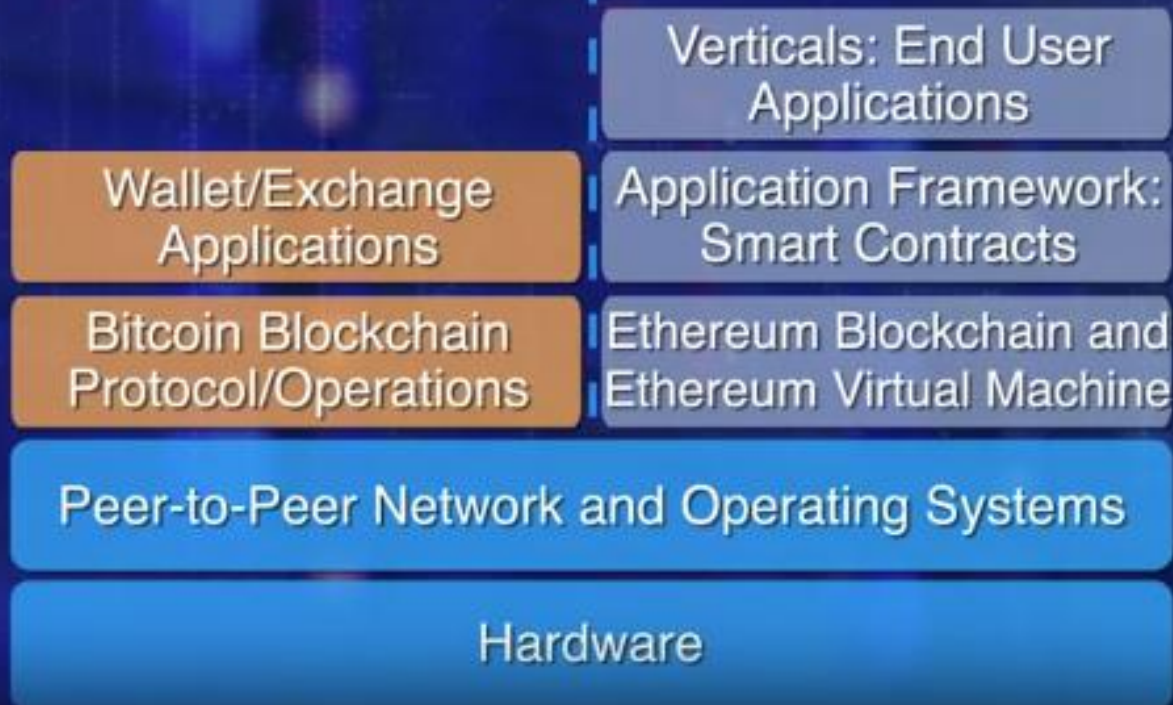


Ethereum Blockchain : Smart Contracts

Bitcoin Stack

Ethereum Stack



Ethereum Stack

Verticals: End User
Applications

Application Framework:
Smart Contracts

Ethereum Blockchain and
Ethereum Virtual Machine (EVM)

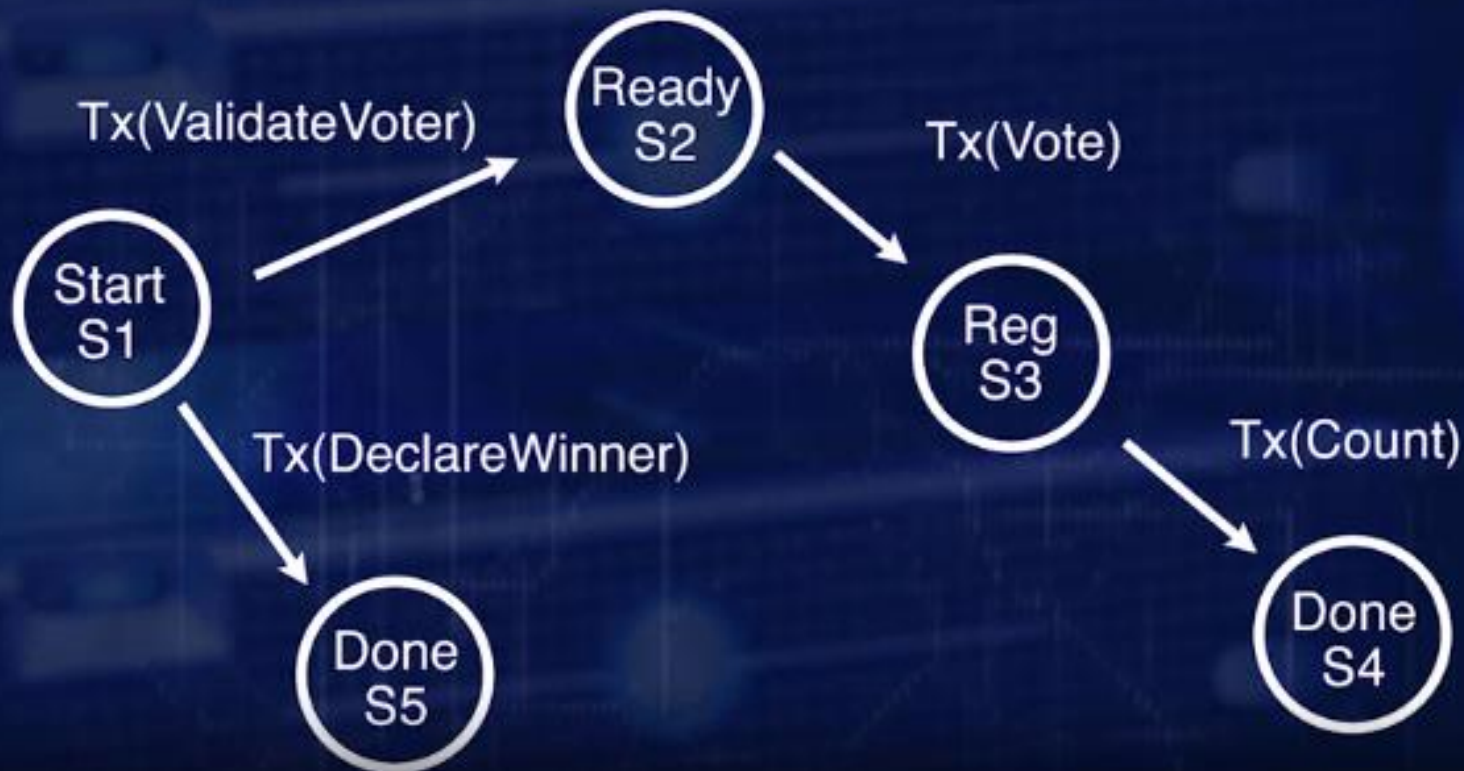
Peer-to-Peer Network and Operating Systems

Hardware

Bitcoin Transaction



Smart Contract Transaction





Currency Transfer

Service

Product

Utility

00101

11001

Transaction



Class-like
with data
& functions

Smart contract for decentralized storage

```
pragma solidity ^ 0.4.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) {
        storedData = x;
    }

    function get () constant returns
    (uint) {
        return storedData;
    }
}
```


Smart Contracts

- Executable code
- Turing Complete
- Function like an external account
 - Hold funds
 - Can interact with other accounts and smart contracts
 - Contain code
- Can be called through transactions

Code Execution

- Every node contains a virtual machine (similar to Java)
 - Called the Ethereum Virtual Machine (EVM)
 - **Compiles** code from high-level language to bytecode
 - Executes smart contract code and broadcasts state
- ***Every full-node on the blockchain processes every transaction and stores the entire state***

Gas

- Halting problem (infinite loop) – reason for Gas
 - Problem: Cannot tell whether or not a program will run infinitely from compiled code
 - Solution: charge fee per computational step to limit infinite loops and stop flawed code from executing
- Every transaction needs to specify an estimate of the amount of gas it will spend
- Essentially a measure of how much one is willing to spend on a transaction, even if buggy

Gas Cost

- Gas Price: current market price of a unit of Gas (in Wei)
 - Check gas price here: <https://ethgasstation.info/>
 - Is always set before a transaction by user
- Gas Limit: maximum amount of Gas user is willing to spend
- Helps to regulate load on network
- Gas Cost (used when sending transactions) is calculated by $\text{gasLimit} * \text{gasPrice}$.
 - All blocks have a Gas Limit (maximum Gas each block can use)

B. Smart Contract Programming

- Solidity (javascript based), most popular
 - Not yet as functional as other, more mature, programming languages
- Serpent (python based)
- LLL (lisp based)

B. Smart Contract Programming

Solidity

Solidity is a language similar to JavaScript which allows you to develop contracts and compile to EVM bytecode. It is currently the flagship language of Ethereum and the most popular.

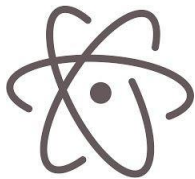
- [Solidity Documentation](#) - Solidity is the flagship Ethereum high level language that is used to write contracts.
- [Solidity online realtime compiler](#)

Serpent

Serpent is a language similar to Python which can be used to develop contracts and compile to EVM bytecode. It is intended to be maximally clean and simple, combining many of the efficiency benefits of a low-level language with ease-of-use in programming style, and at the same time adding special domain-specific features for contract programming. Serpent is compiled using LLL.

- [Serpent on the ethereum wiki](#)
- [Serpent EVM compiler](#)

B. Smart Contract Programming



[Atom Ethereum interface](#) - Plugin for the Atom editor that features syntax highlighting, compilation and a runtime environment (requires backend node).

[Atom Solidity Linter](#) - Plugin for the Atom editor that provides Solidity linting.



[Vim Solidity](#) - Plugin for the Vim editor providing syntax highlighting.

[Vim Syntastic](#) - Plugin for the Vim editor providing compile checking.

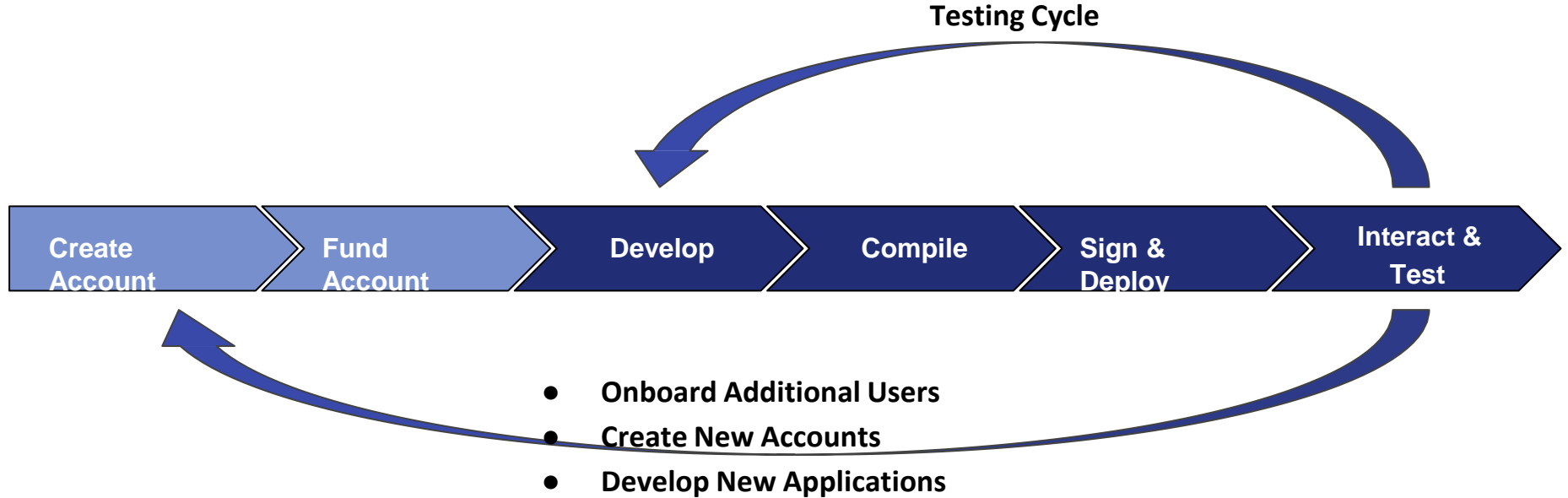
B. Smart Contract Programming: Solidity

```
contract Example {  
  
    uint value;  
  
    function setValue(uint pValue) {  
        value = pValue;  
    }  
  
    function getValue() returns (uint) {  
        return value;  
    }  
  
}
```

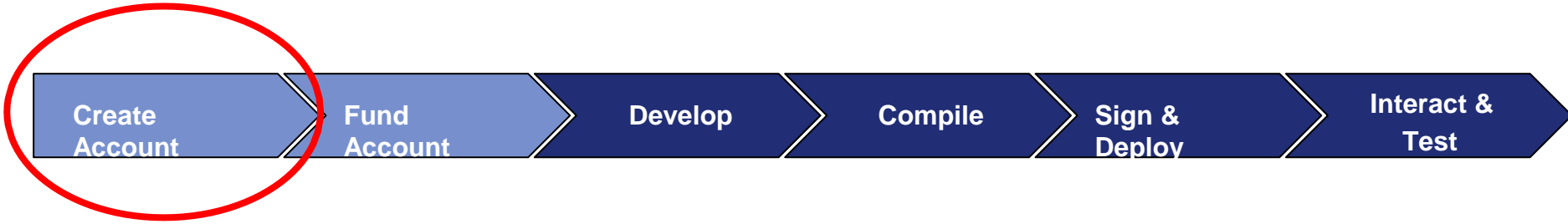
B. Smart Contract Programming: Solidity

```
var logIncrement =  
    OtherExample.LogIncrement({sender: userAddress,  
uint value});  
  
logIncrement.watch(function(err, result) {  
    // do something with result  
})
```

C. Development Workflow

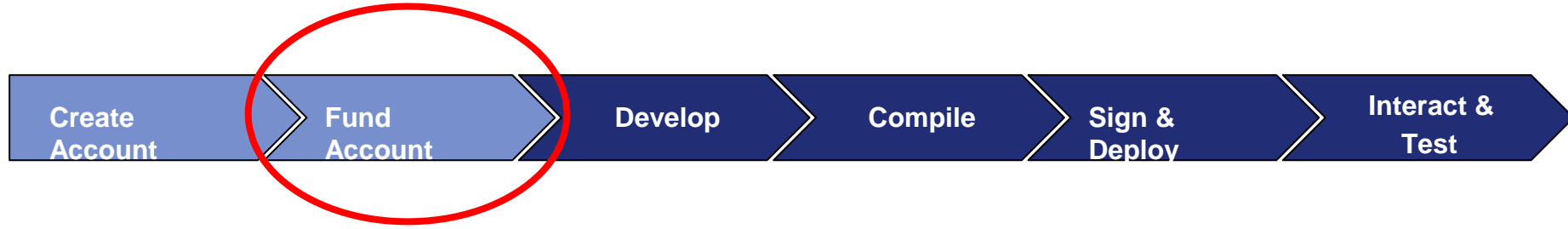


C. Development Workflow: Create Account



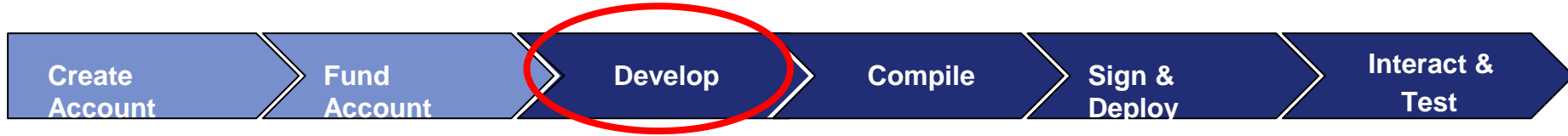
- Programmatically: Go, Python, C++, JavaScript, Haskell
- Tools
 - MyEtherWallet.com
 - MetaMask
 - TestRPC
 - Many other websites

C. Development Workflow: Fund Account



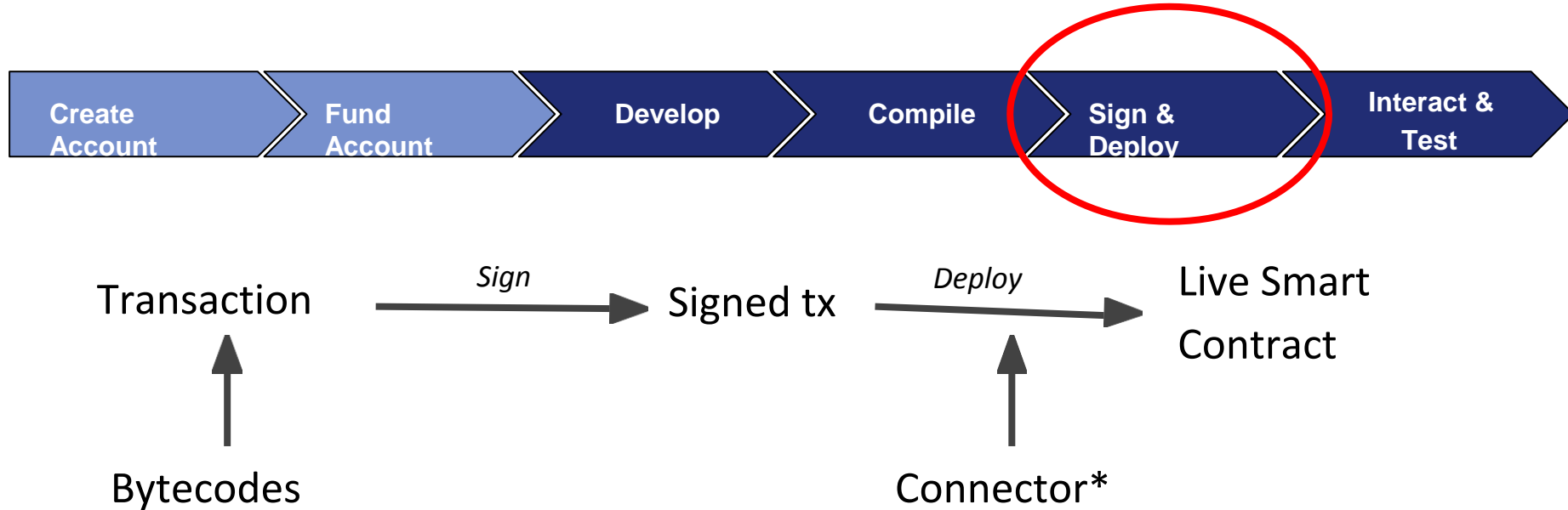
- From friends
- Faucet
- Exchanges (for public blockchain)

C. Development Workflow: Develop



- **Ethereum Application Components:**
 - **Base application:** can be developed in **any** language
 - **Smart contract:** developed in Solidity or one of the other contract compatible languages
 - **Connector library:** facilitates communication between base application and smart contracts (Metamask)

C. Development Workflow: Sign and Deploy



*Library that facilitates communication and connection with Blockchain; Connects your code to a running node.

C. Development Workflow: TestRPC



TestRPC/TestChain

- Local development or Test Blockchain
- <https://github.com/ethereumjs/testrpc>

C. Development Workflow: TestRPC

- EthereumJS TestRPC: <https://github.com/ethereumjs/testrpc> is suited for development and testing
- It's a complete blockchain-in-memory that runs only on your development machine
- It processes transactions instantly instead of waiting for the default block time – so you can test that your code works quickly – and it tells you immediately when your smart contracts run into errors
- It also makes a great client for automated testing
- Truffle knows how to use its special features to speed up test runtime by almost 90%.

Why Smart Contracts?



Transfers assets other
than value or cryptocurrency



Specifies rules for an
operation on blockchain



Implements policies for
transfer of assets in
decentralized networks



Represents business logic layer



Includes messages that
invoke functions

Smart Contracts - Defined

- Smart contract work with the application-specific semantics and constraints of the transaction and verify, validates, and executes them.
- Most of all, since it is deployed on the blockchain, the smart contract leverages the immutable recording and trust model of the blockchain.
- Since a smart contract is deployed in the blockchain, it is an immutable piece of code, and once deployed, it cannot be changed.

Solidity

Javascript
Java
C++



EVM

Solidity Structure

1. Data or state variables
2. List of Functions:
 - a. Constructor
 - b. Fallback
 - c. View
 - d. Pure
 - e. Public
 - f. Private
 - g. Internal
 - h. External
3. User defined types in struct and enums
4. Modifiers
5. Events

Class-like
with data
& functions

Smart contract for decentralized storage

```
pragma solidity ^ 0.4.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) {
        storedData = x;
    }

    function get () constant returns
    (uint) {
        return storedData;
    }
}
```




Pragma directive



Name of the
contract



Data or the state
variables that define
the state of the
contract



Collection of functions
to carry out the intent
of a smart contract



Integrated Development Environment (IDE) Remix

Remix supports test environments

Javascript VM

Injected Web3 (e.g., Metamask)

Web3 Provider (e.g., Ethereum node)



Design



Code



Test

Greeter



string yourName;



Greeter ()



set (string name)

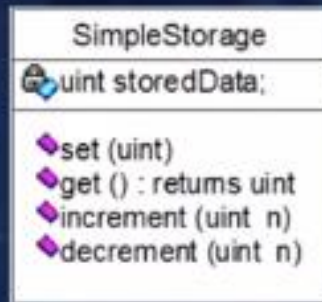


hello () : string


```
pragma solidity ^0.4.0;
```

```
contract Greeter {  
    /* Define variable greeting of the type string */  
    string public yourName;;  
  
    /* This runs when the contract is executed */  
    function Greeter() public {  
        yourName = "World";  
    }  
  
    function set(string name)public {  
        yourName = name;  
    }  
  
    function hello() constant returns (string) {  
        return yourName;  
    }  
}
```

Demo : Greeter.sol



storedData state transitions

```
pragma solidity ^0.4.0;
uint storedData;
function set(uint x) public {
    storedData = x;
}
function get() constant public returns (uint) {
    return storedData;
}
function increment (uint n) public {
    storedData = storedData + n;
}
function decrement (uint n) public {
    storedData = storedData - n;
}
}
```

Demo : SimpleStorage.sol

Processing Smart Contracts

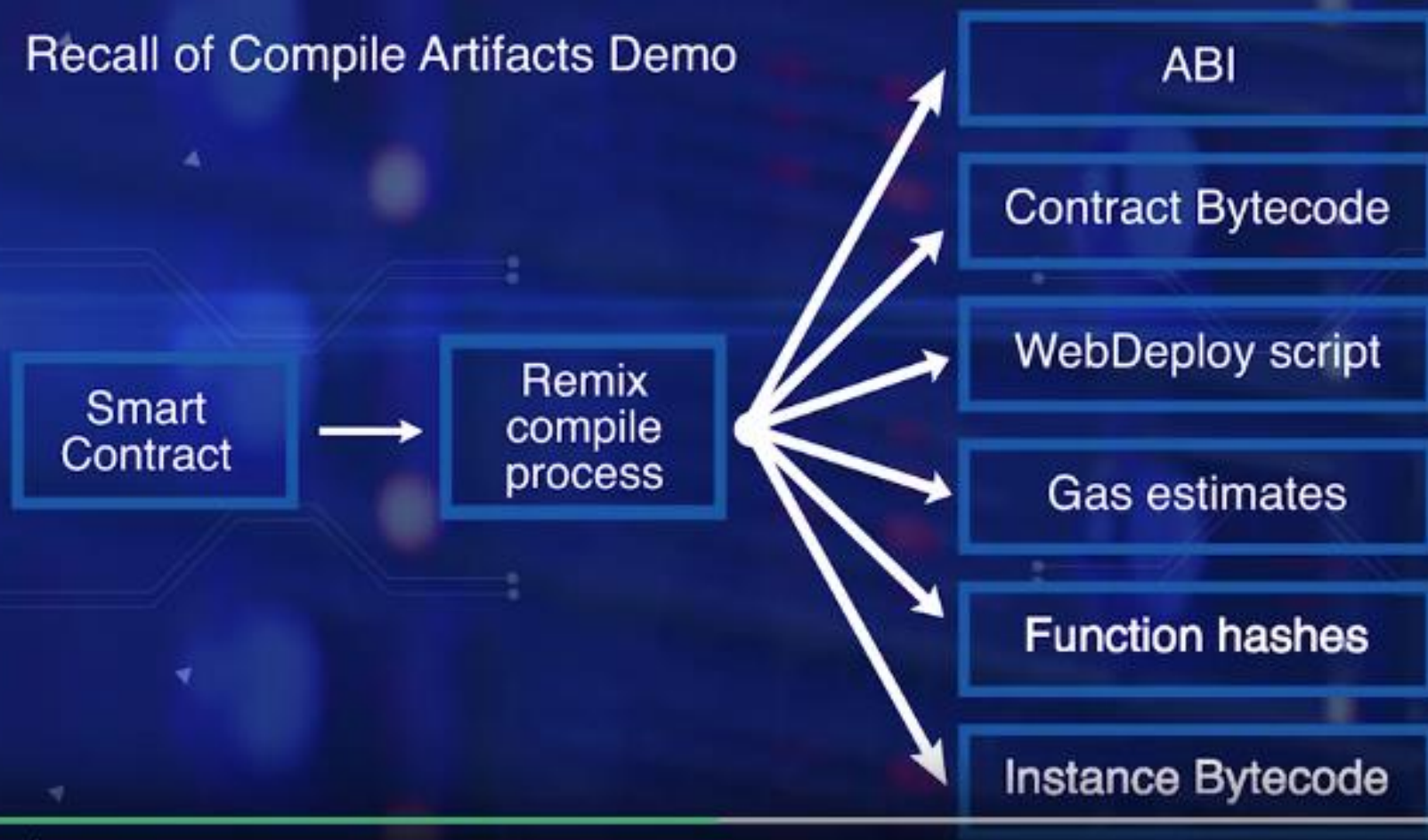
A smart contract can be created,

- on behalf of an externally owned account,
- by application programmatically from the command-line interface and
- by a script of commands from high level applications and user interface or UI.
- It can also be created from inside a smart contract.
- The address of the Smart Contract is computed by hashing the account number of externally owned account EOA and the nonce.

Compiler Artifacts

Demo : Compilation Artifacts

Recall of Compile Artifacts Demo



Compile Artifacts

Name of the contract

Web3 deploy module that provides the script code for invoking the smart contract from a web application

Bytecode executed for the contract “creation” on the EVM

Gas estimates for the execution of the functions

ABI: Application Binary Interface, details functions, parameters and return value

Actual runtime bytecode of the smart contract

Deploying Smart Contracts



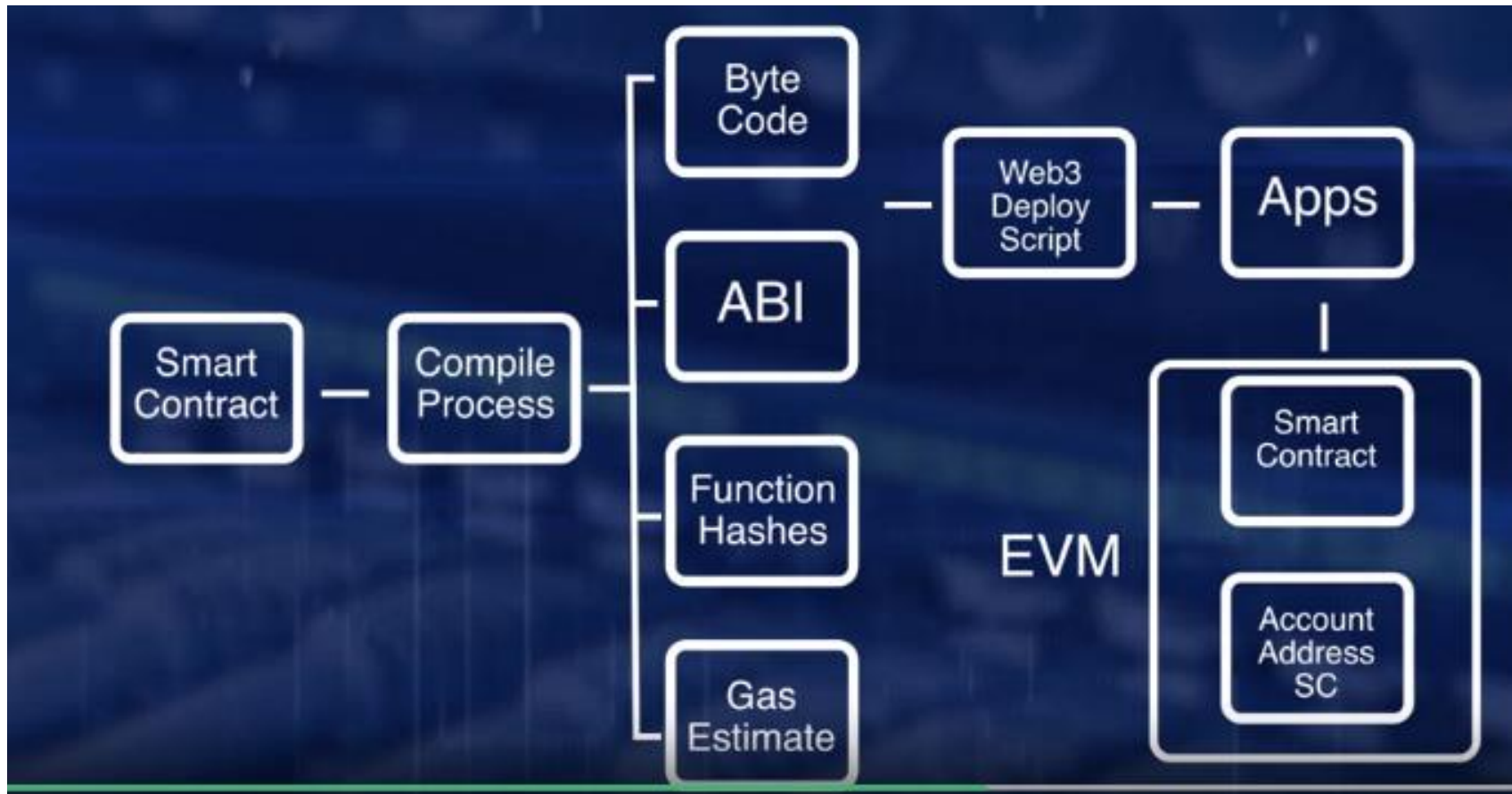
Remix IDE

Another smart contract

A command line interface

Another high level
application

A Web application



Solidity: Structure

1. Data or state variables
2. List of Functions:
 - a. Constructor
 - b. Fallback
 - c. View
 - d. Pure
 - e. Public
 - f. Private
 - g. Internal
 - h. External
3. User defined types in struct and enums
4. Modifiers
5. Events

```
function function-name(parameter-list) scope returns() {  
    //statements  
}
```

Fallback function is a special function available to a contract. It has following features –

- It is called when a non-existent function is called on the contract.
- It is required to be marked external.
- It has no name.
- It has no arguments
- It can not return any thing.
- It can be defined one per contract.
- If not marked payable, it will throw exception if contract receives plain ether without data.

View functions ensure that they will not modify the state. A function can be declared as `view`. The following statements if present in the function are considered modifying the state and compiler will throw warning in such cases.

- Modifying state variables.
- Emitting events.
- Creating other contracts.
- Using `selfdestruct`.
- Sending Ether via calls.
- Calling any function which is not marked `view` or `pure`.
- Using low-level calls.
- Using inline assembly containing certain opcodes.

Pure functions ensure that they not read or modify the state. A function can be declared as pure. The following statements if present in the function are considered reading the state and compiler will throw warning in such cases.

- Reading state variables.
- Accessing `address(this).balance` or `<address>.balance`.
- Accessing any of the special variable of block, tx, msg (`msg.sig` and `msg.data` can be read).
- Calling any function not marked pure.
- Using inline assembly that contains certain opcodes.

Pure functions can use the `revert()` and `require()` - functions to revert potential state changes if an error occurs.

external: External functions are part of the contract interface, which means they can be called from other contracts and via transactions. An external function `f` cannot be called internally (i.e. `f()` does not work, but `this.f()` works). External functions are sometimes more efficient when they receive large arrays of data.

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: 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.

Basic Data Types & Statements

Solidity

uint : unsigned int of 256 bits





int : integer positive and negative value accepted 256 bits




string: string of characters

bool : that supports logic true and false value

- Default modifier is private.
- You explicitly state the public modifier, if that is what is intended

Bidder

 string name;
 uint bidAmount;
 bool eligible;
 uint minBid;

 setName()
 setBidAmount()
 determineEligibility()

Demo : Bidderdata.sol

Demo : Bidder.sol

Specific Data Types

Objectives

Explain important data structures of Solidity:

address

mapping

message (msg)

Explain Solidity events that log events and push data to an application level listener

Address: It is a special Solidity define composite data type.

- It can hold a 20-byte ethereum address.
- Address is a reference address to access a smart contract.
- Address data structure also contains the balance of the account in Wei.
- It also supports a function transfer, to transfer a value to a specific address

```
<address>.balance (uint256):
```

```
balance of the Address in Wei
```

```
<address>.transfer(uint256 amount):
```

```
transfer given amount of Wei to Address
```

Mapping:

- Mapping is a very versatile data structure that is similar to a key value store, it also can be thought of as a hash table.
- The key is typically a secure hash of a simple Solidity data type such as address and the value in key-value pair can be any arbitrary type.

```
mapping (uint => string) phoneToName;
```

```
struct customer {  
    uint idNum;  
    string name;  
    uint bidAmount;  
};
```

```
mapping (address => customer) custData;
```

Message:

- Message is a complex data type specific to smart contract.
- It represents the call that can be used to invoke a function of a smart contract.



```
address adr = msg.sender
```

```
uint amt = msg.value
```

- msg.sender that holds the address of the sender,
- msg.value that has the value in Wei sent by the sender

Coin

◆ address minter;
◆ mapping (address => uint) balances;

◆ Coin()
◆ mint()
◆ send()
◆ event sent()

Demo : Coin.sol

Data Structures

Objectives

Explain the syntax and usage of arrays, enum and struct data types of Solidity.

Illustrate the use of time units pre-defined in Solidity.

Ballot

```
struct Voter;  
struct Proposal;  
Proposal[] proposals;  
mapping(address => Voter) voters;  
address chairperson;
```

```
Ballot( )  
register( )  
vote( )  
winningProposal( )
```

Ballot Smart Contract:

The creator is the chairperson who gets a weight of 2 for her vote, others get a weightage of 1 for their 1 vote.

Each voter has to be registered first, by the chairperson, before they can vote. They can vote only once.

Ballot Smart Contract:

A constant function is included to enable the client applications to call to obtain the result.

The constant modifier of the function prevents it from changing any state of the smart contract.

This call comes directly to the smart contract and not via a transaction, so it is not recorded on the blockchain.

- Struct is a composite data type of a group of related data that can be referenced by a single, meaningful, collective name.
- Individual elements of the struct can be accessed using the dot notation.

```
struct voter {  
    uint weight;  
    bool voted;  
    uint8 vote;  
}
```


Ballot

```
struct Voter;  
struct Proposal;  
Proposal[] proposals;  
mapping(address => Voter) voters;  
address chairperson;
```

```
Ballot( )  
register( )  
vote( )  
winningProposal( )
```

Demo : Ballotv1.sol

Time units in Solidity



Epoch & Unix Timestamp Conversion Tools

The current Unix epoch time is **1519848950**

www.epochconverter.com

The **Unix epoch** (or **Unix time** or **POSIX time** or **Unix timestamp**) is the number of seconds that have elapsed since January 1, 1970 (midnight UTC/GMT), not counting leap seconds

Unix Epoch time

Used in timestamping the block time when a block is added to the blockchain.

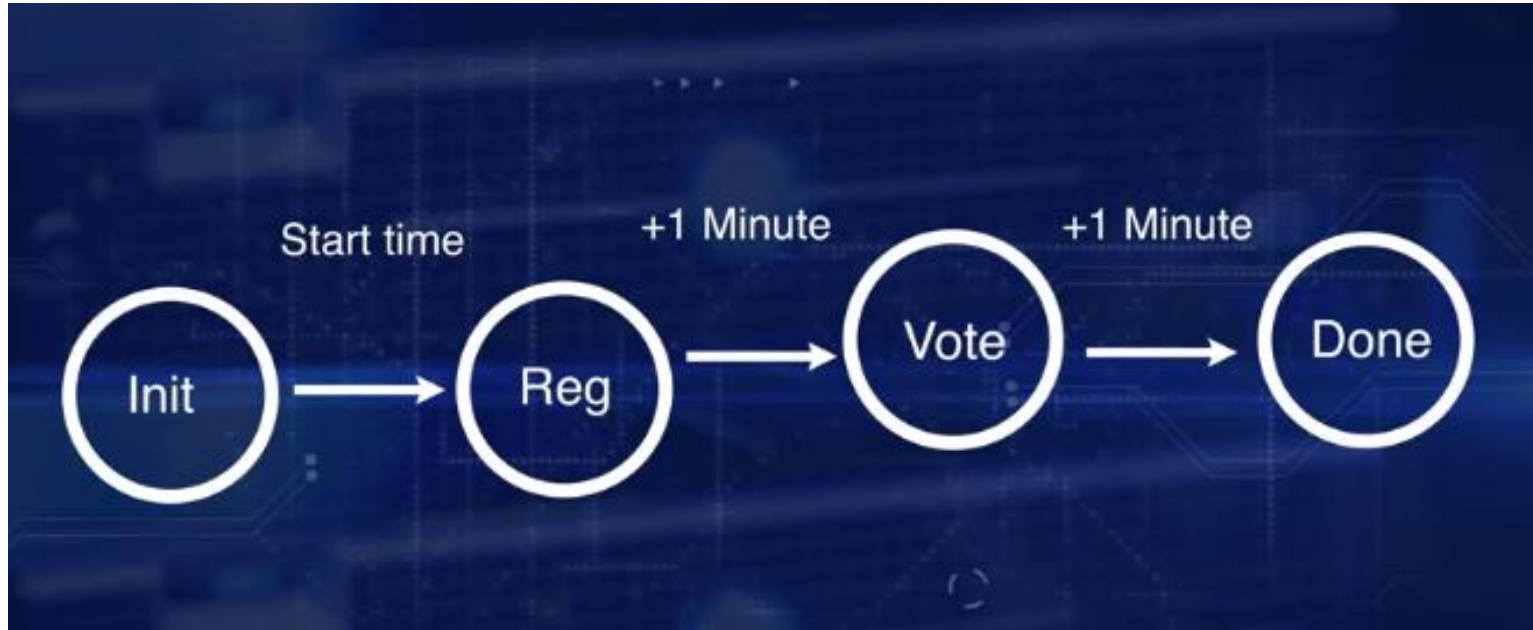
All the transactions confirmed by the block also have the block time as their confirmation time.

A variable called “now” defined by Solidity returns the block timestamp. This variable is often used for evaluating time-related conditions.

```
if (now >= creationTime + 1 day) stage = Stage.RegDone;  
if (now <= voteStartTime + 60 minutes) ...then allow to vote
```

Enum data type

- Enum or enumerated data type, allows for user defined data types with limited set of meaningful values.
- It is mostly used for internal use and are not supported currently at the ABI level of solidity.



Demo : StateTrans.sol

Access Modifiers & Applications

Objectives

Explain function modifiers

Explain “require” clause

Illustrate assert declaration

Discuss revert () function

Modifiers can change behavior of a function.

- It is also known as a function modifier since it is specified at the entry to a function and executed before the execution of the function begins.
- You can think of a modifier as a gatekeeper protecting a function.
- A modifier typically checks a condition using a require and if the condition failed, the transaction that call the function can be reverted using the revert function.

Modifier and required clauses using the Ballot smart contract functions:

Define modifier for clause “onlyBy(chairperson)”

Adding the special notation (_;) to the modifier definition that includes the function

Using the modifier clause in the function header

//step 1

modifier onlyBy(address _account)

{

require(msg.sender == _account);

_; /*Note this step 2*/

}

//Step 3:

use the modifier clause in the
header function register(address toVoter)

```
public onlyBy(chairperson) {  
    if ( voters[toVoter].voted) return;  
    voters[toVoter].weight = 1;  
    voters[toVoter].voted = false;  
}
```

```
function register(address toVoter) public  
onlyBy(chairperson) {  
    if ( voters[toVoter].voted) revert ();  
    voters[toVoter].weight = 1;  
    voters[toVoter].voted = false;  
}
```



```
function payoff (address better) public
{
    /*compute & payoff all the betters */
    assert (bank.balance >10000);
    /* revert the call and any state transitions if
    bank balance falls below a reserve of 10000 */
}
```

Rules, laws, policies,
governance
coded as modifier



Function guard conditions
Modifiers (Tx revertible)
referenced in the function
header

Function:

Function Header

Input arguments
validation using "require"
(Tx revertible)

Function Code

Assertion (Tx revertible)

Execution order

Rule: atLeast 5 sellers have registered



```
Modifier atLeast5Sellers{  
  require( numSellers>=5);  
  ..;}
```



Function buy (..) payable
atLeast5sellers..returns(..)

Enough value to buy the
selected item?

Function code: collect
value and transfer digital
item

Assertion
(itemTransferred);

Execution order

Example: Online bazaar

Reference:

Bina Ramamurthy, Smart Contracts [MOOC]. Coursera.
<https://www.coursera.org/learn/smarter-contracts>