// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

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

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

import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol";

// 1. Burnable Token

contract BurnableToken is ERC20Burnable {

constructor(string memory name, string memory symbol) ERC20(name, symbol) {

\_mint(msg.sender, 1000000 \* 10 \*\* decimals());

}

}

// 2. Mintable Token

contract MintableToken is ERC20, Ownable {

constructor(string memory name, string memory symbol) ERC20(name, symbol) Ownable(msg.sender) {}

function mint(address to, uint256 amount) public onlyOwner {

\_mint(to, amount);

}

}

// 3. Capped Token

contract CappedToken is ERC20, Ownable {

uint256 private immutable \_cap;

constructor(string memory name, string memory symbol, uint256 cap\_) ERC20(name, symbol) Ownable(msg.sender) {

require(cap\_ > 0, "Cap must be positive");

\_cap = cap\_;

}

function cap() public view returns (uint256) {

return \_cap;

}

function mint(address account, uint256 amount) public onlyOwner {

require(totalSupply() + amount <= cap(), "Cap exceeded");

\_mint(account, amount);

}

}

// 4. Pausable Token

import "@openzeppelin/contracts/security/Pausable.sol";

contract PausableToken is ERC20, Pausable, Ownable {

constructor(string memory name, string memory symbol) ERC20(name, symbol) Ownable(msg.sender) {

\_mint(msg.sender, 1000000 \* 10 \*\* decimals());

}

function pause() public onlyOwner {

\_pause();

}

function unpause() public onlyOwner {

\_unpause();

}

}

// 5. Taxed (Fee) Token

contract TaxedToken is ERC20, Ownable {

address public taxWallet;

uint256 public taxRate = 200; // 2% tax (basis points)

uint256 public constant MAX\_TAX = 1000; // 10% max tax

constructor(string memory name, string memory symbol, address \_taxWallet) ERC20(name, symbol) Ownable(msg.sender) {

require(\_taxWallet != address(0), "Zero address");

taxWallet = \_taxWallet;

\_mint(msg.sender, 1000000 \* 10 \*\* decimals());

}

function setTaxRate(uint256 \_taxRate) external onlyOwner {

require(\_taxRate <= MAX\_TAX, "Tax too high");

taxRate = \_taxRate;

}

function setTaxWallet(address \_taxWallet) external onlyOwner {

require(\_taxWallet != address(0), "Zero address");

taxWallet = \_taxWallet;

}

// Custom transfer function instead of overriding \_transfer

function transfer(address recipient, uint256 amount) public override returns (bool) {

\_customTransfer(\_msgSender(), recipient, amount);

return true;

}

function transferFrom(address sender, address recipient, uint256 amount) public override returns (bool) {

uint256 currentAllowance = allowance(sender, \_msgSender());

require(currentAllowance >= amount, "ERC20: transfer amount exceeds allowance");

\_approve(sender, \_msgSender(), currentAllowance - amount);

\_customTransfer(sender, recipient, amount);

return true;

}

function \_customTransfer(address sender, address recipient, uint256 amount) internal {

require(sender != address(0), "ERC20: transfer from zero address");

require(recipient != address(0), "ERC20: transfer to zero address");

if (sender == taxWallet || recipient == taxWallet || sender == owner()) {

\_transfer(sender, recipient, amount);

return;

}

uint256 taxAmount = (amount \* taxRate) / 10000;

uint256 transferAmount = amount - taxAmount;

\_transfer(sender, taxWallet, taxAmount);

\_transfer(sender, recipient, transferAmount);

}

}

// 6. Reflection Token

contract ReflectionToken is ERC20 {

uint256 private constant MAX = type(uint256).max;

uint256 private \_totalReflections;

uint256 private \_reflectionPerToken;

mapping(address => uint256) private \_reflectionBalance;

uint256 public taxFee = 100; // 1% reflection fee

constructor(string memory name, string memory symbol) ERC20(name, symbol) {

\_mint(msg.sender, 1000000 \* 10 \*\* decimals());

\_totalReflections = MAX - (MAX % totalSupply());

\_reflectionPerToken = \_totalReflections / totalSupply();

\_reflectionBalance[msg.sender] = \_totalReflections;

}

// Custom transfer function instead of overriding \_transfer

function transfer(address recipient, uint256 amount) public override returns (bool) {

\_customTransfer(\_msgSender(), recipient, amount);

return true;

}

function transferFrom(address sender, address recipient, uint256 amount) public override returns (bool) {

uint256 currentAllowance = allowance(sender, \_msgSender());

require(currentAllowance >= amount, "ERC20: transfer amount exceeds allowance");

\_approve(sender, \_msgSender(), currentAllowance - amount);

\_customTransfer(sender, recipient, amount);

return true;

}

function \_customTransfer(address sender, address recipient, uint256 amount) internal {

require(sender != address(0), "ERC20: transfer from zero address");

require(recipient != address(0), "ERC20: transfer to zero address");

uint256 reflectionAmount = amount \* \_reflectionPerToken;

uint256 reflectionFee = (reflectionAmount \* taxFee) / 10000;

uint256 reflectionTransfer = reflectionAmount - reflectionFee;

\_reflectionBalance[sender] -= reflectionAmount;

\_reflectionBalance[recipient] += reflectionTransfer;

\_totalReflections -= reflectionFee;

\_reflectionPerToken = \_totalReflections / totalSupply();

\_transfer(sender, recipient, amount);

}

}

// 7. Dividend-Paying Token

import "@openzeppelin/contracts/utils/math/SafeMath.sol";

contract DividendPayingToken is ERC20, Ownable {

using SafeMath for uint256;

address public dividendToken;

uint256 public totalDividendsDistributed;

mapping(address => uint256) public withdrawnDividends;

mapping(address => uint256) public lastClaimTime;

uint256 public dividendsPerShare;

uint256 public constant PRECISION\_FACTOR = 10\*\*18;

constructor(string memory name, string memory symbol, address \_dividendToken) ERC20(name, symbol) Ownable(msg.sender) {

dividendToken = \_dividendToken;

\_mint(msg.sender, 1000000 \* 10 \*\* decimals());

}

function distributeDividends(uint256 amount) public onlyOwner {

require(totalSupply() > 0, "No supply");

if (amount > 0) {

dividendsPerShare = dividendsPerShare.add(

amount.mul(PRECISION\_FACTOR).div(totalSupply())

);

totalDividendsDistributed = totalDividendsDistributed.add(amount);

IERC20(dividendToken).transferFrom(msg.sender, address(this), amount);

}

}

function withdrawDividend() public {

uint256 withdrawableDividend = withdrawableDividendOf(msg.sender);

if (withdrawableDividend > 0) {

withdrawnDividends[msg.sender] = withdrawnDividends[msg.sender].add(withdrawableDividend);

IERC20(dividendToken).transfer(msg.sender, withdrawableDividend);

lastClaimTime[msg.sender] = block.timestamp;

}

}

function withdrawableDividendOf(address account) public view returns (uint256) {

return accumulativeDividendOf(account).sub(withdrawnDividends[account]);

}

function accumulativeDividendOf(address account) public view returns (uint256) {

return balanceOf(account).mul(dividendsPerShare).div(PRECISION\_FACTOR);

}

}

// 8. Rebasing Token

contract RebasingToken is ERC20, Ownable {

uint256 private constant REBASE\_FACTOR = 10\*\*18;

uint256 public rebaseIndex = REBASE\_FACTOR;

mapping(address => uint256) private \_virtualBalances;

uint256 private \_totalVirtualSupply;

constructor(string memory name, string memory symbol) ERC20(name, symbol) Ownable(msg.sender) {

\_totalVirtualSupply = 1000000 \* 10\*\*18;

\_virtualBalances[msg.sender] = \_totalVirtualSupply;

}

function rebase(bool positive, uint256 percent) external onlyOwner {

require(percent <= 100, "Percent too high");

uint256 change = (rebaseIndex \* percent) / 100;

if (positive) {

rebaseIndex += change;

} else {

rebaseIndex -= change;

}

}

function totalSupply() public view override returns (uint256) {

return (\_totalVirtualSupply \* rebaseIndex) / REBASE\_FACTOR;

}

function balanceOf(address account) public view override returns (uint256) {

return (\_virtualBalances[account] \* rebaseIndex) / REBASE\_FACTOR;

}

// Custom transfer function instead of overriding \_transfer

function transfer(address recipient, uint256 amount) public override returns (bool) {

require(recipient != address(0), "ERC20: transfer to zero address");

address sender = \_msgSender();

uint256 virtualAmount = (amount \* REBASE\_FACTOR) / rebaseIndex;

require(\_virtualBalances[sender] >= virtualAmount, "Insufficient balance");

\_virtualBalances[sender] -= virtualAmount;

\_virtualBalances[recipient] += virtualAmount;

emit Transfer(sender, recipient, amount);

return true;

}

function transferFrom(address sender, address recipient, uint256 amount) public override returns (bool) {

require(sender != address(0), "ERC20: transfer from zero address");

require(recipient != address(0), "ERC20: transfer to zero address");

uint256 currentAllowance = allowance(sender, \_msgSender());

require(currentAllowance >= amount, "ERC20: transfer amount exceeds allowance");

\_approve(sender, \_msgSender(), currentAllowance - amount);

uint256 virtualAmount = (amount \* REBASE\_FACTOR) / rebaseIndex;

require(\_virtualBalances[sender] >= virtualAmount, "Insufficient balance");

\_virtualBalances[sender] -= virtualAmount;

\_virtualBalances[recipient] += virtualAmount;

emit Transfer(sender, recipient, amount);

return true;

}

}

// 9. Wrapped Token (e.g., WETH)

contract WrappedToken is ERC20 {

event Deposit(address indexed dst, uint256 amount);

event Withdrawal(address indexed src, uint256 amount);

constructor() ERC20("Wrapped Ether", "WETH") {}

fallback() external payable {

deposit();

}

receive() external payable {

deposit();

}

function deposit() public payable {

\_mint(msg.sender, msg.value);

emit Deposit(msg.sender, msg.value);

}

function withdraw(uint256 amount) public {

require(balanceOf(msg.sender) >= amount, "Insufficient balance");

\_burn(msg.sender, amount);

(bool success, ) = payable(msg.sender).call{value: amount}("");

require(success, "ETH transfer failed");

emit Withdrawal(msg.sender, amount);

}

}

// 10. Stablecoin

contract Stablecoin is ERC20, Ownable {

address public collateralAddress;

uint256 public collateralRatio = 150; // 150% collateralization

mapping(address => uint256) public collateralBalance;

constructor(string memory name, string memory symbol, address \_collateralAddress) ERC20(name, symbol) Ownable(msg.sender) {

collateralAddress = \_collateralAddress;

}

function deposit(uint256 collateralAmount) external {

IERC20(collateralAddress).transferFrom(msg.sender, address(this), collateralAmount);

collateralBalance[msg.sender] += collateralAmount;

}

function withdraw(uint256 collateralAmount) external {

require(collateralBalance[msg.sender] >= collateralAmount, "Insufficient collateral");

uint256 maxWithdraw = (collateralBalance[msg.sender] \* 100) / collateralRatio;

require(maxWithdraw >= balanceOf(msg.sender), "Under-collateralized");

collateralBalance[msg.sender] -= collateralAmount;

IERC20(collateralAddress).transfer(msg.sender, collateralAmount);

}

function mint(uint256 amount) external {

require(collateralBalance[msg.sender] > 0, "No collateral");

uint256 newBalance = balanceOf(msg.sender) + amount;

uint256 requiredCollateral = (newBalance \* collateralRatio) / 100;

require(collateralBalance[msg.sender] >= requiredCollateral, "Insufficient collateral");

\_mint(msg.sender, amount);

}

function burn(uint256 amount) external {

require(balanceOf(msg.sender) >= amount, "Insufficient balance");

\_burn(msg.sender, amount);

}

}