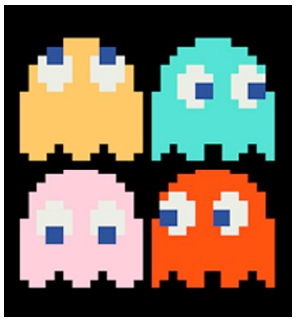
Homework Exercise 8: Blinky



Goal: The purpose of this assignment is to give you experience applying functions to your programming … it's abstraction!

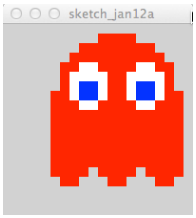
In the ancient game of Pac-Man there were four ghosts; we will focus on the red one, named Blinky. Blinky and the other ghosts are built with rectangles and simple colors, because the technology of the day didn't allow for anything more complicated.

Our goal is *to program the ghosts so they can be located in different positions, and make their eyes move*.

Getting Started. The first matter to consider before programming is how to approach the problem. This typically involves deeply understanding what must be done, and abstracting to build up parts to solve the problem.

What must be done: The ghost bodies are all the same shape; only their colors and positions vary. The whites of their eyes are all the same shape and color, but their position changes. The irises (blue) are also all the same shape and color, but they also change position.

Abstracting the Parts: This simple problem can be solved with three functions. What should they be? Recall from earlier discussions and examples, that when we abstract, we look for coherent parts of a problem that, if programmed as functions, become the "large instructions" to solve a problem. What should they be? Our goal will be to draw Blinky, the red ghost, looking straight ahead. However, each of the functions will be parameterized so that we can draw the others using the same functions, but with different parameters.



Part 1. The Plan. Figure out the three functions. These should be pretty obvious, but if not, check the "Getting Started" discussion above. Give the specification for each of the three functions for this problem using the following form:

Function name: *you think it up*
Parameters, if any, (give names & datatypes): *things that change need parameters*
Returned value, if any:
Short description (in English) of what the function does:

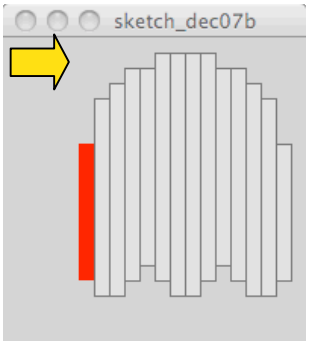
These should be written out using a word processor, because it is your plan for the intended work, and you will need to turn it in. Notice – there is never one correct answer; there are different ways to abstract this.

Part 2. Build the Functions. Write the necessary Processing code to implement the functions.

Although the programming is straightforward – it is very similar to exercises you have already done – the following suggestions might help make you more efficient.

Suggestion 1. Write the setup( ) and the draw( ) functions. You will need these in order to display your work. In setup( ) you will need a 400x400 size canvas, or so. Also, you don't want the rectangles that make up the figure to be outlined, so specify noStroke();. In draw( ) you will want a call to whatever function you are working on at the moment. This will change as you do your work. Finally, you may want to set up the three functions that you will be writing, giving each a name, its parameters and their data types, and the other punctuation. (They don't have to contain any code at the moment.) Remember, they don't go inside of draw( ), they go after it.

Suggestions 2. We assume each "square" region in the ghosts picture above as covering 10x10 pixels on the screen – so the blue irises are 20x20, because they look to be 2 squares by 2 squares. The easiest way to draw the ghost is to draw vertical strips each of 10 pixels wide. The plan at right shows the necessary strips to form a ghost. Notice that they are different lengths and heights, but all of the units are multiples of 10. So, beginning at some arbitrary position on the canvas, say, x, y (illustrated by the yellow arrow), the first strip, must be 0 pixels to the right of x, and (estimating) 60 pixels below y. The rectangle is 10 pixels wide, we know that, and (estimating) it's 90 pixels long. Then we define the rectangle drawing function as

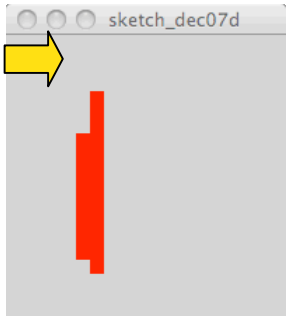


plan for drawing Blinky

```
rect( x+0, y+60, 10, 90 );  //From left, bar 1
```

We want x and y mentioned because we want to draw ghosts in different places, so x and y will be parameters to the function. We need to estimate the positions and sizes of the rectangles because our plan does not tell us how big they should be. We could work the whole thing out on graph paper, and know exactly where the positions are, but it is easier to "estimate and adjust."

We continue in the same manner to complete the implementation.



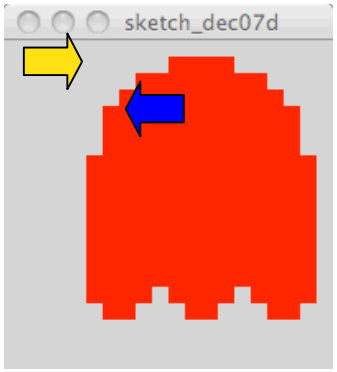
The next rectangle is positioned 10 pixels to the right of x, but it's not as far down from y as the last one was. Suppose it's 30 pixels less far down, then the rectangle will be something like,

```
rect( x+10, y+30, 10, 130 );  //From left, bar 2
```

The length is 130 because the second strip starts 30 pixels higher and is 10 pixels longer than the previous one, which was 90: 130 = 30 + 90 + 10.

And we continue in this manner.

Suggestion 3. When planning the eyes, we will also use the same arbitrary point, x, y that we have been using. However, the whites – which will also be built using 10-wide strips – need to be offset from a base position.



That is, the two white parts, which stay the same distance apart, can move from their base position in the "center," left or right, up or down, according to the first picture of the ghosts above. So we define a second base point (blue arrow) as the base for the whites and refer to its position as u, v. This position is x+20, y+30. So the first white strip, which is 10 down from the blue position and 30 long is at

```
int u = x + 20;          //blue base x relative to yellow
int v = y + 30;          //blue base y relative to yellow
<two more statements go here; see next>
rect(u, v+10, 10, 30);   //draw left strip of left eye
```

The other white strips are positioned similarly.

Suggestion 4. The whites shift left by subtracting 10 from u, and they shift right by adding 10 to u. So, if we had a parameter, call it lr, that had the value of -1, 0, 1 depending on whether the shift is left, none or right, we could write

```
u = u + 10*lr;          //adjust u position if white must shift L or R
```

This adjusts the value of u, the x-direction, by -10, 0 or 10 depending on whether lr is -1, 0 or 1. We can do the same thing with v using a similar variable ud for up/down

```
v = v + 10*ud;          //adjust v position if white must shift U or D
```

These two instructions can go just before the definition of the white strips, indicated above.

Suggestion 5. The blue irises work just like the whites. They can either get their own relative position, offset from the yellow arrow, or they can use the blue one, that is, u,v. We will use the blue position in this discussion.

We will shift the blue just as we shifted the white. Referring to the ghost figure above, we notice that if the white is already positioned, then the blue can shift left (-10) from its looking forward position, right (+10), down (+10) or up (-20). This last seems odd, but it is because the eye is longer than it is wide, making the situation asymmetric. So, assuming the u, v are defined as above (including white adjustment), and lrb and udb are the blue adjustment integers, then the blue is draw by

```
u = u + (10*lrb);          // left is lrb=-1, right is lrb=1
v = v + (10*udb);          // down is udb=1, up is udb=-2
```

```
rect(u+10, v+20, 20, 20);    // draw the iris (both strips)
```

Part 3. Show Your Stuff! *Make sure your program is fully commented.* (It's an important part of your grade.) Then modify your draw( ) code to show the four ghosts as they are in the original figure above.

Wrap Up. You have used abstraction, figuring out three functions that allow you to draw the Pac-Man ghosts. There was work in getting all of the rectangles right, but overall, the suggestions made the task direct. The process applied here works all the time, and you will find many places to use it.

Submit your two files, the "design" file (.doc) and the program file (.pde) in the course drop box.