Solidity Notes

Types of Variables

Some of the most commonly used types are: `boolean`, `string`, `int`, `uint`, `address`, `byte`.

Function

Functions are reusable pieces of code designed to perform specific tasks. There are different types of functions:

1. **Public Functions**
   Functions that can be called both internally (from within the contract) and externally (from other contracts or users).

2. **View Functions**
   Functions that can read the state of the contract but cannot modify it. They do not cost gas when called externally unless part of a transaction.

   Use case: To retrieve data from the contract without making any changes to it.

```solidity
function getValue() public view returns (uint) {
    return value;  // Only reads the state, does not modify it
}
```

3. **Pure Functions**
   Functions that neither read nor modify the state of the contract. They only work with their local arguments and perform calculations.
   Key Point: Like view functions, they do not cost gas unless called within a transaction, as they are purely computational.
   Use Case: For computations or logic that does not depend on contract state or blockchain data.

```solidity
function add(uint a, uint b) public pure returns (uint) {
    return a + b;  // Performs computation without accessing state
}
```

Payable Function
The `payable` modifier allows a function to receive Ether (the native cryptocurrency of the Ethereum blockchain).

Data Storage Locations
There are three primary data storage locations in Solidity: `storage`, `memory`, and `calldata`. Each has its own purpose and is used for different types of data handling.

1. **Storage**

Data stored in `storage` is persistent and saved on the blockchain, meaning it remains available even after function or transaction execution is complete.
- **Persistent**: Data remains in the contract's state.
- **Expensive**: Writing/modifying data in storage is costly in terms of gas.
- **Default for State Variables**: State variables declared in a contract are stored in `storage` by default.
Use Case: Permanent data like contract state variables.

```solidity
  uint public counter;  // State variable stored in `storage`

  function incrementCounter() public {
      counter += 1;  // Modifying data stored in `storage`
  }
```

2. **Memory**
`memory` is a temporary, modifiable storage location used during function execution. Data in `memory` exists only for the duration of the function call and is erased afterward.
- **Temporary**: Data is discarded after function execution completes.
- **Modifiable**: You can modify data stored in `memory`.
- **Gas Cost**: More expensive than `calldata` but cheaper than `storage`.
Use Case: Temporary data that needs to be modified within a function.

```solidity
  function modifyArray(uint[] memory arr) public returns (uint[] memory) {
      arr[0] = 42;  // Modifying the array
      return arr;
  }
```
Note: `memory` can only be specified for `array`, `struct`, or `mapping`.

3. **Calldata**

`calldata` is a temporary, immutable (read-only) storage location for function arguments passed externally.
- **Read-Only**: Data cannot be modified.
- **Gas Efficient**: Cheaper than `memory` because data is not copied.
- **Used in External Calls**: Primarily for external function parameters.

Use Case: Passing data from external sources without modifying it.

```solidity
  function readArray(uint[] calldata arr) public pure returns (uint) {
      return arr[0];  // Can only read, cannot modify
  }
```

EVM Storage Location:

The EVM storage locations table indicates the different areas where the Ethereum Virtual Machine (EVM) can read and write data during smart contract execution. The table categorizes these locations based on their functionality:

Write & Read: Locations like the stack and memory that are used for both reading and writing data.

Write: Locations such as logs where data can only be written, often used for event recording.

Read: Locations like transaction data and chain data that are available only for reading, providing information about the blockchain and the current transaction state.

Write & Read	**Write**	**Read**
Stack	Logs	Transaction Data
Memory		Chain Data
Storage		Gas Data
Transient Storage		Program Counter
Call Data		Code
		Return Data

Mapping

In Solidity, a `mapping` is a special data structure that functions like a hash table or dictionary in other programming languages. It allows you to store key-value pairs where each key maps to a specific value.

Syntax:

```solidity
mapping(KeyType => ValueType) public myMapping;
```

Payable Indicator
A red button in Solidity indicates that the function is `payable`, meaning you can send any native cryptocurrency through this function.

Reverting Transactions
When a transaction reverts, it undoes everything that it has previously done.

Transaction Fields

- **Nonce**: Transaction count for the account
- **Gas Price**: Price per unit of gas
- **Gas Limit**: Max gas that this transaction can use (default is 21000)
- **To**: Address that this transaction is sent to
- **Value**: Amount of wei to send
- **Data**: What to send to the `To` address
- **v, r, s**: Components of the transaction signature

Smart Contract Connectivity Problem / Oracle Problem

The smart contract connectivity problem refers to challenges and limitations associated with smart contracts interacting with external systems or data sources. By design, smart contracts operate on a blockchain and are deterministic, meaning they can only interact with data and events on the blockchain itself. This creates issues when smart contracts need to interact with external systems, real-world data, or external APIs.

Interface vs. ABI

- **Interface**: Defines function signatures in Solidity that allow one contract to interact with another. It’s a blueprint for how to call functions.
- **ABI (Application Binary Interface)**: Compiled representation that allows external applications (like web apps or wallets) to interact with a smart contract, specifying how to encode/decode function calls and data.

In simple terms:
- **Interface** is used within smart contracts (in Solidity).
- **ABI** is used for external applications interacting with smart contracts.

Gas Efficiency

Make contracts more gas-efficient using these keywords:

- **Constant**
  - Assigned at declaration.
  - Value is known at compile time.
  - Ideal for fixed values that do not change across deployments.
  - More gas-efficient due to compile-time embedding.

- **Immutable**
  - Assigned during constructor execution.
  - Value can be dynamic and based on deployment parameters.
  - Suitable for values that are set once upon deployment.
  - Offers a balance between flexibility and gas efficiency.

Example Comparing Both
```solidity
contract Token {
    string public constant NAME = "MyToken";  // Known at compile time
    address public immutable OWNER;           // Set during deployment

    constructor(address _owner) {
        OWNER = _owner;
    }
}
```

Receive Vs FallBack

Receive Function

- **Purpose:** Handles plain Ether transfers sent to the contract **without any data**.

Declaration:

```
        receive() external payable {
            // Logic for handling Ether reception
        }
```

- **Characteristics:**
  - Must be declared as external and payable.
  - Cannot have arguments or return values.

- **Triggered When:**
    - The contract receives Ether with empty calldata (no data).
    - Examples: Direct Ether transfers using .send(), .transfer(), or .call{value: amount}("") without data.

Fallback Function

- **Purpose:** Handles:
    - Calls to non-existent functions.
    - Ether transfers that include data.
    - Plain Ether transfers if receive does not exist.

```
fallback() external [payable] {
    // Logic for handling unmatched calls or Ether with data
}
```

- **Characteristics:**
    - Must be declared as external.
    - Can be marked as payable to accept Ether.
    - Cannot have arguments or return values.
- **Triggered When:**
    - The calldata does not match any existing function.
    - Ether is sent with data.
    - Ether is sent without data, but receive does not exist.
