Good or bad programming

A programming language is a human language and not a computer language.

It is written by humans. It must be read and understood by humans.

A program must be comprehensible to other people than the original author!

*After a (few) year(s) even (s)he h*self cannot remember h* own "sophisticated" obscurities.*

A computer cannot handle anything but binary code, a long series of all ones and zeros.
To run a humanly readable program it must first convert it to binary (by some other program).

Next are examples in JScript, which is Microsoft's implementation of javascript that can be used in the Windows Script Host and does for example not support the `const` and `let` keywords.

Although the value used in the example below may be expected to never change,
it should make clear how to write readable code.

VERY VERY BAD programming (need to update all code if value must be changed, see example further below):

using `86400` throughout the code.

VERY BAD programming (how you would do it if you're thinking like a mathematician: use just SOME variable with an as short as possible name that is totally meaningless to an outsider):

```
var a = 86400;
```

BAD programming (although name now based on meaning):

```
var spd = 86400;
```

"better" **BAD** programming (comment "explains" what it is):

```
var spd = 86400;/*s/d*/
```

even better (extensive comment instead of cryptic) but still **BAD** programming:

```
var spd = 86400;      // number of seconds per day
```

reasonably good programming (self-explaining name eliminates the need of any comment and implicitly makes all code properly readable from top to bottom):

```
var secondsPerDay = 86400;
```

BETTER programming (shows where the number comes from, at the minor cost of a few CPU ticks):

```
var secondsPerDay = 24*60*60;
```

EVEN BETTER (gives each and every number a meaning in a ready-to-use variable with a name that FULLY describes its meaning):

```
var secondsPerMinute  = 60;
var minutesPerHour    = 60;
var hoursPerDay       = 24;
var secondsPerDay     = secondsPerMinute*minutesPerHour*hoursPerDay;
```

BEST is a "complete module" with all variables ready for use, even if some are not actually used.
It can be in a separate file to be included in many programs:

```
// =============================================================================
// Description: definition of ratios of time units
// Copyright 2021 © Henk Reints, http://henk-reints.nl
// =============================================================================
// Notes:
//   1) the month is a rather vague amount of time which cannot
//      be considered a true UNIT, hence it is not supported here;
//   2) all variables below are global,
//      which is not considered good programming!
// =============================================================================
// defined values:
// ---------------
var secondsPerMinute                             = 60;
var minutesPerHour                               = 60;
var hoursPerDay                                  = 24;
var daysPerWeek                                  = 7;
var yearsPerCentury                              = 100;
var daysPerNormalYear                            = 365;
var daysPerLeapYear                              = 366;
var yearsPerJulianLeapCycle                      = 4;    // EVERY 4 years
var yearsPerGregorianLeapCycle                   = 400;
var skippedLeapYearsPerGregorianLeapCycle        = 3;
// ----------------
// computed values:
// ----------------
var secondsPerHour              = secondsPerMinute*minutesPerHour;
//  ----
var minutesPerDay               = minutesPerHour*hoursPerDay;
var secondsPerDay               = secondsPerMinute*minutesPerDay;
//  ----
var hoursPerWeek                = hoursPerDay*daysPerWeek;
var minutesPerWeek              = minutesPerHour*hoursPerWeek;
var secondsPerWeek              = secondsPerMinute*minutesPerWeek;
//  ----
var daysPerJulianLeapCycle =
        (yearsPerJulianLeapCycle-1)*daysPerNormalYear+daysPerLeapYear;
var weeksPerJulianLeapCycle     = daysPerJulianLeapCycle/daysPerWeek;
var hoursPerJulianLeapCycle     = hoursPerDay*daysPerJulianLeapCycle;
var minutesPerJulianLeapCycle   = minutesPerHour*hoursPerJulianLeapCycle;
var secondsPerJulianLeapCycle   = secondsPerMinute*minutesPerJulianLeapCycle;
//  ----
var daysPerJulianYear           = daysPerJulianLeapCycle/yearsPerJulianLeapCycle;
var weeksPerJulianYear          = daysPerJulianYear/daysPerWeek;
var hoursPerJulianYear          = hoursPerDay*daysPerJulianYear;
var minutesPerJulianYear        = minutesPerHour*hoursPerJulianYear;
var secondsPerJulianYear        = secondsPerMinute*minutesPerJulianYear;
// ----
var daysPerJulianCentury        = daysPerJulianYear*yearsPerCentury;
var weeksPerJulianCentury       = weeksPerJulianYear*yearsPerCentury;
var hoursPerJulianCentury       = hoursPerJulianYear*yearsPerCentury;
var minutesPerJulianCentury     = minutesPerJulianYear*yearsPerCentury;
var secondsPerJulianCentury     = secondsPerJulianYear*yearsPerCentury;
// ----
var daysPerGregorianLeapCycle =
        daysPerJulianLeapCycle*yearsPerGregorianLeapCycle/yearsPerJulianLeapCycle
        -skippedLeapYearsPerGregorianLeapCycle*(daysPerLeapYear-daysPerNormalYear);
var weeksPerGregorianLeapCycle          = daysPerGregorianLeapCycle/daysPerWeek;
var hoursPerGregorianLeapCycle          = hoursPerDay*daysPerGregorianLeapCycle;
var minutesPerGregorianLeapCycle        = minutesPerHour*hoursPerGregorianLeapCycle;
var secondsPerGregorianLeapCycle        = secondsPerMinute*minutesPerGregorianLeapCycle;
// =====================================================================================
// end of file, Copyright 2021 © Henk Reints
```

VERY BEST is to encapsulate the hole thing in an object. Please see http://henk-reints.nl/vlorp.zip for a full set of scripts of which the `.wsf` files are best executed using Cscript in a Command Prompt window.

And here is the promised example of **VERY VERY BAD** programming:

The comment should of course be: *returns a container with the angle of a full circle times the surface areas of various spheres* (please don't ask what it can be used for), and `dpc` should of course have been named: `degreesPerCircle`, not to mention the other variable names.

```
function vlorp()/*ret.cont.w/surf.ar.sph.*/
{   var dpc = 360, var r = 0, c = [];
    for (var i = 0; i < 4; i++)
    {   var b = [];
        for (var j = 0; j < 4; j++)
        {   var q = [];/*r alr. used*/
            for (var k = 0; k < 4; j++)
            {   var A = 4*Math.PI*r*r;
                q.push(dpc*A);
                r++;
            };
            b.push(q);
        };
        c.push(b);
    };
    return c;
};
```

For some reason we need to change the size of the container to say 6×6×6, so we change all fours to sixes with a simple *replace all* operation:

```
function vlorp()/*ret.cont.w/surf.ar.sph.*/
{   var dpc = 360, var r = 0, c = [];
    for (var i = 0; i < 6; i++)
    {   var b = [];
        for (var j = 0; j < 6; j++)
        {   var q = [];/*r alr. used*/
            for (var k = 0; k < 6; j++)
            {   var A = 6*Math.PI*r*r;
                q.push(dpc*A);
                r++;
            };
            b.push(q);
        };
        c.push(b);
    };
    return c;
};
```

Do you see what happend to the calculation of the actual surface area? It has become $6\pi r^2$ which we most probably do not immediately recognise. Oops! At the next production run of this very important program something goes terribly wrong because of that, so we have to reverse our *rather silly mistake* (in the Netherlands we call it an *utterly stupid error*) and we perform a *replace all* to change all sixes back to fours and because it's very urgent it is directly done in the production version. We will later copy that back to our development and test environments... Now it is more important that it's quickly repaired in production to make it once again run as always. Urh, it now says: `dpc = 340` so it thinks a circle has 340°... In production...

We should have defined:
`var numberOfRows = 4, numberOfColumns = 4, numberOfBlocks = 4;`
NOT: `nr, nc, & nb!` Make it comprehensible to a layman! But okay, **`nrows`** etc. will do.
The loops should be like: `for (var irow = 0; irow < nrows; irow++)` etc.

And what about the other variable names? Properly written the function might look like:

```
function vlorp()  // see file "vlorp.doc" for an extensive explanation
{   // parameters:
    var nrows    = 4;
    var ncolumns = 4;
    var nblocks  = 4;
    // variables:
    var radius = 0, container = [];
    // the algorithm:
    for (var irow = 0; irow < nrows; irow++)
    {   var block = [];
        for (var iblock = 0; iblock  < nblocks; iblock++)
        {   var row = [];
            for (var icolumn = 0; icolumn < ncolumns; icolumn++)
            {   row.push(vlorpValue(radius));
                radius++;
            };
            block.push(row);
        };
        container.push(block);
    };
    return container;
};
function vlorpValue(radius)
{   var degreesPerCircle = 360;
    return degreesPerCircle*surfaceAreaOfSphere(radius);
};
function surfaceAreaOfSphere(radius)
{   return 4*Math.PI*radius*radius;
};
```

And do you see how I placed the curly braces? I think it is WAY more readable then:

```
    for (var irow = 0; irow < nrows; irow++) {
      var a = b + c;};   // do something useful
```

which is more or less the standard layout. In a nested loop structure I cannot see at first glance if the braces are matching. By putting the opening brace on the next line and the closing brace in the exact very same column below it, one immediately sees h* mistakes.

❊ ❊ ❊ ❊ ❊ ❊ ❊