Chapter 5, JavaScript Boss Statements: Becoming Turing-Complete

John M. Morrison

October 26, 2018

Contents

0 Introduction and Orientation

Let us begin with the Church-Turing Hypothesis: *All computers are created equal inasmuch is that they can solve the same set of problems. Some computers are just faster than others.* A computer language is *Turing-complete* if it can be used to solve any of this set of problems. In this chapter, the JavaScript language will become Turing-complete. We seem to be a very long way from this seemingly lofty goal, but in reality we are not.

Once we achieve this goal, what will happen in subsequent chapters is that we will add to the basic set of tools we create here, deal with the idea of managing collections of objects using *data structures*, and develop the means by which JavaScript can make web pages execute complex instructions based on user interaction.

So far, any JavaScript code has executed line-by-line. All of the code we have seen so far hs consisted of *worker statements*. These are just grammatically complete imperative sentences. Here are some examples.

`let x = 5;`	Create the variable `x` and make it point at 5.
`x++;`	Increment the value of the variable `x`.
`x = Math.pow(x, 3);`	Find the cube of `x` and assign it back to `x`.
`console.log("freep");`	Put the string `"freep"` to the console.

A worker statement in a JavaScrpt file should end with a semicolon. The semicolon in JavaScript behaves like a period in written English; is signifies a full stop.

You might properly ask these questions.

1. Can we skip code?
2. Can we respond differently to different user actions? Can our code make decisions?
3. How do we store a set of statements under a name so we can use them over and over again?
4. How can we get something to happen repeatedly?

All of these have something in common: all entail our next object of study: the *boss statement*.

We shall use the term *block* to describe any piece of code enclosed in curly braces. Boss are grammatically incomplete sentences. Also, they are "boss" because they own a block of code. So they look like this.

```
bossStatement
{
    //code
}
```

The boss statement is also boss because it exerts some sort of control over the block of statements it owns, and therefore over the flow of control in a program. In this chapter We will begin by learning about three types of boss statement, function headers, conditional boss statements, and the `while` boss statement.

We will begin by looking at JavaScript functions, which will allow us to save a procedure under a name. We will start by reminding ourselves about the pertinent definitions from mathematics.

1 Mathematical Functions

It will be very useful for us to review the concept of a function in the mathematical sense and to expand this definition to realms not usually thought about in a mathematics class. Let us review the *mathematical* definition of function.

A *function* is an object consisting of three parts.

- A set A called the *domain* of the function
- A set B called the *codomain* of the function
- A rule f tying each element of A to a unique element of B. If $x \in A$, we will write $f(x)$ for the rule f applied to the element x.

To fully define a function we must know all three of these things. When we see this we will write $f : A \to B$. Note that for any given input the same output must be generated each time the function is applied.

For example the function `parseInt` is a mathematical function. Its domain is the set of all numerical strings. Its codomain is the set of integers. The rule is: re-interpret the numerical string as a integer. Here we see it at work on an example.

```
> parseInt("145")
145
```

It accepts the string `"145"` and returns the number 145. This is indeed a function in the mathematical sense, but it is a safe bet you didn't discuss this example in your math class.

When you hear someone say "the function $f(x) = x^3+4x$" in a mathematical context, what is really being said? In math class you adopt a convention that

dictates what is *really* being said is that the domain and codomain are the set of real numbers and the rule tying each element x of the domain to an element of the codomain is described by

$$f(x) = x^3 + 4 * x.$$

The variable name x used in our description here is immaterial. What is important is the rule: *cube a number and add it to four times the number*. That rule could just as well be embodied by

$$f(\text{cow}) = \text{cow}^3 + 4 * \text{cow}.$$

The names x and cow are called *arguments* of the function. The choice of an argument name does not affect the action of the function. The argument name is really just a placeholder; it is a convenience that helps us to describe the action of the rule.

It is interesting to note that the `Math` object contains a JavaScript function that is *not* a mathematical function. Let us look at the function `Math.random()` in the console.

```
> Math.random()
0.9340458753527254
> Math.random()
0.03806318070385406
> Math.random()
0.037572017900803445
```

You should try this; notice that your results will almost surely not be the same ones you see here. This JavaScript function is not a mathematical functions for two reasons.

Firstly, it has no input. Oops.

Secondly, it is inconsistent. Even if it took an input, it can have different values when it runs at different times. For a mathematical function, we have said that for each element x of the domain the rule $f(x)$ must return the *same* value in the codomain. So, even though `Math.random` is in the Math object, it is not a mathematical function.

Now let us discuss the function `console.log`. It accepts a string as input. It returns no value; it simply puts the string to the console. When mathematical functions do their job, they leave nothing behind. This function leaves stuff on the console after it is done running. It is not a mathematical function.

What you will now see is that the JavaScript function is a more general construct. Its purpose is to store a procedure under a name.

2 Writing JavaScript Functions

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.

We have seen that JavaScript has built-in functions. We have met the `Math` object which has all manner of nice scientific calculator functions. Now see this.

```
> typeof(Math.cos)
"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. The boss statement `function(x)` will allow us to define a function that accepts one input. You should read `function(x)` as "to define a function of one variable x," Notice that this is a grammatically incomplete sentence. The block is needed to complete it.

```
> f = function(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*. Arguments act as inputs to a function. When we call a function and give it values for its arguments we are said to be *passing* those values to the function.

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.

Formatting Functions If you are writing code in a JavaScript file, you should format your functions like this.

```
square = function(x)
{
    return x*x;
}
```

You will notice that the boss statement `function(x)` has no semicolon at the end. It is an error to place one there; doing so will "decapitate" your function. Don't be Henry VII. It causes ugly problems.

There is a second syntax for function definition you will also see which looks like this.

```
function square(x)
{
    return x*x;
}
```

Both of these do exactly the same thing.

Programming Exercises open a console session.

1. Create a function called `square` that squares a number.
2. Create a function called `cube` that calls `square` to cube a number.
3. Create a function called `lastChar` that accepts a nonempty string as input and which returns the string's last character.
4. Which of the functions you created here are mathematical functions?

3 Three Critical Properties of Functions

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

- **Inputs** When you create a function, you need to be aware of the types of inputs it makes sense for the function to handle. You can have zero or more inputs.
- **Outputs** You can return one JavaScript object from a function via the `return` statement. This will be the function's output.

- **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.

The action of a function can be described in terms of *preconditions* and *postconditions*. Preconditions describe what should be true before a function is called; this is how you specify the types of arguments that should be passed for your function to work correctly. Outputs and side effects are postconditions of a function; they are what is true once the function has done its work.

If you work in a professional programming shop, you will be asked to *document* your functions, so others can reuse them without having to puzzle through your code.

For example, to document the function `square`, you might do this.

```
/*
*  precondition: x is a number
*  postcondition:  returns x*x.  This is a pure function.
*/
function square(x)
{
    return x*x;
}
```

The designation "pure function" means that this function behaves as a mathematical function. It has no side-effect and it is consistent.

Programming Exercises Now you will get a chance to write some functions.

1. Write a function named `function sinDeg(x)` that computes the sine of the angle `x`, where `x` is given in degrees.
2. Repeat the first exercise for the cosine and tangent functions.
3. Write a function `function thirdSide(a, b, theta` that computes the length of the third side of a triangle if you know the lengths `a` and `b` of the other two sides and the angle `theta` between them in degrees.
4. Write a function called `function greet(name)` that, when passed a name such as "thar Matey", puts `"Hello thar Matey!"` to the console. Note here that no `return` statement is needed.

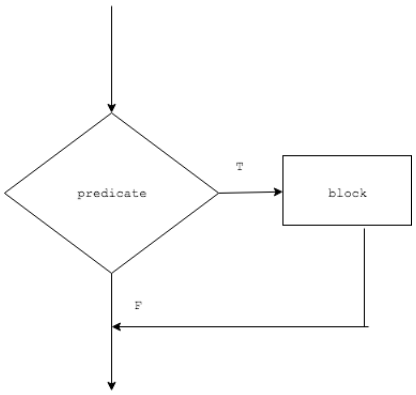
4 Making Decisions: the `if` statement

We are all familiar with the absolute value function from math class. The absolute value of a number is the geometric distance from that number to 0. You will often here this equivalent formulation: if the number is less than zero, strip off its sign; otherwise leave it alone.

We will use a new statement, the `if` statement, to create a JavaScript function to compute the absolute value of a number. The `if` statement looks like this.

```
if(predicate)
{
    //block o' code
}
```

This diagram shows the action of this boss statement.



The quantity `predicate` is a boolean-valued expression; to wit, it evaluates to either `true` or `false`. If the predicate is true, the block executes. If not, the block is skipped.

So, to write an absolute value function, if the value `x` is negative we can change its sign with the worker statement`x = -x;`. Otherwise, we will do nothing.

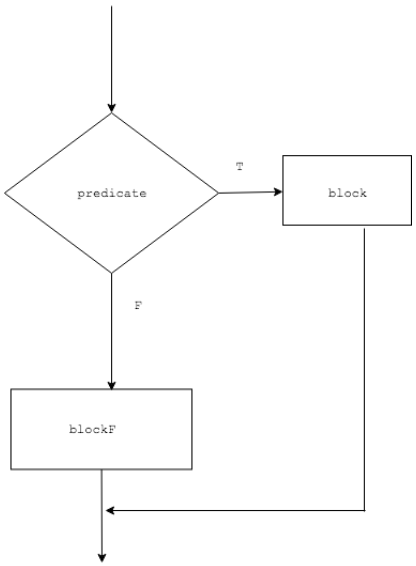
```
function abs(x)
{
    if(x < 0)
    {
        x = -x;
    }
```

```
    return x;
}
```

We now paste this into a console and see what happens. It seems sensible to test it for a positive value, a negative value, and zero. What we are doing here is to test each path of execution, as well as the border case (0).

```
> function abs(x)
{
    if(x < 0)
    {
        x = -x;
    }
    return x;
}
undefined
> abs(5)
5
> abs(-5)
5
> abs(0)
0
```

So here is a question: What if you wanted this?



What we are looking for here is for one block to execute if the predicate is true and another to execute if it is false. This can be done with an `else` statement.

There is a catch: the `else` statement must occur right after the `if`'s block ends. It is an error to have other code between an `if` and an `else`. Here is what the statement looks like.

```
if(predicate)
{
    //this block executes if the predicate is true
}
else
{
    //this block executes if the predicate is false
}
```

By way of example, let us implement the `signum` function, `sgn`. This function returns 1 if its argument is positive, -1 if the argument is negative, and 0 if it is passed a 0. Let us rough it in and document it.

```
/*
*  precondition: x is a number.
*  postcondition: returns 1 if x > 0, -1 if x < 0 and 0 if x is 0.
*  This is a pure function.
*/
function sgn(x)
{
}
```

Why document first? It's a smart practice because you have specified *exactly* what the function is to do. It also suggests how you should test the function to make sure it works.

Now let's go to work.

```
/*
*  precondition: x is a number.
*  postcondition: returns 1 if x > 0, -1 if x < 0 and 0 if x is 0.
*  This is a pure function.
*/
function sgn(x)
{
    let out = 0;
    if(x != 0)
    {
        if(x > 0)
        {
            out = 1;
```

```
        }
        else
        {
            out = -1;
        }
    }
    return out;
}
```

We cheated; we used an `if` statement inside of an `if` statement! This sort of thing is called *nesting* and it occurs often.

Note the thread of logic. We set our output variable to zero; if `x` is 0, then the `if` statement is skipped and we return 0. Otherwise, we handle the case where `x` is nonzero. Now we test our shiny new function and see it looks pretty good.

```
> function sgn(x)
{
    let out = 0;
    if(x != 0)
    {
        if(x > 0)
        {
            out = 1;
        }
        else
        {
            out = -1;
        }
    }
    return out;
}
undefined
> sgn(5)
1
> sgn(-5)
-1
> sgn(0)
0
```

At this time, you might ask, "This artifice you used was slick and pretty, but it seems like a headache to do it every time I want to have a three or more way fork in conditional logic." Objection noted and accepted as eminently reasonable. Happily there is a way. We will just drop it in here and you will be pleased.

```
/*
```

```
*  precondition: x is a number.
*  postcondition: returns 1 if x > 0, -1 if x < 0 and 0 if x is 0.
*  This is a pure function.
*/
function sgn(x)
{
    let out = 0;
    if(x > 0)
    {
        out = 1;
    }
    else if( x < 0)
    {
        out = -1;
    }
    else
    {
        out = 0;
    }
    return out;
}
```

Paste this into the console and watch it work. Here is a summary of the rules for conditional logic.

Simple `if` If the `if`'s predicate is true, its block is executed; otherwise, the block is ignored.

`if` with `else` An `if` with an `else` is a linked set of boss statements. If the `if`'s predicate is true, its block is executed and the `else` block is ignored. If the `if`'s predicate is false, its block is ignored and the `else` block is executed.

`if-else if-else` progression This is a linked set of boss statements. If the predicate of the `if` is true, the `if`'s block is executed and the rest of the progression is skipped.

If the predicate of the `if` is false, the progression will keep trying the predicates belonging to the `else if`s. If one of them is true, that block is executed, and the rest of the progression is ignored.

Finally, if the predicates of the `if` and all of the `else if`s are all false, the `else`'s block executes. You can elect not to have an `else` at the end of this progression, but it is a good practice to have it so you can put out an error statement in the event an illegal value is passed to the function containing the progression.

Programming Exercises Write these functions. Remember, you have your own solution manual: paste them into the console and test them.

1. Write a function called `monthName` that accepts a number 1-12 as an argument and which returns the corresponding month as a string. For example 1 returns `"January"`, 2 returns `"February,"` etc. Return the string `"ILLEGAL YOU FOOL"` if an illegal number is passed to the function.
2. The rule for leap years is as follows. If a year is divisible by 4 it leaps. If a year is divisible by 100, there is an exception; it does not leap. However, if a year is divisible by 400, there is *an exception to the exception*. Write a function named `leapFactor(year)` that returns a 1 if a year leaps and a 0 otherwise.
3. Write a function `canDo(age)` that does the following. If age is 18 or over, return the string `"You can vote"`. if the age is 21 or over, return the string `"You can order a drink"`, and if the age is 65 or over, return the string `"You can apply for Medicare"`. If the age us under 18, return the string `"You are still a minor"`.

5 And Now `while` we Repeat Ourselves

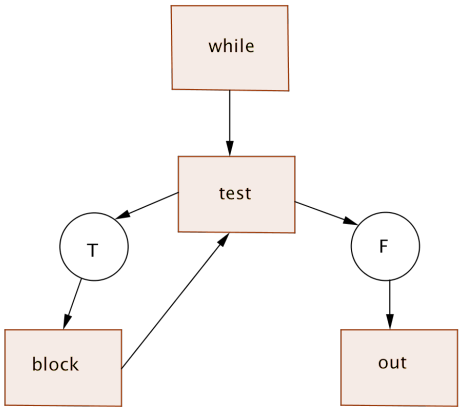
Now we will go after the third objective of this chapter: repetition. You will now meet a new boss statement, the `while` statement. It looks like this.

```
while(predicate)
{
    //block o'code
}
out;
```

You can think of it as a "sticky `if`" because it keeps executing the block of code repeatedly until its predicate is false. The statement `out` is just the next statement beyond the loop.

This construct is an example of what is called `iteration`; it is a repetition structure.

This *loop* diagram illustrates its action.



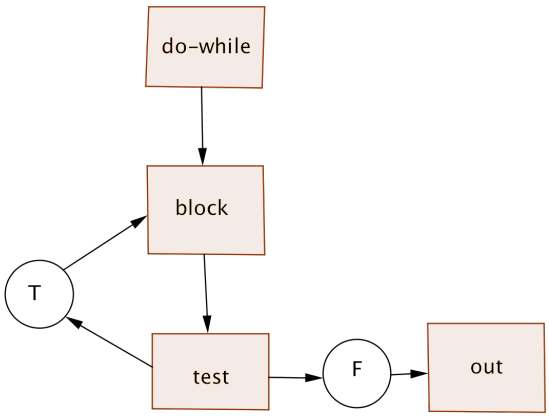
So let's walk through this. The loop is encountered and the predicate `test` runs. If it evaluates to `false`, you are out of the loop; otherwise, the block runs. The cycle of `test-block` is repeated. As soon as `test` becomes false, you are out of the loop. You are guaranteed that at the end of a `while` loop that its test is false.

It is possible for a `while` loop never to run its block. If `test` is false when it is first encountered, you go to `out` and the loop is finished.

The `while` loop has a variant called a `do-while` loop that looks like this.

```
do
{
    block;
}while(test);
out;
```

A big difference is that the test occurs *after* each repetition of the block. Therefore this loop's block is guaranteed to run at least once. Here is the diagram for this loop



Now we describe how this loop works. First, its block executes. Then the test is carried out. If the test evaluates to true, the block executes; otherwise, you are out of the loop.

Important! Design Comments You should use the regular `while` loop about 99.44% of the time. There are certain situations where it is advantageous and clearer to use it. For nearly all situations, the `while` loop is a cleaner and better way of doing things.

There are two other disreputable keywords you will see in OPC (other peoples' code). This rogue's gallery consists of `break` and `continue`. The `break` command breaks out of a loop. It then voids the guarantee that the loop's test is false at the time you exit the loop. That is very bad. The other, `continue`, will cause control to pass to the top of the block and for the block to re-execute without the test occurring.

Smart design will virtually always obviate the need for these two crutches. Avoid them like the plague. The need for them develops if you have designed your loop's test improperly.

Now we roll out a powerful new tool that allows us to do something repeatedly; this is called `iteration`.

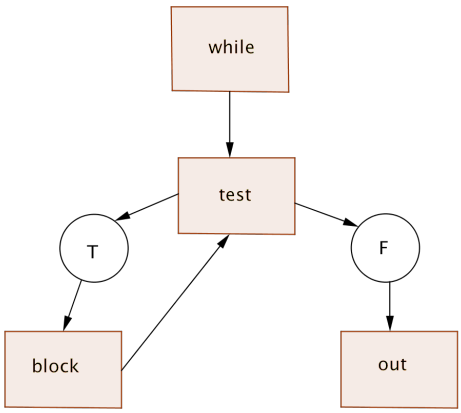
We begin with the `while` loop. Its code looks like this

```
while(test)
{
    block;
```

```
}
out;
```

The object `test` is a boolean-valued expression (predicate). The `block` represents a block of code. The statement `out` is just the next statement beyond the loop.

This *loop* diagram illustrates its action.



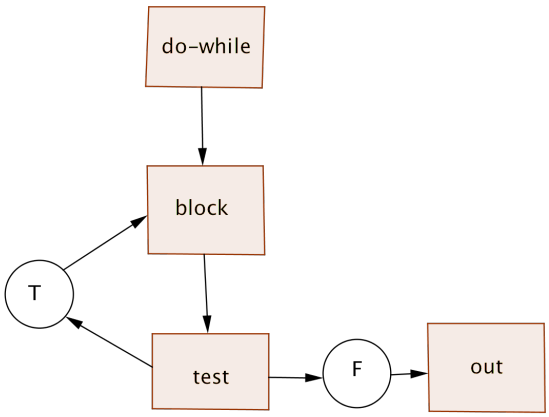
So let's walk through this. The loop is encountered and the predicate `test` runs. If it evaluates to `false`, you are out of the loop; otherwise, the block runs. The cycle of `test-block` is repeated. As soon as `test` becomes false, you are out of the loop. You are guaranteed that at the end of a `while` loop that its test is false.

It is possible for a `while` loop never to run its block. If `test` is false when it is first encountered, you go to `out` and the loop is finished.

The `while` loop has a variant called a `do-while` loop that looks like this.

```
do
{
    block;
}while(test);
out;
```

A big difference is that the test occurs *after* each repetition of the block. Therefore this loop's block is guaranteed to run at least once. Here is the diagram for this loop



Now we describe how this loop works. First, its block executes. Then the test is carried out. If the test evaluates to true, the block executes; otherwise, you are out of the loop.

Important! Design Comments You should use the regular `while` loop about 99.44% of the time. There are certain situations where it is advantageous and clearer to use it. For nearly all situations, the `while` loop is a cleaner and better way of doing things.

There are two other disreputable keywords you will see in OPC (other peoples' code). This rogue's gallery consists of `break` and `continue`. The `break` command breaks out of a loop. It then voids the guarantee that the loop's test is false at the time you exit the loop. That is very bad. The other, continue, will cause control to pass to the top of the block and for the block to re-execute without the test occurring.

Smart design will virtually always obviate the need for these two crutches. Avoid them like the plague. The need for them develops if you have designed your loop's test improperly.

Hanging and Spewing Hanging in JavaScript causes the little doughnut of death to spin interminably as your page fails to load. It ends in an ugly "Aw Snap!" page from Chrome. This is caused by a failure of a loop to terminate in a finite number of steps.

Here is a common n00b programming error. Suppose we want to compute

the nth triangle number; these numbers are defined by

$$T(n) = 1 + 2 + \cdots + n = \sum_{k=1}^{n} k.$$

We compute a few by hand $T(1) = 1$, $T(2) = 1+2 = 3$, and $T(4) = 1+2+3+4 = 10$. So we write this

```
/*
* precondition: n is a positive integer or 0.
* postcondition: returns the nth triangle number
*/
function triangle(n)
{
    let k = 0;
    let total = 0;
    while(k <= n)
    {
        total += k;
    }
    return total;
}
```

The value of `k` starts off at `0`, and it never changes. Hence, you are an eternal prisoner of this loop. Correcting this is easy. Just insert a `k++;` (Right?!) at the end of the loop's body and you will achieve the intended effect. In a `while` or `do-while` loop, you want to be sure that "progress is being made" towards making the loop's test evaluate to `false`. This error is known as *infinite loop*.

Spewing occurs in an infinite loop when the loop's body causes text to be generated in the console or on a page. The text just keeps coming until the browser freezes or rings down the curtain on the problem. If you put a console.log statement inside of the loop in the `triangle` function, you can run it and see it spew.

The moral of the story is this: *When using a `while` loop, ensure that progress is being made toward the predicate becoming false.*

Programming Exercises For some of these exercises, the string method `repeat` will come in handy. These exercises will help you get into the groove with the `while` loop.

1. Write a function `rect(ch, nrows, ncols)` that prints a rectangle consisting of the character `ch` that has `nrows` rows and `ncols` columns. For example `rect(z, 5, 6)` should print

```
zzzzzz
zzzzzz
zzzzzz
zzzzzz
zzzzzz
```

2. Write a function `triangle(ch, nrows)` that prints a rectangle consisting of the chracter `ch` that has `nrows` rows and `ncols` columns. For example `triangle($, 5)` should print

```
$
$$
$$$
$$$$
$$$$$
```

3. Write a function `halfDelta(ch, nrows)` that prints a rectangle consisting of the chracter `ch` that has `nrows` rows and `ncols` columns. For example `halfDelta(Q, 5)` should print

```
    Q
   QQ
  QQQ
 QQQQ
QQQQQ
```

   Notice that the bottom row is at the first character in the row.

4. Write a function `parallelogram(ch, nrows, ncols, pitch)` that prints a rectangle consisting of the chracter `ch` that has `nrows`, `ncols` columns, and a pitch of `pitch`. For example `parallelogram("p", 5, 6, 3)` should print

```
         pppppp
      pppppp
   pppppp
pppppp
```

   The pitch is just a measure of the shear to the right the parallelogram takes. There is a variety of solutions to this; can you think of at least two?

5. Write a function `smear(word, n)` whose action looks like this. The `length` property of a string will be a handy-dandy tool here.
   The call `smear("Moose", 5)` prints this.

```
MMMMM
ooooo
ooooo
sssss
eeeee
```

The call `smear("Regalis", 1)` yields this

```
R
e
g
a
l
i
s
```

6 And Now Back to Web Pages

We have greatly advanced the power of the JavaScript language. Now it is time to see what we can do with this on a web page. Suppose you have a web page that is going to prompt the user for a number and which will perform some action based on that input. A basic piece of good web security is to validate the user's entry to prevent possible problems.

Let's write two functions: `isValidInteger` and `isValidPositiveInteger` that return `true` when a valid value is passed to them.

We begin with `isValidInteger`. What constitutes a valid integer? It could start with a sign, i.e. a + or a -, or have no sign. Succeeding characters must all be digits. So, it might be a good idea to check if a character is a digit. It is a good idea to make a separate function to check if a one-characters string is a digit. This is a handy function with a single purpose.

```
function isDigit(ch)
{
    if(ch.length == 1)
    {
        return "0" <= ch && ch  <= "9";
    }
    return false;
}
```

Now let us think about our approach. If the first character is a + or a -, we will move over one and ignore it. If it is a non-digit we will bail and return `false`. So our start looks like this.

```
function isValidInteger(s)
{
    let k = 0;
    if(s[k] == "+" || s[k] == "-")
    {
```

```
        k++;
    }
    //now the rest should just be digits.
}
```

Next, we will go through the rest of the string; if any character is a non-digit, we return `false` and we are done.

```
function isValidInteger(s)
{
    let k = 0;
    if(s[k] == "+" || s[k] == "-")
    {
        k++;
    }
    let n = s.length;
    while(k < n)
    {
        if(!isDigit(s[k]))
        {
            return false;
        }
        k++;
    }
    //if we are here, all of s's characters are digits.
    return true;
}
```

Now let us make the `isValidPositiveInteger` function. Here there is no + or - to deal with. We just check if everything is a digit.

```
function isValidPositiveInteger(s)
{
    let k = 0;
    let n = s.length;
    while(k < n)
    {
        if(!isDigit(s[k]))
        {
            return false;
        }
        k++;  //go to the next character
    }
    //they are all digits now.
    return true;
}
```

Do you notice we have repeated ourselves? This is a violation of the 11th commandment: *Thou shalt not maintain duplicate code*. We can fix this by changing the implementation of one of the functions. You can see that we can shorten this by having one function call the other as follows.

```
function isValidInteger(s)
{
    let k = 0;
    if(s[k] == "+" || s[k] == "-")
    {
        k++;
    }
    //s.substring(k) should be digits only
    return isValidPositiveInteger(s.substring(k));
}
```

6.1 JavaScript Modal Dialog ("popup") Boxes

There are three major types of popups that are used in JavaScript to obtain text input or send messages to the user.

- The function `alert` creates a popup that conveys a message. Its argument is a string, which is the message to be conveyed.
- The function `prompt` creates a popup that asks the user for text. It requires two arguments. The first is the message on the box, the second is the default text if the user enters nothing. The text typed in by the user is returned to the caller.
- The function `confirm` produces a box with two buttons marked "OK and "Cancel." It accepts one argument, a message from the page. It returns true if the user hits the OK button and false if the user hits the cancel button.

Create this page named `demo.html`.

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8"/>
    <title>JavaScript Dialog Demo</title>
</head>
<body>

<h2> JavaScript Dialog Demo</h2>
```

```
<p>There are three major types of popups that are used in JavaScript to obtain
textual information from a user.  You can click on the box types to see an
example of each in the list below.  Notice how we can apply an
onclick attribute to any element in the body of a page. When it is
clicked, the JavaScript in the quotes is executed.</p>

<ul>
        <li>The function <code onclick="alert('I am an alert box.');">alert</code> creates a
a string, which is the message to be conveyed.</li>
    <li>The function <code
        onclick="prompt('I am a prompt box.', 'foo');">prompt</code>
    creates a popup that asks the user for text.  It requires two arguments.
    The first is the message on the box, the second is the default text if the
    user enters nothing.  The text typed in by the user is returned to the
    caller.</li>
    <li>The function <code onclick="confirm('Don\'t ask me what I am.');">
    confirm</code> produces a box with two buttons
    marked "OK and "Cancel."  It accepts one argument, a message from the page.
    It returns <code>true</code> if the user hits the OK button and
    <code>false</code> if the user hits the cancel button.</li>
</ul>
</body>
</html>
```

Open it with your browser and see the three types of boxes. We will use these boxes to react to obtain user input. We can then use conditional logic, which we are about to discuss, to carry out different actions based on the user's input.

6.2 Nagging

Let's "nag" the user until he enters a valid integer. This is an actual use case for the `do-while` construct.

```
function getInteger(message)
{
    let s = "";
    do
    {
        s = prompt(message, "");
    }while(!isValidInteger(s));
    //we nagged until s was a valid integer.
    return parseInt(s);
 }
```

To test this, paste the functions we have made so far into a console session. then do this.

```
let v = getInteger("Enter an Integer:");
```

Enter some gibberish and you will see this function nag until you enter an actual integer.

7 The Dope on Scope

We now have new facilities to use in writing JavaScript programs. Now let us get down in the grammatical nitty-gritty and discuss a few facts of life.

So far, we have been cavalier about the lifetimes of variables and the lifetimes of functions. Take a look at this little session.

```
> function square(x)
{
    return x*x;
}
undefined
> square(5)
25
> x
VM151:1 Uncaught ReferenceError: x is not defined
    at <anonymous>:1:1
```

Shouldn't `x` be 5? Instead, what we see here is that JavaScript has never heard of the variable `x`. What happened here and why?

The call `5` causes `5`'s memory address to be copied and sent to `square`. Now the `x` in `square(x)` points at the value `5`. This gets recorded in `square`'s private symbol table. Yes, functions when they are executing, have their own private symbol tables. These consist of the arguments, which point to the values that got passed in and any variables created inside of that function.

Now we turn to the statement inside of `square`. JavaScript first evaluates the expression `x*x`; it fetches `5` from its private symbol table and substitutes it in for `x`. The result is `25`. Next, the `return` statement does two things. Firstly, it sends the value `25` back to the caller. Secondly, as soon a function encounters a `return` statement, its execution ends and its private symbol table vanishes. Any variable you create inside of a function is said to be *local* to that function, and it only can be seen when that function is executing.

A Word of Caution Always create variables with the `let` keyword; it is now preferred to its predecessor `var`. If you do not use `var` or `let`, you might be

writing unnecessarily on the global symbol table.

If you create a variable using `let`, here is now to determine its lifetime. Find the closest set of enclosing curly braces containing the variable. This is called the variable's *block*. Variables created with `let` are never visible outside of their blocks.

If you create a variable using `var`, the variable has `function scope`; it is not visible outside the function in which it was created. If is function is short, there is no difference between `var` and `let`.

Variables created outside of any function are global and are visible everywhere. Any portion of the program could potentially modify them, and that renders their behavior unpredictable.

You might be tempted to think, "Why all of this modesty? Isn't it good to have variables visible everywhere?" Here is a good reason. Think about using a function in the Math object. Do you know the names of its local variables? Could there be a name conflict? Because of this modesty, you are relieved from having to worry about that. This business of hiding variables is a big advantage to using functions, because *you don't have to worry* ***how*** *a function works, you just need to know* ***WHAT*** *is does.* You have seen this at work already; you notice we have not exercised any care at all about using different names for variables in different functions. This leads us to an important principle.

`The Blofeld Principle` *Never let a variable outlive its usefulness.* Once you are done with a variable, you want it gone.

Masking Take a look at this program.

```
function f()
{
    let y = 1;
    let count = 0;
    while(count < 1)
    {
        let y = 2;
        count++;
    }
    return y
}
console.log(f());
```

Run it and you will see that it returns 1, not 2. This is because, inside of the `while` loop, you have created a *second* `y` inside of the function `f`, which dies when the while loop ends after one run. The purpose of this artificial example is to show that if you redeclare a variable inside of an inner block, you create a new variable that lives in that scope.

Do-Now Exercise Edit the script above and remove the `let` inside of the `while` loop. Why is the result different or not different?

8 Polycephaly is Frowned Upon: An Exhortation on Design

When writing a function it is wise to have it do one specific task. If your argument list is very long, or the body of the function goes on for more than a screenful or so, you may be creating a hydra-headed monster that will be hard for others to understand (pssst..... one reason we write functions is so we or others can use them later), and if it has a problem, be painful to debug.

Cerberus, the three-headed dog

Consider the case study we just did on nagging a user. Notice how we proceeded in stages. We decided we wanted to test if a string is a valid numerical string. We began to think about the pieces of the problem. One of them was to check if a character is a digit. This is the kind of straightforward task you might want to put in a function. It is not to hard to think of other ways you might use this function (check if a string is a valid social security number).

Once we knew how to test if a character is a digit, we were ready to write the loops necessary to test if a string is a valid integer or positive integer.

Notice also that we give names to function that are evocative of their actions. Boolean-valued functions should, for the most part, have their names start with `is`. This is a hint to other programmers that the function is returning a boolean.

This process we have described of decomposing a problem into smaller and smaller pieces until they become easy to code is called *top down design*.

Programming Exercises

1. Write a function that checks a string to see if it contains only upper-case letters. Think about writing a "helper" as we did when we wrote `isDigit`

2. Write a function that checks a string to see if it is a valid 24-bit color hex code. You should allow there to be a prefix of `0x`, `0X`, or `#`, or allow no prefix at all.

9 Introducing the JavaScript Canvas

The `canvas` element in HTML provides an environment in which graphical objects can be rendered on a web page. Controlling this rendering is done by JavaScript. We will apply some of the things we have learned in this chapter to create drawings on a canvas. We will also introduce something called the *load event*, which is broadcast to a web page when it is done loading. We can use this event to delay the execution of code until the page has loaded.

Here is why this is important. Generally, you put your JavaScript in the head of your page or you pull it in from another file using the `script` tag's `src` attribute. When we use a canvas, we will give it an `id`. To get a handle to this canvas, we will use `document.getElementById`. Remember, pages load from top to bottom. So, if the JavaScript attempts to get the `id` of an element on a page that does not yet exist, we get a kafkaesque error message. Hence, we should delay the calling of our JavaScript code until the page is loaded. Let us go through the basic setup.

First, a little minimal CSS so we can see our canvas clearly on the page.

```
h1, h2, .display
{
    text-align:center;
}
canvas
{
    border:solid 1px black;
    background-color:white;
}
body
{
    background-color:#FFF8E7;
}
```

Notice the `body` tag in this HTML.

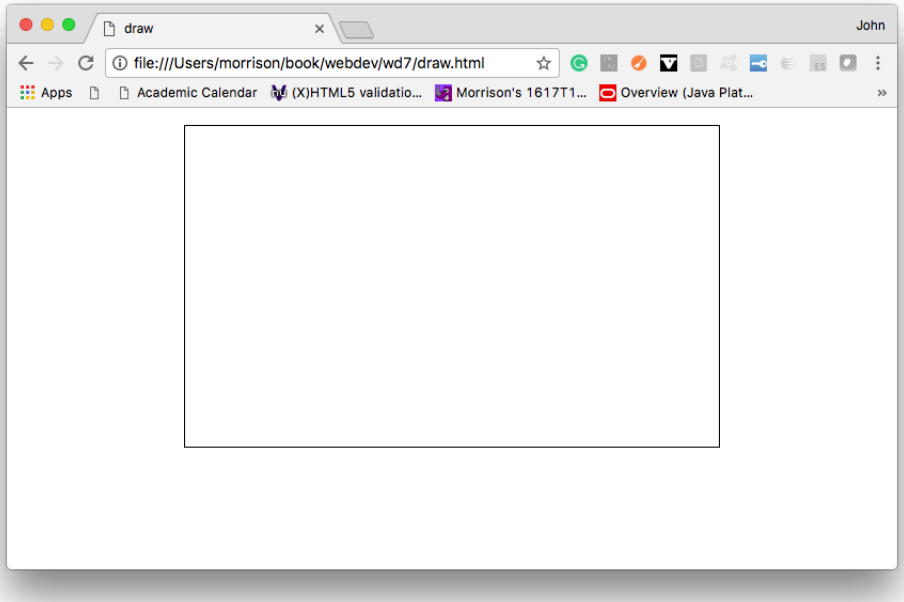
```
<!doctype html>
<html>
<head>
<title>canvas</title>
<meta charset="utf-8"/>
```

```
<link rel="stylesheet" href="canvas.css"/>
<script type="text/javascript" src="canvas.js"></script>
</head>
<body onload="init();">
    <h2>Canvas Demonstration</h2>

    <p class="display">
        <canvas height="500" width="700" id="surface">
            Get a modern browser, chump.
        </canvas>
    </p>
</body>
</html>
```

The `onload` attribute tells what JavaScript code should be called when the page is done loading. This is your first example of an *event*; we will discuss these in ample detail later.

Also, notice that you need to give your canvas a size. An `id` is needed so JavaScript can draw on it. Open your HTML file with your browser and you will see this.



Finally, you create the file `canvas.js`

```
function init()
{
    let c = document.getElementById("surface");
    let pen = c.getContext("2d");
}
```

Since the function `init` is not called until the page has loaded, you will have a pointer to your canvas. Canvases, conveniently enough, come with a pen. The second line of code shows you how to get the canvas's pen.

9.1 Draw.... or choose to die

How do we use this pen? Well, here is an easy example Modify `init` as follows.

```
function init()
{
    let c = document.getElementById("surface");
    let pen = c.getContext("2d");
    pen.fillRect(100,50,60,60);
}
```

Now it's time for you to do the drawing, Pard.

Lee Van Cleef

do-now Exercise

1. Figure out what the four numbers mean in `pen.fillRect` by experimenting. You will see that the canvas has a coördinate system. How is is system different from the cartesian plane you learned about in math class?

2. Here is how to draw a line. You need to use `pen.moveTo` to get the pen where you want it and `pen.lineTo` to get it to draw to the desired destination. Here is the drill. It draws a line from `(x,y)` to `(a,b)`.

   ```
   pen.beginPath();
   pen.moveTo(x,y);
   pen.lineTo(a,b);
   pen.stroke();
   ```

3. The property `pen.fillStyle` controls the color of the pen when it is filling a region. Try setting `pen.fillStyle = "red"`, then draw something. What other colors can you change the pen to? See if you can find a dozen or so by experimenting.

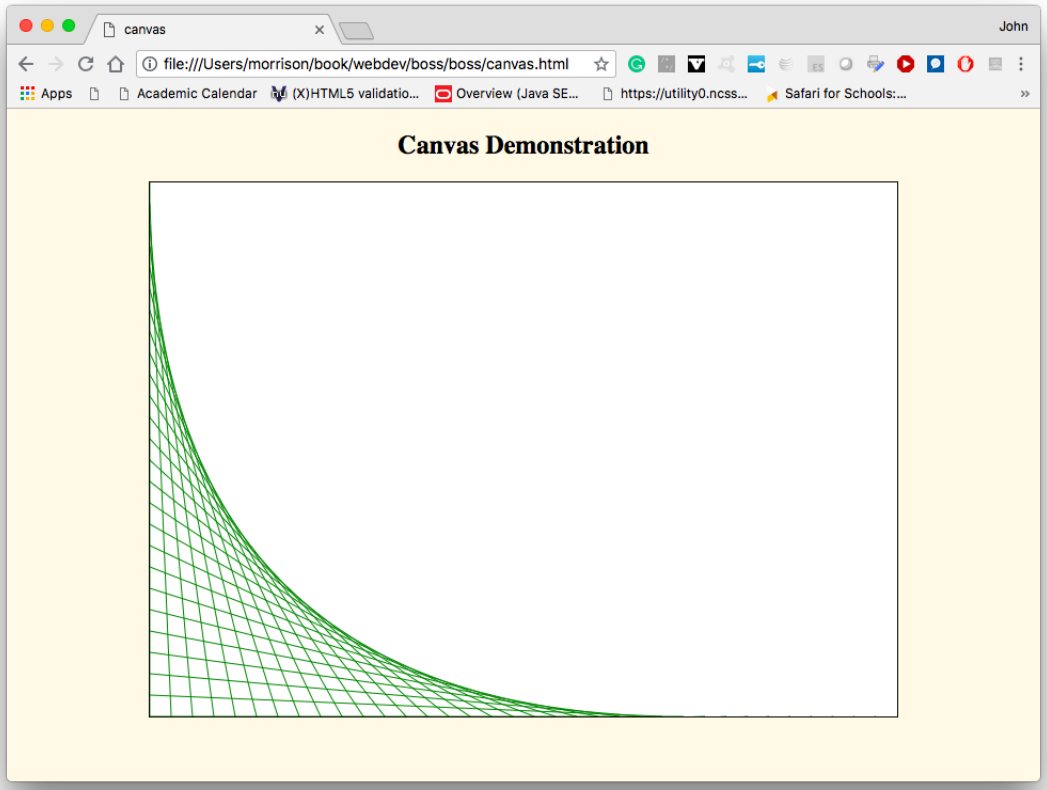
4. You can set the pen to any hex color easily. For example, `pen.fillStyle = #FFF8E7;` fills the pen with cosmic latte.

5. What does `pen.strokeRect` do? It takes four arguments. What about `pen.strokeStyle`?

6. Here is a function that draws a line. You give it a pen, the two start coördinates, the two end coördinates, and it will draw a line for you. Add it right after the `init` function.

   ```
   function line(pen, x, y, a, b)
   {
       pen.beginPath();
       pen.moveTo(x,y);
       pen.lineTo(a,b);
       pen.stroke();
   }
   ```

   Use this function to create this cool picture; a loop will do this very nicely. A canvas, incidentally, knows its height and width. For our canvas here use `c.height` and `c.width`.



7. Go on IMDB and learn about Lee Van Cleef. He just might be the best villain ever.

10 Terminology Roundup

- **arguments** These are the inputs to a JavaScript function
- **block** Thh is any piece of code enclosed in a pair of matching curly braces.
- **block (of a variable)** This is the code contained in the closes set of matching curly braces containing the variable's creation.
- **block scope** Variables created with `let` are invisible outside of their block. This is block scope.
- **boss statement** These are grammatically incomplete programming statments. Boss statements need a block to complete them. For example `if(x > 5)` translatest to , "if x ¿ 5," This is a grammatically incomplete sentence.

- **call** This is the process of invoking a JavaScript function.
- **canvas** This is an HTML5 element which can be drawn in using JavaScript.
- **data structure** This is a container that stores related items u
- **document** These are comments placed in your code to help others use and understand it.
- **event** This is an object that is emitted by the browser at certain times (load, unload), or when a user interacts with a page. We attach listeners to events that fire their code when an event of their type is emitted. Examples include mouse clicks, keyboard hits, and turning of the mouse wheel. A
- **function (mathematical)** is an object consisting of three parts.
- A set A called the
- **domain** of the function
- A set B called the
- **codomain** of the function and a rule tying each element of the domain to some element of the codomain. A
- **function (JavaScript)** is an object which stores a procedure under a name.
- **function scope** Variables created with `var` are invisible outside of the function in which they are created. This is function scope.
- **graphics context** This is the canvas object's "pen."
- **infinite loop** This occurs when a program cannot break out of a loop. In these instance, you often must just kill the tab running the code containing one of these. This
- **loop** This refers to any repetition strucuture in JavaScript such as `while`.
- **local variable**. This refers to a variable created inside of a function.
- **nesting** This occurs when one boss statement is inside of the block of another.
- **pass** This is what you do when you send an input to a function. For instance, in `Math.sqrt(5)`, the value 5 is being passed to the square root function.
- **preconditions** These are things that should be true before a function is called. This includes the number and types of arguments that the function needs to run correctly
- **postconditions**These are things that should be true after a function is called. This includes describing any teturn value or side effects.
- **top-down design** This is the practice of breaking a problem into smaller and smaller pieces until they become easy to code.

- **Turing-complete** A langugage is Turing-complete if it has the ability to solve any solvable computational problem. The JavaScript language is Turing-complete.
- **worker statements**. These are grammatically complete programming statments. For example, the statment `x = x + 5;` is a worker statement since it reads, "Assign the value of `x` plus 5 to `x`."