Building DeFi

Lesson 11: Summary

Prof. Joerg Osterrieder

Spring 2026

L11 Summary: Building DeFi

Four sections covered today:

1. **DeFi Building Blocks** — Layers, composability, and the constant product formula $x \cdot y = k$
2. **SimpleDEX** — Interface pattern, reserve tracking, `addLiquidity`, and `swap`
3. **Price Calculation** — AMM `amountOut` formula, slippage protection, numerical examples
4. **Escrow Contract** — `enum` state machine, `payable`/`msg.value`, time-locks, dispute resolution
5. **Testing and Security** — Checks-Effects-Interactions, common pitfalls, 10-point security checklist

Key shift from previous lessons:

- L10 taught *how to write* individual Solidity contracts
- L11 asks *how to compose* those contracts into working DeFi protocols

---

L11 = L06 (DeFi concepts) + L07 (smart contract security) + L10 (Solidity) combined into two working contracts

Section 1: DeFi Building Blocks

DeFi = composable smart contracts — each layer builds on the one below:

Layer	What it does	Example
Layer 1: Tokens	Base building block; ERC-20 standard	MyToken (L10)
Layer 2: Protocols	Swap, lock, or lend tokens	SimpleDEX, Escrow (today)
Layer 3: Aggregators	Combine multiple protocols	1inch routing across DEXes

The constant product formula — the heart of every AMM:

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

- x = reserve of Token A; y = reserve of Token B; k = constant product
- k stays fixed after every swap — determines how much output a trader receives
- Larger swap $\Rightarrow$ worse rate (price impact / slippage)

Why composability works: Every ERC-20 token exposes the same `approve` + `transferFrom` interface. Any protocol can move any token without knowing its internal logic.

"Money Legos" — DeFi protocols snap together because they share the ERC-20 interface standard

Three building blocks of the DEX:

1. IERC20 Interface

- Declares *what functions exist* on any ERC-20 without needing its source code
- `interface` keyword — no implementation, only signatures
- `external` visibility — callable only from outside
- Enables cross-contract calls: `tokenA.transferFrom(...)`

2. Reserve Tracking

- `reserveA`, `reserveB` — internal state used for $x \cdot y = k$
- Updated after every swap via `balanceOf(address(this))`
- `lpShares` mapping tracks each provider's proportional stake

3. Core Functions

Function	Purpose
`addLiquidity`	Deposit both tokens into pool
`swap`	Trade one token for the other

addLiquidity flow:

1. Caller: `approve(DEX, amount)` on both tokens
2. DEX: `transferFrom(caller, this, amount)` for both
3. DEX: Update `reserveA`, `reserveB`, `lpShares`
4. Emit `LiquidityAdded` event

The DEX does not create tokens — it is a pool contract that holds and swaps tokens deposited by liquidity providers

Section 3: Price Calculation

The `amountOut` formula (derived from $x \cdot y = k$):

$$\texttt{amountOut} = \frac{\texttt{reserveOut} \times \texttt{amountIn}}{\texttt{reserveIn} + \texttt{amountIn}}$$

Numerical example (swap 10 Token A, starting reserves 100/100, $k = 10{,}000$):

Step	Reserve A	Reserve B	amountOut B	Note
Initial	100	100	—	$k = 10{,}000$
Swap 10 A	110	—	$(100 \times 10)/(100 + 10) = 9$	Integer division
After swap	110	91	—	$k \approx 10{,}010$ (rounding dust)

Slippage protection — the `_minOut` parameter:

- Caller sets minimum acceptable output before submitting the transaction
- `require(amountOut >= _minOut, "Slippage too high")` reverts if reserves moved
- Defends against front-running (MEV) — if another swap reduces output, the transaction reverts

Integer division note: Solidity has no floating-point. $1000 \div 110 = 9$ (truncates, not rounds). DeFi uses 18-decimal tokens to minimise this rounding error.

Without _minOut, a front-runner can manipulate reserves between your submission and execution

Section 4: Escrow Contract

Key Solidity concepts introduced in the Escrow:

`enum` — **named state machine:**

```
enum State {
    AWAITING_PAYMENT,   // 0
    AWAITING_DELIVERY,  // 1
    COMPLETE,           // 2
    REFUNDED            // 3
}
```

State transitions:

AWAITING_PAYMENT
—deposit()→ AWAITING_DELIVERY
—confirmDelivery()→ COMPLETE
—refund() / claimExpired()→ REFUNDED

New keywords:

- `payable` — function/address can receive ETH
- `msg.value` — ETH sent with the call (in wei)
- `address(this).balance` — ETH held by the contract
- `payable(addr).transfer()` — send ETH to an address
- `block.timestamp` — current Unix time (seconds)
- `1 days` — Solidity time unit (= 86,400 seconds)

Time-lock: If the seller never delivers, the buyer calls `claimExpired()` after `deadline` to auto-refund — no arbiter needed.

The `enum` prevents invalid state transitions — the compiler enforces that only the four defined states are possible

Checks-Effects-Interactions (CEI) — the most important Solidity security pattern:

Step	Action	Why
1	**Checks**	Validate all inputs with `require` before touching state
2	**Effects**	Update contract state (balances, flags)
3	**Interactions**	Call external contracts *last*

6-item security checklist (from the full 10-point list):

- ✓ Reentrancy: CEI pattern applied
- ✓ Access control on all admin functions
- ✓ Input validation (`require`) on all entry points
- ✓ Integer overflow (use Solidity ≥ 0.8)
- ✓ Slippage protection on all swaps
- ✓ Events emitted for all state changes

Common pitfalls covered:

- Reentrancy (2016 DAO hack: \$60M), unchecked return values, front-running, timestamp manipulation
- Fix: update state *before* external calls; use `_minOut`; treat `block.timestamp` as coarse ($\pm$15 sec)

CEI is not optional — the DAO hack exploited exactly the pattern of interacting before updating state

Key Formulas and Course Conclusion

Key Formulas from L11:

Constant product invariant:

$$x \cdot y = k$$

AMM output amount:

$$\texttt{amountOut} = \frac{\texttt{reserveOut} \times \texttt{amountIn}}{\texttt{reserveIn} + \texttt{amountIn}}$$

LP share allocation:

$$\texttt{shares} = \frac{\texttt{amountA} \times \texttt{totalShares}}{\texttt{reserveA}}$$

Escrow state machine:

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

What you built across L10 and L11:

Contract	Lesson	Real-world
ERC-20 Token	L10	Any fungible token
SimpleDEX	L11	Uniswap V2
Escrow	L11	OpenSea, Kleros

Token + DEX + Escrow = a mini DeFi stack.

Every major DeFi protocol uses these same patterns. Production code adds:

- Professional security audits
- Gas optimisation
- DAO governance
- Proxy upgradeability

Course complete: you have built DeFi from concepts (L06–L07) to working Solidity code (L10–L11)

Thank You

Questions?