CS251

Programming in Solidity

Agenda

- Solidity basics
- Interacting with smart contracts
- Understanding gas costs
- Security considerations
- Common patterns

Useful links

- http://bit.do/cs251solidity
- https://gist.github.com/abandeali1/74d8b73f457add6b1bf7255a90b0adf5
- https://remix.ethereum.org/

Value types

- uint256
- address (bytes20)
  - balance, transfer, call, delegatecall
- bytes32
- bool

Reference types

- structs
- arrays
- bytes
- strings
- mappings

Globally available variables

- block
  - blockhash, coinbase, difficulty, gaslimit, number, timestamp
- gasLeft()
- msg
  - data, sender, sig, value
- tx
  - gasprice, origin
- abi
  - encode, encodePacked, encodeWithSelector, encodeWithSignature
- keccak256
- ecrecover
- require, assert

Function visibilities

- external
- internal
- public
- private
- pure
- view

Using imports

- Inheritance
  - `contract A is SafeMath {}`
  - `uint256 a = safeAdd(b, c);`
- Libraries
  - `using SafeMath for uint256;`
  - `uint256 a = b.safeAdd(c);`

ERC20 tokens

- https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
- A standard API for fungible tokens that provides basic functionality to transfer tokens or allow the tokens to be spent by a third party.
- An ERC20 token is itself a smart contract that contains its own ledger of balances.
- A standard interface allows other smart contracts to interact with all ERC20 tokens, rather than using special logic for each different token.

ERC20 token interface

- `function transfer(address _to, uint256 _value) external returns (bool);`
- `function transferFrom(address _from, address _to, uint256 _value) external returns (bool);`
- `function approve(address _spender, uint256 _value) external returns (bool);`
- `function totalSupply() external view returns (uint256);`
- `function balanceOf(address _owner) external view returns (uint256);`
- `function allowance(address _owner, address _spender) external view returns (uint256);`

How are ERC20 tokens transferred?

- The `transfer` function checks a few conditions, updates balances of the sender and receiver, and logs an event.
- Alice wants to transfer 100 StanfordCoin to Bob. She calls StanfordCoin.transfer(Bob.address, 100). What is happening under the hood?

ABI encoding and decoding

- Every function has a 4 byte selector that is calculated as the first 4 bytes of the hash of the function signature.
  - In the case of `transfer`, this looks like bytes4(keccak256(“transfer(address,uint256)”);
- The function arguments are then ABI encoded into a single byte array and concatenated with the function selector. ABI encoding simple types means left padding each argument to 32 bytes.
- This data is then sent to the address of the contract, which is able to decode the arguments and execute the code.
- Fallback function

Calling other contracts

- Addresses can be cast to contract types.
  - `IERC20Token tokenContract = IERC20Token(_token);`
  - `ERC20Token tokenContract = ERC20Token(_token);`
- When calling a function on an external contract, Solidity will automatically handle ABI encoding, copying to memory, and copying return values.
  - `tokenContract.transfer(_to, _value);`

Gas cost considerations

- Everything costs gas, including processes that are happening under the hood (ABI decoding, copying variables to memory, etc).
- How often to we expect a certain function to be called? Is the bottleneck the cost of deploying the contract or the cost of each individual function call?
- Are the variables being used in calldata, the stack, memory, or storage?

Stack variables

- Stack variables are generally the cheapest to use and can be used for any simple types (anything that is <= 32 bytes).
  - `uint256 a = 123;`
- All simple types are represented as bytes32 at the EVM level.
- Only 16 stack variables can exist within a single scope.

Calldata

- Calldata is a read-only byte array.
- Every byte of a transaction's calldata costs gas (68 gas per non-zero byte, 4 gas per zero byte).
  - All else equal, a function with more arguments (and larger calldata) will always cost more gas.
- It is cheaper to load variables directly from calldata, rather than copying them to memory.
  - For the most part, this can be accomplished by marking a function as `external`.

Memory

- Memory is a byte array.
- Complex types (anything > 32 bytes such as structs, arrays, and strings) must be stored in memory or in storage.
  - string memory name = “Alice”;
- Arguments must be copied to memory before calling an `internal` function or when a contract makes an external call (AKA calling a function on another contract).
- Memory is cheap, but the cost of memory grows quadratically.

Storage

- Using storage is very expensive and should be used sparingly.
- Writing to storage is most expensive.
- Reading from storage is cheaper, but still relatively expensive.
- mappings and state variables are always in storage.
- Some gas is refunded when storage is deleted or set to 0 (checkout https://gastoken.io/ for an interesting use of this).
- Variables < 32 bytes can be packed into 32 byte slots.

Event logs

- Event logs are a cheap way of storing data that does not need to be accessed by any contracts.
- Events are stored in transaction receipts, rather than in storage.
- Log arguments can be indexed for quick filtering using a block’s bloom filter.

Security considerations

- Are we checking math calculations for overflows and underflows?
- What assertions should be made about function inputs, return values, and contract state?
- Who is allowed to call each function?
- Are we making any assumptions about the functionality of external contracts that are being called?

Common patterns

- Approve and call
- Off-chain signed messages with on-chain verification
- Compressing data using 32 byte hash
- Low level calls

Questions?

- https://0xproject.com/
- https://github.com/0xProject/0x-monorepo/tree/development/packages/contracts