

L17: ERC-20 Token Standard

Module B: Ethereum & Smart Contracts

Blockchain & Cryptocurrency Course

December 2025

By the end of this lesson, you will be able to:

- Explain the purpose and importance of the ERC-20 standard
- Describe the six required ERC-20 interface functions
- Understand the allowance mechanism for delegated transfers
- Implement a basic ERC-20 token from scratch
- Use OpenZeppelin's audited ERC-20 implementation
- Analyze real-world ERC-20 tokens (USDC, DAI, LINK)

What is ERC-20?

ERC-20 is the dominant fungible token standard on Ethereum:

- **ERC:** Ethereum Request for Comments (proposal process)
- **20:** Proposal number (introduced November 2015 by Fabian Vogelsteller)
- **Fungible:** Each token is identical and interchangeable (like currency)
- **Standard:** Common interface enables interoperability

Why Standardization Matters:

- Wallets (MetaMask, Ledger) support all ERC-20 tokens without custom code
- Exchanges can list new tokens easily
- Smart contracts can interact with any ERC-20 token
- Developers use common patterns and libraries
- Reduces fragmentation and security risks

Usage: Currencies, governance tokens, stablecoins, utility tokens, rewards

Six mandatory functions:

```
interface IERC20 {  
    // Returns the total token supply  
    function totalSupply() external view returns (uint256);  
  
    // Returns the balance of an account  
    function balanceOf(address account) external view returns (uint256);  
  
    // Transfers tokens from caller to recipient  
    function transfer(address recipient, uint256 amount) external returns (bool);  
  
    // Returns remaining tokens that spender is allowed to spend on behalf of owner  
    function allowance(address owner, address spender) external view returns (uint256);  
  
    // Sets amount as allowance of spender over caller's tokens  
    function approve(address spender, uint256 amount) external returns (bool);  
  
    // Transfers tokens from sender to recipient using allowance mechanism  
    function transferFrom(address sender, address recipient, uint256 amount)  
        external returns (bool);  
}
```

Two mandatory events:

```
interface IERC20 {  
    // Emitted when tokens are transferred  
    event Transfer(address indexed from, address indexed to, uint256 value);  
  
    // Emitted when allowance is set via approve()  
    event Approval(address indexed owner, address indexed spender, uint256 value);  
}
```

Why Events Are Required:

- Enable off-chain indexing of token transfers
- Wallets and explorers listen to Transfer events to update balances
- DEXs track Approval events for trading activity
- Much cheaper than storing transfer history in storage
- Indexed parameters allow efficient filtering (e.g., all transfers to address X)

Not required but widely used:

```
// Returns the name of the token (e.g., "US Dollar Coin")
function name() public view returns (string memory);

// Returns the symbol of the token (e.g., "USDC")
function symbol() public view returns (string memory);

// Returns the number of decimals (e.g., 18 for ETH-like, 6 for USDC)
function decimals() public view returns (uint8);
```

Decimals Explained:

- Tokens are stored as integers (no floating point on EVM)
- decimals defines how to display token amounts
- Example: 1 USDC = 1,000,000 units (6 decimals)
- Example: 1 DAI = 1,000,000,000,000,000 units (18 decimals)
- Convention: 18 decimals (same as ETH) unless special reason

Understanding Allowance Mechanism

Problem: How can a smart contract spend your tokens?

Solution: Two-step approve + transferFrom pattern

❶ **User calls approve():**

- `token.approve(spenderAddress, amount)`
- Sets allowance: "spenderAddress can spend up to amount of my tokens"
- Emits Approval event

❷ **Spender calls transferFrom():**

- `token.transferFrom(userAddress, recipient, amount)`
- Transfers tokens from user to recipient
- Decreases allowance by amount
- Emits Transfer event

Use Cases:

- DEX trading: User approves DEX contract to spend tokens
- Staking: User approves staking contract to lock tokens
- Subscription: User approves service to charge tokens periodically

Scenario: Alice wants to trade 100 DAI for ETH on Uniswap

Step-by-Step:

- ① Alice calls `DAI.approve(UniswapRouter, 100e18)`
 - Sets allowance: Uniswap can spend 100 DAI
 - Gas cost: approximately 45,000
- ② Alice calls `UniswapRouter.swapTokensForETH(100e18, ...)`
 - Uniswap calls `DAI.transferFrom(Alice, UniswapPool, 100e18)`
 - DAI transferred from Alice to Uniswap pool
 - Alice's allowance decreases to 0
 - Alice receives ETH in return
- ③ If Alice wants to trade again, she needs to approve again

Infinite Approval:

- Users often approve `type(uint256).max` to avoid repeated approvals
- Risk: If DEX contract is hacked, all approved tokens can be stolen
- Tradeoff: Convenience vs security

Basic ERC-20 Implementation (Part 1)

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

contract BasicERC20 {
    string public name;
    string public symbol;
    uint8 public decimals;
    uint256 public totalSupply;

    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

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

    constructor(string memory _name, string memory _symbol, uint8 _decimals,
        uint256 _initialSupply) {
        name = _name;
        symbol = _symbol;
        decimals = _decimals;
        totalSupply = _initialSupply * 10**_decimals;
        balanceOf[msg.sender] = totalSupply; // Mint all tokens to deployer
        emit Transfer(address(0), msg.sender, totalSupply);
    }
}
```

Basic ERC-20 Implementation (Part 2)

```
function transfer(address recipient, uint256 amount) public returns (bool) {
    require(recipient != address(0), "Transfer to zero address");
    require(balanceOf[msg.sender] >= amount, "Insufficient balance");

    balanceOf[msg.sender] -= amount;
    balanceOf[recipient] += amount;
    emit Transfer(msg.sender, recipient, amount);
    return true;
}

function approve(address spender, uint256 amount) public returns (bool) {
    require(spender != address(0), "Approve to zero address");

    allowance[msg.sender][spender] = amount;
    emit Approval(msg.sender, spender, amount);
    return true;
}
```

Basic ERC-20 Implementation (Part 3)

```
function transferFrom(address sender, address recipient, uint256 amount)
    public returns (bool) {
    require(sender != address(0), "Transfer from zero address");
    require(recipient != address(0), "Transfer to zero address");
    require(balanceOf[sender] >= amount, "Insufficient balance");
    require(allowance[sender][msg.sender] >= amount, "Insufficient allowance");

    balanceOf[sender] -= amount;
    balanceOf[recipient] += amount;
    allowance[sender][msg.sender] -= amount;

    emit Transfer(sender, recipient, amount);
    return true;
}
```

Note: This is a minimal implementation. Production tokens should use OpenZeppelin.

Common ERC-20 Vulnerabilities:

① Approve Race Condition:

- Problem: Changing allowance from A to B allows spender to spend $A+B$
- Attack: Spender front-runs second `approve()` to spend both amounts
- Solution: Use `increaseAllowance()` and `decreaseAllowance()`

② Integer Overflow (pre-0.8.0):

- Problem: `balanceOf[user] + amount` could overflow
- Solution: Use Solidity 0.8+ (built-in overflow checks) or `SafeMath` library

③ Reentrancy in Transfer Hooks:

- Problem: If token calls external contract during transfer, reentrancy possible
- Solution: Follow checks-effects-interactions pattern

④ Fee-on-Transfer Tokens:

- Some tokens deduct fees on transfer (e.g., `SAFEMOON`)
- Breaks assumption that `transfer(amount)` sends exactly `amount`
- DEXs must check balance before/after to handle correctly

Industry-standard audited implementation:

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

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

contract MyToken is ERC20 {
    constructor(uint256 initialSupply) ERC20("My Token", "MTK") {
        _mint(msg.sender, initialSupply * 10**decimals());
    }
}
```

Advantages:

- Battle-tested code (used by USDC, LINK, thousands of projects)
- Security audits by leading firms
- Gas-optimized
- Extensions available: ERC20Burnable, ERC20Pausable, ERC20Permit
- Regular updates for new security best practices

Useful ERC-20 extensions:

1 ERC20Burnable:

```
function burn(uint256 amount) public {
    _burn(msg.sender, amount); // Destroy tokens, decrease totalSupply
}
```

2 ERC20Pausable:

```
function pause() public onlyOwner {
    _pause(); // Disable all transfers
}
```

3 ERC20Permit (EIP-2612):

```
function permit(address owner, address spender, uint256 value, uint256 deadline,
    uint8 v, bytes32 r, bytes32 s) public {
    // Approve via signature, no separate transaction needed
}
```

USD Coin (USDC) - Circle's stablecoin:

Key Features:

- Pegged 1:1 to US Dollar
- Backed by cash and short-term US Treasuries
- 6 decimals (not 18, to match fiat cents)
- Upgradable proxy pattern
- Blacklist function (regulatory compliance)
- Pausable (emergency circuit breaker)

Contract Structure:

- **Proxy:** 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48 (fixed address)
- **Implementation:** Upgradable, currently v2.2
- **Master Minter:** Can add/remove minters
- **Minters:** Authorized addresses that can mint USDC
- **Blacklist:** Can freeze addresses (e.g., sanctioned entities)

DAI - MakerDAO's decentralized stablecoin:

Key Features:

- Pegged to 1 USD via collateralized debt positions (CDPs)
- Over-collateralized by crypto assets (ETH, wBTC, USDC)
- 18 decimals (standard)
- Permissionless minting via Maker Vaults
- No admin blacklist or pause (more decentralized than USDC)

ERC-20 Extensions:

- `permit()` - EIP-2612 gasless approvals
- `mint()` - Only callable by Maker Vat (core contract)
- `burn()` - Destroy DAI when repaying debt

Address: 0x6B175474E89094C44Da98b954EedeAC495271d0F

Real-World Example: Chainlink (LINK)

LINK - Chainlink's utility token:

Key Features:

- Used to pay node operators for oracle services
- Fixed supply: 1 billion LINK (no minting)
- Standard ERC-20 with `transferAndCall()` extension
- 18 decimals

`transferAndCall()` Pattern:

```
function transferAndCall(address to, uint256 value, bytes calldata data)
    external returns (bool) {
    transfer(to, value);
    // Call recipient contract's onTokenTransfer() function
    require(to.call(abi.encodeWithSignature("onTokenTransfer(address,uint256,bytes)",
        msg.sender, value, data)));
    return true;
}
```

Use Case: Single transaction to send LINK and trigger oracle request

Common minting approaches:

1 Fixed Supply (e.g., Bitcoin, LINK):

- All tokens minted at deployment
- No future minting possible
- Deflationary if burning is enabled

2 Owner-Controlled Minting (e.g., USDC):

- Authorized addresses can mint
- Requires trust in minter
- Used for stablecoins (mint when fiat deposited)

3 Algorithmic Minting (e.g., DAI, UNI):

- Minting via smart contract logic
- No human discretion
- DAI: Mint by depositing collateral
- UNI: Minted via liquidity mining rewards

4 Inflationary Rewards (e.g., staking tokens):

- Continuous minting at fixed rate
- Distributed to stakers/validators

Why burn tokens?

1 Reduce Supply:

- Increase scarcity and potentially value
- Example: BNB quarterly burns (Binance uses profits to buy back and burn)

2 Stablecoin Redemption:

- Burn when user redeems for fiat
- Example: USDC burn when user withdraws USD

3 Fee Burning (EIP-1559 model):

- ETH burns base fee to reduce supply
- Some tokens adopt similar mechanics

4 Proof of Burn:

- Destroy tokens on one chain to mint on another
- Used in cross-chain bridges

```
function burn(uint256 amount) public {
    _burn(msg.sender, amount); // Decrease balance and totalSupply
}
```

Extensions to standard ERC-20:

❶ EIP-2612 (Permit):

- Approve via off-chain signature (meta-transactions)
- Saves one transaction for users
- Used by DAI, USDC

❷ ERC-20 Snapshot:

- Capture token balances at specific block
- Used for governance voting (prevent double voting)

❸ ERC-20 Votes:

- Delegation of voting power
- Used by UNI, COMP governance tokens

❹ Rebase Tokens (e.g., Ampleforth):

- Token balance changes automatically based on price
- Non-standard, breaks ERC-20 assumptions

❺ Wrapped Tokens (e.g., WETH):

- ERC-20 wrapper around native ETH
- Enables ETH to be used in ERC-20 contracts

- ❶ **ERC-20 Standard:** Defines common interface for fungible tokens, enabling ecosystem-wide interoperability
- ❷ **Core Functions:** `totalSupply`, `balanceOf`, `transfer`, `approve`, `allowance`, `transferFrom`
- ❸ **Allowance Mechanism:** Two-step `approve` + `transferFrom` enables smart contracts to spend user tokens
- ❹ **Security:** Use OpenZeppelin implementation, beware of `approve` race condition and reentrancy
- ❺ **Decimals:** Tokens stored as integers, `decimals` defines display precision (convention: 18)
- ❻ **Real-World Usage:** USDC (centralized, 6 decimals, blacklist), DAI (decentralized, 18 decimals, permit), LINK (fixed supply, `transferAndCall`)
- ❼ **Token Economics:** Minting strategies (fixed, controlled, algorithmic) and burning (scarcity, redemption, fee model)

- ❶ Why do stablecoins like USDC use 6 decimals instead of 18?
- ❷ What are the security tradeoffs between approving a limited amount vs infinite approval for DEX trading?
- ❸ How does the EIP-2612 permit() function improve user experience compared to standard approve()?
- ❹ What are the regulatory implications of USDC's blacklist function compared to DAI's permissionless design?
- ❺ How might fee-on-transfer tokens break assumptions in DEX contracts or other DeFi protocols?

Coming up next:

- Non-fungible tokens (NFTs) and the ERC-721 standard
- ownerOf, tokenURI, safeTransferFrom functions
- Metadata standards and IPFS integration
- ERC-1155 multi-token standard (fungible + non-fungible)
- Batch operations and gas efficiency
- Real-world examples: CryptoPunks, Bored Apes, ENS domains

Preparation:

- Browse NFT collections on OpenSea
- Review ERC-721 interface specification (eips.ethereum.org)
- Explore IPFS basics (ipfs.io)