CrowdSale contract:

In crowd sale smart-contract, I have imported three .sol files. One is **IERC20**, second is **Ownable** and third is **INonfungiblePositionManager,** and in **INonfungiblePositionManager** I have imported 4 interfaces.

Version of solidity is **0.8.9** and in two interfaces I have also imported **abicoder v2** which allows structs, nested and dynamic variables to be passed into functions, returned from functions and emitted by events.

About CrowdSale contract:

Variables:

TokenId: Id of created pool.
Early_bonus_token: bonus tokens of early users.
Time_of_buy: buy time of tokens.
Early_users: set early users for early bonus.
Initial users: created for set of early_users.

Events:
TokenBuy: address of owner, address of buyer and address of token amount.
Set_sale: time_of_buy, early_bonus_of_token, initial_users.
IncreaseLiquidityCurrentRange: tokenId, add amount of token0, add amount of token1.

- Pass address in **INonfungiblePositionManager** of **NonfungiblePositionManager.**

Constructor:

Pass address of **TOKEN_A** contract and **TOKEN_B** contract.

Functions:

In the **increaseLiquidityCurrentRange** function I pass **tokenId** of token **amount of token0** and **amount of token 1**, and this function returns liquidity, **amount of token0** and **amount of token 1.**
This function will **approve NonfungiblePositionManager** and the **amount of token0** from **TOKEN_A**.
This function will **approve NonfungiblePositionManager** and the **amount of token1** from **TOKEN_B**.

In the **set_sale** function I am doing **set time, early_bonus** and **early_users_quantity** for buying tokens.

In the **reset_sale** function will reset the values of **set time, early_bonus** and **early_users_quantity**.

In the **resetInitialUsers** will reset the value of **inital_users**.

In the **buy_token** function user will input tokens_a in exchange of token_b then token_a will be transferred in contract and token_b will be transferred to user's address.

In the **getContractBalance**: function returns the amount of contract balance.

Receive function will be called, when anyone sends money on the contract address.

Explain Imports:

1. First import is from **openzeppelin** of **IERC20** that defines the functions and events that are required for the ERC20 token standard and in crowd-sale smart-contract, I am using two ERC20 token contracts **(TOKEN_A, TOKEN_B)**, those I created for swapping and I am using these contracts addresses in IERC20.

2. Second import is from **openzeppelin** of Ownable**.** The Ownable.sol contract provides the most basic single account ownership to a contract. Only one account will become the owner of the contract and can perform administration-related tasks.
   In contract I have used this for the set_sale function, because every person can't be a caller.

3. Third import is from uniswap v3-periphery of **INonfungiblePositionManager.**
   In this interface I have created four structs and seven functions.

   INonfungiblePositionManager:

   In **MintParams struct** I have used:
   token0: The address of the token0 for a specific pool
   token1: The address of the token1 for a specific pool
   Fee: The fee associated with the pool
   tickLower: The lower end of the tick range for the position

tickUpper: The higher end of the tick range for the position
amount0Desired: The amount of token0
amount1Desired: The amount of token1
amount0Min: The min amount of token0
amount1Min: The min amount of token1
Recipient: Recipient address
Deadline: transaction approval time

In **IncreaseLiquidityParams struct** I have used:
Token ID: The ID of the token that represents the position
amount0Desired: The amount of token0
amount1Desired: The amount of token1
amount0Min: The min amount of token0
amount1Min: The min amount of token1
Deadline: transaction approval time

In **decreaseLiquidityParams struct** I have used:
Token ID: The ID of the token that represents the position
Liquidity: The liquidity of the position
amount0Min: The min amount of token0
amount1Min: The min amount of token1
Deadline: transaction approval time

In **CollectParams struct** I have used:
token Id: The ID of the token that represents the position
Recipient: Recipient address
amount0Max: The max amount of token0
amount1Max: The max amount of token1

In **positions function** I am doing:
Returns the position information (nonce, operator, token0, token1, fee, tickLower, tickUpper, liquidity........) associated with a given token ID.

In **createAndInitializePoolIfNecessary function** I am doing:
Creates a new pool if it does not exist, then initializes if not initialized

In **mint function** I am doing:
Creates a new position wrapped in a NFT

In **increaseLiquidity function** I am doing:

Increases the amount of liquidity in a position, with tokens paid by the msg.sender

In **decreaseLiquidity function** I am doing:
Decreases the amount of liquidity in a position and accounts it to the position

In **collect function** I am doing:
Collects up to a maximum amount of fees owed to a specific position to the recipient

In **burn function** I am doing:
Burns a token ID, which deletes it from the NFT contract. The token must have 0 liquidity and all tokens must be collected first.

IERC20:

In this interface I have wrote two events and six functions:

Events:

Transfer: address of sender, address of receiver and amount.
Approval: address of owner, address of spender and value.

Functions:

totalSupply: amount of total supply of token.
balanceOf: value of input address.
Transfer: value transfer of (address to) input.
Allowance: Returns the remaining number of tokens that `spender` will be allowed to spend on behalf of `owner` through transferFrom. This is zero by default. This value changes when {approve} or {transferFrom} are called.

Approve: Sets `amount` as the allowance of `spender` over the caller's tokens. Returns a boolean value indicating whether the operation succeeded.

TransferFrom: Moves `amount` tokens from `from` to `to` using the allowance mechanism. `amount` is then deducted from the caller's allowance. Returns a boolean value indicating whether the operation succeeded.

Ownable:

In **Ownable contract** I have imported a **context.sol** file and in this file I have created an **abstract contract** in which I have created two functions one is **_msgSender** and second is **_msgData**.
Provides information about the current execution context, including the sender of the transaction and its data. While these are generally available via msg.sender and msg.data

In the Ownable **contract** I have written the address of _owner with private state.
Event of **OwnershipTransferred** in which put **previous owner address** and **new owner address.**
In **constructor:** Initializes the contract setting the deployer as the initial owner.
In **modifier** of **onlyOwner** throws if called by any account other than the owner.
In the **owner** function doing: Returns the address of the current owner.
In the _checkOwner function doing: Throws if the sender is not the owner.
In the **renounceOwnerShip** function doing: Renouncing ownership will leave the contract without an owner, thereby removing any functionality that is only available to the owner.
In the **transferOwnerShip** function doing: Transfers ownership of the contract to a new account (`newOwner`). Can only be called by the current owner.
In the **_transferOwnerShip** function doing: Transfers ownership of the contract to a new account (`newOwner`). Internal function without access restriction.