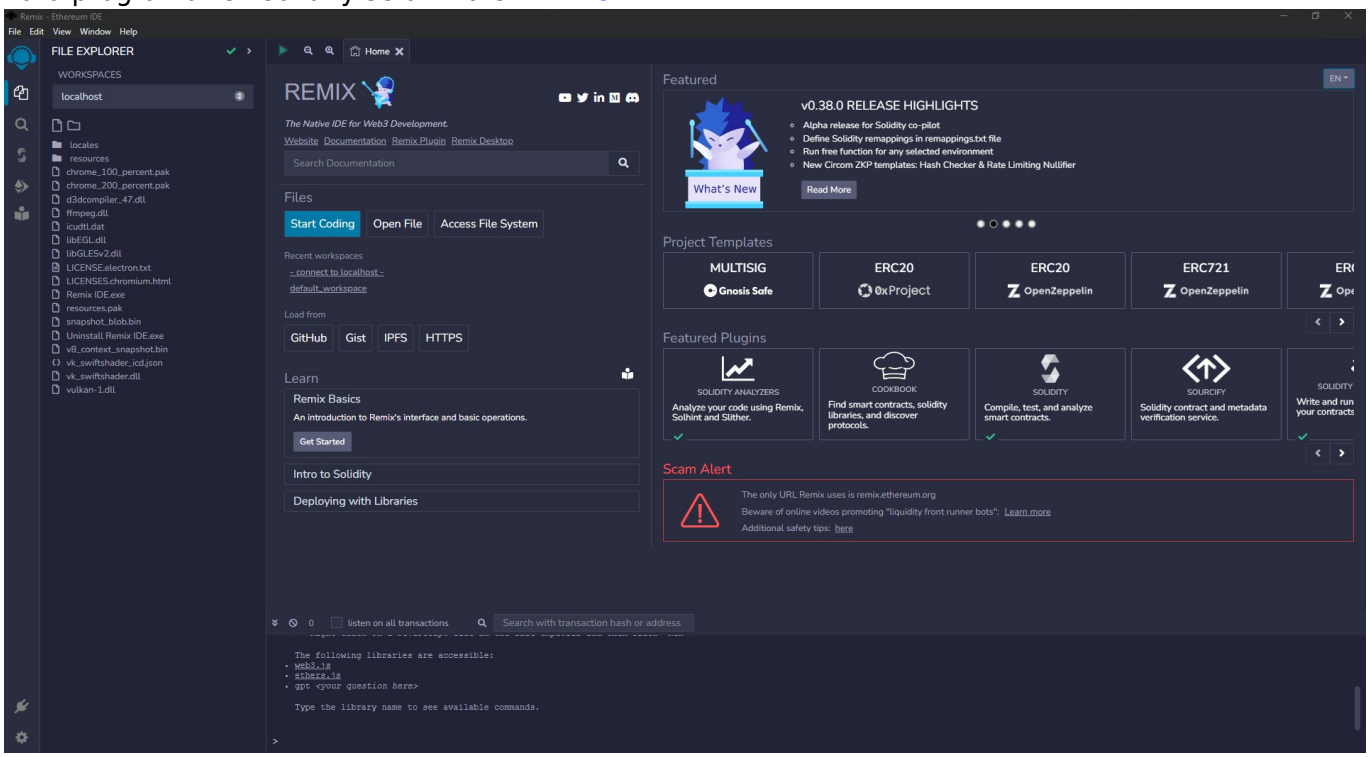
Introducción a la Programación en Solidity

Para programar en solidity se utiliza el IDE Remix:



componentes de un Smart Contract

Pragma

Lo primero que se escribe en un contrato inteligente es la versión pragma, que indica la versión del compilador que debe usar el código. Generalmente se debe poner un rango de versiones que sean compatibles con el código, por ejemplo, si quisiéramos que el código fuera compilable por las versiones entre la 0.6.12 y la 0.9.0 escribiríamos:

```
pragma solidity >=0.6.12 <0.9.0;
```

Para crear un contrato vacío, a continuación del pragma se puede introducir lo siguiente:

```
contract NombreDelContrato{

}
```

Variables y constantes

Las variables de estado se almacenan permanentemente en el almacenamiento de un contrato. Se podría decir que es como si se escribiera en una base de datos. La asignación de memoria es estática y no se puede cambiar. Las variables locales deben ser declaradas dentro de una función y no se

almacenará en la blockchain, mientras que las variables globales proporcionan información sobre la blockchain.

```
contract variable {
  uint IntSinSigno = 10; //Variable de tipo Unisgned integer
}
```

Las constantes son variables que no se pueden modificar, se usan para ahorrar costes de gas (Gas es una cantidad que se cobra por cada transacción.) Las constantes se suelen poner en mayusculas por convención para diferenciarlas de las variables.

```
contract Constante {
  address public constant MY_ADDRESS = 0x29384093845093ifSDOAsdjas;
  uint public constant MY_UINT = 325;
}
```

Operaciones matemáticas

Se pueden realizar más o menos las mismas operaciones que en otros lenguajes de programación:

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

En algunas ocasiones es necesario una conversión entre tipos de datos:

```
uint8 x =1;
uint y = 2;

uint8 z = x* uint8(y) //Se convierte y al mismo tipo que X para la operación
```

Estructuras de datos

En Solidity, al igual que en C, tenemos el tipo struct que se puede utilizar para agrupar elementos relacionados, estas estructuras pueden ser declaradas de la siguiente forma:

```
struct Persona {
  uint edad;
  uint nombre;
}
```

Para crear una instancia de esta estructura se hace lo siguiente:

```
Persona manuel = Persona(25, "Manuel");
```

Enums

Permite la creación de un tipo de datos personalizado con un conjunto de valores constantes de la siguiente manera:

```
contract Enum{
  enum Semana{
    Lunes, //devuelve 0
    Martes, //devuelve 1
    Miércoles, //devuelve 2
    Jueves, //devuelve 3
    Viernes, //devuelve 4
    Sábado, //devuelve 5
    Domingo //devuelve 6
  }

  Semana public semana;

  //función para obtener el valor del enum
  function get() pulic view returns (Semana){
    return semana;
  }

  //función para modificar el enum
  function set(Semana _semana) public{
    semana = _semana;
  }

  //actualizar a un día de la semana específico
  function domingo() public{
    semana = Semana.domingo
  }

  //devolver al primer valor
  function reset() public{
    delete semana
  }

}
```

Arrays

Permiten almacenar una colección de elementos del mismo tipo, facilita su ordenación, iteración y búsqueda. Existen los siguientes tipos:

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

//Array de longitud fija de 10 elementos de tipo string
```

```
sting[10] ArrayString;

//array dinámico
uint[] ArrayDinamico;

//array dinámico de structs
Persona[] personas;
```

Para manejar Array dinámicos disponemos de las siguientes operaciones:

```
//añadir struct de tipo persona al array
personas.push(manuel);

//crear e insertar un objeto struct al array
personas.push(Persona(34, "Alberto"));
```

Funciones

Las funciones nos permiten modularizar y optimizar el código creando pequeñas funcionalidades personalizadas. En Solidity un ejemplo de función sería el siguiente:

```
//function <nombre de la función>(<variables de entrada>) <visibilidad>
function nombreFuncion(String memory _nombre, uint _cantidad) public{
    //contenido de la función
  }

//llamada a la función
nombreFuncion("nombre", 123);
```

Las funciones son siempre públicas de forma predeterminadas lo cual no es muy seguro contra ataques, por lo que se suele recomendar marcarlas como “private”. Las funciones pueden contener alguno de los siguientes modificadores:

- pure: prohíbe el acceso o modificación del estado
- view: deshabilita cualquier modificación de estado
- payable: permite el pago de Ether (ETH) con una llamada
- virtual: este modificador permite cambiar el comportamiento de la función o contratos derivados
- override: Esta función cambia el comportamiento de otra función o contrato.

Funciones útiles

Devolver variables y valores

```
String dato = "dato"
function decirDato() public returns (string memory){
  return dato;
```

```
}
```

Modificadores

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

Hashing

Una función hash asigna una entrada a una identificación única determinista. cualquier modificación en dicha entrada modificará el valor hash. Sirve para generar números pseudoaleatorios. Ethereum tiene las siguientes funciones hash:

- SHA-256
- RIPEMD-160
- keccak256

Se puede llamar a estas funciones hash de la siguiente forma:

```
  keccak256(abi.encodePacked("nombre"));
```

Evetos

Permiten que el Smart Contract reporte que algo ha sucedido en la blockchain al front end de su aplicación. Un evento se puede implementar de la siguiente forma:

```
event IntegerAdded(uint x, uint y, uint result);
function add(uint _x, uint _y) public returns (uint){
  uint result = _x + _y;
  emit IntegersAdded(_x, _y, result);
  return result;
}
```

Mapas

Tipos de datos complejos similares a las hashtables. En el siguiente ejemplo se puede ver como se alamcena un uint con el saldo de un usuario. El tipo de datos address almacena la clave y uint el valor.

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

Variables especiales y funciones

msg.sender(address): se refiere al remitente del mensaje que invocó la función actual. Se puede utilizar para actualizar un mapping:

```
contract NumeroFavorito {
  mapping(address=>uint) numeroFavorito;
  function establecerNumero(uint _numero) public{
    numeroFavorito[msg.sender] = _myNumber;
  }
  function CualEsMiNumeroFavorito() public view returns (uint){
    return numeroFavorito[msg.sender];
  }
}
```

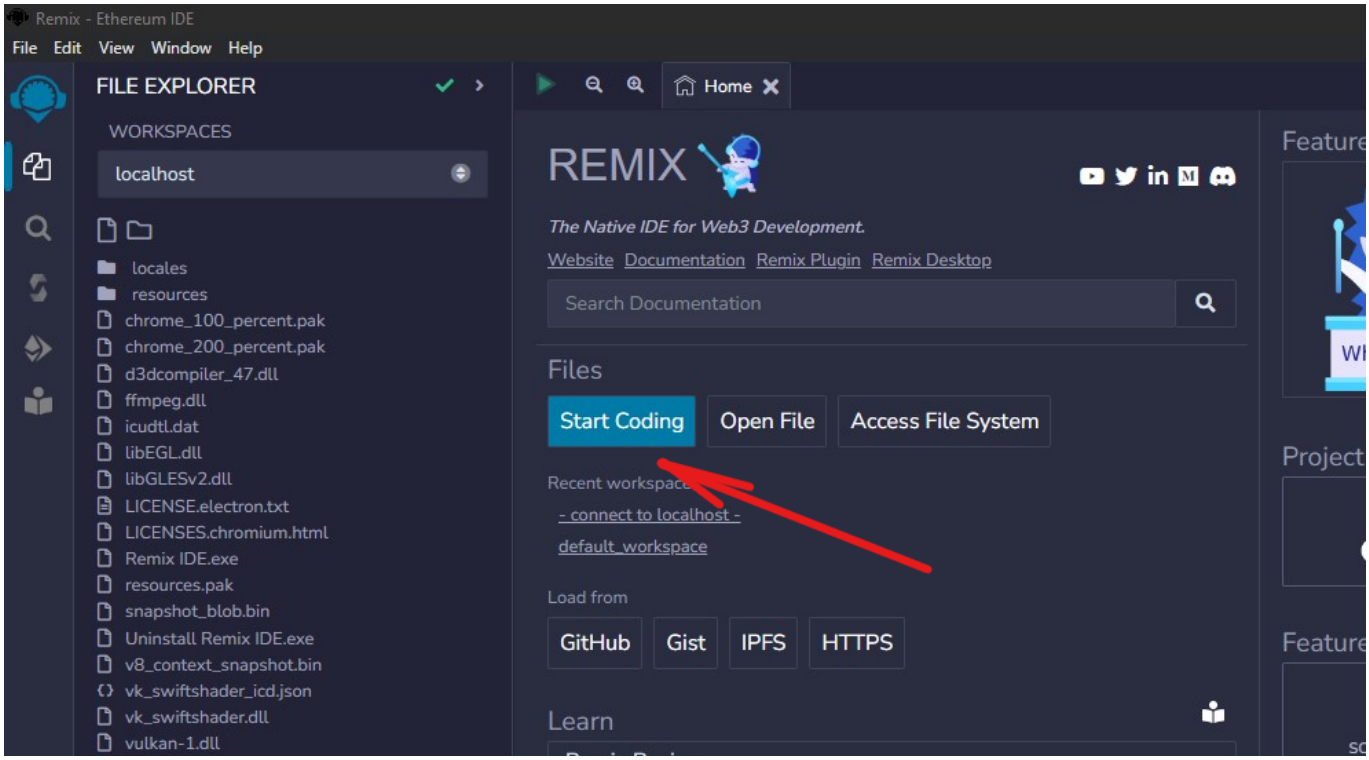
Estructuras de control

- if-else
- while
- do-while
- for

```
if(x<1){
  return 0;
} else if (x=0){
  return 1;
} else{
  return 2;
}
```

Hola Mundo en un Smart Contract

Comenzaremos creando un Smart Contract de prueba con el típico “Hello World”, para ello pulsaremos en Start Coding:



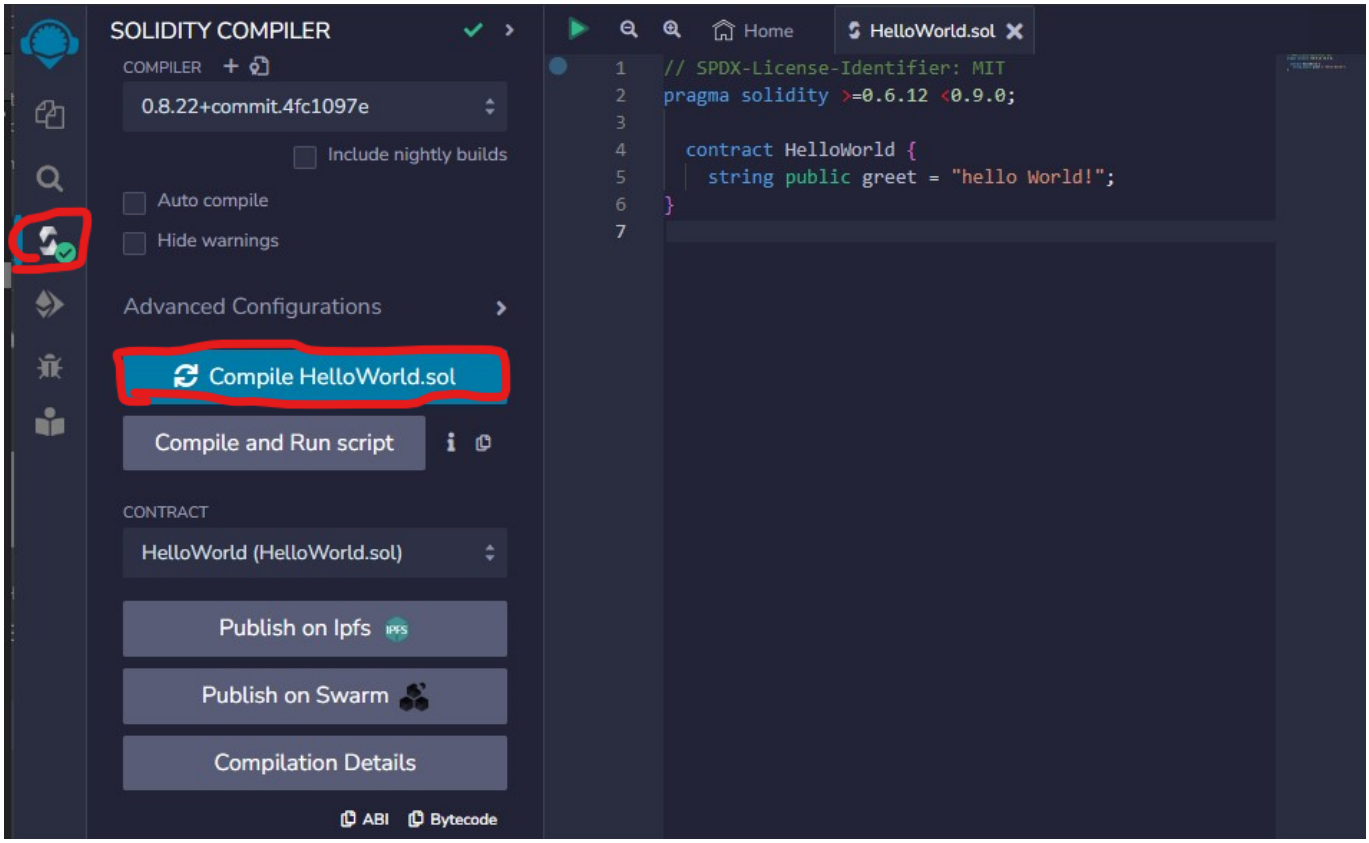
Para hacer un “Hola Mundo” escribimos el siguiente código:

HolaMundo.sol

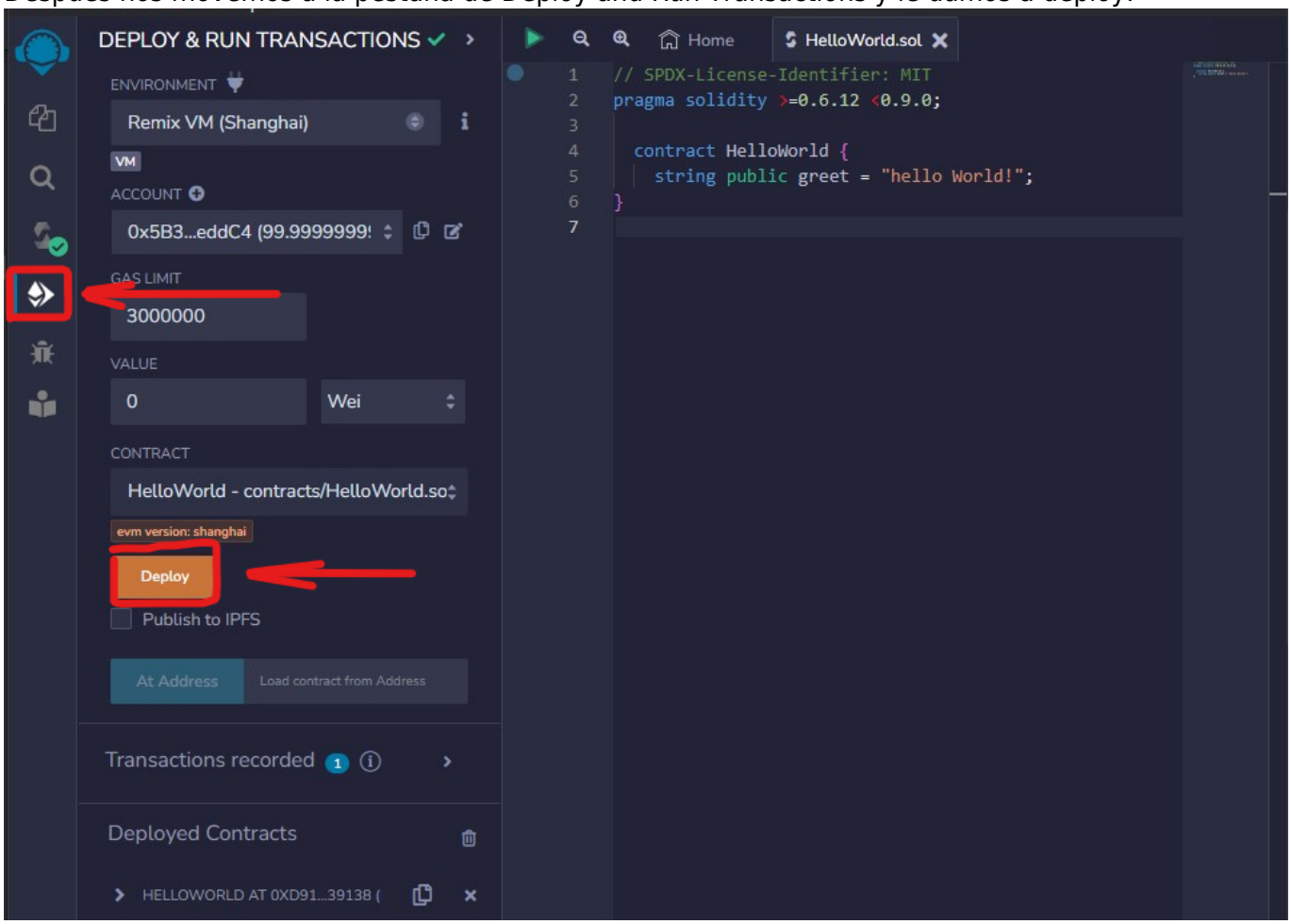
```
// SPDX-License-Identifier: MIT
pragma solidity >=0.6.12 <0.9.0;

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

Tras eso iremos a la pestaña de solidity compiler y le daremos a compile:



Después nos movemos a la pestaña de Deploy and Run Transactions y le damos a deploy:



Finalmente podemos ir a la pestaña de Deployed Contracts, seleccionar el contrato que acabamos de enviar y pulsar en el botón greet para ver e mensaje;