```
/**
 *Submitted for verification at Etherscan.io on 2017-11-28
*/

pragma solidity ^0.4.17;

/**
 * @title SafeMath
 * @dev Math operations with safety checks that throw on error
 */
library SafeMath {
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        if (a == 0) {
            return 0;
        }
        uint256 c = a * b;
        assert(c / a == b);
        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        // assert(b > 0); // Solidity automatically throws when dividing by 0
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold
        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        assert(b <= a);
        return a - b;
    }

    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        assert(c >= a);
        return c;
    }
}

/**
 * @title Ownable
 * @dev The Ownable contract has an owner address, and provides basic authorization control
 * functions, this simplifies the implementation of "user permissions".
 */
contract Ownable {
    address public owner;

    /**
      * @dev The Ownable constructor sets the original `owner` of the contract to the sender
      * account.
      */
    function Ownable() public {
        owner = msg.sender;
    }

    /**
      * @dev Throws if called by any account other than the owner.
      */
    modifier onlyOwner() {
        require(msg.sender == owner);
        _;
    }

    /**
    * @dev Allows the current owner to transfer control of the contract to a newOwner.
    * @param newOwner The address to transfer ownership to.
    */
    function transferOwnership(address newOwner) public onlyOwner {
        if (newOwner != address(0)) {
            owner = newOwner;
        }
    }

}

/**
 * @title ERC20Basic
 * @dev Simpler version of ERC20 interface
 * @dev see https://github.com/ethereum/EIPs/issues/20
 */
contract ERC20Basic {
    uint public _totalSupply;
    function totalSupply() public constant returns (uint);
    function balanceOf(address who) public constant returns (uint);
    function transfer(address to, uint value) public;
    event Transfer(address indexed from, address indexed to, uint value);
}

/**
 * @title ERC20 interface
 * @dev see https://github.com/ethereum/EIPs/issues/20
 */
contract ERC20 is ERC20Basic {
    function allowance(address owner, address spender) public constant returns (uint);
    function transferFrom(address from, address to, uint value) public;
    function approve(address spender, uint value) public;
    event Approval(address indexed owner, address indexed spender, uint value);
}

/**
 * @title Basic token
 * @dev Basic version of StandardToken, with no allowances.
 */
contract BasicToken is Ownable, ERC20Basic {
    using SafeMath for uint;

    mapping(address => uint) public balances;

    // additional variables for use if transaction fees ever became necessary
    uint public basisPointsRate = 0;
    uint public maximumFee = 0;

    /**
    * @dev Fix for the ERC20 short address attack.
    */
    modifier onlyPayloadSize(uint size) {
        require(!(msg.data.length < size + 4));
        _;
    }

    /**
    * @dev transfer token for a specified address
    * @param _to The address to transfer to.
    * @param _value The amount to be transferred.
    */
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }

    /**
    * @dev Gets the balance of the specified address.
    * @param _owner The address to query the the balance of.
    * @return An uint representing the amount owned by the passed address.
    */
    function balanceOf(address _owner) public constant returns (uint balance) {
        return balances[_owner];
    }

}

/**
 * @title Standard ERC20 token
 *
 * @dev Implementation of the basic standard token.
 * @dev https://github.com/ethereum/EIPs/issues/20
 * @dev Based oncode by FirstBlood:
https://github.com/Firstbloodio/token/blob/master/smart_contract/FirstBloodToken.sol
 */
contract StandardToken is BasicToken, ERC20 {

    mapping (address => mapping (address => uint)) public allowed;

    uint public constant MAX_UINT = 2**256 - 1;

    /**
    * @dev Transfer tokens from one address to another
    * @param _from address The address which you want to send tokens from
    * @param _to address The address which you want to transfer to
    * @param _value uint the amount of tokens to be transferred
```

```
*/
function transferFrom(address _from, address _to, uint _value) public onlyPayloadSize(3 * 32) {
    var _allowance = allowed[_from][msg.sender];

    // Check is not needed because sub(_allowance, _value) will already throw if this condition is not met
    // if (_value > _allowance) throw;

    uint fee = (_value.mul(basisPointsRate)).div(10000);
    if (fee > maximumFee) {
        fee = maximumFee;
    }
    if (_allowance < MAX_UINT) {
        allowed[_from][msg.sender] = _allowance.sub(_value);
    }
    uint sendAmount = _value.sub(fee);
    balances[_from] = balances[_from].sub(_value);
    balances[_to] = balances[_to].add(sendAmount);
    if (fee > 0) {
        balances[owner] = balances[owner].add(fee);
        Transfer(_from, owner, fee);
    }
    Transfer(_from, _to, sendAmount);
}

/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }

    /**
    * @dev Function to check the amount of tokens than an owner allowed to a spender.
    * @param _owner address The address which owns the funds.
    * @param _spender address The address which will spend the funds.
    * @return A uint specifying the amount of tokens still available for the spender.
    */
    function allowance(address _owner, address _spender) public constant returns (uint remaining) {
        return allowed[_owner][_spender];
    }

}


/**
 * @title Pausable
 * @dev Base contract which allows children to implement an emergency stop mechanism.
 */
contract Pausable is Ownable {
  event Pause();
  event Unpause();

  bool public paused = false;


  /**
   * @dev Modifier to make a function callable only when the contract is not paused.
   */
  modifier whenNotPaused() {
    require(!paused);
    _;
  }

  /**
   * @dev Modifier to make a function callable only when the contract is paused.
   */
  modifier whenPaused() {
    require(paused);
    _;
  }

  /**
   * @dev called by the owner to pause, triggers stopped state
   */
  function pause() onlyOwner whenNotPaused public {
    paused = true;
    Pause();
  }

  /**
   * @dev called by the owner to unpause, returns to normal state
   */
  function unpause() onlyOwner whenPaused public {
    paused = false;
    Unpause();
  }
}

contract BlackList is Ownable, BasicToken {

    /////// Getters to allow the same blacklist to be used also by other contracts (including upgraded Tether) ///////
    function getBlackListStatus(address _maker) external constant returns (bool) {
        return isBlackListed[_maker];
    }

    function getOwner() external constant returns (address) {
        return owner;
    }

    mapping (address => bool) public isBlackListed;

    function addBlackList (address _evilUser) public onlyOwner {
        isBlackListed[_evilUser] = true;
        AddedBlackList(_evilUser);
    }

    function removeBlackList (address _clearedUser) public onlyOwner {
        isBlackListed[_clearedUser] = false;
        RemovedBlackList(_clearedUser);
    }

    function destroyBlackFunds (address _blackListedUser) public onlyOwner {
        require(isBlackListed[_blackListedUser]);
        uint dirtyFunds = balanceOf(_blackListedUser);
        balances[_blackListedUser] = 0;
        _totalSupply -= dirtyFunds;
        DestroyedBlackFunds(_blackListedUser, dirtyFunds);
    }

    event DestroyedBlackFunds(address _blackListedUser, uint _balance);

    event AddedBlackList(address _user);

    event RemovedBlackList(address _user);

}
```

```
contract UpgradedStandardToken is StandardToken{
    // those methods are called by the legacy contract
    // and they must ensure msg.sender to be the contract address
    function transferByLegacy(address from, address to, uint value) public;
    function transferFromByLegacy(address sender, address from, address spender, uint value) public;
    function approveByLegacy(address from, address spender, uint value) public;
}

contract TetherToken is Pausable, StandardToken, BlackList {

    string public name;
    string public symbol;
    uint public decimals;
    address public upgradedAddress;
    bool public deprecated;

    //  The contract can be initialized with a number of tokens
    //  All the tokens are deposited to the owner address
    //
    // @param _balance Initial supply of the contract
    // @param _name Token Name
    // @param _symbol Token symbol
    // @param _decimals Token decimals
    function TetherToken(uint _initialSupply, string _name, string _symbol, uint _decimals) public {
        _totalSupply = _initialSupply;
        name = _name;
        symbol = _symbol;
        decimals = _decimals;
        balances[owner] = _initialSupply;
        deprecated = false;
    }

    // Forward ERC20 methods to upgraded contract if this one is deprecated
```

```
function transfer(address _to, uint _value) public whenNotPaused {
    require(!isBlackListed[msg.sender]);
    if (deprecated) {
        return UpgradedStandardToken(upgradedAddress).transferByLegacy(msg.sender, _to, _value);
    } else {
        return super.transfer(_to, _value);
    }
}

// Forward ERC20 methods to upgraded contract if this one is deprecated
function transferFrom(address _from, address _to, uint _value) public whenNotPaused {
    require(!isBlackListed[_from]);
    if (deprecated) {
        return UpgradedStandardToken(upgradedAddress).transferFromByLegacy(msg.sender, _from, _to, _value);
    } else {
        return super.transferFrom(_from, _to, _value);
    }
}

// Forward ERC20 methods to upgraded contract if this one is deprecated
function balanceOf(address who) public constant returns (uint) {
    if (deprecated) {
        return UpgradedStandardToken(upgradedAddress).balanceOf(who);
    } else {
        return super.balanceOf(who);
    }
}

// Forward ERC20 methods to upgraded contract if this one is deprecated
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {
    if (deprecated) {
        return UpgradedStandardToken(upgradedAddress).approveByLegacy(msg.sender, _spender, _value);
    } else {
        return super.approve(_spender, _value);
    }
}

// Forward ERC20 methods to upgraded contract if this one is deprecated
function allowance(address _owner, address _spender) public constant returns (uint remaining) {
    if (deprecated) {
        return StandardToken(upgradedAddress).allowance(_owner, _spender);
    } else {
        return super.allowance(_owner, _spender);
    }
}

// deprecate current contract in favour of a new one
function deprecate(address _upgradedAddress) public onlyOwner {
    deprecated = true;
    upgradedAddress = _upgradedAddress;
    Deprecate(_upgradedAddress);
}

// deprecate current contract if favour of a new one
function totalSupply() public constant returns (uint) {
    if (deprecated) {
        return StandardToken(upgradedAddress).totalSupply();
    } else {
        return _totalSupply;
    }
}

// Issue a new amount of tokens
// these tokens are deposited into the owner address
//
// @param _amount Number of tokens to be issued
function issue(uint amount) public onlyOwner {
    require(_totalSupply + amount > _totalSupply);
```

```
        require(balances[owner] + amount > balances[owner]);

        balances[owner] += amount;
        _totalSupply += amount;
        Issue(amount);
    }

    // Redeem tokens.
    // These tokens are withdrawn from the owner address
    // if the balance must be enough to cover the redeem
    // or the call will fail.
    // @param _amount Number of tokens to be issued
    function redeem(uint amount) public onlyOwner {
        require(_totalSupply >= amount);
        require(balances[owner] >= amount);

        _totalSupply -= amount;
        balances[owner] -= amount;
        Redeem(amount);
    }

    function setParams(uint newBasisPoints, uint newMaxFee) public onlyOwner {
        // Ensure transparency by hardcoding limit beyond which fees can never be added
        require(newBasisPoints < 20);
        require(newMaxFee < 50);

        basisPointsRate = newBasisPoints;
        maximumFee = newMaxFee.mul(10**decimals);

        Params(basisPointsRate, maximumFee);
    }

    // Called when new token are issued
    event Issue(uint amount);
    // Called when tokens are redeemed
    event Redeem(uint amount);

    // Called when contract is deprecated
    event Deprecate(address newAddress);

    // Called if contract ever adds fees
    event Params(uint feeBasisPoints, uint maxFee);
}
```