

**Capstone Project 1**

**DEMARK - DECENTRALIZED MARKET**

Convention Document

Version: 1.1

CODE: DEMARK-1.0

**Mentor:** Man Nguyen Duc

**Team member:**

Ha Truong Van

Dat Ngo Ha Van

Nhan Vo Hoang Quoc

Phuong Tran Nhat

**INTERNATIONAL SCHOOL**

**Project Information**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Project acronym** | DEMARK | | | | |
| **Project Title** | Demark - decentralized market for digital assets | | | | |
| **Start Date** | Aug 31, 2018 | | **End Date** | | Dec 10, 2018 |
| **Lead Institution** | International School, Duy Tan University | | | | |
| **Project Mentor & contact details** | Man Nguyen Duc  Email: mannd@duytan.edu.vn  Tel: 0904 235 945 | | | | |
| **Product Owner** | Ha Truong Van | | | | |
| **Scrum Master** | Phuong Tran Nhat | | | | |
| **Team members** | **Name** | **Email** | | **Tel** | |
|  | Ha Truong Van | vanha30111997@gmail.com | | 0969356097 | |
|  | Dat Ngo Ha Van | ngohavandat93ndc@gmail.com | | 01288446176 | |
|  | Nhan Vo Hoang Quoc | quocnhan810@gmail.com | | 01674559527 | |
|  | Phuong Tran Nhat | nhatphuongb1@gmail.com | | 01692502010 | |
|  |  |  | |  | |

**Proposal Document**

|  |  |  |  |
| --- | --- | --- | --- |
| **Document Title** | Demark Convention Document | | |
| **Reporting Period** | September 12, 2018 | | |
| **Team Information** | **Name** | **Role** |  |
|  | Nhan Vo Hoang Quoc | Team member |  |
|  | Ha Truong Van | Product owner & team member |  |
|  | Dat Ngo Ha Van | Team member |  |
|  | Phuong Tran Nhat | Scrum master & Team member |  |
| **Date** | September 12, 2018 | **Filename** | DEMARK\_CONVENTION\_1.1.docx |
| **Access** | Project and Duy Tan University Program | |  |

# **Document History**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Document History |  |
| **Version** | **Date** | **Comments** |  |
| V1.0 | September 10, 2018 | * Draft of document |  |
| V1.1 | September 12, 2018 | * Official document |  |

# **Document Approvals**

​The following signatures are required for approval of this document.

|  |  |  |
| --- | --- | --- |
| Man Nguyen Duc  *Mentor* |  | Date: |
| Ha Truong Van  *Product Owner* |  | Date: |
| Phuong Tran Nhat  *Scrum Master* |  | Date: |

TABLE OF CONTENTS

[**JavaScript coding Guidelines**](#_1t3h5sf)

[Naming convention:](#_4d34og8)

[General Guidelines](#_2s8eyo1)

[**JQuery Guidelines**](#_17dp8vu)

[**Solidity coding Guidelines**](#_3rdcrjn)

# JavaScript coding Guidelines

## Naming convention:

1. Names should not be too small like ex1, i, j, k etc. or should not be very large. Iterator variables in the loops can be an exception here.
2. Name should be simple, easy, readable and most importantly SHORT
3. Names should not start with a numeral (0-9). Such names will be illegal.
4. **Variable Names:**
5. Use Lower Camel Case (variableNameShouldBeLikeThis) for variable names.

* Name should start with single letter which defines it's data type. This helps the developer to know the behavior of the variable and gets an idea about what kind of data it has in it.
* Refer to the table below for some of the data types widely used along with the conventions.

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Data Type** | **Prefix letter** | **Sample Variable Name** |
| 1 | int | i | iCounter |
| 2 | Float | f | fPrice |
| 3 | Boolean | b | bFlag |
| 4 | String | s | sFullName |
| 5 | Object | o | oDiv |

1. Constants: There is nothing as constants in JavaScript. But if you want for particular variable developer should not try to modify the value then the name should be in all capital letters (CONSTANTNAME). Specify the prefix similar to above table.
2. Function Names / Method Names:
3. JavaScript function/method names should use Lower Camel Casing (functionNamesLikeThis)
   * In case your function returns a value. Prefix the function name with the letter indicating the type of values returned. Refer table mentioned above for Prefixing a letter for variable names.
   * Function names should be intuitive and should use concise words explaining the existence of the function.
   * **For e.g.:** if you want to name a code block that checks whether the user is of legal age or not for driving, you name the function as “isAboveEighteen()”  while this seems to serve the purpose but a more appropriate function name would be “isLegalAge()”.

## **General Guidelines**

1. Declarations with var: Always
2. Prefer ' over "
3. JS allows both single quotes or double quotes
4. But for consistency single-quotes (') are preferred to double-quotes ("). This is helpful when creating strings that include HTML.

var msg = 'This is some "HTML"';

1. Always use semicolons to terminate JS statements
2. Cache the length in the loops.

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| var foo = document.getElementsByTagName('p');  for(var i=0, len=foo.length; i<len; i++) {}; | var foo = document.getElementsByTagName('p');  for(var i=0; i<foo.length; i++) {}; |

1. Use {} Instead of new Object()   
   **Example:**

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| var o = {     name: 'Jeffrey',     lastName = 'Way',     someFunction : function() {        console.log(this.name);     }}; | var o = new Object();  o.name = 'Jeffrey';  o.lastName = 'Way';  o.someFunction = function() {     console.log(this.name);  } |

1. Use [] Instead of new Array()
2. Always use strict comparison ( === Instead of == )
3. Using try/catch (Note: Exceptions Are For Exceptional Cases)

* Try/catch are expensive
* Do not use nested try/catch, instead use try/catch at the topmost level
* Do not ignore exceptions

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| try {      doStuff();  } catch(ignore) {      log(ignore);  } | try {      doStuff();  } catch(ignore) {      // Do nothing, just ignore.  } |

* Do not use try/catch within loops.

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| try {      while(condition) {          stuff();      }  } catch(e) {      log(e);  } | while(condition) {      try {          stuff();      } catch(e) {       log(e);      }  } |

1. Curly Love, use curly braces it even if it is not necessary.

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| var bCheck = true;  if(bCheck) {  alert(bCheck);  }  else {  alert(bCheck);  } | var bCheck = true;  if(bCheck)  alert(bCheck);  else  alert(bCheck); |

1. Minimize DOM access: Accessing DOM elements with JavaScript is slow so in order to have a more responsive page, you should:

Cache references to accessed elements:

var msg = document.getElementById('someDiv') //store(cache) the object in a variable

if (msg)

{

msg.style.display = 'none'

}

1. Perform null checks for the object before accessing or updating any of its properties. Refer below the sample for same:

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| var msg = document.getElementById('someWrongDivName')  if (msg)  {  msg.style.display = 'none'  }  else  {  alert('No error shown')  } | var msg = document.getElementById('someWrongDivName')    msg.style.display = 'none' //msg is null here |

# **JQuery Guidelines**

1. Use ID selector whenever possible. Finding a DOM element by its ID is the fastest way, both in JavaScript and in jQuery. Whenever possible, you should always use the ID selector instead of using classes or tag names, or other ways.
2. Avoid Loops. Nested DOM Selectors can perform better. Avoid unnecessary loops. If possible, use the selector engine to address the elements that are needed
3. Don't mix CSS with jQuery
4. Avoid multiple $(document).ready() calls
5. $.ajax performs a massive amount of work to allow us the ability to successfully make asynchronous requests across all browsers. You can use $.ajax method directly and exclusively for all your AJAX requests
6. If we have an element with id “refiner” and we want to add two classes “addColor” and “addBackground”, we can do it by putting the two class names in addClass method separated by a space.
7. Leverage Event Delegation (a.k.a. Bubbling). Every event (e.g. click, mouseover, etc.) in JavaScript “bubbles” up the DOM tree to parent elements. This is incredibly useful when we want many elements (nodes) to call the same function. Instead of binding an event listener function too many nodes — very inefficient — you can bind it once to their parent, and have it figure out which node triggered the event. For example, say we are developing a large form with many inputs, and want to toggle a class name when selected.

|  |
| --- |
| **// Inefficient**  $('#myList li).bind('click', function(){  $(this).addClass('clicked');  // do stuff  });  **// Instead, we should listen for the click event at the parent level:**  $('#myList).bind('click', function(e){  var target = e.target, // e.target grabs the node that triggered the event.  $target = $(target);  // wraps the node in a jQuery object  if (target.nodeName === 'LI') {  $target.addClass('clicked');  // do stuff  }  }); |

# **Solidity coding Guidelines**

## **Introduction**

This guide is intended to provide coding conventions for writing solidity code. This guide should be thought of as an evolving document that will change over time as useful conventions are found and old conventions are rendered obsolete.

Many projects will implement their own style guides. In the event of conflicts, project specific style guides take precedence.

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.

But most importantly: know when to be inconsistent -- sometimes the style guide just doesn't apply. When in doubt, use your best judgement. Look at other examples and decide what looks best. And don't hesitate to ask!

## **Code Layout**

**Indentation**

Use 4 spaces per indentation level.

**Tabs or Spaces**

Spaces are the preferred indentation method.

Mixing tabs and spaces should be avoided.

**Blank Lines**

Surround top level declarations in solidity source with two blank lines.

Yes::

contract A {

...

}

contract B {

...

}

contract C {

...

}

No::

contract A {

...

}

contract B {

...

}

contract C {

...

}

Within a contract surround function declarations with a single blank line.

Blank lines may be omitted between groups of related one-liners (such as stub functions for an abstract contract)

Yes::

contract A {

function spam() public;

function ham() public;

}

contract B is A {

function spam() public {

...

}

function ham() public {

...

}

}

No::

contract A {

function spam() public {

...

}

function ham() public {

...

}

}

.. \_maximum\_line\_length:

**Maximum Line Length**

Wrapped lines should conform to the following guidelines.

1. The first argument should not be attached to the opening parenthesis.

2. One, and only one, indent should be used.

3. Each argument should fall on its own line.

4. The terminating element, :code:`);`, should be placed on the final line by itself.

Function Calls

Yes::

thisFunctionCallIsReallyLong(

longArgument1,

longArgument2,

longArgument3

);

No::

thisFunctionCallIsReallyLong(longArgument1,

longArgument2,

longArgument3

);

thisFunctionCallIsReallyLong(longArgument1,

longArgument2,

longArgument3

);

thisFunctionCallIsReallyLong(

longArgument1, longArgument2,

longArgument3

);

thisFunctionCallIsReallyLong(

longArgument1,

longArgument2,

longArgument3

);

thisFunctionCallIsReallyLong(

longArgument1,

longArgument2,

longArgument3);

Assignment Statements

Yes::

thisIsALongNestedMapping[being][set][to\_some\_value] = someFunction(

argument1,

argument2,

argument3,

argument4

);

No::

thisIsALongNestedMapping[being][set][to\_some\_value] = someFunction(argument1,

argument2,

argument3,

argument4);

Event Definitions and Event Emitters

Yes::

event LongAndLotsOfArgs(

adress sender,

adress recipient,

uint256 publicKey,

uint256 amount,

bytes32[] options

);

LongAndLotsOfArgs(

sender,

recipient,

publicKey,

amount,

options

);

No::

event LongAndLotsOfArgs(adress sender,

adress recipient,

uint256 publicKey,

uint256 amount,

bytes32[] options);

LongAndLotsOfArgs(sender,

recipient,

publicKey,

amount,

options);

**Source File Encoding**

UTF-8 or ASCII encoding is preferred.

**Imports**

Import statements should always be placed at the top of the file.

Yes::

import "owned";

contract A {

...

}

contract B is owned {

...

}

No::

contract A {

...

}

import "owned";

contract B is owned {

...

}

**Order of Functions**

Ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier.

Functions should be grouped according to their visibility and ordered:

- constructor

- fallback function (if exists)

- external

- public

- internal

- private

Within a grouping, place the ``view`` and ``pure`` functions last.

Yes::

contract A {

function A() public {

...

}

function() public {

...

}

// External functions

// ...

// External functions that are view

// ...

// External functions that are pure

// ...

// Public functions

// ...

// Internal functions

// ...

// Private functions

// ...

}

No::

contract A {

// External functions

// ...

// Private functions

// ...

// Public functions

// ...

function A() public {

...

}

function() public {

...

}

// Internal functions

// ...

}

**Whitespace in Expressions**

Avoid extraneous whitespace in the following situations:

Immediately inside parenthesis, brackets or braces, with the exception of single line function declarations.

Yes::

spam(ham[1], Coin({name: "ham"}));

No::

spam( ham[ 1 ], Coin( { name: "ham" } ) );

Exception::

function singleLine() public { spam(); }

Immediately before a comma, semicolon:

Yes::

function spam(uint i, Coin coin) public;

No::

function spam(uint i , Coin coin) public ;

More than one space around an assignment or other operator to align with

another:

Yes::

x = 1;

y = 2;

long\_variable = 3;

No::

x = 1;

y = 2;

long\_variable = 3;

Don't include a whitespace in the fallback function:

Yes::

function() public {

...

}

No::

function () public {

...

}

**Control Structures**

The braces denoting the body of a contract, library, functions and structs

should:

\* open on the same line as the declaration

\* close on their own line at the same indentation level as the beginning of the

declaration.

\* The opening brace should be proceeded by a single space.

Yes::

contract Coin {

struct Bank {

address owner;

uint balance;

}

}

No::

contract Coin

{

struct Bank {

address owner;

uint balance;

}

}

The same recommendations apply to the control structures ``if``, ``else``, ``while``,

and ``for``.

Additionally there should be a single space between the control structures

``if``, ``while``, and ``for`` and the parenthetic block representing the

conditional, as well as a single space between the conditional parenthetic

block and the opening brace.

Yes::

if (...) {

...

}

for (...) {

...

}

No::

if (...)

{

...

}

while(...){

}

for (...) {

...;}

For control structures whose body contains a single statement, omitting the

braces is ok \*if\* the statement is contained on a single line.

Yes::

if (x < 10)

x += 1;

No::

if (x < 10)

someArray.push(Coin({

name: 'spam',

value: 42

}));

For ``if`` blocks which have an ``else`` or ``else if`` clause, the ``else`` should be

placed on the same line as the ``if``'s closing brace. This is an exception compared

to the rules of other block-like structures.

Yes::

if (x < 3) {

x += 1;

} else if (x > 7) {

x -= 1;

} else {

x = 5;

}

if (x < 3)

x += 1;

else

x -= 1;

No::

if (x < 3) {

x += 1;

}

else {

x -= 1;

}

**Function Declaration**

For short function declarations, it is recommended for the opening brace of the

function body to be kept on the same line as the function declaration.

The closing brace should be at the same indentation level as the function

declaration.

The opening brace should be preceded by a single space.

Yes::

function increment(uint x) public pure returns (uint) {

return x + 1;

}

function increment(uint x) public pure onlyowner returns (uint) {

return x + 1;

}

No::

function increment(uint x) public pure returns (uint)

{

return x + 1;

}

function increment(uint x) public pure returns (uint){

return x + 1;

}

function increment(uint x) public pure returns (uint) {

return x + 1;

}

function increment(uint x) public pure returns (uint) {

return x + 1;}

You should explicitly label the visibility of all functions, including constructors.

Yes::

function explicitlyPublic(uint val) public {

doSomething();

}

No::

function implicitlyPublic(uint val) {

doSomething();

}

The visibility modifier for a function should come before any custom

modifiers.

Yes::

function kill() public onlyowner {

selfdestruct(owner);

}

No::

function kill() onlyowner public {

selfdestruct(owner);

}

For long function declarations, it is recommended to drop each argument onto

it's own line at the same indentation level as the function body. The closing

parenthesis and opening bracket should be placed on their own line as well at

the same indentation level as the function declaration.

Yes::

function thisFunctionHasLotsOfArguments(

address a,

address b,

address c,

address d,

address e,

address f

)

public

{

doSomething();

}

No::

function thisFunctionHasLotsOfArguments(address a, address b, address c,

address d, address e, address f) public {

doSomething();

}

function thisFunctionHasLotsOfArguments(address a,

address b,

address c,

address d,

address e,

address f) public {

doSomething();

}

function thisFunctionHasLotsOfArguments(

address a,

address b,

address c,

address d,

address e,

address f) public {

doSomething();

}

If a long function declaration has modifiers, then each modifier should be

dropped to its own line.

Yes::

function thisFunctionNameIsReallyLong(address x, address y, address z)

public

onlyowner

priced

returns (address)

{

doSomething();

}

function thisFunctionNameIsReallyLong(

address x,

address y,

address z,

)

public

onlyowner

priced

returns (address)

{

doSomething();

}

No::

function thisFunctionNameIsReallyLong(address x, address y, address z)

public

onlyowner

priced

returns (address) {

doSomething();

}

function thisFunctionNameIsReallyLong(address x, address y, address z)

public onlyowner priced returns (address)

{

doSomething();

}

function thisFunctionNameIsReallyLong(address x, address y, address z)

public

onlyowner

priced

returns (address) {

doSomething();

}

Multiline output parameters and return statements should follow the same style recommended for wrapping long lines found in the :ref:`Maximum Line Length <maximum\_line\_length>` section.

Yes::

function thisFunctionNameIsReallyLong(

address a,

address b,

address c

)

public

returns (

address someAddressName,

uint256 LongArgument,

uint256 Argument

)

{

doSomething()

return (

veryLongReturnArg1,

veryLongReturnArg2,

veryLongReturnArg3

);

}

No::

function thisFunctionNameIsReallyLong(

address a,

address b,

address c

)

public

returns (address someAddressName,

uint256 LongArgument,

uint256 Argument)

{

doSomething()

return (veryLongReturnArg1,

veryLongReturnArg1,

veryLongReturnArg1);

}

For constructor functions on inherited contracts whose bases require arguments,

it is recommended to drop the base constructors onto new lines in the same

manner as modifiers if the function declaration is long or hard to read.

Yes::

contract A is B, C, D {

function A(uint param1, uint param2, uint param3, uint param4, uint param5)

B(param1)

C(param2, param3)

D(param4)

public

{

// do something with param5

}

}

No::

contract A is B, C, D {

function A(uint param1, uint param2, uint param3, uint param4, uint param5)

B(param1)

C(param2, param3)

D(param4)

public

{

// do something with param5

}

}

contract A is B, C, D {

function A(uint param1, uint param2, uint param3, uint param4, uint param5)

B(param1)

C(param2, param3)

D(param4)

public {

// do something with param5

}

}

When declaring short functions with a single statement, it is permissible to do it on a single line.

Permissible::

function shortFunction() public { doSomething(); }

These guidelines for function declarations are intended to improve readability.

Authors should use their best judgement as this guide does not try to cover all

possible permutations for function declarations.

**Mappings**

TODO

**Variable Declarations**

Declarations of array variables should not have a space between the type and

the brackets.

Yes::

uint[] x;

No::

uint [] x;

**Other Recommendations**

* Strings should be quoted with double-quotes instead of single-quotes.

Yes::

str = "foo";

str = "Hamlet says, 'To be or not to be...'";

No::

str = 'bar';

str = '"Be yourself; everyone else is already taken." -Oscar Wilde';

* Surround operators with a single space on either side.

Yes::

x = 3;

x = 100 / 10;

x += 3 + 4;

x |= y && z;

No::

x=3;

x = 100/10;

x += 3+4;

x |= y&&z;

* Operators with a higher priority than others can exclude surrounding

whitespace in order to denote precedence. This is meant to allow for

improved readability for complex statement. You should always use the same

amount of whitespace on either side of an operator:

Yes::

x = 2\*\*3 + 5;

x = 2\*y + 3\*z;

x = (a+b) \* (a-b);

No::

x = 2\*\* 3 + 5;

x = y+z;

x +=1;

**Naming Conventions**

Naming conventions are powerful when adopted and used broadly. The use of

different conventions can convey significant \*meta\* information that would

otherwise not be immediately available.

The naming recommendations given here are intended to improve the readability,

and thus they are not rules, but rather guidelines to try and help convey the

most information through the names of things.

Lastly, consistency within a codebase should always supercede any conventions

outlined in this document.

**Naming Styles**

To avoid confusion, the following names will be used to refer to different

naming styles.

* b (single lowercase letter)
* B (single uppercase letter)
* lowercase
* lower\_case\_with\_underscores
* UPPERCASE
* UPPER\_CASE\_WITH\_UNDERSCORES
* CapitalizedWords (or CapWords)
* mixedCase (differs from CapitalizedWords by initial lowercase character!)
* Capitalized\_Words\_With\_Underscores

**note**:: When using initialisms in CapWords, capitalize all the letters of the initialisms. Thus HTTPServerError is better than HttpServerError. When using initialisms is mixedCase, capitalize all the letters of the initialisms, except keep the first one lower case if it is the beginning of the name. Thus xmlHTTPRequest is better than XMLHTTPRequest.

**Names to Avoid**

* l - Lowercase letter el
* O- Uppercase letter oh
* I - Uppercase letter eye

Never use any of these for single letter variable names. They are often

indistinguishable from the numerals one and zero.

**Contract and Library Names**

Contracts and libraries should be named using the CapWords style. Examples: SimpleToken, SmartBank, CertificateHashRepository, Player.

**Struct Names**

Structs should be named using the CapWords style. Examples: MyCoin, Position, PositionXY.

**Event Names**

Events should be named using the CapWords style. Examples: Deposit, Transfer, Approval, BeforeTransfer, AfterTransfer.

**Function Names**

Functions other than constructors should use mixedCase. Examples: getBalancetransfer,verifyOwneraddMember,changeOwner.

**Function Argument Names**

Function arguments should use mixedCase. Examples: initialSupply, account, recipientAddress, senderAddress, newOwner.

When writing library functions that operate on a custom struct, the struct

should be the first argument and should always be named ``self``.

**Local and State Variable Names**

Use mixedCase. Examples: totalSupply, remainingSupply, balancesOf, creatorAddress, isPreSale, tokenExchangeRate.

**Constants**

Constants should be named with all capital letters with underscores separating

words. Examples: MAX\_BLOCKS, TOKEN\_NAME, TOKEN\_TICKER, CONTRACT\_VERSION.

**Modifier Names**

Use mixedCase. Examples: onlyBy, onlyAfter, onlyDuringThePreSale.

**Enums**

Enums, in the style of simple type declarations, should be named using the CapWords style. Examples: TokenGroup, Frame, HashStyle, CharacterLocation.

**Avoiding Naming Collisions**

* single\_trailing\_underscore\_

This convention is suggested when the desired name collides with that of a

built-in or otherwise reserved name.

**General Recommendations**

TODO