Desarrollo de Smart Contracts en Solidity

Introducción

Solidity es un lenguaje de programación de alto nivel utilizado para implementar contratos inteligentes en la blockchain de Ethereum. Fue influenciado por C++, Python y JavaScript y está diseñado para ser fácil de leer y entender. Los contratos inteligentes son programas que se ejecutan de manera autónoma en la blockchain y pueden controlar la transferencia de activos digitales.

Este documento proporciona una visión general de los conceptos clave y las características del lenguaje Solidity, incluyendo variables, constantes, ámbito, tipos, operaciones aritméticas, operaciones lógicas, operadores bit a bit, operaciones de comparación, sentencias condicionales, sentencias de repetición y funciones.

Variables

En Solidity, las variables son contenedores para almacenar datos. Se pueden declarar dentro de contratos, funciones o en cualquier ámbito de bloque.

Tipos de Variables

- **State Variables**: Declaradas fuera de cualquier función, son almacenadas en la blockchain.
- **Local Variables**: Declaradas dentro de funciones y solo existen mientras la función está siendo ejecutada.
- **Global Variables**: Variables especiales proporcionadas por la blockchain.

Constantes

Las constantes son variables cuyos valores no pueden ser cambiados después de su inicialización.

[crayon-6a9d4efa9f70b526337985/]

Ámbito

El ámbito de una variable define la región del código en la que la variable es accesible. En Solidity, el ámbito puede ser:

- **Público**: Accesible dentro y fuera del contrato.
- **Privado**: Accesible solo dentro del contrato en el que está declarado.
- **Interno**: Accesible dentro del contrato y los contratos derivados.
- **Externo**: Accesible solo desde fuera del contrato.

Tipos

Tipos Simples

- **Boolean**: Representa valores true o false.
- **Integer**: Enteros con y sin signo (int, uint).
- **Address**: Representa direcciones de Ethereum.

Tipos Complejos

- **Arrays**: Colecciones de elementos del mismo tipo.
- **Structs**: Colecciones de elementos de diferentes tipos.
- **Mappings**: Asociaciones de clave-valor.

Variables de Entorno

Solidity proporciona variables globales que contienen información sobre la blockchain y las transacciones, como msg.sender, msg.value, block.number, etc.

Operaciones Aritméticas

Sumar

[crayon-6a9d4efa9f711339845446/]

Restar

[crayon-6a9d4efa9f714657881641/]

Multiplicar

```
uint product = a * b;
```

Dividir

```
uint quotient = a / b;
```

Resto

```
uint remainder = a % b;
```

Conversiones entre Sistemas Numéricos

Solidity permite conversiones entre diferentes tipos de datos numéricos:

```
int8 smallInt = int8(bigInt);
uint8 smallUint = uint8(bigUint);
```

Operaciones Lógicas

And

```
bool a = true;
bool b = false;
bool result = a && b;
```

Or

```
bool result = a || b;
```

Not

```
bool result = !a;
```

Operadores Bit a Bit

- **AND**: &
- **OR**: |
- **XOR**: ^
- **NOT**: ~
- **Shift Left**: <<
- **Shift Right**: >>

```
uint a = 4; // 0100 in binary
uint b = 2; // 0010 in binary
uint andResult = a & b; // 0000
uint orResult = a | b; // 0110
uint xorResult = a ^ b; // 0110
uint notResult = ~a; // 1011
uint shiftLeft = a << 1; // 1000
uint shiftRight = a >> 1; // 0010
```

Operaciones de Comparación

eq (Equal to – Igual a)

```
bool result = (a == b);
```

lt (Less than – Menos que)

```
bool result = (a < b);
```

gt (Greater than – Más que)

```
bool result = (a > b);
```

ge (Greater than or Equal to – Mayor o igual a)

```
bool result = (a >= b);
```

le (Less than or Equal to – Menor o igual a)

```
bool result = (a <= b);
```

ne (Not equal to – No es igual a)

```
bool result = (a != b);
```

Sentencias Condicionales

If

```
if (a > b) {
    // Código
}
```

Else

```
if (a > b) {
    // Código
} else {
```

```
    // Código
}
```

ElseIf

```
if (a > b) {
    // Código
} else if (a == b) {
    // Código
} else {
    // Código
}
```

Switch (No disponible en Solidity)

Solidity no soporta sentencias switch. En su lugar, se deben usar sentencias if-else.

Sentencias de Repetición

Bucle While

```
uint i = 0;
while (i < 10) {
    i++;
}
```

Bucle Do-While

```
uint i = 0;
do {
    i++;
} while (i < 10);
```

Bucle For

```
for (uint i = 0; i < 10; i++) {
    // Código
}
```

Bucle Foreach (No disponible en Solidity)

Solidity no soporta bucles foreach. Se deben usar bucles for o while.

Funciones

Las funciones en Solidity son bloques de código reutilizables que pueden ser llamados desde otras partes del contrato.

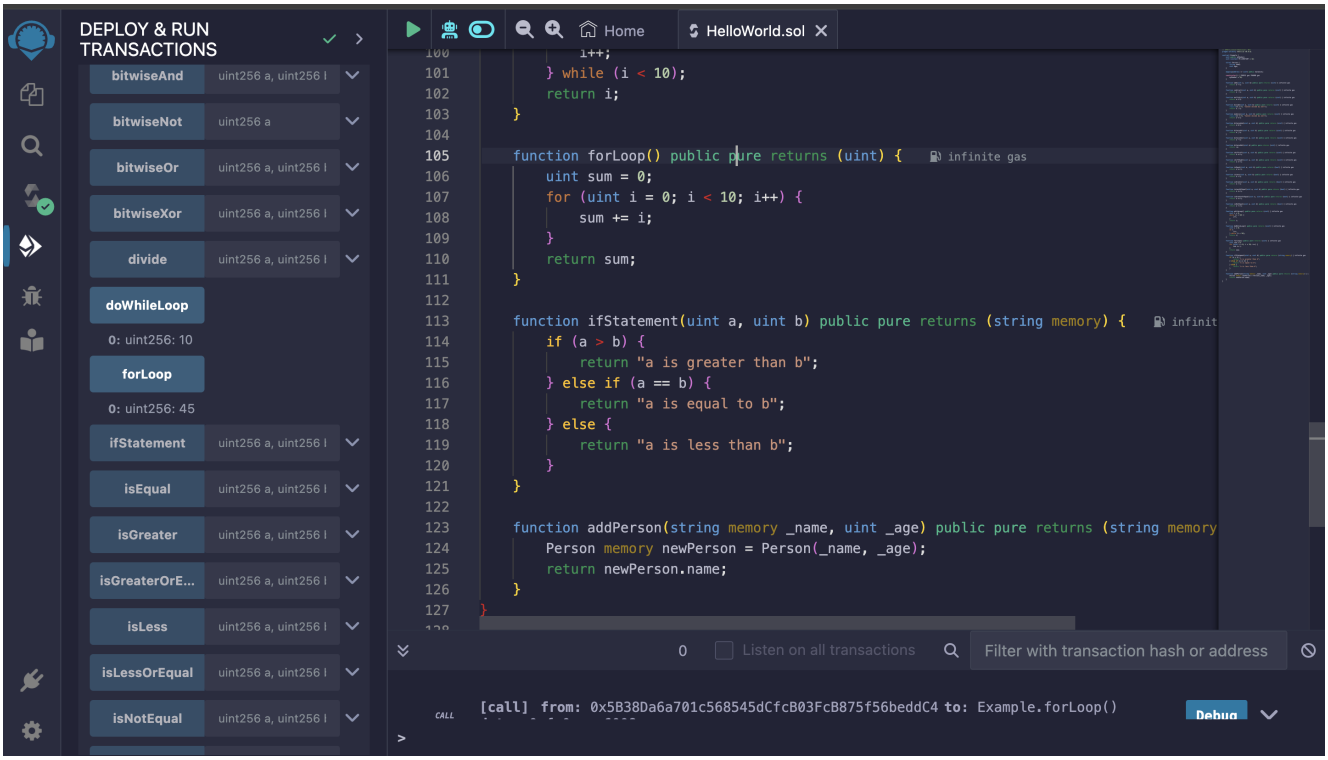
```
function add(uint a, uint b) public pure returns (uint) {
    return a + b;
}
```

Funciones Modificadoras

- **Pure**: Indica que la función no accede ni modifica el estado del contrato.
- **View**: Indica que la función no modifica el estado del contrato pero puede leerlo.
- **Payable**: Indica que la función puede recibir Ether.

Ejemplo integrador

[crayon-6a9d4efa9f716637495415/]


DEPLOY & RUN TRANSACTIONS
bitwiseAnd
uint256 a, uint256 l
bitwiseNot
uint256 a
bitwiseOr
uint256 a, uint256 l
bitwiseXor
uint256 a, uint256 l
divide
uint256 a, uint256 l
doWhileLoop
0: uint256: 10
forLoop
0: uint256: 45
ifStatement
uint256 a, uint256 l
isEqual
uint256 a, uint256 l
isGreater
uint256 a, uint256 l
isGreaterOrE...
uint256 a, uint256 l
isLess
uint256 a, uint256 l
isLessOrEqual
uint256 a, uint256 l
isNotEqual
uint256 a, uint256 l
Home
HelloWorld.sol
i++;
} while (i < 10);
return i;
}
function forLoop() public pure returns (uint) { infinite gas
uint sum = 0;
for (uint i = 0; i < 10; i++) {
sum += i;
}
return sum;
}
function ifStatement(uint a, uint b) public pure returns (string memory) { infinit
if (a > b) {
return "a is greater than b";
} else if (a == b) {
return "a is equal to b";
} else {
return "a is less than b";
}
}
function addPerson(string memory _name, uint _age) public pure returns (string memory
Person memory newPerson = Person(_name, _age);
return newPerson.name;
}
}
0
Listen on all transactions
Filter with transaction hash or address
CALL
[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Example.forLoop()
Debug
>