Programming For Astronomy v0.001

A Guide

Contents

1. Introduction

This project started to document general methods of writing software for astronomy on the iMac / OS X platform. The notes here are used in the development of our COS Astrolab project, and were originally included in the user guide for Astrolab. However, as these programming notes and standards are not specific to one application, we have split them out into their own manual. The notes may include examples from Astrolab, but will include other examples unrelated to that project. These notes may also be of use for other science programming projects, not just astronomy.

The first section in this document covers JavaScript, more sections covering other environments may be added later. Note that some of the styles are similar in the 'C', Java, JavaScript and Objective C programming languages as they have tended to evolve from roots in 'C'.

Languages we have come across so far are

FORTRAN
C
C++
Objective C
JAVA
JavaScript
Pascal
Python

Clearly it would be better to avoid having to develop in too may formats, and the easiest development environments and user interfaces are preferable. Weird and trendy syntax is to be avoided as far as possible – clarity is again preferable.

Astronomy, Astrophysics and Cosmology have differing definitions. Astronomy can be described as the observational subject, Astrophysics attempts to explain the observations and Cosmology is the study of the Universe as a whole. As there is much overlap between in the subjects in books and courses we will usually describe everything under 'Astronomy' for the purposes of this project.

2. JavaScript use in Astrolab

Note that the html Comment 'tags' at the start and end of our .js files are added inside the script tag for older browsers which may not cope with code in the body of the html page, however this may not matter when code is loaded inside head tag, or newer browsers may not require these comment tags at all anyway. They cause no issues if we leave them in. I have seen errors reported when running pages through W3C HTML Validator if my 'strict code' is missing, so even if the error could be ignored I have see no point in having errors reported.

The two back slash characters stop the closing comment 'tag' being processed when the comment encloses JavaScript, again this may only matter when code is in the body of a HTML page.

To be consistent this comment standard is used within all style and JavaScript tags in this project, although they may all be removed eventually if they do not cause browser issues, to simplify coding, unless intended to contain actual notes like this.

(note do not use repeated // for comments over several consecutive lines in Javascript, use the /* */ enclosing comments instead)

Do not include the script tag in xx.js files.

2.1. Constants

The ALFoundation.js file contains Global constant variables used in all calculation code (Note: do not include the script tag in xx.js files).

Java standard for naming constants is ALL_CAPS with words separated by _ character.

The object format below is for defining a direct instance of an object - in this case constant 'objects' for use in other Astrolab pages. When the .js file is loaded in a page the values can be referred to as per the example below.

```
var CONST = { }
```

is an example of an object literal, consisting of a comma separated list of property specifications enclosed in curly braces. Each property specification in an object literal consists of a property name followed by a colon and the property value, e.g.

```
var SPEED_OF_LIGHT = { value: 2.99792458e8, error:0, desc:"Speed of Light",
units:"m/s",  origin:"2010 CODATA"};
```

We have not stuck to the strict constant naming format within the object, but we will try and use the Java standard naming within apps so example for a constant for proton mass:

```
var PROTON_MASS = CONST.mp  // declare at start of code as a global var
```

This makes the name for readable and rather than typing CONST.mp all through the code, but the final result should be the same. It is also possible to create an object with a function for constants and properties, and then create a 'new' instance of that object for every constant. However the method below seems to be the simplest approach in this case. It is also possible to create objects within objects as below , so CONST.AU.value and CONST.SUN.RADIUS.value can show properties at different levels.

```
var CONST = {

// PHYSICAL CONSTANTS
             AU: { value:149597870700, error:0, desc:"AU in m as fixed by IAU Aug
2012",  units:"m", origin:"" },

               C: { value: 2.99792458e8, error:0, desc:"Speed of Light",
units:"m/s",  origin:""},

               G: { value:6.67384e-11, error:8.0e-15, desc:"Gravitational
constant", units:"N m2 kg-2 / m3 kg-1 s-2", origin:"" },


      MOON: {

                     RADIUS: { value:1738000.0 }

            },

      SUN: {

                    RADIUS: { value:6.96E8 },
                  luminosity: {value:3.826E+26 },
                    EFF_T_SUN: { value:5770 }
           }
};
```

2.2. Example Calculation Page - <head> layout

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<title>Calculation Template ------------ [cCalc000Template]</title>

<!-- Load common code for calculation pages -->

<link rel="stylesheet" href="../fDefault.css" type="text/css" />

<script language="JavaScript" src="../fEnvironment.js" type="text/javascript">
</script>

<script language="JavaScript" src="../cJscriptLib/ALFoundationObj.js"
type="text/javascript">
</script>

<script language="JavaScript" type="text/javascript">
<!--

// Note missing out the start and end comment lines within 'script' tag can give
html validation errors.

function pageStart() {

  displayMenu();

}

function displayMenu() {

if (APP_ENV == "PDA")   {

    var menu = document.getElementById("menuCell");

    menu.innerHTML = '<a title="Main Page" href="../mMainTab.html"> mMainTab</a>
&gt; cCalc000Template';

/*
possible alternative route add a cell on end of row with text
var firstRow = document.getElementById("headerTable").rows[0];
var menu = firstRow.insertCell(-1);
menu.innerHTML = '<a href="mMain.html"> mMain </a> &gt cInfo016AppMap';
*/

  }

}

// -->
</script>

<!-- End of common code -->

<!-- Above is Standard Header For All Calculation Application Pages -->


<!-- Page Specific Code for Calculation Pages-->

<script language="JavaScript" type="text/javascript">

<!--

/*
Description:
Short program description
*/
```

```
// define all Global variables


var width = 0;          // Formation time coefficient
var length = 0;         // Crater diameter in km
var decPlaces = 4;      // initial number of decimal places for results formatting

function main() {

// local variables

area = 0;

// get input parameters

    width = document.inputDataForm1.input4.value;
    length = document.inputDataForm1.input5.value;

// calculations

     area = width * length;
```

3. The JavaScript Style Guide

The following sections are still under revision.

Their main function is to list recognised standards used in JavaScript code for this project, which in turn should make the code easy to read and understand how it works. The lessons learned can then be used to write further JavaScript code in this or any other project or even move on to code in Java itself.

Sections cover:

Code Style
Comments
Commenting Scripts in JavaScript
Function Declarations
Indentation
JavaScript Files
Names
Quotes
Statements
Symbols
Variable Declarations
Whitespace

These sections are based on the Sun documents under Code Conventions for the Java Programming Language, but it is modified for JavaScript, as this language is not exactly the same as the Java language, but we try to stick to the same naming conventions and formatting which are mainly designed to make the code readable and easier to maintain.

When storing files on the development platform all directory names are in interCaps convention. Web pages in the application follow Java interCaps function name convention and may have a lower case prefix of 'f'ramework or 'c'ore. Core functions are intended to be independent of the framework and can be run under mobile or standalone versions of the application. These are the parts of the application that could be ported to another language / platform.

Supporting documents follow the Java ClassName convention.

There is no real connection with Java, this naming is simply chosen for convenience and consistency.

3.1.1. Code Style

Always put else on its own line, as shown above.

Don't use else after return, i.e.

```
if (x < y)
    return -1;
if (x > y)
    return 1;
return 0;
```

Both `i++` and `++i` are acceptable.

Name inline functions, this makes it easier to debug them. Just assigning a function to a property doesn't name the function, you should to do this:

```
var offlineObserver = {
  observe: function offlineObserve(aSubject, aTopic, aState)
  {
    if (aTopic == "network:offline-status-changed")
      setOfflineUI(aState == "offline");
  }
}
```

3.1.2. Comments

Be generous with comments. It is useful to leave information that may be read after by people (possibly yourself) who will need to understand what you have done. The comments should be well-written and clear, just like the code they are annotating.

It is important that comments be kept up-to-date. Erroneous comments can make programs even harder to read and understand.

Make comments meaningful. Focus on what is not immediately visible. Don't waste the reader's time with stuff like

```
i = 0; // Set i to zero.
```

Generally use line comments. Save block comments (/* */)for formal documentation, comments over more than one line, and for commenting out.

3.1.3. Commenting Scripts in JavaScript

For older browsers which may not support JavaScript it is possible to hide the script as follows - the JavaScript engine allows the string "<!- -" to occur at the start of a <script> element, and ignores further characters until the end of the line. JavaScript interprets "//" as starting a comment extending to the end of the current line. This is needed to hide the string "-->" from the JavaScript parser in older browsersat the end of a script before </script>. We use this standard to enclose all code within JavaScript and style tags, example below:

```
<script type="text/javascript">
<!--  to hide script contents from old browsers
function square(i){
  document.write("Value of I is", i ,"<BR>")
  return i * i
 }
// end hiding contents from old browsers  -->
</script>
```

As applications using JavaScript require a browser that does support it then these comments may be irrelevant but we are leaving them in all code for now, as some code may be in the <body> of a html page as well as the <head>.

3.1.4. Form Fields

ID vs. Name in Form Fields

When creating a form and form field, you should give them both an id and a name and make them the same, e.g.

```
<form id="form1" name="form1">
<input type="text" id="form1Field1" name="form1Field1" />
<input type="text" id="form1Field2" name="form1Field2" />
</form>
```

This is because browsers use the name field when they submit to a server, but when working with JavaScript it's best to use id (and the associated document.getElementById()) to identify elements. You could use something like document.getElementsByName(), but it is a lot harder to work with, because names aren't guaranteed to be unique and thus that function returns a collection of elements, not just one.

You should also always name your name and id the same thing (and not reuse that identifier anywhere else), because of IE's poor implementation of getElementById() which can sometimes return an element whose "name" matches the "id" you asked for.

Always use both, make them the same string, and make that string unique in the document.

3.1.5. Function Declarations

All functions should be declared before they are used. Inner functions should follow the `var` statement. This helps make it clear what variables are included in its scope.

There should be no space between the name of a function and the `(` (left parenthesis) of its parameter list. There should be one space between the `)` (right parenthesis) and the `{` (left curly brace) that begins the statement body. The body itself is indented four spaces. The `}` (right curly brace) is aligned with the line containing the beginning of the declaration of the function.

```
function outer(c, d) {
    var e = c * d;

    function inner(a, b) {
        return (e * a) + b;
        }

    return inner(0, 1);
    }
```

This convention works well with JavaScript because in JavaScript, functions and object literals can be placed anywhere that an expression is allowed. It provides the best readability with inline functions and complex structures.

```
    function getElementsByClassName(className) {
        var results = [];
        walkTheDOM(document.body, function (node) {
            var a;                  // array of class names
            var c = node.className; // the node's classname
            var i;                  // loop counter

// If the node has a class name, then split it into a list of simple
names.
// If any of them match the requested name, then append the node to
the set of results.

            if (c) {
                a = c.split(' ');
                for (i = 0; i < a.length; i += 1) {
                    if (a[i] === className) {
                        results.push(node);
                        break;
```

```
                    }
                }
            }
        });
        return results;
    }
```

If a function literal is anonymous, there should be one space between the word `function` and the `(` (left parenthesis). If the space is omited, then it can appear that the function's name is `function`, which is an incorrect reading.

```
    div.onclick = function (e) {
        return false;
    };

    that = {
        method: function () {
            return this.datum;
        },
        datum: 0
    };
```

Use of global functions should be minimized.

When a function is to be invoked immediately, the entire invocation expression should be wrapped in parens so that it is clear that the value being produced is the result of the function and not the function itself.

```
var collection = (function () {
    var keys = [], values = [];

    return {
        get: function (key) {
            var at = keys.indexOf(key);
            if (at >= 0) {
                return value[at];
            }
        },
        set: function (key, value) {
            var at = keys.indexOf(key);
            if (at < 0) {
                at = keys.length;
            }
```

```
            keys[at] = key;
            value[at] = value;
        },
        remove: function (key) {
            var at = keys.indexOf(key);
            if (at >= 0) {
                keys.splice(at, 1);
                value.splice(at, 1);
            }
        }
    };
}());
```

3.1.6. Indentation

The basic unit of indentation is four spaces.

Any further indents required are in blocks of four spaces.

Tabs are not to be used at all because there still is not a standard for the placement of tabstops. The use of spaces can produce a larger filesize, but the size is not significant if the application is split into simple pages. Line Length

Avoid lines longer than 80 characters. When a statement will not fit on a single line, it may be necessary to break it. Place the break after an operator, ideally after a comma. A break after an operator decreases the likelihood that a copy-paste error will be masked by semicolon insertion. The next line should be indented 8 spaces.

A blank line is left between sections in JavaScript and HTML code.

```
var result = prompt(aMessage,
               aInitialValue,
                  aCaption);

var IOService = Components.classes["@mozilla.org/network/io-
service;1"]

    .getService(Components.interfaces.nsIIOService);
```

3.1.7. JavaScript Files

JavaScript programs should be stored in and delivered as `.js` files.

JavaScript code should not be embedded in HTML files unless the code is specific to a single session, otherwise the code in HTML adds significantly to pageweight.

3.1.8. Names

Names should be formed from the 26 upper and lower case letters (`A` .. `Z`, `a` .. `z`), the 10 digits (`0` .. `9`), and `_` (underbar). Avoid use of international characters because they may not read well or be understood everywhere. Do not use `$` (dollar sign) or `\` (backslash) in names.

Do not use _ (underbar) as the first character of a name. It is sometimes used to indicate privacy, but it does not actually provide privacy. If privacy is important, use the forms that provide private members. Avoid conventions that demonstrate a lack of competence.

Most variables and functions should start with a lower case letter.

Constructor functions which must be used with the `new` prefix should start with a capital letter. JavaScript issues neither a compile-time warning nor a run-time warning if a required `new` is omitted. Bad things can happen if `new` is not used, so the capitalization convention is the only defense we have.

- Use interCaps for names and enumeration values;
- JavaScript does not have macros or constants, however consistency is important, so constants are all capitals with the "_" between words as in Java, e.g. `UPPER_CASE`.
- Convenience constants for interface names should be prefixed with nsI, e.g.

```
const nsISupports = Components.interfaces.nsISupports;
const nsIWBN = Components.interfaces.nsIWebBrowserNavigation;
```

- Enumeration values should be prefixed with the letter `k`, e.g. `const kDisplayModeNormal = 0;`.
- Global variables should be prefixed with the letter `g`, e.g. `var gFormatToolbar;`.
- Arguments (parameter names) should be prefixed with the letter `a`.
- Event handler functions should be prefixed with the word `on`, in particular try to use the names `onLoad`, `onDialogAccept`, `onDialogCancel` etc. where this is unambiguous.
- Function names, local variables and object members have no prefix.
- Try to declare local variables as near to their use as possible; try to initialize every variable.

3.1.9. Quotes

Always Quote Attribute Values

For HTML Attribute values should always be enclosed in quotes.

Double style quotes are the most common, single style quotes are also allowed, but we will always try to use double.

In JavaScript we will always use double quotes around strings, but in some rare situations, like when the attribute value itself contains quotes, it is necessary to use single quotes:

```
var name = 'John "ShotGun" Nelson'
```

OR

```
  document.write('<a title="Main Page" href="mmain.html" >Main
Page</a> &gt Drake Equation');
```

3.1.10. Statements

3.1.11. Simple Statements

Each line should contain at most one statement. Put a `;` (semicolon) at the end of every simple statement. Note that an assignment statement which is assigning a function literal or object literal is still an assignment statement and must end with a semicolon.

JavaScript allows any expression to be used as a statement. This can mask some errors, particularly in the presence of semicolon insertion. The only expressions that should be used as statements are assignments and invocations.

3.1.12. Compound Statements

Compound statements are statements that contain lists of statements enclosed in `{ }` (curly braces).

- The enclosed statements should be indented four more spaces.
- The `{` (left curly brace) should be at the end of the line that begins the compound statement.
- The `}` (right curly brace) should begin a line and be indented to align with the beginning of the line containing the matching `{` (left curly brace).
- Braces should be used around all statements, even single statements, when they are part of a control structure, such as an `if` or `for` statement. This makes it easier to add statements without accidentally introducing bugs.

3.1.13. Labels

Statement labels are optional. Only these statements should be labeled: `while`, `do`, `for`, `switch`.

`return` Statement

A `return` statement with a value should not use `( )` (parentheses) around the value. The return value expression must start on the same line as the `return` keyword in order to avoid semicolon insertion.

`if` Statement

The `if` class of statements should have the following form:

```
if (condition) {
    statements
}

if (condition) {
    statements
} else {
    statements
}

if (condition) {
    statements
} else if (condition) {
    statements
} else {
    statements
}
```

`for` Statement

A `for` class of statements should have the following form:

```
for (initialization; condition; update) {
    statements
}
```

```
for (variable in object) {
    if (filter) {
        statements
    }
}
```

The first form should be used with arrays and with loops of a predeterminable number of iterations.

The second form should be used with objects. Be aware that members that are added to the prototype of the *object* will be included in the enumeration. It is wise to program defensively by using the `hasOwnProperty` method to distinguish the true members of the *object*:

```
for (variable in object) {
    if (object.hasOwnProperty(variable)) {
        statements
    }
}
```

`while` Statement

A `while` statement should have the following form:

```
while (condition) {
    statements
}
```

`do` Statement

A `do` statement should have the following form:

```
do {
    statements
} while (condition);
```

Unlike the other compound statements, the `do` statement always ends with a `;` (semicolon).

`switch` Statement

A `switch` statement should have the following form:

```
switch (expression) {
case expression:
    statements
default:
    statements
}
```

Each `case` is aligned with the `switch`. This avoids over-indentation.

Each group of *statements* (except the `default`) should end with `break`, `return`, or `throw`. Do not fall through.

`try` Statement

The `try` class of statements should have the following form:

```
try {
    statements
} catch (variable) {
    statements
}

try {
    statements
} catch (variable) {
```

```
        statements
    } finally {
        statements
     }
```

`continue` Statement

Avoid use of the `continue` statement. It tends to obscure the control flow of the function.

`with` Statement

The `with` statement should not be used.

3.1.14. Symbols

- Spaces around braces used for in-line functions or objects, except before commas or semicolons, e.g.

```
function valueObject(aValue) { return { value: aValue }; };
```

- Otherwise function braces must always be on their own line, i.e.

```
function toOpenWindow(aWindow)
{
  aWindow.document.commandDispatcher.focusedWindow.focus();
}
```

- Otherwise spaces are not necessary inside brackets e.g. parameter lists, array subscripts. This includes wrapping an in-line JavaScript object in parentheses, or the `for (;;)` construct - the space normally required after the first semicolon is inhibited by the second semicolon, the space after the second semicolon is inhibited by the close parenthesis.

- Prefer double quotes, except in in-line event handlers or when quoting double quotes.

- Braces are not indented relative to their parent statement. Stick to the style used in existing files, but when creating new files you may choose your favourite of the following acceptable constructs:

```
if (dlmgrWindow)
  dlmgrWindow.focus();
else
  dlmgr.open(window, null);
```

```
if (dlmgrWindow) {
  dlmgrWindow.focus();
} else {
  dlmgr.open(window, null);
}
```

```
if (dlmgrWindow) {
  dlmgrWindow.focus();
}
else {
  dlmgr.open(window, null);
}
```

```
if (dlmgrWindow)
{
```

```
  dlmgrWindow.focus();
}
else
{
  dlmgr.open(window, null);
}
```

- Use \uXXXX unicode constants for non-ASCII characters. The character set for XUL, DTD, script and properties files is UTF-8 which is not easily readable.

3.1.15. Variable Declarations

All variables should be declared before used. JavaScript does not require this, but doing so makes the program easier to read and makes it easier to detect undeclared variables that may become implied globals. Implied global variables should never be used.

i) JavaScript variables must start with a letter or underscore "_" and follow the interCaps convention

ii) JavaScipt is case sensitive.

The `var` statements should be the first statements in the function body.

It is preferred that each variable be given its own line and comment. They should be listed in alphabetical order.

```
var currentEntry; // currently selected table entry
var level;        // indentation level
var size;         // size of table
```

JavaScript does not have block scope, so defining variables in blocks can confuse programmers who are experienced with other C family languages. Define all variables at the top of the function.

Use of global variables should be minimized.

3.1.16. Variable Scope and Lifetime

The scope is the region of the program for which the variable is declared. Variables may be either Global, or Local. Local variables are available only within the section of code in which they were defined. Changes to Local variables are not reflected 'outside' their area of definition. When you exit this area, the variable is destroyed. Global variables are available to other JavaScript code. Generally, variables declared as "var" are Global variables. Variables declared inside a function as "var" are local to the function. Variables defined inside a function without the "var" are Global variables. The lifetime of Global variables starts when they are declared, and ends when the page is closed.

```
<script type="text/javascript">

var altitude = 5;  //GLOBAL

function square( ) {
   base = 17;                          //GLOBAL
   sqr = 0.5*(base + altitude);
   return sqr;
}
```

```
function perimeter() {
   var side = 7.5;                          //LOCAL
   prm = 2*side + base;
   return prm;

}
</script>
```

3.1.17. Whitespace

Blank lines improve readability by setting off sections of code that are logically related.

- Lines should not contain trailing spaces, even after binary operators, commas or semicolons.
- A keyword followed by `(` (left parenthesis) should be separated by a space.

```
while (true) {
```

- A blank space should not be used between a function value and its `(` (left parenthesis). This helps to distinguish between keywords and function invocations.
- All binary operators except `.` (period) and `(` (left parenthesis) and `[` (left bracket) should be separated from their operands by a space.
- No space should separate a unary operator and its operand except when the operator is a word such as `typeof`.
- Separate binary operators with spaces.
- Spaces after commas and semicolons, but not before.
- Each ; (semicolon) in the control part of a `for` statement should be followed with a space.
- Spaces after keywords, e.g. `if (x > 0)`.
- One (or two) blank lines between block definitions. Also consider breaking up large code blocks with blank lines.
- End the file with a newline. (This applies mainly to emacs users.)

4. References:

JavaScript – The Definitive Guide, David Flanagan, O'Reilly, 4th Edition

Boccas et al. 2006, "Laser Guide Star Upgrade of Altair at Gemini North", from the Documents link on the ALTAIR web page.
Gemini web pages for NIRI and ALTAIR instruments:
http://www.gemini.edu/sciops/instruments/
Harris W.E., 1996, AJ, 112, 1487, "A Catalogue of Parameters for Globular Clusters in the Milky Way."
Hodapp K.W. et al. 2003, PASP, 115, 1388, "The Gemini Near-Infrared Imager (NIRI)"

Software references

DeepSkyStacker 3.2.2 http://deepskystacker.free.fr/english
FITSview for Microsoft Windows http://www.nrao.edu/software/fitsview/fvwin.html

5. Spreadsheets - Using Excel in Astronomy

Excel stores dates and times as a number representing the number of days since 1900-Jan-0, plus a fractional portion of a 24 hour day: `ddddd.tttttt` . This is called a **serial date**, or **serial date-time**.

5.1. Dates

The integer portion of the number, `ddddd`, represents the number of days since 1900-Jan-0. For example, the date 19-Jan-2000 is stored as 36,544, since 36,544 days have passed since 1900-Jan-0. The number 1 represents 1900-Jan-1. It should be noted that the number 0 does **not** represent 1899-Dec-31. It does not. If you use the `MONTH` function with the date 0, it will return January, not December. Moreover, the `YEAR` function will return 1900, not 1899.

Actually, this number is one greater than the actual number of days. This is because Excel behaves as if the date 1900-Feb-29 existed. It did not. The year 1900 was not a leap year (the year 2000 is a leap year). In Excel, the day after 1900-Feb-28 is 1900-Feb-29. In reality, the day after 1900-Feb-28 was 1900-Mar-1 . This is not a "bug". Indeed, it is by design. Excel works this way because it was truly a bug in Lotus 123. When Excel was introduced, 123 has nearly the entire market for spreadsheet software. Microsoft decided to continue Lotus' bug, in order to fully compatible. Users who switched from 123 to Excel would not have to make any changes to their data. As long as all your dates later than 1900-Mar-1, this should be of no concern.

5.2. Times

The fractional portion of the number, `ttttt`, represents the fractional portion of a 24 hour day. For example, 6:00 AM is stored as 0.25, or 25% of a 24 hour day. Similarly, 6PM is stored at 0.75, or 75% percent of a 24 hour day.

As you can see, any date and time can be stored as the sum of the date and the time. For example, 3PM on 19-Jan-2000 is stored internally as 36544.625. When you enter a time without a value, such as entering 15:00 into a cell, the date portion is a zero. The zero indicates that there is no date associated with the time. You should remember that entering just a time does **not** automatically put in the current date.