Value Investment Tool
Code Conventions

This document specifies the coding conventions for our project. It outlines:
file, function, and variable naming
code commenting
code organization and architecture
data structure / algorithmic efficiency considerations.

By adhering to these standards, we will be led to produce code that is consistent, reliable, readable, and efficient. It will also lead to code that is appreciated by our evaluators (i.e. the course staff).

CONSTANTS
Named constants should be in all caps, words separated by underscores.
Example:
```
var NUM_ROWS = 4;
```

FILE NAMES
Names of files (html, css, js, png, jpg, python, etc.) should be written in all lower case, separated by underscores.
Example:
basketball_photo.png

VARIABLE NAMES, FUNCTION NAMES

Python

Variables and function names should use "snake casing" (i.e. all lowercase, words separated by underscores).

Example:

```
web_scraper_tool.py
unique_identifiers = []

def display_hello():
    print "hello"
```

HTML, CSS, JavaScript

Variable names, function names, id names for all of these languages should all be camel case.

Examples:

```
var mySortedList;

.centerTable {
}
```

INDENTATION

Indentation should be four spaces.

Example:

```
if (x == 3) {
    console.log("hello");
}
```

NOUNS AND VERBS

The names of functions should be verbs (i.e. communicate action).
Example:
playSoccer (i.e. not soccer)

Variables that hold data should be nouns.
Example:
var soccerBall = "";

If the variable is a collection, its name should have plural spelling.
Example:
var soccerBalls = [];

OPERATOR, ARITHMETIC, CONDITONAL SPACING

Loops and conditionals (if, for, foreach, while) should have a single space after the loop keyword.
Example:
for ()
if ()
not this:
for()
if()

The sections inside of a loop , as well as the operands/operators, should be separated by one space
Example:
for (var i = 0; i < 10; i++)
Not:
for (var i=0;i<10;i++)

Arithmetic operators should be separated by one space from their operands:
(x - y) / 4
not:
(x-y)/4

BRACES

When working with functions and code blocks, the opening brace should be on the same line as the code that precedes it. This is super important:

YES:

```
if (x == 3) {
   alert("hello");
}

var x = function() {
      return 2 * x;
}
```

NO:

```
if (x == 3)
{
   alert("hello");
}

var x = function()
{
      return 2 * x;
}
```

The opening brace should have a single space between itself and the preceding parentheses.

Example:

```
if (x == 3) {
}
```

If-else clauses often benefit from this style of writing:

```
if (x == 3) {
   alert("Joe");
} else if (x == 4) {
   alert("Sally");
} else {
   alert("Becky");
}
```

This style is known as the "K&R style" (after the creators of the C programming language), and often provides the best clarity.

However, if the material contained inside the braces is substantial, separating them can sometimes provide more clarity to the reader:

```
if (x == 3) {
  //many statements
}

else {
  //many more statements
}
```

This is particularly true if the conditionals have comments that involve something a bit sophisticated and non-obvious. For example:

```
//radiation levels exceed government guidelines
if (x == 3) {
  //statements
}

else {
  //more statements
}
```

But for simple stuff, the K&R style is often best.

COMMENTS

Comments should be informative, efficient, and unobtrusive. "Comment art" should be avoided.
Example:

```
/
*
**
**************************************
DONT DO THIS *************************
**************************************
**
*
*/
```

Unusual, complex functions should be briefly explained through comments.

NAMING STYLE

Variable names should not be overly terse. Do not take vowels out of words (like they often do in C).

Good:

```
removeDuplicateLetters(string);
```

Bad:

```
rmvDplLtr(s);
```

variables that store booleans, and functions that return booleans, often benefit by having the word "is" or "has" at the beginning of the variable name.

Examples:

```
hasEntries = false;
isPrime(3);
```

It is easy to see that the above are working with booleans.

This is context dependent. If you are writing a boolean variable or function, think if it will help an outside reader understand that it is a boolean by putting is or has in front of it. If so, do it.

```
}
```

CODE ARCHITECTURE AND ORGANIZATION

Code should be modularized into component parts. Even if the modular part is called only once, break the functions into logical parts anyway. In our application, there should be no need for functions that are dozens of lines long. Rather than creating a small collection of large functions, we should be creating a large collection of small functions.

Example: Suppose you were going to create some code that relates to going on vacation. This process consists of several parts. So create a separate function for each part. Then call the functions in a master function. Essentially, something like this:

```
composeVacation() {
   chooseTravelDates();
   chooseTravelLocation();
   bookAirline();
   bookHotel();
   writeTravelItenerary();
}
```

This is much preferable to a giant single 60-line function named composeVacation:

```
composeVacation() {

   //many lines of  code relating to choosing travel dates
         ...
         ...
   //many lines of code relating to choosing travel location
         ...
         ...
   //many lines of code relating to choosing book airline
         ...
         ...
   //many lines of code relating to choosing book hotel
         ...
   //many lines of code relating to writing travel itinerary
}
```

Also, avoid daisy chaining functions whenever possible. Daisy chaining is when you get to the end of a function, you pass the data to another function rather than returning. Then when you get to the end of that function, you pass the data to the next function again not returning. It's like the functions never end or return, they just keep going from one to the next.

Example of daisy-chaining (i.e. bad function inter-dependencies):

```
function a(3);

function a(x) {
    x += 10;
    b(x); //this is bad... a daisy chain.  the data is going from a() to b() to c() to d()
}

function b(x) {
    y = 2 * x;
    c(y); //the daisy chain continues... data going from b() to c() to d().
}

function c(y) {
    d(y - 5); //still going!  when will this function sequence ever end?
}

function d(z) {
    alert("z"); //finally the sequence stopped.  jeez!
}
```

When you write code like that, you effectively wrote one function, disguised as several. It signals a strong liklihood of bad code design from a logic / architecture standpoint.

Rather, you should get the return value out of the first function you call, store it in a variable, and then pass that value into the next function.

GOOD:
```
var x = 3;
x = a(x);
x = b(x);
c(x);

function a(x) {
    return x + 10; //good: the value is returned to the caller rather than passed to b().
}

function b(x) {
    return 2 * x; //good: the value is returned to the caller rather than passed to c().
}

function c(x) {
    alert(x);
}
```

Some more guidelines:

Use descriptive variable and function names

Our app should use page-wide variables where appropriate.

If there is a way to make the function easier to read by using less-cryptic syntax, use the less-cryptic syntax.

Avoid unnecessarily cryptic syntax such as ternary statements.

There are many other things we can say regarding code architecture and organization, but what is said here will go a long way.

ALGORITHM AND DATA STRUCTURE EFFICIENCY

We should choose data structures carefully. To quote Linus Torvalds (creator of Git and Linux):

"I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships."

We should be trying to find ways to get O(1) performance. That depends a lot on the data structures. We should not be looping over a list to find something, when we can store it in a map and get it in constant time.

We should not be re-creating and duplicating data structures unncessarily. We should drive our data into maps upon page load, and make those maps available to all of the functions on the page (i.e. store the maps at the page-global level).

If there is a O(n) operation, could it have been done in O(1) using a different data structure setup?

If there is a O(n^2) routine, that is low-performance. Is there a better way? If we are stuck with it, are we minimizing the number of times we have to call this routine?