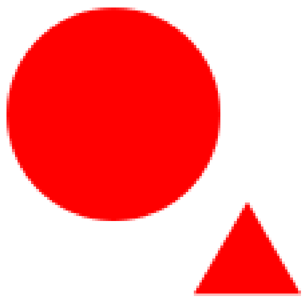
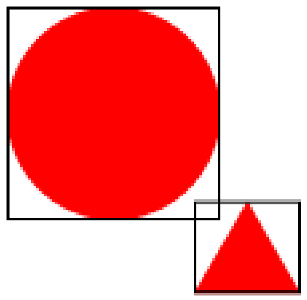
# Exercise 6: A 2D Collision Engine

In this exercise we will be making a collision engine for a 2D video game. The engine will be able to test for collisions between three kinds of shapes: circles, rectangles, and equilateral triangles. Arbitrary collisions between different shapes is both complicated and slow, so most game engines just approximate collisions using what are called bounding boxes. The idea is that we draw an imaginary box around every shape and just check if those boxes intersect, since checking if boxes overlap is easy. So to check if there is a collision between this circle and this triangle:



We would just draw boxes around them:

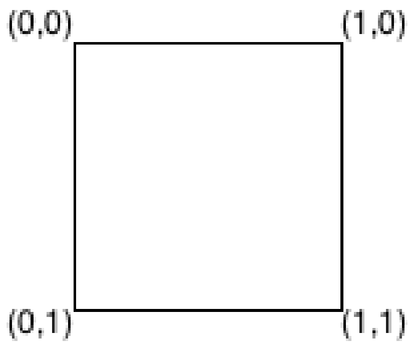


And see that they do, in fact, collide! Its not perfect but its good enough for most games.

Now collisions are a tricky business, so we have implemented the game loop and collision checking for you. However the file you have been given has no shapes, nor their bounding boxes!

## Drawing, Graphics, and Coordinates in Computerland

Graphics coordinates work a little differently than coordinate systems you are probably used to. When graphing functions in high school the coordinate (0,0) was usually in the bottom right or center of whatever we were drawing, with higher y values meaning higher up on the y axis. But in computerland we flip that: (0,0) is in the top left[1]. So our coordinate system looks like:



By the way, this is the code to draw the square above:

```
(define bx (square 100 "outline" "black"))
(place-image bx 100 100 (square 200 "solid" "transparent"))
```

Which just draws a 100x100 square onto a 200x200 background. To add the text we do:

```
(define p0 (text "(0,0)" 12 "black"))
(define p1 (text "(1,1)" 12 "black"))
(define p2 (text "(1,0)" 12 "black"))
(define p3 (text "(0,1)" 12 "black"))
(define bx (square 100 "outline" "black"))
(place-image/align
 p3 50 150 "right" "top"
 (place-image/align
  p2 150 50 "left" "bottom"
  (place-image/align
   p0 50 50 "right" "bottom"
   (place-image/align
    p1 150 150 "left" "top"
    (place-image
     bx 100 100 (square 200 "solid" "transparent"))))))
```

This works by building up an image from smaller images. The `place-image` function places the center of the image that its first argument on to the given location on its

[1]The reasons for this are largely historical. In the days before all these fancy graphics computers just had text, and that text started in the upper left hand corner. And now we're stuck with it.

last argument. The `place-image/align` function places the image in the same way, but instead of placing its by the first images center, its placed by the point described by the fourth and fifth arguments. So `"top" "left"` uses the top-left corner. Remember, if you're ever confused about a function does or how to use it, hit "F1" to open up the documentation.

## Shapes and Methods

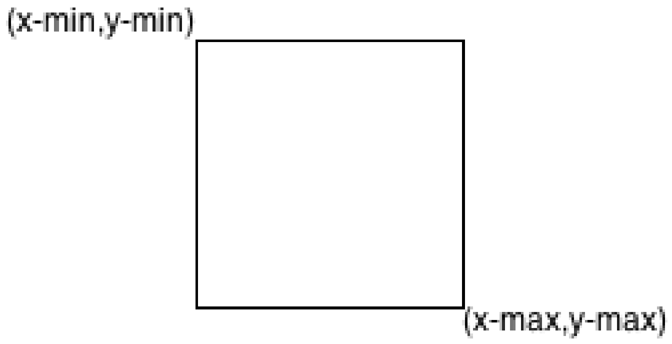
The file you have been given has three unimplemented functions: `create-triangle`, `create-rectangle`, and `create-circle`. Your job is to implement these functions so that they each return a structure for that shape. These structures must implement three methods: `render`, `move`, and `get-bounding-box`.

- `(render shape? image?)` → `image?` — This method should draw your shape onto the given image. There are functions for creating images (`circle`, `triangle`, etc) and a function to add images to another image at a position using `place-image`. You should use these functions.

  Note: We won't be testing the render method. Just do something reasonable for it so you can see what's going on when you run the simulation.

- `(move shape? posn?)` → `shape?` — This method produce a shape like `shape` but moved by `posn`. This means that the shape should be moved horizontally by the given posn's x coordinate and moved vertically by the given posn's y coordinate. So if the shape was at `(make-posn 10 10)` and was told to be moved by `(make-posn 5 5)`, the new shape should be at `(make-posn 15 15)`. The provided function `posn+` will probably be useful here. Remember that this method is *functional*. It should not mutate the given shape, but return a new one.

- `(get-bounding-box shape?)` → `bounding-box?` — This method should return the bounding box for the given shape. A bounding box is a represented by the structure `(make-bounding-box x-min y-min x-max y-max)`, where `x-max` is the x coordinate of the upper left hand corner of the bounding box, `y-min` is the y coordinate of the upper left hand corner, `x-max` is the x coordinate of the lower right hand corner, and `y-max` is the y coordinate of the lower right hand corner:



## This is Where You Come In

As mentioned before you will need to implement three structures, one for each shape. But each shape will have something in common: a position on the screen. So, you should have a common parent type that has a position!

```
(define-struct shape (posn))
```

Remember the position here represents the center of the shape.

### Rectangles

Time to implement `(create-rectangle pos width height)`. This function takes the center of the rectangle, and the width and height of the rectangle, and will return a structure you create.

Rectangles are probably the easiest of the shapes, since their bounding box is just well, themselves, offset by the current position. So, for a rectangle of width `10` and height `8` at position `(make-posn 10 10)`, the bounding box would be `(make-bounding-box 5 6 15 14)`. (Hint: you should turn that into a `check-expect`!).

Do you remember the syntax for methods and structs with that inherit from other structs?

```
; we can't call it rectangle, that function exists
(define-struct (rect shape) (width height)
   #:methods
   (define (move self v) ...)
   (define (render self scene) ...)
   (define (get-bounding-box self) ...))
```

You just need to fill in the `...`s with the correct code!

### Circles

Now on to implementing `(create-circle pos radius)`. This function takes the position of the center of the circle, and the circles radius, and returns a structure of your own creation.

Circles are also fairly simple. They have one important attribute: their radius (and of course the position from the super structure). Can you write this one on your own?

### Triangles

Now on to implementing `(create-triangle pos side-length)`. This function takes the position of the center of the triangle, and the side length of the triangle, and your structure.

This one is a little harder. How much geometry do you remember? Moving should be very similar to the last two problems, and drawing can be accomplished with the `triangle` function, which draws equilateral triangles. But how does one find the bounding box of an equilateral triangle given only the side length? To do this we will the height and width of the triangle, which luckily are the same as the height and width of the bounding box. You should be able to do with the Pythagorean Theorem and the length of the sides.

## Testing it

You should, of course, write lots of tests. There are two good kinds of tests to write: high level tests that check if two shapes intersect (the provided `intersects?` function is what you want here). And tests that check if the bounding box is correct. For example, here are some of the tests I wrote:

```
(check-expect
 (intersects? (create-rectangle (make-posn 10 10) 10 10)
              (create-rectangle (make-posn 10 6) 10 10))
 #t)

(check-expect
 (get-bounding-box (create-rectangle (make-posn 10 10) 10 10))
 (make-bounding-box 5 5 15 15))
```

But its hard to see whats going on sometimes when you're down inside of tests. It also helps to visually inspect what is going on, to direct you to areas that might need more tests. For this purpose we have given you a `(start)` function. When run this function should give you a window. Any time you click on the window a random shape will be created, moving in a random direction. The little simulation also has gravity, so the shapes will slowly fall to the ground. Shapes that collide are frozen in place, and their bounding boxes are drawn. Frozen shapes will not collide with unfrozen shapes. Click about in this to see how your shapes interact and collide!