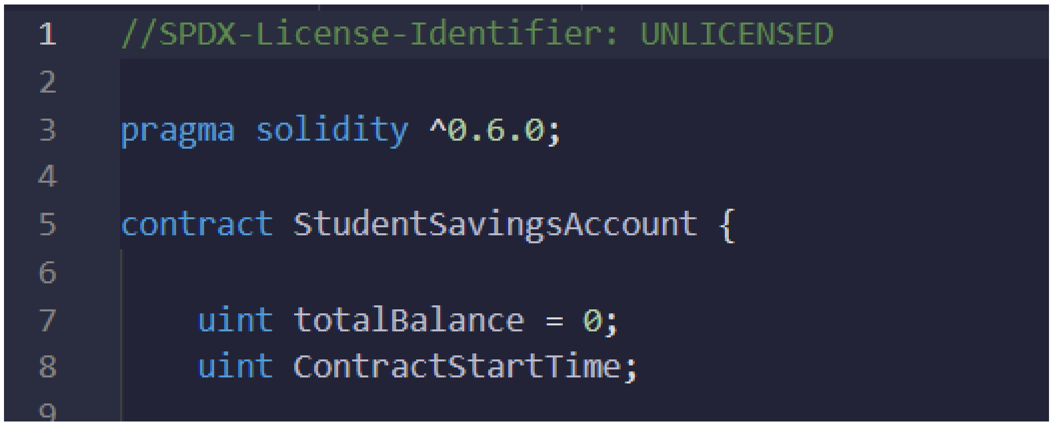
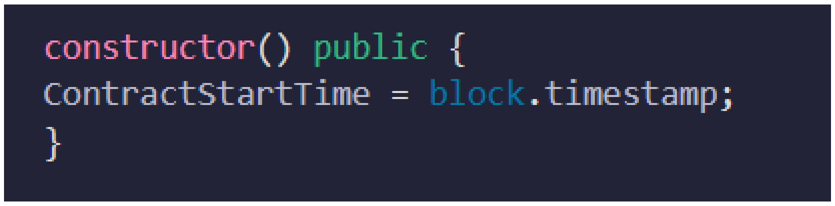
we created student savings account by using the keyword contract. Under the contract function, total balance showing the current balance and contract start time are involved with uint data type. The uint data type is an unsigned integer, meaning its value must be non-negative. (SOLIDITY COMMENTS, 2018)



```
//SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.6.0;

contract StudentSavingsAccount {

    uint totalBalance = 0;
    uint ContractStartTime;
```

Fig. 1: Creating contract for student savings account.

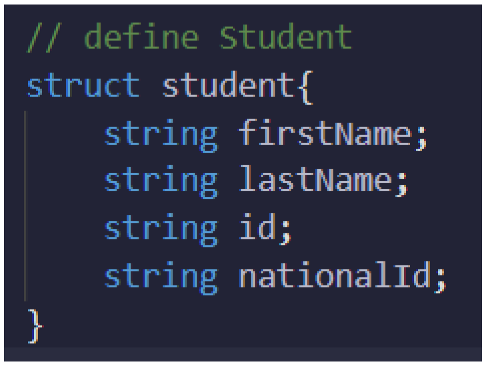
Constructor is an important function in solidity that is used by writing constructor keyword. It is also used to initialize state variables of a contract and for a contract we can use only one constructor. A constructor is used as public or internal for the whole program. Here contract start time is block.timestamp that refers the current time of the smart contract.



```
constructor() public {
ContractStartTime = block.timestamp;
}
```

Fig. 2: The constructor includes the current contract start time.

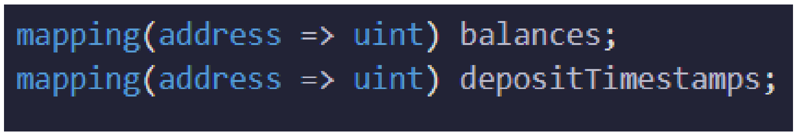
After completeing constructor function and its codings, we have defined student names, id number and their national id number to record their initial data into smart contracts.



```
// define Student
struct student{
    string firstName;
    string lastName;
    string id;
    string nationalId;
}
```

Fig. 3: Defining Student.

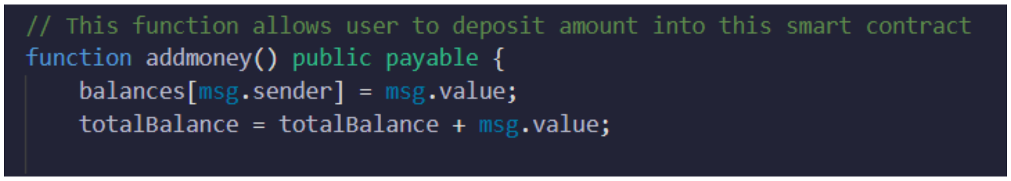
Mapping is type of dictionary where it stores the data in the form of key-value pairs, a key can be any of the built-in data types but reference types are not allowed while the value can be of any type. (Solidity – Mappings, 2022) So, it involves a key type as well as a value type. We have addressed the keywords balances and deposit time stamps in mapping. We will work further with these keywords in next steps.



```
mapping(address => uint) balances;
mapping(address => uint) depositTimestamps;
```

Fig 4: Mapping function shows addressing balances and depositTimestamps keywords.

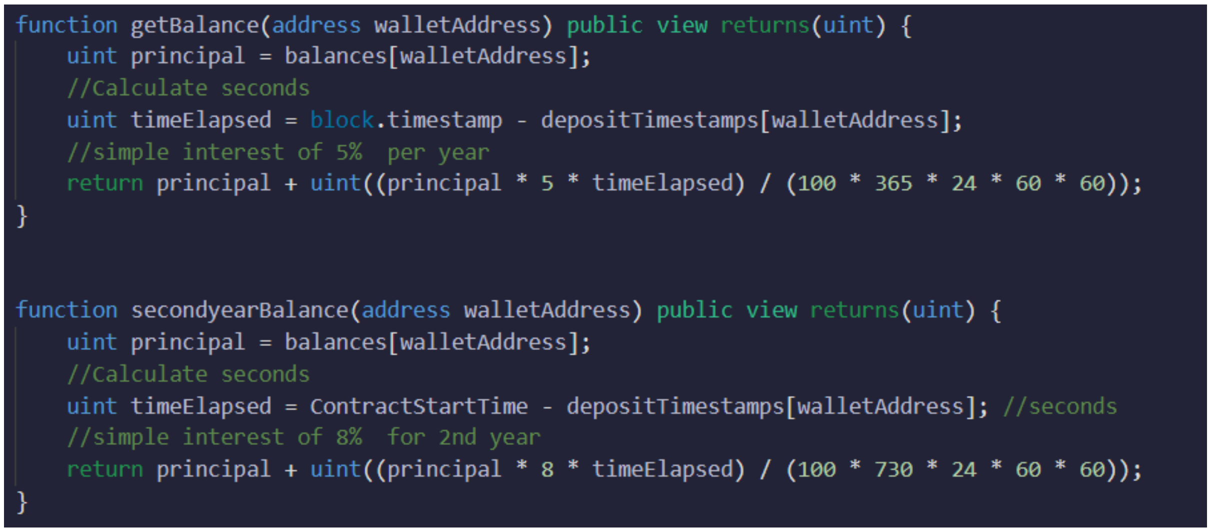
We have written add money function in the smart contract where payable keyword allows user to receive money. The function adds balances (it shows the current balance) and total balance (it shows total balance of the contract, plus added money). Moreover, msg. value is the amount of ether sent with a message to a contract. (Crescenzi, 2018)



```
// This function allows user to deposit amount into this smart contract
function addmoney() public payable {
    balances[msg.sender] = msg.value;
    totalBalance = totalBalance + msg.value;
```

Fig 5: Add money Function in smart contract.

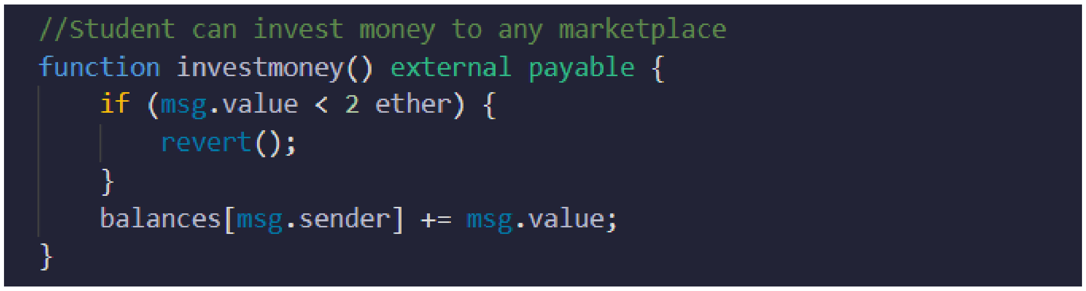
The function getbalance is issued for generating interests of one and two years for total deposited money in the smart contract. The keyword principal means the balances at present in the wallet address of user. However, the view function reads state variables that cannot be changed after defining them in the contract. The return statement is required because we want to return the value from the function of getbalance. The time elapsed is calculated by deducting deposit time stamps from block.timestamp. A block timestamp is a time of block generation and the time is specified in milliseconds that have passed since the beginning of the Unix epoch. (Block timestamp, n.d.) After then, the money with interests will be counted by multiplying principal value with interest rate and time elapsed) and also it is divided by total money (100 ether) multiplying with total year (365 days or730 days), total hours (for one day), total minutes (depending on the day), total seconds (depending on the minutes).



```
function getBalance(address walletAddress) public view returns(uint) {
    uint principal = balances[walletAddress];
    //Calculate seconds
    uint timeElapsed = block.timestamp - depositTimestamps[walletAddress];
    //simple interest of 5%  per year
    return principal + uint((principal * 5 * timeElapsed) / (100 * 365 * 24 * 60 * 60));
}


function secondyearBalance(address walletAddress) public view returns(uint) {
    uint principal = balances[walletAddress];
    //Calculate seconds
    uint timeElapsed = ContractStartTime - depositTimestamps[walletAddress]; //seconds
    //simple interest of 8%  for 2nd year
    return principal + uint((principal * 8 * timeElapsed) / (100 * 730 * 24 * 60 * 60));
}
```

Fig 6: Adding getBalance functions to produce interest rates for different years.

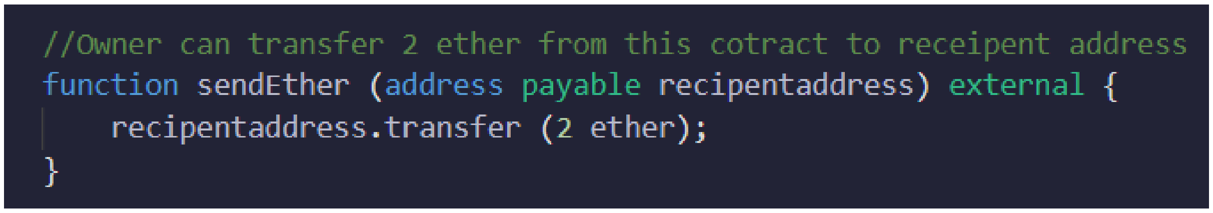
Student can invest their savings in various financial products such as, stocks, bonds etc. But they can invest money less than 2 ethers from their smart contract. Otherwise, the sending money will go back to his account automatically.



```
//Student can invest money to any marketplace
function investmoney() external payable {
    if (msg.value < 2 ether) {
        revert();
    }
    balances[msg.sender] += msg.value;
}
```

Fig 7: For investing money, created investmoney() function.

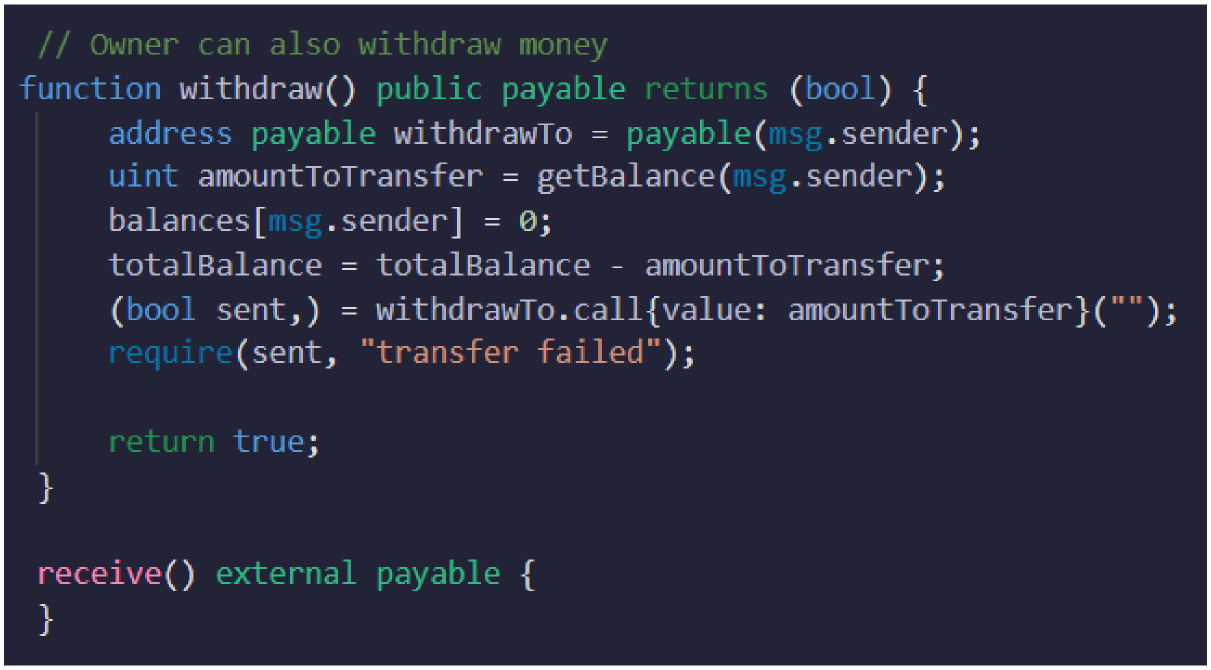
Owner can transfer money to the wallet address of their family members, friends or others from the bank through smart contract. But they can send only 2 ethers.



```
//Owner can transfer 2 ether from this cotract to receipent address
function sendEther (address payable recipentaddress) external {
    recipentaddress.transfer (2 ether);
}
```

Fig 8: sendEther function allows to transfer ethers from user address to recipient address.

Owner can withdraw money from his account with interests. Total balance will be shown by deducting amount of withdrawal money (amountToTransfer) from total balance. After the bank has transferred the money to the user the datatype sent keyword must be shown, otherwise it will show ‘transfer failed’ message. After successful withdrawal signal in the contract, owner can receive money.



```
 // Owner can also withdraw money
function withdraw() public payable returns (bool) {
     address payable withdrawTo = payable(msg.sender);
     uint amountToTransfer = getBalance(msg.sender);
     balances[msg.sender] = 0;
     totalBalance = totalBalance - amountToTransfer;
     (bool sent,) = withdrawTo.call{value: amountToTransfer}("");
     require(sent, "transfer failed");

     return true;
 }

 receive() external payable {
 }
```

Fig 9: function withdraw() used for withdrawal purpose.