Decentralized Systems

Solidity (cnt’d)

Solidity: contract scheleton

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.2 <0.9.0;

contract DummyContract {

        state variables

        events

        modifiers

        constructor

        functions
}
```

Solidity: modifiers

- Modifiers can **change the behavior of functions**
- Called before function execution, for example to restrict access, validate inputs, guard against reentrancy hack

  1. Declaration with the keyword **modifier**

  2. Usage with the **name of the modifier** in the declaration of the function; the modifier's code is executed before the code in the function itself

- **Can be reused** in the smart contract

Solidity: modifiers

```
// Modifier to restrict the access only to the owner
modifier onlyOwner() {
  // Check if the caller is the owner
   require(msg.sender == owner, "Not authorized");
   _;      // Continue execution of the function
}
```

See examples at https://solidity-by-example.org/function-modifier/


Start
DEMO

Solidity: events

- Used to **log** and **broadcast information** from a smart contract to the outside world, they notify external applications of important actions within the contract

  1. Declaration with the keyword **event**

  2. Usage with the keyword **emit** inside the body of a function

- **Each emitted event**
  - generates a **log entry** on the Ethereum blockchain
  - has a **cost** in **gas**

Solidity: events

- Smart contract WishOfDay: we can add one event emitted each time a new wish is written in the blockchain

Solidity: events

- Smart contract WishOfDay: we can add one event emitted each time a new wish is written in the blockchain

1. **Declaration**

```
event WishAdded(uint256 _data, string _message, string indexed _author);
```

The keyword indexed is used to make authors filterable when querying the event logs

Solidity: events

- Smart contract WishOfDay: we can add one event emitted each time a new wish is written in the blockchain

1. **Declaration**

```
event WishAdded(uint256 _data, string _message, string indexed _author);
```

The keyword indexed is used to make authors filterable when querying the event logs

2. Usage

```
function setOneWish(string memory _message, string memory _author)
public {
 ...
 ...
  emit WishAdded(block.timestamp, _message, _author);
}
```

Solidity: events

- Events contribute to the **transparency** and **accountability** of smart contracts since they provide a **historical record of actions** and state changes within a contract, making it easier to verify the correctness of a contract's behavior

- Powerful tool that helps make smart contracts more interactive and responsive, bridging the gap between on-chain and off-chain systems

Solidity: transaction logs

- **Transaction logs**
  - are **stored in a special area** of the blockchain called the transaction log, which is **accessible externally** but **not within smart contracts** themselves
  - are **cheaper to store** compared to storing data in the contract storage
  - consist of **log entries**, each of which can have multiple **topics** and **data fields**

Solidity: transaction logs

- A **log entry** has the following fields
  - **From:** the address of the smart contract that generated the log entry
  - **Topic:** event signature (hash of the event's name and its parameter types)
  - **Event:** the event name
  - **Args:** the parameters associated with the event. At most three parameters can be **indexed**, and they can be used for filtering *(indexed parameters are hashed and placed in separate topics to make searching for these values efficient)*


Start
DEMO

Solidity: payable

- Functions and addresses declared **payable** can **receive and handle Ether**
- Ether sent to a payable function is read in **msg.value**
- The balance of a contract is read in **address(this).balance**
- **tranfer()** is the preferred way to send Ether, automatically reverts transactions in case of errors
- **send()** is less common, returns a boolean value to indicate success or failure, and it requires manual checking of the return value to handle potential errors

Solidity: payable

- A contract receiving Ether can also use the functions
  - **receive()** external payable
  - **fallback()** external payable

- receive() is automatically called (if present) whenever a contract receives a message with empty calldata, e.g., when Ether is sent to the contract without a specific function call
- fallback() is automatically called (if present) whenever a contract receives a message that is not handled by any of the contract's other functions; it does not take any arguments and it is no longer considered a best practice to handle Ether with fallback()

Solidity: errors

- Compile time errors (easier to correct)
- Run time errors
  - Out of gas
  - ~~Overflow and Underflow errors~~
  - Revert errors

Solidity: errors

- **revert()**, **require()**, and **assert()** are three different functions used to handle different types of conditions and errors in Solidity

Solidity: errors

- **revert()**, **require()**, and **assert()** are three different functions used to handle different types of conditions and errors in Solidity
- **revert(condition,message)**
  - if condition is **false**, cancel and **revert** the current transaction, **without returning any remaining gas** to the sender; send a message to explain the reason of the revert
  - if condition is **true**, **continue**
- On Etherscan it is possible to find out why a transaction was reverted

Solidity: errors

- **revert()**, **require()**, and **assert()** are three different functions used to handle different types of conditions and errors in Solidity
- **require(condition,message)**
  - used for validating user inputs and contract state to ensure that certain conditions are met
  - if condition is **false**, the transaction is **reverted** and the **remaining gas is sent back to the sender**; send a message to explain the reason of the revert
  - if condition is **true**, **continue**

Solidity: errors

- **revert()**, **require()**, and **assert()** are three different functions used to handle different types of conditions and errors in Solidity
- **assert(condition)**
  - used to check for **conditions that should never evaluate to false**;
    if the condition is false, the transaction will be reverted, and all remaining gas consumed; no message is sent back to the sender
  - used for debugging and ensuring the contract's internal consistency, catches **situations that should never occur** in a correctly functioning contract

Solidity: assert() example

```
function sum (uint256 a, uint256 b) public pure returns (unit256) {
    uint256 result = a + b;
    assert (result >= a);   // if false, possible overflow :(
    return result;
}
```

- If assert() sees something “wrong”, halts the execution and burns all remaining gas

Solidity: SafeMath library

- Before Solidity 0.8, this library was used for **preventing integer overflow and underflow** in arithmetic operations
  https://docs.openzeppelin.com/contracts/2.x/api/math#SafeMath
- Unexpected behavior can lead to security vulnerabilities, especially when dealing with tokens or financial transactions

Solidity 0.8 added **built-in overflow and underflow checks** to the language level.
This means that the compiler will automatically check for these errors and revert the transaction if they are found.

Solidity: libraries

- Solidity libraries can be included in smart contract using the **import** statement (with local and external files)

When choosing a (Solidity) library consider its

- **security**: the library should be audited and have a good reputation
- **gas efficiency**: the library should be gas-efficient to minimize the cost of deploying and using the library
- **community**: the library should have a large and active community that can provide support

Abstract view of the storage