```
pragma solidity 0.6.2;

import "./erc721.sol";
import "./erc721-token-receiver.sol";
import "../math/safe-math.sol";
import "../utils/supports-interface.sol";
import "../utils/address-utils.sol";

/**
 * @dev Implementation of ERC-721 non-fungible token standard.
 */
contract NFToken is
  ERC721,
  SupportsInterface
{
  using SafeMath for uint256;
  using AddressUtils for address;

  /**
   * List of revert message codes. Implementing dApp should handle showing the correct message.
   * Based on 0xcert framework error codes.
   */
  string constant ZERO_ADDRESS = "003001";
  string constant NOT_VALID_NFT = "003002";
  string constant NOT_OWNER_OR_OPERATOR = "003003";
  string constant NOT_OWNER_APPROWED_OR_OPERATOR = "003004";
  string constant NOT_ABLE_TO_RECEIVE_NFT = "003005";
  string constant NFT_ALREADY_EXISTS = "003006";
  string constant NOT_OWNER = "003007";
  string constant IS_OWNER = "003008";

  /**
   * @dev Magic value of a smart contract that can recieve NFT.
   * Equal to: bytes4(keccak256("onERC721Received(address,address,uint256,bytes)")).
   */
  bytes4 internal constant MAGIC_ON_ERC721_RECEIVED = 0x150b7a02;

  /**
   * @dev A mapping from NFT ID to the address that owns it.
   */
  mapping (uint256 => address) internal idToOwner;

  /**
   * @dev Mapping from NFT ID to approved address.
   */
  mapping (uint256 => address) internal idToApproval;

   /**
   * @dev Mapping from owner address to count of his tokens.
```

20160408090312
1515820203003A
電子署名者：
20160408090312
1515820203003A
日付：2020.03.01
23:24:15 +09'00'

```
  */
  mapping (address => uint256) private ownerToNFTokenCount;

  /**
   * @dev Mapping from owner address to mapping of operator addresses.
   */
  mapping (address => mapping (address => bool)) internal ownerToOperators;

  /**
   * @dev Emits when ownership of any NFT changes by any mechanism. This event emits when
NFTs are
   * created (`from` == 0) and destroyed (`to` == 0). Exception: during contract creation, any
   * number of NFTs may be created and assigned without emitting Transfer. At the time of any
   * transfer, the approved address for that NFT (if any) is reset to none.
   * @param _from Sender of NFT (if address is zero address it indicates token creation).
   * @param _to Receiver of NFT (if address is zero address it indicates token destruction).
   * @param _tokenId The NFT that got transfered.
   */
  event Transfer(
    address indexed _from,
    address indexed _to,
    uint256 indexed _tokenId
  );

  /**
   * @dev This emits when the approved address for an NFT is changed or reaffirmed. The zero
   * address indicates there is no approved address. When a Transfer event emits, this also
   * indicates that the approved address for that NFT (if any) is reset to none.
   * @param _owner Owner of NFT.
   * @param _approved Address that we are approving.
   * @param _tokenId NFT which we are approving.
   */
  event Approval(
    address indexed _owner,
    address indexed _approved,
    uint256 indexed _tokenId
  );

  /**
   * @dev This emits when an operator is enabled or disabled for an owner. The operator can manage
   * all NFTs of the owner.
   * @param _owner Owner of NFT.
   * @param _operator Address to which we are setting operator rights.
   * @param _approved Status of operator rights(true if operator rights are given and false if
   * revoked).
   */
  event ApprovalForAll(
    address indexed _owner,
    address indexed _operator,
```

```
    bool _approved
  );

  /**
   * @dev Guarantees that the msg.sender is an owner or operator of the given NFT.
   * @param _tokenId ID of the NFT to validate.
   */
  modifier canOperate(
    uint256 _tokenId
  )
  {
    address tokenOwner = idToOwner[_tokenId];
    require(tokenOwner == msg.sender || ownerToOperators[tokenOwner][msg.sender],
NOT_OWNER_OR_OPERATOR);
    _;
  }

  /**
   * @dev Guarantees that the msg.sender is allowed to transfer NFT.
   * @param _tokenId ID of the NFT to transfer.
   */
  modifier canTransfer(
    uint256 _tokenId
  )
  {
    address tokenOwner = idToOwner[_tokenId];
    require(
      tokenOwner == msg.sender
      || idToApproval[_tokenId] == msg.sender
      || ownerToOperators[tokenOwner][msg.sender],
      NOT_OWNER_APPROWED_OR_OPERATOR
    );
    _;
  }

  /**
   * @dev Guarantees that _tokenId is a valid Token.
   * @param _tokenId ID of the NFT to validate.
   */
  modifier validNFToken(
    uint256 _tokenId
  )
  {
    require(idToOwner[_tokenId] != address(0), NOT_VALID_NFT);
    _;
  }

  /**
   * @dev Contract constructor.
```

```
 */
constructor()
   public
{
   supportedInterfaces[0x80ac58cd] = true; // ERC721
}


/** ERC721LT
 * @dev TransferPermission setter getter.
 * @param  bool _transferPermission  if false then we can not transfer NFT.
 */

// can we transfer NFT ?   true is OK , false is NG .this varriable is changed by contract owner.
bool private _transferPermission = false ;

//TransferPermission setter getter
function getTransferPermission()public view  returns (bool) {
   return _transferPermission;
}
function _setTransferPermission( bool _newState) internal {
   _transferPermission = _newState;
}

/** ERC721LT
 * @dev Transfers the ownership of an NFT from one address to another address. This function can
 * be changed to payable.
 * @notice Throws unless `msg.sender` is the current owner, an authorized operator, or the
 * approved address for this NFT. Throws if `_from` is not the current owner. Throws if `_to` is
 * the zero address. Throws if `_tokenId` is not a valid NFT. When transfer is complete, this
 * function checks if `_to` is a smart contract (code size > 0). If so, it calls
 * `onERC721Received` on `_to` and throws if the return value is not
 * `bytes4(keccak256("onERC721Received(address,uint256,bytes)"))`.
 * @param _from The current owner of the NFT.
 * @param _to The new owner.
 * @param _tokenId The NFT to transfer.
 * @param _data Additional data with no specified format, sent in call to `_to`.
 */
function safeTransferFrom(
   address _from,
   address _to,
   uint256 _tokenId,
   bytes calldata _data
)
   external
   override
{
   require(_transferPermission == true);
   _safeTransferFrom(_from, _to, _tokenId, _data);
```

```
}

/**
 * @dev Transfers the ownership of an NFT from one address to another address. This function can
 * be changed to payable.
 * @notice This works identically to the other function with an extra data parameter, except this
 * function just sets data to ""
 * @param _from The current owner of the NFT.
 * @param _to The new owner.
 * @param _tokenId The NFT to transfer.
 */
function safeTransferFrom(
  address _from,
  address _to,
  uint256 _tokenId
)
  external
  override
{
  require(_transferPermission == true);
  _safeTransferFrom(_from, _to, _tokenId, "");
}

/**
 * @dev Throws unless `msg.sender` is the current owner, an authorized operator, or the approved
 * address for this NFT. Throws if `_from` is not the current owner. Throws if `_to` is the zero
 * address. Throws if `_tokenId` is not a valid NFT. This function can be changed to payable.
 * @notice The caller is responsible to confirm that `_to` is capable of receiving NFTs or else
 * they maybe be permanently lost.
 * @param _from The current owner of the NFT.
 * @param _to The new owner.
 * @param _tokenId The NFT to transfer.
 */
function transferFrom(
  address _from,
  address _to,
  uint256 _tokenId
)
  external
  override
  canTransfer(_tokenId)
  validNFToken(_tokenId)
{
  require(_transferPermission == true);
  address tokenOwner = idToOwner[_tokenId];
  require(tokenOwner == _from, NOT_OWNER);
  require(_to != address(0), ZERO_ADDRESS);

  _transfer(_to, _tokenId);
```

```
}

/**
 * @dev Set or reaffirm the approved address for an NFT. This function can be changed to payable.
 * @notice The zero address indicates there is no approved address. Throws unless `msg.sender` is
 * the current NFT owner, or an authorized operator of the current owner.
 * @param _approved Address to be approved for the given NFT ID.
 * @param _tokenId ID of the token to be approved.
 */
function approve(
  address _approved,
  uint256 _tokenId
)
  external
  override
  canOperate(_tokenId)
  validNFToken(_tokenId)
{
  address tokenOwner = idToOwner[_tokenId];
  require(_approved != tokenOwner, IS_OWNER);

  idToApproval[_tokenId] = _approved;
  emit Approval(tokenOwner, _approved, _tokenId);
}

/**
 * @dev Enables or disables approval for a third party ("operator") to manage all of
 * `msg.sender`'s assets. It also emits the ApprovalForAll event.
 * @notice This works even if sender doesn't own any tokens at the time.
 * @param _operator Address to add to the set of authorized operators.
 * @param _approved True if the operators is approved, false to revoke approval.
 */
function setApprovalForAll(
  address _operator,
  bool _approved
)
  external
  override
{
  ownerToOperators[msg.sender][_operator] = _approved;
  emit ApprovalForAll(msg.sender, _operator, _approved);
}

/**
 * @dev Returns the number of NFTs owned by `_owner`. NFTs assigned to the zero address are
 * considered invalid, and this function throws for queries about the zero address.
 * @param _owner Address for whom to query the balance.
 * @return Balance of _owner.
 */
```

```
function balanceOf(
   address _owner
)
   external
   override
   view
   returns (uint256)
{
   require(_owner != address(0), ZERO_ADDRESS);
   return _getOwnerNFTCount(_owner);
}

/**
 * @dev Returns the address of the owner of the NFT. NFTs assigned to zero address are considered
 * invalid, and queries about them do throw.
 * @param _tokenId The identifier for an NFT.
 * @return _owner Address of _tokenId owner.
 */
function ownerOf(
   uint256 _tokenId
)
   external
   override
   view
   returns (address _owner)
{
   _owner = idToOwner[_tokenId];
   require(_owner != address(0), NOT_VALID_NFT);
}

/**
 * @dev Get the approved address for a single NFT.
 * @notice Throws if `_tokenId` is not a valid NFT.
 * @param _tokenId ID of the NFT to query the approval of.
 * @return Address that _tokenId is approved for.
 */
function getApproved(
   uint256 _tokenId
)
   external
   override
   view
   validNFToken(_tokenId)
   returns (address)
{
   return idToApproval[_tokenId];
}

/**
```

```
  * @dev Checks if `_operator` is an approved operator for `_owner`.
  * @param _owner The address that owns the NFTs.
  * @param _operator The address that acts on behalf of the owner.
  * @return True if approved for all, false otherwise.
  */
function isApprovedForAll(
    address _owner,
    address _operator
)
    external
    override
    view
    returns (bool)
{
    return ownerToOperators[_owner][_operator];
}

/**
  * @dev Actually preforms the transfer.
  * @notice Does NO checks.
  * @param _to Address of a new owner.
  * @param _tokenId The NFT that is being transferred.
  */
function _transfer(
    address _to,
    uint256 _tokenId
)
    internal
{
    address from = idToOwner[_tokenId];
    _clearApproval(_tokenId);

    _removeNFToken(from, _tokenId);
    _addNFToken(_to, _tokenId);

    emit Transfer(from, _to, _tokenId);
}

/**
  * @dev Mints a new NFT.
  * @notice This is an internal function which should be called from user-implemented external
  * mint function. Its purpose is to show and properly initialize data structures when using this
  * implementation.
  * @param _to The address that will own the minted NFT.
  * @param _tokenId of the NFT to be minted by the msg.sender.
  */
function _mint(
    address _to,
    uint256 _tokenId
```

```
  )
    internal
    virtual
  {
    require(_to != address(0), ZERO_ADDRESS);
    require(idToOwner[_tokenId] == address(0), NFT_ALREADY_EXISTS);

    _addNFToken(_to, _tokenId);

    emit Transfer(address(0), _to, _tokenId);
  }

  /**
   * @dev Burns a NFT.
   * @notice This is an internal function which should be called from user-implemented external burn
   * function. Its purpose is to show and properly initialize data structures when using this
   * implementation. Also, note that this burn implementation allows the minter to re-mint a burned
   * NFT.
   * @param _tokenId ID of the NFT to be burned.
   */
  function _burn(
    uint256 _tokenId
  )
    internal
    virtual
    validNFToken(_tokenId)
  {
    address tokenOwner = idToOwner[_tokenId];
    _clearApproval(_tokenId);
    _removeNFToken(tokenOwner, _tokenId);
    emit Transfer(tokenOwner, address(0), _tokenId);
  }

  /**
   * @dev Removes a NFT from owner.
   * @notice Use and override this function with caution. Wrong usage can have serious
consequences.
   * @param _from Address from wich we want to remove the NFT.
   * @param _tokenId Which NFT we want to remove.
   */
  function _removeNFToken(
    address _from,
    uint256 _tokenId
  )
    internal
    virtual
  {
    require(idToOwner[_tokenId] == _from, NOT_OWNER);
    ownerToNFTokenCount[_from] = ownerToNFTokenCount[_from] - 1;
```

```
    delete idToOwner[_tokenId];
  }

  /**
   * @dev Assignes a new NFT to owner.
   * @notice Use and override this function with caution. Wrong usage can have serious
consequences.
   * @param _to Address to wich we want to add the NFT.
   * @param _tokenId Which NFT we want to add.
   */
  function _addNFToken(
    address _to,
    uint256 _tokenId
  )
    internal
    virtual
  {
    require(idToOwner[_tokenId] == address(0), NFT_ALREADY_EXISTS);

    idToOwner[_tokenId] = _to;
    ownerToNFTokenCount[_to] = ownerToNFTokenCount[_to].add(1);
  }

  /**
   * @dev Helper function that gets NFT count of owner. This is needed for overriding in enumerable
   * extension to remove double storage (gas optimization) of owner nft count.
   * @param _owner Address for whom to query the count.
   * @return Number of _owner NFTs.
   */
  function _getOwnerNFTCount(
    address _owner
  )
    internal
    virtual
    view
    returns (uint256)
  {
    return ownerToNFTokenCount[_owner];
  }

  /**
   * @dev Actually perform the safeTransferFrom.
   * @param _from The current owner of the NFT.
   * @param _to The new owner.
   * @param _tokenId The NFT to transfer.
   * @param _data Additional data with no specified format, sent in call to `_to`.
   */
  function _safeTransferFrom(
    address _from,
```

```
    address _to,
    uint256 _tokenId,
    bytes memory _data
  )
    private
    canTransfer(_tokenId)
    validNFToken(_tokenId)
  {
    address tokenOwner = idToOwner[_tokenId];
    require(tokenOwner == _from, NOT_OWNER);
    require(_to != address(0), ZERO_ADDRESS);

    _transfer(_to, _tokenId);

    if (_to.isContract())
    {
      bytes4 retval = ERC721TokenReceiver(_to).onERC721Received(msg.sender, _from, _tokenId,
_data);
      require(retval == MAGIC_ON_ERC721_RECEIVED, NOT_ABLE_TO_RECEIVE_NFT);
    }
  }

  /**
   * @dev Clears the current approval of a given NFT ID.
   * @param _tokenId ID of the NFT to be transferred.
   */
  function _clearApproval(
    uint256 _tokenId
  )
    private
  {
    if (idToApproval[_tokenId] != address(0))
    {
      delete idToApproval[_tokenId];
    }
  }

}
```