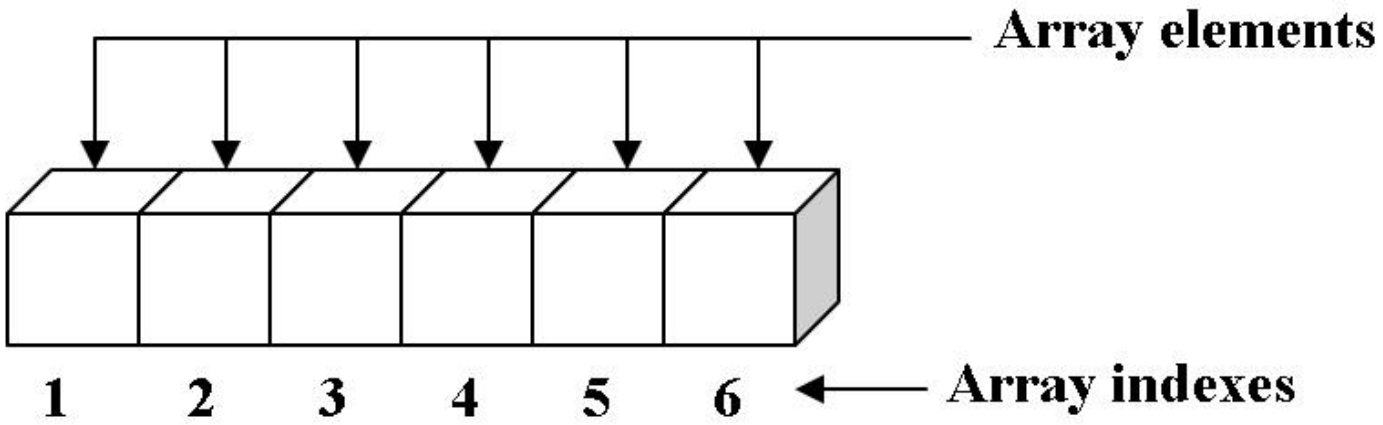
Curso desarrollo Blockchain Ethereum con Solidity

Clase 4

Arrays

Arrays

- Recordemos que un **array** es un tipo de estructura que contiene un grupo de elementos
- Los arrays en Solidity son tipos de referencia
- Existen diferentes tipos de arrays, entre los cuales se encuentran los **dynamic array** y los **fixed array**
- Son de base cero, es decir que el primer elemento se encuentra en la posición 0 y el último en la longitud menos uno



One-dimensional array with six elements

Arrays

SOLIDITY

Fixed array

En Solidity, un array fijo se declara de la siguiente manera

```
uint256[] ejemploDeArrayFijo = new uint256[](5);
```

Y los valores pueden ser asignados de la siguiente manera

```
ejemploDeArrayFijo[0] = 115;
ejemploDeArrayFijo[1] = 125;
ejemploDeArrayFijo[2] = 145;
ejemploDeArrayFijo[3] = 165;
ejemploDeArrayFijo[4] = 185;

ejemploDeArrayFijo[5] = 200;//ERROR!
```

Si el índice al que se quiere acceder está fuera del array, esto provocará un **error** que será devuelto y no continuará la ejecución del contrato

Arrays

SOLIDITY

Dynamic array

En Solidity, un array dinámico se declara de la siguiente manera

```
uint256[] ejemploDeArrayDinamico;
```

Y los valores pueden ser asignados de la siguiente manera

```
ejemploDeArrayDinamico.push(120);
ejemploDeArrayDinamico.push(1150);
ejemploDeArrayDinamico.push(50);
ejemploDeArrayDinamico.push(1900);
ejemploDeArrayDinamico.push(333);
```

Es posible recuperar la longitud del array en todo momento usando la propiedad **length**

```
ejemploDeArrayDinamico.length;
```

Validaciones (if vs required)

Validaciones (if vs required)

Imaginemos que tenemos el siguiente contrato

browser/educacionItContract.sol

```
1  pragma solidity ^0.4.17;
2  //import "./jds.sol";
3
4  contract JuegoDeApuestas {
5      address[] public apostadores;
6
7      function apostar() public payable {
8          apostadores.push(msg.sender);
9      }
10
11     function obtenerListadoDeApostadores() public view returns (address[]) {
12         return apostadores;
13     }
14 }
15
16
```

Qué sucede si no enviamos nada de ether a la función “apostar”?

Validaciones (if vs required)

SOLIDITY

Necesitaremos entonces agregar una llamada a una función global para validar que se está enviando la cantidad deseada de Ether. De manera que la función ha de quedar así

```
function apostar() public payable {
    require(msg.value >= 1 ether);
    apostadores.push(msg.sender);
}
```

Require impide que la ejecución continúe si la condición no se cumple. En este caso, cualquier intento por llamar a la función con menos de 1 ether terminará en error. Estos errores pueden ser vistos en la consola de debug.

Validaciones (if vs required)

SOLIDITY

Supongamos que usamos un if para validar que se haya enviado el ether deseado

```
function apostar() public payable {
    if(msg.value >= 1 ether) {
        apostadores.push(msg.sender);
    }
    else {
        //Qué hacemos acá?
    }
}
```

Tendríamos que controlar que sucede en los casos en que no, dado que la transacción se ejecutaría sin inconvenientes

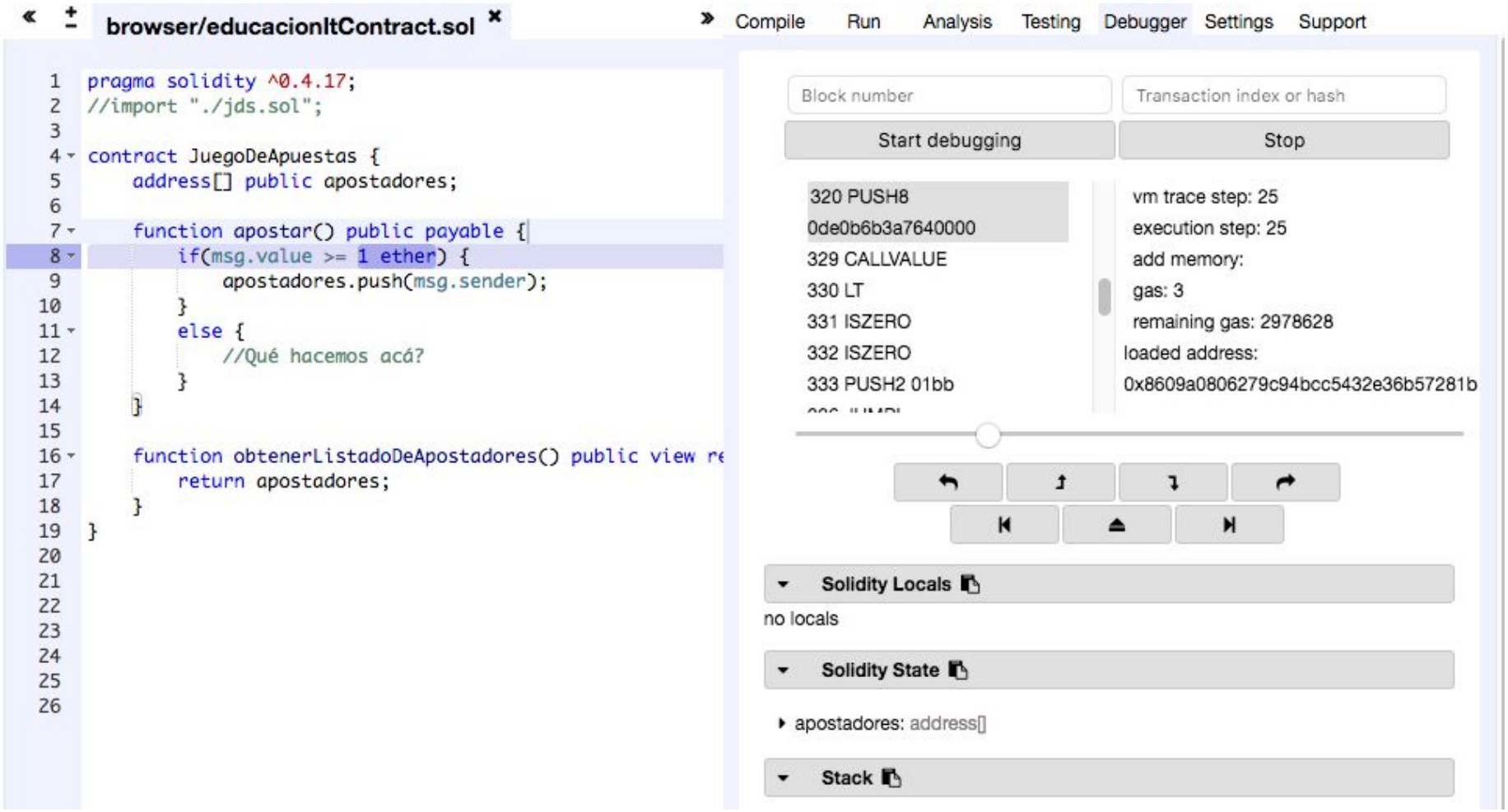
[vm] **from:**0xca3...a733c **to:**JuegoDeApuestas.apostar() 0x0c2...739ef **value:**0 wei **data:**0xc18...11269 **logs:**0 **hash:**0xfec...d4696

status	0x1 Transaction mined and execution succeed
transaction hash	0xfec154a36bf4a1dfba4803bdd2fe0e3b7b32d9cddb9a9e12f47978005a3d4696
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	JuegoDeApuestas.apostar() 0x0c2e77121daf0270d26bf0a7e9ab0faa8bf739ef
gas	3000000 gas
transaction cost	21410 gas
execution cost	138 gas
hash	0xfec154a36bf4a1dfba4803bdd2fe0e3b7b32d9cddb9a9e12f47978005a3d4696
input	0xc18...11269
decoded input	{}
decoded output	{}
logs	[]
value	0 wei

Debugging con Remix

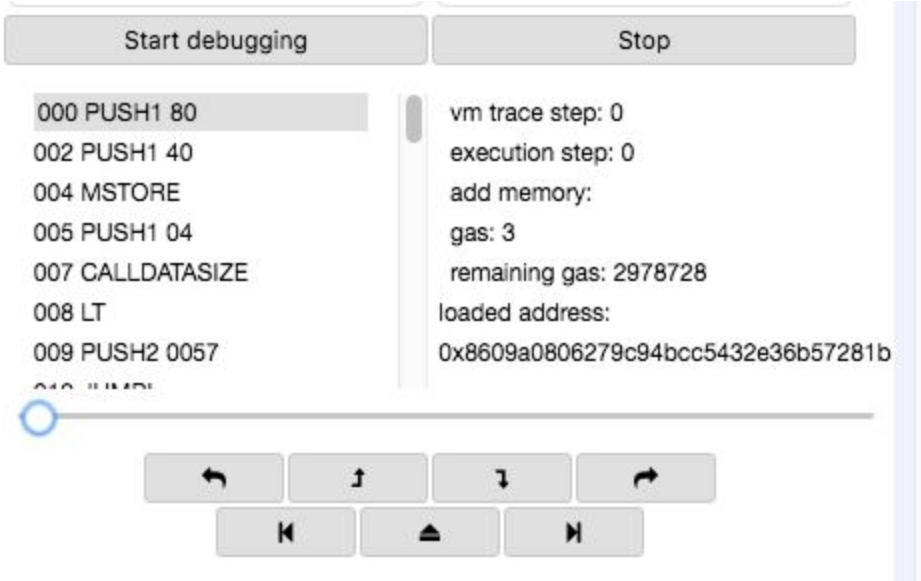
Debugging con Remix

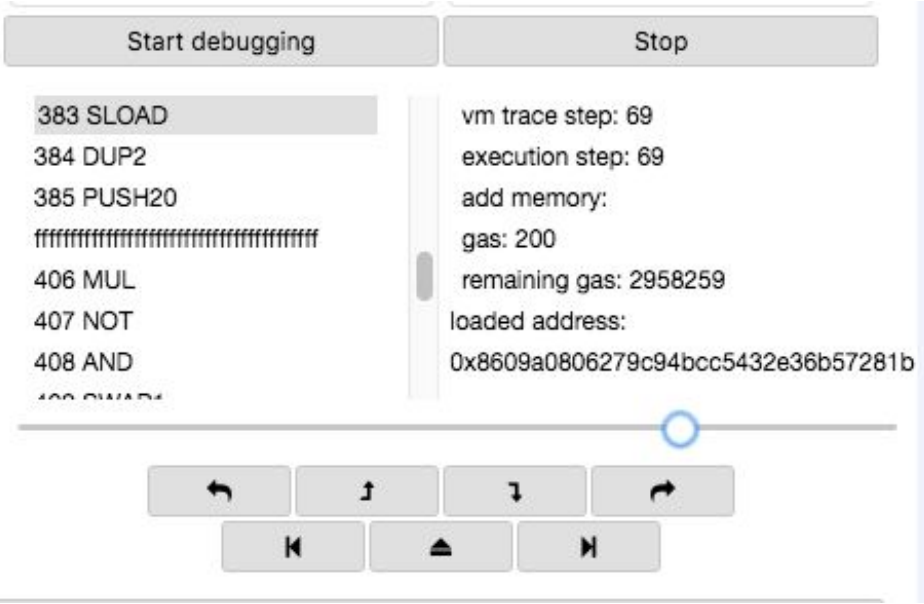
Una de las funciones más interesantes que ofrece Remix, es la de debuggear nuestros contratos



Debugging con Remix

- En el panel superior derecho de Remix, entre otras cosas, es posible ver los costes de GAS antes y despues de cada ejecución de cada línea de nuestro contrato.
- También es posible avanzar y retroceder tantas veces como se desee
- Es posible navegar dentro y fuera de una fución
- Se puede dejar de debuggear con simplemente presionar la opción de ”Stop”





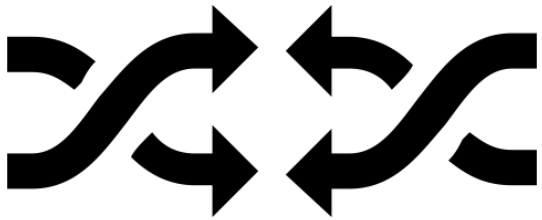
Generación aleatoria

Generación aleatoria

En Solidity, en Ethereum en realidad, dada la distribución real de la red, la no centralización y las inherentes condiciones por ser una Blockchain **NO existe** una manera real de **generar un número aleatorio**. No obstante, existen multiples maneras de simular esta tarea
De momento, utilizaremos variables del bloque, el momento actual, entre otras, para generar un número pseudo-aleatorio

```
function miFuncionRandom() private view returns (uint) {
    return uint(keccak256(block.difficulty, now, apostadores));
}
```

Keccak256 nos es provisto por la simple utilización de Solidity, y nos permite generar un hash en base a los valores pasados por parámetro, los cuales para el caso de ejemplo, son la dificultad actual del bloque, el momento actual y un array de direcciones al que llamamos apostadores.



Modificadores de función

Modificadores de función

Entre una de las buenas prácticas que se deben tener en cuenta al desarrollar contratos inteligentes es la de no repetir el código siempre que sea posible

browser/educacionItContract.sol

```
pragma solidity ^0.4.17;
//import "./jds.sol";

contract JuegoDeApuestas {
    address[] public apostadores;
    address public owner;

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

    function apostar() public payable {
        if(msg.value >= 1 ether) {
            apostadores.push(msg.sender);
        }
        else {
            //Qué hacemos acá?
        }
    }

    function elegirGanador() public view miModificadorDeFuncion {

    }

    function obtenerListadoDeApostadores() public view returns (address[]) {
        return apostadores;
    }


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

```

Modificadores de función

- Un modificador de función, en Solidity, se define con la palabra reservada **modifier**
- Puede o no recibir parámetros
- No le aplican los modificadores de accesibilidad
- Deben terminar en guión bajo, punto y coma (**_;**) para que pueda continuar la ejecución
- Puede o no tener llamadas a require
- Puede tener más de un require
- Se ejecutará **siempre antes** que la primer línea de la función sobre la que se aplique

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

Unidades y variables globales

Unidades y variables globales

Al desarrollar con Solidity para Ethereum, contamos con multiples Unidades y variables disponibles de manera global para su utilización en nuestros contratos inteligentes. Estas variables pueden ser clasificadas en:

- Unidades de tiempo
- Unidades de Ether
- Propiedades del Bloque*
- Propiedades de la Transacción*
- Funciones matemáticas / criptográficas
- Funciones de dirección (address functions)
- Funciones de contrato

Unidades y variables globales

Unidades de tiempo

Sufijos como **seconds**, **minutes**, **hours**, **days**, **weeks** y **years** utilizados después de numeros literales pueden usarse para convertir unidades de tiempo donde los segundos son la unidad de base

Tabla de equivalencias

```
1 == 1 seconds
1 minutes == 60 seconds
1 hours == 60 minutes
1 days == 24 hours
1 weeks == 7 days
1 years == 365 days
```

Unidades y variables globales

Unidades de Ether

Un numero literal puede tomar un sufijo como el **wei**, el **finney**, el **szabo** o el **ether** para convertirlo entre las subdenominaciones del Ether. Se asume que un numero sin sufijo para representar la moneda Ether está expresado en Wei

Tabla de equivalencias

1000000000000000000	Wei	Copy
1000000000000000	Kwei	Copy
1000000000000	Mwei	Copy
1000000000	Gwei	Copy
1000000	Szabo	Copy
1000	Finney	Copy
1	Ether	Copy

Unidades y variables globales

Funciones matemáticas / criptográficas

- **assert(bool condition)**: lanza excepción si la condición no está satisfecha.
- **addmod(uint x, uint y, uint k) returns (uint)**: computa (x + y) % k donde la suma se realiza con una precisión arbitraria y no se desborda en 2**256.
- **mulmod(uint x, uint y, uint k) returns (uint)**: computa (x * y) % k donde la multiplicación se realiza con una precisión arbitraria y no se desborda en 2**256.
- **keccak256(...) returns (bytes32)**: computa el hash de Ethereum-SHA-3 (Keccak-256) de la unión (compactada) de los argumentos.
- **sha3(...) returns (bytes32)**: equivalente a keccak256(). sha256(...) returns (bytes32): computa el hash de SHA-256 de la unión (compactada) de los argumentos.
- **ripemd160(...) returns (bytes20)**: computa el hash de RIPEMD-160 de la unión (compactada) de los argumentos.
- **ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)**: recupera la dirección asociada a la clave pública de la firma de tipo curva elíptica o devuelve cero si hay un error (ejemplo de uso).
- **revert()**: aborta la ejecución y revierte los cambios de estado a como estaban.

Unidades y variables globales

Funciones de dirección (address functions)

- **<address>.balance (uint256):** balance en Wei de la dirección.
- **<address>.transfer(uint256 amount):** envía el importe deseado en Wei a la dirección o lanza excepción si falla.
- **<address>.send(uint256 amount) returns (bool):** envía el importe deseado en Wei a la dirección o devuelve false si falla.
- **<address>.call(...) returns (bool):** crea una instrucción de tipo CALL a bajo nivel o devuelve false si falla.
- **<address>.callcode(...) returns (bool):** crea una instrucción de tipo CALLCODE a bajo nivel o devuelve false si falla.
- **<address>.delegatecall(...) returns (bool):** crea una instrucción de tipo DELEGATECALL a bajo nivel o devuelve false si falla.

Unidades y variables globales

Funciones de contrato

- **this (el tipo del contrato actual)**: el contrato actual, explicitamente convertible en Address.
- **selfdestruct(address recipient)**: destruye el contrato actual y envía los fondos que tiene a una dirección especificada.
- **Además**, todas las funciones del contrato actual se pueden llamar directamente, incluida la función actual.

Managers

Managers

Uno de los usos más frecuentes en un contrato inteligente hecho con Solidity, es la utilización de Managers, es decir direcciones de Ethereum que tienen más privilegios que otras. Algo similar a un administrador, owner, propietario, root, etc.

Analicemos el siguiente fragmento del contrato

```
function elegirGanador() public  {
    uint index = miFuncionRandom() % apostadores.length;
    apostadores[index].transfer(address(this).balance);
    apostadores = new address[](0);
}
```

Cualquiera podría llamar a la función "elegirGanador" y no sería correcto
Para restringirlo, se ha de utilizar un require donde solo el manager pueda invocar a la función

```
function elegirGanador() public  {
    require(msg.sender == owner);
    uint index = miFuncionRandom() % apostadores.length;
    apostadores[index].transfer(address(this).balance);
    apostadores = new address[](0);
}
```

Managers

Si bien es perfectamente válido, haciendo uso de los modificadores de función antes vistos, nuesto contrato quedaría de la siguiente manera

```
pragma solidity ^0.4.17;
//import "./jds.sol";

contract JuegoDeApuestas {
    address[] public apostadores;
    address public owner;

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

    function apostar() public payable {
        require(msg.value >= 1 ether);
        apostadores.push(msg.sender);
    }

    function elegirGanador() public unicamenteElOwner {
        uint index = miFuncionRandom() % apostadores.length;
        apostadores[index].transfer(address(this).balance);
        apostadores = new address[](0);
    }

    modifier unicamenteElOwner() {
        require(msg.sender == owner);
        _;
    }
    function miFuncionRandom() private view returns (uint) {
        return uint(keccak256(block.difficulty, now, apostadores));
    }
}
```

Tuplas

Tuplas

- Las tuplas con secuencias de valores agrupados
- Sirven para agrupar, como si fuesen un único valor, varios valores que por su naturaleza deben ir juntos
- Las tuplas son inmutables
- Una vez que son creadas no pueden ser modificadas
- Pueden ser retornadas en funciones dentro de un smartContract
- Se definen entre paréntesis

Tuplas

Analizando la siguiente función del contrato

```
function funcionQueRetornaTuplaDosValores() public pure returns(string,uint) {
    return ("un valor de texto", 1012);
}
```

Se puede apreciar que al invocarla, efectivamente se retornan dos valores y, en este caso, de distintos tipos de datos

Por supuesto, pueden haber tuplas con conjuntos de datos del mismo tipo como el caso siguiente

```
function funcionQueRetornaDosStrings() public pure returns(string,string) {
    return ("un valor de texto", "otro texto mas");
}
```

Versionado

Versionado

Al igual que cualquier otro proyecto de cualquier otro lenguaje, los contratos inteligentes desarrollados con Solidity pueden ser versionados con diferentes motores como ser GIT, TFS, SVN, etc.

La mayoría de los contratos inteligentes actuales son subidos de manera pública a Github.com dado que esto brinda mayor confianza entre aquellas personas que aceptan y/o analizan utilizarlos.
Dado que pese a que sean públicos, si se encuentran bien desarrollados la seguridad no es un problema, esta nueva modalidad cada vez se ve con mayor agrado entre desarrolladores y consumidores finales de estos productos