# SOLIDITY 0.6.x

Matteo Berti

# CONTENTS

# INTRODUCTION

# SOLIDITY

Solidity is an object-oriented, high-level language for implementing smart contracts.

- Statically typed
- Compiled to bytecode and executed on the EVM
- Complex user-defined types using mappings and structs
- ABI facilitating multiple type-safe functions
- Supports (multiple) inheritance
- Libraries

Solidity was proposed in 2014 by Gavin Wood. At present it is the primary language on Ethereum as well as Monax and its Hyperledger Burrow blockchain.

# GAS

**Gas** is a special unit used in Ethereum to measure how much work an action (or a set of actions) takes to perform.

It helps to ensure an appropriate fee is being paid by transactions, avoiding to unnecessarily overloading the network with not valuable work.

**NOTE**: transaction fees in Bitcoin are based to the size (in KB) of the transaction.

For each transaction the submitter specifies :

- `gas price` (ETH per GAS)
- `gas limit` (Max amount of GAS)

Too low `gas price` → no miner would take the transaction
Too low `gas limit` → out-of-gas exception (gas is lost)

Total transaction fee = `gas used * gas price`

# SOLIDITY

# EVM

Every node in the Ethereum Network runs an **Ethereum Virtual Machine** instance which allows them to agree on executing the same instructions.

Solidity code is compiled to bytecode and then executed on the EVM, a stack based architecture with 256-bit registers.

**Stack**

256-bit register stack.

Most recent 16 items can be accessed/manipulated at once.

Can only hold 1024 items.

**Memory**

Used by complex opcodes to retrieve or pass data (pointer from stack to memory).

Memory is not persistent.

**Storage**

Where each account can store data indefinitely.

Acts as a public database (no fees to read it).

Expensive to write it.

# CONTRACTS

# SOLIDITY

# CONTRACTS

Each contract has a constructor which is executed only once during deployment.

SOLIDITY

```
pragma solidity >=0.6.0 <0.7.0;

contract MyContract {
   uint256 x = 8;

   constructor() public {
      x = 2;
   }
}
```

ASSEMBLY

```
MSTORE(0X40, 0X80)
SSTORE(0X0, 0X8)
JUMPI(TAG1, ISZERO(CALLVALUE))
REVERT
TAG1 JUMPDEST
SSTORE(0X0, 0X2)
CODECOPY(0x0, 0x22, 0x35)
RETURN
INVALID
```

```
64B memory allocation
Store 8 at position 0
If ETH ≠ 0
Revert state changes
Else continue
Store 2 at position 0
Copy arguments to
memory and return
End execution in EVM
```

**NOTE**: **internal** constructor makes the contract **abstract**.

# FUNCTIONS

# SOLIDITY

# VISIBILITY

## external

External functions are part of the contract interface, so they can be called from other contracts and via transactions. A call to a local external function: `this.extFun()`.

## public

Public functions are part of the contract interface and can be either called internally or via messages.
For **public state variables** an automatic getter function is generated.

## internal

Internal functions can only be accessed internally, i.e. from within the current contract or contracts deriving from it.

## private

Private functions are only visible for the contract they are defined in (not derived).
**State variables are private** by default, i.e. other contracts cannot read them.

**NOTE**: everything is public on the blockchain.

# SOLIDITY

# VISIBILITY

SOLIDITY

```
pragma solidity >=0.6.0 <0.7.0;

contract MyContract {
    function pubFun(uint256[10] memory arr)
             public returns (uint256 v) {
        v = arr[0];
    }

    function extFun(uint256[10] calldata arr)
             external returns (uint256 v) {
        v = arr[0];
    }
}
```

>=0.5.0

**pubFun** loads all parameters in memory because it can be called also internally (and it's executed via jumps and pointers to memory).

PUBFUN ASSEMBLY

```
CALLDATACOPY(0x80, 0x4, 0x140)
...
MSTORE(0x1c0, 0x1)
RETURN
```

!

Execution cost: **443** GAS

EXTFUN ASSEMBLY

```
CALLDATALOAD(0x4)
...
MSTORE(0x80, 0x1)
RETURN
```

Execution cost: **230** GAS

# SOLIDITY

# STATE SPECIFIERS

## view

Functions can be declared view in which case they promise **not** to **modify** the state.

SOLIDITY

```
contract MyContract {
    uint v = 10;

    function vFun() public view returns(uint) {
        return v;
    }

    function pFun() public pure returns(uint) {
        return 10;
    }
}
```

## pure

Functions can be declared pure in which case they promise **not** to **read** from **or** **modify** the state.

CALLER'S ASSEMBLY

```
STATICCALL($GAS, $ADDR, $WEI,
           $IN, $INSIZE,
           $OUT, $OUTPUT)
```

Both specifiers use `STATICCALL`, which enforces the **state** to stay **unmodified** as part of the EVM execution, but does not guarantee the state is not read.

# SOLIDITY

# CUSTOM MODIFIERS

**Modifiers** can be used to change the behaviour of functions in a declarative way. They are inheritable properties of contracts and may be overridden by derived contracts.

SOLIDITY

```
pragma solidity ^0.6.0;

contract MyContract {
    address owner = 0x123...f

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

    function reset() public onlyOwner {
        owner = 0x123...f
    }
}
```

ASSEMBLY

```
JUMPI(TAG2, EQ(CALLER, 0x123...f))
REVERT
TAG2 JUMPDEST
* RESET FUNCTION CODE *
...
STOP
```

The compiler merges the modifier code with the function code

# SPECIAL FUNCTIONS

# SOLIDITY

# RECEIVE and FALLBACK

**Receive**

A contract can have at most one `receive` function, which is executed when receiving plain ETH with **no calldata**, e.g. via `.send()` or `.transfer()`.

Contract-to-contract `.send()` and `.transfer()` functions have no way to specify the gas amount (whereas `call()` does), Solidity imposes a maximum of 2300 GAS.

**Fallback**

A contract can have at most one `fallback` function, which is executed when a function that does **not match any function** declared in the contract is called.

It could be marked as `payable`, in this case is executed in the same circumstances as receive but when some calldata is passed.

# SOLIDITY

# RECEIVE and FALLBACK

SOLIDITY

```
pragma solidity ^0.6.0;

contract MyContract {
    event Log(string data);

    receive() external payable {
        emit Log("Receive");
    }

    fallback() external payable {
        emit Log("Fallback");
    }
}
```

ASSEMBLY

```
MSTORE(0x40, 0x80)
JUMPI(TAG2, CALLDATASIZE)
* RECEIVE CODE *
...
STOP
TAG2 JUMPDEST
* FALLBACK CODE *
...
STOP
INVALID
```

```
64B memory alloc
if CALLDATASIZE
== 0x0:
Receive code

else:
Fallback code
```

# SOLIDITY

# SPECIAL VARIABLES

There are special variables and functions which always exist in the global namespace and are mainly used to provide **information** about the **blockchain** or are general-use **utility** functions.

**Block**

`blockhash(uint blockNumber)` (hash of one of 256 most recent blocks)
`block.coinbase` (current block miner's address)
`block.difficulty`
`block.gaslimit`
`block.number`
`block.timestamp`

**Function call**

`gasleft()`
`msg.data` (complete calldata)
`msg.sender`
`msg.sig`
`msg.value`

**Transaction**

`now`
`tx.gasprice`
`tx.origin` (sender of the tx)

# SOLIDITY

# SPECIAL VARIABLES

SOLIDITY

```
pragma solidity ^0.6.0;

contract MyContract {
   function glob() public payable returns
                                (address,
                                    uint,
                              bytes memory,
                                    uint,
                                 address) {
      return (msg.sender,
              msg.value,
              msg.data,
              tx.gasprice,
              block.coinbase);
   }
}
```

ASSEMBLY

```
...
CALLER
CALLVALUE
PUSH(0x0)
CALLDATASIZE
GASPRICE
COINBASE
...
```

Then loads CALLDATASIZE bytes at position 0x0 of calldata to memory

# SOLIDITY

# ABI

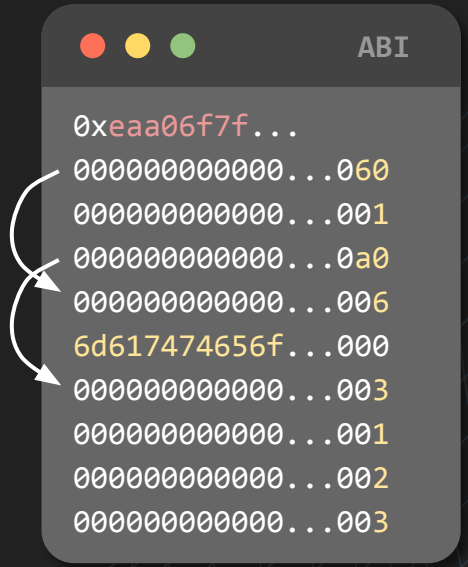
The Contract **Application Binary Interface** (ABI) is the standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction.

SOLIDITY

```
pragma solidity ^0.6.0;

contract MyContract {
   function greet(bytes memory,
      bool, uint[] memory) public {
   }
}
```

```
greet("matteo", true, [1, 2, 3])
```




```
First 4B of keccak(
  "greet(bytes,bool,uint256[])"
)
Location of first param. data
Boolean true value
Location of third param. data
Length of byte array
UTF-8 encoding of "matteo"
Elements in uint256 array
First entry of array
Second entry of array
Third entry of array
```

# ERROR HANDLING

# SOLIDITY

# ERROR HANDLING

## `require(bool, string)`

The **`require`** function should be used to ensure valid conditions that cannot be detected until execution time.

```assembly
JUMPI(TAG2, $COND)
...
MSTORE($MSG)
REVERT
TAG2 JUMPDEST
* REST OF FUNC *
```

## `revert(string)`

The **`revert`** function is another way to trigger exceptions from within other code blocks and revert the current call.

```assembly
...
MSTORE($MSG)
REVERT
...
```

**REVERT**: refund the remaining gas and reverts all changes

## `assert(bool)`

The **`assert`** function should only be used to test for internal errors, and to check invariants.

```assembly
JUMPI(TAG2, $COND)
INVALID
TAG2 JUMPDEST
* REST OF FUNC *
```

**INVALID**: uses all remaining gas and reverts all changes.

# SOLIDITY

# ERROR HANDLING

## try-catch

SOLIDITY

```
pragma solidity ^0.6.0;

contract MyContract {
   function test() public {
      try new Contract(getOwner())
          returns (Contract contr) {
         contr.foo();
      } catch Error(string memory reason) {
         // handle error
      } catch (bytes memory data) {
         // handle error
      }
   }

   function getOwner() {...}
}
```

**NOTE**: when calling a function can be specified the amount of gas, or is passed 63/64 of available gas by default. If a function asserts false all gas passed is lost.

try-catch works only with **external calls**. Only exception inside try guards are caught, those inside getOwner() or try block are not.

<- If a reason string is provided (i.e. from revert() or require())
<- This is executed if: error signature doesn't match any other clause, error during decoding err message, failing assertion in the external call, no error data provided,...

# SOLIDITY

# ERROR HANDLING

## try-catch

SOLIDITY

```
pragma solidity ^0.6.0;

contract MyContract {
   function test() public {
      try new Contract(getOwner())
          returns (Contract contr) {
         contr.foo();
      } catch Error(string memory reason) {
         // handle error
      } catch (bytes memory data) {
         // handle error
      }
   }

   function getOwner() {...}
}
```

ASSEMBLY

```
JUMPI(TAG3, $TRY_CLAUSE_RESULT)
...
JUMPI(TAG2, ISZERO(RETURNDATASIZE))
* STRING CATCH CLAUSE CODE *
...
TAG2 JUMPDEST
* BYTES CATCH CLAUSE CODE *
...
TAG3 JUMPDEST
* TRY BLOCK CODE *
```

try block code is not protected from catch blocks

# CONTRACT CALLS

# SOLIDITY

# TYPECASTING

Wrong type conversions within a contract are detected at compile time, so no specific code is generated at runtime (everything is a sequence of **bytes**).
Instantiating a contract from a wrong address is detected at runtime.

SOLIDITY

```
pragma solidity ^0.6.0;

contract MyContract {
   function caller() public {
      Contract c = Contract(0xf93...WRONG);
      c.action();
   }
}
```

ASSEMBLY

```
JUMPI(TAG2, NOT(ISZERO(EXTCODESIZE)))
REVERT
TAG2 JUMPDEST
CALL($GAS, $ADDR, ...)
* CALLED CONTRACT INIT CODE AND FUN *
```

The size of a called contract is checked at runtime.

**NOTE**: if ETH are sent to the wrong contract that doesn't have the called function, its fallback/receive is executed and ETH are lost, or if there is no fallback and the function (or its par.) is wrong then is reverted.

# SOLIDITY

# CONTRACT CALLS

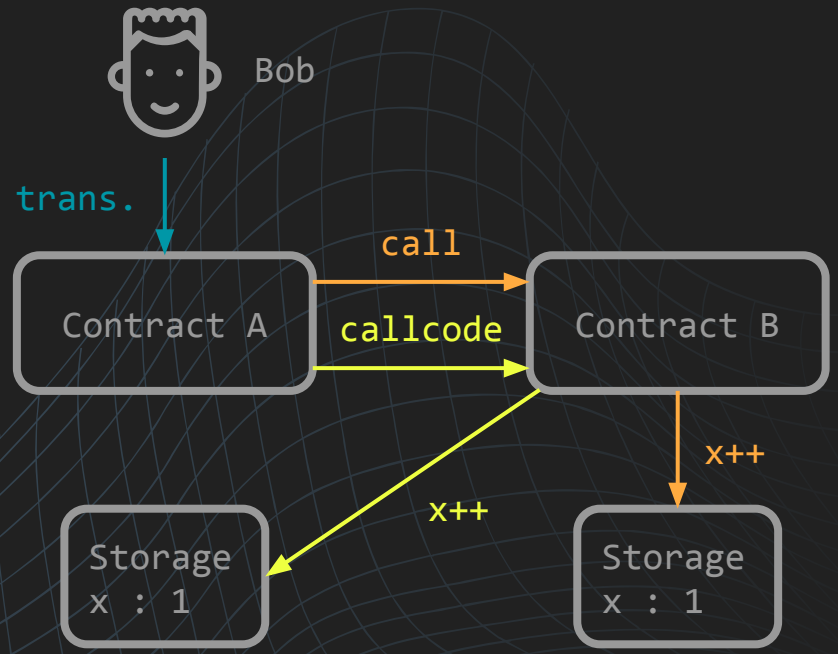
There are (were) four ways to call a contract: CALL, CALLCODE, DELEGATECALL, STATICCALL

## CALL vs CALLCODE

```solidity
addr.call(abi.encodeWithSignature("inc()"));
...
addr.callcode(abi.encodeWithSignature("inc()"));
```

**NOTE**: `callcode` executes an external function in the context of its contract. This must be used with caution because malicious code could even `selfdestroy` the caller.

WARNING: callcode has been deprecated in favor of delegatecall.



# SOLIDITY

# CONTRACT CALLS

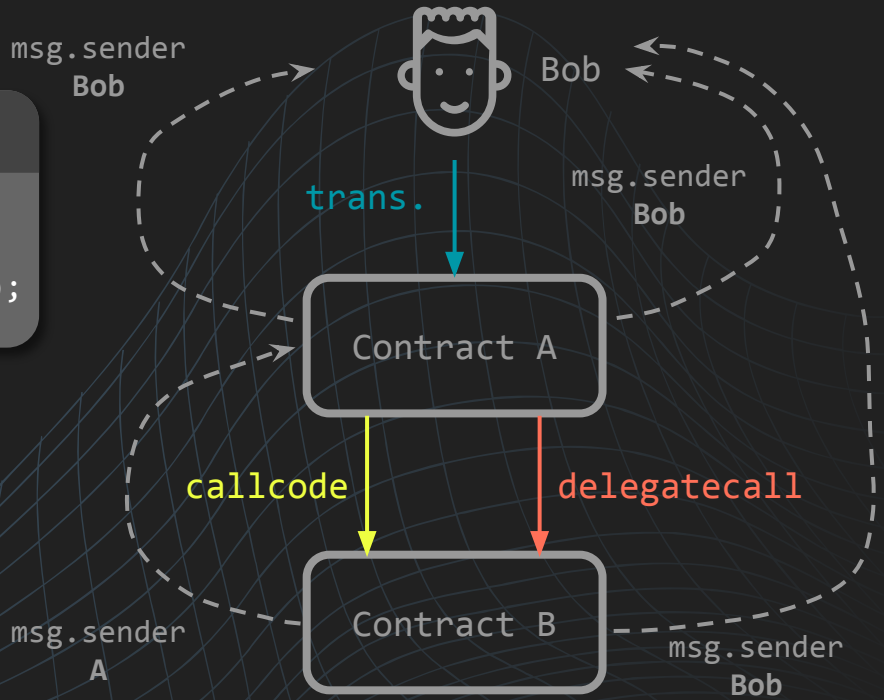
There are (were) four ways to call a contract: `CALL`, `CALLCODE`, `DELEGATECALL`, `STATICCALL`

## CALLCODE vs DELEGATECALL

```solidity
addr.callcode(abi.encodeWithSignature("inc()"));
...
addr.delegatecall(abi.encodeWithSignature("inc()"));
```

## STATICCALL

Cannot be called by a contract, is used at compile time to prevent changes of the contract status (for pure and view functions).



# EVENTS

# SOLIDITY

# EVENTS

Solidity events give an abstraction on top of the **EVM's logging** functionality.

Applications can subscribe and listen to these events through the RPC interface of an Ethereum client.

When an event is called, its arguments are stored in the **transaction's log** - a special data structure in the blockchain.

The log and its event data is not accessible from within contracts.

SOLIDITY

```
pragma solidity ^0.6.0;

contract MyContract {
   event PersonCreated(string indexed name,
                       uint indexed age,
                       address addr);

   function create() public {
      emit PersonCreated(
                     "Matteo",
                     23,
                     address(0x123...789));
   }
}
```

# SOLIDITY

# LOGS

The EVM currently has 5 opcodes for **emitting event logs**: LOG0, LOG1, LOG2, LOG3, LOG4. These opcodes are used to create log records which can be used to describe an event within a smart contract (e.g. token transfer or change of ownership).

Each log record consists of both **topics** and **data**.

`topics`

32B “words” used to describe the event.

- First topic: `keccak(“EventName(uint)”)` [only for non-anonymous events]
- Other topics: `indexed` args of event

Topics > 32B are hashed to 32B

`data`

Including data is a lot cheaper than topics.

While topics are limited to 4*32B, event data is not (used for arrays, bytes, strings).

**TIP**: events are often used instead of storage because cost much less gas.

# SOLIDITY

# LOGS

SOLIDITY

```
pragma solidity ^0.6.0;

contract MyContract {
   event PersonCreated(string indexed name,
                       uint indexed age,
                       address addr);

   function create() public {
      emit PersonCreated(
                     "Matteo",
                     23,
                     address(0x123...789));
      }
}
```

JAVASCRIPT

```
var createdEvent =
myContract.PersonCreated({age: 23})

createdEvent.watch(function(err, result) {
   if (err)
      return;

   console.log("Found ", result);
})
```

`...PersonCreated({age : 23})` filters all persons aged 23 this is possible because age is declared **`indexed`** (is a topic).

# RANDOM NUMBER GENERATOR

SOLIDITY

```solidity
pragma solidity ^0.6.0;

contract RollDice {
   event RequestedRandomness(uint8 min, uint8 max);
   event GenerateRandomness(string value);

   function rollDice(uint8 min, uint8 max) pure external {
       emit RequestedRandomness(min, max);
   }

   function diceResult(
               string calldata randomValue,
               bytes32 hash, uint8 v, bytes32 r, bytes32 s)
               pure external {
      require(ecrecover(hash, v, r, s) == 0x94...F1 &&
              keccak256(
                 abi.encodePacked(
                    "\x19Ethereum Signed Message:\n",
                    uint2str(bytes(randomValue).length),
                    randomValue)
              ) == hash);
      emit GenerateRandomness(randomValue);
   }

   function uint2str(uint _i)
            private pure
            returns (string memory _uintAsString) {...}
}
```

JAVASCRIPT

```javascript
var web3 = new Web3(Web3.givenProvider || WEBSOCKET)
var contract = new web3.eth.Contract(CONTR_ABI, CONTR_ADDR)

contract.events.RequestedRandomness((err, event) => {
   [... generate a random number in range min-max ...]

   var signature = web3.eth.accounts.sign
                        (random.toString(), SENDER_PRIV_KEY)

   contract.methods.diceResult(signature['message'],
                               signature['messageHash'],
                               signature['v'],
                               signature['r'],
                               signature['s'])
                   .send({from: SENDER_ADDR})
                   .catch(function(err) {console.log(err)})
})

contract.events.GenerateRandomness((err, event) => {...})
```

# THANK YOU!

QUESTIONS?