Tutorial session the first:

The goals of this tutorial is to get you started to Racket, and to have you set up the Handin
Plugin you will use to submit assignments, and to have you start making animations!

Exercise 0, Getting DrRacket & the handin client setup:

This exercise is not in pairs.

1. Download & Install Racket from http://download.racket-lang.org/

2. Startup DrRacket and go to "File -> Install Package" and
enter http://users.eecs.northwestern.edu/~sfq833/handin-client.zip

And press install. When the installation is finished, press close.

3. Restart DrRacket

4. Go to "File -> Manage EECS 111 Handin Account"

Fill out the form. The username MUST be your netid, and the email MUST be your northwestern email.

5. Now open a new file in DrRacket

In the bottom left select `Choose Language...` -> `Intermediate Student with lambda`

Now, define a function `minutes->hours` that accepts a number of minutes and computes how many whole
hours these minutes represent.

Remember that this means you're defining a variable called minutes->hours, and you're giving it a
value which is a procedure. You make procedures using lambda expressions. Since your procedure
is going to take the number of minutes as its input, your lambda expression will look something
like:

(lambda (minutes) … whatever …)

That says "make me a procedure, whose input will be named minutes, and whose output is the expression whatever." You need to decide what the whatever is.

6. submitting the assignment:

Click the "EECS 111 Handin" button. It has a small yellow icon and is next to the "Step" button.

Select the assignment you are submitting, enter your password, and press submit.

This submission will not be graded. We are simply using it to make sure everyone gets a handin
account properly set up. This exersize is the only one you will be submitting this tutorial.

Exercise 1, Playing with Images:

In a new file.

At the top of this file put `(require 2htdp/image)` and hit "Run".

Now lets play a with some images. In the Interactions Pane (that other panel with the ">" in it),
type `(rectangle 100 100 "solid" "red")`. And voila, we have an image! And we can check, how big it
is too, with `(image-width (rectangle 100 100 "solid" "red"))`. Once last easy one, we can frame a
blue rectangle by placing it on a slightly larger rectange:

```
(overlay (rectangle 90 90 "solid" "blue")
         (rectangle 100 100 "solid" "red"))
```

Now its your turn. Define a function `red-frame`, consumes an image and draws a red frame around it.
This should behave exactly like the previous example, but it should work on any sized image.

Copy an image into DrRacket and frame it.

Now lets see exactly whats going on, using the stepper.

Remember that bit in lecture about "substitution"? The stepper shows you exactly how your program is
evaluated, one substitution at at time. Launch the stepper via the "Step" button (with the green |►), and step through a
few examples of `frame`.

Exercise 2, Animation:

This exercise, and the rest, are in pairs.

In a new file.

use `(require 2htdp/image)` and `(require 2htdp/universe)`

The `2htdp/universe` library helps us create animations! For starters lets create a circle that slowly grows.

Animation works just like the movies: we draw one picture for every frame. So, lets figure out how to draw a circle:

```
(require 2htdp/image)
(require 2htdp/universe)

;; our animation will be 100x100 pixels
(define size 100)

;; make a circle of size `t` draw it into the middle of a blank background
(define growing-circle
  (lambda (t)
    (place-image
     (circle t "solid" "black")
     (/ size 2); where the center of the circle goes, horizontally. `0` means all the way to the left
     (/ size 2); where the center of the circle goes, vertically. `0` means all the way to the top
     (empty-scene size size))))
```

Cool! You can read more about what functions are available for animation and drawing opening the help desk via `Help -> Help desk, and searching the documentation for `2htdp/image` or `2htdp/universe`.

Play with the `growing-circle` function a bit. Step through it with the stepper once or twice.

And now, lets animate it!

```
;; a little bit of magic that calls `growing-circle`
;; on successivly larger numbers, and displays the image that results
(animate growing-circle)
```

And that's it!

Now lets see what you can do. First, figure our how to make the circle grow at half the speed. And then a quarter.

Now, take this slowly growing circle and have it move around the screen, starting at the top left, and moving to the bottom right.

Now make the circle change color with time! (Do you remember how to make colors from numbers? If not, look up `color` in the `2htdp/image` documentation`). Try to make the color change in interesting ways.

Now, right now the circle grows without bounds. But that's no fun, right? After a while you just have a screen that's one solid color. Can you modify the program so that the circle grows and shrinks, always staying within the screen? (the `modulo`, `max`, and `min` functions can be your friend here.)

Exercise 3, Things get more complicated:

This exercise is open ended. In it we will learn to make interactive animations, using `big-bang`.

`2htdp/universe` allows you to model a world. Every time the clock ticks, `big-bang` uses one of your functions to update or create a new world, which becomes the current world. It can use another one of your functions to create an image of your world.

Our first "world" will represent the number of ticks passed:

```
;; calculates the next state (increments the time)
(define next-state
 (lambda (s) (add1 s)))

(define state-draw
 (lambda (t)
  ... Put the drawing function from the last exercise here! ...))

;; and with this
(big-bang 1
 (on-tick next-state)
 (to-draw state-draw))
```

Now lets change it up. Instead of changing the current state every tick, lets do it when the mouse moves:

```
```

```
;; change world-draw to take some new arguments
;; takes the state, the x and y coords and the event
(define next-state
  (lambda (s x y event) (add1 s)))

;; and change big-bang to look like this
(big-bang 1
  (on-mouse next-state)
  (to-draw state-draw))
```

Now, change `next-state` and `to-draw` to change your circle based on how far the mouse is from the center of the screen. (the pythagorean theorem will probably be helpful here).

Did you make it this far? Well done! Now go experiment more with `big-bang`! Try making a simple game, or just extend the animation we have!