```solidity
// SPDX-License-Identifier: Bhide License
pragma solidity ^0.8.0;

contract Bank {
  // mapping(type => type)
  mapping(address => uint256) private balances;

  // payable is necessary because the function accepts a value (amount) as a parameter (EXTERNAL SOURCE AHE MHANUN)
  function deposit(uint256 amount) public payable {
    balances[msg.sender] += amount;
  }

  function withdraw(uint256 amount) public {
    require(balances[msg.sender] >= amount, "Insufficient balance");
    balances[msg.sender] -= amount;
  }

  // view does not modify values within the contract (return kartana lihaycha)
  function getBalance() public view returns (uint256) {
    return balances[msg.sender];
  }
}
```

/*

This Solidity code defines a simple smart contract for a Bank where users can deposit and withdraw funds, as well as check their balance. I'll break down each section, covering theory and specific Solidity concepts.

SPDX License Identifier

solidity

Copy code

// SPDX-License-Identifier: Bhide License

The SPDX license identifier is a comment placed at the top of the code to specify the software license. This code is labeled with a custom "Bhide License," indicating a specific license created by the author, potentially for personal or educational purposes. It's required by Solidity compilers to comply with licensing requirements in open-source development.

Pragma Directive

solidity

Copy code

```
pragma solidity ^0.8.0;
```

The pragma solidity ^0.8.0; directive specifies that this code should be compiled with Solidity version 0.8.0 or higher (but not breaking 0.9.0). It ensures compatibility and access to the latest language features and security enhancements present in the specified version.

Contract Declaration

solidity

Copy code

```
contract Bank {
    ...
}
```

A contract in Solidity is a collection of code and data that resides at a specific address on the Ethereum blockchain. Here, the contract is named Bank. Once deployed, this contract can interact with users on the blockchain network, enabling them to use its functionality.

State Variables

solidity

Copy code

```
mapping(address => uint256) private balances;
```

Mappings: In Solidity, mappings are a way to create key-value pairs. Here, balances is a mapping from an address type (representing Ethereum addresses) to a uint256 type (representing unsigned integers, often used for currency amounts).

address: Ethereum addresses are unique 20-byte identifiers for accounts and contracts.

uint256: Unsigned integers (256-bit) can store large values, suitable for tracking balances in a financial system.

Privacy Modifier (private): The private keyword restricts access to the variable, meaning only functions within the Bank contract can access balances. This ensures that no other contracts or external entities can directly alter or view the balances.

Deposit Function

solidity

Copy code

```
function deposit(uint256 amount) public payable {
    balances[msg.sender] += amount;
}
```

Parameters: The function deposit takes an input parameter amount, which is of type uint256 and represents the amount to be deposited into the user's balance.

Access Modifier (public): The public modifier makes this function accessible by anyone on the blockchain network, allowing users to call this function.

payable Modifier: The payable modifier allows the function to receive Ether (native cryptocurrency of Ethereum). This is crucial for a deposit function, as it signifies that the function can accept Ether from external sources, ensuring the function can handle transfers of Ether.

msg.sender: This is a global variable in Solidity that contains the address of the entity calling the function. msg.sender allows the contract to recognize who initiated the transaction, essential for determining which address should be credited with the deposited amount.

State Change (+=): The line balances[msg.sender] += amount; increases the caller's balance by the amount specified in the transaction, updating the contract's balances mapping.

Withdraw Function

solidity

Copy code

```
function withdraw(uint256 amount) public {
    require(balances[msg.sender] >= amount, "Insufficient balance");
    balances[msg.sender] -= amount;
}
```

Parameters: Similar to deposit, the withdraw function takes an amount parameter, which represents the amount the user wishes to withdraw.

require Statement: require is a Solidity function used to enforce conditions. Here, it checks if balances[msg.sender] is greater than or equal to amount. If this condition fails, the transaction reverts with an error message, "Insufficient balance." This prevents the user from withdrawing more than their current balance.

State Change (-=): If the require check passes, balances[msg.sender] -= amount; reduces the user's balance by the specified amount, updating the mapping accordingly.

getBalance Function

solidity

Copy code

```
function getBalance() public view returns (uint256) {
    return balances[msg.sender];
}
```

Function Purpose: The getBalance function returns the current balance of the calling user.

Access Modifier (public): The function is public, allowing anyone to call it.

view Modifier: The view modifier indicates that this function doesn't modify the state of the contract. Functions marked as view simply read data and do not alter it, so they don't incur gas costs unless called within a transaction.

Return Statement: return balances[msg.sender]; provides the caller's balance stored in the balances mapping.

Summary of Key Solidity Concepts

Mappings: Mappings are similar to hash tables and are primarily used to store data with key-value pairs in Solidity. They allow easy retrieval of values associated with specific keys but cannot be iterated over directly.

Access and Privacy Modifiers:

public: Functions marked public are accessible by anyone on the blockchain network.

private: Variables marked private are restricted to the containing contract.

view: A function marked view indicates that it only reads data, not modifying it.

msg.sender: A global variable representing the address of the caller, allowing contract functions to access user-specific data or execute user-specific actions.

require Statement: Used to enforce conditions, helping maintain correct logic and prevent errors (like insufficient balance) before a transaction proceeds.

Payable Functions: Functions marked as payable can receive Ether and are essential in contracts that deal with financial transactions.

This smart contract exemplifies basic functionality for managing balances in a decentralized system. It's a useful model for building more advanced contracts that handle funds in a secure, transparent manner.

*/