Table of Contents

Melon Reporting Thesis

Thesis by Simon Emanuel Schmid and Benjamin Zumbrunn

Spring Semester 2018

University of Applied Sciences Northwestern Switzerland FHNW, School of Engineering

Coaches: Prof. Dr. Sarah Hauser, Markus Knecht

Abstract

Introduction

Organisation

The main entry point for this project is the open source Github repository github.com/melonproject/reporting-thesis.

The documentation can be found (and commented) on Gitbooks: schmidsi.gitbooks.io/melon-reporting.

Project management is done with a simple Kanban board as a Github Project: github.com/melonproject/reporting-thesis/projects/1

Single tasks are managed as Github Issues: github.com/melonproject/reporting-thesis/issues

/

Strategy

Vision

Creating functionality on top of the Melon protocol that automates reporting/auditing almost completely:

> a) Something that a real fund manager would be able to confidently say: “This solves my reporting issues and makes my life a lot easier”
>
> b) Something that can be show-cased to FINMA (and other regulators) and show them how: "This will make *their* life over-seeing a lot easier”

Hypothesis

It is possible to extract and visualize all relevant data from the Melon protocol on the Ethereum blockchain in a way that could be legally acceptable by regulators. Furthermore, this data can be audited and digitally signed and a track record of these audits can be placed on the blockchain again.

Boundaries

Legal

This is a technical thesis and therefore we do not deeply research into the legal aspects of fund management and reporting. But we will find ways how technology can support the legal processes.

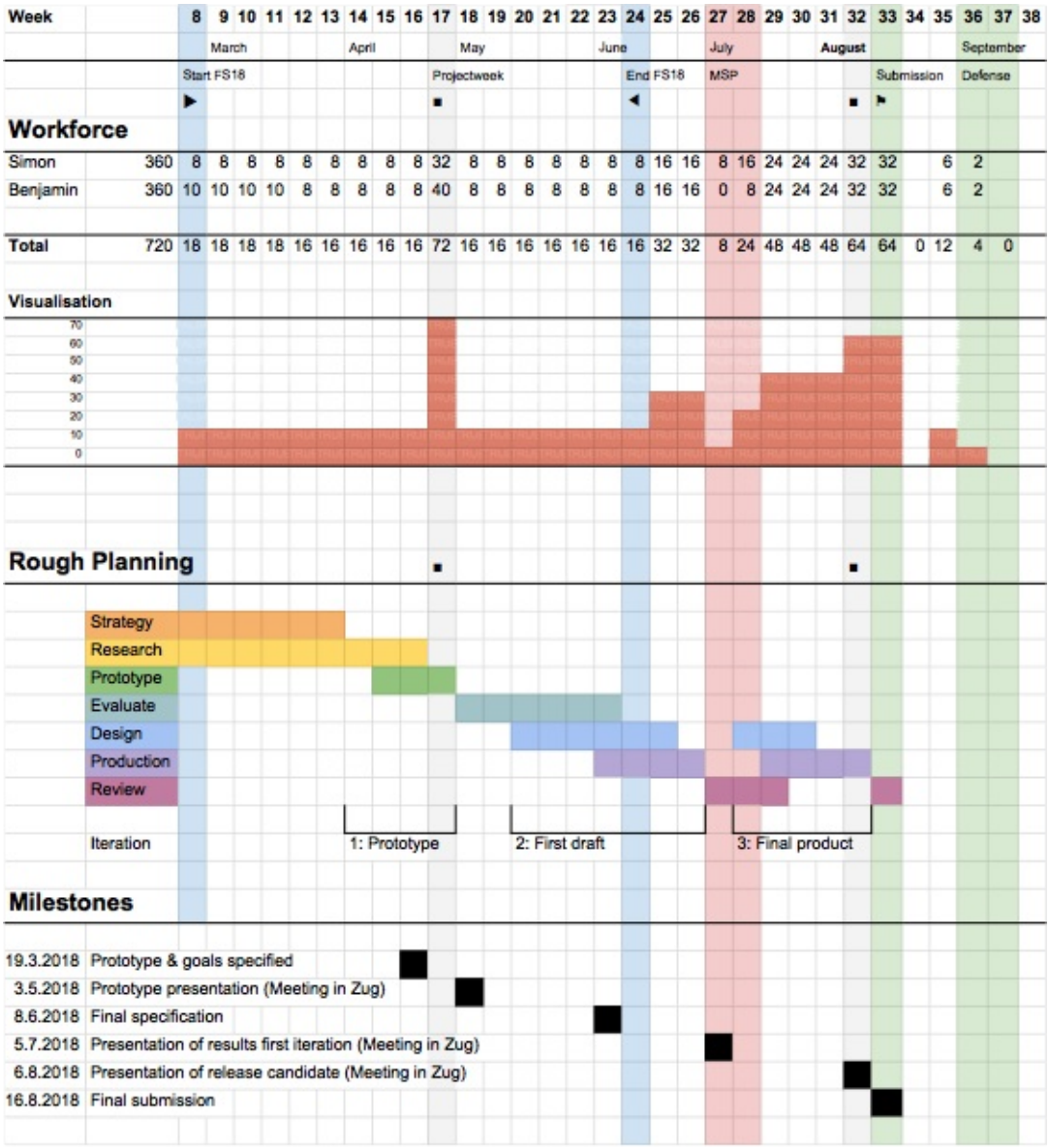
Technical

Risks

Name	Counter measures	Risk (*)
Distractions from classes	Clear timeboxing. Clear planning. Buffers.	2 * 1 = 2
Distractions from job	Clear timeboxing. Clear planning. Clear and upfront communication of availabilities. Support from management.	1 * 1 = 1
Project team out of sync	Weekly team work time slots. Open communication. Weekly reports.	2 * 2 = 4
Dependency on systems out of project scope	Apply subsystem decomposition and isolation techniques from the beginning.	3 * 2 = 6
Losing focus / distraction by details	Weekly reports to coaches and project sponsors to gather feedback.	3 * 2 = 4

(*) Probability of occurrence (1-3) * severity (1-3) = risk

Planning



Strategy

In a first step the foundation of the project is laid. Vision, hypothesis and boundaries are defined and agreed upon with the stakeholders and a rough planning is created. Furthermore repositories are created, docs initialized, etc.

Deadline: 30.3.2018

Expected results:

- Project vision
- Hypothesis
- Project boundaries
- Rough planning
- Basic setup of environment

Research

The next step is to deep dive into the problem domain. We read and summarize the important articles about the topic and learn the underlying technologies. Also part of research are interviews.

Deadline: 20.4.2018

Expected results:

- Overview & summary of material: Articles, law, templates, ...
- Knowledge of underlying technologies (Blockchain, Solidity, React, Redux, digital signing, etc.)
- Transcripts of interviews

Prototype

The collected knowledge from the research phase is now transformed into a first prototype which can be challenged by the stakeholders and test users.

Deadline: 27.4.2018

Expected results:

- Prototypes
- Wireframes

Evaluate

Testing & discussing the prototype with the stakeholders and test users give us valuable insights for the further development.

Deadline: 22.6.2018

Expected results:

- Evaluation reports
- User testing reports

Design

Already during the evaluation phase we start the design phase to have an iterative process: The prototype and wireframes are adjusted from the feedback but also the work on the final mockups and software architecture is started.

Deadline 1: 22.6.2018 **Deadline 2**: 27.7.2018

Expected results:

- Mockups
- Software Architecture
- Specifications

Production

Iterative development and finishing a release candidate. This will also take place in combination with design and review phase. Basically in the following loop: Design -> Production -> Review -> Design -> ...

Deadline 1: 29.6.2018 **Deadline 2**: 10.8.2018

Expected results:

- Final product first and second version

Review

Collect feedback of the release candidate, finish documentation and submission of the thesis.

Deadline: 17.8.2018

Expected results:

- Review reports
- Final thesis report

Milestones

19.3.2018 - Prototype & goals specified

- *Via email*
- Specified what's in the prototype and what not
- General project goals specified with Melonport

3.5.2018 - Prototype presentation (Meeting in Zug)

- Presentation of the finished prototype in Zug
- All stakeholders invited

8.6.2018 - Final specification

- *Via email*
- The specification for the final product is finished and agreed upon

5.7.2018 - Presentation of results second iteration (Meeting in Zug)

- Presentation of the results of second iteration in Zug
- All stakeholders invited

6.8.2018 - Presentation of release candidate (Meeting in Zug)

- Presentation of the release candidate in Zug
- Collection of last feedback and adjustments for final release
- All stakeholders invited

16.8.2018 - Final submission

- Official submission of thesis as bound paper
- Final version deployed

Iterations

Iteration 1: Prototype

The first iteration is a functional prototype. The goal is to have something clickable as soon as possible. For this phase, it is not yet important to have all fields and data. But something that can be shown to fund managers to collect first valuable feedback.

Deadline: 27.4.2018

Iteration 2: First draft

The second iteration aims already at the final goal & specification knowing that it is not possible to cover all topics yet. Still, it should be as functional as possbile to gather more detailed user feedback already.

Deadline: 29.6.2018

Iteration 3: Final product

Finally, all feedback is collected and the final product can emerge from the first draft.

Deadline: 10.8.2018

Summary of Meetings

- 27.3.2018 - 3pm - In Brugg with coaches: 5.2B31
- 17.4.2018 - 3pm - In Brugg with coaches: 5.2B31
- 3.5.2018 - 2pm - In Zug with all stakeholders
- 22.5.2018 - 4pm - In Brugg with Markus Knecht: 5.2B31
- 12.6.2018 - 3pm - In Brugg with all coaches: 5.2B31
- 5.7.2018 - 2pm - In Zug with all stakeholders
- 17.7.2018 - 3pm - In Brugg with coaches: 5.2B31
- 6.8.2018 - 2pm - In Zug with all stakeholders
- 13.8.2018 - 3pm - In Brugg with coaches: 5.2B34

Journal

We send a short status update every week to our stakeholders. They are also stored here for reference.

Calendar Week 8

We started working on the thesis and made a first broad overview over the topics:

- Benjamin started to research into blockchain development particularly Solidity.
- Simon setup the repository and got in touch with possible project supporters from PwC and read a bit into the domain of legal reporting of collective investment schemes.

Calendar Week 9

In the second week we already deep dived into the specific domains:

- Benjamin set up the Solidity development environment according to the Melon setup with dapp.tools, parity dev chain but also looked into truffle suite.
- Simon researched MiFID II and PRIIP and started the glossary for these confusing abbreviations. Furthermore, he finished the strategy part.

Calendar Week 10

This week we had the official kick-off meeting with all stakeholders. See minutes in Appendix.

- Simon updated the project plan according to the feedback and discussed the KIID template provided by PwC with Mona
- Benjamin further read into Solidity, especially data structures like strings and byte64.

Research

http://www.fundinfo.com/en/home/

https://www2.deloitte.com/content/dam/Deloitte/lu/Documents/financial-services/IM/lu_priips-key-investor-document.pdf

Example #1 FTIF - Templeton Global Total Return Fund - A

- http://www.fundinfo.com/en/isin/LU0170475312/
- KIID

EFAMA

- European PRIIPs Template (EPT)
- “Comfort” EPT (CEPT)

Prototype

Evaluate

Design

Interchangeable Fund Data Format

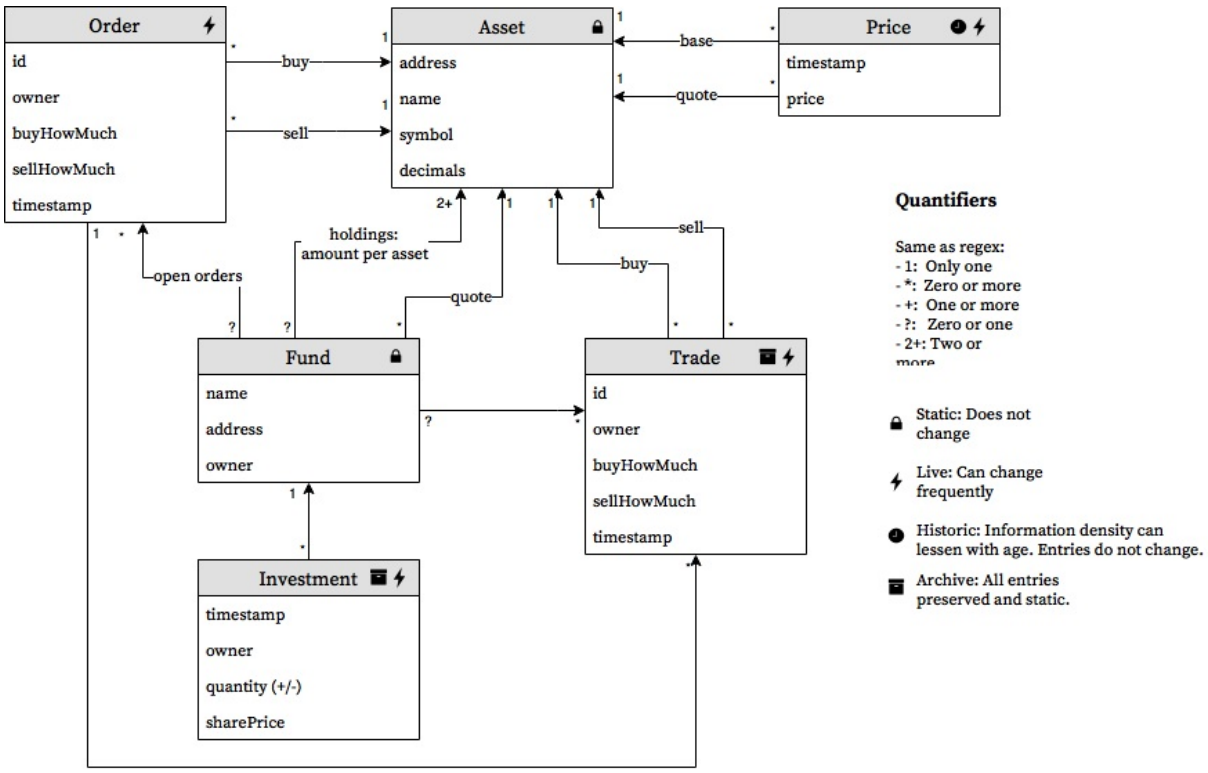
```
{
  "name": "Example Fund",
  "inception": "yyyy-mm-dd hh:mm:ss",
  "description": "This fund is high risk",
  "manager": "0xbad...a55",
  "nav": 1000,
  "quoteSymbol": "MLN",
  "gav": 1100,
  "timestamp": "yyyy-mm-dd hh:mm:ss",
  "holdings": [
    {
      "symbol": "ETH",
      "amount": 1000
    }
  ],
  "trades": [
    {
      "buySymbol": "ETH",
      "sellSymbol": "MLN",
      "buyAmount": 100,
      "sellAmount": 50,
      "timestamp": "yyyy-mm-dd hh:mm:ss",
      "market": "0xdead...beef"
    }
  ],
  "audits": [
    {
      "timestamp": "yyyy-mm-dd hh:mm:ss",
      "auditor": "0xdead...beef1",
      "dataHash": "QmXZcdco6wZEA2paGeUnoshSB4HJiSTDxagqXerDGop6or",
      "signature": "0x23rasdfasdlfjhasldkfhas"
    }
  ]
}
```

Linked issues:

- https://github.com/melonproject/reporting-thesis/issues/7

Minimalistic Data ERM

The following ERM is a representation of the minimalistic data requirements of a fund to build charts and reports. All data can be derived from this model.



Explanation

- The fund is in the center. Once setup a fund cannot change its name/address/owner
- Every fund has one quote asset in which its value is denominated. Usually MLN.
- Total number of shares can be retrived with the following formula: $a \neq 0$

Production

Software Architecture

https://lernajs.io

Meetings

Kick-off meeting 2018-03-07

Present:

- Simon Emanuel Schmid
- Benjamin Zumbrunn
- Mona El Isa (Melonport)
- Reto Trinkler (Melonport)
- Sarah Hauser (FHNW)
- Markus Knecht (FHNW)

FHNW

- Ms. Hauser is away for the whole month of may
- Always send protocols from meetings to both coaches
- Always send Decisions about the project (with melonport) to both coaches

Thesis release

- A poster and an interactive demo is required
- Two printed versions of the thesis must be provided for the FHNW
- Send in a draft of the thesis as soon as possible for review
- Plan three Iterations of work/evaluate
- Define milestones and give more detail on planned work

Project

We might be able to assemble the benchmark data from coinmarketcap or similar platforms.

Our Gitbook solution for project reporting is agreed by the coaches.

Possible verification of the generated reports:

- History of all concerning trades
- Hyperlinks which point directly to the blockchain on IPFS (see IPFS Mesh)

Dates

- We will send a doodle with tuesday-dates (3pm-6pm) for coach reviews at Campus Brugg/Windisch

Progress Meetings in Zug

We already fixed the following dates for progress meetings with all stakeholders. They will happen in the Melonport Office in Zug.

- May 3, 14:00: Presentation of the prototype
- July 5, 14:00: Discuss Design & Architecture before final sprint
- August 6, 14:00: Last meeting before submission

This is a summary of Solidity In Depth.

Layout of source files

Versions

```
pragma solidity ^0.4.0;
```

`^` means "only works with compilers 0.4.x"

Import

```
import "filename"; // import from 'global' or same directory
import * as symbolName from "filename";
import "filename" as symbolName; // same as above
import {symbol1 as alias, symbol2} from "filename";

import "./x" as x; // strictly import from same directory
```

Path prefix remapping

If we clone github.com/ethereum/dapp-bin/ locally to /usr/local/dapp-bin, we can use:

```
import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol" as it_mapping;
```

And run the compiler with:

```
solc github.com/ethereum/dapp-bin/=/usr/local/dapp-bin/ source.sol
```

Comments

```
// single line

/*
    Multiline
*/
```

Natspec comments

```
/// single line
/** multiline */
```

Example:

```
/** @title Shape calculator. */
contract shapeCalculator {
```

```
    /** @dev Calculates a rectangle's surface and perimeter.
    * @param w Width of the rectangle.
    * @param h Height of the rectangle.
    * @return s The calculated surface.
    * @return p The calculated perimeter.
    */
    function rectangle(uint w, uint h) returns (uint s, uint p) {
        s = w * h;
        p = 2 * (w + h);
    }
}
```

Structure of a contract

State Variables

Permanently stored in contract storage.

```
unit storedData;
```

Functions

```
function bid() {...}
```

Function modifiers

Amend the semantics of a function, mostly used for require.

Declaration:

```
modifier onlySeller() { ... }
```

Usage:

```
function abort() onlySeller { ... }
```

Events

Interfaces for EVM logging

Declaration:

```
event HighestBidIncreased(address bidder, uint amount);
```

Trigger:

```
emit HighestBidIncreased(msg.sender, msg.value);
```

Struct Types

Group several variables

```
struct Voter {
    uint weight;
    bool voted;
    address delegate;
    uint vote;
}
```

Enum Types

```
enum State { Created, Locked, Inactive }
```

Types

Value Types

Booleans

```
bool t = true;
bool f = false;
```

Operators: !, &&, ||, ==, !=

Integers

Aliases for int256 and uint256:

```
int i = -1;
uint j = 1;
```

Declare size from `(u)int8` to `(u)int256`, e.g.:

```
int24 max = 8388608;
```

Comparisons: <=, <, ==, !=, >=, > Bit operators: &, |, ^, ~ Arithmetic: +, -, , /, %, *, <<, >>

Fixed point numbers

They are not fully supported in Solidity yet, they can only be declared.

Address

20 byte value. Operators: <=, <, ==, !=, >=, >

Members:

Query the **balance** of an address:

```
uint b = a.balance;
```

Send ether in units of wei to an address:

```
a.transfer(10); // transfers 10 wei to address a
```

If the execution fails, the contract will stop with an exception If a is a contract address, its code will be executed with the transfer call.

Low level counterpart of transfer (returns 'false' on fail):

```
bool success = a.send(10);
```

NOTE: send is dangerous, use transfer or the withdraw pattern.

call, callcode, delegatecall: interface with contracts that do not adhere with the ABI.
NOTE: Only use as last resort, they break the type-safety of Solidity. Arguments of call are padded to bytes32 type. Returns bool that indicates if the function terminated (true) or threw an exception (false).

```
address nameReg = 0x72ba7d8e73fe8eb666ea66babc8116a41bfb10e2;
nameReg.call("register", "MyName");
nameReg.call(bytes4(keccak256("fun(uint256)")), a); // function signature
```

Adjust the supplied gas with .gas():

```
namReg.call.gas(1000000)("register", "MyName");
```

Control the supplied ether value:

```
nameReg.call.value(1 ether)("register", "MyName");
```

Combine both:

```
nameReg.call.gas(1000000).value(1 ether)("register", "MyName");
```

Query the **balance of the current contract**:

```
this.balance;
```

Fixed-size byte arrays

`bytes1`, `bytes2`, `bytes3`, up to `bytes32`
`bytes` is an alias for `bytes1`.

Comparisons and bit operators can be used like on ints.

Index access:

```
bytes2 b2 = "hi";
byte b1 = b2[0]; // access first byte, read only!
```

Length:

```
b1.length; // returns the fixed length of the byte array
```

`bytes` and `string` are not value types!

Rational and integer literals

Decimal fraction literals:

```
1.
.1
1.3
```

Scientific notation is supported:

```
2e10
-2e10
2e-10
2.5e1
```

Division on integers converts to a rational number, but it cannot be stored in any way yet.

```
int i = 5 / 2; // this throws a compiler error
int j = 6 / 2; // this works
```

String literals

```
"use double quotes"
'or single quotes'
```

They are implicitly convertible to `bytes1` ... `bytes32` **if they fit**.

Escape characters are possible:

```
\n // and the like
\xNN // hex
\uNNNN // unicode
```

Hexadecimal literals

```
hex"001122FF"
```

The content must be a hexadecimal string. The value will be the binary representation.. They behave like string literals.

Enums

```
enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
ActionChoices choice = ActionChoices.GoStraight;
```

Function types

internal functions
Can only be called inside the current contract. This is the *default* for functions.

Use internal functions in **libraries**:

```
library SomeLibrary {
    // declare internal functions
}

contract SomeContract {
    using SomeLibrary for *;
    // use internal functions
}
```

external functions
Consist of an address and a function signature.

Use external functions between **contracts**:

```
contract Oracle {
    // define external functions
}

contract OracleUser {
    Oracle constant oracle = Oracle(0x1234567); // known contract
    oracle.query(...);
}
```

Notation:

```
function (<parameter types>) {internal|external} [pure|constant|view|payable] [returns (<return types>)]
```

Access the function type:

```
f // call by name -> internal function
this.f // -> external function
```

Return the ABI function selector:

```
this.f.selector; // type: bytes4
```

Reference Types

Reference types have to be handled more carefully, because storing them in **storage** is expensive.

Data location

Complex types (arrays & structs) have a data location (`storage` or `memory`).

- Default for function parameters is `memory`
- Default for local variables is `storage`
- `storage` is forced for state variables

There is a third location `calldata` where function arguments are stored.

Arrays

Can have a fixed size or can be dynamic.

Array of 5 dynamic arrays of uint:

```
uint[][5] a;
uint a32 = a[2][3]; // access second uint in third array
```

NOTE: notation is reversed!!!

USEFUL - convert from string to bytes:

```
string memory s = "hello world";
b = bytes(s);
```

Create a **getter** for arrays automatically by marking them `public`:

```
int32 public intArray;
```

But values can only be obtained with a numeric index.

Allocating Memory Arrays

Use `new` for arrays with variable length in *memory*.

```
uint[] memory a = new uint[](7);
bytes memory b = new bytes(7);
```

Array Literals / Inline Arrays

Arrays written as an expression:

```
[uint(1), 2, 3]; // evaluates to a 'uint8[3] memory' array
```

The cast on the first element is necessary to define the type. They cannot be assigned to dynamic arrays at the moment.

Members

- length
  Dynamic arrays can be resized in *storage* with `.length`.
  The size of *memory* arrays is fixed once they are created.
- push
  Use `push` to append an element on dynamic storage arrays and `bytes`.

NOTE:
It is not possible to return dynamic content from external function calls. The only workaround now is to use large statically-sized arrays.

Structs

```
struct Funder {
    address addr;
    uint amount;
}

Funder f1 = Funder({addr: msg.sender, amount: msg.value}); // create with names
Funder f2 = Funder(msg.sender, msg.value); // simple create
f1.amount += msg.value; // access
```

Mappings

Formal definition:

```
mapping(_KeyType => _ValueType)
```

- Keytype can be almost anything except for a mapping, dynamically sized array, contract, enum or struct.
- ValueType can be anything, even another mapping.

The key data is not actually stored in a mapping, only its `keccak256` hash. Mappings do not have a length. Mappings are only allowed for **state variables** (or storage reference types in internal functions).

LValue Operators

```
a += e;
a -= e;
a *= e;
a /= e;
a %= e;
a |= e;
a &= e;
a ^= e;
a++;
a--;
++a;
--a;
```

delete

`delete` assigns the initial value for the type of a:

```
int i = 42;
delete i; // i is now 0
```

Delete on dynamic arrays assigns an array of size 0.
Delete on static arrays resets all values.
Delete on structs resets all values.

Conversions

Implicit conversions

Possible when no information is lost:

- `uint8` to `uint17` is possible
- `int128` to `int256` is possible
- `int8` to `uint256` is NOT possible
- uints can be converted to bytes of the same size or larger
- `uint160` can be converted to `address`

Explicit conversions

Use with care!

```
int8 y = -3;
uint x = uint(y);
```

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b will be 0x5678 now -> information loss
```

Type Deduction (var)

It is not necessary everytime to assign a type.

```
uint24 x = 0x123;
var y = x; // y has type uint24 automatically here
```

Units and global variables

Ether units

Possible ways to work with ether units:

- Literal number with suffix (`wei` , `finney` , `szabo` , `ether`)
- Literal number without suffix is always `wei`

Calculation works as expected.

```
2 ether == 2000 finney // evaluates to true
```

Time units

Calculation works as expected.
The base unit is *seconds*.

```
1 == 1 seconds
1 minutes == 60 seconds
1 hours == 60 minutes
1 days == 24 hours
1 weeks == 7 days
1 years == 365 days
```

The suffixes cannot be applied to variables. Do it like this:

```
uint daysAfter = 42;
if (now >= daysAfter * 1 days) {...};
```

Special variables and functions

Block and transaction properties

- `block.blockhash(uint blockNumber)` returns (`bytes32`): hash of the given block - only works for 256 most recent blocks excluding current
- `block.coinbase` (`address`): current block miner's address
- `block.difficulty` (`uint`): current block difficulty
- `block.gaslimit` (`uint`): current block gaslimit
- `block.number` (`uint`): current block number
- `block.timestamp` (`uint`): current block timestamp as seconds since unix epoch
- `gasleft()` returns (`uint256`): remaining gas
- `msg.data` (`bytes`): complete calldata
- `msg.sender` (`address`): sender of the message (current call)
- `msg.sig` (`bytes4`): first four bytes of the calldata (i.e. function identifier)
- `msg.value` (`uint`): number of wei sent with the message
- `now` (`uint`): current block timestamp (alias for block.timestamp)
- `tx.gasprice` (`uint`): gas price of the transaction
- `tx.origin` (`address`): sender of the transaction (full call chain)

Error handling

```
assert(bool condition) // for internal errors
require(bool condition) // for errors in inputs or external components
revert() // abort execution, revert state changes
```

Mathematical and cryptographic functions

```
addmod(uint x, uint y, uint k) returns (uint) // compute (x + y) % k
mulmod(uint x, uint y, uint k) returns (uint): // compute (x * y) % k
keccak256(...) returns (bytes32) // compute Ethereum-SHA-3 hash of args
sha256(...) returns (bytes32) // compute SHA-256 hash of args
sha3(...) returns (bytes32) // alias to keccak256()
ripemd160(...) returns (bytes20) // compute RIPEMD-160 hash of args
ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address) // elliptic curve signature
```

The arguments are packed without padding.
ecrecover example

Address related

```
<address>.balance (uint256) // balance of the address in wei
<address>.transfer(uint256 amount) // send amount of wei to address, throws on failure
<address>.send(uint256 amount) returns (bool) // send amount of wei to address, returns false on failure
```

Contract related

```
this // the current contract, convertible to address
```

```
selfdestruct(address recipient) // destroy current contract, send funds to address
```

Expressions and control structures

Input parameters

```
function taker(uint _a, uint _b) public pure {
    // do something with _a and _b.
}
```

Output parameters

Returning multiple values is possible.

```
function arithmetics(uint _a, uint _b)
    public
    pure
    returns (uint a, uint b)
{
    a = 1;
    b = 2;
}
```

Return parameters are initialized to zero.

Control structures

They can be used the same as in C or Javascript:

- `if`
- `else`
- `while`
- `do`
- `for`
- `break`
- `continue`
- `return`
- `? :`

There is no type conversion from non-boolean to boolean (`1` is not `true` !).

Function calls

Internal function calls

Internal functions (from the same contract) can be used recursively.

External function calls

```
this.g(8);
c.g(8) // where c is a contract instance
```

Amount of wei and gas can be specified when calling functions from other contracts:

```
contract InfoFeed {
    function info() public payable returns (uint ret) { return 42; }
}

contract Consumer {
    InfoFeed feed; // contract instance
    function callFeed() public { feed.info.value(10).gas(800)(); }
}
```

`payable` must be used for `info()` to have the `.value()` option.

WARNING:
Called contracts can change state from our own contracts. Write functions in a way that calls to external functions happen after any changes to state variables in our contract so our contract is not vulnerable to a *reentrancy exploit*.

Named calls

Enclose args in `{}`, then the order doesn't matter:

```
f({value: 2, key: 3});
```

Creating contracts with new & constructors

Contracts can create other contracts with `new`:

```
contract D {
    uint x;
    function D(uint a) public payable { // ctor
        x = a;
    }
}

contract C {
    D d = new D(4); // will be executed as part of C's constructor

    function createD(uint arg) public {
        D newD = new D(arg);
    }
}
```

Assignment

Tuple syntax is possible:

```
function f() public pure returns (uint, bool, uint) {
    return (7, true, 2);
}

var (x, b, y) = f(); // specifying types is not possible here
(x, y) = (2, 7); // assign to predefined variables
(x, y) = (y, x); // swap
```

```
(data.length,) = f(); // rest of the values can be ignored (returns 2 values but we only care about first)
(,data[3]) = f(); // ignore beginning values
(x,) = (1,); // one component tuple
```

Scoping and declarations

For version 0.4.x, a variable declared anywhere in a function is available everywhere in the function (like Javascript). In version 0.5.x, this will change.

Error handling (Assert, Require, Revert, Exceptions)

Solidity uses state-reverting exceptions.

- Use `assert` to check invariants
- Use `require` to ensure input values
- Use `revert` to throw an exception, revert the current call (and done subcalls)

Catching exceptions is not yet possible.

`assert` will use all gas, `require` uses none.

Contracts

Creating Contracts

With web3js: web3.eth.Contract

Only one constructor is allowed --> ctor overloading is not possible. Cyclic dependencies between contracts are not possible.

Visibility and getters

There are two kinds of function calls in Solidity, so there are four types of **visibilities for functions**.

external

Called via other contracts and transactions.

To call it from inside a contract, we have to use `this.f()`.

public (default)

Call internally or via messages. For public state variables, a getter is generated automatically.

internal

Functions and state variables can only be accessed from within the *contract (or deriving contracts)*.

private

Functions and state variables can only be accessed from within the *contract*.

NOTE: private stuff is still visible for everyone, just not accessible!

Getter functions

The compiler automatically creates getter-functions for public state variables.

This:

```
contract Complex {
    struct Data {
        uint a;
        bytes3 b;
        mapping (uint => uint) map;
    }
    mapping (uint => mapping(bool => Data[])) public data;
}
```

will generate the following function:

```
function data(uint arg1, bool arg2, uint arg3) public returns (uint a, bytes3 b) {
    a = data[arg1][arg2][arg3].a;
    b = data[arg1][arg2][arg3].b;
}
```

Function modifiers

Modifiers can change the behaviour of functions. The function body is inserted where the special symbol **_** appears.

```
modifier onlyOwner {
    require(msg.sender == owner);
    _;
}

function changePrice(uint _price) public onlyOwner {
    price = _price;
}
```

Multiple modifiers can be used in a whitespace-separated list. All symbols visible in the function are visible for the modifier.

Constant state variables

State variables can be declared as `constant`. Then they have to be assigned from an expression which is a constant at compile time.

These functions are allowed:

- `keccak256`
- `sha256`
- `ripemd160`
- `ecrecover`
- `addmod`
- `mulmod`

The only supported types valid for now are **value types** and **strings**.

Funtions

View functions

Promise **not to modify state**. Can be declared with `view`.

These things are considered to modify state:

- Writing to state variables
- Emitting events
- Creating other contracts
- Using `selfdestruct`
- Sending ether
- Calling functions not marked `view` or `pure`
- Using low-level calls
- Using inline assembly with opcodes

NOTE: getter methods are marked `view`.

Pure functions

Functions that do **not read from or modify the state**.

These things are considered to read from state:

- Reading from state variables
- Accessing `this.balance` or `<address>.balance`
- Accessing `block`, `tx` or `msg`
- Calling functions not marked with `pure`
- Inline assembly with opcodes

Fallback function

The unnamed function. This is called when no other function matches the function identifier.

Sending ether to this contract will cause an exception (no other functions are defined):

```
uint x;
function() public { x = 1; }
```

If ether is sent to this contract, there is no way to get it back:

```
function() public payable { }
```

Function overloading

Function overloading is possible (but not for ctors).

```
contract A {
    function f(uint _in) public pure returns (uint out) {
        out = 1;
    }
```

```
    function f(uint _in, bytes32 _key) public pure returns (uint out) {
        out = 2;
    }
}
```

If there is not exactly one candidate for the function, resolution fails. For example: `f(unit8)` and `f(uint256)` fails when `f` is called with a `uint8` value or below.

Events

The "logging" mechanism of ethereum.

SPV proofs are possible: If an external entity supplies a contract with an SPV proof, it can check that the log actually exists in the blockchain (but block headers have to be supplied).

Up to three arguments can receive the attribute `indexed`. We can then search for these arguments. Indexed arguments will not be stored themselves, we can only search for these values. If arrays are used as indexed arguments, their `keccak256` hash will be stored.

With `anonymous`, the signature of the event is not stored. All non-indexed arguments will be stored in the data part of the log.

Event example:

```
contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );

    function deposit(bytes32 _id) public payable {
        emit Deposit(msg.sender, _id, msg.value); // 'Deposit' is now filterable with JS
    }
}
```

Look for events with *javascript*:

```
var abi = /* abi as generated by the compiler */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at("0x1234...ab67" /* address */);

var event = clientReceipt.Deposit();

// watch for changes
event.watch(function(error, result){
    // result will contain various information
    // including the argumets given to the `Deposit`
    // call.
    if (!error)
        console.log(result);
});

// or callback to start watching immediately
var event = clientReceipt.Deposit(function(error, result) {
    if (!error)
        console.log(result);
});
```

There is also a **low-level interface** for logs.

Inheritance

Solidity supports multiple inheritance. All functions are *virtual* (most derived function is called).

The code from inherited contracts is copied into one contract.

Use `is` to derive from another contract.

If a contract doesn't implement all functions, it can only be used as an interface:

```
function lookup(uint id) public returns (address adr); // 'abstract' function
```

Multiple inheritance is possible:

```
contract named is owned, mortal { ... }
```

Functions can be overridden by another function with the same name and the same number/types of inputs.

If the constructor takes an argument, it must be provided like this:

```
contract PriceFeed is named("GoldFeed") { ... }
```

To specifically access functions from base contracts, use `super`:

```
contract Base is mortal {
    function kill() public { super.kill(); }
}
```

Constructors

Constructor functions can be `public` or `internal`.

```
contract B is A(1) {
    function B() public {}
}
```

An `internal` ctor marks the contract as abstract!

Arguments for base constructors

```
contract Base {
    uint x;
    function Base(uint _x) public { x = _x; }
}

contract Derived is Base(7) {
    function Derived(uint _y) Base(_y * _y) public {
    }
}
```

Abstract contracts

Contracts where at least one function is not implemented.

This is a function declaration:

```
function foo(address) external returns (address);
```

Careful: this is a function type (variable which type is a function):

```
function(address) external returns (address) foo;
```

Interfaces

- Cannot have any functions implemented
- Cannot inherit other contracts or interfaces
- Cannot define variables
- Cannot define structs
- Cannot define enums

Use keyword `interface` :

```
interface Token {
    function transfer(address recipient, uint amount) public;
}
```

Libraries

Libraries are assumed to be stateless. They are deployed only once at a specific address.

Restrictions in comparison to contracts

- No state variables
- Cannot inherit or be inherited
- Cannot receive ether

Example:

```
library Set {
  // type of Data is 'storage reference', so only the storage address, not the content is saved here
  struct Data { mapping(uint => bool) flags; } // will be used in the calling contract!

  function insert(Data storage self, uint value)
      public
      returns (bool)
  {
      if (self.flags[value])
          return false; // already there
      self.flags[value] = true;
      return true;
  }
  // ...

contract C {
    Set.Data knownValues;

    function register(uint value) public {
        // library functions can be called without a specific instance!
        // the 'instance' is the current contract...
        require(Set.insert(knownValues, value));
    }
```

Using for

Attach library functions to any type:

```
using A for B; // where A is the library and B the type
```

With the example from above:

```
contract C {
    using Set for Set.Data; // this is the crucial change
    Set.Data knownValues;

    function register(uint value) public {
        // Here, all variables of type Set.Data have
        // corresponding member functions.
        // The following function call is identical to
        // `Set.insert(knownValues, value)`
        require(knownValues.insert(value));
    }
}
```

We can also extend elementary types:

```
using Search for uint[];
```

CISA

(FINMA)

Collective Investment Schemes Act / Kollektivanlagengesetz, KAG

https://www.admin.ch/opc/en/classified-compilation/20052154/index.html

CISO

Collective Investment Schemes Ordinance / Kollektivanlagenverordnung, KKV

https://www.admin.ch/opc/en/classified-compilation/20062920/index.html

CISO-FINMA

Ordinance of the Swiss Financial Market Supervisory Authority on Collective Investment Schemes / Kollektivanlagenverordnung-FINMA, KKV-FINMA

https://www.admin.ch/opc/en/classified-compilation/20140344/index.html

EFAMA

European Fund and Asset Management Associaton

http://www.efama.org/SitePages/Home.aspx

ERM

Entity relationship model. A technical tool to model and visualize entities and their relations.

ESMA

European Securities and Market Authority

https://www.esma.europa.eu

FCA

Financial Conduct Authority UK https://www.fca.org.uk/

FINMA

Swiss Financial Market Supervisory Authority / Eidgenössische Finanzmarktaufsicht

https://www.finma.ch

https://www.finma.ch/en/authorisation/institutions-and-products-subject-to-the-collective-investment-schemes-act/

ISO 4217

Currency codes, usually 3 characters whereas the first two are indicating the country and the 3rd the currency name. Example: CHF, USD, ...

Special currencies start with an X. Therefore, XBT would be the official code for Bitcoin, this isn't yet standardised.

KIID

Key Investor Information Document. A summarized 1-2 pages document that contains most relevant documentation for retail investors.

MiFID II

EU Markets in Financial Instruments Directive. MiFID I originally from 2004 it's successor MiFID II took effect on January 2018.

- https://en.wikipedia.org/wiki/Markets_in_Financial_Instruments_Directive_2004

PRIIP

Packaged Retail and Insurance-based Investment Products

> A PRIIP is defined as: an investment where, regardless of its legal form, the amount repayable to the retail investor is subject to fluctuations because of exposure to reference values or to the performance of one or more assets that are not directly purchased by the retail investor; or an insurance-based investment product which offers a maturity or surrender value that is wholly or partially exposed, directly or indirectly, to market fluctuations.
>
> The aim of the PRIIPs Regulation is to encourage efficient EU markets by helping investors to better understand and compare the key features, risk, rewards and costs of different PRIIPs, through access to a short and consumer-friendly Key Information Document (KID). How information in the KID should be calculated and presented is set out in the PRIIPs Regulatory Technical Standards (RTSs).

- Source: https://www.fca.org.uk/firms/priips-disclosure-key-information-documents

Prospectus

> A prospectus, in finance, is a disclosure document that describes a financial security for potential buyers.

https://en.wikipedia.org/wiki/Prospectus_(finance)

Solidity

Specialized language to develop smart contracts

SRRI

> The synthetic risk and reward indicator (SRRI) is used to classify investment funds into one of three different risk categories (low risk, medium risk, high risk). It is calculated on the basis of Austrian and European regulatory requirements. This indicator forms an integral part of the Key Investor Information Document (KIID) and gives the historical volatility of the fund unit price on a scale from 1 to 7.

SRRI	Risk category	Volatility intervals
1	Low risk	0% to <0.5%
2		≥0.5% to <2.0%
3	Medium risk	≥2.0% to <5.0%
4		≥5.0% to <10.0%
5		≥10.0% to <15.0%
6		≥15.0% to <25.0%
7	High risk	≥25.0%

http://fundglossary.erste-am.com/srri/

UCITS

Undertakings For The Collective Investment Of Transferable Securities

https://www.investopedia.com/terms/u/ucits.asp
https://en.wikipedia.org/wiki/Undertakings_for_Collective_Investment_in_Transferable_Securities_Directive_2009