# Summary

- Solidity Syntax and Semantics

Reading: Solidity Documentation https://docs.soliditylang.org

Part 1:



- Headers, Comments
- Basic Types
- Address Types
- Contract Types & declarations
- Function Types & declarations
- Receive and Fallback functions
- Contract creation and destruction

# Solidity

- A high level smart contract language
- Syntax influenced by C++, Javascript, Python
- Statically typed, object-oriented with inheritance, user defined types
- Compiles to Ethereum Virtual Machine bytecode
- Currently, the main high level language used on Ethereum
- Integrations for various IDEs/editors: Virtual Studio, IntelliJ, Emacs,Vim
- The easiest way to explore it for small programs is the Remix browser based IDE
  https://remix.ethereum.org
- For larger programs, a popular development framework is the Truffle Suite:
  https://www.trufflesuite.com



- Significant changes still being made between language versions (these slides version 0.8)

# Main points of novelty

With respect to standard (object-oriented) programming languages, the main differences are:

- ability to attach value (in Eth) to function calls
- gas cost of computation/gas management
- objects identified by address (hash of public key)
- underlying memory model of Ethereum virtual machine (storage, memory, calldata, stack)
- embedded ``assembly level programming'' in EVM bytecode
- builtin support for cryptography (particularly, hash functions and signatures)

# Header / Includes

The compiler versions assumed by a Solidity program are defined by a header such as

**pragma solidity >= 0.5.2** (constraints use npm - node.js package manager - syntax)



Import statements (similar to Javascript):



**import "filename";**
= import all global symbols from the file



**import * as nameSpace from "filename";** or, equivalently,
**import "filename" as nameSpace**
= symbolName has global symbols from the file as members
**symbol** in **filename** is then referenced as **nameSpace.symbol**

**import (symbol1 as alias, symbol2) from "filename"**
= import only **symbol1**, **symbol2**, rename **symbol1** as **alias**

# Comments

```
// This is a single-line comment.

/*
This is a
multi-line comment.
*/
```

# Types

**bool** (operations: !, &&, ||, ==, != using C syntax, lazy evaluation)

**int8, int16, int24, …., int256, int**
(signed integers of various lengths, steps of 8, int = int256)

**uint8, uint16, …, uint256, uint**
(unsigned integers **uint = uint256**)



(operations: +, - , *, /, % (remainder), ** (exponentiation)
bitwise boolean operations: &, |, ^ (xor), ~ (negation)



From Version 0.8: overflow of arithmetic operations causes an EVM **revert**

For wraparound semantics for arithmetic, use, e.g., **unchecked { x+ y }**

# Variable declarations

Variables are declared with a type in the forms

**type [visibility] variable** (initially, the default value of type, not null)

**type [visibility] variable = initial value**

Examples:



**bool success ;** (initially, false)



**bool public success ;**



**uint internal count = 0 ;**

# Address Type

**address** (20 byte address, i.e. hash of public key)

**address payable** (can **transfer** and **send** to this)

operations: <=, <, = , >=, >



Hexadecimal literals (39-41 digits) passing an address checksum test are of type address:



0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF



A conversion from 20 bytes to upper+lowercase Hex (for extra checksum bits) is described in
https://eips.ethereum.org/EIPS/eip-55

**msg.sender** always has type **address** (from version 0.8)

Can convert x of type **address** to type **address payable** with payable(x)

# Contract Types

Contract declarations create an associated type:

```
Contract MyNewMoney {

// variable declarations

/* function declarations, including:
      constructor
      receive function
      fallback function
*/

}
```



creates a new type MyNewMoney.

Casting a value **x** of contract type to type **address, address payable**:

**address(x)**
**payable(address(x))**

# Function declarations

Functions in a contract are declared using syntax

```
function <functionName> ( <arguments> )
        <visibility>
        [ pure | view | payable ]
        [ returns <resultType> ] {
// code
}
```



- **pure** = neither reads nor writes the state
- **view** = may read but does not write the state
- **payable** = a payment may be attached when calling this function

# Visibility declarations

The following declarations can be applied to contract variables and functions

**external** (on functions): callable from outside the contract via messages and transactions
*cannot* be called internally

**internal** : can only be accessed internally

**public** : can be called internally or via messages
For public variables *varname* an *external* getter function *varname()* is automatically generated



**private** : only visible for the contract in which it is defined and not in derived contracts.



(Some of these differences relate to handling of memory vs storage vs calldata distinctions)

**NOTE:** these refer to access control rules to be enforced by the compiler.
They do NOT refer to visibility of information on the blockchain.
**Everything** on the public blockchain is visible to everybody.

```
Contract Example {

uint public x ;

function add(uint a, uint b) public pure returns (uint) {
      uint y = a+ b ;
      return y ;
      }

function setX(uint a) external {
      x = a ;
      }

function xPlus(uint a) public view returns (uint output) {
      output = x+ a ;
      }

function incX() internal {
      x = x+1 ;
      }

}
```



External interface:

```
Contract Example {
   function x() returns (uint);
   function add(uint,uint) returns (uint)
   function setX(uint) ;
   function xPlus(uint) returns (uint)
}
```

# Function Calls

A contract may call a function on itself or on another contract, identified by its *address.*

Calling function **fname** on the contract with address **x** is denoted by

**x.fname(arguments)**

Calling a function on the calling contract



**fname(arguments)**



**this.fname(arguments)**



**Note:** the *EVM* does not have a notion of function at that level:

- Each contract address has just **one** EVM bytecode program as **contract code**

- Solidity compiles to bytecode that looks for a function identifier
  hash(name(argument types))
  in the early bytes of the **calldata** and jumps to the the appropriate part of the contract code representing that function.

# Call modifiers:

An amount of Ether value can be associated with a function call using
"**{value: amount}**"

Example:

**x.deposit{value:100 wei}("Good service! Here is a tip")**



By default, *all* the gas balance of the caller is made available to the contract called. To restrict this, use the modifier "**gas:amount**"



Example:

**x.deposit.{value:100 wei, gas:40000}("Good service! Here is a tip")**

# Functions always callable on an address

Functions of an **address x**:

**x.balance : uint256** (query the balance of x, in Wei)
**x.code : bytes memory** (code, if any, at x)
**x.codehash : bytes32** (hash of code, if any, at x)

EVM Call operations:

**x.call(bytes memory) : (bool, bytes memory)**
**x.staticcall(bytes memory) : (bool, bytes memory)**
**x.delegatecall(bytes memory) : (bool, bytes memory)**



Functions of a **payable address x**:

**x.transfer(uint256)** (transfer the amount in Wei to address x, with 2300 gas)
**x.send(uint256) : bool** (EVM send operation, with 2300 gas)

# The difference between transfer and send

Both **transfer** and **send** forward 2300 gas to the recipient (not adjustable), so that it can run its receive or fallback function.

This could cause an exception by the recipient (e.g., stack overflow, or out-of-gas). In this case:

**send** returns false

**transfer** aborts (causing caller to abort)



So use transfer only when you want to abort on failure of the transfer, and when using **send**, always check the return result. Don't assume it worked and the money was transferred!

# Receive and Fallback Functions

A contract can have at most one **receive** function, specified in the form

**receive() external payable { //code …. }**

If present, this is invoked by **.transfer** and **.send** operations



A “fallback function” in a contract is a function specified with either



**fallback () external**
**fallback () external payable**



If present, this is invoked by an undefined function call, or receipt of a **send** or **transfer** when there is no receive function.

There may be at most one such function defined.

# Gas limits and Fallback/Receive Functions

**send** and **transfer** are called with only 2300 gas, in which case the receive/fallback function

— should not: write to storage, create a contract, call an external function, send ether
(these all cost more than 2300 gas!)

— can: log a receipt

(A fallback function can decide what to do by first testing if **msg.gas** is enough.)

# Messages

The transaction or message calling the current contract can be accessed by the keyword

**msg**



It has the following members:

**msg.value : uint** (amount of wei sent with the message)

**msg.sender : address** (sender of the message)

**msg.gas : uint** (amount of gas sent with the message)

**msg.data : bytes** (complete calldata)

**msg.sig : bytes4** (first 4 bytes of the call data, i,e., function identifier)

# Creating a contract from a contract

```
contract D {
  uint public x;
  constructor(uint a)  payable {
    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);
    newD.x();
  }

  function createAndEndowD(uint arg, uint amount) public payable { // Send ether along with the creation
    D newD = (new D).{value:amount}(arg);
    newD.x();
  }
}
```

Constructor (optional)  is called at creation time

d, newD are new  contracts of type D



(Example from Solidity Documentation Section 3.4)

# Destruction

Contracts can also self-destruct by calling

**selfdestruct(address payable recipient)**

This transfers the remaining balance of the account to **recipient**



Warning: if code invoked on a contract by a **delegatecall** contains a **selfdestruct**, then this will also destroy the contract, so know what the code you are invoking does !

# Summary

Solidity, Part 1:

- Headers, Comments
- Basic Types
- Address Types
- Contract Types & declarations
- Function Types & declarations
- Receive and Fallback functions
- Contract creation and destruction