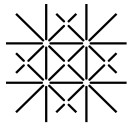


University of Basel

Center for Innovative Finance

Smart Contracts and Decentralized Finance

Global Variables, Transfers, and Events

Prof. Dr. Fabian Schär
University of Basel

Release Ver.: (Local Release)
Version Hash: (None)
Version Date: (None)

Time on the Blockchain

Goal: Contract users must be able to set an end time for their auction.

How does a contract keep track of time?

Solidity offers `global variables`.

Example: `block.timestamp` gives us the time that the miner sets in the block where our transaction is included.

Security consideration: This information can be influenced by the miner.

Other Block Variables

- `block.basefee` returns current block's base fee.
- `block.chainid` returns current chain id.
- `block.coinbase` returns the miner address of the current block.
- `block.difficulty` returns current block difficulty.
- `block.gaslimit` returns current block gaslimit.
- `block.number` returns the current block number.

↗ Solidity Documentation: Block and Transaction Properties.

Other Global Variables

Frequently used

- `msg.sender` sender address of the message (current call). *This is not necessarily the origin of the transaction, it can be the address of another contract that made an internal call.*
- `msg.value` ether amount in wei sent with the message.

Others

- `msg.data` complete calldata.
- `msg.sig` first four bytes of the calldata indicating the function that is called.
- `tx.origin` address of the **original** sender of the transaction.
- `tx.gasprice` gas price of the transaction.
- `blockhash(blockNumber)` hash of the indicated block.

↗ Solidity Documentation: Cheatsheet — Global Variables.

Units

Solidity supports **time** and **monetary** units natively.

Ether Units

```
1 uint amount = 1 ether;
2 assert(amount == 1e18); // Will Pass
```

Most common are `wei` (1), `gwei` (10^9), and `ether` (10^{18}).

Time suffixes can be used to convert to seconds. Supported are `seconds`, `minutes`, `hours`, `days`, `weeks`.

Time Units

```
1 uint twoHours = 2 hours;
2 assert(twoHours == 2 * 60 * 60); // Will Pass
```

Suffixes and Variables

Keep in mind: Usage of variables with suffixes is disallowed.

Hours to Seconds

```
1 uint numberOfHours = 2;
2 uint twoHours = numberOfHours hours; // Not allowed
3 uint twoHours = numberOfHours * 1 hours; // Use this
      instead
```

Add Duration to the Auction

Auction Parameters

```
1 // Auction parameters
2 address public beneficiary;
3 uint public endTime; // As UNIX timestamp
4
5 constructor (address _beneficiary, uint
      _durationMinutes) {
6   beneficiary = _beneficiary;
7   endTime = block.timestamp + _durationMinutes * 1
        minutes;
8 }
```

Function Restrictions and Requirements

Often, functions in a contract need to be **restricted** in some way.

Goal: `bid()` should only be callable during an active auction.

Solidity offers `require(<condition>, <error message>);`:

- if `<condition>` evaluates to `false`, the **whole** transaction is **reverted.**
- `<error message>` is optional but improves usability.

Restricted bid() Function

```
1 function bid() public {
2   require(block.timestamp < endTime, 'Auction Ended'
        );
3 }
```

Building the bid() function

Accept and store valid bids:

- Add `payable` modifier to be able to receive ether.
  $\Rightarrow$ *All ether received are in full control of the contract.*
- Check if bid (`msg.value`) is higher than previous valid bid.
- Store the new highest bid and bidder (`msg.sender`).

bid() Function

```
1 function bid() public payable {
2   require(block.timestamp < endTime, 'Auction Ended'
        );
3   require(msg.value > highestBid, 'Bid too small');
4   highestBid = msg.value;
5   highestBidder = msg.sender;
6 }
```

Refunding the Outbidden

Goal: Whenever a new valid bid is received, the previous bidder is refunded.

Remember: A smart contract is unable to initiate transactions by itself.

Options:

1. Refund as part of the new bid transaction.
   $\rightarrow$ *Passing execution to another address introduces significant security risk.*
2. Allow the user to withdraw his invalidated bids.
   $\rightarrow$ **Preferred option**

Mappings

Goal: Keep track of how much ether a previous bidder can withdraw.

Mappings store `key => value` pairs, similar to a hash map or dictionary in other programming languages:

- The key's keccak256 hash points to the storage slot for the value.
- Mappings span the full storage space of 2^{256} slots.
- Declaration: `mapping(keyType => valueType) <visibility> <variable name>;`

Visibility: If a mapping is declared as `public`, it automatically has a getter function in form of `<variable name>(<keyType> key) returns (<valueType>)`.

Store withdrawable funds per address

pendingReturns Mapping

```
1  contract SimpleAuction {
2    // Allowed withdrawals of previous bids
3    mapping(address => uint) public pendingReturns;
4
5    function bid() public payable {
6      require(block.timestamp < endTime, 'Auction
           ended');
7      require(msg.value > highestBid, 'Bid too low');
8      if (highestBid != 0) {
9        pendingReturns[highestBidder] += highestBid;
10     }
11     highestBid = msg.value;
12     highestBidder = msg.sender;
13   }
14 }
```

$\Rightarrow$ **Using += prevents overwriting previous unclaimed refunds.**

More on Mappings

Mappings are very powerful data structures with some drawbacks:

- Lack of length property.
- Cannot be enumerated or returned.
- Cannot be cleared easily.

$\Rightarrow$ **No straightforward way to reconstruct storage state.**

Separately storing all keys used can be a workaround.

Mappings can be **nested**.
`mapping(address => mapping(uint => bool))` will store a `uint => bool` mapping for each address.

Sending Ether from a Contract to an Address

Goal: Return ether to outbidden user as part of `withdraw()`.

For security reasons, Solidity differentiates between `payable` and non-payable addresses.

To be able to send ether to an address, it needs to be explicitly declared as `payable:`

```
address payable payableAddress =
payable(normalAddress);
```

The Three Options of a Contract to Transfer Ether

1. `address payable.transfer(<amount>)`
   Forwards 2300 gas and tries to transfer `<amount>` ether to the target address. Transfer failure reverts the whole transaction.

2. `address payable.send(<amount>)`
   Same as `transfer` but does not revert on failure. Returns `true` or `false` depending success.

3. `address payable.call {value: <amount> }("")`
   Low level call to the address. Forwards all remaining gas or an exact value, if so specified via `{value: <amount>, gas: <gas amount>.}`

For simplicity and security reasons, we are using `transfer`. But there are good reasons to use call instead. Especially to ensure forward-compatibility regarding gas costs.

Building the withdraw() function

withdraw() Function

```
1 function withdraw() external returns (uint amount) {
2   amount = pendingReturns[msg.sender];
3   if (amount > 0) {
4     pendingReturns[msg.sender] = 0;
5   payable(msg.sender).transfer(amount);
6   }
7   // optional: return amount;
8 }
```

- Add a returns statement to the function to show the amount of ether withdrawn.
- Specify the name of the return value to avoid declaring the variable and an explicit return statement.
- To prevent re-entrancy attacks, update the balance before transfer.

Ending the Auction

Goal: After endTime, the highest bid is to be transferred to the beneficiary.

Remember: Smart contracts can not do anything by themselves.

$\Rightarrow$ `auctionEnd()` to close the auction and transfer the bid is needed.

Structuring guidelines for functions interacting with other addresses:

1. Check all conditions.
2. Apply all internal state changes.
3. Interact with other addresses.

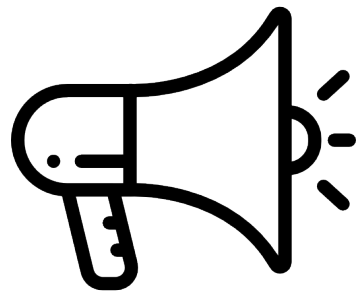
The auctionEnd() Function

auctionEnd() Function

```
1  function auctionEnd() external {
2    // 1. Check all conditions
3    require(!hasEnded, 'Auction already ended');
4    require(block.timestamp >= endTime, 'Wait for
         auction to end');
5
6    // 2. Apply all internal state changes
7    hasEnded = true;
8
9    // 3. Interact with other addresses
10   payable(beneficiary).transfer(highestBid);
11 }
```

Events

The Blockchain is fully transparent. Does this mean that one can obtain the full history of bids?



Yes, using a lot of resources: Access an archive node and call the state at every block to check the bid variable.

Should the bid of each user be stored permanently then?

No, this would be very expensive and causes state bloat.

Solution: Write data into logs.

- **Cheap:** It is stored in a different place than the state.
- **Verifiable:** It is part of the block hash.
- **Downside:** It cannot be accessed as part of a transaction.

Declaring Events

Declaration:

- `event <EventName> (<parameterType> <indexed> <parameterName>, ...)`
- Once emitted, the corresponding log can be queried from a full node.

Indexing:

- Indexed parameters can be used to filter logs.
- Slightly more expensive to emit.
- Maximum of three indexed parameters per log.

Emission: `emit <EventName> (<value>,... );`

$\Rightarrow$ Most node clients offer functionality to subscribe to events and get notified whenever a specific contract emits a specified event.

Building Events for the Auction

NewBid and AuctionEnded Events

```
1  // Events
2  event NewBid(address indexed bidder, uint amount);
3  event AuctionEnded(address winner, uint amount);
4
5  function bid() public payable { // ...
6    highestBid = msg.value;
7    highestBidder = msg.sender;
8    emit NewBid(msg.sender, msg.value);
9  }
10 function auctionEnd() external {
11   // 1. Check all conditions...
12   // 2. Apply all internal state changes
13   hasEnded = true;
14   emit AuctionEnded(highestBidder, highestBid);
15   // 3. Interact with other addresses
16   payable(beneficiary).transfer(highestBid);
17 }
```

Update of the Auction Contract

Exercise 1:

1. Add the code snippets shown in this slide deck to the SimpleAuction contract.
2. Deploy the contract, and interact with it using different addresses.