JavaScript Turtle Graphics

The Concept

JavaScript is an important language, because it is the language used in browers to bring life to web pages. It dynamically resizes elements so that the same page can be used on a mobile phone or desktop computer. It dynamically loads data, like weather data or news stories, as it changes. It provides motion and interest to some pages. It checks user inputs before a request is sent back to the server.

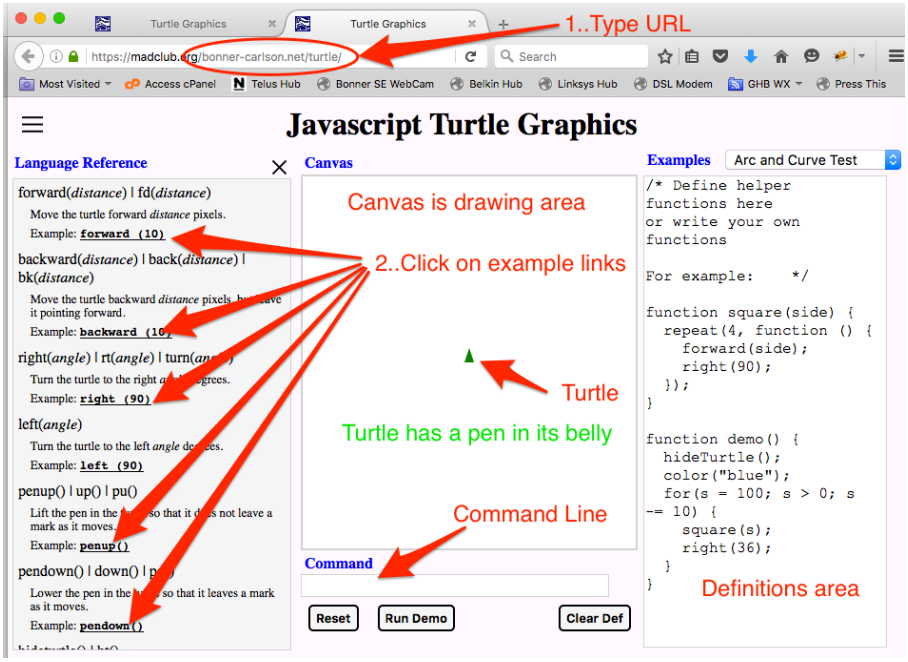
Learning JavaScript programming in a graphics environment is a way to get introduced to JavaScript. The JavaScript Turtle Graphics page at http://bonner-carlson.net/turtle is written in JavaScript and it provides an environment for exploring JavaScript and its use of graphics using traditional turtle graphics functions.

The screen is divided into three basic areas: language reference, canvas, and definitions.

The Language reference contains the functions that may be used to build graphics on the canvas. The underlined examples may be clicked upon to see their effect immediately on the canvas, as they are commands for moving the turtle and its pen.

The canvas is the area where drawing takes place. The drawing is done by the turtle which is modeled as a triangle pointing in the direction of travel. Imagine that the turtle has a pen in its belly and that the pen leaves a mark as the turtle moves. Under the canvas is a command line where commands can be executed individually. Below that is a set of button that are used to clear the canvas and to control the execution of the definitions.

The definitions area allows for the creation of new functions. This may be user entered functions, or examples loaded with the examples selector above the definitions area.

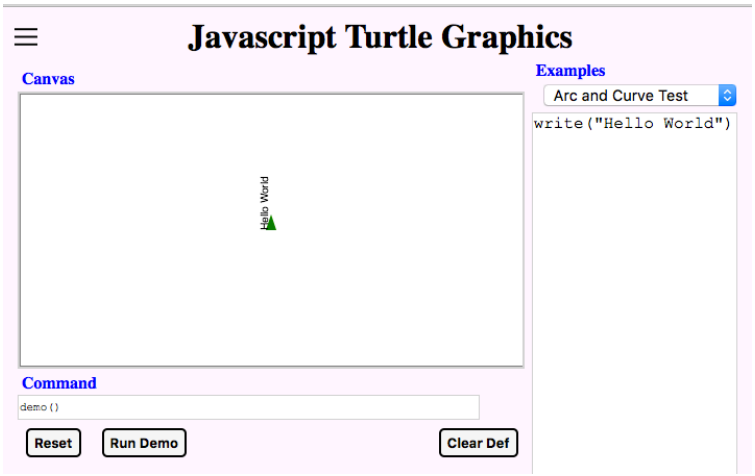


Hello World

When programmers are faced with a new language, the first thing that they do is to write a simple program that tests their understanding of the language, syntax and operation of the new language. Typically this is a "Hello World" program. For this environment such a program would be as follows:

```
write ("Hello World")
```

Click on the Run Demo button.



What happened? You should see Hello World printed on the canvas, but printed sideways. Text is written in the direction that the turtle is pointing.

Write() is a function. Functions perform a desired action and are composed of other functions or instructions. Functions allow a programmer to write code without having to worry about all of the underlying details. Turtle graphics is written using functions that hide the details from the user.

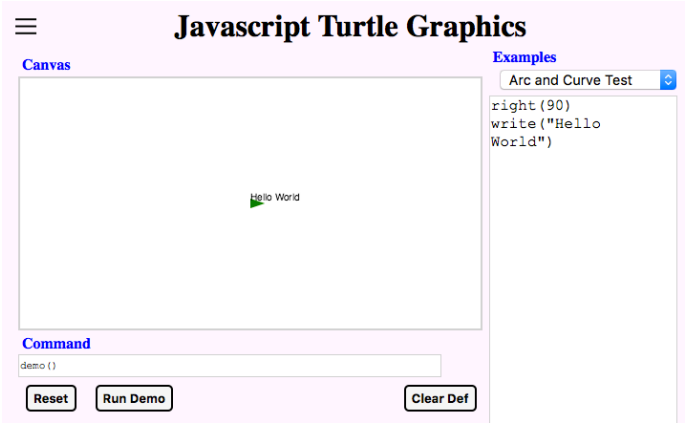
Case is important to JavaScript, so **Write** is different than **write** and is different than **wriTe**.

All functions in JavaScript are followed by open and close parentheses. This is special syntax or grammar that tells JavaScript that the preceding name is a function and not something else. Some functions take no parameters, in which case there is nothing between the open and close parentheses. Some functions take a number of parameters. The write function takes a single parameter, a string that you want to write to the canvas. The syntax for a string is to enclose it in quotation marks. This distinguishes the string from other characters. Either single or double quotes are acceptable, as long as it is the same type are used on both ends. Allowing both is one way that the syntax allows the embedding of the other type within the string. So, you could use "G'day world", if you want the Australian version of “Hello World.”

The program as it is, isn't quite right. the **write()** function displays a text string in the direction that the turtle is moving without moving the turtle. We want the text to go from left to right, so we need to turn the turtle 90° right. We do this with the addition of a **right(90)** function. Because this is getting more complex, lets use the definitions area, so we can edit the programs and make changes to it as necessary. The resulting program for the definitions area is:

```
right (90)
write ("Hello World")
```

click on the Run Demo button, you should see something like the following:



Right also takes a parameter like write, but right's parameter is the number of degrees to turn right.

Part of JavaScript's syntax, is that white space is not too important so you can add spaces, tabs or carriage returns (newlines) here or there to make the code easier to read. Each JavaScript statement is usually written in a single line, although an exceptions is made for long statements that read better with multiple lines. Each line in formal JavaScript must end in a semi-colon ';' as in:

```
right( 90);
write ( "Hello World");
```

We are writing in a more casual syntax that does not require this semi-colon. Just beware that some syntax checkers will require more semi-colons. If multiple codes statements are placed on one line, the code statements must be separated with semi-colons.

Comments are important to remind you or the next reader what the code is attempting to do. (Sometimes the code misses the intent, so it is important to state the intent, and to keep that up to date as the code changes.). JavaScript has two types of comments. A comment that just tacks on the end of a line starts with a double slash '//' and goes to the end of the line. A mulit-line comment starts with slash-star '/*' and ends with a star-slash '*/". Multi-line comments may not be nested. as the comment end with the scanning of the first star-slash after the slash-star.

So both **right()** and **write()**, are functions. These hide the details of their implementation for their user. JavaScript allows its users to define their own functions. We can define a 'Hello World' function, 'hi' as:

```
function hi () {
  right( 90);
  write ( "Hello World");
}
```

function is a key word in JavaScript that tells it that you want to define a function. This is followed by the name of a function. You should pick a unique name for the function, as this definition will override any previous definition. The open and close parentheses tell JavaScript that the function definition includes no parameters in this case. The open and close curly brace are used to mark the beginning and end of a block of statements to be executed when the **hi** function is invoked.

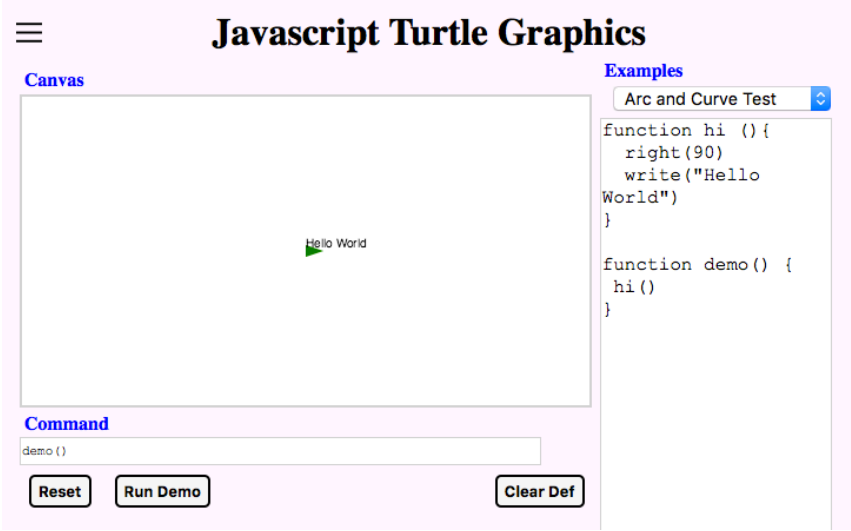
Try to execute the program by pressing on the Run Demo. Nothing should happen or you will get a error message.

The error message occurs because there is no **demo()** function defined. So lets adds a demo function to the program:

```
function hi () {
  right( 90);
  write ( "Hello World");
}

function demo () {
}
```

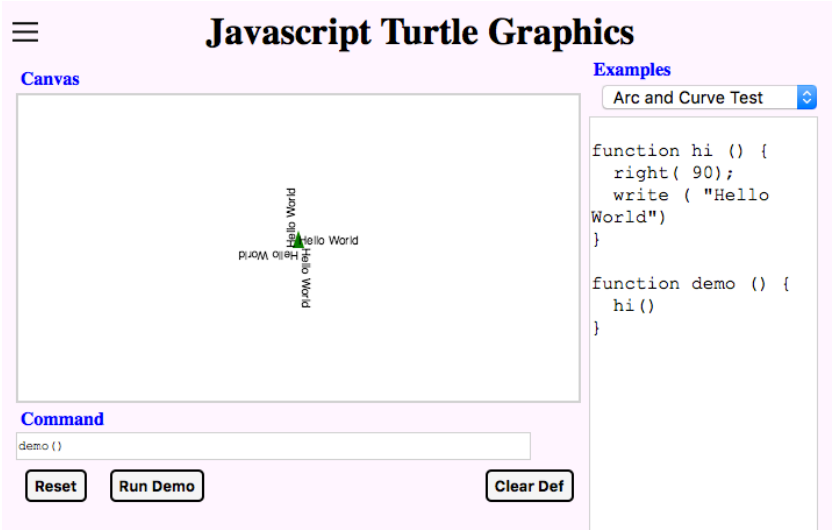
The error has gone away but nothing has executed. Why? Because the helloWorld function was never called or invoked. Let's add a call to the **hi()** function to the **demo()** function.

```
function demo () {
  hi()
}
```



Now when you press the Run Demo button, the turtle should print "Hello World" on the canvas.

What happens if you press the Run Demo button more than once? Why?

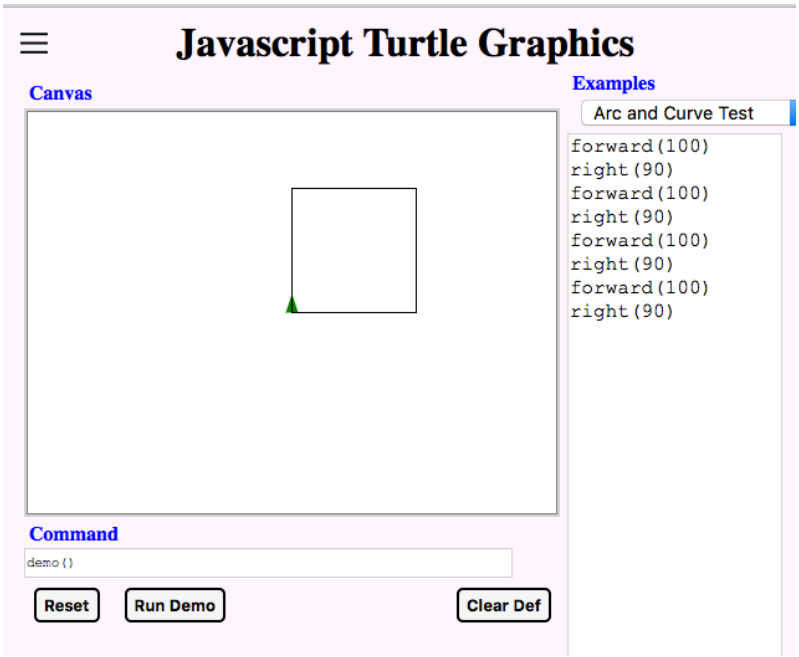


We could fix that by clearing the screen by invoking the **clear()** command before writing to the canvas.

```
function demo () {
  clear()
  helloWorld()
}
```

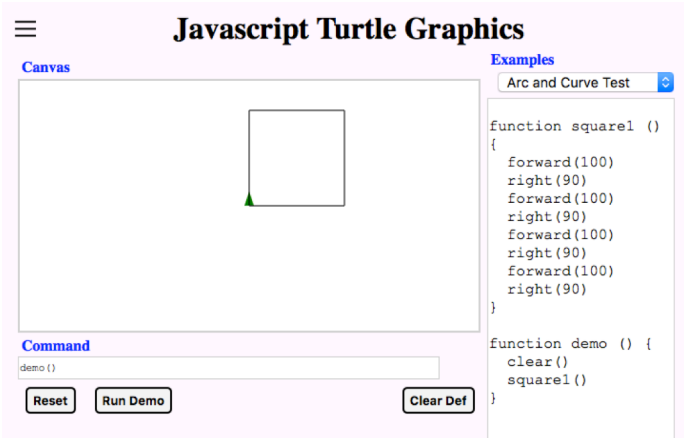
Basic Graphics

OK, but Turtle Graphics is a graphics program, shouldn't we be doing some graphics? Let's see it draw some lines as sort of a graphical Hello World example. Let's try to do a simple square.



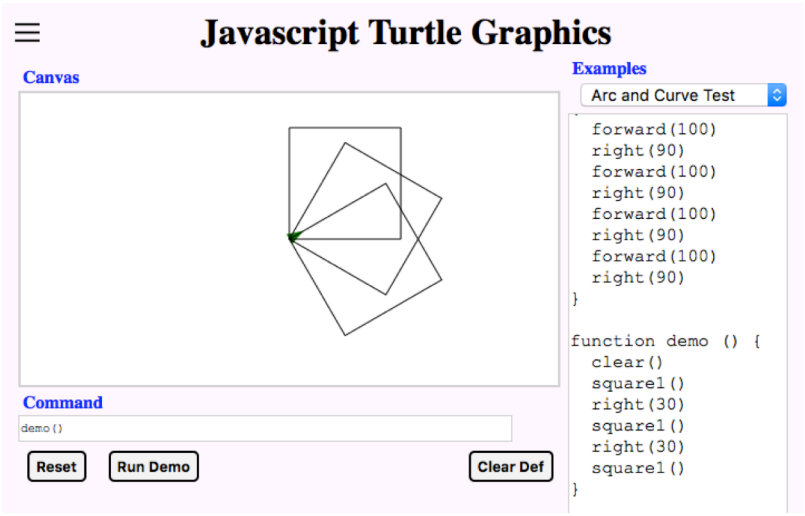
Ok, let's do the same thing, but do it with the function call that we learned previously.

```
function square1 () {
  forward(100)
  right(90)
  forward(100)
  right(90)
  forward(100)
  right(90)
  forward(100)
  right(90)
}

function demo () {
  clear()
  square1()
}
```



Let's see some of the power of a function with a simple change to draw 3 squares rotated 30° about the starting point. Change the **demo()** function as follows:

```
function demo () {
  clear()
  square1()
  right(30)
  square1()
  right(30)
  square1()
}
```



Great! now looking at square1, there is a lot of repetition. There are a couple of ways to do this. Using the Turtle Graphics function **repeat()**, the code would look something like:

```
// square with repeat
function el () {
  forward (100)
  right (90)
}

function square () {
  repeat (4, el)
}
```

But that is messy and not really the JavaScript way of doing things. Let's get rid of that with a **while** statement. The simplest JavaScript program using **while** is something like:

```
var i = 0;
  while ( i < 4) {
    i = i + 1
}
```

What does this do? The reserved word **var** is used to define a variable, or more specifically where a variable is defined. **i** is a variable being declare. The statement **i = 0**, is an assignment of the value 0 to the variable i. This could be read. set variable i to 0. **i = 0** is just a short hand form of that. This is definitely not the way arithmetic symbols are used. Sometimes a variable is declared without setting it to an initial value. In general this is a fairly bad idea. It is definitely an error to use a variable before its value is set.

The next statement is the while statement. 'While' is a key word and says to repeat the following statement or group of statements surrounded by an open and close curly brace, while the condition enclosed between the open and close parentheses evaluates to true. The only statement within the curly brackets is i = i + 1.

So what happens when this is run.

the variable **i** is allocated and set to 0
since **i** (being 0) is less than 4, the group of statements is executed
i is set to **i** + 1... **i** is now 1
control returns to the **while** statement
since **i** (being 1) is less than 4, the group of statements is executed
i is set to **i** + 1... **i** is now 2
control returns to the **while** statement
since **i** (being 2) is less than 4, the group of statements is executed
i is set to i + 1... **i** is now 3
control returns to the while statement
since **i** (being 3) is less **than** 4, the group of statements is executed
i is set to **i** + 1... **i** is now 4
control returns to the **while** statement
since **i** (being 4) is no longer less than 4, the group of statements is skipped over

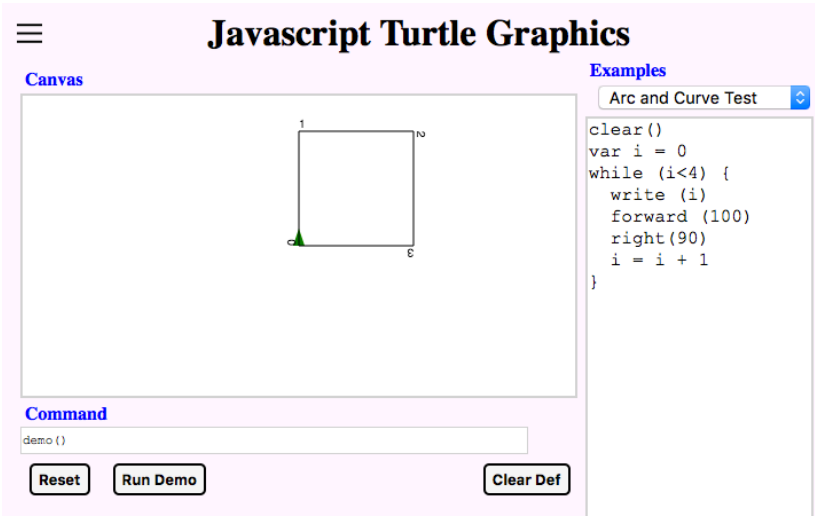
This is all hard to see, so let's make it more visible. Let's print the value **i** inside of the loop. Since the **write()** function doesn't move the turtle, we need to do that as well. This is a common tool used by programmers to see inside of a program.

```
clear()
right(90)
var i = 0
while ( i < 4) {
  write (i)
  forward (15)
  i = i + 1
}
```

The bottom line is that the inner statement is executed exactly 4 times. This type of loop is also called an *iterative* loop as it is executed a counted number of times. The control variable is sometimes called an *iterator*.

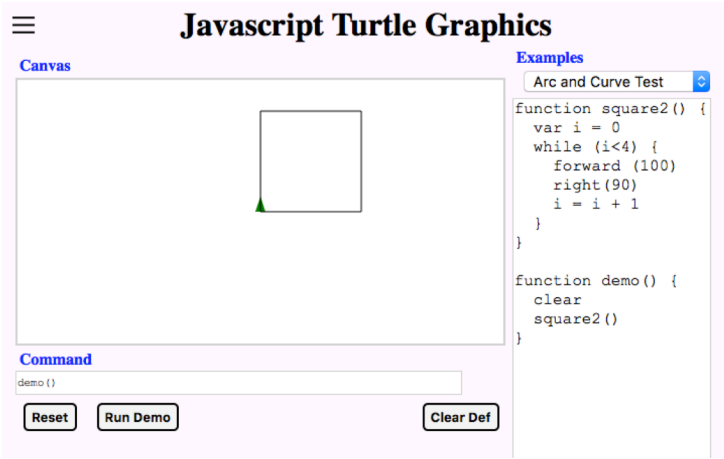
```
clear()
```

```
var i = 0
while ( i < 4) {
  write (i) // not necessary, but just to show the iterator value
  forward (100)
  right(90)
  i = i + 1
}
```



This shows the iterator, but that really isn't necessary, so let's remove the **write(i)** statement and make into a proper function with a **demo()** function calling it. While you are at it, move the clear statement from the square function to the demo function to remove this side effect. You get:

```
function square2 () {
var i = 0
  while ( i < 4) {
    forward (100)
    right(90)
    i = i + 1
  }
}

function demo() {
  clear()
  square()
}
```



Wow! OK, now let's try to add a parameter to the basic **square3()** function. We'll tell JavaScript that we want to pass a parameter in the function definition:

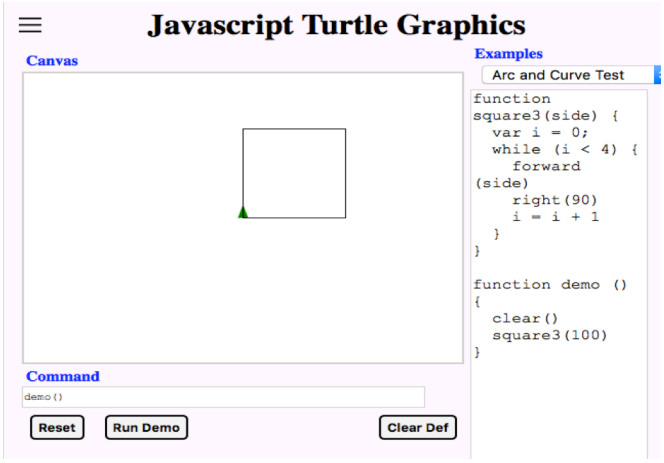
```
function square3(side) {
```

To use that value, we just substitute it for the fixed 100 size of the forward line function as in:

```
forward (side)
```

Putting the two together, the code should look something like:

```
function square3(side) {
  var i = 0;
  while ( i < 4) {
    forward (side)
    right(90)
    i = i + 1
  }
}
```

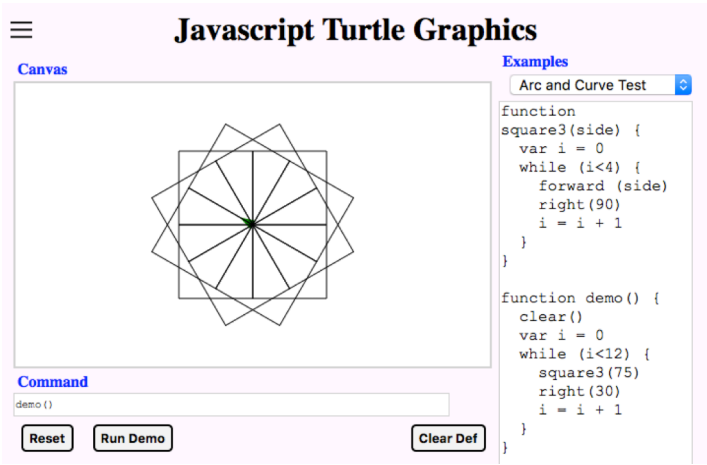


We also need to modify the function call in the demo() function to use the new parameter.

```
function demo () {
  clear()
  square3(100)
}
```

Hey, isn't that just like we did before. That was a lot of work to change the routine. Why would you do that? To show off the power of a parameterized function we can now use the iterator, so let's put in an iterator into the demo program that turns the square while changing its size. Let's do this one step at a time. First just turn it.

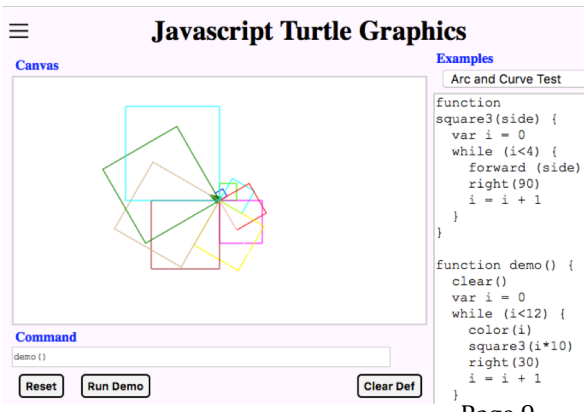
```
function demo () {
  clear()
  var i = 0
  while (i < 12) {
    square3(75)
    turn (30)
    i = i + 1
  }
}
```



Try this to ensure that it is drawing 12 squares.

The next change also change the size of the squares with each iteration within the **demo()** function.

```
function demo () {
  clear()
  var i = 0
  while ( i < 12) {
    color (i)
    square3(i * 10)
    turn (30)
    i = i + 1
  }
}
```



Great. You may have noticed something funny. Both functions use the same variable name **i** for the iterator. Don't they conflict? No. Both are defined as local variables, which means they only have meaning within the local context of the function in which they are defined. If **i** had been defined at the global level as in:

```
var i = 0;

function square4(side){
 ...
}

function demo(){
 ...
}
```

both functions would access the same variable and this would make the routines much harder to debug because both functions would be changing the value of **i**. In general it is better to use local variables than ones defined globally for several reasons:

1. the same name can be used without conflict.
2. only the code within the function can access the variable

Sometimes globals are necessary when you need to share data between functions or when a variable needs to last longer than just the time that a function is called (a property called *persistence*).

Variables default to global, so be sure to define them locally by including the **var** keyword.

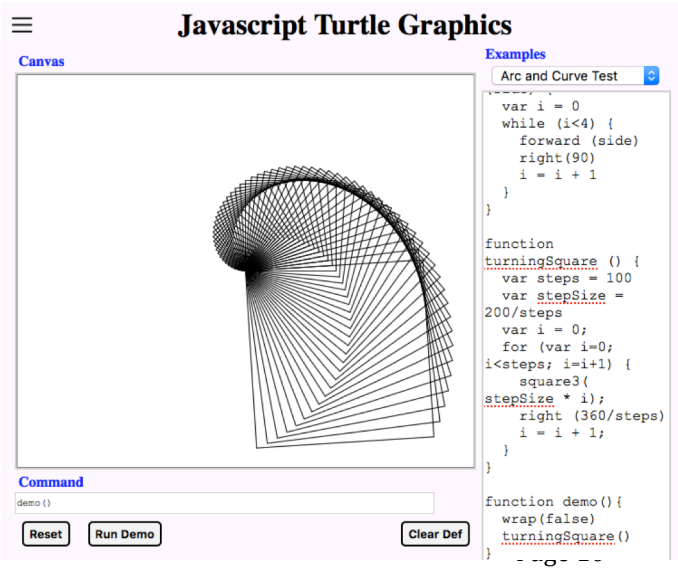
This also introduced the **color()** function. It works with simple colors, if you use a number between 0 and 15. This works great in this case.

Review a bit
we have

- defined functions
- called or invoked those functions
- defined local variables
- learned a bit about global variables
- set and accessed variables
- done iteration to repeat something a number of times

Try the preceding example or a turning square with more and smaller steps.

```
function turningSquare () {
  var steps = 100
  var stepSize = 200/steps
  var i = 0;
  while ( i < steps) {
    square3( stepSize*i);
    right( 360/steps)
    i = i + 1;
  }
}
```



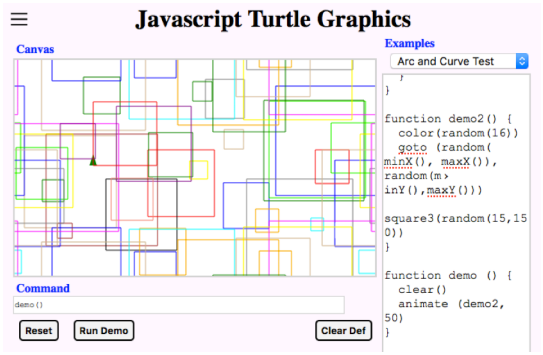
Let's try to place random-colored, random-sized squares at random places on the canvas. It sounds like we need a random number generating function. Random() fills the need. This has two forms: one with one number and one with two numbers. The single number form that generates an integer between 0 and the value supplied. The two number form that generates an integer between the two numbers.

Random colors can uses numbers between 0 and 15, so random (15) will work for a random color.

To position the start of the square anywhere on the canvas, the **goto(x,y)** function can be used. A Cartesian coordinate system is used with x=0, y=0 at the center of the canvas. (see example of Cartesian coordinates.) The values at the edges of the canvas will vary from machine to machine and with the size of the particular window. There are functions to retrieve the minimum and maximum X values, **minX()** and **maxX()** respectively. **minY()** and **maxY()** do the same for the Y values.

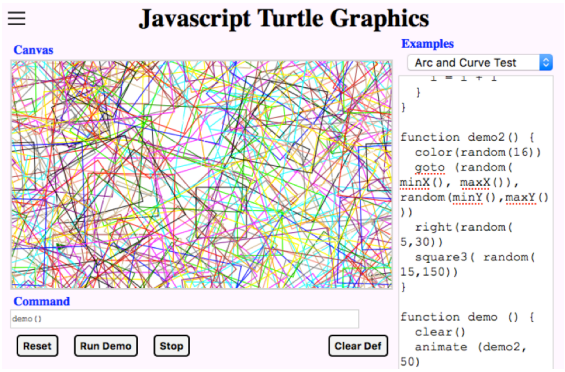
The last function to introduce is the animate function. It just repeated calls the function named in its first parameter after a delay of the number of milliseconds (1/1000 second). The program can be stopped by pressing the **Stop** button.

```
function demo2() {
  color(random(15))
  goto (random( minX(), maxX()),
    random(minY(), maxY()))
  square3( random( 15,150))
}

function demo(){
  clear()
  animate( demo2(), 100)
}
```



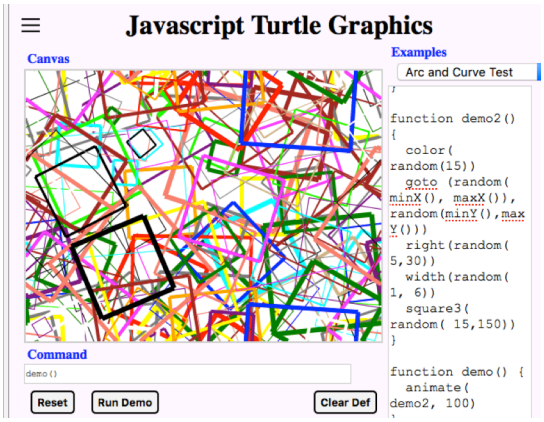
The squares can be rotated as well by throwing in a turn (random(something)) into the mix.

```
function demo2() {
  color(random(15))
  goto (random( minX(), maxX()),
      random(minY(), maxY()))
  right(random( 5, 30))
  square3( random( 15,150))
}
```



The line widths of squares can also be varied by throwing in a **width (random(**something**))** into the mix.

```
function demo2() {
  color(random(15))
  goto (random( minX(), maxX()),
      random(minY(), maxY()))
  right(random( 5, 30))
  width(random( 1, 6)
  square3( random( 15,150))
}
```



We can generalize the square function to make it generate polygons.

What additional parameter(s) is(are) needed?

Can you calculate the corner angle?

What about a five-pointed star? It is nearly the same problem as the polygon. Both end up in the same place and same direction as the start.

In a polygon, how many times to you go around the center point.

How does that affect the calculation of the angle?

In a five-pointed star, how many times do you go around the center point?

Can you use that number to calculate the angle that you need to turn?

Can you generalize this?

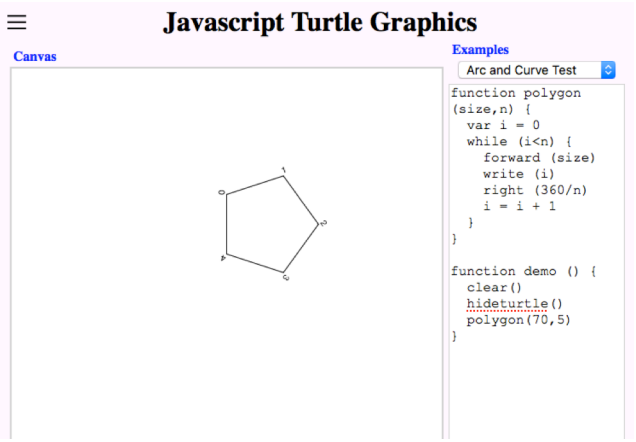
Are there exceptions?

Is there a maximum?

The answers are on the next page if you get stuck, but try to solve these without looking.

For the square, take 360 degrees, divided by 4 gives 90 degrees for each angle. For a triangle, take 360 degrees, divided by 3 gives 120 degrees for each angle. Generalizing for polygons, take 360 degrees and divide by the number of sides to get the angle.

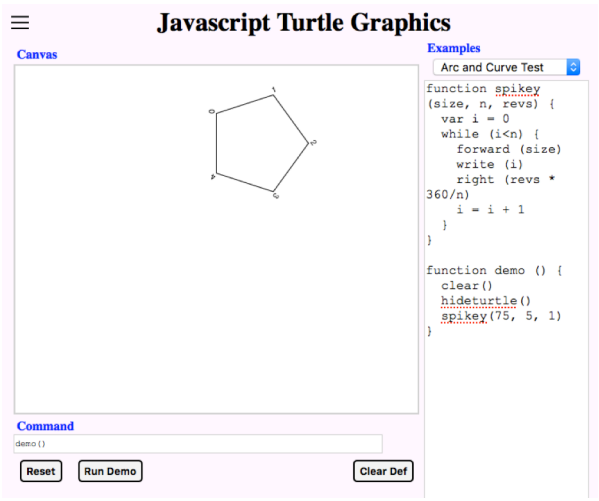
```
function polygon (size,n) {
  var i = 0
  while (i < n) {
    forward (size)
    write (i)
    right (360/n)
    i = i + 1
  }
}
```



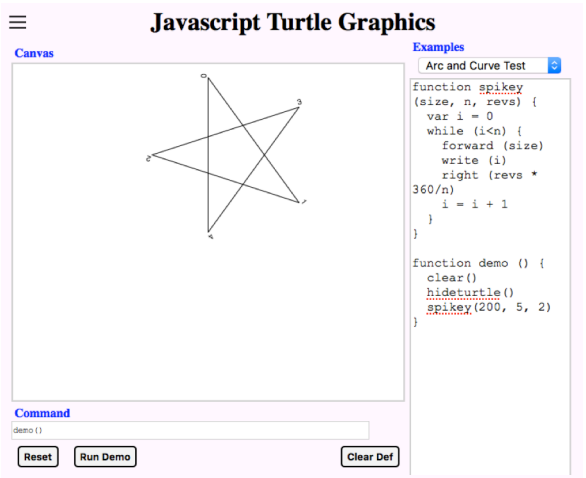
Now lets try to draw a five-pointed star. When you draw the star free hand, you go around the center point twice and you make 5 turns. You can use that to determine the angle. 2 * 360 /5 = 240.

The limit to the number of times that you can go around the center point is ***n/2***. Going around more times, produces the same result as a lower number, but the resulting figure is reflected. If the number of points is divisible by the number of revolutions, the figure will close prematurely. So ***n*** works best if it is prime. With the number of revolutions close to ***n/2***, the figure is most spikey. As ***n*** gets smaller, the opening in the center gets bigger and bigger.

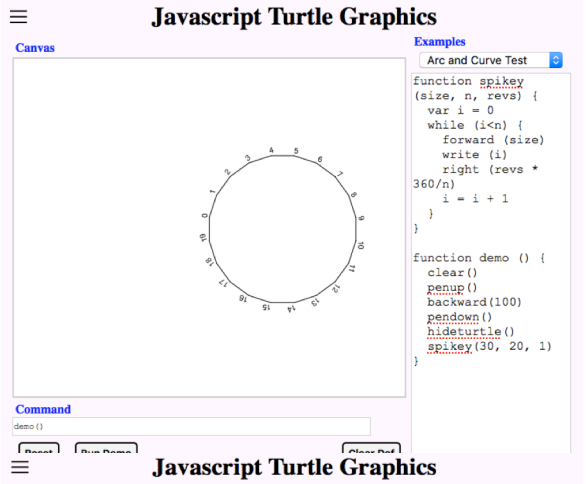
```
function spikey (size,n,revs)
{
  var i = 0
  while (i < n) {
    forward (size)
    write (i)
    right (revs*360/n)
    i = i + 1
  }
}

function pentagon (size) {
  spikey (size, 5, 1)
}
```



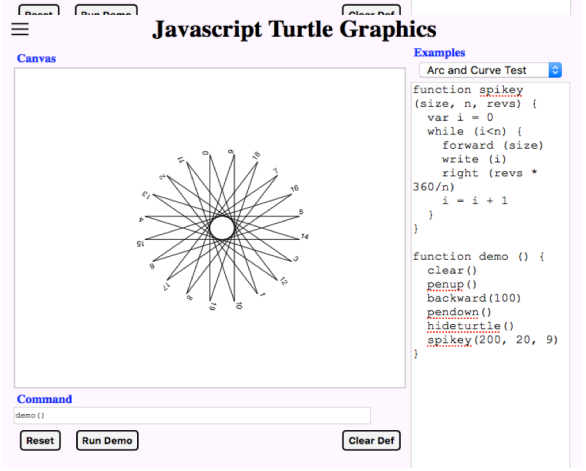
```
function star (size) {
  backward (size/2)
  spikey (size, 5, 2)
}
```




```
function polygon (size, sides) {
  spikey (size, sides, 1)
}
```




```
function starN (size, points) {
  backward (size/2)
  spikey (size, points,
      Math.floor(points/2) )
}
```

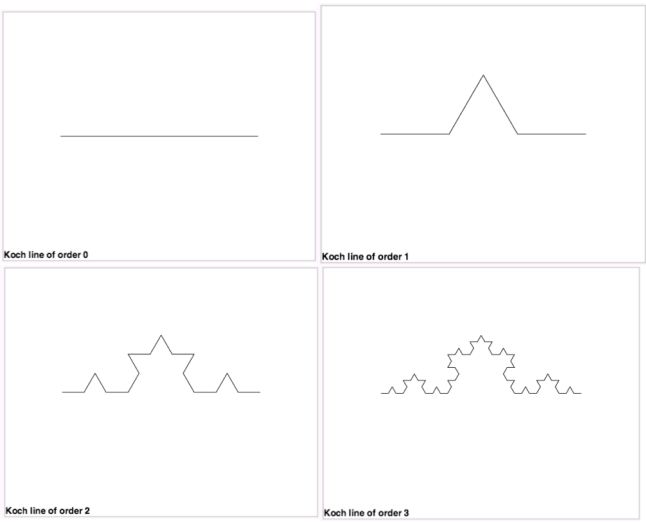


Other variations…

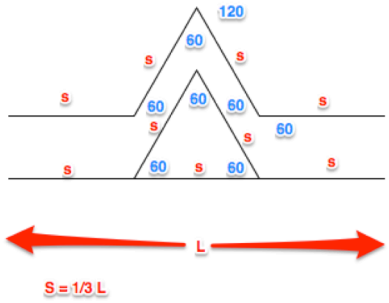
```
//spikey( 200, 41, 20)
//spikey( 200, 41, 20)
//spikey( 200, 47, 23)
//spikey( 200, 51, 25)
//n must be not be divisible by
revs
//revs is best at about n/2
```

Koch Line Fractal

Fractals are a geometric shape that looks similar at any resolution. This is like a branching tree or the streams flowing into a river. We'll look at two simple fractals and then try to generate them with turtle graphics.



A simple fractal is the Koch line. Basically you take a line and divide it into three equal parts. Make equilateral triangle with the sides equal to the length of the three segments. Replace the middle segment with the side of the triangle. Remove that middle segment. The details are in the figure below.



Since this fractal is made of pieces that look the same, this is a good place to use a *recursive* function, which is a function that calls itself. The function either draws a straight line or a bumped line. If it draws a bumped line, the line segments may either be straight or bumped depending on the order desired.

The first key of a recursive function is that it has to limit how many times it calls itself, otherwise it would just be an infinite loop. In this case, the function can stop either when the line gets too short to bump out or after a certain number of times. The latter is easier to implement in this case and we can do some other things with the routine, so lets use that method.

Lets call a straight line, order 0, and lets call the bumped out line order 1. Basically we want to be able to divide and bump out the line to any order up to the resolution of the screen.

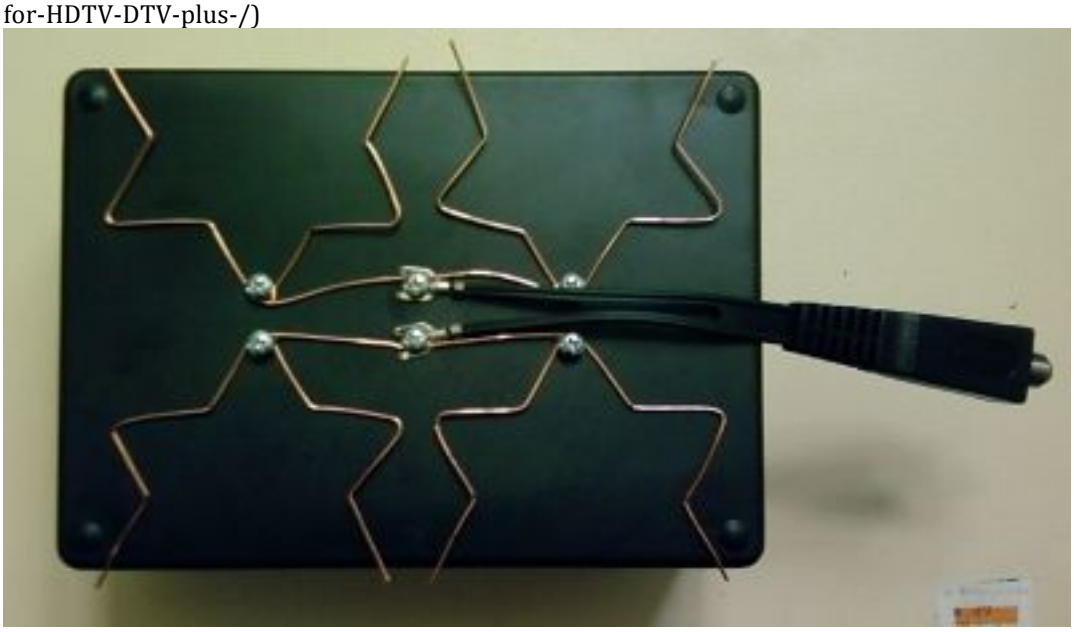
So your function has two inputs: order and length. If order is 0, just draw a straight line and quit. Otherwise, divide the length into three parts and replace the middle part with a bumped out line. When you draw the bumped out line, don't just draw it directly, but let the recursive function draw the line for the next order lower. So an order 1 figure would draw 4 order 0 lines. An order 2 figure would draw 4 order 1 lines, each of those lines consisting of 4 order 0 lines. And so on.

For more fun, draw a Koch star that is an equilateral triangle of Koch lines.

Use the animate () function to draw the figure for various orders with a pause between each order. You may want to use a global variable as the iterator.

A Koch line has practical uses. It was recently discovered that an antenna made with a fractile has some very interesting properties: it was more compact and it was less directional. That made it a prime candidate for modern cell phones and other electronic devices. You can make your own TV antenna using Koch lines (see http://www.instructables.com/id/How-to-make-a-fractal-antenna-

for-HDTV-DTV-plus-/)

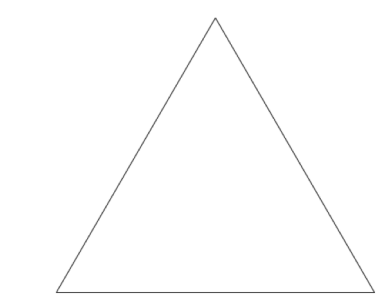


Sierpinski Triangle Fractile

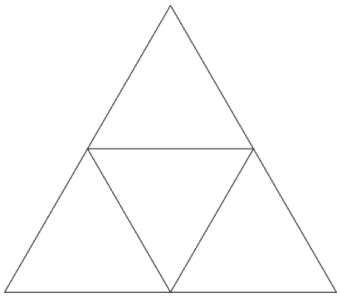
Sierpinski came up with several fractals. One of them is the Sierpinski triangle shown to the right.

Again this is a candidate for a recursive routine. It would either draw a simple triangle (order 0) or a triangle that is subdivided into three triangles (which in turn may be simple or further divided depending on the order desired).

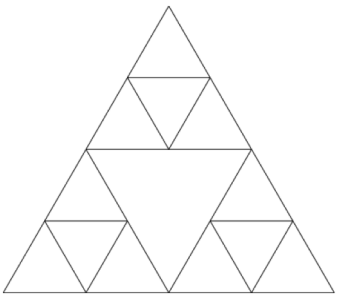
Invariance is a good property to keep in mind as you work through this example. Invariance means that the external settings (properties) remain the same before and after the function call. The external settings could be color, position, angle, line width, etc. for turtle graphics.



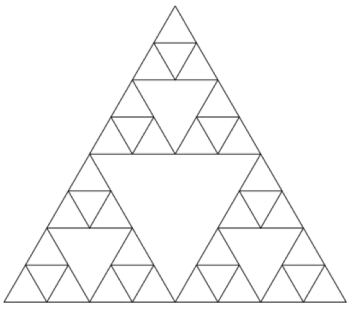
Sierpinski triangle of order 0



Sierpinski triangle of order 1



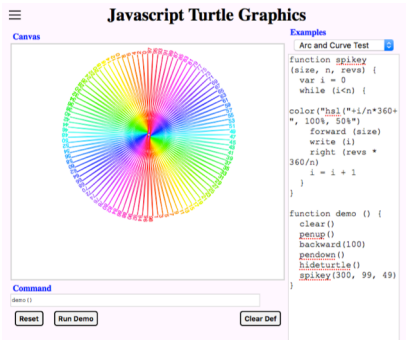
Sierpinski triangle of order 2



Sierpinski triangle of order 3

What is the Next Step???

- Learn about the **for()** loop instruction
- Set up a demo of finding pi with a random number generator. Hint: use a square that is 1 unit by 1 unit and a quarter of a circle with a radius of 1 unit. Remember the Pythagorean theorem.
- Play with random colors or color around a color wheel, hint: **color (random(15))** or **color("hsl("+i/n*360+", 100%, 50%")**
- Investigate other fractals and draw them
- Investigate tessellations and draw them
- Do an animated graphics demonstration
- Make the page web accessible
  - add to a server, perhaps on a Raspberry Pi with Apache.



- Learn more about JavaScript, HTML, and CSS using resources:
  - Read a book from it-ebooks.info, like JavaScript for Kids
  - Take a JavaScript course from Khan Academy
  - Get hands on experience with Code.org
  - Find a particular feature at W3School
- Learn about code development tools
  - Browser based debugging tools
  - "lint" programs to check CSS and HTML syntax
  - "minify" programs to make your final code smaller