# UNIT 6. USER-DEFINED FUNCTIONS PART 1

**Summary**
Here we learn a generalized extension of making your own named code blocks, like the `draw` and `setup` block. This is not actually a hard unit, but is a little lengthy because of the code examples. You will learn to make your own commands. In this section, you will…

- Lean about user-defined functions and how they form part of a program.
- Learn how user-defined functions help you make larger programs that are more manageable.
- See how user-defined functions complicate variable scope rules.
- Learn about how user-defined functions can call each other, to further simplify your programs.
- Learn a technique called top-down programming to help you attack difficult problems

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

- Create your own user-defined functions.
- Use your custom functions within other functions.
- Use your custom functions to avoid repeating code.
- Apply the top-down technique to solve problems.

**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.

## 6.1 Introduction


As your programs get larger and larger, and more complex, you (very!) soon reach the limit of the human brain to keep it all organized and effective. While some people believe that they are good enough to keep their programs in one big long block (like ours so far), there is a lot of research that says they're wrong, and highlights the benefits of taking a more structured approach to programming. In addition, you will find yourself repeating the same code over and over again. We need to learn new techniques that help us manage our larger programs, organize them, and re-use code whenever possible to save work. Enter user-defined functions! To be clear, you are the user here, who will be defining your own functions.

User-defined functions are a way to create your own commands. So far, you have used many commands like `rect`, `ellipse`, and so on. Actually, in the last unit you learned how to make functions when you made your draw and setup block. User-defined functions is just a simple but powerful extension of what you already learned.

**Advanced:** You may hear about functions, procedures, subroutines, or methods, which all refer to a very similar concept. To make things more confusing, there isn't always perfect agreement on what they mean. The most important difference is between methods and functions, procedures, or subroutines. Methods are specifically functions that are linked to an object, which ends up giving it special properties. Other things you may hear include functions *returning* (providing) data and procedures not *returning* data. At this point, don't worry about these subtle differences.


One of the largest programs we have actually written so far is our good old cat face. Are you tired of it yet? Don't worry, we're almost done with it. If we look at the cat face example, not only do we notice that the `draw` block gets really long, but also that it breaks down nicely into clear units of work. These units include drawing a head, eyes, ears, etc.

Here is the whole code again.

```
/*******************
 * 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);
}
```

Now, let's imagine that we had new commands for all the components of the cat face. We could really simplify the `draw` block if we used these commands. To be clear, at this point I am just making up hypothetical command names – this code will not run.

```
void draw()
{
 background(0);
 // copy the mouse position into the nose position
 noseCenterX = mouseX;
 noseCenterY = mouseY;
 drawCatHead();
 drawCatEars();
 drawCatEyes();
 drawCatWhiskers();
 // draw the nose after whiskers for nice overlap effect
 drawCatNose();
}
```

If this worked it would have a great deal of benefit. One, it enables us to see an

overview of the draw block without scrolling through pages. Two, we have a lot fewer comments! Since we have the command names, it is obvious what we are doing. The only comment left explains the nose / whiskers draw order. At this point, we assume that the code that actually does the work – the lines, ellipses, etc. is somewhere else.

But how can we create such commands? We need to learn the new syntax!

## 6.2 Basic Functions: Syntax

Luckily for us, you have already learned the syntax to create a new function – you did this for the `draw` and `setup` blocks.

To create a new function (a new command) we need two basic things:

- function name: they keyword used to invoke (use) the function. E.g., `ellipse` is a function name
- some code: the processing code to run every time the function is invoked (called).

With this in hand, we can create a new function using the following syntax:

```
void functionName()
{
   …//code;
}
```

The `void` keyword means that the function does not create any data result, and the empty brackets () means the function does not require any data to run. We will learn more about that later in the course.

Look familiar? See? We already learned the syntax. The main difference between our existing draw and setup commands, and these new user-defined functions, is that draw and setup already have defined roles, and Processing knows when to use them. Your new commands will not be used at all until you do it yourself!

The first line, which currently starts with `void`, is called the ***function header***. The function block is called the ***function body***.

***You must put this code outside of your startup and draw blocks, along with your globals***. You can place them before or after those blocks, it's up to you and ends up being a matter of style. If you try to put them inside an existing function, it will not work.

Whatever we put as `functionName` we then can use in our program to run the command. For example, the following is a perfectly valid user-defined function to draw a centered circle that takes up half the screen:

```
void drawCenteredCircle()
{
  int canvasSize = 500; // should be in a global!
  int circleSize = canvasSize/2;
  ellipse(canvasSize/2, canvasSize/2,
          circleSize, circleSize);
}
```

But now that we've created this command, how can we use it? Simple: we just use it like any other command, by typing its name with brackets. In this case:

```
drawCenteredCircle();
```

To be clear, here is how the whole program may look.

```
void drawCircle()
{
  int canvasSize = 500; // should be in a global!
  int circleSize = canvasSize/2;
  ellipse(canvasSize/2, canvasSize/2,
          circleSize, circleSize);
}

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

void draw()
{
  drawCircle();
}
```

You can call your new `drawCircle` command from anywhere, and as often as you like. From the perspective of the draw block, this is just another command just like `line`, `ellipse`, etc.

Now, let's go back and look at the cat example from the beginning of this unit. We already have the `draw` block made with hypothetical commands. To finish the example, we can create the new commands and copy the code over. The only thing to consider is that your new functions can work with the global variables exactly like

the draw and setup block. That is, nothing has changed. Try it yourself before looking at the solution posted here.

```
/*******************
 * 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 drawCatHead()
{
  ellipse(headCenterX, headCenterY, 300, 300);
}

void drawCatEars()
{
  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);
}

void drawCatEyes()
{
```

```
  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);
}

void drawCatWhiskers()
{
  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);
}

void drawCatNose()
{
  ellipse(noseCenterX, noseCenterY, noseSize, noseSize);
}

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

```
  drawCatWhiskers();
  // draw the nose after whiskers for nice overlap effect
  drawCatNose();
}
```

The first thing you may notice is that this change actually made the program longer! That is okay, because each piece that you will work on at any given time is now smaller. Each function or the `draw` block is itself very small and easy to manage.

## 6.3 Functions and Code Execution Order

Something that often trips up people here is the concept that Processing jumps around while running your code. Previously, it always ran top to bottom within your `draw` block. Now, when it sees your new command, it pauses the `draw` block, jumps to your new command, does all of that, then jumps back. Within the `draw` block you still read it top to bottom, but you need to understand that it jumps out every time a command is run. The same actually happens, e.g., when the command `background` is run, but since we didn't write that command, we don't think about it.

The flow of the program is exactly unchanged from what we learned in the previous Unit. First, Processing creates and initializes the global variables. Second, it runs the `setup` block. Third, it runs the `draw` block. ***The change now, is that the draw block itself can run your new commands***. What exactly happens when your command is run?

- 1) draw block is paused
- 2) your command is completely run, top to bottom
- 3) draw block is resumed where it left off.

Let's look at the cat face draw block.

```
void draw()
{
  background(0);
  // copy the mouse position into the nose position
  noseCenterX = mouseX;
  noseCenterY = mouseY;
  drawCatHead();
  drawCatEars();
  drawCatEyes();
  drawCatWhiskers();
  // draw the nose after whiskers for nice overlap effect
  drawCatNose();
```

```
}
```

When this starts, everything happens as expected. The background is cleared to black, the comment is ignored, and the mouse position is copied into the nose position. However, when the `drawCatHead()` command is encountered, the draw block pauses where it is

```
void draw()
{
 background(0);
  // copy the mouse position into the noseposition
  noseCenterX = mouseX;
  noseCenterY = mouseY;
  drawCatHead();
  drawCatEars();
  drawCatEyes();
  drawCatWhiskers();
  // draw the nose after whiskers for nice overlap effect
  drawCatNose();
}
```

PAUSED

And jumps off to call the `drawCatHead` code

```
void drawCatHead()
{
  ellipse(headCenterX, headCenterY, 300, 300);
}
```

When this is complete, the draw block un-pauses, and continues where it left off. Then it encounters the `drawCatEars` command, pauses again, jumps to execute the code, etc.

**Important**: ***You should start developing your ability to trace through code this way, to understand what happens at what spot, what pauses, where it jumps to, etc. This will be crucial to completing your assignments and exams as the course progresses***.

## 6.4 Functions Calling Other Functions

I said earlier that you can call your user-defined functions from anywhere that you would use any other command. Until now, that meant inside the `setup` and `draw` blocks only. Now that you have other blocks with your own functions, you may

wonder if you can call other commands within those. Well, you definitely can! In fact, we could imagine that our cat face draw block could be simplified even further. We can take all the commands that draw the components of the cat face, and wrap them in another function called `drawCatFace` as follows:

```
void drawCatFace()
{
  drawCatHead();
  drawCatEars();
  drawCatEyes();
  drawCatWhiskers();
  // draw the nose after whiskers for nice overlap effect
  drawCatNose();
}
```

Then, the draw block is simplified even further:

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

This makes sense, for example, if you were going to draw a lot of animals. Now the draw block is clean and simple and leaves room for more code.

In this case, the pausing and jumping works as you may expect, it cascades.

- `draw` block runs
- `draw` block pauses at `drawCatFace`, and jumps to the function
- `drawCatFace` runs. Then pauses at `drawCatHead` and jumps to the function
- `drawCatHead` is completely executed, and jumps back
- `drawCatFace` un-pauses, then pauses again at `drawCatEars`, and jumps to the function.
- Etc...
- `draw` block un-pauses when `drawCatFace` is complete and continues.

**Advanced:** Can user-defined function call themselves? Or, can a call b, which then calls a, making a loop? **YES**. This is called recursion, a topic we do not cover properly in this course. Recursion is difficult, and you should avoid it in this course in general,

as it is very easy to get wrong. In fact, with recursion done incorrectly, you can crash Processing and lose your work. **Save often!**

## 6.5 Functions and Scope

Remember from our earlier discussions that scope is the range within which a variable exists. Outside of that scope, you cannot access or work with that variable. In general, ***any time we have a block we have scoping rules***. There is no exception here with functions, and in fact, you'll find that the scoping matches the rules you learned when we started active processing.

Since functions have their own code block, then each function has its own isolated variable scope. For example, the following code does not work:

```
void drawCircle()
{
  ellipse(location,location,5,5);
}

void draw()
{
  int location = 10;
  drawCircle();
}
```

Although you create a variable called `location`, and it is set before the `drawCircle` command is called, inside the `drawCircle` function the `location` variable is out of scope – it cannot see it. Processing will not run this code, saying that it cannot find the variable. Again, this is just what we already learned with the `draw` and `setup` blocks. How do you fix this? Make the variable global.

What about having the same variable name in multiple functions? What happens then?

```
void drawCircle()
{
  int location = 15;
  ellipse(location,location,5,5);
}

void draw()
{
  int location = 10;
```

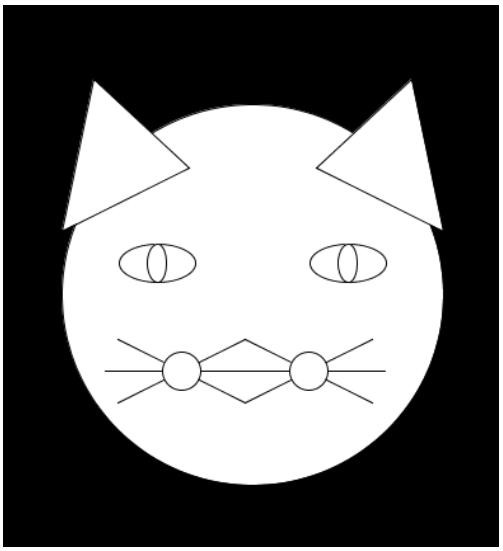
```
  drawCircle();
}
```

variables in different scopes can have the same name, but they do not share data – they are completely separate. Again, this is the same as we learned earlier. In this case, although they have the same name, to the computer, they are different variables.

**One last thing:** ***variables within a function are called local variables***

## 6.6 Reusing code

I promised at the beginning of this unit that user-defined functions help you to re-use code. Let's quickly look at an example of how this may work. What if we wanted to have a mutant cat – one with two noses! Since we already have the code to draw a cat nose, it would be great if we could somehow re-use that to draw a second nose. Luckily, our example is setup to help make this easy.



There are two keys to seeing how this may work. One is realizing that we can use our user-defined functions again and again. Two, is remembering that our nose code uses global variables, `noseCenterX` and `noseCenterY`, to determine where to draw the nose. So our strategy is simple:

- set some desired spacing between the noses, and save it in a global variable
- set the nose center variables to the left or right nose, and call our nose and whisker drawing functions
- change the nose center variables, and re-call the same functions

And it works! Here is my new `drawCatFace` code. Of course, the draw block no longer copies the mouse location into the nose variables.

```
void drawCatFace()
{
  drawCatHead();
  drawCatEars();
  drawCatEyes();

  // draw nose and whiskers 1
  noseCenterX = mouseX - NOSE_SPACING/2;
```

```
  noseCenterY = mouseY;
  drawCatWhiskers();
  // draw the nose after whiskers for nice overlap effect
  drawCatNose();

  // draw nose and whiskers 2
  noseCenterX = mouseX + NOSE_SPACING/2;
  drawCatWhiskers();
  // draw the nose after whiskers for nice overlap effect
  drawCatNose();
}
```

By putting that code into our user-defined functions, we were able to re-use it very simply. Now, if we find a bug in the nose or whisker code, and we fix it, it's fixed for both noses! We don't have to change the code in just one spot. As an exercise, it may be interesting to go back and modify the original cat face without functions, to have two noses, to see how messy it becomes.

## 6.7 **Top Down Programming**

Perhaps the biggest advantage of being able to make you own functions is the ability to break your program into smaller pieces as you see fit, to simplify your problem. This is perhaps the single most useful thing you will use in programming, the ability to take ridiculously hard problems (like, make the newest flashy video game) and organize it so that you can focus on one small piece at a time (like, draw a cat eye or move a nose). This approach to programing will continue throughout the degree, through Object Oriented Programming, data structures, algorithms, and so forth – basically, we (as humans!) are not that smart and need to develop better tools and techniques to make us smarter.

Now that you can make your own commands, you can learn a common technique for helping to simplify your programming job, called top-down programming. Simply put, top-down programming is where you first look at the overall problem to be solved, and slowly break it down into smaller and smaller chunks until you have something you can code up. For example, a top-down approach to the cat face example is to first define the high level structure: the cat has a face, eyes, ears, and a nose. Then, we start to think about how to do each of those parts. For example, a nose has whiskers and a circle. Finally, we write the code to make those parts.

Alternatively, a bottom-up approach is the opposite: we start by building robust pieces that do specific jobs, and then connect them together to solve problems. Lego is generally done bottom-up – you start with simple blocks and start building it into something, without necessarily having an overall plan.

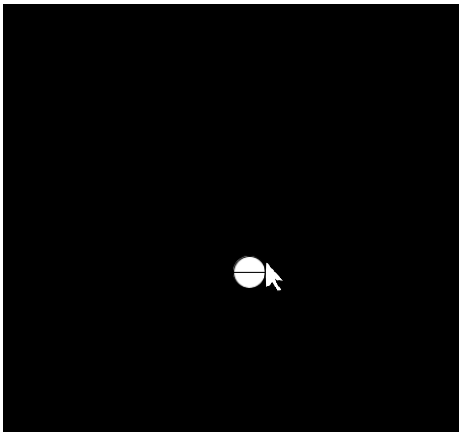
 

In the real world, there are benefits and problems associated with both styles. You will find yourself going back and forth depending where you are in a project, or which actual problem you are solving.

For this course, y***ou learn top-down programming as a means of helping you take a problem that may be somewhat overwhelming, and break it down into chunks until those chunks are manageable***.


You will get a lot of practice with this throughout the course, and I strongly recommend you use this approach when you feel overwhelmed. We will go through a toy example with somewhat simple code for illustration purposes.

We will make a program that has a planet moving around the screen. It will do a few things: move toward the mouse, with some friction to slow it down, and move in a repeating horizontal orbit from left to right. The planet itself is a circle with a line through it. There is a lot happening here, so it can be helpful to use top-down programming to simplify the problem. The general approach to the problem will be to calculate the desired move, do the move, and draw it.



We will approach this with four steps:

1. Write the program as a series of steps in comments, in English.
2. Turn each step into a function name (command)
3. Create empty functions
4. Start implementing the functions

**(Step 1)** Using top-down programming, first we plan the steps of the program in English and make sure it makes sense on that level. Here is an example of what our English version may look like.

- Clear the background to black
- Calculate the attraction to the mouse
- Add friction
- Update the orbit
- Move the planet based on the calculated move
- Draw the planet.

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 and listing the tasks that need to happen. The next step is to toss all of this list into our Processing program (as comments!) and convert them to function calls.

**(Step 2)** Wouldn't it be nice if we could simply convert this into commands and have it work? Imagine that you could simply take our above list and generate:

```
void draw()
{
  background(0);
  attractToMouse();
  addFriction();
  updateOrbit();
  movePlanet();
  drawPlanet();
}
```

In fact, this is what user defined functions are all about. We can think about the program abstractly, from a high level. We determine what needs to be done in English first (often in comments). We turn the comments into new commands. We then create those commands: for each command, we only focus on a narrow problem. For example, in `attractToMouse()` we don't worry about drawing or the orbit, just the calculation to move the planet to the mouse.

**(Step 3)** We next just implement empty functions, to make the skeleton of the program. Once we are done this, the program should run – but nothing will happen.

```
void attractToMouse()
{
}

void addFriction()
{
}

void updateOrbit()
{
}

void movePlanet()
{
}

void drawPlanet()
{
```

```
}
```

Just to re-cap, by this point, we have invented a bunch of functions that we think will help us do our job, and wrote up those functions as empty stubs. We haven't done any actual problem solving yet.

**(Step 4)** Start implementing the functions. The nice thing about this approach is that you can, for the most part, think about one problem at a time. Also, while you are implementing a function you can develop the required global variables and constants.

***Important: your initial functions may actually be a bad idea, and may not work. Don't be averse to re-factoring, that is, re-organizing your solution as you see problems with it or see a better approach.***

First, let's do the easy one. Drawing the planet. We need to have global variables for the planet position, since it will move around and we need to remember where it was last time. We also need a global constant to specify the planet size:

```
// main planet
final int PLANET_SIZE = 30;
int planetX = 0;
int planetY = 0;
```

With these globals, then we can draw the planet. The nice thing here is that we can use the planet location variables and the size, and solve the basic geometry without worrying about the other tasks. Here is my draw code, but try implementing by yourself first.

```
void drawPlanet()
{
  ellipse(planetX, planetY, PLANET_SIZE, PLANET_SIZE);
  line(planetX-PLANET_SIZE/2, planetY,
       planetX+PLANET_SIZE/2, planetY);
}
```

Next, let's attract to the mouse. All that this code does, is to determine what the move would be to move the ball to the mouse. This should be a simple subtraction calculation. However, we need new globals to store our guessed move. This is better than modifying the planet position directly, since then we can scale our move (e.g., for friction!) by dividing it, before actually performing the move.

```
// keep track of our calculated move
int planetMoveX = 0;
```

```
int planetMoveY = 0;
```

To do this, we just calculate how far the ball is from the mouse, and add it to the planned move. As the risk of being repetitive, try to solve this problem only thinking about the task at hand, and let yourself forget about the other tasks (like the orbit or drawing). This is one of the key benefits of using user defined functions.

```
void attractToMouse()
{
  int diffX = mouseX - planetX;
  int diffY = mouseY - planetY;
  planetMoveX = planetMoveX + diffX;
  planetMoveY = planetMoveY + diffY;
}
```

The next function in our list is to add friction to our calculation. There are a lot of ways to add friction (some physically correct, and some not ☺ ), but what we will do is divide our planned move by some amount. This way, larger moves get cut more, and smaller moves less. We need a global to store the friction amount

```
final int FRICTION_DIV = 10;
```

And to add the friction, we just divide the intended movement by our friction amount.

```
void addFriction()
{
   planetMoveX = planetMoveX / FRICTION_DIV;
   planetMoveY = planetMoveY / FRICTION_DIV;
}
```

Now is a good time to do the move code. If we assume that the move variables are correctly done, then we can just add the planned move to the planet location and we're done.

```
void movePlanet()
{
  planetX = planetX + planetMoveX;
  planetY = planetY + planetMoveY;
}
```

Now you should have a program that runs quite well, and a planet that approaches the mouse as you move it around. To continue the program, you need to solve the

orbit problem. In this case, I recommend using a fixed orbit size, and using modulo to make it loop when it gets too large. Try finishing this exercise on your own time. Note: you may have to modify your drawing code to get this to work!

Again, to re-iterate the benefits of this approach, you have so far solved most of the program without thinking about the orbit at all. This method enables you to focus in on one small piece at a time, and solve the whole problem part by part. Top-down programming:

1. Helps you focus on smaller problems and not be overwhelmed
2. Helps make self-commenting code, via descriptive function names
3. Helps you make a larger plan as you go

**Check your Understanding**

## 6.8 Check your Understanding: Exercises

Exercise 1. Create a space ship shooting at an enemy (unfortunately, not animated). Here is the draw and setup blocks.

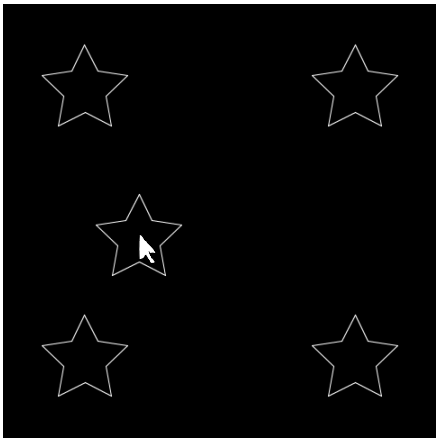
```
int shipX = 0;
int shipY = 0;
void setup()
{
  size(500,500);
}

void draw()
{
  background(0);
  shipX = mouseX;
  shipY = mouseY;
  drawSpaceship();
  drawEnemy(); // above spaceship
  drawMissile(); // between ship and enemy
}
```

a. Implement the user-defined functions specified with your own solutions.

b. Can you make the bullet animate, moving from the spaceship to the enemy, and repeating?

Exercise 2. The following program draws several “stars”. You can decide yourself how to draw a star, but using two triangles (one upside own) is a good way to do it.

a. Make two global variables: `starX` an `starY` that specify where to draw a star.
b. Make a user-defined function that draws a star at the globally-defined location
c. In you draw loop, draw several stars by alternately changing `starX` and `starY`, and calling your function. Also, draw one under the mouse.



Exercise 3. You will make a program that lets you draw fairly generic faces. You will have functions to draw eyes, nose, mouth, and head, which use global variables to determine their properties. Then you make other functions that modify the globals and call your helper functions to make different faces. It is up to you whether you take a bottom-up or top-down approach on this one.

a. Make global variables to specify all the parameters, e.g., head width and height, eye width and height, pupil ratio, etc. You have some flexibility here. Also, make variables for each item to specify *where* to draw them.
b. Make user-defined functions that draw the parts using the global variables.
c. Make two user-defined functions: `drawNormalFace()` and `drawSillyFace()`. These set the proper global variables and use your other user-defined functions to draw two different faces at two different locations on the screen.

## 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.