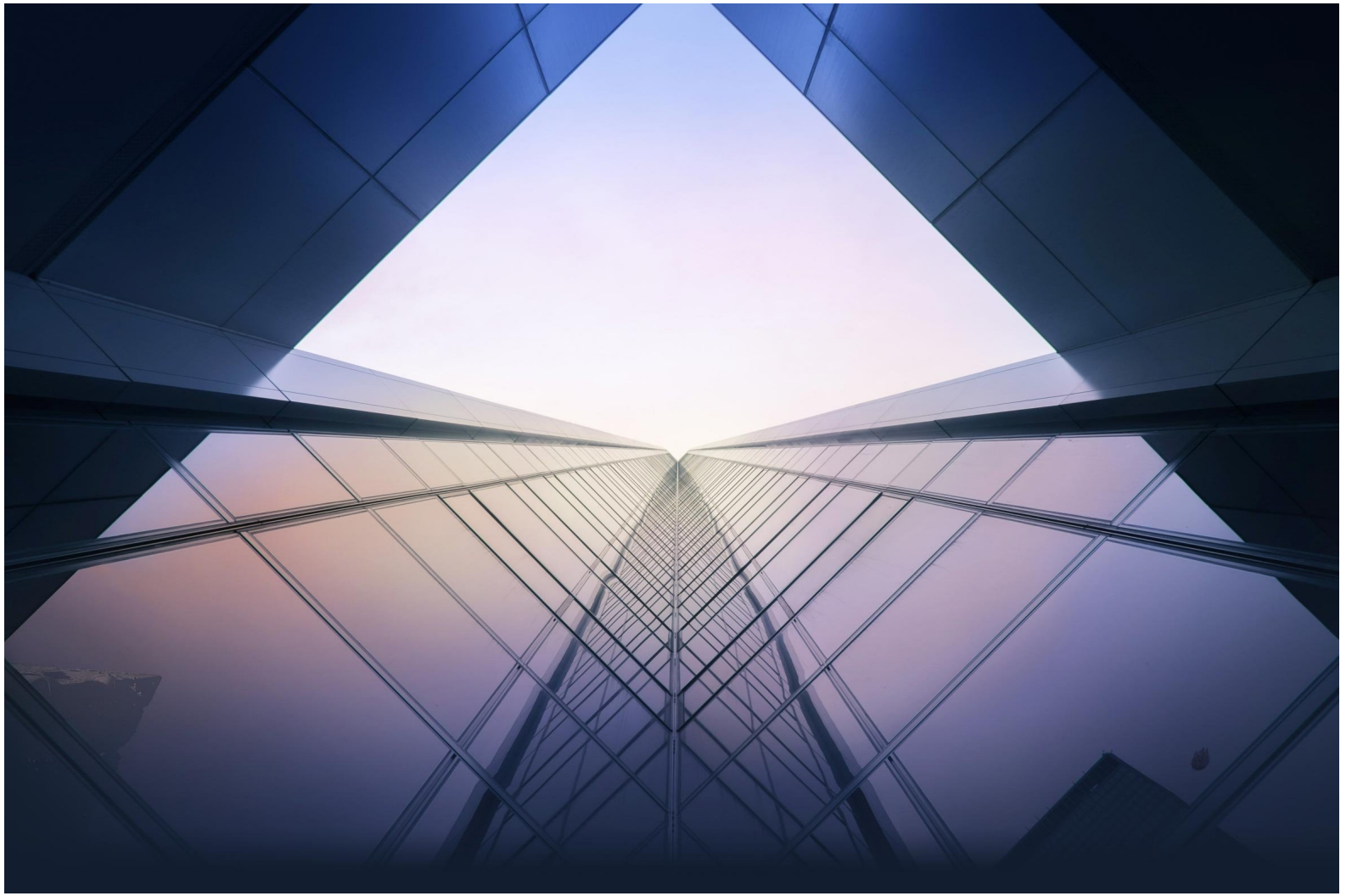
Introducción al Desarrollo de Aplicaciones Descentralizadas (dApps) para la Blockchain Ethereum

Miguel Angel Astor
miguel.astor@ciens.ucv.ve

Agenda

- Introducción a Ethereum
- Contratos Inteligentes
- El Lenguaje de Programación Solidity
- Herramientas y Flujo de Trabajo
- Tokens en Ethereum
- Conclusiones

Introducción a Ethereum



Ethereum

- Ethereum es una plataforma de cómputo distribuido basada en Blockchain.
- Centrada en una criptomoneda llamada *ether* (ETH)

Origenes de Ethereum

- Propuesto en 2013 por V. Buterin.
- Especificación inicial en el Ethereum white-paper y especificación de la EVM en el Ethereum yellow-paper.

Roadmap de Ethereum

- El desarrollo de Ethereum sigue estas etapas:
  - **Olympic** *(5-2015)*: Primera versión pública.
  - **Frontier** *(7-2015)*: Versión beta pública.
  - **Homestead** *(3-2016)*: Primera versión estable.
  - **Metropolis**: Segunda versión estable. Publicada en dos etapas:
    - **Byzantium** *(10-2017)*: Cambios a la EVM. Introdujo zk-SNARKS.
    - **Constantinople** *(2-2019)*: Nuevas operaciones en la EVM.
  - **Serenity** *(Versión futura)*: Cambio completo a PoS.

Características Técnicas de Ethereum

- Algoritmo de consenso Ethash tipo PoW:
  - Resitente a minado ASIC.
  - Basado en el algoritmo Keccak (SHA-3).
- Distinción entre transacciones y mensajes.
- Los bloques se generan cada 14 segundos.
- No hay límite a la emisión de ETH.

Características Técnicas de Ethereum

- Ejecución descentralizada.
- Ejecución de código mediante una máquina virtual.
- Cálculo de comisiones según cantidad y tipo de operaciones realizadas en la máquina virtual (*gas*).
- Dos tipos de cuenta:
  - Cuentas de propiedad externa (claves pública y privada).
  - Cuentas de contrato (sin claves).

Cuentas en Ethereum

- Toda cuenta en Ethereum contiene:
  - Dirección.
  - Balance en Wei ($1x10^{-18}$ ETH).
  - Nonce.
  - Código (opcional).
  - Almacenamiento (vacío por defecto).

Transacciones y Mensajes

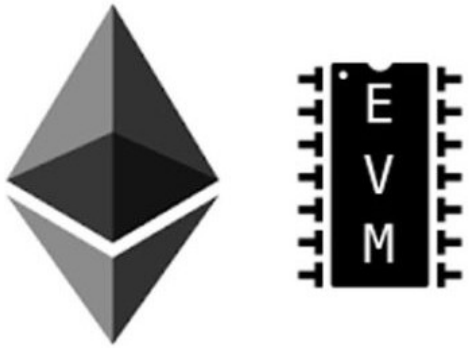
- Mensajes:
  - Pueden ser creados externamente o por contratos.
  - Pueden contener datos arbitrarios.
  - Pueden retornar respuestas.
- Transacciones:
  - Paquete de datos que almacena un mensaje de origen externo.
  - Almacenadas en la Blockchain.

Función de Transición de Estado

1. Verificar que la transacción sea válida.
2. Calcular $fee = max\_gas * gas\_price$.
3. Hacer $gas = max\_gas$.
4. Transferir valor de emisor a receptor.
    1. Si el receptor es un contrato, ejecutar su código.
5. En error, revertir cambios excepto pago de comisión.
6. En éxito, pagar al minero y reembolsar gas no usado.

La Máquina Virtual de Ethereum EVM

- Máquina de pila.
- Palabras de 256 bits.
- Definida como una tupla:
  - (Estado, Transacción, Mensaje, Código, Pila U Mem, PC, Gas)



Modelo de Ejecución de la EVM

- Posee 3 tipos de memoria:
  - Pila LIFO de 32 bits.
  - Memoria principal, lineal y direccionada por palabras.
  - Almacenamiento persistente de 256 bits.
- Al instanciar la EVM se inicializa la pila y la memoria a 0 (cero).
- Manipular el almacenamiento implica cobros y/o reembolsos adicionales.

Modelo de Ejecución de la EVM

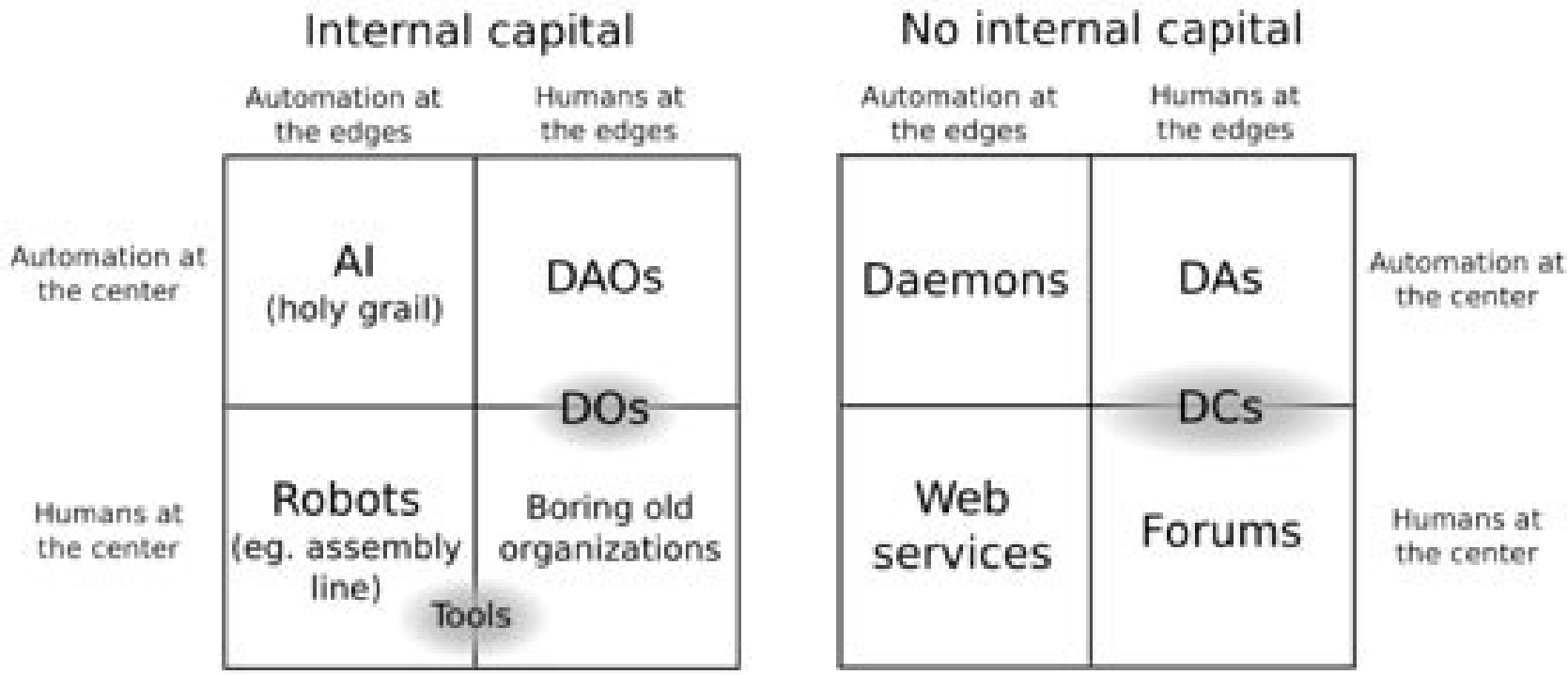
- Control de flujo excepcional:
  - *Stack underflow*.
  - Op-Code no válido.
  - Excepción Out-of-Gas.
  - Excepciones provocadas por el programa.

Aplicaciones Descentralizadas

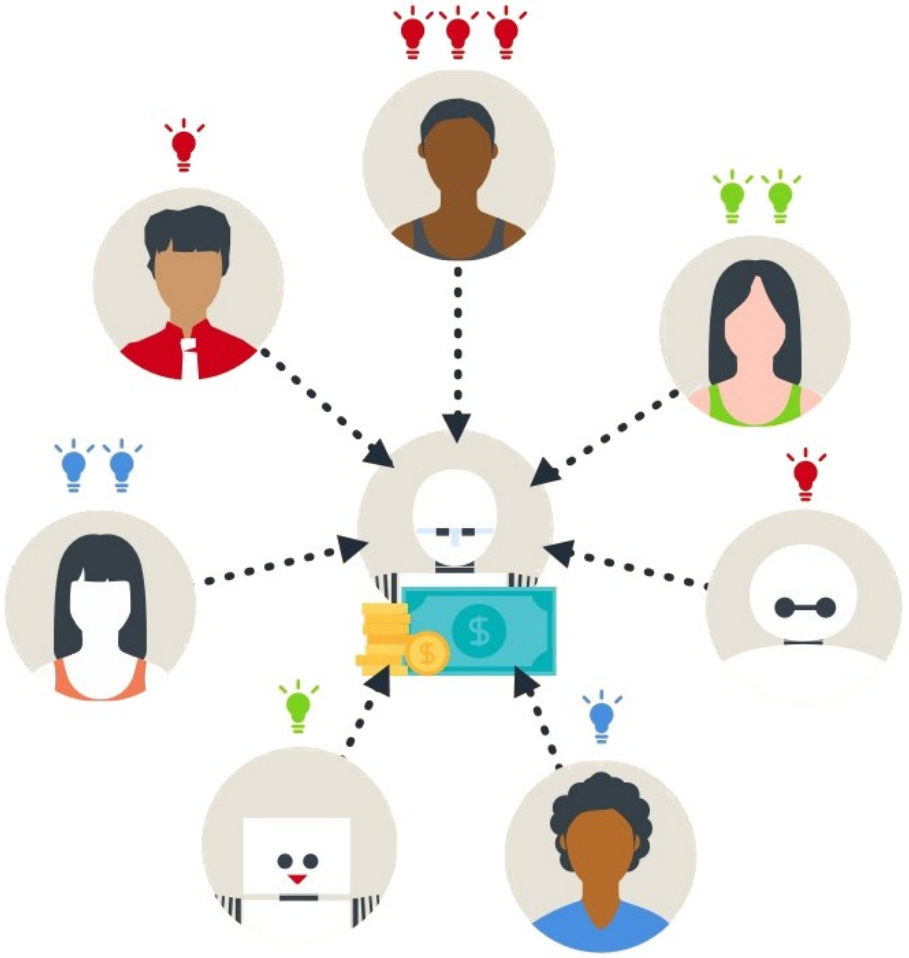
- Aplicaciones conformadas por un *frontend* que interactúa con un *backend* apoyado en la Blockchain de Ethereum.
  - El *frontend* puede estar escrito en cualquier lenguaje que pueda interactuar con la Blockchain.
  - El *backend* es un contrato inteligente.

DO’s, DAO’s y otros conceptos

- Con Ethereum es posible pensar en comunidades y organizaciones descentralizadas y autónomas.



Contratos Inteligentes



Antecedentes

- El término “contrato inteligente” fue inventado en 1996 por Nick Szabo, para hacer referencia a sistemas de *E-Commerce* con lógica de negocios programable.

Programación en Bitcoin

- Bitcoin permite programar transacciones con *Script*:
  - Lenguaje similar a Forth.
  - No es Turing-completo.
  - Limitado por razones de seguridad.
  - Ejecutado en una máquina virtual.

Contratos Inteligentes en Ethereum

- También son llamados “objetos autónomos”.
- Son programas que permiten embeber lógica de negocios arbitraria en las transacciones de Ethereum.
- Modelo de cómputo Turing-completo:
  - Basado en una máquina de pila (la EVM).
  - Ejecución descentralizada.
  - Estado persistente.

Ethereum Como Plataforma de Cómputo

- El objetivo de Ethereum es ser una plataforma de cómputo distribuido primero y un sistema financiero después.
  - Los contratos inteligentes son las bases de las dApp’s.
  - Las dApp’s son el mecanismo de cómputo de Ethereum:
    - Modelo de red *Peer*-2-*Peer* (P2P).
    - Almacenamiento persistente transaccional verificado mediante la Blockchain de Ethereum.
    - Ejecución asíncrona y concurrente en los nodos de la red Ethereum.

Operaciones Sobre Contratos Inteligentes

- Ethereum posee cuatro operaciones que se pueden aplicar sobre los contratos inteligentes:
  - **Despliegue**: Enviar un contrato a la red.
  - **Ejecución**: Ejecutar una función pública del contrato.
    - Siempre implica un costo en *gas*. Puede incluir transferencias de valor.
  - **Consulta**: Examinar el estado del contrato.
    - Nunca implica costos ni transferencia de valor.
  - ***Destrucción***: Desactivación del contrato.

Programación de Contratos Inteligentes

- Hay dos formas de programar contratos inteligentes:
  - **Manualmente**: Uso directo de las herramientas del compilador *SOLC* y un nodo Ethereum por el desarrollador.
  - **Frameworks**: Uso de Truffle u otro framework para simplificar y automatizar el proceso de desarrollo.
- La programación se realiza con el lenguaje de programación Solidity.

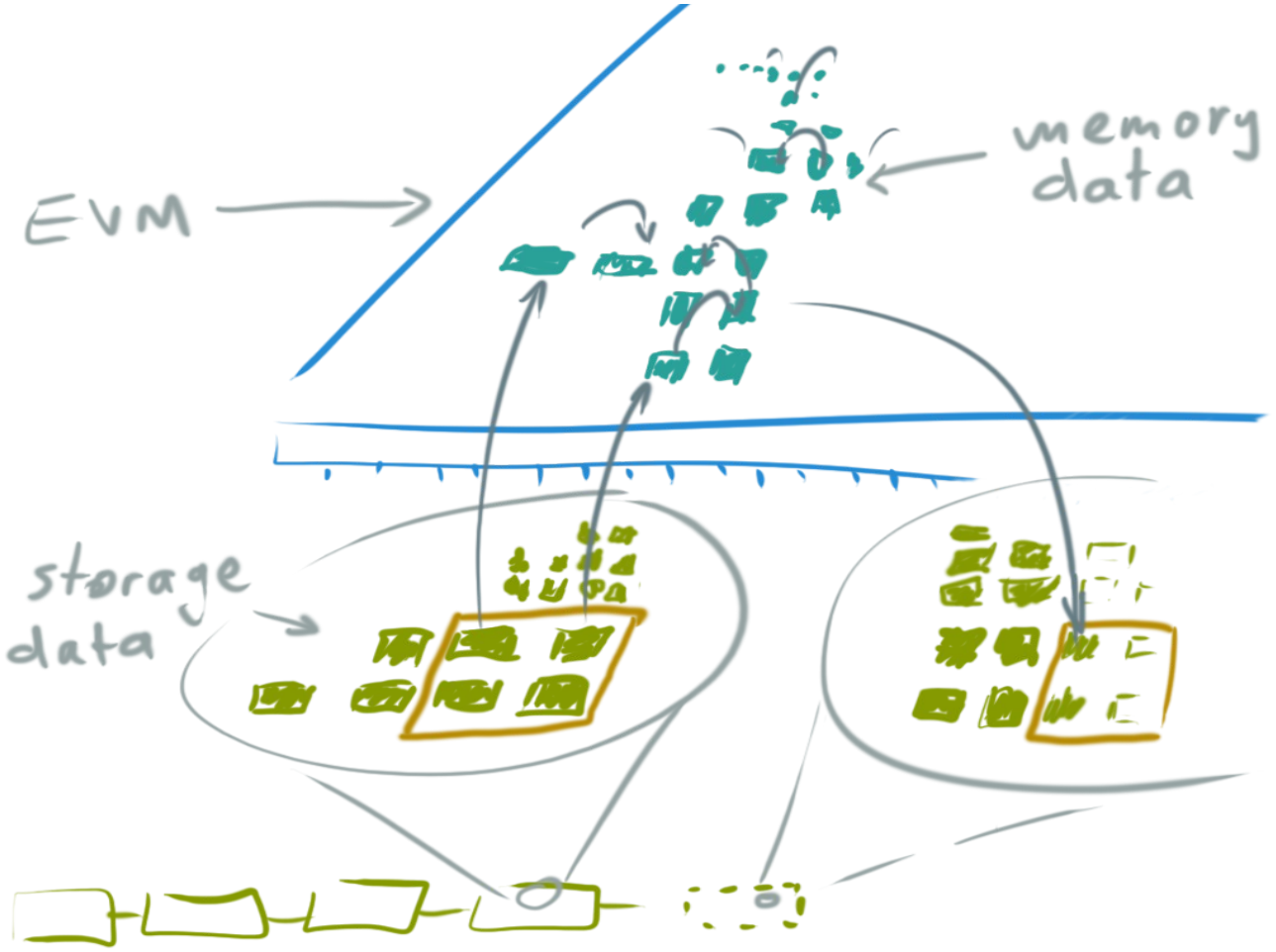
Hello, Solidity!

```
pragma solidity >0.4.99 <0.6;
contract Greeter {
    address payable owner;
    string greeting;
    constructor() public {
        owner = msg.sender;
        greeting = “Hello, World!”;
    }
    function kill() {
        if (msg.sender == owner) selfdestruct(owner);
    }
    function greet() constant returns (memory string) {
        return greeting;
    }
}
```

Características Técnicas de Solidity!

- Solidity es un lenguaje orientado a objetos:
  - Encapsulamiento.
  - Herencia múltiple.
  - Polimorfísmo.
  - Despacho virtual.
- Lenguaje para dominio específico (DSL) que se compila a *bytecode* de la EVM.

Tipos de datos y categorías de almacenamiento



Tipos de datos elementales en Solidity

- Tipos elementales:
  - Enteros con y sin signo, de 8 a 256 bits
  - Direcciones de 20 bytes
  - Cadenas de caracteres
  - Booleanos
  - Enumerados
  - Arreglos de tamaño fijo

Enteros con y sin signo

- Enteros de 1 a 32 bytes (8 a 256 bits) con/sin signo
  - `uint8 unsigned_eight_bits;`
  - `int16 signed_sixteen_bits;`
  - `uint24 unsigned_twenty_four_bits;`
  - `int32 signed_thirty_two_bits;`
  - `...`
  - `uint248 unsigned_etc_bits;`
  - `uint256 unsigned_etc_bits;`
- Operadores lógicos, aritméticos y bit-a-bit (complemento a 2)
- Truncamiento al convertir de tipos grandes a pequeños

Direcciones

- Son objetos que representan una dirección pública de Ethereum
  - `address an_address;`
- Soportan el calificador payable
  - `address payable an_address;`
- Las direcciones `payable` pueden recibir *ether* con el método transfer.
- Pueden convertirse de/a *uint160* o de/a *bytes20*

Cadenas de caracteres

- Similares a las de Python
  - `string message = “Hello, World!”;`
  - `string in_spanish = ‘Buenos días, señoras y señores!’;`
- Soportan escapes típicos de C/C++
  - `\n`, `\t`, `\r`, `\”`, `\’`, `\\`, `\xNN` y `\uNNNN`
- Los caracteres se codifican con UTF-8
- Se pueden convertir de/a bytes o de/a bytes1 .. bytes32
- Los caracteres ***NO*** son indexables individualmente

Booleanos

- Similares a los de C++
  - `bool a = true;`
  - `bool b = false;`
- Usan lógica de cortocircuito

Enumerados

- Similares a los de C/C++ (clásicos, no como *enum class* en c++11)
  - `enum Directions { NORHT, SOUTH, EAST, WEST }`
  - `Directions a_direction = Directions.NORTH;`
  - `uint256 as_uint = uint256(a_direction);`
  - `Directions other_direction = Directions(1);`
- Pueden convertirse de/a cualquier tipo entero explícitamente

Arreglos de tamaño fijo

- Arreglos de 1 a 32 bytes
  - `byte a;`
  - `byte1 b;`
  - `...`
  - `byte32 d;`

- Poseen un atributo público *length*
- Son indexables
- Mismos operadores que los enteros
- Pueden convertirse de/a enteros

Tipos de datos compuestos en Solidity

- Tipos compuestos:
  - Arreglos de tamaño dinámico
  - Registros (*structs*)
  - Asociaciones (*mappings*)

Arreglos

- Arreglos de cualquier tipo
  - `uint256[10] ten_numbers;`
  - `uint256[] lotsa_numbers;`
  - `uint256[20][10] fixed_matrix; // !`
  - `uint256[][10] dynamic_matrix; // !`
  - `uint256[20][] other_matrix public; // !`

- Pueden tener tamaño fijo o dinámico
- Solidity genera *getters* automáticamente si son declarados `public`
- Indexados desde cero (0)

Registros

- Funcionan como los *structs* de C
  - `struct Date {`
    - `uint16 day;`
    - `uint8 month;`
    - `uint64 year;`
  - `}`

  - `Date d;`
- No pueden contener referencias a si mismos
- Pueden contener arreglos o *mappings* arbitrarios

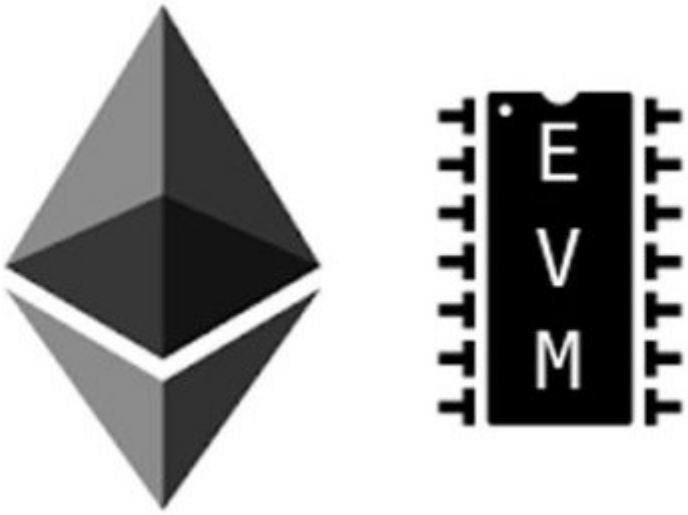
Asociaciones

- Asociaciones clave-valor

  - `mapping(address => string) a_mapping;`
  - `mapping(address => uint256) token_balances;`
  - `mapping(string => MyStruct) other_mapping public;`

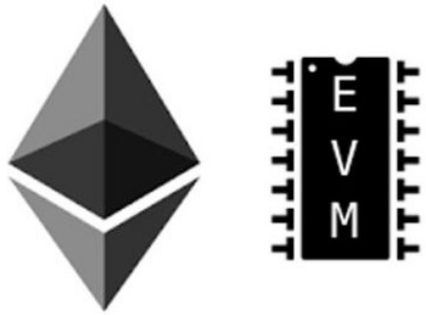
- Las claves pueden ser cualquier tipo elemental
- El valor puede ser cualquier tipo de Solidity excepto otro *mapping*
- Solidity genera *getters* automáticamente si son declarados `public`

Clases de almacenamiento



Modelo de memoria de la EVM

- Tres memorias de trabajo:
  - Pila (palabra de 256 bits)
  - Memoria principal (palabra de 256 bits)
  - Almacenamiento o Estado



Almacenamiento de variables compuestas

- Toda variable compuesta debe definir una categoría de almacenamiento al ser usada en una función
  - memory o storage
  - Aplica para *strings*, arreglos y *structs*
  - Los *mappings* siempre usan storage
- Opcional antes de Solidity 0.5
- La categoría de almacenamiento afecta la semántica de las asignaciones

Diferencias entre *storage* y *memory*

- Las asignaciones entre *storage* y *memory* siempre implican una copia de la estructura de datos
- Asignaciones de *memory* a *memory* o de storage a storage se realizan por referencia
- Cualquier otra asignación a *storage* se realiza por copia
- Los atributos de un contrato siempre son *storage*
- Asignar a *storage* es más costoso que a *memory*

Declaración de clase de almacenamiento

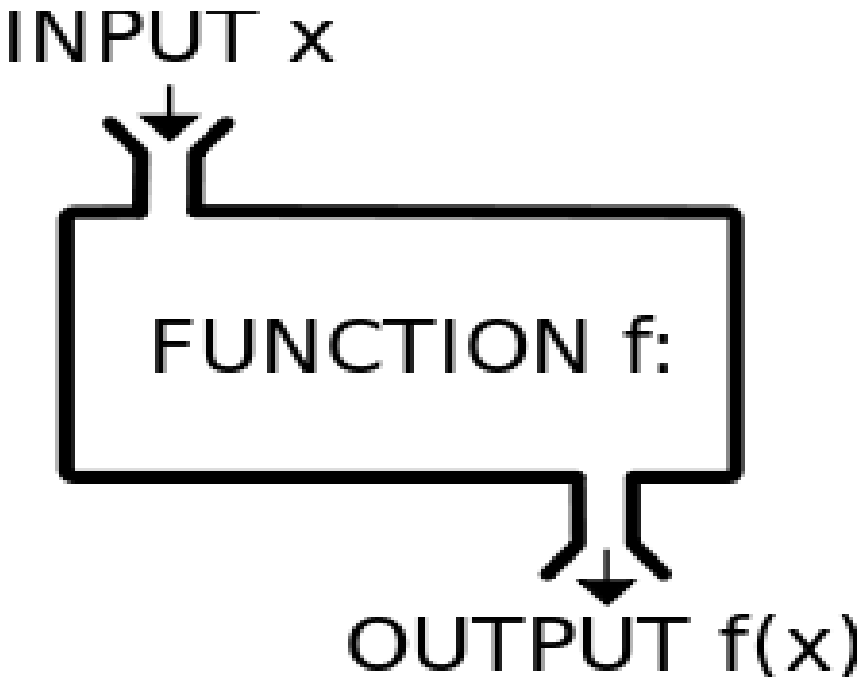
```
uint256[] arr; // Siempre es storage

function f(uint256[] memory param) {
  param[3] = 89;
}

function g(uint256[] storage param) {
  param[3] = 89;
}
```

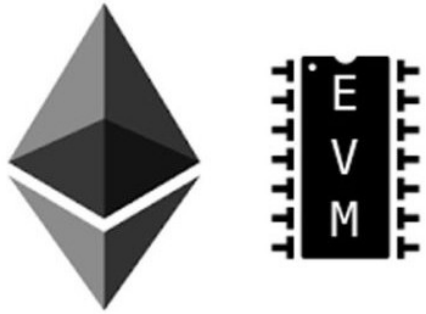
- ¿Será lo mismo llamar f(arr) que g(arr)?

Calificadores de funciones



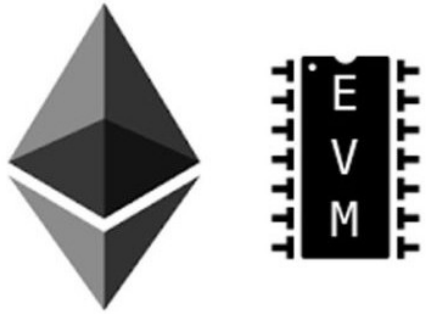
Visibilidad de funciones

- Hay cuatro (4) clases de visibilidad
  - external: Definen la interfaz pública del contrato
  - public: Similar a external. Pueden llamarse internamente
  - internal: Equivalente a protected en OOP
  - private: Equivalente a private en OOP

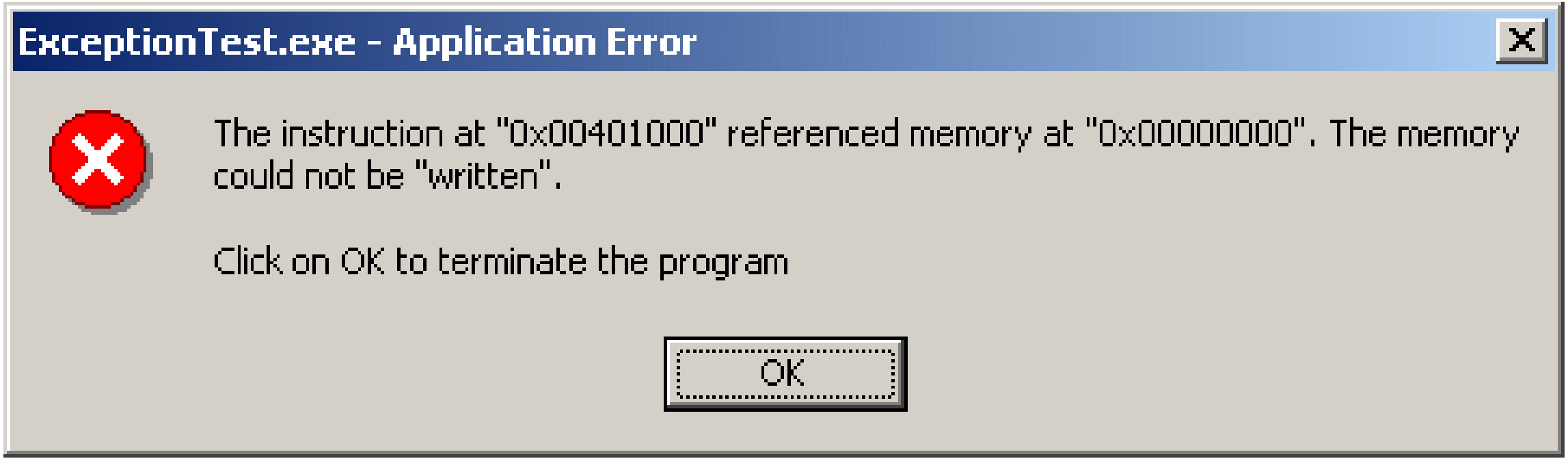


Clases de funciones

- Hay cuatro (4) clases de funciones
  - non-payable: Pueden modificar el estado sin recibir Ether
  - view: Solo pueden consultar el estado
  - pure: No pueden modificar ni consultar el estado
  - payable: Pueden modificar el estado y recibir/transferir Ether



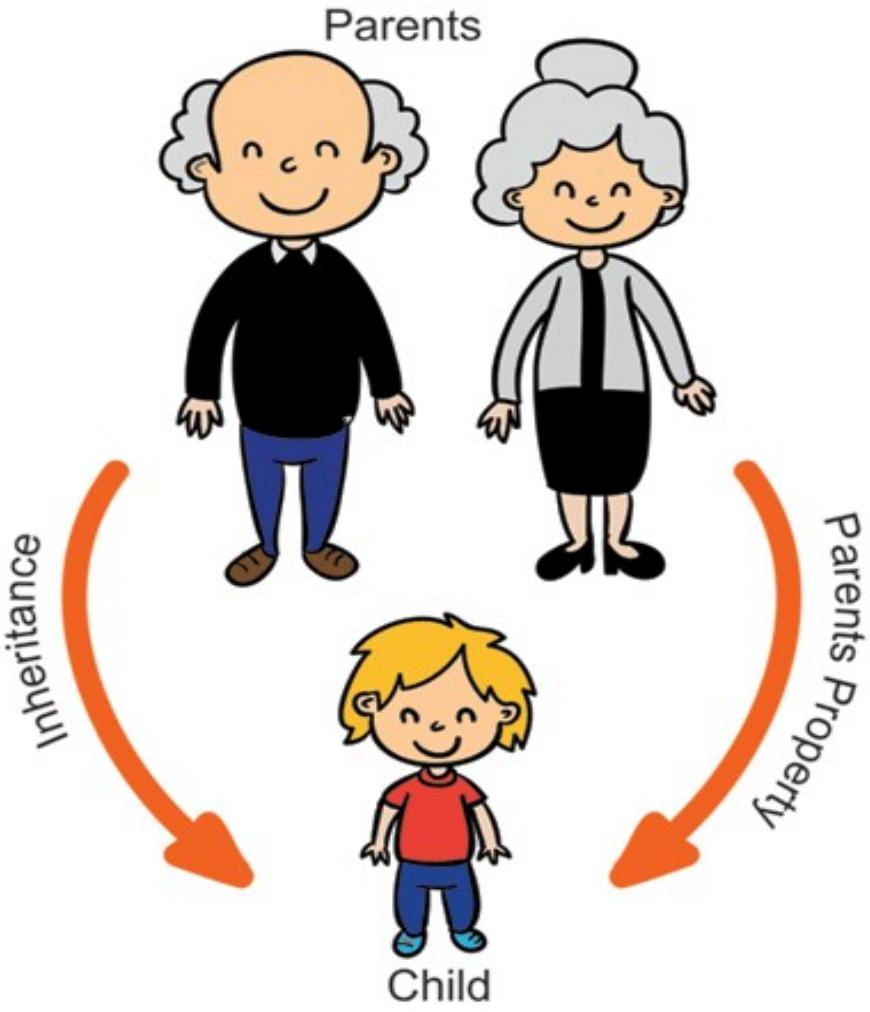
Flujo de control excepcional



Excepciones en Solidity

- Hay tres(3) funciones de generación de excepciones
  - assert(bool expression [, string message])
    - Para verificar invariantes
    - Consume el gas utilizado
  - require(bool expression [, string message])
    - Para verificar entradas y estado
    - No consume gas
  - revert([string message])

Herencia de contratos y modificadores



Herencia de contratos

```
contract A {
  uint256 internal value;
  constructor(uint256 v) {
    value = v;
  }
}

contract B is A(1989) {
  function getValue() public view returns (uint256) {
    return value;
  }
}
```

Modificadores de funciones

```
contract Ownable {
  address payable owner;

  constructor() public {
    owner = msg.sender;
  }

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

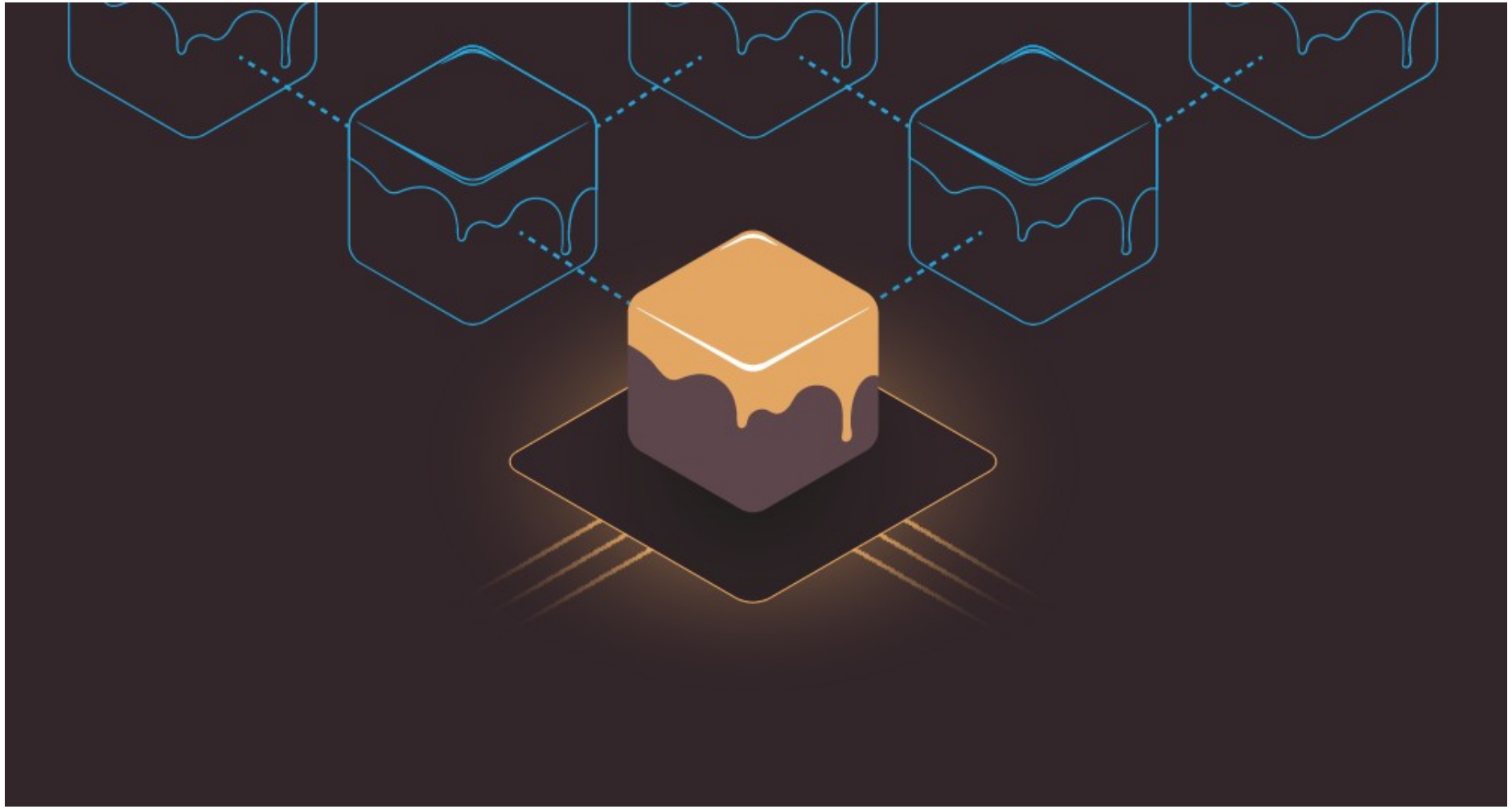
Como usar un modificador

```
pragma solidity >0.4.99 <0.6;

import “./Ownable.sol”;

contract OnlyOwnerCanKill is Ownable {
  function kill() public onlyOwner {
    selfdestruct(owner);
  }
}
```

Truffle y Ganache



Truffle

- Framework de desarrollo de aplicaciones descentralizadas (dApp’s) para Ethereum
- Basado en Node.js
- Permite programar y probar dApp’s y desplegar contratos a la Blockchain

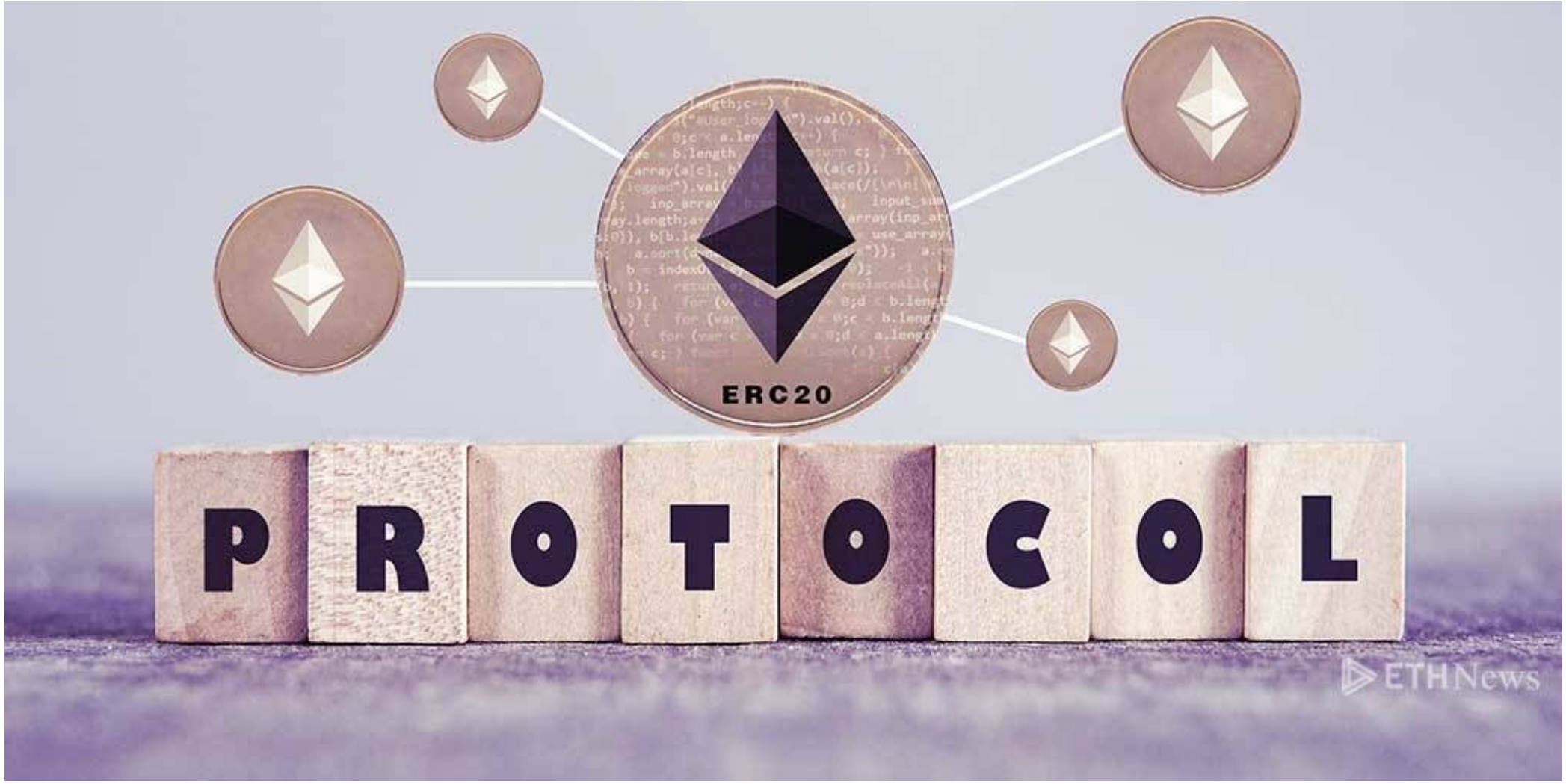
El *Framework* Truffle

- Integración con Node.js y NPM.
- Gestión automática de artefactos.
- Pruebas unitarias (Mocha.js).
- Despliegue a múltiples redes.
- Consola interactiva de depuración basada en Node.js.
- http://trufflframework.com/

Ganache

- Simulador determinista de nodos Ethereum
- Permite hacer despliegues y explorar el estado de la Blockchain sin necesidad de un nodo Ethereum real

Tokens en Ethereum



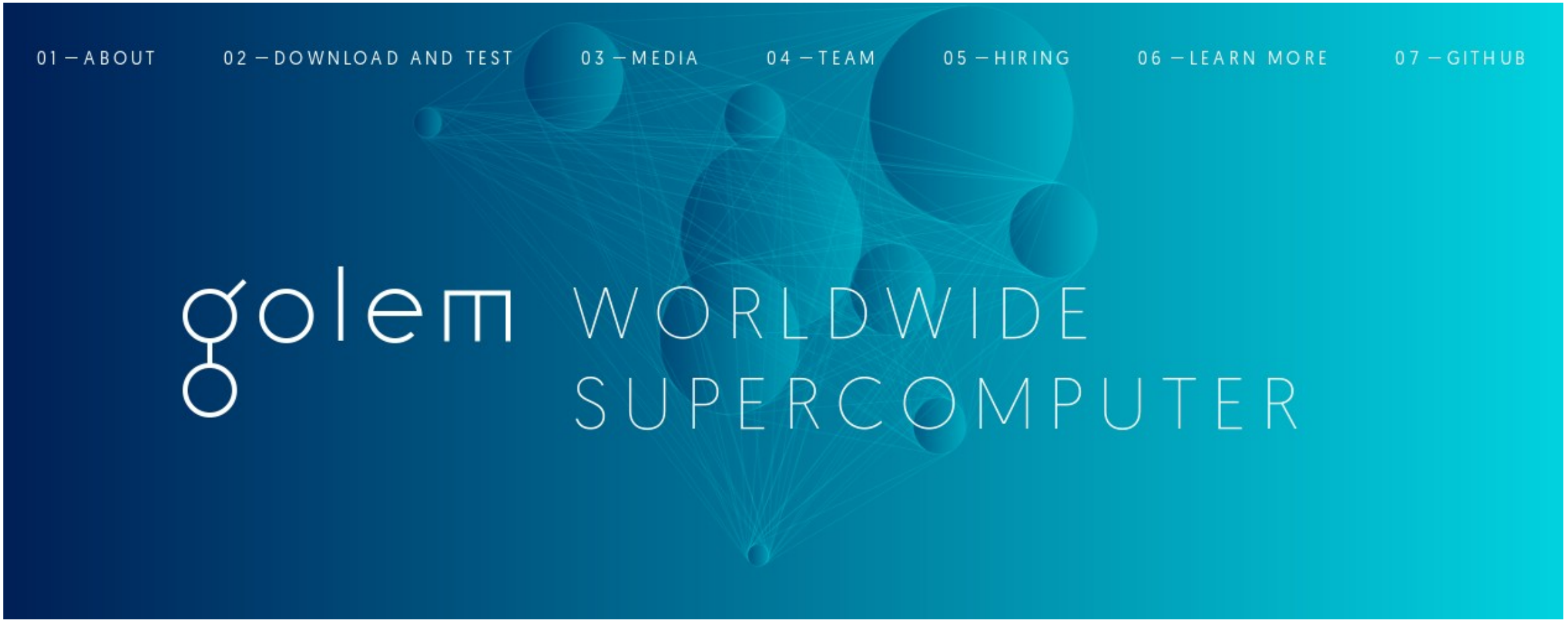
Fundamentos

- Un *token* es un contrato inteligente que representa un bien digital:
  - Siguen una interfaz estándar (pe. ERC-20).
  - Acuñables o completamente creados desde el inicio.
  - Pueden transferirse entre cuentas Ethereum.
  - Pueden ser fungibles o no.
- Un *token* **NO** es una criptomoneda.

Tokens ERC-20

- Estándar de funcionalidades básicas para tokens:
  - Fungibles.
  - Divisibles.
  - Transferibles.
  - Delegables.
  - Públicos.
- Define 9 funciones necesarias.

https://golem.network/



Tokens no fungibles ERC-720

- ERC-20 no permite representar bienes digitales que no son fungibles:
  - Por ejemplo: títulos de propiedad.
- ERC-720 define una interfaz para este tipo de *tokens*.
  - No fungibles.
  - Indivisibles.
  - Delegables.
  - Públicos.

https://www.cryptokitties.co/

What is CryptoKitties?

CryptoKitties is a game centered around breedable, collectible, and oh-so-adorable creatures we call CryptoKitties! Each cat is one-of-a-kind and 100% owned by you; it cannot be replicated, taken away, or destroyed.

https://cryptozombies.io/

Programación de *Tokens* y Contratos

- Open Zeppelin es una biblioteca que incorpora bases para la implementación de *tokens* y contratos inteligentes siguiendo buenas prácticas y patrones de diseño.
- Instalación por NPM.
- Integración directa con el *framework* Truffle.

https://openzeppelin.org/

bash

```
[~]$ npm install zeppelin-solidity
Installing [============] 100%
> zeppelin-solidity@1.2.0 install
> scripts/install.sh
[~]$ vim ExampleToken.sol
[~]$ truffle console
> var myToken = ExampleToken.deployed();
> myToken.totalSupply()
100000000000000000000
> myToken.transfer(...)
true
```

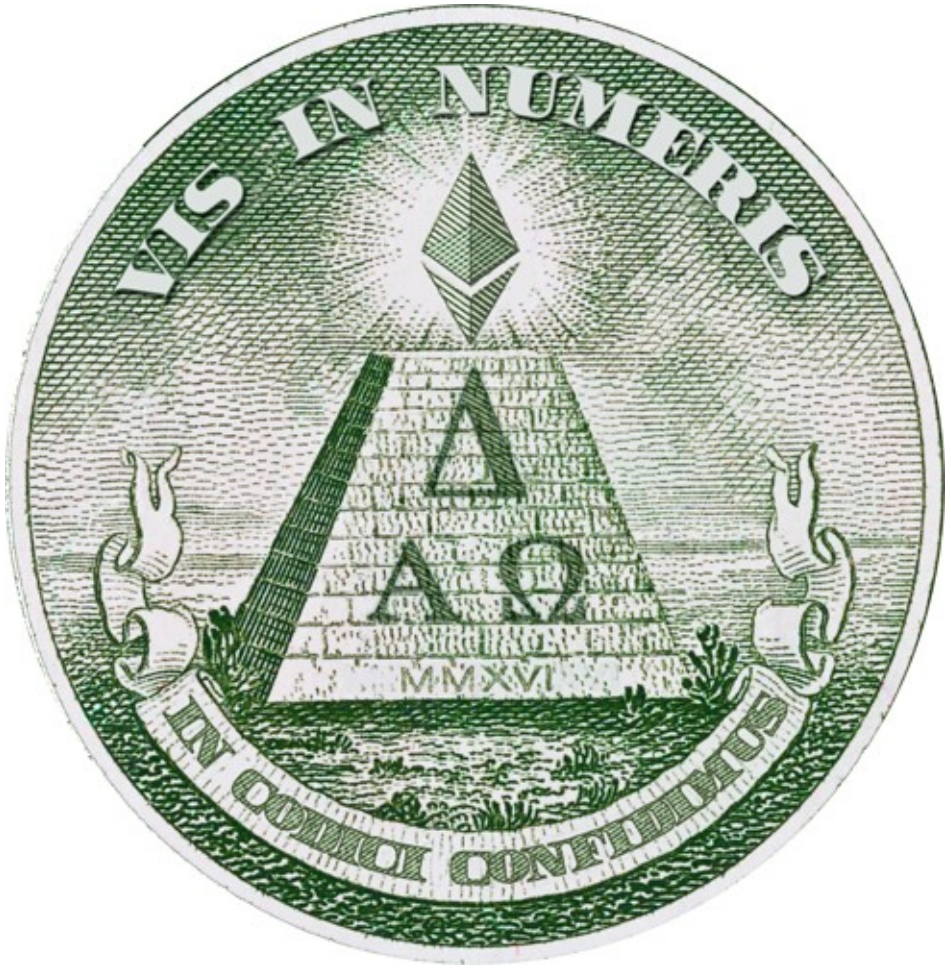
vim ExampleToken.sol

```
pragma solidity ^0.4.11;
import "zeppelin-solidity/contracts/token/StandardToken.sol";

contract ExampleToken is StandardToken {
 string public name = "ExampleToken";
 string public symbol = "EGT";
 uint public decimals = 18;
 uint public INITIAL_SUPPLY = 10000 * (10 ** decimals);

 function ExampleToken() {
  totalSupply = INITIAL_SUPPLY;
  balances[msg.sender] = INITIAL_SUPPLY;
 }
}
```

Conclusiones



Conclusiones

- Solidity tiene muchas similitudes como lenguaje a Python y C/C++
- Sin embargo, Solidity también es una criatura única en muchos aspectos
- Punto importante para recordar: la arquitectura de la EVM es MUY diferente a la arquitectura de Von Neumann típica en las computadoras modernas

Referencias

- https://github.com/ethereum/wiki/wiki
- https://solidity.readthedocs.io/en/develop/types.html
- https://truffleframework.com/truffle
- https://truffleframework.com/ganache

¿Preguntas?