this new x value, and, your draw block sends in the appropriate data, and make sure it works. You can test it by deleting the global `webX` variable.

Now, instead, replace the `drawSpiderWeb` commands in the `draw` block with the following:

```
  drawSpiderWeb(mouseX+100);
  drawSpiderWeb(mouseX-100);
```

You should see two spider webs always to the left and right of the mouse, although the y coordinate is not yet fixed. As you can see, now we can re-use our code with slight modifications, by sending data to the function. Try changing the code with different values to see it work.

Important: ***Remember the underlying code above- a local variable is created, and the data copied in.*** It will help you see that data is always only copied into a function, and placed into a new local variable. If you modify this value, the original doesn't change, it was only a copy. We will see more of this below.

Further, since this is a local variable, your same scope rules apply:

- The parameter variable is only accessible from within the function
- The parameter variable gets destroyed every single time the function finishes running.

14.3 **Example: status print**

Let's do another example. Let's make a function that can print a status message. It will print some text at the bottom left corner of the screen. In addition it will surround

 © James Young, 2016

the text with dashes and put the length at the end. For example, if you use

```
statusPrint("System OK!");
```

It will put the text at the bottom left corner of the screen: `-System OK!- 10`

In this case, we make a new function that takes a single string parameter, and then constructs the output text, before printing it to screen.

This should be reasonable to do with the tools you learned so far, so try it on your own. Keep in mind that you need one parameter, and it needs to be of the `String` type. Then, you use that parameter inside the function as a local variable.

```
void statusPrint(String message)
{
  message = "-"+message+"- "+message.length();
  text(message,0,height-1);
}
```

The nice thing about this is that you can use this anywhere in your code to print a status message on the screen. This code is reusable, and, you can send data to it to have your own custom status messages. It does not rely on any changing globals (except the canvas size), and so you can use this in any program.

14.4 **Multiple Parameters**

So far our examples only take one parameter at a time into your function. Luckily, it is very simple to add multiple parameters. All that you do is place a comma between them in the function header:

```
void functionName(type1 param1, type2 param2)
{
  //… code
}
```

You can have as many parameters as you want but most functions only have 1-3. If you find yourself with 10 parameters, there is probably a better way to do what you're trying.

Let's update our spider web example to not only take the `x` coordinate, but also the `y` coordinate. Therefore, we take a second integer parameter for this. Our new function header looks like this:

```
void drawSpiderWeb(int x, int y)
```

 © James Young, 2016

And again, inside the function we use the new variable, `y`, like a local variable.

There is one very important point here: the order of the parameters here MUST MATCH the order that you use them when you call the function. That is, if you call `drawSpiderWeb` and give it the `y` first, and then the `x`, it won't work as you expected. It will copy the first integer into `x`, and the second into `y`, to match the setup you have in the function header.

Our updated function:

```
void drawSpiderWeb(int x, int y)
{
  fill(0);
  stroke(webColor);
  for (int i = webRings-1; i >=0; i--)
  {
    float ringSize = (i+1)*webSpacing;
    ellipse(x, y, ringSize, ringSize);
  }
}
```

And in the draw loop, you call it as follows, and the spider web will stick to the left and the right of the mouse.

```
  drawSpiderWeb(mouseX+100, mouseY);
  drawSpiderWeb(mouseX-100, mouseY);
```

You can also mix and match types, just be sure again that the order that you put the parameters in the function call (the command) match how you put them in the function header. Try to update the above example by sending in the spacing of the spider web rings, and the number of rings, as parameters.

14.5 **Getting Data Back from Functions**

Sometimes functions don't only do useful things like draw on the screen, but also can do calculations and checks that provide results that you may want to use elsewhere in your program.

We have already seen a range of function that give you data. `min`, `max`, and `sqrt` are examples of function that not only take parameters, but after the work is done, it sends data back to you that you can use somewhere. Look at the following line:

```
int biggest = max(variable1, variable2);
```

 © James Young, 2016

Here, you are sending two pieces of information into the max function as parameters. When max is done, it gives you data back that, in this case, you store the data in your variable `biggest`. Let's see how to provide similar functionality in your own programs, by making functions that can provide data.

First, ***one important point to learn is that functions in Processing (and most languages) can only return one piece of data***. Period. No exceptions. This follows what you have seen in practice: `random` gives one number, `max` gives one number, `min` gives one number, etc.

Advanced: As you gain experience, you will soon see that this limitation is not acceptable. Different languages have different ways of getting around this. Some languages simply allow you to return multiple pieces of data in the language itself. Others, like C, provide a structure called a "struct" that enables you to group data together. However, throughout your CS career, you'll most likely use object oriented programming to overcome this limitation. We use none of these techniques in this course.

Let's start with a toy example. Let's make a function to calculate and return the maximum of two numbers, called `myMax` that we can use like this:

```
int bigger = myMax(10,20); // expect 20 as the result
```

First do this using the tools we already have. Make a function to calculate the larger of two numbers using an `if` statement

```
void myMax(int a, int b)
{
      int max = a;
      if (b>a)
      {
            max = b;
      }
}
```

This function takes two integer parameters, `a`, and `b`, and uses a simple technique to calculate which is larger. It assumes that `a` is larger, but then checks if `b` is, and if so, saves that one instead into `max`.

If you try to run this code as-is, it won't work: when you try to store the result from `myMax` into a variable, as in the upper line, Processing will complain because `myMax` doesn't provide any data that can be stored.

We need to learn two new things to make this function work the way we want it to,

 © James Young, 2016

to return data:

- Modify the function header to tell processing that the function returns data.
- In the function, specify which data should be returned.

The first point is easy. In the header, currently we have:

```
void functionName(type parameter1, type parameter2, …)
```

The `void` actually means that the function does not return anything. To specify a return, all that we have to do is to change `void` to the return type. In this case, that would be `int`, since our maximum of the two parameters is an integer:

```
int myMax(int a, int b)
```

Now processing expects the function to return an integer when it is done. At this point, if you try to run your program, it won't work. Processing will complain because, although you told it that your function returns an integer, you didn't specify which integer should be returned.

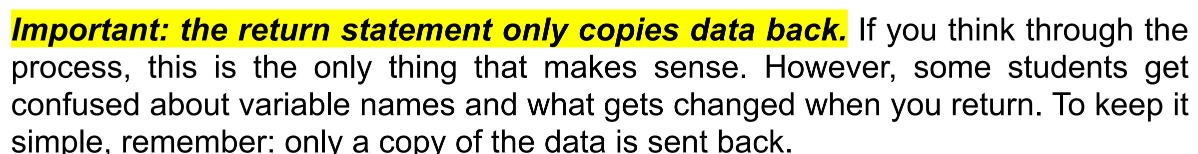
We can specify this with a new keyword, `return`. In the function, we do

```
return someData;
```

The **return** ***keyword does two things: it ends the function call and returns to the part of the program that called it, and it copies the data back to the caller.***

At this point, it is highly recommended that you put the `return` command at the end of the function as the last command. In fact, some instructors deduct marks if you do not do it this way. That is, do not use `return`s mid-function.

In the real world, experienced programmers use `return` all throughout a function, e.g., inside an `if` statement or a loop. This is not necessarily a bad thing, and can even be a good practice, if done properly. The problem is that beginner programmers have not yet developed the experience necessary to understand when this can be a bad idea and can lead to bugs, so we strongly recommend (or sometimes require) `return` to be at the end of a function. Even worse, some beginner programmers use `return` throughout a function as a crutch because they are weak at using boolean logic and control structures properly.



Important: the return statement only copies data back. If you think through the process, this is the only thing that makes sense. However, some students get confused about variable names and what gets changed when you return. To keep it simple, remember: only a copy of the data is sent back.

 © James Young, 2016

Let's update our example, and show how it could be used:

```
int myMax(int a, int b)
{
      int max = a;
      if (b>a)
      {
            max = b;
      }
      return max;
}

void draw()
{
  int small = 10;
  int large = 20;
  int bigger = myMax(small,large);
}
```

Here, the program starts at the `draw` block as usual. When `myMax` is encountered, it jumps to the `myMax` block, and copies the small and large values into `myMax`'s local variables `a` and `b`. Once `myMax` is done, the `return` statement jumps back to the `draw` block, and copies the `myMax` result back (from the `max` variable) and stores it in `bigger`. Whew!

14.6 **Example: Randomly Moving Video Game Enemies**

We are going to make a program with some squares moving randomly around the screen. Imagine that these are moving-target enemies in a video game, so we'll call them "bad guys." First we'll implement one bad guy, then three, and then update the code to use functions. The purpose of this example is to get practice, and to also reiterate and see how nicely functions can be used to clean up code as your programs grow.

Let's first do one enemy.

Start by setting some globals to define the bad guy's size and color; each bad guy will be different, as we scale up to multiple bad guys. We also want some globals to keep track of the current position. Start this guy at `0,0`. Finally, let's set some finals to define the maximum move and the background color:

 © James Young, 2016

```
final int MAX_MOVE = 20;
final int BG_COLOR = 0;
float badGuy1Size = 20;
int badGuy1Color = 255;
float badGuy1X = 0;
float badGuy1Y = 0;
```

Now, in our `draw` block, we need to make the guy move randomly. We need to handle `x` and `y` separately. First, let's do `x`. Since the maximum that the block can move is 20 in either direction, we actually have 40 possible movement positions (20 to the right, 20 to the left). So, we need to generate a random number between -20 and 20. We have done this in earlier examples, so I will not explain here how it works, but you should be able to figure the following code out:

```
float move = random(MAX_MOVE*2)-MAX_MOVE;
```

So `move` can be from -20 to 19.99, effectively 20. To move the bad guy, we simply add this to the bad guy's current `x` location., and use `max` and `min` to make sure it stays on the screen:

```
  badGuy1X += move;
  badGuy1X = min(badGuy1X, width-1);
  badGuy1X = max(badGuy1X, 0);
```

Actually, it goes off the right side until the edge. How can you fix this?

Now, we need to repeat all of the above for the `y` coordinate, including creating a new random move.

```
  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy1Y += move;
  badGuy1Y = min(badGuy1Y, height-1);
  badGuy1Y = max(badGuy1Y, 0);
```

Finally, let's draw the bad guy. Don't forget to clear the background at the beginning of your draw block.

```
fill(badGuy1Color);
stroke(badGuy1Color);
rect(badGuy1X, badGuy1Y, badGuy1Size, badGuy1Size);
```

So this isn't so bad, but your senses should start tingling when you see that repeated code for moving the `x` and `y` coordinate. Let's now scale this example up to three independent bad guys. Set them to different sizes and colors. Do this on your own.

 © James Young, 2016

Now we see a real mess… Here is my code:

```
// bad guy 1
  float move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy1X += move;
  badGuy1X = min(badGuy1X, width-1);
  badGuy1X = max(badGuy1X, 0);

  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy1Y += move;
  badGuy1Y = min(badGuy1Y, height-1);
  badGuy1Y = max(badGuy1Y, 0);

  fill(badGuy1Color);
  stroke(badGuy1Color);
  rect(badGuy1X, badGuy1Y, badGuy1Size, badGuy1Size);

  // bad guy 2
  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy2X += move;
  badGuy2X = min(badGuy2X, width-1);
  badGuy2X = max(badGuy2X, 0);

  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy2Y += move;
  badGuy2Y = min(badGuy2Y, height-1);
  badGuy2Y = max(badGuy2Y, 0);

  fill(badGuy2Color);
  stroke(badGuy2Color);
  rect(badGuy2X, badGuy2Y, badGuy2Size, badGuy2Size);

  // bad guy 3
  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy3X += move;
  badGuy3X = min(badGuy3X, width-1);
  badGuy3X = max(badGuy3X, 0);

  move = random(MAX_MOVE*2)-MAX_MOVE;
```

 © James Young, 2016

```
  badGuy3Y += move;
  badGuy3Y = min(badGuy3Y, height-1);
  badGuy3Y = max(badGuy3Y, 0);

  fill(badGuy3Color);
  stroke(badGuy3Color);
  rect(badGuy3X, badGuy3Y, badGuy3Size, badGuy3Size);
```

First, we can simplify it by creating a new function for drawing. This takes parameters for the location, size, and color, so the function can be generic and not use globals:

```
drawBadGuy(x, y, size, color)
```

The header for this will take four parameters, and return nothing, so our function header will be

```
void drawBadGuy(float x, float y, float size, int col)
```

Then, inside the function, we just use the local variables to set the colors and draw:

```
void drawBadGuy(float x, float y, float size, int col)
{
  fill(col);
  stroke(col);
  rect(x, y, size, size);
}
```

Now we can simplify our draw block code to use this function for each bad guy instead of repeating the code:

```
// bad guy 1
  float move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy1X += move;
  badGuy1X = min(badGuy1X, width-1);
  badGuy1X = max(badGuy1X, 0);

  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy1Y += move;
```

 © James Young, 2016

```
  badGuy1Y = min(badGuy1Y, height-1);
  badGuy1Y = max(badGuy1Y, 0);

  drawBadGuy(badGuy1X, badGuy1Y, badGuy1Size, badGuy1Color);

  // bad guy 2
  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy2X += move;
  badGuy2X = min(badGuy2X, width-1);
  badGuy2X = max(badGuy2X, 0);

  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy2Y += move;
  badGuy2Y = min(badGuy2Y, height-1);
  badGuy2Y = max(badGuy2Y, 0);

  drawBadGuy(badGuy2X, badGuy2Y, badGuy2Size, badGuy2Color);

  // bad guy 3
  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy3X += move;
  badGuy3X = min(badGuy3X, width-1);
  badGuy3X = max(badGuy3X, 0);

  move = random(MAX_MOVE*2)-MAX_MOVE;
  badGuy3Y += move;
  badGuy3Y = min(badGuy3Y, height-1);
  badGuy3Y = max(badGuy3Y, 0);

  drawBadGuy(badGuy3X, badGuy3Y, badGuy3Size, badGuy3Color);
```

It's a little nicer, but as you can see there is still a lot of repetition. We need to think about how to put the repeated commands into a function. In this case, there is a challenge: the problem is that our code changes the values inside the `badGuyX` and `badGuyY` variables. If we change those inside globals inside a function, then our function is not generic and we need a special function for each bad guy. Alternatively, imagine we created a command to be used something like this:

```
moveBadGuy(x, y, xmin, xmax, ymin, ymax);
```

 © James Young, 2016

which moves a bad guy, currently at x and y, and makes sure it stays within the minimum and maximum. We could use this, for example, as:

```
moveBadGuy(badGuy1X, badGuy1Y, 0, width-1, 0, height-1);
```

The problem here is that when we call this new function, all our data (bad guy X and Y) is copied into the function. Since it is only copied, any changes we do within that function will not be reflected back in our badguy variables. The only way to get information back from this function is to return it.

Unfortunately, a function can only return one piece of data, and we want a function to update both the x and y components of the bad guy. This means we cannot have a function that changes both the badGuyX and Y at the same time.

Let's reconsider the approach – notice how the movement for the x and the y are very similar. It generates a random number, it adds it to the x or y, and then checks to make sure it is not too big or too small.

In this case, the differences between the x and y cases are:

- We start with the X **or** Y coordinate
- We cap the X **or** Y to the width or height
- We store in the X **or** Y

Everything else stays the same. What if we created a function that worked on only one coordinate at the time? Something like…

```
float doMove(float position, float minimum, float maximum)
```

For example, then we could rewrite the above movement code to:

```
// bad guy 1
badGuy1X = doMove(badGuy1X, 0, width-1);
badGuy1Y = doMove(badGuy1Y, 0, height-1);
```

Assuming that the doMove function takes a position, adds a random move to it, caps it to safe screen bounds, then this is a great simplification!! So, let's try to make the doMove function. First, the header takes the three parameters, all of them are floating point, and returns the floating point result.

```
float doMove(float position, float minimum, float maximum)
{
```

Then, we use these parameters inside the doMove block to do the work. Remember,

 © James Young, 2016

we use these as local variables and completely ignore what names or variables are used in the draw block that calls this function (`badGuyX`, etc.) as all the values are copied in:

```
float doMove(float position, float minimum, float maximum)
{
  float move = random(MAX_MOVE*2)-MAX_MOVE;
  position += move;
  position = min(position, maximum);
  position = max(position, minimum);
  return position;
}
```

The return statement is absolutely necessary!! Remember that when the function is called, the data is only copied in. The changes made inside the function do not reflect back in the original code, since only the copies are changed. In this case, the changed copy needs to be copied back to the main code.

There is one more thing to notice here. This function only uses globals that are final and constant – all the changing data comes in via the parameters. This is good practice and a good habit to get into.

Using this new function we can now re-write that draw code more simply. I include the complete final program below.

Even though we now have two additional functions, the `draw` code becomes a lot easier to wrap our heads around. Repetitive code is kept to a minimum since it is factored out to the functions. Scroll back and look at the initial example, how long and repetitive it was, to compare.

Here is the final program. Notice that although, overall, it is still pretty long, each section – trying to solve a specific problem – is much smaller and more easily digestible. You are able to focus on one aspect at a time instead of solving everything at once.

```
final int MAX_MOVE = 20;
final int BG_COLOR = 0;
float badGuy1Size = 20;
int badGuy1Color = 255;
float badGuy1X = 0;
float badGuy1Y = 0;
float badGuy2Size = 40;
int badGuy2Color = 100;
```

 © James Young, 2016

```
float badGuy2X = 0;
float badGuy2Y = 0;
float badGuy3Size = 5;
int badGuy3Color = 180;
float badGuy3X = 0;
float badGuy3Y = 0;

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

void drawBadGuy(float x, float y, float size, int col)
{
  fill(col);
  stroke(col);
  rect(x, y, size, size);
}

float doMove(float position, float minimum, float maximum)
{
  float move = random(MAX_MOVE*2)-MAX_MOVE;
  position += move;
  position = min(position, maximum);
  position = max(position, minimum);
  return position;
}

void draw()
{
  background(BG_COLOR);

  // bad guy 1
  badGuy1X = doMove(badGuy1X, 0, width-1);
  badGuy1Y = doMove(badGuy1Y, 0, height-1);
  drawBadGuy(badGuy1X, badGuy1Y, badGuy1Size, badGuy1Color);

  // bad guy 2
  badGuy2X = doMove(badGuy2X, 0, width-1);
  badGuy2Y = doMove(badGuy2Y, 0, height-1);
```

```
  drawBadGuy(badGuy2X, badGuy2Y, badGuy2Size, badGuy2Color);

  // bad guy 3
  badGuy3X = doMove(badGuy3X, 0, width-1);
  badGuy3Y = doMove(badGuy3Y, 0, height-1);
  drawBadGuy(badGuy3X, badGuy3Y, badGuy3Size, badGuy3Color);
}
```

14.7 **Common Function Pitfalls**

Function are a core part of computer programming and you will use them from now on, ubiquitously, in your programming career. Once you get the hang of them they will be really simple. In the meantime, there are a few hiccups that often catch people.

The key one is remembering that ***the data is only copied***. Some people think that if you use a variable in a function call, like…

```
int maxSize = 50;
drawRandomSquares(maxSize);
```

Then somehow what happens inside the function can change the `maxSize` value. The code above *guarantees* that `maxSize` still has size 50, no matter what happens inside the function call.

Sometimes, to make things more confusing, the function will have the same parameter name as the variable you are using. For example:

```
void changeData(int data)
{
 data = data * 2;
}
void draw()
{
 int data = 4;
 changeData(data);
 println(data);
}
```

Here, in the `draw` block, we call `changeData` by passing the `data` variable as the parameter. Even though `changeData` has a parameter called `data`, because of scoping, these are completely different variables. Even though `data` changes inside the function, the original `data` remains unchanged. The `println` output is 4, regardless of what happens inside the function.

 © James Young, 2016

Let's break this down. The function draw runs. Creates a local variable called data, and sets it to equal 4. Then it calls the changeData function, and copies the information in data (4) to it. Only the number 4 is copied, the function has no clue what variable that data came from.

Inside the changeData function, a new local variable is created called data. This coincidentally has the same name as the variable in draw, but since they are in different blocks (in different scopes) they are completely unrelated. The data passed to the function (4) is copied into our new local variable, data. Inside changeData, we multiply this by 2, and set the data variable to 8. We return to the draw function, but no data is returned. The local variable in the function, data, is destroyed and the data lost.

Since the information in the data variable in draw was only copied, the data in our draw block didn't change. It is still set to 4. The println prints out 4.

This can be quite confusing. The best way to get this is to type up the program and tinker with it.

Another point about the above example, is that some people think that the parameter name in the function call must match the name in the function header. Here, the variable in the draw block could be called anything, and the code would work the same. The same name used, data, is just a coincidence.

The function scope rules also explain another case:

```
int myMin(int a, int b)
{
  int result = a;
  if (b<a)
    result = b;
  return result;
}

int myMax(int a, int b)
{
  int result = a;
  if (b<a)
    result = b;
  return result;
}
```

Here, notice how two function use the same variable names for parameters. Is this okay? Yes – because of local scope. Each function has its own scope, so can have

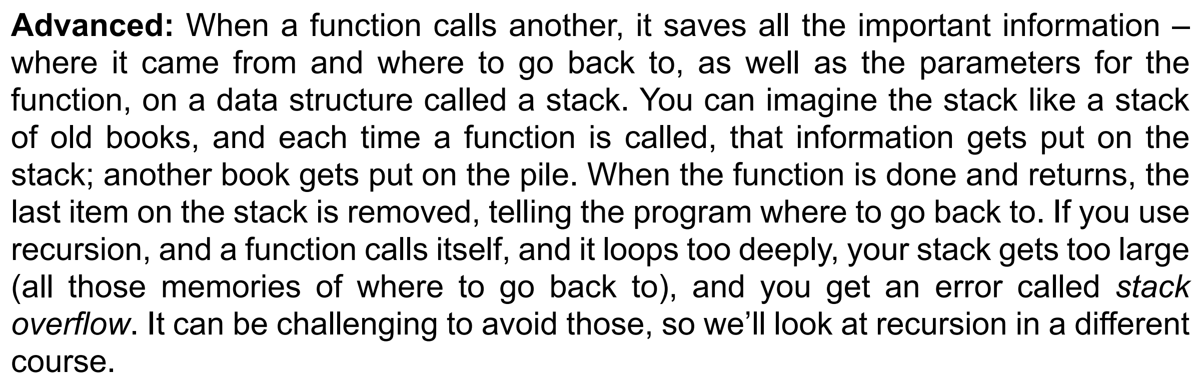
 © James Young, 2016

variable names. Outside of the variable scope, it cannot be viewed or accessed, so you don't need to worry about it.

A trick for dealing with these issues is to ***keep tunnel vision***. When you are working on a function, look at your local parameters only and use that information to do what work you need to do. Forget the rest of the program. Don't look at how other code is calling the function. Likewise, when you are using a function, just be sure that you know what it does, and that you are passing the correct data to it. Don't worry about what exactly is going on (or what the parameters and variables are called) inside the function. For example, you don't know what happens inside the line command but you use it all the time! Whoever wrote the `line` command, implemented it without knowing (or caring…) about exactly how you will use it!

14.8 **Functions Calling other Functions: Palindrome Tester**

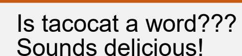
Any function can call any other function, this is not just restricted to the `draw` block. In fact, your `draw` block is just a function that processing is calling repeatedly. Inside the `draw` function you can call your other functions. Inside your other functions, you can call even more! This can go on and on (and does!). In fact, a function can call itself (or function A can call function B, which then calls function A), a technique called recursion. This is a hard topic, however, and one beyond the scope of this course.



Advanced: When a function calls another, it saves all the important information – where it came from and where to go back to, as well as the parameters for the function, on a data structure called a stack. You can imagine the stack like a stack of old books, and each time a function is called, that information gets put on the stack; another book gets put on the pile. When the function is done and returns, the last item on the stack is removed, telling the program where to go back to. If you use recursion, and a function calls itself, and it loops too deeply, your stack gets too large (all those memories of where to go back to), and you get an error called *stack overflow*. It can be challenging to avoid those, so we'll look at recursion in a different course.

Let's work through an example with nested function calls: a program to test if a string is a palindrome.

A palindrome is a word that, if you reverse it, gives you the same word. For example "`tacocat`" reversed gives you "`tacocat`". To test if a word is a palindrome, what we need to do is to first reverse the word, and then compare it against the original.



In this example, we will take a top-down approach to developing our solution. That is, we will work from the `draw`

 © James Young, 2016

loop first, and create functions as we go.

The particular technique employed in this example is very common in programming. Sometimes, you want a task done, but haven't quite figure out how to do it yet. So, what you do is to offload the problem until later: create a fake function and pretend it does what you want, and use it in place to solve your current problem. Then, you come back and implement that function later. This helps you focus on one problem at a time and minimize how much you get overwhelmed. To make this concrete, let's make our `draw` block call some new function `isPalindrome` to see if some text is a palindrome:

```
void draw()
{
  background(0);
  String test = "racecar";
  if (isPalindrome(test))
  {
    text(test+" is a palindrome", 10, 250);
  }
}
```

At this point, `isPalindrome` doesn't exist – we just made it up. In this case, you can see that the function takes a `String` parameter, and returns a `boolean` result – true if the string is a palindrome, and false if not. We created this fake function because it lets us focus on our `draw` block – we test some string, and if it's a palindrome, put some text on the screen. Otherwise, do nothing. Now we can leave the `draw` block alone and move on to the next problem.

Because we made this fake function, this won't compile yet. We now need to implement `isPalindrome`. First, put in a ***stub*** or ***skeleton*** of the function, to make your program run:

```
boolean isPalindrome(String test)
{
  return false;
}
```

We do no work and return garbage data, just to make the program run. It should run – but not work – now.

Once you write up the function header, you should forget about how the function is used; all you need to know is that it takes a string, and returns true if it's a palindrome, and false otherwise. This is an example of the tunnel vision discussed earlier.

To test if a string is a palindrome, we first reverse it, and then compare to the original. This is two separate problems, so let's focus on one at a time. Let's assume we have some function to reverse a string (let's make one up: `reverseString`) that takes a `String` parameter and returns the reversed string. If we have that, then palindrome checking is easier:

```
boolean isPalindrome(String test)
{
  String reverse = reverseString(test);
  boolean match = (reverse.equals(test));
  return match;
}
```

Here, we reverse the string, and compare it to the original, returning the result. Again, create a stub for your new `reverseString` function to get your program to run. It still won't work correctly, however.

```
String reverseString(String original)
{
  return "";
}
```

Again, we return garbage to just get it run. Here I randomly chose the empty string.

Reversing a word is a little tricky. The way we do it is to use a for loop to pick out each character one by one, and then put the string back together in the opposite order.

We can make a `for` loop to go through this string character by character. However, where do we put the new characters? We need a new string to put them into. Let's call this `reverse`. Then, the `for` loop goes through each character, and adds them to `reverse`.

```
String reverse;
for (int i = 0; i < original.length(); i++)
{
  char c = original.charAt(i); // grab char i from original
  reverse = reverse + c; // toss it onto the reverse
}
```

There are two problems in this code. Can you spot them?

The first problem is that the variable `reverse` needs an initial value. In our code,

 © James Young, 2016

the first time we use `reverse`, we *add* `c` onto it, but since we never originally set it to a value, Processing gets confused – we can't add to it if the value is not yet defined. What would be a reasonable initial value? We can set this to the empty string when we create it.

```
String reverse = "";
```

The other problem is not so obvious. When we write code like this, it is always a good idea to think about how you can test if it's working properly. In this case, let's toss the strings to console to see if the reversing works. After the for loop:

```
println(original);
println(reverse);
```

If you run this code, you can see that the two strings printed are the same! The reverse didn't work.

It can be confusing to see why this is, so we need a debugging strategy here. Inside the for loop, let's insert a `println` statement to show us what is happening to the `reverse` variable as it gets created:

```
println(reverse);
```

```
n
no
not
notA
notAP
notAPa
notAPal
notAPali
notAPalin
notAPalind
notAPalindr
notAPalindro
```

If you run this now, and look at the console, you can see that the reverse actually just becomes a copy of the original string. The reason for this is because we are taking the characters from left to right from the original string, and adding them on the *end* of the new string. To reverse it, we instead need to add the characters onto the beginning:

```
reverse = c + reverse;
```

Finish up by making sure to return the appropriate value from your function, and now your program should work as expected! Try different strings, and test if they are palindromes.

Great! We created functions that call other functions. We kept tunnel vision and implemented only one section at a time, simplifying our mental load. The following flow chart shows how the functions called each other.

 © James Young, 2016

14.9 **Best Practices – global and local variables**

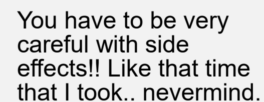
The way we use Processing so far brings up an issue when using functions: we have both global and local variables.

You should avoid using global variables in a function as much as possible, except for final constants. If you start modifying global variables in a function it becomes hard to keep track of and it doesn't scale well to larger programs. For example, you may forget that you modify the global in one function, and this can make another function work incorrectly.

You are much better off thinking of functions as stand-alone as often as possible. This will help you better get a grasp of how functions are properly used, and how data is sent to and returned from them.

If there is some changing, specific data (like, what color to use, where to draw a bad guy, etc.) that can vary whenever the function is called, then pass that information to the function as a parameter. A function should have all the required information to do its job from its parameters as much as possible, and from any global constants (finals).

Functions likewise should not change any global information and should return the results through the return mechanism. If a function changes other values this is called a ***side effect*** of the function, and is generally considered to be higher risk. The problem is that when you use a function with side effects, they may not be obvious and you may have forgotten about them, which leads to bugs. Hilarity ensures (or rather, hours of banging your head against the wall debugging).



This relates to the idea of keeping tunnel vision. As much as possible, when implementing functions, try to make it independent of the rest of the program. If you can avoid using globals – and use parameters instead – then do it, as it makes it easier to implement. Keeping an entire program in your head at once is impossible so we need techniques to simplify the problem into smaller chunks. If we have a good function, then implementing the function lets us solve one simple problem while ignoring the rest of the program. For example, given:

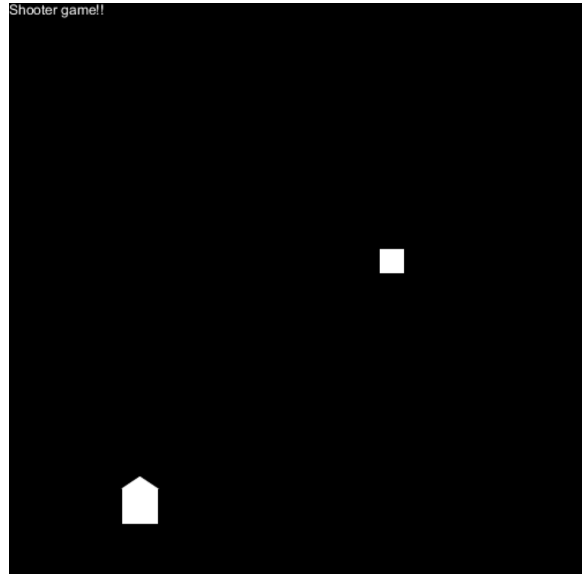
```
double addTax(double price) {
      //…
      return?
}
```

You know you have a `double` value (price) and need to return a `double`, all you need to do is think about how to add the tax. You don't need to know how this function

 © James Young, 2016

will be used.

14.10 **Example: Top-Down Programming for a Space Shooter**



Let's make a shooter video game with one bad guy. The space ship is linked to the `mouseX` (so that it moves left and right but not up and down). If the mouse button is pressed, the ship fires a bullet, but only one bullet is out at a time. The bullet goes up the screen until it either hits the bad guy or goes off the edge of the screen. If the user clicks while a bullet is flying, nothing happens.

In this example, let's try to make our functions completely stand-alone, and not use any globals at all – everything that the functions need should come in as parameters. If you do the related exercise at the end of the chapter, you will really see how this is a good idea.

We will approach this with four steps:

1. Write the program in `draw` as a series of steps in comments, in English.
2. Create fake functions, and use them in `draw` to get our work done
3. Create empty functions, stubs
4. Start implementing the functions

Using to-down programming, first plan the steps of the program in English and make sure it makes sense on that level.

- draw a title at the top of the screen
- move and Draw the player's ship
- check if a bullet should be shot, and if so, shoot it
- move and Draw the bullet
- move and Draw the bad guy
- check if the bullet hit the bad guy, and if so, stop the game
- draw "WIN" if the game is over.

Why is it bad "guy", isn't that sexist??? I'm a guy, and I'm not bad!!

Whew! Did I miss anything? **Point:** We are thinking through our program without doing any computer programming at all. We are just planning things out.

 © James Young, 2016

If we want to avoid using globals in our user-defined functions, then we need to keep all the game logic in the draw loop, since this is the only place we try to modify globals.

Type the above comments into your draw loop. At this point, my program looks like this:

```
// Game details
final int BG_CLR = 0;

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

void draw()
{
  background(BG_CLR);
  //draw a title at the top of the screen
  //move and Draw the player’s ship
  //check if a bullet should be shot, and if so, shoot it
  //move and Draw the bullet
  //move and Draw the bad guy
  //check if the bullet hit the bad guy, and if so, stop the
     // game
  //draw "WIN" if the game is over.
}
```

Now we start trying to turn our comments into computer code. I like to start with the easy stuff, which in this case, is drawing a few strings on screen: drawing the title and the win message (if the game is over). At this point, I don’t want to worry about how I’m going to do it, so I’m going to make a new function that I’ll manage later, to help me. I’ll make one called `drawMessage` that takes a message, a location, and a color. This enables me to forget about those details for later and think about my `draw` logic. Drawing the title is easy. I’ll also make up some globals (give them reasonable values):

```
// draw title
drawMessage(TITLE, TITLE_X, TITLE_Y, TITLE_CLR);
```

 © James Young, 2016

The second draw is a little trickier, since I only draw the end-game message if the game is over. I'll create a boolean to tell me if the bad guy is dead, and only draw if that's true. Now my draw block looks as follows. Make sure to create the proper global variable and the constants:

```
void draw()
{
 background(BG_CLR);
 // draw title
  drawMessage(TITLE, TITLE_X, TITLE_Y, TITLE_CLR);

  //move and Draw the player's ship
  //check if a bullet should be shot, and if so, shoot it
  //move and Draw the bullet
  //move and Draw the bad guy
  //check if the bullet hit the bdad guy, and if so, stop the
  //game

  //draw "WIN" if the game is over.
  if (badGuyIsDead)
    drawMessage(WIN, WIN_X, WIN_Y, WIN_CLR);
}
```

Now, create the stub for drawMessage, and make sure the program runs, even though nothing will happen.

At this point, there are two ways we can go: we can implement drawMessage now and see if it works, or, we can keep working on draw to see if we can figure out the other pieces. I'm going to keep working on draw.

Now, let's move down the list. Moving the player is easy – the y coordinate is fixed at the bottom of the screen, and the x coordinate is linked to the mouse. Make a global constant for the y location, and a variable for x, and set x to the mouseX in the draw block. Drawing is a little more annoying, because of that funny shape (see the picture), so let's make a fake function and handle it later. To draw the ship, we need the ship's location, size, size of the nose (see the pic), and the color, so we can make a function that takes those things. Here is my code:

```
// move and draw player ship
playerX = mouseX;
drawPlayerShip(playerX, PLAYER_Y,
               PLAYER_SIZE, PLAYER_NOSE, PLAYER_CLR);
```

 © James Young, 2016

Also, make the stub for `drawPlayerShip` so that the program runs.

The next step is tricky. We first need global variables to keep track of the bullet's location. We also need constants for the bullet color, and how fast the bullet should move. Finally, we need to keep track of whether a bullet is currently being shot or not – remember, we can only have one at a time – so we can use a boolean for this. Once we have variables setup, we can start code.

At this point, thinking of whether a bullet should shoot takes some boolean logic, and it's a little confusing, so let's make a new function called `checkShoot` that gives us a boolean to tell us if we fire a new bullet. We just tell it if a bullet is already moving or not. If we should shoot, then we move the bullet to the player's location, and set our "moving" flag to true. This is a little tricky, so try thinking through it. Here is my code:

```
// check if we should fire a bullet, and if so, shoot it
if (checkShoot(playerBulletMoving))
{
  playerBulletMoving = true; // it's alive!!
  playerBulletX = playerX;
  playerBulletY = PLAYER_Y;
}
```

Also, implement the stub for `checkShoot` – it takes and gives a boolean. For now, return a garbage value so that the program runs.

Moving forward, now we move and draw the bullet. Again, create new functions to avoid thinking about the details, and instead focus on the high level game logic. Here, if we have a moving bullet, then we call `moveBullet` with to move it: give it the current `y` and the speed, and it gives us the new `y` value back. After the bullet is moved, we should also check to see if it ran off the top of the screen – if so, kill it (set our flag to false). Finally to draw the bullet, let's make a function called `drawBullet` that takes the position and color, and draws it for us.

```
//move and Draw the bullet
if (playerBulletMoving)
{
  playerBulletY = moveBullet(playerBulletY, -BULLET_SPEED);
  if (playerBulletY < 0) // end of screen
      playerBulletMoving = false;
}
drawBullet(playerBulletX, playerBulletY, BULLET_CLR);
```

 © James Young, 2016

We do something similar for moving and drawing the bad guy. You have some flexibility here, as to how you want it to move. I'm going to use the random move from our earlier example in this chapter. Think through what variables and globals you need. Also, as we already have a flag telling us if the bad guy is dead, we should check that and not move a dead bad guy!

```
//move and Draw the bad guy
if (!badGuyIsDead)
{
  badGuyX = doMove(badGuyX, BAD_GUY_MOVE, 0, width-1);
  badGuyY = doMove(badGuyY, BAD_GUY_MOVE, 0, height-1);
}
drawBadGuy(badGuyX, badGuyY, BAD_GUY_SIZE, BAD_GUY_CLR);
```

We only have one piece left – check if the bullet hit the bad guy. Again, let's offload the hard stuff (checking if the bullet hit the bad guy!) into a function. I created `checkHit` that takes the bad guy location and size (as its square) and the bullet location. It returns true if the bullet is inside the bad guy, and false otherwise. If true, we set our dead flag to true.

```
//check if the bullet hit the bdad guy, and if so, stop th
//game
if (checkHit(badGuyX, badGuyY, BAD_GUY_SIZE,
    playerBulletX, playerBulletY))
    badGuyIsDead = true;
```

Whew! We turned that English into high-level game logic. We thought about what work needs to be done – what needs to be drawn, what conditions checked – and what logic we need to make our game happen – if a bullet is flying, if the bad guy is dead – and made a good first pass on the high level game logic. By using user-defined functions, we were able to leave some of the annoying logic (like drawing the player's ship) behind to solve later, and focus on the game problem.

Before going forward, make sure that your program runs – nothing will happen, but it shouldn't raise errors either. This is important, because you want to be able to test pieces as you build them.

I will give much less detail on implementing the actual functions, as you should work on this on your own. The `drawMessage` function should be straight forward, and the other draw functions, `drawPlayerShip`, `drawBullet`, and `drawBadGuy` just take a little bit of tricky geometry at worst. `checkShoot` is simple boolean logic (returns true if the mouse is pressed and we don't already have a bullet), `doMove` is based off our earlier randomly moving square example, and `checkHit` we saw in the

 © James Young, 2016

boolean chapter: test if a point is inside a box.

Your strategy here should be piece wise: implement a function, test the behavior, and move on. You will surely encounter bugs, but try to fix whatever you can before moving on. An easy way to do it is to implement all the drawing functions first (so you can see the player), then the enemy move (so you can see him moving around), and then the bullet firing. Finally, test if the bullet hits the enemy.

14.11 **Example: Dice Game**

Let's do another quick example, this time with less hand holding. We will make a gambling dice game simulator to test out how much you may win in a game. Here are the pieces

- Throw a six sided dice
- Update your bank balance based on the roll
  - 1 -> you win 10% of your money (multiply by 1.1)
  - 2 -> you win 20% of your money (multiply by 1.2)
  - 3 -> you win 30% of your money (multiply by 1.3)
  - 4 -> you win 40% of your money (multiply by 1.4)
  - 5 or 6 -> you lose 35% of your money (multiply by .65)
- Keep playing until you either fall below $2 or go above $1000. Start with $10.
- Play one round per `draw` loop.

Try to spend some time thinking through how this should work as a program. As a starting point, here is my English version.

- If we are still playing
  - Throw a die
  - Update our balance based on the result
  - Check if we should stop
- Display statistics (how many rolls, and our current balance)

Let's first put a skeleton of the draw and then consider what data needs to pass around

```
void draw()
{
  background(BG_CLR);
  if (!finished)
  {
    throwDice( ??? );
    updateBalance( ???);
    shouldStop( ???);
  }
```

 © James Young, 2016

```
  displayStats( ???);
}
```

The throw dice function probably doesn't need any data, as the number of dice sides, etc., should be fixed as a global final. However, it does give us an integer (a roll) that we need to store.

```
int roll = throwDice();
```

To update the balance, we need to tell the function our current balance, AND, what the roll was, so it needs two pieces of data. Further, it will pass us back the updated balance, so we should store that as our new balance.

```
balance = updateBalance(balance, roll);
```

In order to determine if we should stop or not, we need only to check the balance. Also, we need to store the result. We already have the boolean `finished` – let's update that based on the result.

```
finished = shouldStop(balance);
```

Finally, we need to display the statistics. It will need our current balance. Also, it will need to know how many rolls we simulated, so we need a new counter variable that updates every time we do a roll. Be careful – make sure to update this inside the `if` statement (if not finished), since once done, we will still display stats but not do more rolls.

```
displayStats(balance, rolls);
```

Here is my final draw block:

```
void draw()
{
  background(BG_CLR);
  if (!finished)
  {
    int roll = throwDice();
    balance = updateBalance(balance, roll);
    finished = shouldStop(balance);
    rolls++;
  }
  displayStats(balance, rolls);
```

 © James Young, 2016

```
}
```

Now we can implement the functions based on the headers created here. These are pretty straight forward, so try doing it yourself.

Check your Understanding

14.12 Check Your Understanding: Exercises

Exercise 1. Create a function that helps calculate compound interest: it takes three parameters, `principle`, `rate`, and `years`. Use floating point numbers, but remember that this would not be acceptable for real money. Calculate the final amount by adding the rate to the principle once per year, for the given number of years. You can use a loop for this, or look up the `pow` command for a short cut. Return the amount as a floating point.

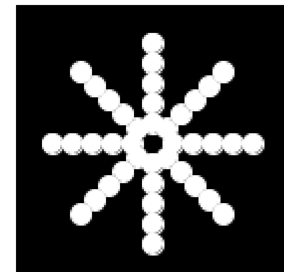
Exercise 2. Make a program that has two user defined functions: `roundUp` and `roundDown`. These take a single floating point number and return an integer. To round, try to do it yourself with logic and a cast, or, look up the ceiling and floor functions.

a. Make a new function called `roundEven` that takes a floating point number as a parameter, calls `roundUp` and `roundDown`, and then returns whichever integer result is even. In the case where neither is even, return the next even number larger than the one given.

Exercise 3. Update the shooter game example (14.10) to have three bad guys! This should really highlight why having stand-alone functions that do not modify globals is a good idea.

Exercise 4. In this exercise, you will draw a "waving octopus" graphic at the mouse location.

a. Draw `NUM_ARMS` "arms", each consisting of a line of `CIRCLES_PER_ARM` small circles, with diameter `CIRCLE_SIZE`. The values 8, 5, and 10 are used in the image on the right. Make the arms stick straight out from the mouse position, not curved as they are shown here. The center of each circle in an arm should be a distance of `CIRCLE_SPACING` from the centers of the adjacent circles in the same arm (or from the mouse position, in the case of the closest circle). Do this by writing the following two small functions, exactly as



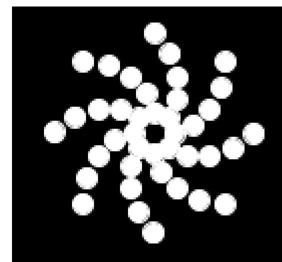
specified:

`void drawCircle(float theta, float distance, float xc, float yc)` – this function draws one circle diameter `CIRCLE_SIZE` at `distance` away from the point `(xc,yc)`, in the `theta` direction. For example, `drawCircle(PI/2, 100, 50, 50)` should draw a circle centered at `(50,150)` – straight down 100 pixels from `(50,50)`.

`void drawOctopus(float x, float y)` – use two nested `for` loops to draw an “octopus” with its center at `(x,y)` using the global constants. One loop goes through each arm, and the other through the ring. Use `drawCircle` to draw all circles. The arms should stick out at evenly-spaced angles in a circle around `(x,y)`.

Call `drawOctopus` from `draw()` to draw an “octopus” at the mouse location. You should see a kind of star, with arms straight out, from the mouse.

b. Make sure the above works before attempting this one. Make the arms “wave” by using a slightly different angle (theta) for each of the circles that make up an “arm”, so that the arm curves, instead of sticking straight out. Change the `drawOctopus` function to be `void drawOctopus(float warp, float x, float y)`. Add `warp` to the angle of the first (closest) circle in an arm, `2*warp` to the angle for the second one, `3*warp` to the angle for the third one, etc. If `warp=0` then the arms will be straight, as before, but larger or smaller values for warp will make the arms curve to the right or left. (They will no longer be exactly distance apart, but that’s OK.)



Change the `draw()` function to use a different warp value from one frame to the next. Start with 0, and every frame, a value `warpSpeed` should be added to the amount of warp, until it reaches `WARP_LIMIT`. Then the warp value should start to decrease, going back to 0 and continuing down to `-WARP_LIMIT`, at which point it should start to increase again, repeating the cycle forever. Hint: Note that `warpSpeed` is a variable, not a constant, so that its value can change from positive to negative and back again.

Exercise 5. We humans are really bad at understanding probabilities, odds, and how they scale up with populations. We all have some anecdotal story of something amazing that happened to us or a relative, or some magical cure for ailments (Last time I had a cold, I drank ginger tea, and I got better right away!! Ginger tea must cure colds!).

Let’s do a simulation of a population of people rolling dice. We’ll start with a *2* sided dice to simulate a coin toss. If I asked you how likely you are to get, say, 10 heads in a row, you know that it’s very unlikely. However, if only a

 © James Young, 2016

thousand people around the world do it, someone will likely do it just by random chance.

```
Round: 1, 502 people left.
Round: 2, 246 people left.
Round: 3, 126 people left.
Round: 4, 58 people left.
Round: 5, 29 people left.
Round: 6, 17 people left.
Round: 7, 6 people left.
Round: 8, 3 people left.
Round: 9, 2 people left.
Round: 10, 1 people left.
```

Imagine we lined up 1000 people and asked them all to toss a coin. If you get heads, stay, tails leave. On the first toss, we expect about 500 people left. Do it again, on the second toss, about 250 people left. And so on. Eventually, there will be 1 or 2 people left, and those people will feel amazing- they tossed so many heads in a row!

First, let's setup the program. One nice trick is that we can put the command `noLoop()` in the setup block, and then the draw will only execute once:

Our `draw` block will do the following

- Start with an initial number of people (say, 1000)
- While there is still more than 1 person left
  - Call the `doARound` function with the number of people you have, and save its return as the number of people left.
  - Increase the number of rounds you ran by 1.
  - Call the `printUpdate` with your people count and rounds to display the status

That's it! Your `draw` block will loop while people are left, and, count how many rounds it took.

You need to implement the following methods:

`int diceRoll(int sides)` – takes the number of dice sides as a parameter, and returns a random dice roll, where 1 <= roll <= sides.

`void printUpdate(int people, rounds)` – uses `println` to give the current status, as in the image.

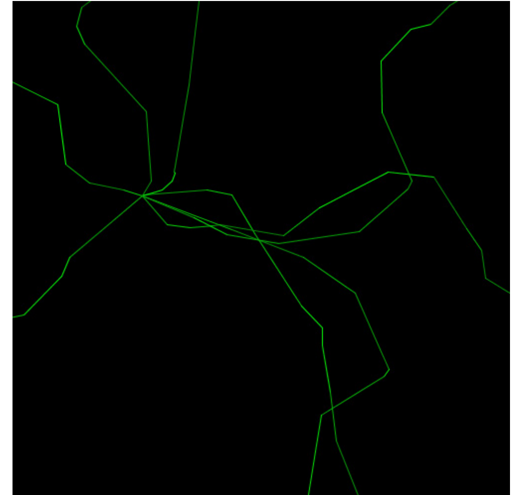
`int doARound(int people)` – takes the number of people left, and simulates a round of dice rolling:

- Use a variable to remember how many people to cut from the pool
- Use a for loop to go through each person
  - Roll a 2 sided dice
  - If the dice number is NOT 1, cut the person.

- Return the new number of people – old number minus the cut number.

This will tell you how many rounds you get until you have 1 or fewer people left. In other words, how many rounds did some people last, getting a 1 (say, a heads) each time? Try changing the parameters – change the 2-sided dice (a coin) to a 6 sided dice. Try changing the number of people. Does your intuition work?

Exercise 6. For this lab you make lightning come out of your mouse!! (just like the Emperor in Star Wars). You will need functions – to make things simpler – and a while loop. The while loop is useful because, since each stroke of lightning is random, you don't know how long it'll take to get to the edge of the screen.



You will use the following global variables: `FLASHES` (how many strings of lightning, the picture has 8), `MAX_PIECE_SIZE` (the longest lightning segment before it shifts direction again, `MAX_SHIFT` (the maximum angle it changes direction on each segment).

Your draw loop uses a for loop to call your function `drawLightning` `FLASHES` number of times. Each time, you call `drawLighning` with the mouse position.This way, you'll get `FLASHES` number of lightning streaks from the mouse.

Implement the following functions:

`shiftAngle` – takes an angle, and the maximum shift amount as parameters. Returns the new, shifted angle. Use `MAX_SHIFT` to generate a random shift, in either the clockwise or countreclockwise direction, and add it to angle. Return the new angle.

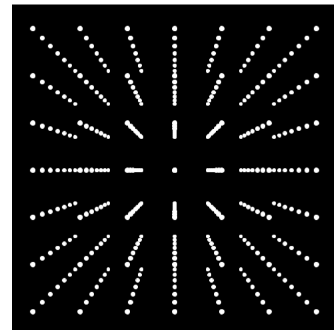
`onScreen` – takes an `x` and a `y`, and returns true of that coordinate is on the screen, false otherwise.

`drawLightning` – takes a start position and draws a stroke of lightning to the edge of the screen.

- Start at a random angle
- While the `x,y` is on the screen (call `onScreen`)
  - Calculate a random lightning segment size within the valid range
  - Use `cos` and `sin` with your angle and new size to determine the end point of the segment. Remember to add the current position to the result.

 © James Young, 2016

- ➢ Draw a line from `x,y` to the `endpointX` and `endpointY` (use a random green color if you want! ☺)
- ➢ Set your new `x` and `y` as the end point you calculated, for use next time
- ➢ Update the angle using shiftAngle.

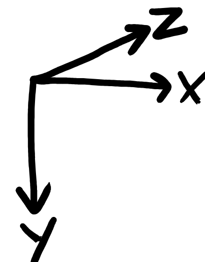


Exercise 7. 3D! Current computer screens are not capable of doing real 3D – they're flat! Anything "3D" you see on a computer screen is an illusion – completely faked. There are a lot of great ways to fake 3D, and to trick people's eyes into seeing 3D. One of the most fundamental ways is by simulating perspective:

- Objects that are further away appear to be smaller
- Objects that move further away appear to converge.

Look at the railroad image. Because of perspective, the tracks appear to get closer together as they get further away. The point where they appear to meet each other in the distance is the "vanishing point". Also, the trees in the distance are not much bigger in the picture than a single railroad tie in the foreground. Our brains pick up on these cues and we know that this is actually a 3D scene because we know that the tree is bigger than the tie.

We will draw a 3D grid of "spheres" (circles when drawn in 2D), as shown in the image.



We have used "flat" 2D `(x,y)` coordinates a lot. To use 3D, we need to add a z coordinate to get `(x,y,z)`. Look at the diagram to the right. Our usual `X` axis goes to the right across your screen, and the `Y` axis goes downward on your screen. The new `Z` axis goes straight into your screen, away from you.

These are virtual coordinates – they're not measured in pixels, and our screen does not really have a `z` axis, so what we need to do is **project** our virtual 3D points `(x,y,z)` onto normal 2D canvas points `(px,py)`. ("`px`" means "projected x"). Note: Processing has a built-in way to handle 3D points, but we won't use that here. You can investigate Processing's 3D capabilities on your own, if you're interested.

 © James Young, 2016

The basic idea behind perspective projection is to divide the `x` and `y` coordinates by the `z` coordinate to get `px` and `py`. That way, the larger the `z` coordinate is, the more `px` and `py` will decrease, and appear to move toward the "vanishing point" (0,0) – as the railroad tracks did.

But if we just divide by z itself, our `px` and `py` coordinates will get *very* small *very* quickly and you'll only see a cluster of dots at the center of the screen. Use the constant `PERSPECTIVE` (set at 0.002) to control the amount of perspective (how fast `px` and `py` converge as z gets larger). Divide `x` and `y` by `PERSPECTIVE*z` instead of just `z` to get `px` and `py`. Object sizes (such as the diameter of a circle or sphere) also must be divided by `PERSPECTIVE*z` in the same way, since objects also must appear smaller when they are farther away.

Note that perspective can make things shrink, as well as get bigger as they get closer (because `PERSPECTIVE` is a very small number). Our illusion works best when we look at objects far away "into the screen", not directly in front of our noses, and so we'll want `z` values that are large compared to x and y.

a. Write a function `void drawProjectedCircle(float x, float y, float z, float diam)` which draws a "sphere" (a circle, really) with a center at the 3D point `(x,y,z)`, and a diameter of `diam`. Use the projection method described above to modify `x, y`, and `diam` before drawing the circle.

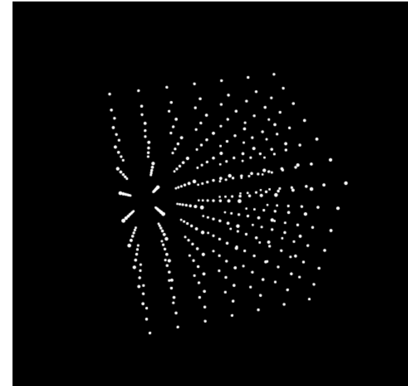
   There is one problem: if you happen to get `z=0`, you will divide by 0 (things are so close that they appear to be infinitely large – they're right in the middle of your eyeball! And z<0 is even worse – beyond infinity?) You need what is called a clipping plane in graphics. Your `drawProjectedCircle` function shouldn't attempt to draw (or even do the calculations) for any point unless `z>0`!

b. Write a function `void drawDotGrid(int minValue, int maxValue, int spacing, float diam)`, to draw a three-dimensional grid of circles. We usually use two nested for loops to generate 2D grid, but here, use *three* nested **`for`** loops to generate iterate over a range of `(x,y,z)` points in a grid, from `minValue` and `maxValue`, spaced `spacing` units apart. For example, `drawDotGrid(-30,30,10,1)` should draw circles of diameter one on a grid spaced 10 apart, from -30 to +30 (a total of 7*7*7 = 343 points). Use `drawProjectedCircle` to do all of the drawing. Add `mouseX` do the `z` coordinate so that the mouse can be used to zoom in.

c. Once you get it working, add an offset so that the "vanishing point" `(0,0)` is moved to the center of the canvas.

 © James Young, 2016

Exercise 8. Update exercise 7 to have 3D rotations controlled by the mouse – now you're playing with power!



First, add a constant "zoom factor" of `150` to all z co-ordinates, instead of `mouseX` as was done above. We'll need the mouse to do other things, and `150` will "push" the cube just far enough "into the screen" to give it a nice size. You could experiment with other values, too. (You *were* going to use a named constant, right?)

To rotate a 2D point `(x,y)` an angle of θ radians around the center of rotation `(0,0)`, giving a new point `(rx,ry)`, the math is not very complicated (`rx` is short for "rotated x"):

$$rx = xcos(\theta) + ysin(\theta)$$

$$ry = -xsin(\theta) + ycos(\theta)$$

To rotate a 3D point `(x,y,z)` around the `X` axis (think of grabbing the `X` axis and spinning it), just leave the `x` coordinate unchanged, and use the above formulae with `y` and `z` instead of `x` and `y`. Similarly, to rotate the point around the Y axis, use the formulae with `x` and `z`. Using the formulae as they are above will spin around the Z axis. Try that first.

Write a function `void rotate(float theta, float a, float b)` which will rotate a point `(a,b)` around the point `(0,0)` by an angle of `theta` radians, using the formulae above, giving a new point `(newA,newB)`. This function could be used with `x` and `y`, or `x` and `z`, or `y` and `z`, so let's just call them `a` and `b` to make them generic.

Now we run into a problem with returning results from functions – only one value can be returned, and we need to return two (`newA` and `newB`). There is a way to solve this, but it's not covered in COMP 1010, so create two global variables `newA` and `newB` and put the answers there. It's not the "proper" way to do it, but it will work.

Now modify the `drawDotGrid` function to use the mouse to spin the cube. Before drawing each circle, use your `rotate` function to spin it by `mouseX/100.0` radians around the Y axis, and then `(height-mouseY)/100.0` radians around the X axis. (If this seems backwards, just think about it and visualize it – it's correct.) The spinning must be done *before* you add the "zoom" factor of 150 to the z co-ordinate.

You should now be able to "grab" the cube and spin it with the mouse.

 © James Young, 2016

Exercise 9. This is a hard one that uses concepts at the edge of functions that you officially cover in the next course. This uses **recursion** – functions that call themselves. Yep, that's right! Like those Russian dolls with one inside another, a function can call itself, which calls itself, which calls itself, and so on – but when does it stop?? That's the trick- the nested function calls need to stop at some point.

You'll make some code to generate a random tree like in the image to the right. The way you do this is with the following algorithm:

- Draw a branch. The first branch starts at the bottom of the screen and is straight up
  - Calculate the end point of the branch by using the starting point, the angle of the branch, and the length of the branch.
- From the end of the branch, you need to draw 3 new branches. Each branch should be a random shift from the existing branch angle. For example, one is slightly to the left, one is quite to the right, and one is straight up. Also, the new branch should be smaller than the old one, so they get smaller.
- Continue to draw three new branches at the end of each new branch, until the branch hits the minimum size. If it's the minimum size, draw a leaf (circle) at the end of it.

This is recursive. At level 1, you have 1 branch, the trunk. Then it splits in three, and you have three branch. Each new branch splits in three, so you have 3*3 branches. At each level, you multiply the number of branches by three. At the end, you have a lot of branches!

To let you focus on the functions and not everything else, I'll give you the globals from my solution:

```
final float MIN_BRANCH_SIZE = 5;
final float START_SIZE = 100;
final float START_ANGLE = 3*PI/2;
final float BRANCH_SHRINK_RATE = 0.7;
final float SPREAD = 1.7; // radians
final int BRANCHES = 3;
final float TREE_START_X = CANVAS_SIZE/2;
final float TREE_START_Y = CANVAS_SIZE-1;
```

The starting branch (the trunk) is `100` pixels. We stop building the tree when our branches get below the minimum size (`5`). Our first branch angle is `2PI/3`, which is straight up. Each time we generate a new branch, it is 70% the length of the

 © James Young, 2016

prior branch. The new branches can spread within a 1.7 radians fan out from the existing angle. We make three new branches at the end of each branch.

The tricky part is turning this into a function. In this case, we only draw it once, so add noLoop() to the setup block so draw only runs once.

```
void draw()
{
  background(BG_CLR);
  drawBranch(TREE_START_X, TREE_START_Y,
             START_ANGLE, START_SIZE);
}
```

The logic for the drawBranch function, which takes four parameters: x, y, angle, and length, as above:

- If the length is too small
  - Draw an ellipse of size length/2 at x,y, and return.
- Draw the line
  - Use the x,y, length, and angle to calculate the end point X,Y of the line
  - Draw the line
- Draw the branches
  - Use a for loop to go from 0<=i<BRANCHES
    - For each branch, calculate the new angle (random(SPREAD) – SPREAD/2) and add to the existing angle.
    - Call drawBranch with the new x,y (the end point of the above line), the new angle, and a new length (old length multiplied by BRANCH_SHRINK_RATE).

That's it! To make it green, I set the color to a random green variant for each line and leaf. This is a pretty short program.

Try playing with the parameters, the spread, shrink rate, etc., to see how the tree looks different.

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)

 © James Young, 2016