Smart Contract

Storage Example

```
pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns
(uint) {
        return storedData;
    }
}
```

Simplest form of a cryptocurrency

```
pragma solidity ^0.8.26;

// This will only compile via IR
contract Coin {
```

```
    // The keyword "public" makes variables
    // accessible from other contracts
    address public minter;
    mapping(address => uint) public
balances;

    // Events allow clients to react to
specific
    // contract changes you declare
    event Sent(address from, address to,
uint amount);

    // Constructor code is only run when the
contract
    // is created
    constructor() {
        minter = msg.sender;
    }

    // Sends an amount of newly created
coins to an address
    // Can only be called by the contract
creator
    function mint(address receiver, uint
amount) public {
        require(msg.sender == minter);
        balances[receiver] += amount;
    }

    // Errors allow you to provide
information about
    // why an operation failed. They are
returned
    // to the caller of the function.
```

```
    error InsufficientBalance(uint
requested, uint available);

    // Sends an amount of existing coins
    // from any caller to an address
    function send(address receiver, uint
amount) public {
        require(amount <=
balances[msg.sender],
InsufficientBalance(amount,
balances[msg.sender]));
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver,
amount);
    }
}
```

Simple Open Auction

```
pragma solidity ^0.8.4;
contract SimpleAuction {
    // Parameters of the auction. Times are either
    // absolute unix timestamps (seconds since 1970-01-01)
    // or time periods in seconds.
    address payable public beneficiary;
    uint public auctionEndTime;

    // Current state of the auction.
    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    // Set to true at the end, disallows any change.
    // By default initialized to `false`.
```

```
    bool ended;

    // Events that will be emitted on changes.
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // Errors that describe failures.

    // The triple-slash comments are so-called natspec
    // comments. They will be shown when the user
    // is asked to confirm a transaction or
    // when an error is displayed.

    /// The auction has already ended.
    error AuctionAlreadyEnded();
    /// There is already a higher or equal bid.
    error BidNotHighEnough(uint highestBid);
    /// The auction has not ended yet.
    error AuctionNotYetEnded();
    /// The function auctionEnd has already been called.
    error AuctionEndAlreadyCalled();

    /// Create a simple auction with `biddingTime`
    /// seconds bidding time on behalf of the
    /// beneficiary address `beneficiaryAddress`.
    constructor(
        uint biddingTime,
        address payable beneficiaryAddress
    ) {
        beneficiary = beneficiaryAddress;
        auctionEndTime = block.timestamp + biddingTime;
    }

    /// Bid on the auction with the value sent
    /// together with this transaction.
    /// The value will only be refunded if the
    /// auction is not won.
    function bid() external payable {
        // No arguments are necessary, all
        // information is already part of
        // the transaction. The keyword payable
        // is required for the function to
        // be able to receive Ether.

        // Revert the call if the bidding
        // period is over.
        if (block.timestamp > auctionEndTime)
            revert AuctionAlreadyEnded();

        // If the bid is not higher, send the
        // Ether back (the revert statement
        // will revert all changes in this
```

```
        // function execution including
        // it having received the Ether).
        if (msg.value <= highestBid)
            revert BidNotHighEnough(highestBid);

        if (highestBid != 0) {
            // Sending back the Ether by simply using
            // highestBidder.send(highestBid) is a security risk
            // because it could execute an untrusted contract.
            // It is always safer to let the recipients
            // withdraw their Ether themselves.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
        emit HighestBidIncreased(msg.sender, msg.value);
    }

    /// Withdraw a bid that was overbid.
    function withdraw() external returns (bool) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because the
recipient
            // can call this function again as part of the
receiving call
            // before `send` returns.
            pendingReturns[msg.sender] = 0;

            // msg.sender is not of type `address payable` and
must be
            // explicitly converted using `payable(msg.sender)` in
order
            // use the member function `send()`.
            if (!payable(msg.sender).send(amount)) {
                // No need to call throw here, just reset the
amount owing
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    /// End the auction and send the highest bid
    /// to the beneficiary.
    function auctionEnd() external {
        // It is a good guideline to structure functions that
interact
        // with other contracts (i.e. they call functions or send
Ether)
        // into three phases:
```

```
        // 1. checking conditions
        // 2. performing actions (potentially changing conditions)
        // 3. interacting with other contracts
        // If these phases are mixed up, the other contract could
call
        // back into the current contract and modify the state or
cause
        // effects (ether payout) to be performed multiple times.
        // If functions called internally include interaction with
external
        // contracts, they also have to be considered interaction
with
        // external contracts.

        // 1. Conditions
        if (block.timestamp < auctionEndTime)
            revert AuctionNotYetEnded();
        if (ended)
            revert AuctionEndAlreadyCalled();

        // 2. Effects
        ended = true;
        emit AuctionEnded(highestBidder, highestBid);

        // 3. Interaction
        beneficiary.transfer(highestBid);
    }
}
```