Create Your Own Cryptocurrency

Lesson 4 of 4: ERC-20 Token Creation

Building Fungible Tokens on Ethereum

From Token Standards to Deployment

Lesson Outline

Duration: 45 minutes — **Prerequisite:** Lessons 1-3 (Blockchain, Cryptography, Smart Contracts)

Section 1: Token Standards

Understanding the building blocks of digital assets

What is a Token?

Tokens vs. Coins
Coins (Native Currency):

- Have their own blockchain
- Used for transaction fees
- Examples: ETH, BTC, SOL

Tokens (Smart Contract Assets):

- Built *on top of* existing blockchains
- Created via smart contracts
- Examples: USDC, LINK, UNI

Token Use Cases

- **Utility:** Access to services (API credits)
- **Governance:** Voting rights (DAO tokens)
- **Security:** Represent ownership shares
- **Stablecoin:** Pegged to real assets
- **NFT:** Unique digital collectibles
- **Gaming:** In-game currencies

Key Insight:
Tokens are programmable money with custom rules encoded in smart contracts.

Over 500,000 tokens exist on Ethereum alone, demonstrating the power of token standards

Why Standards Matter

Without Standards

- × Every token has different functions
- × Wallets need custom code per token
- × Exchanges can't easily list tokens
- × DeFi protocols can't be composable
- × Security audits are complex

The Problem:
Imagine if every website had different HTTP methods!

With Standards (ERC-20)

- ✓ Consistent interface for all tokens
- ✓ One wallet supports all tokens
- ✓ Easy exchange integration
- ✓ DeFi composability ("money legos")
- ✓ Audited reference implementations

The Solution:
ERC-20 defines what every fungible token *must* implement.

ERC = Ethereum Request for Comments — ERC-20 was proposed in 2015 by Fabian Vogelsteller

ERC-20: The Fungible Token Standard

What is ERC-20?
A technical standard defining:

- Required functions every token must have
- Events that must be emitted
- Expected behavior for transfers

Fungibility:

- Each token is identical
- 1 token = 1 token (interchangeable)
- Like dollar bills — any $1 equals any other $1

The Standard Interface

6 Required Functions:

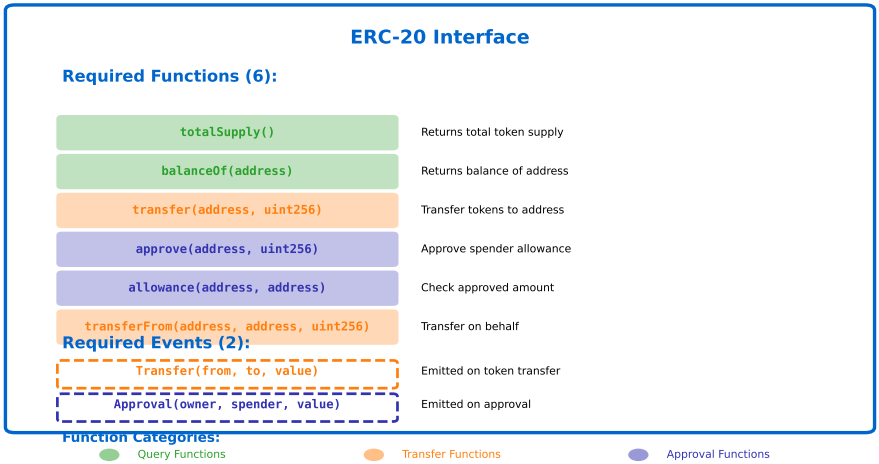
1. `totalSupply()`
2. `balanceOf(address)`
3. `transfer(to, amount)`
4. `approve(spender, amount)`
5. `allowance(owner, spender)`
6. `transferFrom(from, to, amount)`

2 Required Events:

1. `Transfer(from, to, amount)`
2. `Approval(owner, spender, amount)`

Any contract implementing these 6 functions and 2 events is ERC-20 compliant

ERC-20 Token Standard Interface



Other Token Standards

ERC-721 (NFTs)
Non-Fungible Tokens:

- Each token is unique
- Has a `tokenId`
- Digital collectibles
- Art, gaming items
- Proof of ownership

Example: CryptoPunks, Bored Apes

ERC-1155 (Multi-Token)
Best of both worlds:

- Fungible AND non-fungible
- Batch transfers
- Gas efficient
- Gaming use cases
- Mixed collections

Example: OpenSea collections

ERC-4626 (Vaults)
Tokenized yield vaults:

- Standardized yield
- DeFi composability
- Deposit/withdraw
- Share accounting
- Yield strategies

Example: Yearn vaults

Standard	Fungible	Use Case	Year
ERC-20	Yes	Currencies, utility	2015
ERC-721	No	Collectibles, art	2018
ERC-1155	Both	Gaming, mixed	2019

This lesson focuses on ERC-20; ERC-721 and ERC-1155 require additional complexity

Section 2: ERC-20 Functions Deep Dive

Understanding each function in detail

Read Functions: totalSupply() and balanceOf()

totalSupply()

```
function totalSupply()
    external view returns (uint256);
```

- Returns total tokens in existence
- `view` — reads state, no gas cost
- Used by wallets and explorers
- Can be fixed or dynamic

Example:

- USDC: ∼25 billion tokens
- Our token: 1,000,000 max

balanceOf(address)

```
function balanceOf(address account)
    external view returns (uint256);
```

- Returns balance of specific address
- `view` — no gas for external calls
- Most frequently called function
- Wallets use this constantly

Internal Storage:

```
mapping(address => uint256)
    private _balances;
```

These are view functions — they read blockchain state without modifying it (no transaction needed)

Direct Transfer: transfer()

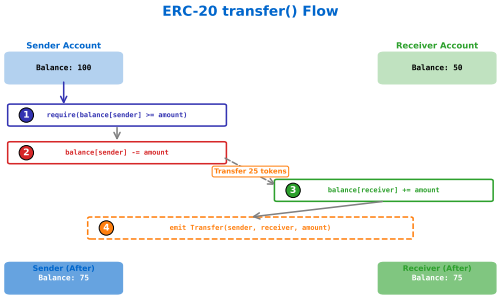
Function Signature

```
function transfer(
    address to,
    uint256 amount
) external returns (bool);
```

How It Works:

1. Caller initiates transfer
2. Contract checks balance
3. Subtracts from sender
4. Adds to recipient
5. Emits `Transfer` event
6. Returns `true` on success

Token Flow Diagram




```
// Internal implementation
function _transfer(address from, address to, uint256 amount) internal {
    require(from != address(0), "Transfer from zero address");
    require(to != address(0), "Transfer to zero address");
    require(_balances[from] >= amount, "Insufficient balance");
    _balances[from] -= amount;
    _balances[to] += amount;
    emit Transfer(from, to, amount);
}
```

Delegation: approve() and allowance()

approve(spender, amount)

```
function approve(
    address spender,
    uint256 amount
) external returns (bool);
```

Purpose:

- Authorize another address to spend
- Essential for DeFi interactions
- Set spending limit (allowance)
- Can be updated or revoked (set to 0)

allowance(owner, spender)

```
function allowance(
    address owner,
    address spender
) external view returns (uint256);
```

Purpose:

- Check remaining allowance
- Used before transferFrom
- Wallets display this info
- Returns 0 if none approved

Internal Storage:

```
mapping(address => mapping(address => uint256)) private _allowances;
// _allowances[owner][spender] = amount
```

Approvals enable smart contracts (DEXs, lending) to move tokens on your behalf

Delegated Transfer: transferFrom()

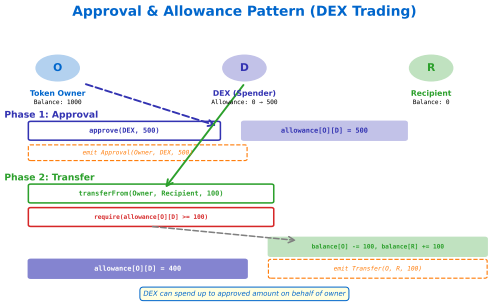
Function Signature

```
function transferFrom(
    address from,
    address to,
    uint256 amount
) external returns (bool);
```

How It Works:

1. Spender calls transferFrom
2. Contract checks allowance
3. Contract checks balance
4. Deducts from allowance
5. Transfers tokens
6. Emits Transfer event

Approval Flow



Real-World Example:

1. You approve Uniswap to spend 1000 USDC
2. When you swap, Uniswap calls `transferFrom(you, pool, 500)`
3. Tokens move from your wallet to liquidity pool

transferFrom() is used by smart contracts — requires prior approve() call

Events: Transfer and Approval

Transfer Event

```
event Transfer(
    address indexed from,
    address indexed to,
    uint256 value
);
```

Emitted When:

- Tokens are transferred
- Tokens are minted (from = 0x0)
- Tokens are burned (to = 0x0)

Used By:

- Block explorers (Etherscan)
- Wallet notifications
- DeFi dashboards

Approval Event

```
event Approval(
    address indexed owner,
    address indexed spender,
    uint256 value
);
```

Emitted When:

- Approval is granted
- Approval is changed
- Approval is revoked (value = 0)

Why `indexed`?

- Enables efficient filtering
- Can search by address
- Max 3 indexed params

Events are stored in transaction logs — cheap storage but cannot be read by contracts

Code: Basic ERC-20 Structure

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

contract BasicToken {
    string public name = "BasicToken";
    string public symbol = "BTK";
    uint8 public decimals = 18;
    uint256 public totalSupply;

    mapping(address => uint256) private _balances;
    mapping(address => mapping(address => uint256)) private _allowances;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    constructor(uint256 initialSupply) {
        totalSupply = initialSupply * 10**decimals;
        _balances[msg.sender] = totalSupply;
        emit Transfer(address(0), msg.sender, totalSupply);
    }

    function balanceOf(address account) external view returns (uint256) {
        return _balances[account];
    }

    function transfer(address to, uint256 amount) external returns (bool) {
        require(_balances[msg.sender] >= amount, "Insufficient balance");
        _balances[msg.sender] -= amount;
        _balances[to] += amount;
        emit Transfer(msg.sender, to, amount);
        return true;
    }
    // ... approve, allowance, transferFrom
}
```

Using OpenZeppelin

Why OpenZeppelin?

- ✓ Battle-tested code
- ✓ Security audited
- ✓ Industry standard
- ✓ Modular extensions
- ✓ Regular updates
- ✓ Community support

Installation:

```
npm install @openzeppelin/contracts
```

Minimal ERC-20 Token

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

import "@openzeppelin/contracts/
    token/ERC20/ERC20.sol";

contract SimpleToken is ERC20 {
    constructor()
        ERC20("SimpleToken", "STK")
    {
        _mint(msg.sender,
              1000000 * 10**decimals());
    }
}
```

That's it! 8 lines for a complete, secure ERC-20 token.

OpenZeppelin is the gold standard for smart contract development — never reinvent the wheel

Section 3: Build MyToken

Hands-on: Creating a complete ERC-20 token

Our Token Specification

MyToken (MTK) Features

Property	Value
Name	MyToken
Symbol	MTK
Decimals	18
Max Supply	1,000,000 MTK
Initial Supply	100,000 MTK

Custom Functions:

- `mint()` — Owner can create tokens
- `burn()` — Anyone can destroy tokens
- `burnFrom()` — Burn with allowance
- `batchTransfer()` — Multiple transfers

Inherited Features
From **ERC20.sol**:

- All 6 standard functions
- Both standard events
- Decimal handling
- Safe math (Solidity 0.8+)

From **Ownable.sol**:

- Owner management
- `onlyOwner` modifier
- Ownership transfer
- Renounce ownership

Design Decision:
Max supply cap prevents unlimited inflation.

This design balances simplicity with useful features for learning and experimentation

Code: MyToken.sol Imports and Setup

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

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

/**
 * @title MyToken
 * @dev ERC-20 token implementation with minting and burning capabilities
 * @notice This contract demonstrates standard token functionality
 */
contract MyToken is ERC20, Ownable {
    // Maximum supply cap for the token
    uint256 public constant MAX_SUPPLY = 1000000 * 10**18;  // 1 million tokens

    // Contract body follows...
}
```

Key Elements:

- SPDX license identifier (required)
- Solidity version pragma
- OpenZeppelin imports
- NatSpec documentation

Inheritance:

- `is ERC20` — Token functions
- `is Ownable` — Access control
- Multiple inheritance with comma
- Order matters for constructors

The constant MAX_SUPPLY is stored in contract bytecode, not storage (gas efficient)

Code: Constructor and Initial Minting

```
/**
 * @dev Constructor that initializes the token
 * @notice Mints initial supply to the contract deployer
 */
constructor() ERC20("MyToken", "MTK") Ownable(msg.sender) {
    // Mint 100,000 tokens to the deployer (10% of max supply)
    _mint(msg.sender, 100000 * 10**18);
}
```

Constructor Breakdown:

1. `ERC20("MyToken", "MTK")`
   - Sets token name
   - Sets token symbol
2. `Ownable(msg.sender)`
   - Sets deployer as owner
   - Enables access control
3. `_mint(msg.sender, ...)`
   - Creates initial tokens
   - Assigns to deployer

Why `10**18`?
ERC-20 tokens use 18 decimals by default:

- 1 token = 10^{18} base units
- Like 1 ETH = 10^{18} wei
- Enables fractional amounts
- `100000 * 10**18` = 100,000 tokens

Initial Distribution:
10% to deployer, 90% available for minting

The _mint function is internal — it creates tokens "from nothing" (from address 0)

Code: Custom Features (Mint and Burn)

```
/**
 * @dev Mint new tokens to a specified address
 * @param to Address that will receive the minted tokens
 * @param amount Amount of tokens to mint (in wei)
 */
function mint(address to, uint256 amount) public onlyOwner {
    require(to != address(0), "Cannot mint to zero address");
    require(totalSupply() + amount <= MAX_SUPPLY, "Exceeds max supply");
    _mint(to, amount);
}

/**
 * @dev Burn tokens from the caller's balance
 * @param amount Amount of tokens to burn
 */
function burn(uint256 amount) public {
    _burn(msg.sender, amount);
}

/**
 * @dev Burn tokens from a specified address (requires allowance)
 */
function burnFrom(address from, uint256 amount) public {
    _spendAllowance(from, msg.sender, amount);
    _burn(from, amount);
}
```

mint() has supply cap protection — burn() is deflationary — burnFrom() requires approval

Code: Full Contract Walkthrough

Contract Structure

```
contract MyToken is ERC20, Ownable {
    // 1. State Variables
    uint256 public constant MAX_SUPPLY;

    // 2. Constructor
    constructor() { ... }

    // 3. Core Functions
    function mint() onlyOwner { ... }
    function burn() public { ... }
    function burnFrom() public { ... }

    // 4. View Functions
    function decimals() override { ... }

    // 5. Utility Functions
    function batchTransfer() public { ... }
}
```

Batch Transfer Feature

```
function batchTransfer(
    address[] calldata recipients,
    uint256[] calldata amounts
) public {
    require(
        recipients.length == amounts.length,
        "Arrays length mismatch"
    );
    for (uint256 i = 0;
         i < recipients.length; i++) {
        _transfer(
            msg.sender,
            recipients[i],
            amounts[i]
        );
    }
}
```

Use Case: Airdrops, payroll

Full source: code/MyToken.sol — Total: 82 lines including documentation

Compile with Hardhat

Project Setup

```
# Initialize project
mkdir my-token && cd my-token
npm init -y
npm install --save-dev hardhat
npx hardhat init

# Install OpenZeppelin
npm install @openzeppelin/contracts
```

Project Structure:

```
my-token/
  contracts/
    MyToken.sol
  scripts/
    deploy.js
  test/
    MyToken.test.js
  hardhat.config.js
  package.json
```

Compilation

```
# Compile contracts
npx hardhat compile

# Expected output:
Compiling 1 file with 0.8.20
Compilation finished successfully
```

Compilation Artifacts:

- `artifacts/` — ABI and bytecode
- `cache/` — Build cache
- ABI used for frontend integration
- Bytecode deployed to blockchain

Common Errors:

- Version mismatch in pragma
- Missing OpenZeppelin install
- Syntax errors in Solidity

Hardhat is the most popular Ethereum development environment — alternatives: Foundry, Truffle

Deploy Script

```
// scripts/deploy.js - Hardhat deployment script
const hre = require("hardhat");

async function main() {
  console.log("Starting MyToken deployment...\n");

  // Get the contract factory
  const MyToken = await hre.ethers.getContractFactory("MyToken");

  // Get deployer account
  const [deployer] = await hre.ethers.getSigners();
  console.log("Deploying with account:", deployer.address);

  // Deploy the contract
  const token = await MyToken.deploy();
  await token.waitForDeployment();

  const tokenAddress = await token.getAddress();
  console.log("MyToken deployed to:", tokenAddress);

  // Verify deployment
  const name = await token.name();
  const symbol = await token.symbol();
  const totalSupply = await token.totalSupply();
  console.log(`Token: ${name} (${symbol})`);
  console.log(`Total Supply: ${hre.ethers.formatUnits(totalSupply, 18)}`);
}

main().catch((error) => { console.error(error); process.exit(1); });
```

Run with: npx hardhat run scripts/deploy.js –network localhost

Testing the Token

Local Testing Steps

1. Start local blockchain:

```
npx hardhat node
```

2. Deploy to local network:

```
npx hardhat run scripts/deploy.js
  --network localhost
```

3. Run test suite:

```
npx hardhat test
```

Hardhat Console:

```
npx hardhat console --network localhost
> const Token = await ethers.
    getContractFactory("MyToken")
> const token = await Token.attach(
    "0x5FbDB...")
> await token.balanceOf(deployer)
```

Sample Test Cases

```
describe("MyToken", function() {
  it("Should have correct name",
    async function() {
      expect(await token.name())
        .to.equal("MyToken");
  });

  it("Should mint initial supply",
    async function() {
      const balance = await token
        .balanceOf(owner.address);
      expect(balance).to.equal(
        ethers.parseEther("100000")
      );
  });

  it("Should transfer tokens",
    async function() {
      await token.transfer(
        addr1.address,
        ethers.parseEther("100")
      );
      // ... assertions
  });
});
```

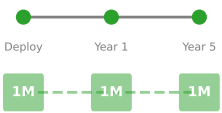
Always test thoroughly on local network before deploying to testnet or mainnet

Section 4: Token Economics

Designing sustainable token models

Token Supply Models

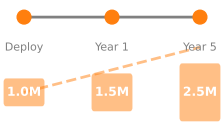
Fixed Supply



- Total supply fixed at creation
- No new tokens can be minted
- Predictable scarcity
- Example: Bitcoin (21M cap)

Anti-inflationary, value preservation

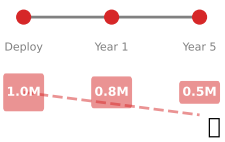
Inflationary Supply



- New tokens minted over time
- Rewards for staking/mining
- Controlled inflation rate
- Example: Ethereum (post-merge)

Incentivizes participation, network growth

Deflationary Supply



- Tokens burned/destroyed
- Transaction fee burns
- Buyback and burn programs
- Example: BNB (quarterly burns)

Increases scarcity, potential price appreciation

Fixed vs Inflationary vs Deflationary

Fixed Supply
"Digital Gold"

- Supply set at creation
- Cannot mint more
- Scarcity drives value
- Example: Bitcoin (21M)

Code:

```
constructor() {
  _mint(msg.sender,
        MAX_SUPPLY);
  // No mint function
}
```

Inflationary
"Rewards & Growth"

- New tokens created
- Rewards for staking
- Funds development
- Example: ETH (post-merge)

Code:

```
function mint(
  address to,
  uint256 amount
) public onlyOwner {
  _mint(to, amount);
}
```

Deflationary
"Burn to Earn"

- Tokens destroyed
- Fee burns
- Increasing scarcity
- Example: BNB

Code:

```
function transfer(...) {
  uint256 fee = amount/100;
  _burn(from, fee);
  _transfer(from, to,
            amount - fee);
}
```

Our Token (MTK): Hybrid — Capped inflation with optional burning

Token economics (tokenomics) significantly impacts long-term value and utility

Distribution Strategies

Common Distribution Methods

Method	Typical %
Team & Founders	15-20%
Investors/VCs	10-20%
Public Sale (ICO/IDO)	10-30%
Community/Airdrops	10-20%
Ecosystem Fund	20-30%
Liquidity	5-15%

Vesting Schedules:

- Team tokens locked 1-4 years
- Cliff period (6-12 months)
- Linear or milestone unlocks
- Prevents dump risk

Distribution Considerations

- ✓ **Fair Launch:** No pre-mine, equal access
- ✓ **Decentralization:** Avoid concentration
- ✓ **Incentive Alignment:** Team locked
- ✓ **Liquidity:** Tradeable from day 1

Red Flags:

- × Team holds ¿50%
- × No vesting schedule
- × Hidden wallets
- × Unlimited minting

Transparent tokenomics build trust — always publish distribution details

Real-World Token Examples

Token	Type	Supply Model	Max Supply	Use Case
USDC	Stablecoin	Elastic	Unlimited	Payments
LINK	Utility	Fixed	1B	Oracle services
UNI	Governance	Inflationary	1B + 2%/yr	DEX voting
AAVE	Governance	Fixed	16M	Lending protocol
BNB	Utility	Deflationary	200M (burns)	Exchange fees
MATIC	Utility	Fixed	10B	L2 scaling

Successful Patterns:

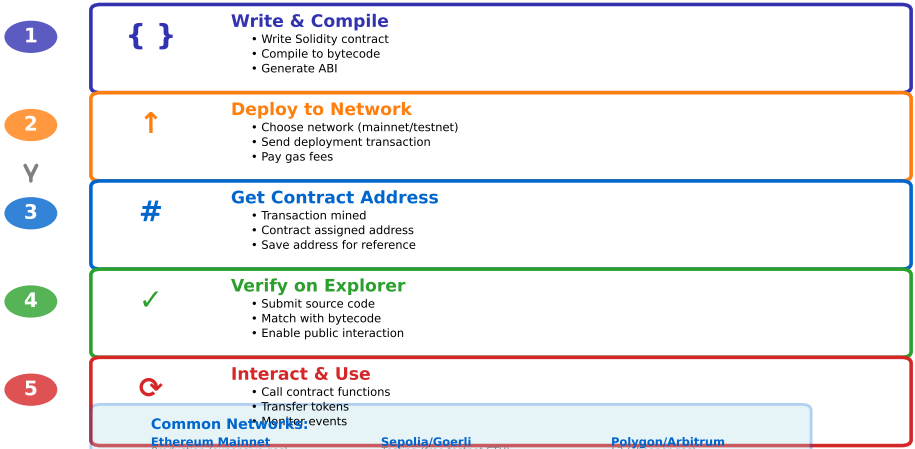
- Clear utility (not just speculation)
- Sustainable tokenomics
- Active development
- Strong community

Questions to Ask:

- Why does this need a token?
- What creates demand?
- Who controls supply?
- What's the unlock schedule?

Study successful tokens to understand what makes sustainable token economics

ERC-20 Token Deployment Workflow



Section 5: Advanced Topics & Summary

Next steps and key takeaways

Optional: Testnet Deployment

Testnet Options

Network	Faucet	Speed
Sepolia	Yes	Fast
Goerli	Limited	Medium
Mumbai (Polygon)	Yes	Fast

Hardhat Config:

```
networks: {
  sepolia: {
    url: process.env.SEPOLIA_URL,
    accounts: [process.env.PRIVATE_KEY]
  }
}
```

Deployment Steps

1. Get testnet ETH from faucet
2. Configure network in Hardhat
3. Set environment variables
4. Deploy:

```
npx hardhat run scripts/deploy.js
  --network sepolia
```

5. Verify on Etherscan:

```
npx hardhat verify
  --network sepolia
  CONTRACT_ADDRESS
```

Cost: Free (testnet ETH has no value)

Testnet deployment is essential practice before mainnet — treat it like production

Security Considerations

IMPORTANT DISCLAIMER

The code in this lesson is for **educational purposes only**. Do NOT deploy to mainnet without professional security audits. Real money is at stake on mainnet!

Common Vulnerabilities

- ! **Reentrancy:** Recursive calls
- ! **Integer overflow:** Pre-0.8.0
- ! **Access control:** Missing checks
- ! **Front-running:** MEV attacks
- ! **Approval race:** approve() issue

Before Mainnet:

- Professional audit ($5k-$100k+)
- Bug bounty program
- Testnet battle-testing
- Formal verification

Security Best Practices

- ✓ Use OpenZeppelin contracts
- ✓ Follow checks-effects-interactions
- ✓ Use `ReentrancyGuard`
- ✓ Implement `Pausable`
- ✓ Use multi-sig for ownership
- ✓ Test edge cases extensively

Security Resources:

- Slither (static analysis)
- Mythril (symbolic execution)
- OpenZeppelin Defender
- Consensys Diligence

Smart contract bugs are permanent and can result in total loss of funds

Final Project Overview

Project: Custom Token
Create your own ERC-20 token with:

1. Custom name and symbol
2. Defined tokenomics (supply model)
3. At least one custom feature
4. Comprehensive test suite
5. Deployed to testnet

Bonus Challenges:

- Add token burning on transfer
- Implement vesting schedule
- Create airdrop function
- Add permit (EIP-2612)

Deliverables

- `MyToken.sol` — Contract code
- `deploy.js` — Deployment script
- `test/*.js` — Test files
- README with tokenomics
- Testnet deployment address
- Etherscan verification

Evaluation Criteria:

- Code quality and documentation
- Test coverage (¿80%)
- Security considerations
- Creative features
- Working deployment

This project integrates all 4 lessons: blockchain, cryptography, smart contracts, and tokens

Key Takeaways

What We Learned

1. **Token Standards**
   - ERC-20 defines fungible tokens
   - Standards enable interoperability
   - 6 functions + 2 events
2. **Core Functions**
   - transfer() for direct sends
   - approve()/transferFrom() for delegation
   - Events for transparency
3. **Development**
   - OpenZeppelin for security
   - Hardhat for tooling
   - Test before deploy

Key Concepts

- ✓ Tokens are smart contracts
- ✓ Standards enable ecosystem
- ✓ Inheritance simplifies code
- ✓ Tokenomics matter
- ✓ Security is paramount

Skills Acquired:

- Read and understand ERC-20
- Create custom tokens
- Deploy with Hardhat
- Design tokenomics
- Evaluate token projects

You now have the foundation to build, deploy, and evaluate ERC-20 tokens

Course Completion & Next Steps

Course Summary

Lesson	Topic
1	Blockchain Fundamentals
2	Cryptography & Security
3	Ethereum & Smart Contracts
4	ERC-20 Token Creation

You Can Now:

- Understand blockchain architecture
- Explain cryptographic security
- Write Solidity smart contracts
- Create and deploy tokens

Continue Learning

- **NFTs:** ERC-721 & ERC-1155
- **DeFi:** AMMs, lending, yield
- **DAOs:** Governance tokens
- **L2 Scaling:** Rollups, sidechains
- **Security:** Auditing, formal methods

Resources:

- Ethereum.org documentation
- OpenZeppelin Learn
- CryptoZombies (interactive)
- Alchemy University
- Speedrun Ethereum

Congratulations on completing the course!

The blockchain space evolves rapidly — continuous learning is essential

Thank You

Questions?

Course: Create Your Own Cryptocurrency

Lesson 4: ERC-20 Token Creation

Build. Deploy. Innovate.