Building DeFi

Lesson 11: From Token Swaps to Escrow Contracts

Prof. Joerg Osterrieder

Spring 2026

Learning Objectives

After this lesson, you will be able to:

1. Build an Automated Market Maker (AMM) DEX contract from scratch
2. Implement a trustless escrow contract with time-locks
3. Understand AMM pricing math (constant product formula $x \cdot y = k$)
4. Deploy and test multi-contract DeFi systems in Remix IDE

Prerequisites: Lessons 1–7 (especially L06: DeFi, L07: Smart Contracts) and Lesson 10 (Solidity Basics)

What you already know from L10:

- Solidity syntax: `pragma`, `contract`, `function`, `mapping`, `struct`, `modifier`
- ERC-20 token standard: `transfer`, `approve`, `transferFrom`
- Remix IDE: compile, deploy, interact

L10

taught how to write tokens and voting contracts — this lesson combines tokens into DeFi protocols

Lesson Outline

1. DeFi Building Blocks
2. Building a Simple DEX
3. Building an Escrow Contract
4. Testing and Security
5. The DeFi Ecosystem

DeFi Building Blocks

Recap: What We Know

Three lessons converge today:

Lesson 6 — DeFi Concepts	$\Rightarrow$	AMMs, liquidity pools, yield farming (*what* they do)
Lesson 7 — Smart Contracts	$\Rightarrow$	Security, reentrancy, game theory (*why* they matter)
Lesson 10 — Solidity Basics	$\Rightarrow$	ERC-20, mappings, modifiers (*how* to write code)
Lesson 11 — Today	$\Rightarrow$	**Combine all three: build working DeFi contracts**

Today's two projects:

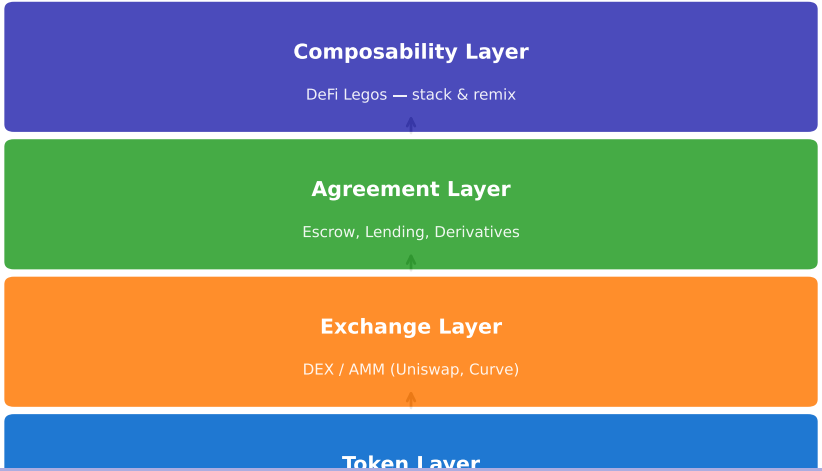
1. **SimpleDEX** — A decentralized exchange using the constant product formula ($x \cdot y = k$)
2. **Escrow** — A trustless payment contract with three roles and time-locks

Tools: Remix IDE only — same environment as L10. No external frameworks.

protocols are just smart contracts that compose together — today you build your own

DeFi Building Blocks

Layered architecture — each layer builds on the one below



The Constant Product Formula

Recap from L06: Automated Market Makers (AMMs) replace order books with a mathematical formula.

$$\boxed{x \cdot y = k}$$

Where:

- x = reserve of Token A in the pool
- y = reserve of Token B in the pool
- k = constant product (stays the same after every swap)

Numerical example: Pool starts with 100 Token A and 100 Token B, so $k = 100 \times 100 = 10{,}000$.
A trader deposits 10 Token A to buy Token B:

1. New reserve A: $100 + 10 = 110$
2. Solve for new reserve B: $110 \times y_{\text{new}} = 10{,}000 \Rightarrow y_{\text{new}} = 90.91$
3. Trader receives: $100 - 90.91 = \mathbf{9.09}$ Token B (not 10!)

Price impact: The larger the swap relative to reserves, the worse the rate. This is **slippage**.

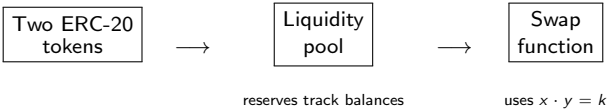
Unisv

V2 uses exactly this formula — we will implement it step by step over the next 10 slides

Building a Simple DEX

DEX Architecture

What our SimpleDEX contract needs:



Contract features:

1. Store references to two ERC-20 token contracts
2. Track reserves (how many of each token the DEX holds)
3. **Add liquidity:** deposit both tokens to fill the pool
4. **Swap:** trade one token for the other using the constant product formula
5. **Slippage protection:** ensure traders get a minimum amount out

How tokens move:

- Traders call `approve(DEX_address, amount)` on the token contract first (from L10)
- Then the DEX calls `transferFrom(trader, DEX, amount)` to pull tokens in
- The DEX calls `transfer(trader, amount)` to send tokens out

The approve + transferFrom pattern from L10 is the foundation of every DeFi protocol

Step 1: What is an Interface?

Before writing the DEX, we need a way to *call* ERC-20 token contracts:

```
interface IERC20 {                          ← interface: describes functions without code
    function totalSupply()
        external view returns (uint256);    ← external: callable only from outside
    function balanceOf(address account)
        external view returns (uint256);
    function transfer(address to, uint256 amount)
        external returns (bool);
    function approve(address spender, uint256 amount)
        external returns (bool);
    function transferFrom(address from, address to,
        uint256 amount) external returns (bool);
}
```

New keywords:

- `interface` — declares *what functions exist* on another contract, but contains no implementation. Think of it as a contract's "menu" — it tells you what you can order, not how the kitchen makes it.
- `external` — visibility modifier meaning "can only be called from outside the contract" (not from within). Interface functions are always `external`.

Why IERC20? Our DEX will hold `IERC20` references to call `transfer` and `transferFrom` on any ERC-20 token — without needing the token's full source code.

Interf

enable cross-contract calls — a contract can interact with any token that follows ERC-20

Step 2: Contract Shell + Tokens

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

// IERC20 interface defined above (or in a separate file)

contract SimpleDEX {
    IERC20 public tokenA;                    ← reference to first ERC-20 token contract
    IERC20 public tokenB;                    ← reference to second ERC-20 token contract
    uint256 public reserveA;                 ← how many tokenA the DEX holds
    uint256 public reserveB;                 ← how many tokenB the DEX holds

    constructor(address _tokenA, address _tokenB) {   ← takes token addresses at deployment
        tokenA = IERC20(_tokenA);            ← wrap address as IERC20 interface
        tokenB = IERC20(_tokenB);
    }
}
```

How it works:

- `IERC20 public tokenA` — stores a reference to a deployed ERC-20 contract. The type `IERC20` means "any contract that has these functions."
- `IERC20(_tokenA)` — casts a raw `address` to the `IERC20` interface, enabling us to call `tokenA.transfer(...)`, `tokenA.transferFrom(...)`, etc.
- `reserveA` / `reserveB` — track how many tokens the DEX holds internally (used for the $x \cdot y = k$ calculation).

At deployment, you pass the addresses of two already-deployed ERC-20 tokens — the DEX does not create tokens

Step 3: Adding Liquidity

```
event LiquidityAdded(                          ← log when someone adds liquidity
    address indexed provider,
    uint256 amountA, uint256 amountB
);

function addLiquidity(uint256 _amountA, uint256 _amountB)
    external                                   ← external:  called by users, not internally
{
    tokenA.transferFrom(                       ← pull tokenA from caller into DEX
        msg.sender, address(this), _amountA
    );
    tokenB.transferFrom(                       ← pull tokenB from caller into DEX
        msg.sender, address(this), _amountB
    );

    reserveA += _amountA;                      ← update internal reserve tracking
    reserveB += _amountB;

    emit LiquidityAdded(msg.sender, _amountA, _amountB);
}
```

Key points:

- The caller must first `approve(DEX_address, amount)` on *both* token contracts (L10 pattern)
- `address(this)` — the DEX contract's own address (the destination for the tokens)
- `transferFrom` pulls tokens from the caller into the DEX — this is the cross-contract call enabled by our `IERC20` interface

Liqui

providers deposit equal value of both tokens — they earn swap fees in real DEXes

Step 4: The Swap Function

```
event Swap(                                    ← log every swap
    address indexed trader,
    address tokenIn, uint256 amountIn,
    address tokenOut, uint256 amountOut
);

function swap(address _tokenIn, uint256 _amountIn,
    uint256 _minOut) external                  ← _minOut = slippage protection (Slide 15)
{
    require(                                   ← check:  must be one of our two tokens
        _tokenIn == address(tokenA) ||
        _tokenIn == address(tokenB),
        "Invalid token"
    );

    bool isA = (_tokenIn == address(tokenA));
    (IERC20 tokenIn, IERC20 tokenOut,          ← assign input/output token references
     uint256 reserveIn, uint256 reserveOut) =
        isA
        ? (tokenA, tokenB, reserveA, reserveB)
        : (tokenB, tokenA, reserveB, reserveA);

    // ... continued on next slide
}
```

Structure: The swap function first validates the input token, then identifies which direction the swap goes (A→B or B→A). The actual price calculation follows on the next slide.

A

single `swap` function handles both directions — the `isA` flag determines which reserves to use

Step 5: Price Calculation

Continuing inside the swap function:

```
    // Pull tokens in from the trader
    tokenIn.transferFrom(msg.sender, address(this), _amountIn);

    // Calculate output using x * y = k
    uint256 amountOut =                          ← the constant product formula
        (reserveOut * _amountIn) /               ← integer division (no decimals!)
        (reserveIn + _amountIn);

    require(amountOut >= _minOut,                 ← slippage check (Slide 15)
            "Slippage too high");

    // Send output tokens to the trader
    tokenOut.transfer(msg.sender, amountOut);

    // Update reserves
    reserveA = tokenA.balanceOf(address(this));  ← sync reserves with actual balances
    reserveB = tokenB.balanceOf(address(this));
```

The formula step by step (swap 10 tokenA, reserves 100/100, $k = 10{,}000$):

```
amountOut = (100 * 10) / (100 + 10) = 1000 / 110 = 9
```

Important: Solidity has **no floating-point numbers**. The division $1000 \div 110 = 9.09$ is truncated to **9**. Integer division always rounds *down*, which means the pool keeps the rounding dust.

Integ

division is why DeFi uses 18-decimal tokens — it minimizes rounding loss on large amounts

Step 6: Slippage Protection

Problem: Between the moment you submit a swap and when it executes, the reserves may change (another trader swaps first). Your expected output could decrease.

Solution: The `_minOut` parameter:

```
function swap(address _tokenIn, uint256 _amountIn,
    uint256 _minOut) external {              ← caller specifies minimum acceptable output
    ...
    uint256 amountOut = (reserveOut * _amountIn)
                        / (reserveIn + _amountIn);

    require(amountOut >= _minOut,             ← revert if output is too low
            "Slippage too high");
    ...
}
```

Numerical example:

Scenario	amountOut	Result (_minOut = 8)
No front-runner	9	$9 \geq 8$: swap succeeds
Front-runner swaps first	7	$7 < 8$: transaction **reverts**

Connection to L07: This is a code-level defense against *front-running* and *MEV* (Miner Extractable Value) — concepts we discussed in L07's security section.

Alway

set _minOut when swapping — without it, a front-runner can drain your trade's value

Step 7: Events + Liquidity Tokens

Adding LP (Liquidity Provider) share tracking:

```
mapping(address => uint256) public lpShares; ← track each provider's share
uint256 public totalShares;                  ← total shares issued

function addLiquidity(uint256 _amountA, uint256 _amountB)
    external
{
    // ... transferFrom calls (Slide 12) ...

    uint256 shares;
    if (totalShares == 0) {                  ← first deposit:  shares = amountA
        shares = _amountA;
    } else {                                 ← subsequent:  proportional to deposit
        shares = (_amountA * totalShares) / reserveA;
    }

    lpShares[msg.sender] += shares;          ← credit shares to provider
    totalShares += shares;

    reserveA += _amountA;
    reserveB += _amountB;

    emit LiquidityAdded(msg.sender, _amountA, _amountB);
}
```

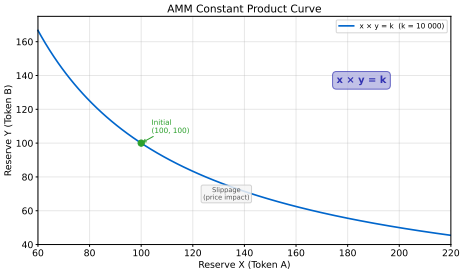
LP share math: If the pool has 100 tokenA with 50 total shares, depositing 20 tokenA earns $20 \times 50/100 = 10$ shares $= 20\%$ of the pool.

Real

DEXes (Uniswap) mint ERC-20 LP tokens instead of internal shares — same math, more composable

The Complete SimpleDEX

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

interface IERC20 {
    function totalSupply() external view returns (uint256);
    function balanceOf(address) external view returns (uint256);
    function transfer(address, uint256) external returns (bool);
    function approve(address, uint256) external returns (bool);
    function transferFrom(address, address, uint256)
        external returns (bool);
}

contract SimpleDEX {
    IERC20 public tokenA;
    IERC20 public tokenB;
    uint256 public reserveA;
    uint256 public reserveB;
    mapping(address => uint256) public lpShares;
    uint256 public totalShares;

    event Swap(address indexed trader, address tokenIn,
        uint256 amountIn, address tokenOut, uint256 amountOut);
    event LiquidityAdded(address indexed provider,
        uint256 amountA, uint256 amountB);

    constructor(address _tokenA, address _tokenB) {
        tokenA = IERC20(_tokenA);
        tokenB = IERC20(_tokenB);
    }

    function addLiquidity(uint256 _amountA, uint256 _amountB)
        external {
        tokenA.transferFrom(msg.sender, address(this), _amountA);
        tokenB.transferFrom(msg.sender, address(this), _amountB);
```



SimpleDEX Checklist:

- ✓ Two-token pool
- ✓ Add liquidity
- ✓ Constant product swap
- ✓ Slippage protection
- ✓ LP share tracking
- ✓ Event logging

Deployment order:

DEX in Action: Walkthrough

Step-by-step in Remix (start with reserves 100 / 100, $k = 10{,}000$):

Tx#	Action	Reserve A	Reserve B	k	Price (A/B)
0	Add liquidity: 100A + 100B	100	100	10,000	1.00
1	Swap 10A $\rightarrow$ B (get 9B)	110	91	10,010	1.21
2	Swap 10A $\rightarrow$ B (get 7B)	120	84	10,080	1.43
3	Swap 10B $\rightarrow$ A (get 12A)	108	94	10,152	1.15

Key observations:

- **Price impact:** Swap #1 gets 9.09 B per 10 A; swap #2 gets only 7.58 B per 10 A — the price moves against you with each trade
- k **increases slightly** because integer rounding always favors the pool (rounding dust stays in reserves)
- **Arbitrage opportunity:** If another market prices A/B at 1.0, a trader would buy cheap A on our DEX until prices equalize

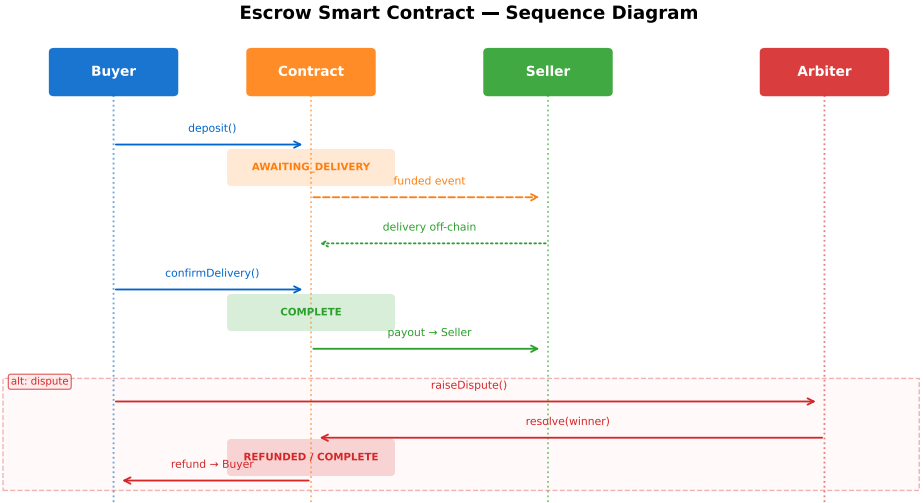
Try it yourself: Deploy two MyToken contracts (from L10), deploy SimpleDEX, add liquidity, and perform swaps. Watch reserves and prices change in real time.

Price discovery through arbitrage is how AMMs track the "real" market price — no oracle needed

Building an Escrow Contract

What is Escrow?

Real-world analogy: When buying a house, the buyer deposits money with a neutral third party (escrow agent). The money is released to the seller only when the deed transfers.



Step 1: State + Roles

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract Escrow {
    address public buyer;                   ← who deposits the payment
    address public seller;                  ← who receives the payment
    address public arbiter;                 ← who resolves disputes
    uint256 public amount;                  ← expected deposit amount (in wei)

    enum State {                            ← enum:  named set of possible states
        AWAITING_PAYMENT,                   ← 0:  contract created, waiting for deposit
        AWAITING_DELIVERY,                  ← 1:  buyer paid, waiting for delivery
        COMPLETE,                           ← 2:  buyer confirmed, seller paid
        REFUNDED                            ← 3:  arbiter refunded the buyer
    }

    State public currentState;              ← tracks the escrow's current status
}
```

New keyword — `enum`:

- An `enum` defines a set of **named constants** stored as integers (0, 1, 2, 3)
- Much more readable than using raw numbers: `State.AWAITING_DELIVERY` vs `1`
- Once set, the state can only be one of these four values — the compiler enforces this

---

`enum` **is ideal for state machines — any process with distinct stages (orders, auctions, escrows)**

Step 2: Deposit (Receiving ETH)

```
constructor(
    address _buyer, address _seller,
    address _arbiter, uint256 _amount
) {
    buyer = _buyer;
    seller = _seller;
    arbiter = _arbiter;
    amount = _amount;
    currentState = State.AWAITING_PAYMENT;    ← initial state
}

function deposit() external payable {         ← payable: function can receive ETH
    require(msg.sender == buyer,              ← only the buyer can deposit
            "Only buyer");
    require(msg.value == amount,              ← msg.value: amount of ETH sent (in wei)
            "Wrong amount");
    require(currentState == State.AWAITING_PAYMENT,
            "Already funded");

    currentState = State.AWAITING_DELIVERY;   ← advance the state machine
}
```

New keywords:

- `payable` — a modifier that allows a function to receive ETH. Without `payable`, sending ETH to this function would revert. Applies to functions *and* addresses.
- `msg.value` — a built-in global variable containing the amount of ETH (in wei) sent with the current transaction. 1 ETH $= 10^{18}$ wei.

What happens: The buyer calls `deposit()` and attaches exactly `amount` wei. The ETH is now held *inside the contract* — neither buyer nor seller can withdraw it unilaterally.

msg.v

Step 3: Confirm Delivery

```
event DeliveryConfirmed(uint256 amount);
event Refunded(uint256 amount);

function confirmDelivery() external {
    require(msg.sender == buyer,              ← only the buyer can confirm
            "Only buyer");
    require(currentState == State.AWAITING_DELIVERY,
            "Not awaiting delivery");

    currentState = State.COMPLETE;            ← advance state before transfer (CEI pattern)

    payable(seller).transfer(                 ← .transfer():  send ETH to an address
        address(this).balance                ← address(this).balance:  ETH held by contract
    );

    emit DeliveryConfirmed(amount);
}
```

New keywords:

- `address(this).balance` — returns the total ETH (in wei) currently held inside this contract
- `payable(seller).transfer(amount)` — sends `amount` wei to the seller's address. The `payable()` cast is required because `.transfer()` only works on payable addresses. Reverts if the send fails.

Checks-Effects-Interactions (from L10):

1. **Check:** `require` validates caller and state
2. **Effect:** Update `currentState` to `COMPLETE`
3. **Interact:** Transfer ETH to seller

State is updated BEFORE the transfer — this prevents reentrancy (the seller's fallback cannot re-enter)

Step 4: Dispute + Refund

```
function refund() external {
    require(msg.sender == arbiter,            ← only the arbiter can refund
            "Only arbiter");
    require(currentState == State.AWAITING_DELIVERY,
            "Not awaiting delivery");         ← can only refund before completion

    currentState = State.REFUNDED;            ← effect: update state first

    payable(buyer).transfer(                  ← return all ETH to the buyer
        address(this).balance
    );

    emit Refunded(amount);
}
```

The arbiter's role:

- The arbiter is a trusted third party (could be a DAO, a multisig, or even another smart contract)
- They can only act when the escrow is in `AWAITING_DELIVERY` state
- Once the buyer confirms delivery, the arbiter *cannot* reverse it — `COMPLETE` is final

State transitions so far:

$$\texttt{AWAITING\_PAYMENT} \xrightarrow{\text{deposit}} \texttt{AWAITING\_DELIVERY} \xrightarrow[\text{refund}]{\text{confirm}} \texttt{COMPLETE} \text{ or } \texttt{REFUNDED}$$

In a real system, the arbiter might require evidence (off-chain) before issuing a refund

Step 5: Time-Locks

```
uint256 public deadline;                        ← expiration timestamp

constructor(
    address _buyer, address _seller,
    address _arbiter, uint256 _amount,
    uint256 _durationDays                       ← escrow duration in days
) {
    // ... role assignments (Slide 22) ...
    deadline = block.timestamp                  ← block.timestamp:  current time (Unix seconds)
             + (_durationDays * 1 days);        ← Solidity unit:  1 days = 86400 seconds
}

function claimExpired() external {
    require(msg.sender == buyer,
            "Only buyer");
    require(block.timestamp >= deadline,        ← check:  has the deadline passed?
            "Not yet expired");
    require(currentState == State.AWAITING_DELIVERY,
            "Wrong state");

    currentState = State.REFUNDED;
    payable(buyer).transfer(address(this).balance);
    emit Refunded(amount);
}
```

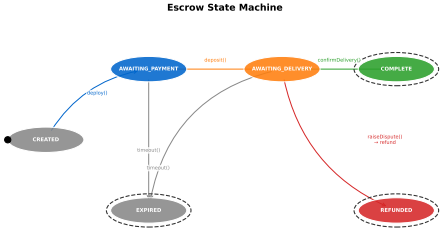
New keyword — `block.timestamp`:

- Returns the current block's timestamp as a Unix epoch (seconds since Jan 1, 1970)
- If the seller never delivers, the buyer can reclaim funds after the deadline — no arbiter needed

can be slightly manipulated by miners (±15 seconds) — do not use for precision timing

The Complete Escrow Contract

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.20;
3
4  contract Escrow {
5      address public buyer;
6      address public seller;
7      address public arbiter;
8      uint256 public amount;
9      uint256 public deadline;
10
11     enum State { AWAITING_PAYMENT, AWAITING_DELIVERY,
12                  COMPLETE, REFUNDED }
13     State public currentState;
14
15     event DeliveryConfirmed(uint256 amount);
16     event Refunded(uint256 amount);
17
18     constructor(address _buyer, address _seller,
19         address _arbiter, uint256 _amount,
20         uint256 _durationDays) {
21         buyer = _buyer; seller = _seller;
22         arbiter = _arbiter; amount = _amount;
23         deadline = block.timestamp + (_durationDays * 1 days);
24         currentState = State.AWAITING_PAYMENT;
25     }
26
27     function deposit() external payable {
28         require(msg.sender == buyer, "Only buyer");
29         require(msg.value == amount, "Wrong amount");
30         require(currentState == State.AWAITING_PAYMENT,
31                 "Already funded");
32         currentState = State.AWAITING_DELIVERY;
33     }
34
```



State transitions:

- AWAITING_PAYMENT $\xrightarrow{\text{deposit()}}$ AWAITING_DELIVERY
- AWAITING_DELIVERY $\xrightarrow{\text{confirmDelivery()}}$ COMPLETE
- AWAITING_DELIVERY $\xrightarrow{\text{refund() / claimExpired()}}$ REFUNDED

Game Theory Connection

Why does escrow work? Let us analyze rational strategies (connecting to L07):

Scenario	Without Escrow	With Escrow
Seller delivers, buyer pays	Best case	Same (COMPLETE)
Seller does not deliver	Buyer loses ETH	Arbiter refunds buyer
Buyer refuses to confirm	—	Seller appeals to arbiter
Both disappear	Funds lost	Time-lock auto-refunds

Escrow as a commitment device (from L07):

- The buyer *commits* to paying by locking ETH in the contract
- The seller knows the payment is guaranteed if they deliver
- The arbiter provides a credible threat of refund, incentivizing honest behavior
- The time-lock ensures no funds are permanently locked

Nash Equilibrium: Both parties cooperate (deliver and confirm) because:

- Buyer's best response: confirm delivery $\rightarrow$ get goods
- Seller's best response: deliver $\rightarrow$ get paid
- Defecting (not delivering / not confirming) triggers arbiter or time-lock

Testing and Security

Common Solidity Pitfalls

Revisiting L07 vulnerabilities — now with code-level fixes:

Vulnerability	What Goes Wrong	Code-Level Fix
Reentrancy (L07)	External call re-enters your function before state updates	Update state *before* external calls (CEI pattern)
Integer overflow (pre-0.8)	`uint8(255) + 1 = 0`	Solidity 0.8+ auto-reverts on overflow
Unchecked return value	`transfer()` fails silently	Use `require(token.transfer(...))`
Front-running (L07)	Miner reorders transactions for profit	Add `_minOut` parameter (slippage)
Timestamp dependence	Miner manipulates `block.timestamp`	Use for coarse checks only ($\pm$15 sec tolerance)
Missing access control	Anyone can call admin functions	Use `modifier onlyOwner` (L10)

Our contracts already handle most of these:

- SimpleDEX: slippage protection (`_minOut`), CEI pattern
- Escrow: access control (`require(msg.sender == buyer)`), state machine prevents re-entry

is not optional — a single vulnerability can drain all funds from a contract forever

The Checks-Effects-Interactions Pattern

BAD: Interaction before effect (vulnerable to reentrancy)

```
// DANGEROUS - do NOT write code like this!
function withdraw() external {
    uint256 bal = balances[msg.sender];
    require(bal > 0, "No balance");

    payable(msg.sender).transfer(bal);        ← INTERACT: sends ETH (attacker can re-enter!)
    balances[msg.sender] = 0;                ← EFFECT: too late --- attacker already re-entered
}
```

GOOD: Effect before interaction (safe)

```
// SAFE - Checks-Effects-Interactions pattern
function withdraw() external {
    uint256 bal = balances[msg.sender];       ← CHECK: read the balance
    require(bal > 0, "No balance");           ← CHECK: validate

    balances[msg.sender] = 0;                ← EFFECT: zero balance BEFORE sending

    payable(msg.sender).transfer(bal);        ← INTERACT: safe --- re-entry finds zero balance
}
```

Why this works: If the recipient is a malicious contract that re-enters `withdraw()`, it finds `balances[msg.sender] = 0` and the `require` reverts.

Our escrow uses this pattern: `currentState = State.COMPLETE` (effect) happens *before* `.transfer()` (interaction) on Slide 23.

CEI

is the single most important security pattern in Solidity — the 2016 DAO hack ($60M lost) exploited this exact flaw

Testing in Remix

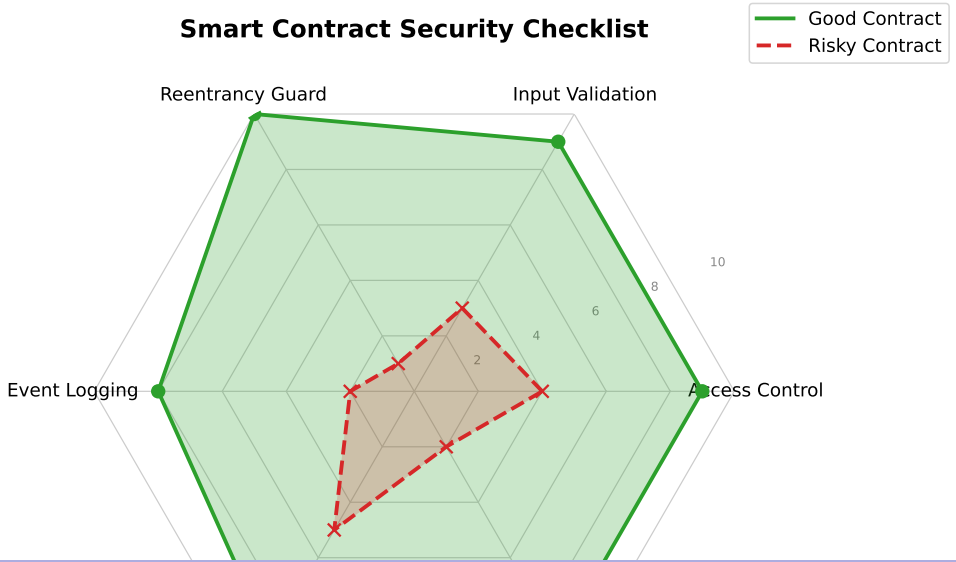
Remix built-in testing (Solidity Unit Testing plugin):

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "remix_tests.sol";                    ← Remix testing library
import "../contracts/MyToken.sol";           ← import the contract to test

contract MyTokenTest {
    MyToken token;

    function beforeEach() public {           ← runs before each test
        token = new MyToken(1000);           ← deploy fresh contract
    }

    function testInitialBalance() public {   ← test function (prefix: ''test'')
        Assert.equal(                        ← check expected value
            token.balanceOf(address(this)),
            1000, "Should have 1000 tokens"
        );
    }

    function testTransfer() public {
        token.transfer(address(0x1), 100);
        Assert.equal(token.balanceOf(address(this)), 900,
            "Sender should have 900");
    }
}
```

How to run: Click the "Solidity Unit Testing" plugin in Remix → select test file → click "Run Tests." Green = pass, red = fail.

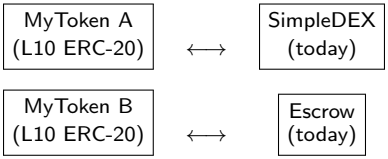
Alway

test your contracts before deployment — once deployed, smart contracts cannot be changed


Smart Contract Security Checklist
Good Contract
Risky Contract
Reentrancy Guard
Input Validation
Access Control
Event Logging
2
4
6
8
10

Composability: Money Legos

How our contracts connect into a mini DeFi stack:



What we built across L10 and L11:

- **ERC-20 Tokens** — the universal building block (L10)
- **SimpleDEX** — swap any two tokens using $x \cdot y = k$ (today)
- **Escrow** — lock ETH until conditions are met (today)

Real-world protocols built on the same patterns:

Pattern	Our Contract	Real Protocol	TVL
AMM swap	SimpleDEX	Uniswap	\$5B+
Escrow	Escrow	OpenSea, Kleros	\$1B+
Lending	(next course)	Aave, Compound	\$10B+

Every major DeFi protocol uses the same building blocks you learned in L10 and L11 — just with more features and audits

The DeFi Ecosystem

Visual Break

From Classroom to Mainnet

What production DeFi code adds beyond our contracts:

Feature	Our Version	Production Version
Security	Basic CEI + require	Professional audit ($50k–$500k)
Upgradeability	None (immutable)	Proxy patterns (L07)
Gas optimization	Not considered	Assembly-level optimizations
Verification	Manual Remix testing	Formal verification (Certora, etc.)
Governance	Owner-controlled	DAO voting (L09)

Resources for continued learning:

- **Solidity docs:** https://docs.soliditylang.org
- **CryptoZombies:** https://cryptozombies.io — learn Solidity by building a game
- **Ethernaut:** https://ethernaut.openzeppelin.com — security challenges
- **Scaffold-ETH:** https://scaffoldeth.io — full-stack dApp development

Career paths: Smart contract developer, security auditor, DeFi protocol designer, MEV researcher

The gap between classroom contracts and production DeFi is auditing and testing — the Solidity is the same

Key Takeaways + Questions

What you learned today:

1. **AMM mechanics:** Constant product formula $x \cdot y = k$, price impact, slippage protection
2. **Escrow with time-locks:** State machines with `enum`, receiving ETH with `payable`/`msg.value`, automatic refund via `block.timestamp`
3. **Security patterns:** Checks-Effects-Interactions, common pitfalls, 10-point security checklist
4. **Composability:** ERC-20 tokens + DEX + Escrow = a mini DeFi stack using interfaces for cross-contract calls

Questions for reflection:

1. Why does integer division in the AMM formula always favor the pool (not the trader)?
2. What would happen if the escrow had no time-lock and both buyer and arbiter disappeared?
3. How would you add a fee (e.g., 0.3%) to the SimpleDEX swap function?
4. Why is the `interface` pattern important for DeFi composability?

Course complete! You have built DeFi from scratch — from concepts (L06–L07) to code (L10–L11).

materials: digital-ai-finance.github.io/crypto-economics