

Blockchain

Blockchain & Money

 Smart Contracts

Curso 2024/2025

 **Francisco Hernando Gallego**

 fhernando@uva.es

 **Diego Martín de Andrés**

 diego.martin.andres@uva.es



De la evolución del dinero a los contratos inteligentes

- El primer día vimos:
 - La evolución del dinero: del trueque al blockchain.
 - Cómo las cadenas de bloques permiten registrar información sin intermediarios.
- Hoy:
 - Aprendemos cómo usar esa tecnología para **automatizar reglas y acuerdos: los contratos inteligentes.**
 - Exploramos casos de uso más allá del dinero: trazabilidad, identidad, automatización, etc.

Día 4 – Contratos Inteligentes y Remix IDE

- ¿Qué es un Smart Contract? (contrato automático)
- ¿Para qué sirven?
- ¿Qué es una DApp? (aplicación sobre blockchain)
- Retos principales
- Solidity paso a paso
- Remix IDE: tutorial guiado y ejercicios prácticos

Historia y evolución de los Smart Contracts

- **1994:** *Nick Szabo* propone la idea de contratos automáticos.
 - ✨ **Ejemplo:** una **máquina expendedora**: si metes dinero, te da una lata.
Nadie tiene que verificar nada, el mecanismo lo hace solo.
- **2009:** *Bitcoin* incluye una forma limitada de automatizar pagos.
 - 📄 **Analogía:** piensa en un **cuaderno compartido** (blockchain): todos ven lo que se escribe, y nadie puede borrar ni cambiar lo que ya está escrito.
- **2015:** *Ethereum* lleva esto más lejos: crea un sistema que **ejecuta programas enteros dentro de la blockchain**.
 - 🧠 **¿Qué permite?** Automatizar cualquier proceso digital: votaciones, permisos, intercambios, registros...

◆ Session 4.1: Smart Contracts y DApps

Ethereum y las nuevas aplicaciones

- **Ethereum** fue la primera blockchain pensada no solo para dinero, sino para **ejecutar programas completos**.
 -  Estos programas se llaman **Smart Contracts**.
 -  Ethereum tiene un lenguaje propio: **Solidity**, parecido a JavaScript.
 -  Todos los contratos se guardan en la blockchain: son visibles, públicos, y no se pueden modificar.



◆ Session 4.1: Smart Contracts y DApps

Nuevas aplicaciones

- ¿Qué es una DApp?
 - 🔍 Es una **aplicación descentralizada**: como una app del móvil, pero que usa **blockchain** por debajo.
 - 🎯 **Objetivo**: interactuar con contratos inteligentes.
 - ⚙️ Tiene dos partes:
 - a. **Frontend**: la parte visual (como una web).
 - b. **Backend en blockchain**: el contrato inteligente que hace el trabajo.

◆ Session 4.1: Smart Contracts y DApps

Nuevas aplicaciones

-  Surgen nuevas blockchains además de Ethereum: *Solana, Cardano, BNB Chain*, que también permiten contratos inteligentes.
-  Gracias a eso, aparecen nuevas aplicaciones:
 - **DeFi**: finanzas sin bancos.
 - **NFTs**: coleccionables digitales únicos.
 - **DAOs**: organizaciones donde se vota todo automáticamente con contratos.

¿Qué es un Smart Contract?

- Programa automático en blockchain.
- Analogía:
Como una máquina expendedora: pones dinero y te da el producto sin pedirle a nadie.
- Hace tareas solo cuando se cumplen ciertas condiciones.
- No necesita personas para funcionar.

Características

- No necesita intermediarios.
- Nadie puede cambiarlo una vez creado.
- Todos pueden ver cómo funciona.
- **Transparencia:** Todos ven el código.
- **Inmutabilidad:** No se puede cambiar.
- **Siempre hace lo mismo:** Si le das lo mismo, responde igual.

Elementos de un contrato inteligente

- **Funciones:** Hacen tareas.
 - *Ejemplo:* Pulsar el botón de abrir puerta.
- **Eventos:** Avisan cuando pasa algo.
 - *Analogía:* Una alarma que suena cuando abres la puerta.
- **Estructuras:** Guardan varios datos juntos.
 - *Ejemplo:* Una ficha de alumno: nombre, edad, nota.

Más elementos de un contrato

- **Variables:** Guardan datos.
- **Visibilidad:** Quién puede usar los datos o funciones.
 - `public` : todos pueden ver y usar.
 - `private` : solo el contrato puede usar.
 - `internal` : solo este y los hijos pueden usar.
 - `external` : solo desde fuera.
- *Ejemplo:* Como una puerta con llave: algunos pueden entrar, otros no.
- **Modificadores:** Reglas extra para funciones.
 - *Ejemplo:* Solo el dueño puede borrar datos.

¿Cómo se relacionan los Smart Contracts con los bloques y los hashes?

- Al desplegar un contrato, se guarda dentro de un **bloque**.
- Ese bloque tiene un **hash único** que asegura que nada se ha cambiado.
- El contrato también tiene una **dirección única (hash del código)**.
- Cada vez que se usa el contrato, se genera una **transacción** que va a otro bloque.
- Así se garantiza la **transparencia e inmutabilidad**.

¿Y las DApps?

- Una DApp es una aplicación que **usa Smart Contracts** para funcionar.
- Cada acción de la DApp crea una **transacción** en la blockchain.
- Todo queda grabado en bloques y verificado con hashes.

💡 Analogía: DApp + Smart Contract = Web con superpoderes

Imagina una **página web normal**, como una tienda online:

- Tiene un servidor que guarda los productos.
- Hay una base de datos que guarda los pedidos.
- El dueño puede cambiar los datos cuando quiera.

🔄 Ahora, imagina una **DApp**:

- Usa **Smart Contracts** para guardar los productos y los pedidos.
- Nadie (ni el dueño) puede cambiar lo que ya está guardado.
- Cada pedido queda grabado **para siempre** en la blockchain (como si se esculpiera en piedra).
- Cada acción tiene su propio **hash único** y se guarda en un **bloque** enlazado al anterior.

Usar una DApp

Ventajas

- **Transparencia:** Todos pueden ver el código y las reglas.
- **Inmutabilidad:** Nadie puede cambiar los datos una vez grabados.
- **Sin intermediarios:** No necesitas confiar en una empresa.
- **Accesible desde cualquier parte del mundo.**

Desventajas

- **Dificultad técnica:** Puede ser complicado para quienes empiezan.
- **Errores permanentes:** Si está mal programado, no se puede corregir fácilmente.
- **Coste:** Cada acción puede costar gas.
- **Velocidad:** Son más lentas que las apps normales.



Debate en clase: ¿Qué es un Smart Contract?

¿Cómo explicarías qué es un Smart Contract a alguien que no sabe nada de tecnología?

- ¿Usarías una analogía?
- ¿Lo compararías con algo de la vida diaria?
- ¿Qué ventajas crees que tiene frente a los contratos tradicionales?



Intercambiamos ideas entre todos.

◆ Transparencia

- Todos pueden ver lo que hace el contrato.
- Las reglas están claras desde el principio.
- Aumenta la **confianza**: nadie puede ocultar condiciones.
- Ejemplo: Un sorteo donde todos pueden ver cómo se elige al ganador.

◆ Inmutabilidad

- Una vez desplegado, **no se puede cambiar**.
- Protege contra manipulaciones o trampas.
- Ejemplo: Un contrato que paga a un trabajador automáticamente el día 30 de cada mes, sin que nadie pueda retrasarlo o cambiarlo.

⚠ ¿Problema?

- Si hay errores en el código, **no se puede editar**.
- Se deben revisar y probar muy bien antes de publicar un contrato.
- A veces se hacen versiones nuevas y se avisa a los usuarios.



Preguntas de comprensión – Historia y evolución

- ¿Quién introdujo el concepto de Smart Contracts y en qué año?
Nick Szabo, en 1994.
- ¿Qué es una blockchain en palabras sencillas?
Un registro digital descentralizado e inmutable que guarda información ordenada por bloques y protegida por criptografía.
- ¿Por qué Ethereum es importante en la historia de los contratos inteligentes?
Porque fue la primera blockchain ampliamente adoptada que permitió programar y ejecutar contratos inteligentes de forma flexible.

Casos de uso comunes

- **DeFi:** préstamos automáticos, intercambios.
- **Juegos y NFTs:** objetos digitales únicos.
- **Seguros automáticos:** pagan solos si pasa algo (ej: vuelo retrasado).
- **Votaciones:** votos transparentes y sin trampas.
- **Seguimiento de productos:** saber de dónde viene algo.

Ejemplo: Contrato de alquiler digital: si pagas, entras; si no, no se abre la puerta.

◆ Casos de uso de Smart Contracts

Gestión de suscripciones

- Contrato para pagar cada mes por un servicio.
- Solo puedes entrar si pagaste.
 - *Ejemplo:* Como un gimnasio: entras solo si pagaste la cuota.

Venta de entradas (ticketing)

- Cada entrada es un NFT (única).
- No se puede copiar ni falsificar.
- Verifica al comprador solo.
 - *Ejemplo:* Billeto digital de concierto que no se puede duplicar.

Microfinanzas y ahorro

- Contratos que guardan dinero hasta llegar a una meta.
- Préstamos automáticos entre personas.
 - *Ejemplo:* Hucha digital que solo se abre si llegas al objetivo.

Contratos de seguros

- Se activan solos si pasa algo (ej: lluvia, accidente).
- Un oráculo trae los datos del mundo real.
 - *Ejemplo:* Oráculo es como un mensajero digital que trae información.

Preguntas de comprensión – Nuevos casos de uso

1. ¿Por qué usar NFTs para entradas?

- Son auténticas, no se pueden copiar, sabes quién la tiene.

2. ¿Qué es un oráculo?

- Un mensajero que trae datos reales a blockchain.

3. ¿Cómo gestiona un contrato las suscripciones?

- Revisa si pagaste y activa/desactiva según la fecha.

Pseudocódigo: ejemplo de contrato sencillo

Contrato SimpleStorage:

Guarda un número (data)

Guarda quién es el dueño (owner)

Al crear, el dueño es quien lo lanza

set(nuevoValor): cambia el número y avisa

get(): devuelve el número guardado

Evento DataChanged: avisa cuando cambia el número

Pseudocódigo: ejemplo básico de contrato

Explicación en pasos:

- Guarda un número.
- Guarda quién lo creó.
- Al crear, pone el dueño.
- `set()`: cambia el número y avisa.
- `get()`: devuelve el número.
- Evento: avisa cuando cambió.

Preguntas de comprensión – Pseudocódigo

1. ¿Qué función cambia el número?

- `set(newData)`

2. ¿Para qué sirve el constructor?

- Pone el dueño al crear el contrato.

3. ¿Por qué hay un evento?

- Para avisar cuando el número cambió.

¿Qué es una DApp?

Aplicación Descentralizada (DApp)

- App que funciona en blockchain.
- Usa contratos inteligentes.
- La parte visual es como una web normal.
- Los datos y reglas están en blockchain.

Analogía: Como una web, pero la base de datos está en blockchain y las reglas en contratos.

Componentes principales

- **Pantalla visual:** Lo que ve el usuario.
- **Conexión Web3:** Une la app a la blockchain.
 - *Ejemplo:* Metamask es tu "llave digital" para firmar.
- **Contratos desplegados:** Las reglas en blockchain.
- **Red blockchain:** Dónde se ejecuta (pruebas o real).

Preguntas de comprensión – DApps

1. ¿En qué se diferencia una DApp?

- Usa blockchain y contratos, no servidores normales.

2. ¿Para qué sirve Metamask?

- Es tu cartera digital y firma tus acciones.

3. ¿Qué diferencia hay entre tener tus fondos en MetaMask o en un exchange (como Binance)?

- MetaMask: tú controlas tu clave privada. Solo tú puedes mover el dinero.
- Exchange: otra empresa guarda tus claves. Si hay un problema, puedes perder el acceso.
- 🧠 Reflexión: ¿Qué es más cómodo? ¿Y qué es más seguro?


Ejemplos reales de DApps

- **Uniswap:** cambiar monedas digitales solo, sin personas.
- **OpenSea:** comprar y vender NFTs.
- **Aave:** pedir y dar préstamos sin bancos.
- **Decentraland:** mundo virtual donde compras terrenos digitales.

Ejemplo: Uniswap es como una casa de cambio automática sin cajero.

Sesión 4.2: Retos técnicos

¿Por qué hay problemas técnicos?

- Ya sabemos crear contratos inteligentes y cómo funcionan las DApps.
- Pero... ¿funciona igual de bien si lo usa poca gente que si lo usan millones?
-  Cuantos más usuarios y más contratos, **más difícil es mantener la red rápida, segura y barata.**
- Vamos a ver algunos de estos retos y cómo se intentan solucionar.

1. Escalabilidad

- Ethereum: solo ~15 transacciones por segundo.
- Visa: ~24,000 por segundo.
- Solana: ~50,000 por segundo.

Soluciones para escalar:



- **Rollups:** Juntar muchas transacciones y enviarlas de golpe.
- **Sidechains:** Cadenas secundarias para ayudar.
- **Sharding:** Dividir la red en partes para trabajar más rápido.

 *Ejemplo:* Como enviar un solo paquete con muchas cartas.

2. Seguridad

Los contratos no se pueden cambiar. Si hay errores, **pueden robar dinero** o hacer cosas que no querías.

- **Reentrancy**

- Un atacante puede llamar muchas veces a una función antes de que termine.
-  Como si alguien entra por la puerta, y antes de que se cierre, vuelve a entrar otra vez muchas veces.
-  Solución: cerrar la puerta primero (guardar los datos) y luego enviar el dinero.

Problemas

- **Overflows**

- Cuando sumas un número muy grande y se pasa del límite.
- Ejemplo: si puedes tener hasta 100 y sumas 1 → vuelve a 0.
- Solución: usar `SafeMath` o versiones modernas de Solidity (≥ 0.8).

3. Costes de ejecución (Gas)

- Cada acción cuesta gas (se paga en ether).
- Cuanto más complejo, más caro.
- Si la red está muy ocupada, sube el precio.
- Optimizar: guardar menos datos, usar código simple.
- *Ejemplo:* Como pagar peaje: más tráfico, más caro.

4. Experiencia de usuario (UX)

- DApps pueden ser difíciles para principiantes.
 - **Wallet:** Tu cartera digital.
 - **Firma:** Tu "firma digital" para aceptar.
- Mejorar: apps más fáciles, menos pasos, a veces sin pagar gas.
- *Nota:* Cada vez es más fácil usar DApps.

5. Interoperabilidad


- Difícil conectar diferentes blockchains.
- Hay proyectos para unirlos (Polkadot, Cosmos).
- Estándares ayudan (ERC-20 para monedas, ERC-721 para NFTs).
- *Ejemplo:* Como usar un cargador universal para todos los móviles.

6. Gobernanza


- Decidir cambios en contratos y proyectos.
- **DAO:** grupo que vota en blockchain.
 - *Ejemplo:* Como una reunión de vecinos, pero todo es digital.
- Dos formas:
 - **On-chain:** Votas en blockchain.
 - **Off-chain:** Votas fuera y luego lo aplican.
- Riesgos: algunos pueden tener mucho poder.

¿Cómo se puede solucionar un contrato mal hecho?


1. Pausar el contrato (`pause`)

- Algunos contratos permiten detener su funcionamiento en emergencias.
-  Gana tiempo para analizar el error y evitar más daños.

2. Usar un proxy (contrato actualizable)

- El contrato principal apunta a otro que sí tiene la lógica.
-  Se puede actualizar el contrato de lógica sin cambiar la dirección.

3. Desplegar una nueva versión

- Se lanza un contrato nuevo y se migran usuarios o fondos.
-  Requiere confianza de los usuarios y puede ser lento.

4. Compensaciones manuales

- El equipo devuelve fondos a los afectados. No siempre es posible, pero mejora la reputación.

Debate: ¿Cómo protegemos un contrato?

1. ¿Qué pasaría si lanzamos un contrato sin probarlo bien?

→ Puede tener fallos graves: robar fondos, bloquear funciones, errores imposibles de corregir.

2. ¿Deberían todos los contratos estar auditados por expertos antes de usarse?

→ Lo ideal sería que sí, sobre todo si manejan dinero. Pero las auditorías son caras y no todos pueden permitírselo.

3. ¿Es mejor que los contratos sean inmutables o permitir actualizaciones en caso de errores?

→ Inmutables = más seguridad y confianza.

→ Pero permitir cambios (con control) ayuda a arreglar errores. Se puede usar un sistema de "proxy" o pausas.

4. ¿Conoces algún caso en el que un contrato falló por un error de seguridad?

→ Sí, por ejemplo **The DAO (2016)**: un fallo en el contrato permitió robar millones en ETH.

→ También el **Ronin Hack (Axie Infinity)** o el ataque a **Nomad Bridge** por errores en el código.

◆ Práctica (4 horas)

🧭 ¿Qué haremos en esta sesión práctica?

Objetivo:

Aprender a crear y probar contratos usando Remix IDE (todo online).

¿Qué haremos?

- Abrir Remix IDE.
- Escribir contratos muy simples.
- Probar funciones y eventos.
- Hacer ejercicios guiados paso a paso.



Transición: de la teoría a la práctica

Ahora que hemos aprendido qué son los Smart Contracts, sus características, y cómo funcionan las DApps, pasamos a una parte fundamental del curso:



¡Vamos a programar!






- Veremos cómo escribir y probar contratos inteligentes de forma sencilla y sin necesidad de instalaciones complejas.
- Usaremos **Remix IDE**, una plataforma online muy útil y completa para comenzar.

Primeros pasos con Solidity – Remix IDE

Vamos a hacer esto paso a paso

1. Abre <https://remix.ethereum.org> en tu navegador.
2. No necesitas cuenta. Todo se guarda solo.

Resumen visual – Remix IDE

-  Crear archivo `.sol`
-  Escribir código
-  Compilar con el icono de ladrillo
-  Desplegar y probar con el cohete
-  Ver resultados y avisos abajo



Mini ejercicio: Tu primer contrato inteligente

Vamos a crear un contrato simple llamado `SimpleStorage` que guarda y devuelve un número.

Este ejercicio te ayudará a entender cómo funcionan las variables, funciones y la estructura básica de un contrato en Solidity.



¿Qué hace este contrato? ¿Por qué es importante?

- Este contrato se llama `SimpleStorage` y es el "Hola Mundo" de Solidity.
- Nos permite aprender cómo guardar, cambiar y leer un número en la blockchain.


Paso a paso explicado

- `uint public data;`
 - Guarda un número visible públicamente.
 - La palabra `public` crea una función automática para leerlo.
- `function set(uint _value)`
 - Cambia el valor de `data` con el valor que le pasamos.
 - Es como “escribir” en la blockchain.
- `function get()`
 - Devuelve el valor actual de `data`.
 - Es como “leer” lo que está guardado.

¿Por qué es útil?

- Aunque es muy básico, este contrato nos enseña los elementos clave:
 - Variables
 - Funciones
 - Visibilidad (`public`)
 - Interacción desde fuera (lectura y escritura)
- Y lo más importante: ¡todo esto ocurre de forma **descentralizada** y **segura** en la blockchain!

1. Crea un archivo nuevo:

- Haz clic en el icono de carpeta  a la izquierda.
- Pulsa "New File" y pon `SimpleStorage.sol`.

2. Escribamos el contrato paso a paso:

Paso 1:

Vamos a crear una variable pública de tipo `uint`.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleStorage {
    uint public data; // Variable pública, cualquiera puede verla
```

Paso 2:

Ahora añadimos una función que nos permite cambiar el valor de esa variable.


```
function set(uint _value) public { // Cambia el valor de 'data'  
    data = _value; // Guarda el nuevo valor  
}
```

Paso 3:


Por último, creamos una función para leer el valor desde fuera del contrato.

```
function get() public view returns (uint) { // Devuelve el valor de 'data'  
    return data; // Retorna el número guardado  
}
```

3. Compila el contrato:

- Haz clic en el icono  "Solidity Compiler" a la izquierda.
- Selecciona `SimpleStorage.sol` y pulsa "Compile".

4. Despliega y prueba:

- Haz clic en el icono  "Deploy & Run Transactions" a la izquierda.
- Asegúrate de que esté seleccionada la opción "Remix VM" o la cuenta adecuada.
- Pulsa "Deploy" para desplegar el contrato.
- Abajo, en la sección "Deployed Contracts", verás el nombre `SimpleStorage` con un desplegable.

✓ Para probar `set()` :

- Haz clic en la función `set` .
- Introduce un número (por ejemplo, `42`) en el campo que aparece.
- Pulsa el botón de confirmación: eso guarda el número en la blockchain.

👁️ Para probar `get()` :

- Haz clic en la función `get` .
- Verás que devuelve el número que guardaste con `set()` .

↺ Puedes cambiar el número varias veces y comprobar que `get()` siempre devuelve el último valor guardado.

◆ Crear un contrato simple: paso a paso

Objetivo

- Guardar un número.
- Poder leer y cambiar el número.
- Avisar cada vez que cambie (evento).

Analogía: Como una caja fuerte digital que avisa con una alarma cada vez que cambias el número guardado.

Código base con evento

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleStorage {
    uint public data;
    address public owner;

    event DataChanged(uint newValue);

    constructor() {
        owner = msg.sender;
    }

    function set(uint _data) public {
        data = _data;
        emit DataChanged(_data);
    }


    function get() public view returns (uint) {
        return data;
    }
}
```

Explicación:

- `data` : número guardado, se puede ver desde fuera.
- `owner` : quién creó el contrato (como la dirección de una casa).
- `event DataChanged` : avisa cuando cambia el número.
- `constructor` : se ejecuta una vez al crear, pone el dueño.
- `set()` : cambia el número y avisa.
- `get()` : devuelve el número, solo lo lee.

¿Dónde vemos el aviso en Remix?

Cuando llamamos a `set()` en nuestro contrato, **se emite un evento** que actúa como un aviso.

1. Ve a la pestaña  **Deploy & Run Transactions**.
2. Llama a `set()` con un número (por ejemplo, 42).
3. Mira en la **parte inferior de Remix**:
 - Aparecerá un bloque con la transacción.
 - Dentro, verás algo como esto:

```
[vm] from: 0x123... | to: SimpleStorage.set(uint256)
logs: {
  "event": "ValueChanged",
  "args": {
    "newValue": "42"
  }
}
```



¿Qué significa esto?

- El contrato emitió un evento `ValueChanged` .
- Es como decir: "📢 ¡He cambiado el valor a 42!"
- Sirve para que otros contratos o interfaces (DApps) puedan escuchar y reaccionar.



Tutorial práctico – Hucha digital con alerta

Vamos a hacer esto paso a paso:

1. Crear un contrato para ahorrar hasta 1 ether.
2. Cuando lleguemos, avisar con un evento.

Paso 1: Estructura del contrato

Primero declaramos las variables:

```
contract Hucha {  
    uint public total; // Cantidad ahorrada hasta ahora  
    uint public objetivo = 1 ether; // Meta de ahorro (1 ether)  
}
```

- `total` : cuánto hemos ahorrado.
- `objetivo` : meta a alcanzar (1 ether).

Paso 2: Evento de aviso

Ahora añadimos un evento para avisar cuando se alcance la meta:

```
event MetaAlcanzada(); // Evento que se emite al llegar al objetivo
```

- Esto es un aviso para cuando llegamos a la meta.

Paso 3: Función de depósito

Ahora creamos la función para depositar dinero:

```
function depositar() public payable { // Permite enviar dinero al contrato
    require(total + msg.value <= objetivo, "Te pasaste!"); // No dejar pasar del objetivo
    total += msg.value; // Suma la cantidad enviada
    if (total == objetivo) { // Si se alcanza la meta
        emit MetaAlcanzada(); // Emite el evento de aviso
    }
}
```

- `msg.value` : dinero enviado.
- `require` : no dejar pasar del objetivo.
- `emit` : avisa si llegamos a la meta.

Paso 4: Código completo

Unimos todo lo anterior en el contrato completo, con comentarios por línea:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Hucha {
    uint public total; // Cantidad ahorrada hasta ahora
    uint public objetivo = 1 ether; // Meta de ahorro (1 ether)



    event MetaAlcanzada(); // Evento que avisa al llegar al objetivo

    function depositar() public payable { // Permite enviar dinero al contrato
        require(total + msg.value <= objetivo, "Te pasaste!"); // No dejar pasar del objetivo
        total += msg.value; // Suma la cantidad enviada
        if (total == objetivo) { // Si se alcanza la meta
            emit MetaAlcanzada(); // Emite el evento de aviso
        }
    }
}
```

Reflexión guiada

- ¿Cómo impedir más depósitos al llegar a la meta?
 - Usar una variable `alcanzado` y bloquear si es `true` .
- ¿Cómo devolver el dinero si no llegamos a la meta?
 - Guardar la fecha y permitir retirar si pasa el tiempo.
- Mejoras de seguridad:
 - Solo el dueño puede cambiar la meta, añadir funciones para recibir dinero seguro.

Vamos a probarlo paso a paso en Remix IDE

1. Copia el código en Remix.
2. Compila con el icono .
3. Despliega con el icono .
4. Llama a `depositar()` enviando dinero.
5. Mira si sale el evento cuando llegas al objetivo.
6. Llama a `total` para ver cuánto hay ahorrado.

Tutorial – Función reset() con control de acceso

Objetivo:

Agregar una función `reset()` para poner el número a 0, solo si eres el dueño.

Código:

Primero explicamos la lógica:



- Solo el dueño puede usar esta función (usamos `require` para comprobarlo).
- Ponemos el número a 0.
- Avisamos con el evento.

Código con comentarios por línea:

Explicación paso a paso

1. Solo el dueño puede usarlo (`require`).
2. Pone el número a 0.
3. Avisa con el evento.

Vamos a probarlo paso a paso en Remix IDE

1. Añade `reset()` al contrato.
2. Compila con .
3. Despliega con .
4. Llama a `reset()` con la cuenta del dueño: funciona.
5. Cambia de cuenta y prueba `reset()`: da error.

Tutorial – Contrato contador

Objetivo:

Contador que suma y resta, pero nunca baja de 0.

Código completo

Primero explicamos los pasos:

- Declaramos la variable contador.
- Creamos eventos para avisar de cambios.
- Hacemos funciones para sumar y restar, evitando negativos.

Código comentado línea a línea:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Counter {
    uint public count; // Contador público


    event CountChanged(uint newCount); // Evento para avisar de cambios

    function increment() public { // Suma 1 al contador
        count += 1; // Incrementa el valor
        emit CountChanged(count); // Avisa del nuevo valor
    }
}
```

Explicación paso a paso

- `increment()` : suma 1 y avisa.
- `decrement()` : solo resta si es mayor que 0.
- `require` : evita que sea negativo.
- Evento: avisa del cambio.

Vamos a probarlo paso a paso en Remix IDE

1. Despliega el contrato con .
2. Pulsa `increment()` varias veces.
3. Pulsa `decrement()` hasta llegar a 0.
4. Si vuelves a pulsar `decrement()`, da error.

Preguntas de comprensión – Buenas prácticas

1. ¿Por qué usar eventos?

- Para avisar cuando algo cambia.

2. ¿Para qué sirven los modificadores?

- Para poner reglas extra a funciones.

3. ¿Por qué comentar el código?

- Para que otros entiendan y evitar errores.

Tutorial – Caja fuerte con contraseña

Objetivo:

Guardar un secreto, solo lo ves si pones la contraseña correcta.

Paso 1 – Declaración del contrato

Primero declaramos el contrato y las variables:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract CajaFuerte {
    bytes32 private passwordHash; // Contraseña guardada en forma segura
    string private secreto; // El secreto guardado
    address public owner; // Dueño del contrato

    constructor(string memory _password, string memory _secreto) { // Se ejecuta al crear el contrato
        passwordHash = keccak256(abi.encodePacked(_password)); // Guarda el hash de la contraseña
        secreto = _secreto; // Guarda el secreto
        owner = msg.sender; // Asigna el dueño
    }
}
```

Explicación:

- `passwordHash` : guarda la contraseña en forma segura (no texto).
- `keccak256` : hace la versión segura de la contraseña.
- `secreto` : lo que queremos ocultar.
- `owner` : quién creó el contrato.

Paso 2 – Función para ver el secreto

Ahora creamos la función para ver el secreto, con comentarios por línea:

```
function verSecreto(string memory _password) public view returns (string memory) {  
    require(  
        keccak256(abi.encodePacked(_password)) == passwordHash, // Compara el hash de la contraseña  
        "Contraseña incorrecta"  
    );  
    return secreto; // Devuelve el secreto si la contraseña es correcta  
}
```

Explicación:

- Compara la contraseña puesta con la guardada (en versión segura).
- Si no es igual, error.
- Si es igual, muestra el secreto.

Vamos a probarlo paso a paso en Remix IDE

1. Despliega el contrato con contraseña `"clave123"` y secreto que quieras.
2. Llama a `verSecreto("clave123")` : debe mostrar el secreto.
3. Llama a `verSecreto("otraClave")` : debe dar error.

Buenas prácticas

- No guardes contraseñas normales, solo versiones seguras.
- Limita los intentos para más seguridad.
- Se puede usar para cajas fuertes, permisos, etc.

◆ Glosario

- **Smart Contract:** Programa autoejecutable en blockchain.
- **Evento:** Registro de acciones para notificar a externos.
- **Gas:** Coste computacional de operaciones en blockchain.
- **Owner:** Dirección que controla permisos especiales.
- **Reentrancy:** Vulnerabilidad que permite llamadas recursivas no autorizadas.
- **View/Pure:** Funciones que no modifican estado (`view` puede leer, `pure` ni lee ni escribe).
- **DAO:** Organización autónoma descentralizada.
- **Blockchain:** Cadena de bloques que almacena información de forma segura y pública.
- **msg.sender:** Dirección que llama a la función en un contrato.

◆ Nuevos ejercicios prácticos

Código completo

Primero explicamos los pasos:

- Guardamos hasta cuándo cada usuario tiene pagada la suscripción.
- Definimos la cuota mensual.
- Creamos funciones para pagar y comprobar si está activa.

Código comentado línea a línea:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Subscription {
    mapping(address => uint) public expirations; // Guarda la fecha de expiración de cada usuario
    uint public monthlyFee = 0.01 ether; // Cuota mensual

    function subscribe() public payable { // Permite pagar la suscripción
        require(msg.value == monthlyFee, "Cuota incorrecta"); // Solo acepta el pago exacto
        expirations[msg.sender] = block.timestamp + 30 days; // Actualiza la fecha de expiración
    }

    function isActive(address user) public view returns (bool) { // Comprueba si la suscripción está activa
        return expirations[user] > block.timestamp; // Devuelve true si no ha expirado
    }
}
```






Explicación paso a paso

- `mapping` : guarda hasta cuándo tienes pagado.
- `subscribe()` : paga y actualiza la fecha.
- `block.timestamp` : fecha actual.
- `isActive()` : ¿sigue activa la suscripción?

Vamos a probarlo paso a paso en Remix IDE

1. Despliega el contrato.
2. Llama a `subscribe()` y pon `0.01 ether` en "Value" antes de pulsar.
3. Llama a `isActive(tu_direccion)` : debe decir `true` .
4. Cambia el tiempo en el código para probar expiración rápido.

Resumen de sección – Conceptos básicos tratados

-  `constructor` : hemos usado esta función para inicializar valores, como el `owner` o el `hash` de una contraseña.
-  `msg.sender` : vimos cómo identificar quién ejecuta una función para restringir el acceso (por ejemplo, a `reset()`).
-  `require` : se ha utilizado para validar condiciones como el pago exacto o evitar decrementos por debajo de cero.
-  `event` : se ha utilizado para notificar cambios de estado, como cuando cambia un valor o el contador.
-  `mapping` : lo usamos para gestionar suscripciones y propietarios de NFTs mediante estructuras clave-valor.



Tutorial – Contrato NFT básico

Objetivo:

Crear NFTs únicos, sin usar librerías externas.

Código completo

Primero explicamos los pasos:

- Contamos cuántos NFTs existen.
- Guardamos quién es el dueño de cada NFT.
- Creamos funciones para crear NFTs y consultar el dueño.

Código comentado línea a línea:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleNFT {
    uint public tokenCount; // Número total de NFTs creados
    mapping(uint => address) public owners; // Dueño de cada NFT

    function mint() public { // Crea un nuevo NFT para quien llama
        tokenCount++; // Suma uno al contador
        owners[tokenCount] = msg.sender; // Asigna el NFT al creador
    }

    function ownerOf(uint tokenId) public view returns (address) { // Consulta el dueño de un NFT
        return owners[tokenId]; // Devuelve la dirección del dueño
    }
}
```

Explicación paso a paso

- `tokenCount` : cuántos NFTs hay.
- `owners` : quién tiene cada NFT.
- `mint()` : crea un NFT nuevo para ti.
- `ownerOf()` : consulta quién es el dueño.

Vamos a probarlo paso a paso en Remix IDE

1. Despliega el contrato.
2. Pulsa `mint()` varias veces.
3. Llama a `ownerOf(1)` , `ownerOf(2)` , etc. para ver el dueño.

Mejoras posibles

- Poner nombre o datos a cada NFT.
- Añadir función para transferir NFTs.
- Usar librerías como OpenZeppelin para más funciones.

Fin de la sesión

-  Historia y fundamentos de Smart Contracts.
-  Diferencias entre blockchains y elementos clave.
-  Desafíos técnicos ampliados.
-  Práctica con Remix IDE sin dependencias.
-  Retos prácticos con eventos y control de acceso.
- En la próxima clase: integración de contratos con frontend Web3.

◆ Ideas para proyectos finales

1. **Aplicación de votación segura** con opción pública/privada y gestión de votos.
2. **Sistema de becas descentralizadas** donde los pagos se liberan por hitos.
3. **Marketplace de productos digitales** con pagos en cripto y entrega automática.
4. **Juego basado en blockchain** (e.g., dados, lotería, cartas coleccionables).
5. **Gestión de certificados educativos** como NFTs verificables.