Chapter 5, JavaScript Functions

John M. Morrison

September 27, 2017

Contents

0 Introduction

Our next object of study is the JavaScript function, which will allow us to store a procedure under a name, and if we wish, return an output from that procedure.

JavaScript has built-in functions. We have met the `Math` object which has all manner of nice scientific calculator functions. Two other very nice examples are `parseInt` and `parseFloat`. These convert a string containing a number into a number. The `parseInt` function is good for whole numbers, and the `parseFloat` function is good for numbers in floating-point notation. Here we demonstrate their action.

```
> x = "123"
"123"
> y = parseInt(x)
123
> typeof x
"string"
> typeof y
"number"
> x = "4.133"
"4.133"
> y = parseFloat(x)
4.133
> typeof y
"number"
```

Now comes the interesting part.

```
> typeof(parseInt)
"function"
> typeof(parseFloat)
"function"
```

Functions are JavaScript objects. The name of a function is just a variable name. We will do two things here. We will make our first function, then show that you can assign it.

```
> function f(x){return x*x;}
undefined
> f(5)
25
> cows = f
f(x){return x*x;}
> cows(5)
25
```

So functions are objects, just as are numbers, strings, and booleans. Functions allow us to remember a set of instructions and compute an output just by invoking them. You can easily imagine that if we do a multi-step procedure over and over again in a program, we can shorten that program by "wrapping" that procedure in a function. This measure de-clutters your code and makes it more readable if you give the function a name evocative of its purpose. When we use a function, we are said to be *calling* it. Information you put inside of the parentheses is a comma-separated (possibly empty) list of items called *arguments*.

Here is a peek at what is to come. We will see that functions are very helpful when we start doing things like placing buttons on a web page that cause actions.

You put the action in a function, and when the button is pressed, the function is called and its procedure is carried out. Functions can call other functions, so a single call to a function can launch a complex sequence of actions.

1 Three Critical Components

A function allows you to store a procedure under a name in response to zero or more inputs. There are three important parts of procedures.

- **Inputs** When you create a function, you need to be aware of the types of inputs it makes sense for the function to handle.
- **Outputs** You can return one JavaScript object from a function via the `return` statement.
- **Side Effects** These are things that are left behind after a function does its work. For example, if you use `console.log("foo");` the string `"foo"` will be placed in the console. This remains after your function is finished running. A function can alter content on a web page. Another possible effect of a function is that it changes something in the global symbol table.

If you are making a function, you should think about all three of these components. We will see all of these features at work in the upcoming sections.

2 Writing JavaScript Functions

When you create a JavaScript function, you should make notes called *comments* about it. The audience for these comments is yourself or another programmer. Your comments will tell what the function does. This way, someone using a complex function you might write can use that function without puzzling though and trying to understand the intricacies of your code. You should describe the input(s) for your function, any return value (output), and any side effects that the function has.

Let us create a very simple, somewhat artificial, example. It will accept a string and give back its last character. Here is a shell for writing a function.

```
lastChar = function(s)
{

}
```

Whoa! Notice that the line `lastChar = function(s)` has no semicolon at the end of it! Why is that? This line is an example of a *boss statement* called a `function header`. Immediately after a boss statement you will see a *block* of code; this is just a sequence of statements enclosed in curly braces, `{ ... }`. The boss statement "owns" its block of code. You should read `lastChar = function(s)` as "to define `lastChar(s)`," Notice that this is a grammatically incomplete sentence, because it is really a half-done assignment, like `x = `.

You will soon meet other types of boss statements. The all have the feature of owning a block of code.

If you put a semicolon after your boss statement, you will decapitate it and chop the body off of the function header. Then you will think you are Henry VIII.

The language keyword `function` says, "we are going to define a function here." The word `lastChar` is the name of the function. Function names are variable names, so the naming rules for variable names apply to functions as well; after all, you see that we are doing an assignment here. After this you see `(s)`; this function expects one object as an *argument*; this is the input. You can have several items inside the parentheses; just separate them with commas; you can also have none. This list is called the *argument list* for the function. We begin by showing you a function with no arguments. This function could return a value or print something to the console. As of now, it does nothing.

```
noArguments = function()
{

}
```

Since we are the maker of this function, we can include directions for its use. We now show a typical way of doing this. All of the text in this program, beginning at `/*` and ending at `*/` is a comment and it is ignored by the browser executing the JavaScript.

```
/*
* precondition:  s is a nonempty string
* postcondition: returns the last character of s.
* This function has no side-effects.
*/
lastChar = function(s)
{
    var n = s.length;
    return s.charAt(n - 1);
}
```

We can test our shiny new function by typing it into a console. We will omit the comments in the console. Soon, you will put your functions in a file that we can easily include on an HTML page; the comments belong in this file. Note the use of the `var` keyword. Always use this when creating a variable.

Here is what to type in the console. You have to do it in one line like so.

```
function lastChar(s){var n = s.length; return s.charAt(n - 1);}
undefined
```

The `undefined` is a normal occurrence. JavaScript has placed this function in memory and it just says `"undefined"`. Now we test it.

```
> lastChar("fooment")
"t"
> lastChar("12345")
"5"
```

Now let us violate the precondition and use our function illegally.

```
> lastChar("")
""
```

This outcome might or might not be desirable to you. Here is something worse. This function makes no sense whatsoever for numbers.

```
> lastChar(123)
VM5531:1 Uncaught TypeError: s.charAt is not a function()lastChar
    @ VM5531:1(anonymous function) @ VM5556:1
```

This is a domain error. Hence, your restriction that the input be a string.

The preconditions for a function are things that should be true before a function is called. As we just saw in the last example, among these are the types of arguments that the function will accept. Here we stipulate that the argument must be a nonempty string. If the user of our function violates the preconditions, it's *caveat emptor* for the impetuous fool.

The postconditions tell what is true when the function's execution ends. On the second line we see a `return` statement. This indicates we are sending back to the caller, or user, of this function, the last character of the string passed in. This style of writing comments is readily understood by other programmers who can learn what your function does without having to puzzle through the code accomplishing the task.

A Useful Shorthand You will often see functions in JavaScript in this style. It is just a variant on the other syntax.

```
/*
* precondition:  s is a nonempty string
* postcondition: returns the last character of s.
*/
function lastChar(s)
{
    var n = s.length;
    return s.charAt(n - 1);
}
```

This is identical to using the header

```
lastChar = function(s)
```

JavaScript has two types of comments. Comments are ignored by the browser, but are useful to someone using your functions. One type is the inline comment, which you have seen just before the code for `lastChar`. It looks like this.

```
/*  the star slash begins an inline comment, which can go on
for many lines.  It is ended with a slash-star like so:  */
```

The other type of comment is a one-line comment. It looks like this: `//`. It causes all text after it on a line to be ignored.

When you do something complicated, it's a good idea to explain it in comments so another programmer can quickly understand what you did.

3 Making and Including JavaScript files

We will demonstrate this using our `lastChar` function. Begin by creating a file named `myFunc.js` and place this code in it.

```
/*
* precondition:  s is a nonempty string
* postcondition: returns the last character of s.
*/
function lastChar(s)
{
    n = s.length;
    return s.charAt(n - 1);
}
```

In the same directory, make a copy of your index file and name it `functionDemo.html`. Then modify it to look like this

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8"/>
<title>JS Function Demonstration</title>
<script type="text/javascript" src="myFunc.js">
</script>

</head>
<body>
<p id="showJS">Your JavaScript will appear here.</p>
<script type="text/javascript">
    document.getElementById("showJS").innerHTML = lastChar("fooment");
</script>
</body>
</html>
```

Let us describe what happens in detail.

1. The `html` element opens.
2. The `head` element opens.
3. The `title` element opens, the title is placed in the tab's title area, and the `title` element closes.
4. The `script` tag opens, the script is read into memory, and the `script` tag closes.
5. The `body` element opens.
6. The `<p>` element is constructed and the text inside is placed on the page.
7. The `script` tag in the body opens, your script is read into memory, and then this `script` tag closes.
8. The script runs, replacing the text with the last character of `"fooment"`.
9. The body closes, then the document ends and it is fully loaded.

Observe that we did not place the second script tag in the head of the document, because the paragraph element it modifies does not exist when the head of the document is being read. You should do exactly this. Then try to load the page; you will see the text `"Your JavaScript will appear here"`. Something went wrong! To find out, open the console from this page. You will see error messages describing the problem.

Pages, in the absence of other information, load from top to bottom. One lamentable thing you see here is that we have JavaScript code mingled with HTML. Let us now work to remedy this.

You can put many functions in a single JavaScript file, and include them in this manner we have seen here. So you see that JavaScript can be linked onto a page in a manner not dissimilar to that of linking in style sheets. Most JavaScript on a web page is read in from external files.

Programming Exercises

1. Put a semicolon at the end of the boss statement `function lastChar(s)`. Load your page and open the console. See what happens. Does the console hiss? Does this error pass unmentioned?
2. Misspell the name of your `.js` file. Open the HTML page and then the console. What error message do you see?
3. Add another function to `myFunc.js` and try calling it from the script at the end of the file.
4. Place the code in the script at the end of the file in a file called `do.js`. Now in place of the code and the existing `script` tag, add the line

   ```
   <script type="text/javascript" src="do.js"></script>
   ```

   This will work the same as the original program.
5. Create a file named degrees.js. In it, create a function named `sinDeg(x)` that takes a number as an argument and which returns the sine of `x` degrees.
6. In the same file create `cosDeg` and `tanDeg` that are degree versions of the cosine and tangent functions.
7. In the same file, create functions `asinDeg(x)`, `acosDeg(x)`, and `atanDeg(x)` that produce angles unitted in degrees for the inverse trigonometric functions.

4 More about Symbol Tables

Critical to understanding JavaScript is understanding what is visible when. We are now going to become stricter about grammar. This will give us more precise control over our programs, prevent unintended goofs, and make our code more efficient.

Now refresh the page with your console; this will reset the console. Make this interactive session.

```
> function f(x){y = x*x; return y;}
undefined
> f(3)
9
```

```
> y
9
```

Now, if you have any sense of decency, you will blanch at the lack of modesty exhibited here. Next, let's see what happens when we attempt to show `x`.

```
x
VM8419:1 Uncaught ReferenceError: x is not defined
    at <anonymous>:1:1(anonymous function) @ VM8419:1
```

Ooh, a nastygram from the browser. Now it's time to discuss the rhyme and reason behind this seemingly capricious behavior. To really understand this, a more detailed discussion of symbol tables is required.

Any time a variable is assigned, an entry to that effect is made in a symbol table. When your program first begins to run, the `global symbol table` is visible. In fact, it is visible for the entire lifetime of your program. It contains the `Math` object, all of the methods for strings, as well as built-in functions such as `parseInt` and `parseFloat`.

Any variable or function you use must be in some visible symbol table. Now let us look at the execution of the function `f` we have created here. There are two ways we can write this function; the other way looks like this.

```
> f = function(x){y = x*x; return y;}
undefined
```

This second grammar shown here is closer to the reality of the process. The right-hand side `function(x){y = x*x; return y;}` is just a means of creating a function object. The assignment attaches that function object to the name `f`.

This function is usable because, when we create it, it is inscribed in the global symbol table. It is visible from the time it is created until the web page it is linked in or present on (the enclosing page) is clicked away or left by the browser.

Now let us talk about `y`. We asked the console for the value of `y`, and it showed `y`. This behaviour is undesired for the same reason that if you are going to school or work, you are asked to have your clothes on. Really, there is no use for the variable `y` outside of the function `f`.

This is the reason we use the `var` keyword. By so doing, if you create a variable inside of a function, it is no longer visible once the function's execution is over. We say that such a variable has *function scope.*

If you do not use the `var` keyword, you will pollute your global symbol table with all sorts of unnecessary items. This can cause confusion and unexpected problems. It is best to obey the this principle

The Blofeld Principle This is named after the Bond villain, Ernst Stavro Blofeld. He got rid of his henchmen once they "outlived their usefulness." The greedy industrialist Mr. Osata learned this to his sorrow on a great scene in *Diamonds are Forever*. We crib from his book and say: *Never let variables outlive their usefulness.*

There is a second keyword, `let`, which is new to ECMA6. Here is how it works. A variable's *block* is the code inside of the closest set of enclosing curly braces around the variable. A variable created using `let` is born when it is created and dies when its block ends. You will subsequently see that functions can contain blocks of code, and when that occurs, you will see the usefulness of `let`. Note: it is an error to create a (global) variable using `let` that is outside of a function.

When we use `var` or `let` to create a variable inside of a function `f`, we say that this variable to be *local to* `f` Without the `var` keyword, the variable `y` gets inscribed in the global symbol table, cluttering it unnecessarily. If you have lots of functions active on a page, the global symbol table gets mighty crowded. And, if you accidentally create two symbols in the global symbol table of the same name, you can have a terrible problem. As a result, we recommend that, when creating a variable inside of a function, that you use the `var` keyword. Here is our program with the `var` keyword included.

```
function f(x)
{
    var y = x*x;
    return y;
}
```

Put this in the console and you will see this error message.

```
> function f(x){var y = x*x; return y;}
undefined
> f(5)
25
> y
VM11190:1 Uncaught ReferenceError: y is not defined
    at <anonymous>:1:1
```

We show our function `f` using the `let` declaration.

```
function f(x)
{
    let y = x*x;
    return y;
}
```

Putting this in to the console triggers the same error, `y` is invisible outside of `f`. Since this function has no interal blocks, `var` and `let` exhibit the same behavior.

Also notice this.

```
> x
VM11193:1 Uncaught ReferenceError: x is not defined()(anonymous function) @ VM11193:1
```

Notice that the parameters, or arguments of a function have function scope. That is why you could not find the value of `x` outside of `f`. Local variables are invisible outside of the functions that contain them. Let us now go into detail as to what happens when the function `f` is called.

1. `f(5)` got called.
2. The value `5` is passed to the parameter `x` and the entry `x -> 5` is entered into `f`'s private symbol table. At this time the global symbol table and `f`'s private symbol table are both visible.
3. The expression `x*x` is evaluated to `25`.
4. The variable `y` is created value `25` is bound to to it. This puts the entry `y -> 25` into `f`'s private symbol table.
5. The statement `return y` does three things
   (a) The value 25 is fetched from the symbol table; now we execute the statement `return 25;`
   (b) The `return` statement ends the function's execution. The private symbol table for `f` becomes inaccessible.
   (c) The value `25` is sent back to the caller (the console here)
   (d) Execution in the console resumes right where it left off and the `f(5)` is replaced with `25`. This is made possible by a breadcrumb called the `return address`, that knows exactly where the caller left off.

At first you might think that it would be convenient to have every variable ever created visible in the local symbol table under the theory that "more is better." More is often not better. One giant gloppy bacon cheeseburger tastes great. After six, seven, or eight, you are on a one-way ride to dyspepsia city.

The Blofeld principle reinforces this. A good rule of program design is: *Keep the number of visible symbols as small as possible.* This has three benefits.

1. We do not have to keep track of vast pools of visible variable names.
2. We do not have to worry about conflicts caused by assignment to a global variable affecting that variable in another function.
3. One function cannot interfere with the inner workings of another.

Once we create a function the goal is to be able to only think about what the function does, not how it does it. The last thing we want to think about is a local variable inside of a function, just as you don't want to say, "Hey ABS, activate now!" when a car comes to a sudden stop in front of you. You just stomp on the brakes.

5 Yes, This Pays Off: Making a Page Respond to a Button Click

This section will serve as an aperitif to the world of event-driven programming that makes web pages interactive. This will comprise our next major area of study.

First of all, you might ask: *What is an event?* Events convey information about user input or internal browser behavior. You are about to learn about the `button` element. When a button is clicked in HTML, it broadcasts a `click` event to your web page. You will give the button an `onclick` attribute, which will tell it what to do when it gets clicked. The `click` event is generated by the user of your page.

One great reason for functions is that the can be called in response to user action on a page, which triggers events. We will now make a our first truly interactive web page.

You will have an opportunity to add features to this little program. Let us begin with a simple HTML page. We make a paragraph element with an ID in it. Place this in file named `buttonDemo.html`.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8"/>
<title>JS Button Demonstration</title>
</head>
<body>
<p id="showJS">Your JavaScript will appear here.</p>
</body>
</html>
```

We introduce a new HTML element, `button`, which (surprise), creates a button. The text element inside of the `button` element will be the label on the button. Now let's add it.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8"/>
<title>JS Button Demonstration</title>
</head>
<body>
<button>Click Me Now!</button>
```

```
<p id="showJS">Your JavaScript will appear here.</p>
</body>
</html>
```

Open the page in the browser window and open your console. You will see a button on your page, which does nothing. Remember, page behavior comes from JavaScript. Next, we will create some JavaScript in a program named `respond.js`.

```
var clickCount = 0;
function showText()
{
    clickCount++;
    var out = "You have clicked " + clickCount + " times.";
    document.getElementById("showJS").innerHTML = out;
}
```

You can see some new features here. On the first line we create a variable named `clickCount` and bind a value of `0` to it. When we link this JavaScript to our page, that variable `clickCount` will be visible in any script on the page. Next, we have a function `showText()`. On the first line of the function you see a new construct: the autoincrement operator. This is a postfix operator that adds 1 to the variable that is its operand. So the first time this function is called, `clickCount -> 1` will be an entry in the global symbol table.

On the next line, we create a string `out` that will be placed on our page. The last line grabs the element with ID `"showJS"` and replaces its inner HTML with the string `out`. This button will tell how many times it has been clicked.

If you refresh the page, all is set back to its original state.

Before this JavaScript can work, we must link it to our HTML page. In the head of the page, add this `script` element.

```
<script type="text/javascript" src="respond.js"></script>
```

We then have to make the button react to the script. For this, there is the attribute `onclick`. Modify the `button` tag as follows.

```
<button onclick="showText();">Click me now!</button>
```

Load your page and click away!

Now let us go through this page and give blow-by-blow coverage of exactly what happens.

1. The line

```
<!DOCTYPE html>
```

says “Browser, this is an HTML5 document.”

2. The line

   ```
   <html>
   ```

   says, ”Browser, this is the start of the document.”

3. The line

   ```
   <head>
   ```

   opens the head of the document.

4. The line

   ```
   <title>JS Function Demonstration</title>
   ```

   says, ”Browser, the title of this document is ’JS Function Demonstration,’ and in response, the browser puts that title in the tab’s title area.”

5. The line

   ```
   <meta charset="utf-8"\verb|/|>
   ```

   says, ”Browser, this document is written in ASCII characters.”

6. The line

   ```
   <script type="text/javascript" src="respond.js"></script>
   ```

   says ”Browser, load the script `respond.js` into memory.” Note that the function and the variable are placed in memory. The function takes no action until it is called.

7. The line

   ```
   </head>
   ```

   says “end the `head` element.”

8. The line

   ```
   <body>
   ```

   says “start the `body` element.”

9. The line

   ```
   <button onclick="showText();">Click me now!</button>
   ```

   says “Create a button with the label ’Click me now’ on it. Also, when the button is clicked, execute the script `showText();`.”

10. The line

    ```
    <p id="showJS">Your JavaScript will appear here.</p>
    ```

    creates a paragraph element marked with the `id showJS`.

11. The line

    ```
    </body>
    ```

    ends the `body` element.

12. The line

    ```
    </html>
    ```

    ends the document.

Programming Exercises In these exercises you will modify `buttonDemo.html` and `respond.js` to give new functionality to this page.

1. In the HTML file, change to button's label to "Add 1."
2. In the HTML file, make a new button with the label "Subtract 1."
3. In `respond.js`, change the name of the function `showText` to addOne, leaving its innards untouched.
4. In `respond.js`, Make a new function named `subOne` that lowers the value of `clickCount` by 1. Do you think there is a `--` operator similar to `++`? Experiment and find out!
5. In the HTML file, insert an appropriate `onclick` attribute for the new button.
6. Test: When you click the add button, the click count should increase by one. When you hit the subtraction button, the click count should decrease by 1.

6 Introducing the `canvas` Element

HTML5 introdced the `canvas` object, which is a surface we can draw on using JavaScript. To make this work, we will avail ourselves of the `load` event; this event is broadcast when a page has loaded.

Begin by creating these three files. First, make `draw.html`

```
<!doctype html>
<html>
<head>
<title>draw</title>
<meta charset="utf-8"/>
<link rel="stylesheet" href="draw.css"/>
<script type="text/javascript" src="draw.js">
</script>
</head>
<body>
</body>
</html>
```

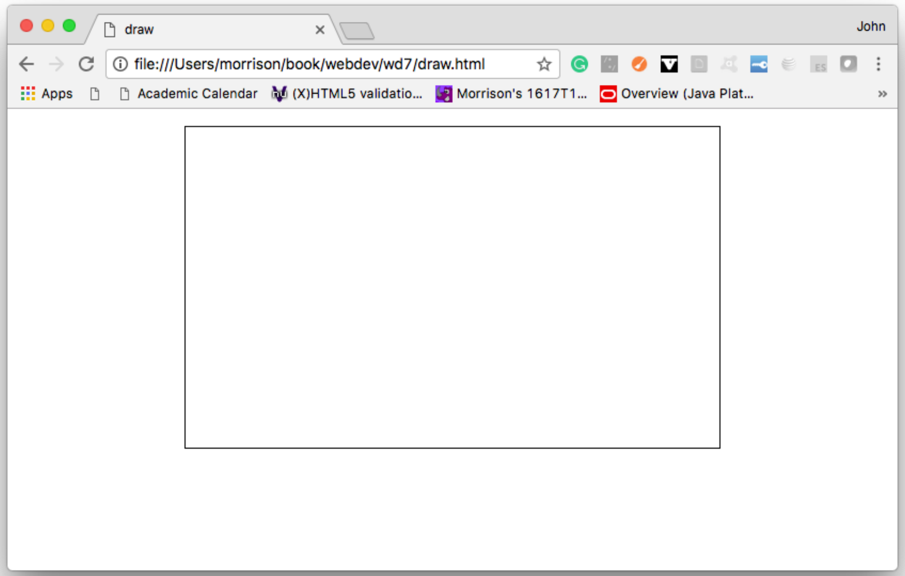
Next, `draw.css`.

```
/*Author: Morrison*/
h1, h2, .display
{
    text-align:center;
}
canvas
{
    border:solid 1px black;
    background-color:white;
}
```

Now modify `draw.html` by adding the canvas element and by giving the body an `onload` attribute.

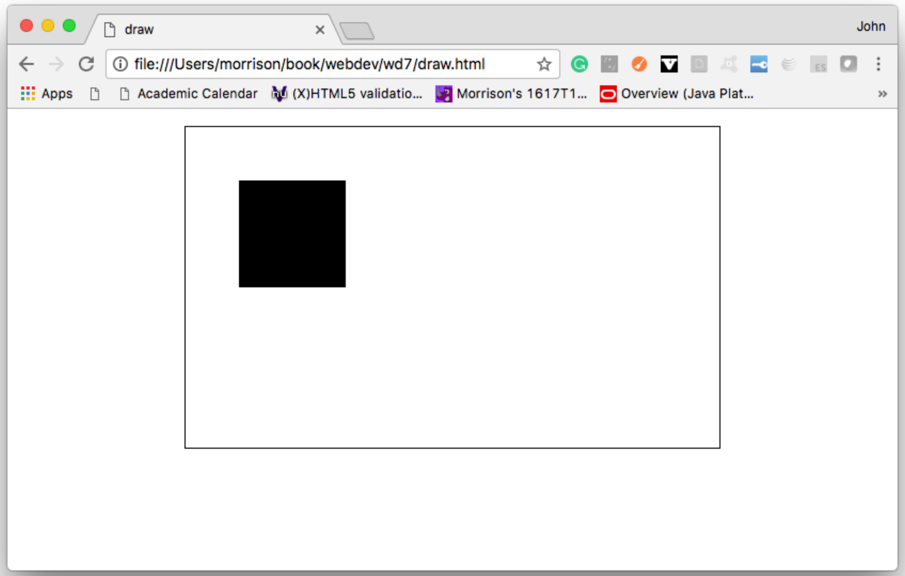
```
<!doctype html>
<html>
<head>
<title>draw</title>
<meta charset="utf-8"/>
<link rel="stylesheet" href="draw.css"/>
<script type="text/javascript" src="draw.js">
</script>
</head>
<body onload="init();">

<p class="display">
    <canvas height="500" width="300" id="drawPanel">
        Get a modern browser!
    </canvas>
</p>
</body>
</html>
```

There are some mysterious items here. The `onload` attribute will ensure tha the function `init()` which we shall define in our JavaScript file will not be called until the page is loaded. The text inside of the `canvas` element is only displayed if the client has an antiquated browser that does not support the `canvas` element. YOu should specify the dimensions of a canvas element as attriubtes in HTML; do not do this in CSS. You need to give the canvas element an id so that JavaScript can manipulate it. We have placed the canvas inside of a paragraph that will center it on the page. Open the file `draw.html` with your browser and you will see this.



Now let us create the file `draw.js`

```
function init()
{
    var c = document.getElementById("drawPanel");
    var g = c.getContext("2d");
    g.fillRect(50,50,100,100);
}
```



So, how did this happen? First the document loads; the calll to `init()` is deferred until the entire page has loaded. Therefore the line

```
var c = document.getElementById("drawPanel");
```

executes wihtout error since the element is in place when the function `init()` is called. The next line

```
var g = c.getContext("2d");
```

gets the graphics pen associated with the `canvas` element. You will use this pen to draw. The next line

```
g.fillRect(50,50,100,100);
```

fills a retangle with the default pen color (black). You can experiment and chenge the parmeters passed to `g.fillRect`. The canvas has a coördinate system: the origin is at the upper left hand corner. The x-coördinate behaves just as it does in math: it gives the vertical position from the left-hand side, unitted in pixels.

The y-coördianate behaves differently: the y-axis points DOWN! The second pair of numbers gives the size of the rectangle to be drawn in units of pixels.

Programming Exercise These are important! Don't skip them!

1. Add then line `g.fillStyle = "blue";` and draw another rectangle in a different spot. What happened? Now try this with a color hex code preceded by a `#`.
2. Draw some overlapping rectangles in the canvas of various colors. What happens?
3. Now try using `g.strokeRect()` in the same manner as you used `g.fillRect()`. What happens?
4. Change colors with `g.strokeStyle()`. What does this do?

6.1 Drawing Paths

You can "connect the dots" by creating a path. Let us begin by making a new set of files. Here is `path.html`

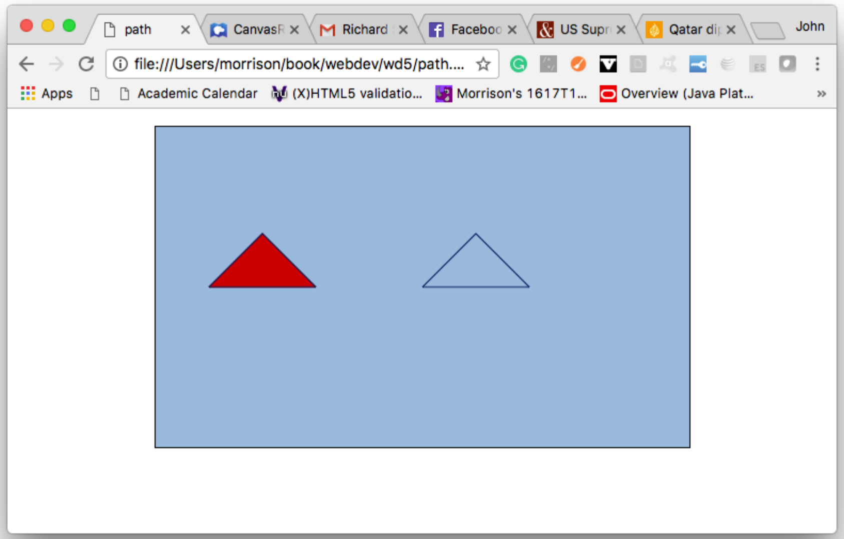
```
<!doctype html>
<html>
<head>
<title>path</title>
<meta charset="utf-8"/>
<link rel="stylesheet" href="path.css"/>
<script type="text/javascript" src="path.js">
</script>
</head>
<body onload="init();">
<p class="display">
    <canvas height="300" width="500" id="drawPanel">
        Get a modern browser!
    </canvas>
</p>
</body>
</html>
```

Next, `path.css`

```
h1, h2, .display
{
    text-align:center;
}
canvas
{
    border:solid 1px black;
    background-color:white;
}
```

Finally, `path.js`.

```
/*Author: Morrison*/
function init()
{
    var c = document.getElementById("drawPanel");
    var g = c.getContext("2d");
    g.fillStyle="#99BADD";
    g.fillRect(0, 0, c.width, c.height);
    g.fillStyle = "#CC0000";
    g.beginPath();
    g.moveTo(100,100);
    g.lineTo(50, 150);
    g.lineTo(150,150);
    g.lineTo(100,100);
    g.fill();
    g.strokeStyle = "#001A57";
    g.moveTo(300,100);
    g.lineTo(250, 150);
    g.lineTo(350,150);
    g.lineTo(300,100);
    g.stroke();
}
```



You can see that `fill` fills a region and that `stroke` draws a patth.

Programming Exercises These exercises will take you through some interesting features involving shapes, colors, and transparency

1. Fill two overlapping circles and use these colors, `"rgba(255, 0, 0, .5)"` and `"rgba(0, 0, 255, .5)"`. What do you see? Try varying the last argument between 0 and 1.0. Draw more overlapping shapes and experiment.
2. Write a function `strokeHexagon(x, y, s)` that will draw the boundary a regular hexagon centered at `(x,y)` with with side length `s` on a convas.
3. Write a function `fillHexagon(x, y, s)` that will fill a regular a regular hexagon centered at `(x,y)` with with side length `s` on a convas.