# Layout of a Solidity Source File

### Version Pragma

* pragma solidity ^0.4.0;

will not compile with a compiler earlier than version 0.4.0 and it will also not work on a compiler starting from version 0.5.0 (constrained by using ^)

### Importing other Source Files

* import "filename";
* import \* as symbolName from "filename";

import "filename" as symbolName;

creates a new global symbol symbolName whose members are all the global symbols from "filename"

### Comments

* Single-line comments (//) and multi-line comments (/\*...\*/) are possible.
* Support Doxygen style comment

# Structure of a Contract

contract SimpleStorage {

uint storedData; // State variable

function bid() public payable { // Function

// ...

}

modifier onlySeller() { // Modifier

require(msg.sender == seller);

\_;

}

function abort() public onlySeller { // Modifier usage

// ...

}

event HighestBidIncreased(address bidder, uint amount); // Event

function bid() public payable {

// ...

HighestBidIncreased(msg.sender, msg.value); // Triggering event

}

// Stuct types

struct Voter { // Struct

uint weight;

bool voted;

}

enum State { Created, Locked, Inactive } // Enum

}

# Types

statically typed language: the type of each variable (state and local) needs to be specified at compile-time

### Value Types

#### Booleans

* + True or false, supports ! && || == !=

#### Integers

* + int / uint : signed and unsigned
  + uint8 to uint256 in step of 8. uint = uint256
  + <= < == != >= > & | ^ ~ + - \* / % \*\* (exponentiation)
  + << >> :
  + fixed / ufixed : Fixed point numbers are not fully supported by Solidity yet. They can be declared, but cannot be assigned to or from

#### Address

* + address: Holds a 20 byte value (size of an Ethereum address)
  + <= < == != >= >
  + Members of Addresses: Balance transfer send call callcode delegatecall

#### Fixed-size byte arrays

* + bytes1, bytes2, bytes3, …, bytes32. byte is an alias for bytes1.

#### Dynamically-sized byte array

* + bytes string

#### Address Literals

* + Hexadecimal literals that pass the address checksum test are of address type : 0xXXXXX

### Function Types

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

### Reference Types

#### Data location

* + memory storage: stored in memory or in storage
  + Forced data location:
    - parameters (not return) of external functions: calldata
    - state variables: storage
  + Default data location:
    - parameters (also return) of functions: memory
    - all other local variables: storage

#### Structs

#### Mappings

* + mapping(\_KeyType => \_ValueType)
  + not iterable

# Units and Globally Available Variables

# Expressions and Control Structures

### Input Parameters

* input parameters are declared the same way as variables are

### Output Parameters

* output parameters can be declared with the same syntax after the returns keyword

function arithmetics(uint \_a, uint \_b) public pure

returns (uint o\_sum, uint o\_product) {

o\_sum = \_a + \_b;

o\_product = \_a \* \_b;

}

### Control Structures

* Most of the control structures from JavaScript are available in Solidity except for switch and goto

### Function Calls

#### Internal Function Calls

* + Functions of the current contract can be called directly

#### External Function Calls

* + this. and instance. will be called “externally”, via a message call and not directly via jumps
  + When calling functions of other contracts, the amount of Wei sent with the call and the gas can be specified with special options .value() and .gas(), respectively
  + Any interaction with another contract imposes a potential danger, especially if the source code of the contract is not known in advance

### Scoping and Declarations

* A variable declared anywhere within a function will be in scope for the *entire function*, not just the block, regardless of where it is declared

### Error handling: Assert, Require, Revert and Exceptions

* Catching exceptions is not yet possible.

# Contracts

### Creating Contracts

### Visibility and Getters

* Functions can be specified as being external, public, internal or private, where the default is public
* For state variables, external is not possible and the default is internal
* external

External functions are part of the contract interface, which means they can be called from other contracts and via transactions. cannot be called internally (i.e. f() does not work, but this.f() works)

* public

Public functions are part of the contract interface and can be either called internally or via messages

* internal

Those functions and state variables can only be accessed internally (i.e. from within the current contract or contracts deriving from it), without using this

* private

Private functions and state variables are only visible for the contract they are defined in and not in derived

Contracts

* Everything that is inside a contract is visible to all external observers
* Getter Functions
  + The compiler automatically creates getter functions for all **public** state variables
* Constant State Variables
  + State variables can be declared as constant
* View Functions
  + Functions can be declared view in which case they promise not to modify the state.
  + The compiler does not enforce yet that a view method is not modifying state.
* Pure Functions
  + Functions can be declared pure in which case they promise not to read from or modify the state.
  + The compiler does not enforce yet that a pure method is not reading from the state.
* Fallback Function
  + unnamed function, executed on a call to the contract if none of the other functions match the given function identifier (or if no data was supplied at all).
  + in order to receive Ether, the fallback function must be marked payable

### Events

### Inheritance

* All function calls are virtual
* Use is

### Abstract Contracts and Interfaces

* Functions not implemented

### Libraries

* if library functions are called, their code is executed in the context of the calling contract
* it can only access state variables of the calling contract if they are explicitly supplied
* No state variables, Cannot inherit nor be inherited, Cannot receive Ether
* The directive using A for B; can be used to attach library functions (from the library A) to any type (B).

# Security Considerations

### Private Information and Randomness

* Everything you use in a smart contract is publicly visible, even local variables and state variables marked private
* Using random numbers in smart contracts is quite tricky if you do not want miners to be able to cheat.

### Re-Entrancy

* Any interaction from a contract (A) with another contract (B) and any transfer of Ether hands over control to that contract (B). This makes it possible for B to call back into A before this interaction is completed.
* Checks-Effects-Interactions pattern

contract Fund {

mapping(address => uint) shares;

function withdraw() public {

var share = shares[msg.sender];

shares[msg.sender] = 0; // Change the value first

msg.sender.transfer(share);

}

}

### Callstack Depth

* External function calls can fail any time because they exceed the maximum call stack of 1024

### tx.origin

* Never use tx.origin for authorization.

### Recommendations

* Restrict the Amount of Ether
  + Restrict the amount of Ether (or other tokens) that can be stored in a smart contract
* Keep it Small and Modular
* Use the Checks-Effects-Interactions Pattern
  + Most functions will first perform some checks (who called the function, are the arguments in range, did they send enough Ether, does the person have tokens, etc.)
  + if all checks passed, effects to the state variables of the current contract should be made
  + Interaction with other contracts should be the very last step in any function
* Include a Fail-Safe Mode