SOLIDITY

SMART CONTRACTS

PRIMEROS PASOS EN SOLIDITY

SMART CONTRACTS

Versión

Todo contrato debe empezar especificando la versión que debe usar el compilador

Código

```
pragma solidity ^0.4.0;
```

Versión

También podemos especificar un rango de versiones:

Código

```
pragma solidity >=0.4.0 <0.7.0;
```

Creación de un contrato

Un contrato es el bloque de construcción más básico

Código

```
contract <nombre_contrato>{
    ...
}
```

Constructor

Se especifican las propiedades que definen el contrato

Código

```
constructor() public{
    ...
}
```

Comentarios

Comentario de una línea:

```
Código
```

```
//Este es un comentario de una línea
```

Bloque de comentarios:

```
Código
```

```
/*
Aquí podemos especificar más de una línea de
comentarios
*/
```

Estándar de comentarios en Solidity

El formato estándar para los comentarios en Solidity es el natspec

Código

```
/// @title <Título del contrato>
/// @author <Autor del contrato>
/// @notice <Explicar lo que hace el contrato o función>
/// @dev <Detalles adicionales sobre el contrato o función>
/// @param <nombre_parametro> <Describir para qué sirve el
parámetro>
/// @return <valor_retorno> <Describir para qué sirve el
valor de retorno de una función>
```

PROPIEDADES DE LAS TRANSACCIONES Y LOS BLOQUES

SMART CONTRACTS

Funciones globales

`block.blockhash(blockNumber)`	Devuelve el hash de un bloque dado
`block.coinbase`	Devuelve la dirección del minero que está procesando el bloque actual
`block.difficulty`	Devuelve la dificultad del bloque actual
`block.gaslimit`	Devuelve el límite de gas del bloque actual
`block.number`	Devuelve el número del bloque actual
`block.timestamp`	Devuelve el timestamp del bloque actual en segundos
`msg.data`	Datos enviados en la transacción
`msg.gas`	Devuelve el gas que queda
`msg.sender`	Devuelve el remitente de la llamada actual

Funciones globales

`msg.sig`	Devuelve los cuatro primeros bytes de los datos enviados en la transacción
`msg.value`	Devuelve el numero de Wei enviado con la llamada
`now`	Devuelve el timestamp del bloque actual
`tx.gasprice`	Devuelve el precio del gas de la transacción
`tx.origin`	Devuelve el emisor original de la transacción

Función keccak256()

Esta función realiza el cómputo del hash de los argumentos introducidos como parámetros

Código

```
keccak256(<values>);
```

Keccak256() y abi.encodePacked()

Para calcular el hash con keccak256() hay que usar abi.encodePacked() para pasar los argumentos a tipo byte

Código

```
//Para poder usar la función abi.encodePacked()
pragma experimental ABIEncoderV2;

contract <nombre_contrato>{
    ...
    //Computo del hash
    keccak256(abi.encodePacked(<values>));
    ...
}
```

TIPOS DE VARIABLES Y OPERADORES

SMART CONTRACTS

Variables

En una variable guardamos y recuperamos los datos que usan en el contrato

Código

```
<tipo_dato> <nombre_variable>;
```

Inicializar una variable

Código

```
<tipo_dato> <nombre_variable> = <valor>;
```

Variables Enteras

Hay dos tipos de variables enteras en Solidity:

Variables enteras sin signo	uint
Variables enteras con signo	int

Código

```
uint <nombre_variable>;
int <nombre_variable>;
```

Variables Enteras

Podemos especificar el número de bits de la variables enteras

Código

```
uint<x> <nombre_variable>;
int<x> <nombre_variable>;
```

<x> varia de 8 a 256 en múliplos de 8

Variables strings

Las variables strings son cadenas de texto UTF-8 de longitud arbitraria

Código

```
string <nombre_variable>;
```

Variables booleanas

Las variables booleanas pueden tomar dos valores:

Código

```
bool <nombre_variable>;
```

Variables bytes

Las variables de tipo bytes contienen una secuencia de bytes

Código

```
bytes<x> <nombre_variable>;
```

<x> varía de 1 hasta 32

Variables address

Las variables de tipo address contienen una dirección de 20 bytes

Código

```
address <nombre_variable>;
```

Enums

Los enums son una manera para el usuario de crear su propio tipo de datos

Código

```
enum <nombre_enumeracion> {valores_enumeracion}
```

Declarar una variable de tipo enum:

Código

```
<nombre_enumeracion> <nombre_variable>;
```

Modificar el valor de una variable enum

Hay dos maneras de modificar el valor de una variable enum:

Código

```
//1. Especificando la opción de la enumeración
<nombre_variable> = <nombre_enumeracion>.<valor_enumeracion>;

//2. Con el índice
<nombre_variable> = <nombre_enumeracion>(<posicion>);
```

Unidades de tiempo

En Solidity tenemos los siguientes sufijos, que nos ayudan a tratar con el tiempo:

`<x> seconds`
`<x> minutes`
`<x> hours`
`<x> days`
`<x> weeks`
`<x> years`

`<x>` es un número entero positivo (1, 2, 3, ...)

Casteo de variables

Podemos transformar un uint (o un int) con y número de bits a un uint (o un int) con x número de bits.

Código

```
uint<x> (<dato_uint<y>);
int<x> (<dato_int<y>);
```

Podemos transformar un int con y número de bits a un uint con x número de bits y viceversa

Código

```
uint<x> (<dato_int<y>);
int<x> (<dato_uint<y>);
```

Modificador public

Si añadimos el modificador public al declarar una variable, se creará una función getter

Código

```
<tipo_dato> [public]* <nombre_variable>;
```

Modificador private y internal

Código

```
<tipo_dato> [public | private | internal]* <nombre_variable>;
```

Private: Las variables private solo son visibles desde dentro del contrato

Internal: Las variables internal solo son accesibles internamente

Modificador memory y storage

Código

```
<tipo_dato> [memory | storage]* <nombre_variable>;
```

Memory: Guardado de manera temporal

Storage: Guardado permanentemente en la Blockchain

Modificador payable

Código

```
address [payable]* <nombre_variable>;
```

Payable: Permite enviar y recibir ether

Operadores matemáticos

+	suma
-	resta
*	multiplicación
/	división
%	módulo
**	exponenciación

Comparadores de enteros

>	mayor estricto
<	menor estricto
>=	mayor o igual
<=	menor o igual
==	igualdad
!=	inigualdad

Operadores booleanos

!	negación
&&	and
||	or
==	igualdad
!=	inigualdad

ESTRUCTURAS DE DATOS

SMART CONTRACTS

Estructuras

Las estructuras nos permiten definir tipos de datos más complejos

Código

```
struct <nombre_estructura>{
    <data_type_1> <nombre_variable_1>;
    <data_type_2> <nombre_variable_2>;
    <data_type_3> <nombre_variable_3>;
    ...
}
```

Trabajar con estructuras

Para crear una variable del tipo “estructura” debemos hacerlo del siguiente modo:

Código

```
//Declarar una variable “struct”
<nombre_estructura> <nombre_variable>;

//Declar y inicializar una variable “struct”
<nombre_estructura> <nombre_variable> =
<nombre_estructura> (<propiedades_estructura>);
```

Mappings

Asociación clave-valor para guardar y ver datos.

Código

```
mapping (_keyType => _valueType) [public]*
<nombre_mapping>;
```

Cómo guardar y ver datos con los mappings

Guardar datos:

Código

```
<nombre_mapping> [_key] = _value;
```

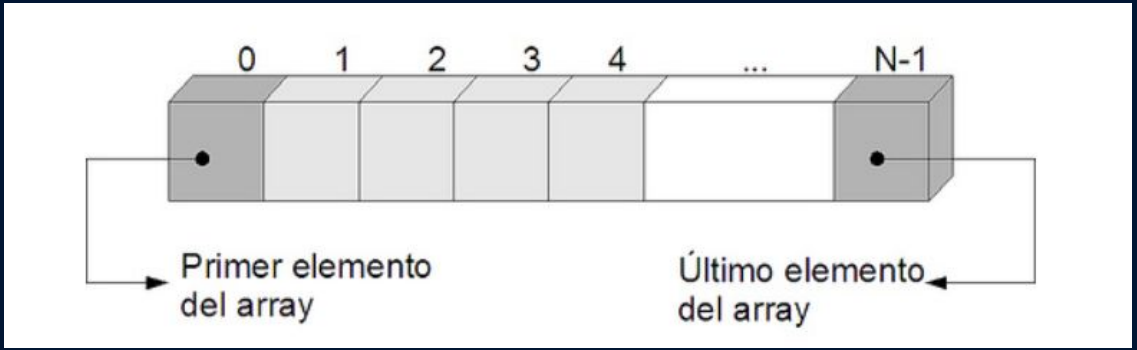
Ver datos:

Código

```
<nombre_mapping> [_key];
```

¿Qué es un array?

Tipo de dato estructurado que almacena un conjunto homogéneo de datos



Arrays

Array de longitud fija

Código

```
<tipo_de_dato> [<longitud>] [public]* <nombre_array>;
```

Array dinámico

Código

```
<tipo_de_dato> [] [public]* <nombre_array>;
```

Trabajar con arrays

Inicializar un array de longitud fija:

Código

```
<tipo_de_dato> [<longitud>] [public]* <nombre_array> =
[<valores>];
```

Inicializar un array dinámico:

Código

```
<tipo_de_dato> [] <nombre_array> [public]* =
[<valores>];
```

Trabajar con arrays

Para acceder a un elemento del array necesitamos su posición

Código

```
<nombre_array>[<posicion>];
```

Para fijar el valor de una posición del array

Código

```
<nombre_array>[<posicion>] = <valor>;
```

Función .push() y .length

La función .push() añade un elemento al final del array

Código

```
<nombre_array>.push(<value>);
```

La función .length devuelve la longitud del array

Código

```
<nombre_array>.length;
```

FUNCIONES

SMART CONTRACTS

Funciones

Las funciones son las unidades ejecutables del código dentro de un contrato

Código

```
function <nombre_funcion>(<tipos_parametro>) [public | private]{
    ...
}
```

Return

Las funciones pueden devolver valores de retorno al ser ejecutadas

Código

```
function <nombre_funcion>(<tipos_parametro>) [public | private]
[returns (<return_types>)]*{
    ...
    [return (<return_values>)]*
}
```

Manejar valores devueltos

Un solo valor de retorno:

Código

```
<type> =
<nombre_funcion>(<parametros>);
```

Múltiples valores devueltos:

Código

```
//Asignación múltiple de todos los valores devueltos
(type_1,type_2,...,type_n)=<nombre_funcion>(<parametros>);

//Asignación de un subconjunto de los valores devueltos
(,,,type_i,,,)=<nombre_funcion>(<parametros>);
```

Modificador view, pure y payable

Código

```
function <nombre_funcion>(<tipos_parametro>) [public | private]
[view | pure | payable]* [returns (<return_types>)]*{
    ...
}
```

View: No modifica los datos pero sí accede a ellos

Pure: No accede ni siquiera a los datos

Payable: Permite recibir ether

Eventos

Los eventos comunican un suceso en la cadena de bloques

Declarar un evento:

Código

```
event <nombre_evento>(types);
```

Emitir un evento:

Código

```
emit <nombre_evento>(values);
```

BUCLES Y CONDICIONALES

SMART CONTRACTS

Sentencia if

La sentencia if ejecuta un bloque de código si la expresión booleana es cierta

Código

```
if(<condicion>){
    //Código a ejecutar si <condición> es verdadera
}
```

Código

```
if(<condicion>){
    //Código a ejecutar si <condición> es verdadera
}else{
    //Código a ejecutar si <condición> es falsa
}
```

Bucle for

El bucle for ejecuta un bloque de código un número finito de veces

Código

```
for(<iniciar_contador>; <comprobar_contador>; <augmentar_contador>){
    //Código a ejecutar en cada iteración
}
```

Bucle while

El bucle while ejecuta un bloque de código mientra se cumpla la expresión booleana

Código

```
while (<condicion>){
    //Código a ejecutar mientras se cumpla <condicion>
}
```

Break

La sentencia break detiene la ejecución de un bucle y sale de él

Código

```
<bucle> (<condicion>){
    ...
    break;
    ...
}
```

INTERACTUANDO CON VARIOS SMART CONTRACTS

SMART CONTRACTS

Herencia

La herencia simplifica y organiza el código

Código

```
contract <nombre_contrato_1>{
    ...
}

contract <nombre_contrato_2> is <nombre_contrato_1>{
    ...
}
```

Imports

Para poder usar un contrato en nuestro proyecto, debemos previamente importarlo

Código

```
// ./ significa mismo directorio
import “./<nombre_contrato>.sol”;

import {<contratos>} from “./<nombre_contrato>.sol”;
```

Librería

El uso de librerías permite facilitar el uso de funciones externas

Código

```
library <nombre_libreria>{
    ...
}
```

ALTERNATIVA: Crear otro Smart Contract con las funciones de la librería y importarlo al Smart Contract principal

Uso de una librería

Para poder usar una librería, debemos seguir los siguientes pasos:

Código

```
//1. Importar la librería (en caso de que no esté dentro del
// archivo del contrato)
import <nombre_libreria> from “./<nombre_archivo>.sol”;

//2. Definir la librería por su nombre y su tipo
using <nombre_libreria> for <tipo_dato>;
```

¡ATENCIÓN!: Podemos usar el valor * para <tipo_dato> que indica que podemos usar cualquier tipo de dato

Interfaz

Para poder interactuar con otro contrato de la blockchain es necesario hacer uso de una interfaz

Código

```
contract <nombre_interfaz>{
    //Declaramos las funciones con las que queremos interactuar
    function <nombre_funcion> ... ;
}
```

Uso de la interfaz

Para poder hacer uso de la interfaz se deben realizar los siguientes pasos:

Código

```
//1. Declarar un puntero que apunte al otro contrato

<nombre_interfaz> <nombre_puntero> =
<nombre_interfaz>(direccion_contrato);

//2. Usar el puntero para usar las funcionalidades del
// otro contrato definidas en la interfaz

<nombre_puntero>.<funcion>(<parmatros>);
```

FUNCIONES AVANZADAS

SMART CONTRACTS

Modificador internal y external

Código

```
function <nombre_funcion>(<tipos_parametro>) [public | private |
internal | external] [view | pure | payable]* [returns
(<return_types>)]*{
    ...
}
```

Internal: Parecido a private. Solo se pueden llamar desde el contrato actual o contratos que deriven de él

External: Solo pueden ser llamadas desde fuera del contrato

Require

Lanza un error y para la ejecución de una función si la condición no es verdadera

Código

```
function <nombre_funcion>(<tipos_parametro>) [public | private |
internal | external] [view | pure | payable]* [returns
(<return_types>)]*{
    require(<condicion>, [<mensaje_condicion_falsa>]*);
}
```

Modifier

Permite cambiar el comportamiento de una función de manera ágil

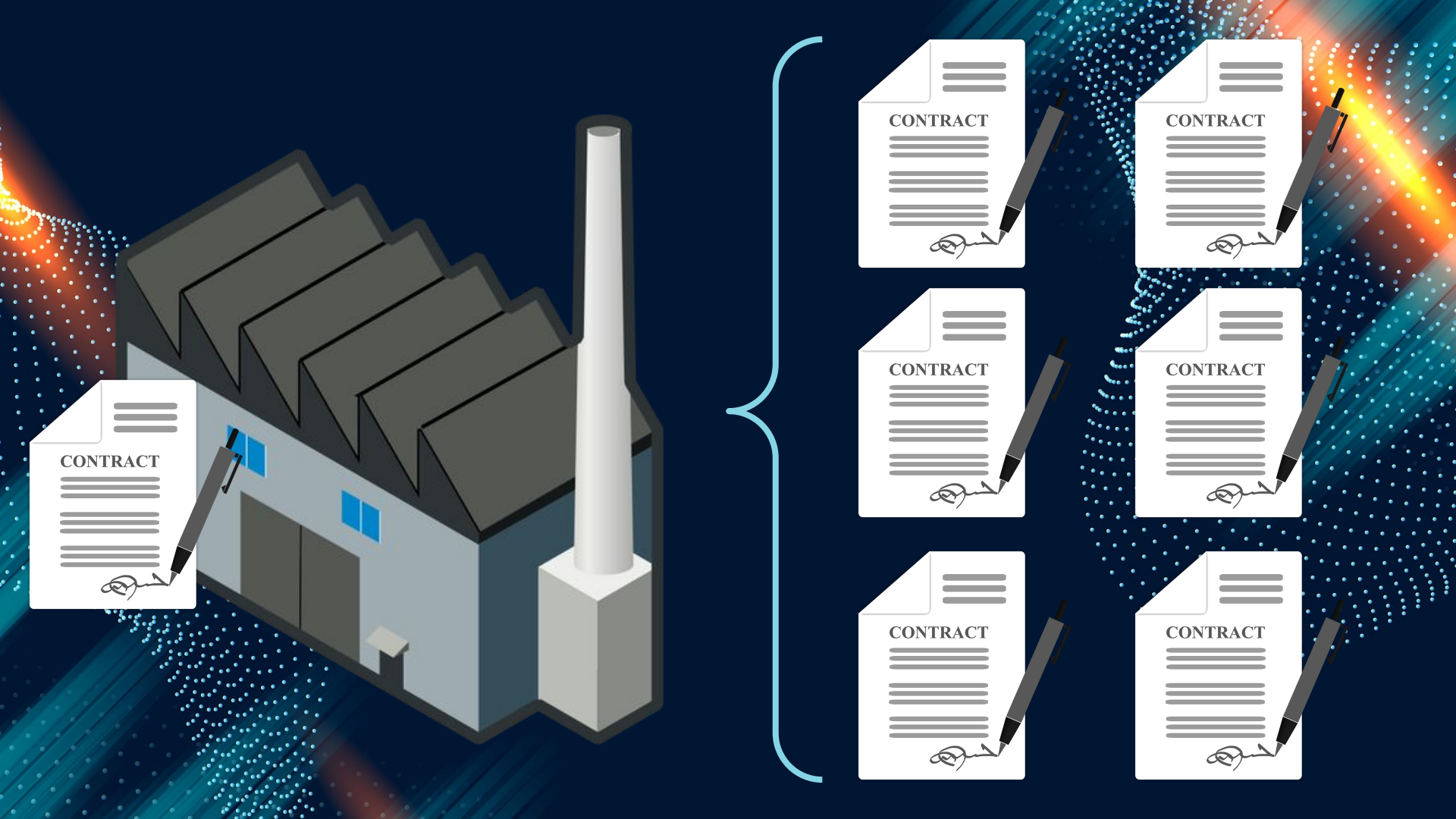
Código

```
modifier <nombre_modificador> (<parametros>)*{
    require(<condicion>);
    _;
}

function <nombre_funcion>(<tipos_parametro>) [public | private |
internal | external] [view | pure | payable]* [returns
(<return_types>)]* [<nombre_modificador> (<parametros>)*]*{
    ...
}
```

FÁBRICA DE CONTRATOS

SMART CONTRACTS


CONTRACT
CONTRACT
CONTRACT
CONTRACT
CONTRACT
CONTRACT
CONTRACT

Factory

Creación de un Smart Contract a partir de una función de otro Smart Contract

Código

```
contract SmartContract1{

Funcion Factory() public {
    address direccion_nuevo_contrato = address (new
    SmartContract2( parametros));
    }

contract SmartContract2{

    constructor (parametros) public {...}

    }
```