

Práctica 1: Introducción a la Programación de Smart Contracts en Solidity

Objetivo

Esta práctica tiene como objetivo aprender los conceptos básicos para programar contratos inteligentes (Smart contracts) en Solidity. Los contratos inteligentes son los componentes fundamentales de la capa de aplicaciones de Ethereum. Son programas informáticos almacenados en la blockchain que siguen la lógica "si esto, entonces aquello", se garantiza que se ejecutarán de acuerdo con las reglas definidas por su código y no se pueden cambiar una vez creados.

Para programar los contratos inteligentes de Ethereum, aprenderemos a utilizar directamente un entorno web que nos permitirá implementarlos, compilarlos y probarlos.

1. Entorno de desarrollo

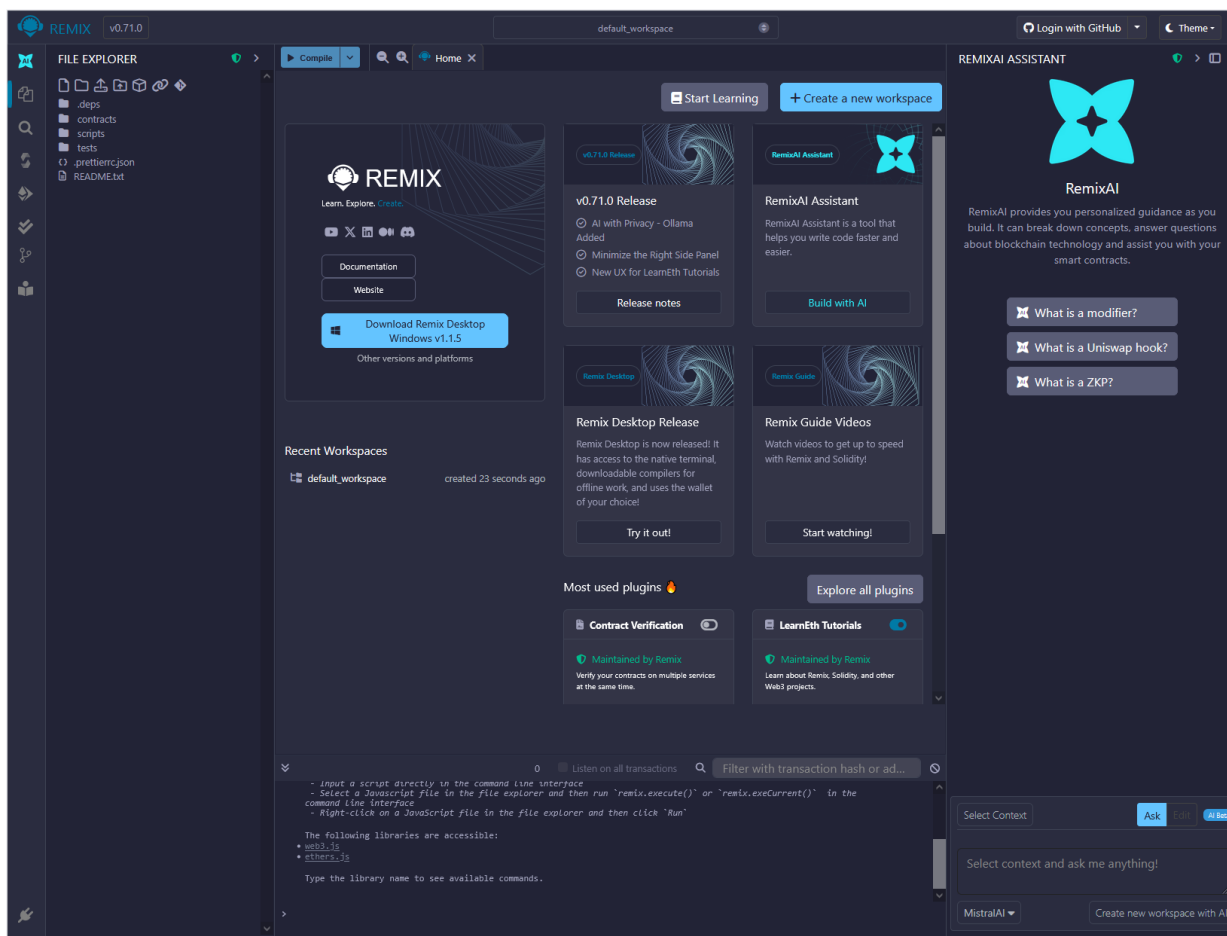
Inicialmente realizaremos toda la programación en un navegador web, por lo que no necesitaremos descargar ni instalar ninguna herramienta o software de desarrollo. Se utilizará Remix: <http://remix.ethereum.org>.

Remix IDE es un entorno de desarrollo integrado (IDE) web y de escritorio de código abierto para escribir, compilar e implementar contratos inteligentes en Solidity. Incluye una serie de funciones útiles como un compilador integrado, un entorno virtual de ejecución sin ningún componente del lado del servidor, un depurador, un entorno de prueba, una herramienta de análisis de código estático y almacenamiento permanente de archivos.

Remix IDE sirve para simular una blockchain, aloja múltiples archivos de Solidity e implementa contratos junto con una interfaz binaria de aplicación (Application Binary Interface, ABI).



Para encontrar más detalles sobre Remix IDE, puedes consultar la [documentación oficial](#).



Crearemos un nuevo fichero Solidity (.sol) sin nombre en la carpeta *contracts*. Lo nombraremos como *HelloWorld.sol*.

Escribiremos las siguientes líneas de código en el archivo *HelloWorld.sol* :

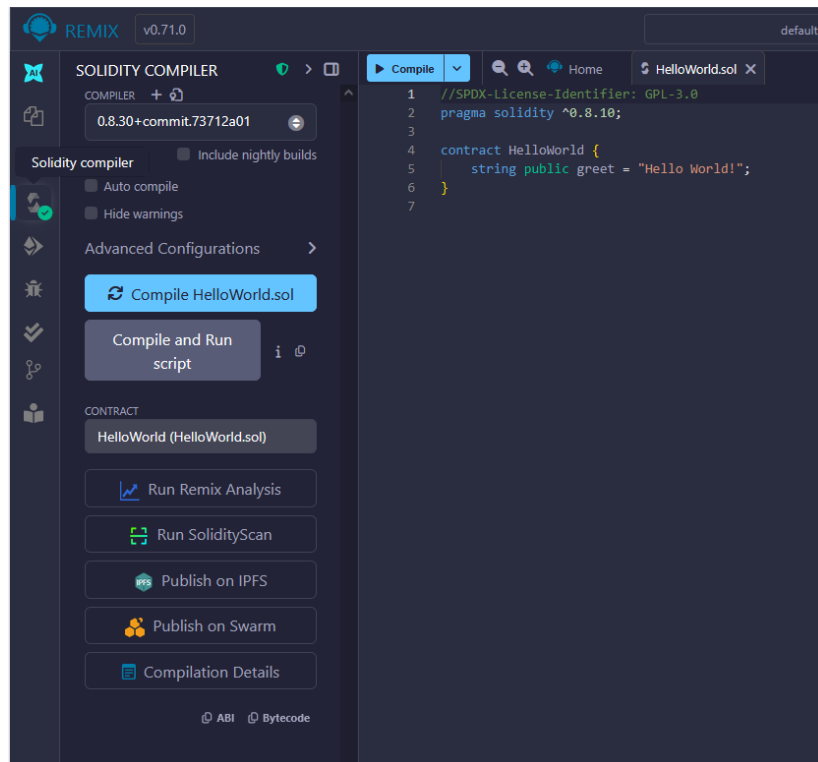
```
//SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.30;

contract HelloWorld {
    string public greet = "Hello World!";
}
```

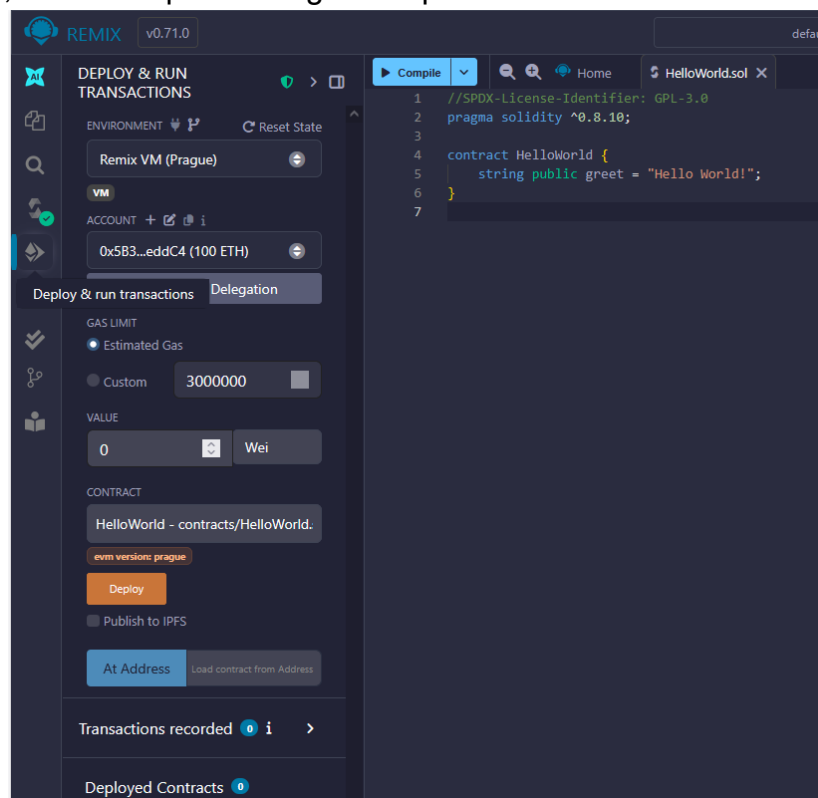
Acabamos de introducir el tipo String. Se utiliza para datos UTF-8 de longitud arbitraria.

Los siguientes pasos en Remix IDE son:

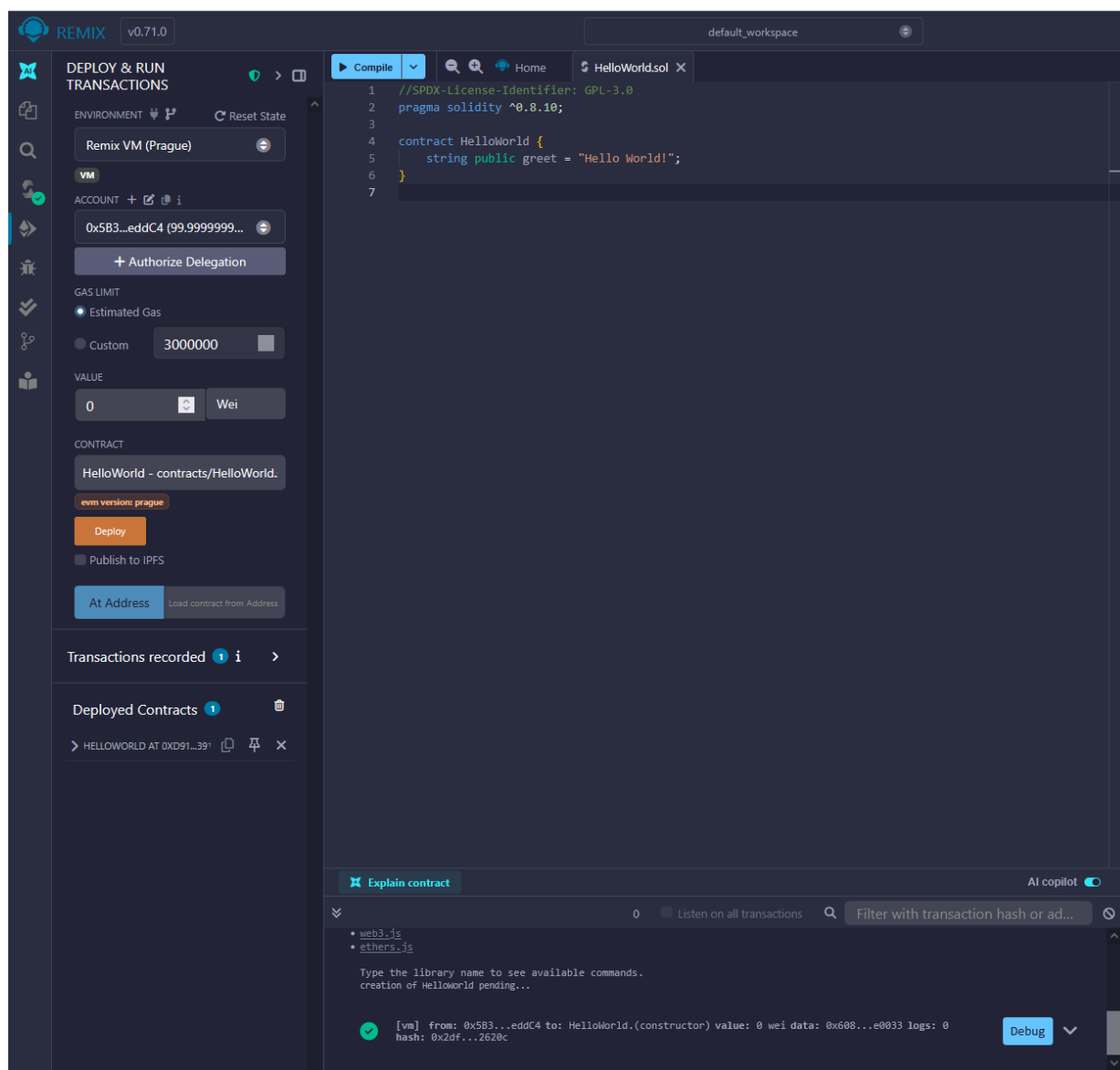
- Compilar *HelloWorld.sol*



- Deploy&Run: seleccionaremos un entorno para probar la blockchain en el navegador, de modo que no tengamos que utilizar herramientas adicionales.

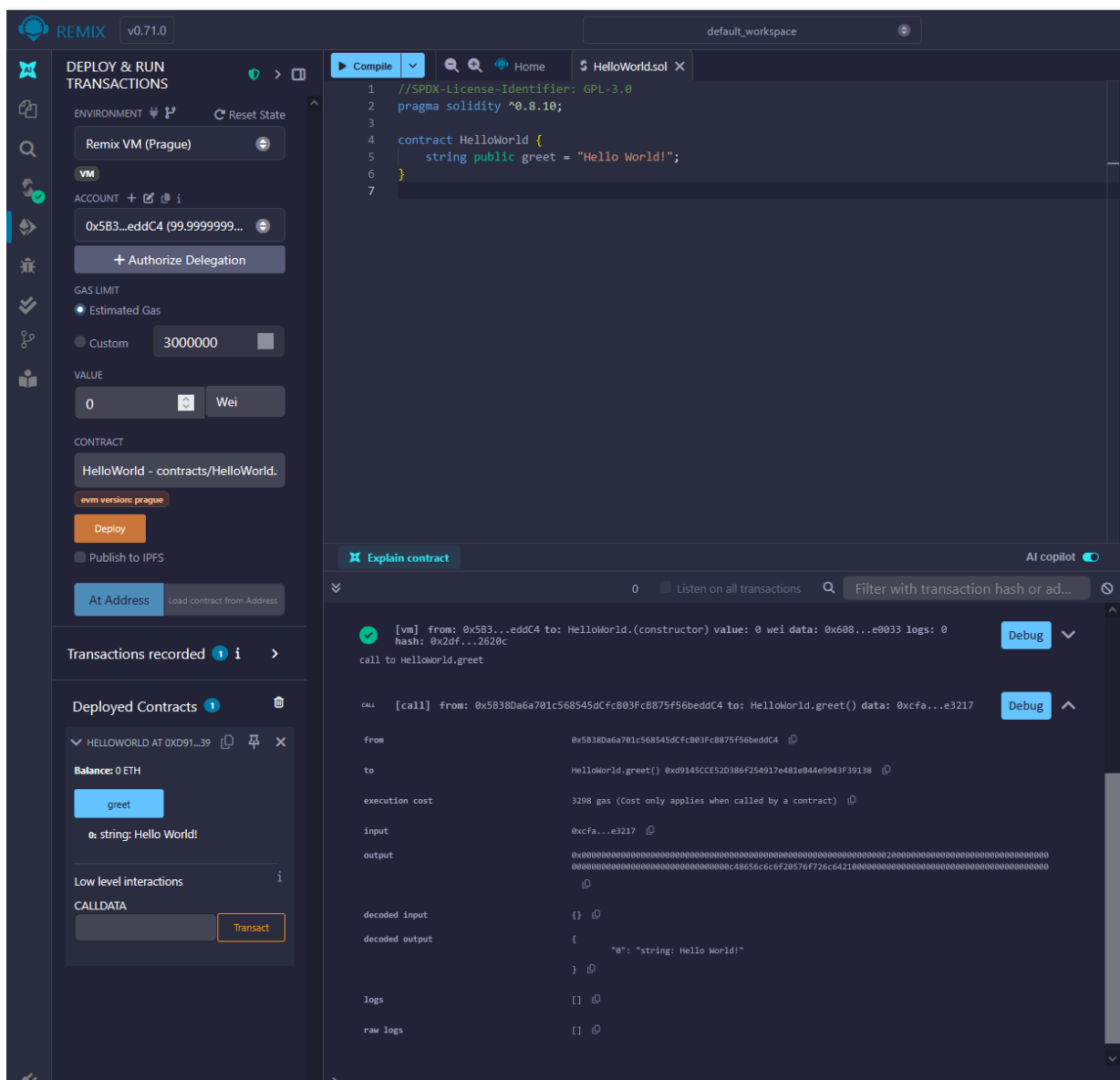


- Deploy: para poder interactuar con el smart contract en una blockchain local (por ejemplo, Remix VM, una blockchain simulada que se ejecuta en la ventana del navegador).



Acabamos de crear nuestro primer contrato inteligente. La única limitación es que se simulará en nuestra máquina local, que imita una cadena de bloques Ethereum.

Para ejecutar este contrato, podemos presionar el botón *greet* y obtendremos el resultado deseado: *Hello World !*



Solidity

Para comenzar con esta práctica de laboratorio, no se necesita ningún conocimiento previo del lenguaje.

A través de ejemplos muy simples nos familiarizaremos con la sintaxis básica necesaria para crear contratos inteligentes de [Ethereum](#) con [Solidity](#).

Solidity es un lenguaje Turing completo que se utiliza para escribir contratos inteligentes para la blockchain Ethereum (entre otras). Está influenciado por C++, Python y JavaScript. Está diseñado para ejecutarse en la máquina virtual Ethereum (EVM, Ethereum Virtual Machine), que está alojada en nodos Ethereum conectados a la blockchain.

El lenguaje está en constante mejora, actualmente en la versión v0.8.30 (Septiembre 2025).

Version Pragma

Esto es lo primero que escribiremos cada vez que iniciemos un proyecto. El código fuente debe comenzar con una primera línea de *version pragma*: una declaración de la versión del compilador Solidity que debe usar el código. Solidity es un lenguaje nuevo y cambia un poco con cada versión. Esta directiva ayuda a evitar fallos con futuras versiones del compilador que puedan introducir modificaciones que no sean compatibles. Debemos proporcionar la versión más baja del compilador que debería compilar este código. Por ejemplo, es posible que deseemos poder compilar nuestros contratos inteligentes con cualquier versión del compilador entre 0.5.0 (inclusive) y 0.8.0 (exclusivo) con el siguiente código:

```
pragma solidity >=0.5.0 <0.8.0;
```



A partir de las versiones del compilador ^0.6.8 recibirás una advertencia si no se agrega un comentario que proporcione el identificador de licencia SPDX ([SPDX license identifier](#)).

Si queremos indicar que el contrato puede ser compilado con cualquier versión a partir de una específica se agrega mediante el símbolo (^).

Podemos declarar un contrato inteligente vacío como este:

```
contract HelloWorld {  
  
}
```



Solidity admite comentarios tanto de estilo C como de estilo C++.

El compilador trata el texto entre // y el final de una línea como un comentario. Un comentario también se define por un texto entre los caracteres /* y */. Esto podría ocupar varias líneas.

2. Variables y constantes

Ahora que hemos creado nuestro contrato, aprendamos cómo Solidity trata las variables. Las variables son ubicaciones de memoria reservadas para almacenar valores.

- Las variables de estado se almacenan permanentemente en el almacenamiento del contrato y, por tanto, en la blockchain Ethereum. Consideradlos como si estuvieras escribiendo en una base de datos. La memoria asignada se asigna estáticamente y no puede cambiar durante la vigencia del contrato.
- Las variables locales se declaran dentro de una función y no se almacenan en la blockchain.
- Las variables globales proporcionan información sobre la blockchain.

Ejemplo:

```
contract MyFirstExample {  
    uint myUnsignedInteger = 10;  
}
```

En este ejemplo, creamos un *uint* (entero sin signo) llamado *myUnsignedInteger* y lo establecemos a 10. Los tipos de datos *int* se utilizan para enteros con signo.



Nótese que `uint` es en realidad un alias de `uint256`, un entero sin signo de 256 bits.

Mientras programamos, necesitaremos emplear varias variables para almacenar información diversa de varios tipos de datos, como punto flotante, doble punto flotante, booleano o dirección (*address*). En este [enlace](#), podemos consultar ejemplos adicionales de tipos de datos primitivos.

Las [constantes](#) son variables que no se pueden modificar. Su uso ahorra costes de [gas](#). Es una convención poner en mayúsculas las variables constantes.

Por ejemplo:

```
contract Constants {
    address public constant MY_ADDRESS =
0x777712359459AaAAbBbbC123ddDdeeeEFFFFCcCc;
    uint public constant MY_UINT = 123;
}
```



Tras su creación, a cada transacción se le carga una cantidad particular de [gas](#), con el objetivo de limitar la cantidad de trabajo requerido para ejecutar la transacción y al mismo tiempo pagar por su ejecución.

3. Operaciones matemáticas

Las siguientes [operaciones](#) son las mismas que en la mayoría de los lenguajes de programación:

- Suma: $x + y$
- Resta: $x - y$,
- Multiplicación: $x * y$
- División: x / y
- Módulo: $x \% y$
- Exponenciación: x Elevado a y , x^y

Ejemplo:

```
uint x = 10 ** 2;
```

En algunos casos, necesitaremos una conversión entre los diferentes tipos de datos:

```
uint8 x = 3;
uint y = 10;
// TypeError: Type uint256 is not implicitly convertible to expected
type uint8.
uint8 z = x * y;
// Para resolver el error, typecast z como uint8
uint8 z = x * uint8(y);
```

En el código anterior, $x * y$ devuelve un *uint*, pero lo almacenamos como *uint8*. Al hacer *typecasting* z como *uint8*, el compilador no arrojará un error. En Solidity podemos realizar [conversión implícita y explícita](#).

4. Tipos Struct y Enum

Tipos Struct

[Structs](#) permiten agrupar datos relacionados (una colección de variables que constan de diferentes tipos de datos).

Las estructuras pueden declararse fuera de un contrato e importarse en otro contrato.

Ejemplo:

```
struct Person {
    uint age;
    string name;
}
```

Para crear una instancia de una estructura, se utiliza la siguiente sintaxis:

```
Person satoshi = Person(39, "Satoshi");
```

Tipos Enum

[Enum](#) pueden crear tipos de datos personalizados con un conjunto de valores constantes. Se utiliza cuando a nuestras variables solo se les debe asignar un valor de un conjunto predefinido de valores.

Ejemplo:

```
contract Enum {
    // Distintos estados en un envío
    enum Status {
        Pending, // devuelve 0
        Shipped, // 1
        Accepted, // 2
        Rejected, // 3
        Canceled // 4
    }

    Status public status;

    function get() public view returns (Status) {
        return status;
    }

    // Actualizer el estado
    function set(Status _status) public {
        status = _status;
    }
}
```

```
// Actualizar a un estado específico (enum)
function cancel() public {
    status = Status.Canceled;
}
// Delete resetea el enum a su primer valor (e.g., 0)
function reset() public {
    delete status;
}
}
```

5. Arrays

Cuando deseemos una colección de valores del mismo tipo, podemos utilizar un [array](#). Los arrays facilitan la iteración, ordenación y búsqueda de elementos dentro de la estructura de datos. Hay dos tipos de arrays: fijos en tiempo de compilación (el tamaño del array debe declararse antes de compilar el contrato) y arrays dinámicos (el tamaño se puede cambiar después de que se haya compilado el contrato).

Ejemplos:

```
// Array de longitud fija de 10 elementos uint
uint[10] fixedArray;

// Array de longitud fija de 10 strings
string[10] stringArray;

// Array dinámico
uint[] dynamicArray;
```

También es posible crear un array de structs. Usando la struct *Person* de la sección anterior:

```
Person[] people; // Array dinámico
```

Arrays públicos

Cuando declaramos un array como público, Solidity crea automáticamente un método *get* para él. La sintaxis es la siguiente:

```
Person[] public people;
```

Otros contratos podrían entonces leer, pero no escribir, de este array.

También podemos añadir elementos al array:

```
// Añadir una persona al array

people.push(satoshi);

// Combinar la creación con el añadido en una línea de código

people.push(Person(29, "Vitalik"));
```

6. Funciones

Una [función](#) se define mediante la palabra clave función y su nombre, seguido del tipo y nombre del parámetro y el cuerpo de la función entre llaves. La sintaxis es de la siguiente forma:

```
function <name of the function>(<parameters>) <visibility specifier>
<state mutability modifier> returns (<return data type> <name of the
variable>)
{
  <function body>
}
```

Un ejemplo de declaración en Solidity puede ser:

```
function sellProducts(string memory _name, uint _amount)
public {

}
```

Esta función *sellProducts* toma dos parámetros: un *string* y un *uint*. La visibilidad de la función se establece como *pública*. El cuerpo de la función está vacío.

Además, se proporcionan instrucciones sobre dónde se debe almacenar la variable *_name*, en memoria. Esto es necesario para todos los argumentos pasados por referencia, como *arrays*, *structs*, *mappings* y *strings*.

Nótese que podemos pasar un argumento a una función de dos maneras diferentes:

- Por valor: el compilador crea una nueva copia del valor del parámetro y se lo pasa a la función. La palabra clave *memory* se utiliza en la declaración de variable (por ejemplo, *string memory _name*).
- Por referencia: se hace referencia a la variable original cuando se llama a la función. Como resultado, si la función modifica el valor de la variable, también cambia el valor de la variable original.



Es una convención nombrar las variables de parámetros de función con un guión bajo (_) para diferenciarlas de las variables globales.

Podemos llamar a la función anterior como:

```
sellProducts("pipe", 100);
```

Funciones públicas vs privadas

En Solidity, las funciones son públicas de forma predeterminada: cualquiera (o cualquier otro contrato) puede llamar a la función de tu contrato y ejecutarla. Esto no siempre es deseable y puede hacer que nuestro contrato sea vulnerable a ciberataques. Por lo tanto, es recomendable configurar las funciones como privadas de forma predeterminada.

Un ejemplo de declaración de una función privada puede ser:

```
uint[] numbers;  
  
function _addNumberToArray(uint _number) private {  
    numbers.push(_number);  
}
```

Tal y como está definida, solo otras funciones dentro de nuestro contrato podrán llamar y ejecutar la función `_addNumberToArray`.

Utilizamos la palabra clave *private* después del nombre de la función. Y al igual que con los parámetros de función, es una convención comenzar los nombres de funciones privadas con un guión bajo (_).

Funciones internas vs externas

Las llamadas a funciones internas se pueden llamar dentro del contexto del contrato actual. Estas llamadas dan como resultado llamadas JUMP simples en el nivel de código de bytes de EVM.

Las llamadas a funciones externas se realizan mediante llamadas de mensajes desde un contrato a otro contrato. En este caso, todos los parámetros de la función se copian en la memoria.

7. Funciones adicionales

En esta sección aprenderemos dos funciones adicionales:

Devolver variables y valores

Para devolver un valor de una función, la declaración de la función contendrá el tipo del valor de [return](#):

```
string course = "BC-MUNICS";

function sayYourCourse() public returns (string memory) {
    return course;
}
```

Modificadores

Un modificador cambia el comportamiento del código en ejecución. La función anterior *sayYourCourse* en realidad no cambia el estado en Solidity (por ejemplo, no modifica ni escribe ningún valor). Podemos utilizar diferentes modifiers en Solidity. En este ejemplo podemos declarar la función como *view*, ya que solo ve los datos, pero no los modifica:

```
function sayYourCourse() public view returns (string memory) {
    return course;
}
```

Podemos declarar funciones como *pure*: el valor de retorno depende solo de los parámetros de su función y no es necesario acceder al estado de los datos en el contrato inteligente:

```
function _IntegersMultiplied(uint x, uint y) private pure returns
(uint) {
    return x * y;
}
```

A continuación se indican los principales modificadores:

- *pure*: este modificador prohíbe el acceso o modificación del estado.
- *view*: este modificador deshabilita cualquier modificación del estado.
- *payable*: este modificador permite el pago de Ether (ETH) con una llamada.
- *virtual*: este modificador permite cambiar el comportamiento de la función o contratos derivados.
- *override*: indica que esta función, modificador o variable de estado público cambia el comportamiento de una función o modificador en un contrato base.

[Lista completa de modificadores](#)



El compilador Solidity emite advertencias para informar de cuándo debe utilizar uno de estos modificadores.

(e.g., “Warning: Function state mutability can be restricted to view” O “Warning: Function state mutability can be restricted to pure”).

8. Hashing

Una función hash actúa como una huella digital de datos y básicamente asigna una entrada a una identificación única determinista. Una ligera modificación en la entrada alterará el valor hash. Se utiliza en varios aspectos del diseño de Ethereum (e.g., en las transacciones, bloques), pero por ahora solo lo probaremos para la generación de números pseudoaleatorios.

Ethereum tiene las funciones hash: SHA-256, RIPEMD-160 y [keccak256](#) integradas. Estas funciones esperan un único parámetro de tipo bytes. Por lo tanto, tenemos que empaquetar codificando ([encodePacked](#)) los argumentos dados antes de llamarlos. Ejemplo:

```
// 7094e6f956c57cf2448641322508237390eb66a1a1526baf5938ae92a7a0a49a

(Puedes obtener el hash con un generador online con la entrada
“BC-MUNICS”).

keccak256(abi.encodePacked("BC-MUNICS"));

// 66a448fbac94ccfc911d450d0ec9add924445de6e79528e790c10695fedb4945

keccak256(abi.encodePacked("BC y DLTs MUNICS"));
```

Debemos tener en cuenta que la generación segura de números aleatorios en blockchain no es tan trivial. Aunque nuestro método no es seguro, sirve como primer ejemplo.

9. Eventos

Los [eventos](#) permiten que el contrato inteligente notifique que algo sucedió en la blockchain al front-end de su aplicación (e.g., una biblioteca de Javascript llamada Web3.js), que puede suscribirse, escuchar eventos y tomar medidas adicionales.

Ejemplo:

```
// Sentencia event para definir un evento: darle un nombre y
definir los datos que pasa cuando se activa
event IntegersAdded(uint x, uint y, uint result);
```

```
function add(uint _x, uint _y) public returns (uint) {
    uint result = _x + _y;
    //Sentencia emit para lanzar el evento y pasarle los datos

    emit IntegersAdded(_x, _y, result);
    return result;
}
```

El front-end de nuestra aplicación podría escuchar el evento:

```
YourContract.IntegersAdded(function(error, result) {
    // operar con el resultado
})
```

Podemos consultar un código de muestra en Remix IDE (*Deploy_web3.js*) para saber cómo Web3.js interactuaría con un contrato a través de [web3.eth.Contract](#).

10. Mappings

[Mappings](#) son tipos de datos complejos en Solidity. Son similares a las tablas hash y esencialmente almacenan claves asignadas a valores.

Ejemplos:

```
mapping (address => uint) public balance;
```

Este *mapping* almacena un *uint* con el saldo del usuario. El tipo de datos *address* se usa para almacenar la clave y el tipo de datos *uint* se usa para almacenar el valor. *Balance* se utiliza como identificador para este *mapping* público.

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

Esta *mapping* almacena el nombre de usuario en función de un ID de usuario (la clave es un *uint* y el valor es un *string*).

11. Variables especiales y funciones

Hay ciertas [variables especiales y funciones](#) que siempre existen y se declaran implícitamente en el espacio de nombres global. Una de estas funciones es *msg.sender* (*address*), que se refiere al remitente del mensaje que llamó/invocó la función actual. Debe considerarse que la ejecución de la función siempre debe comenzar con una llamada externa.

Este ejemplo usa *msg.sender* y actualiza un mapping:

```
contract MyFavoriteNumber {
    mapping (address => uint) favoriteNumber;

    //Cualquier podría llamar a la función setMyNumber y almacenar
    un uint en nuestro contrato, que estaría vinculado a su dirección
    function setMyNumber(uint _myNumber) public {
        favoriteNumber[msg.sender] = _myNumber;
    }

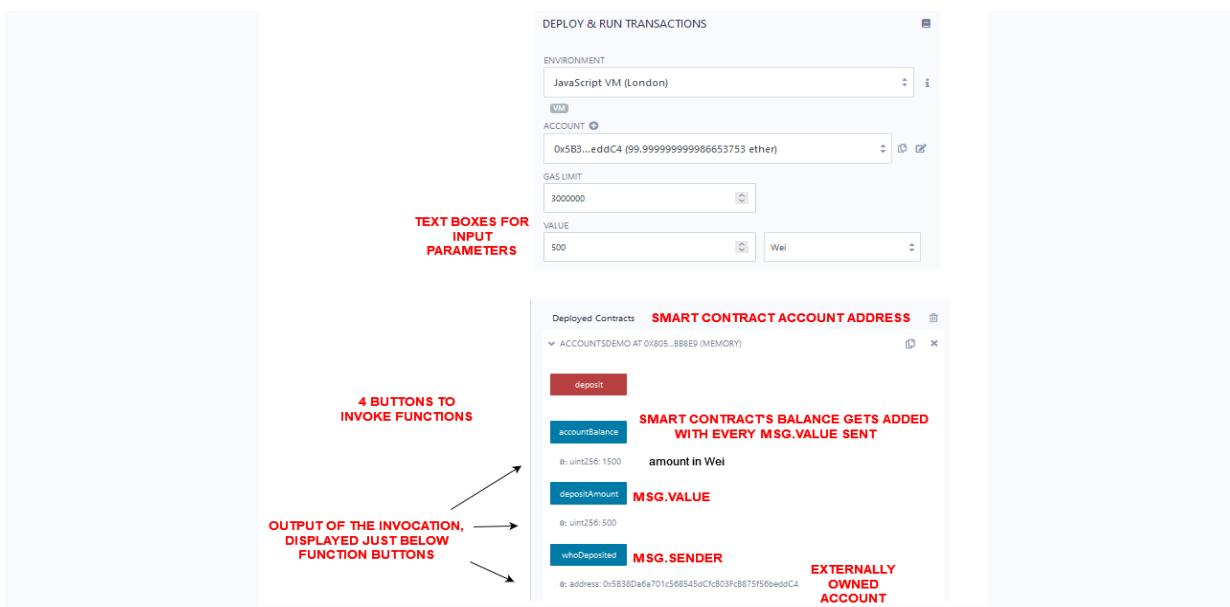
    function whatIsMyFavoriteNumber() public view returns (uint) {
        return favoriteNumber[msg.sender];
    }
}
```

Otro ejemplo:

```
pragma solidity ^0.8.30;

contract AccountsDemo {
    address public whoDeposited;
    uint public depositAmount;
    uint public accountBalance;

    function deposit() public payable { //con el botón
        deposit puedes transferir Wei
        whoDeposited = msg.sender;
        depositAmount = msg.value;
        accountBalance = address(this).balance;
    }
}
```



DEPLOY & RUN TRANSACTIONS

ENVIRONMENT
JavaScript VM (London)

ACCOUNT
0x583...eddC4 (99.9999999999996653753 ether)

GAS LIMIT
3000000

VALUE
500 Wei

TEXT BOXES FOR INPUT PARAMETERS

Deployed Contracts

ACCOUNTSDemo AT 0x805...B88E9 (MEMORY)

deposit

accountBalance

depositAmount

whoDeposited

4 BUTTONS TO INVOKE FUNCTIONS

OUTPUT OF THE INVOCATION, DISPLAYED JUST BELOW FUNCTION BUTTONS

SMART CONTRACT ACCOUNT ADDRESS

SMART CONTRACT'S BALANCE GETS ADDED WITH EVERY MSG.VALUE SENT

MSG.VALUE

MSG.SENDER

EXTERNALLY OWNED ACCOUNT



¿Cuál es el significado del color de los botones de función en Remix?

Los botones naranjas llaman a funciones que manipulan el estado del contrato, los botones azules llaman a funciones de solo lectura (por ejemplo, solo obtener información sobre el estado).



Si a es una variable, hay disponibles varios [operadores compuestos y de incremento/decremento](#) como abreviaturas: $+=$, $-=$, $*=$, $/=$, $\%=$, $|=$, $\&=$, $\^=$, $<<=$ and $>>=$

Por ejemplo:

$a++$ y $a--$ son equivalentes a $a += 1$ / $a -= 1$.

$--a$ y $++a$ tienen el mismo efecto en a pero el valor se devuelve después de la operación.

13. Estructuras de control

Las [estructuras de control](#) alteran el flujo de ejecución del contrato inteligente. Por ejemplo, *if-else* ejecuta código solo bajo ciertas circunstancias, mientras que otras declaraciones como *while* o *for* ejecutan bucles que repiten parte del código una cierta cantidad de veces.

Estructura de control	Estructura general	Ejemplo
If-else	<pre> if (esta condición/expresión es verdadera) { Ejecuta las declaraciones aquí } else if (esta condición/expresión es verdadera) { Ejecuta las declaraciones aquí } else { Ejecuta las declaraciones aquí } </pre>	<pre> contract IfElse { function LessMore(uint x) public pure returns (uint) { if (x < 10) { return 0; } else if (x < 50) { return 1; } else { return 2; } } function LessMore10(uint _x) public pure returns (uint) { // if (_x < 10) { // return 1; // } // return 2; // abreviado return _x < 10 ? 1 : 2; } } </pre>

		<pre> } } </pre>
while	<p><i>Declara e inicializa un contador</i></p> <p><i>while (comprueba el valor del contador usando una expresión o condición) {</i></p> <p><i>Ejecuta las declaraciones aquí</i></p> <p><i>Incrementa el valor del contador</i></p> <p><i>}</i></p>	<pre> contract LoopExample{ function getEvens() pure external returns(uint[] memory) { uint evenArrayLength=10; uint[] memory evenNumbers = new uint[](evenArrayLength); uint counter = 0; uint x=1; while (x <= evenArrayLength*2) { if (x % 2 == 0) { evenNumbers[counter] = x; counter++; } x++; } return evenNumbers; } } </pre>
do-while	<p><i>Declara e inicializa un contador</i></p> <p><i>do {</i></p> <p><i>Ejecuta las declaraciones aquí</i></p> <p><i>Incrementa el valor del contador</i></p> <p><i>while (comprueba el valor del contador usando una expresión o condición hasta que sea verdad) {</i></p> <p><i>}</i></p> <p>Un bucle do-while siempre se ejecuta al menos una vez.</p>	<pre> contract LoopExample{ function getEvens() pure external returns(uint[] memory) { uint evenArrayLength=10; uint[] memory evenNumbers = new uint[](evenArrayLength); uint counter = 0; uint x=1; do { if (x % 2 == 0) { evenNumbers[counter] = x; counter++; } x++; } while (x <= evenArrayLength*2); return evenNumbers; } } </pre>
for	<p><i>for (inicializa el contador del bucle; comprueba la condición del bucle hasta que se verifique; aumenta el valor del contador) {</i></p> <p><i>Ejecuta las instrucciones aquí</i></p> <p><i>}</i></p>	<pre> contract LoopExample{ function getEvens() pure external returns(uint[] memory) { uint evenArrayLength=10; uint[] memory evenNumbers = new uint[](evenArrayLength); uint counter = 0; for (uint x = 1; x <= evenArrayLength*2; x++) { if (x % 2 == 0) { evenNumbers[counter] = x; counter++; } } } } </pre>

		<pre> return evenNumbers; } } </pre>
--	--	--------------------------------------

13. Herencia

La [herencia](#) define una jerarquía entre múltiples contratos que están relacionados entre sí a través de relaciones padre-hijo. Se puede utilizar para herencia lógica (por ejemplo, subclases/clases) o para la reutilización del código, lo que permite dividir la lógica del código en múltiples contratos inteligentes.

Ejemplo:

```

contract Animal {
    function catchphrase() public returns (string memory) {
        return "Nice animal!";
    }
}

//BabyAnimal hereda de Animal: tendrá acceso a las funciones
de Animal y BabyAnimal
contract BabyAnimal is Animal {
    function anotherCatchphrase() public returns (string memory)
{
    return "BabyAnimals are so cute";
}
}

```

Otro ejemplo:

```

contract valueChecker {
    uint8 price = 20;
    event valueEvent(bool returnValue);

    function Matcher(uint8 x) public returns (bool y) {
        if (x >= price) {
            emit valueEvent(true);
            y = true;
        }
    }
}

contract valueChecker2 is valueChecker {
    function Matcher2() public view returns (uint256) {
        return price + 10;
    }
}

```

```
}

```

Cuando tengamos un código muy largo y deseemos dividirlo en varios archivos, se puede importar un archivo a otro mediante [declaraciones de importación](#).

Ejemplo:

```
import "./OtherContract.sol";
//si tenemos un fichero OtherContract.sol en el mismo
//directorio (./) que newContract, lo importará el compilador.

contract newContract is OtherContract {

}
```

Los imports se pueden usar también para reutilizar código de otras librerías.

Un [constructor](#) es una función opcional declarada con la palabra clave *constructor*, que se ejecuta al crear el contrato y donde se puede ejecutar el código de inicialización del contrato. Antes de ejecutar el código del constructor, las variables de estado se inicializan si se especifica su valor o toman un valor predeterminado, en caso contrario. Una vez se ha ejecutado el constructor, el código final del contrato se implementa en la blockchain.

Si no hay constructor, el contrato asumirá el constructor predeterminado, que equivale a *constructor() {}*.

Por ejemplo:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

abstract contract A {
    uint public a;

    constructor(uint a_) {
        a = a_;
    }
}

contract B is A(1) {
    constructor() {}
}
```

14. Gestión de errores y excepciones

Nuestro código puede contener [errores](#) (por ejemplo, desbordamiento del tipo de datos, arrays fuera de índice, no tener *gas* suficiente) que pueden pasar inadvertidos. En versiones anteriores de Solidity, solo había una función *throw()* para el manejo de errores. Tal declaración consumía todo el *gas* y revertía la transacción. Desde Solidity 0.4.10 en adelante, podemos generar un error llamando a *require*, *revert* o *assert*.

Función de gestión de errores	Funcionalidad	Ejemplo
<code>revert ()</code>	Deberíamos usar esta función para capturar condiciones esperadas que indican que una transacción debe terminarse. Todo el gas no utilizado se devuelve.	<pre> contract Account { uint public balance; uint public constant MAX_UINT = 2**256 - 1; function deposit(uint _amount) public { uint oldBalance = balance; uint newBalance = balance + _amount; // (balance + _amount) no hay // desbordamiento (overflow) si (balance + // _amount >= balance) require(newBalance >= oldBalance, "Overflow"); balance = newBalance; assert(balance >= oldBalance); } function withdraw(uint _amount) public { uint oldBalance = balance; // (balance - _amount) no hay // underflow si (balance >= _amount) require(balance >= _amount, "Underflow"); if (balance < _amount) { revert("Underflow"); } balance -= _amount; assert(balance <= oldBalance); } } </pre>
<code>assert (condición)</code>	Deshace todos los cambios de estado y no devuelve el gas no utilizado. Nunca deberíamos encontrar esta función en un código que funcione correctamente, significa que hay un <i>bug</i> .	
<code>require (condición)</code>	Valida la condición pasada como parámetro y revierte la transacción si la comprobación falla.	

Se puede utilizar una declaración [try/catch](#) para detectar un error en una llamada externa. Si la función externa funciona sin ningún error, se ejecuta el bloque *try* y se ignora el bloque *catch*. Si la llamada a la función falla, en su lugar se ejecuta el bloque *catch*.

Prácticas

Ejercicio 1 [0.05 puntos]:

Crearemos un nuevo contrato inteligente en Remix IDE: nombre .TokenContract.sol. Se procederá a pegar el siguiente código fuente.

```
// SPDX-License-Identifier: Unlicensed
pragma solidity 0.8.30;
contract TokenContract {

    address public owner;
    struct Receivers {
        string name;
        uint256 tokens;
    }

    mapping(address => Receivers) public users;

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

    constructor(){
        owner = msg.sender;
        users[owner].tokens = 100;
    }

    function double(uint _value) public pure returns (uint){
        return _value*2;
    }

    function register(string memory _name) public{
        users[msg.sender].name = _name;
    }

    function giveToken(address _receiver, uint256 _amount) onlyOwner public{
        require(users[owner].tokens >= _amount);
        users[owner].tokens -= _amount;
        users[_receiver].tokens += _amount;
    }
}
```

Si la compilación funciona sin errores, podemos ver en la ventana de la consola que el contrato ha sido desplegado y podemos interactuar con él.

Finalizaremos la implementación del contrato para que cualquier token pueda ser comprado con Ether, siempre que el propietario todavía tenga suficientes tokens. 1 token debería costar 5 Ether después de la nueva implementación.

Si la cantidad de Ether enviada no es suficiente o el propietario no tiene suficientes tokens, se debe enviar un mensaje de error. Mostraremos la cantidad de Ether que hay en el contrato inteligente. Probaremos de nuevo el contrato y cuando se finalice su implementación se subirá a una carpeta de GitHub/Gitlab.

Ejercicio 2 [0.05 puntos]:

MetaMask actúa como una cartera que permite a las aplicaciones web comunicarse de manera fácil con la blockchain de Ethereum. Instalaremos la extensión del navegador MetaMask en vuestro navegador preferido: <https://metamask.io>. También está disponible para iOS y Android.

Activemos "Mostrar redes de prueba" (testnets) en *Select a network*.

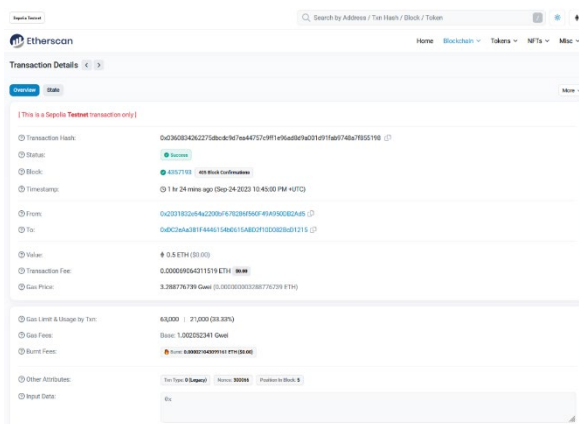
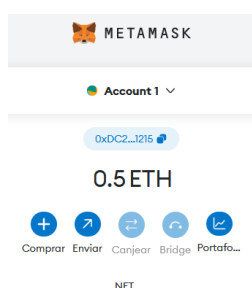
Para conectar MetaMask a una testnet como Sepolia, haremos clic en "Ethereum Mainnet" en la esquina superior izquierda de MetaMask y seleccionaremos "Sepolia" en el menú desplegable.

Para ver las funcionalidades de MetaMask podemos hacer clic en los tres puntos en la esquina superior derecha.

Para obtener ether podemos usar un faucet, una página web o aplicación sencilla mediante la cual puedes conseguir criptomoneda gratis o con poco esfuerzo (e.g., a cambio de realizar tareas muy pequeñas como resolver tests Captcha o ver anuncios). Nacieron para promover el conocimiento y la adopción de las criptomonedas.

Podemos usar los siguientes ejemplos (requieren signup/login):

- <https://www.alchemy.com/faucets/ethereum-sepolia> (requiere crear cuenta en Alchemy).
- <https://cloud.google.com/application/web3/faucet/ethereum>



El objetivo de este ejercicio es:



E3.1

Familiarizarse con las distintas funcionalidades de MetaMask.



E3.2

Obtener ETH de prueba de Sepolia.



E3.3

Enviar algo de Sepolia ETH a tus compañeros de clase.



E3.4

Realizar un seguimiento de las transacciones en Sepolia Blockchain Explorer: <https://sepolia.etherscan.io/>

Las capturas de las transacciones se incluirán en vuestra carpeta de GitHub/Gitlab.

Ejercicio 3 [0.15 puntos]:



Ethereum recomienda múltiples herramientas para experimentar un aprendizaje interactivo. Entre las opciones propuestas destaca CryptoZombies. CryptoZombies es una escuela de programación interactiva creada por el equipo de Loom Network que enseña a crear contratos inteligentes en Solidity mientras construyes tu propio juego cripto-coleccionable. Mediante lecciones de programación interactivas se profundizará en algunos conceptos de Solidity.

Completa *Solidity: Beginner to Intermediate Smart Contracts* y una vez termines sube el código resultante a tu carpeta de GitHub/Gitlab.

Ejercicio 4 [0.05 puntos]:

OpenZeppelin proporciona un conjunto completo de herramientas para desarrollar proyectos de Ethereum. En concreto, puede ser muy útil la biblioteca OpenZeppelin Solidity, donde se pueden ver múltiples ejemplos de código de contratos inteligentes. Escoged y probad uno de los contratos. Guárdalo en un nuevo archivo *.sol y súbelo a tu carpeta de Github/Gitlab.

Recordad que siempre se puede consultar la [documentación de Solidity](#) para obtener explicaciones adicionales y múltiples [ejemplos](#) de código fuente.

Ejercicio 5 (2 personas) [0.2 puntos]:

El objetivo de este ejercicio es crear tu propio contrato inteligente. Se recomiendan los siguientes pasos:

- (1) Análisis y definición del escenario, debe requerir la funcionalidad proporcionada por una blockchain pública como Ethereum.
- (2) Diseño
 - Definir un caso de uso (lógica de negocio, automatización de transacciones de determinados aspectos).
 - Definir el contenido del contrato inteligente, específicamente sus datos, funciones que operan sobre los datos, reglas de operación, entre otros.
 - Definir los usuarios del sistema (e.g., diagrama de casos de uso).
- (3) Implementación
 - Programar el código del contrato inteligente utilizando el conocimiento adquirido en los ejercicios previos.
- (4) Pruebas
 - Realizar pruebas del contrato inteligente en Remix IDE.
 - (Opcional) Las pruebas de software incluyen pruebas unitarias, pruebas de integración o pruebas a nivel de sistema. Además, un entorno de desarrollo de blockchain profesional incluirá el uso de herramientas adicionales: Web3.js en Javascript, [Truffle](#), clientes de Ethereum como [geth](#).

Documentación: Prepara un documento que explique cómo se han realizado los pasos anteriores.

Una vez se finalice el código del contrato inteligente, debe subirse a la carpeta de GitHub/Gitlab junto con el archivo de documentación.



Una vez finalizados los ejercicios envía un correo electrónico de confirmación al profesor de prácticas de la asignatura con acceso también al repositorio GitHub/Gitlab.

Fecha límite para la práctica 1: 21/10/2025 a las 23:55.

Se habilitará también una tarea en Moovi para subir la documentación asociada a los ejercicios de la práctica 1.