Solidity

Ronghui Gu

Fall 2024

Columbia University

Course website: https://verigu.github.io/6998Fall2024/

1 Examples

An Example Contract: NameCoin

```
contract nameCoin {    // Solidity code

    struct nameEntry {

        address owner; // address of domain owner
        bytes32 value;  // IP address

    }

    // array of all registered domains

    mapping (bytes32 => nameEntry) data;
```

An Example Contract: NameCoin

```
function nameNew (bytes32 name)  {
    // registration costs is 100 Wei
    if (data[name] == 0 && msg.value >= 100) {
        data[name].owner = msg.sender;   // record domain owner
        emit Register(msg.sender, name);  // log event
    }
}
```

An Example Contract: NameCoin

```
function nameUpdate (
                bytes32 name,  bytes32 newValue, address newOnwer)  {
    // check if message is from domain owner, and update if 10Wei is paid
    if (data[name].owner == msg.sender && msg.value >= 10) {
        data[name].value = newValue;      // record new value
        data[name].owner = newOwner;   // record new owner
    }
}
```

An Example Contract: NameCoin

```
  function nameLookup (bytes32 name)  {
      return data[name];
  }
} // end of contract
```

Solidity Lang

Main PL for writing smart contracts for Ethereum blockchain

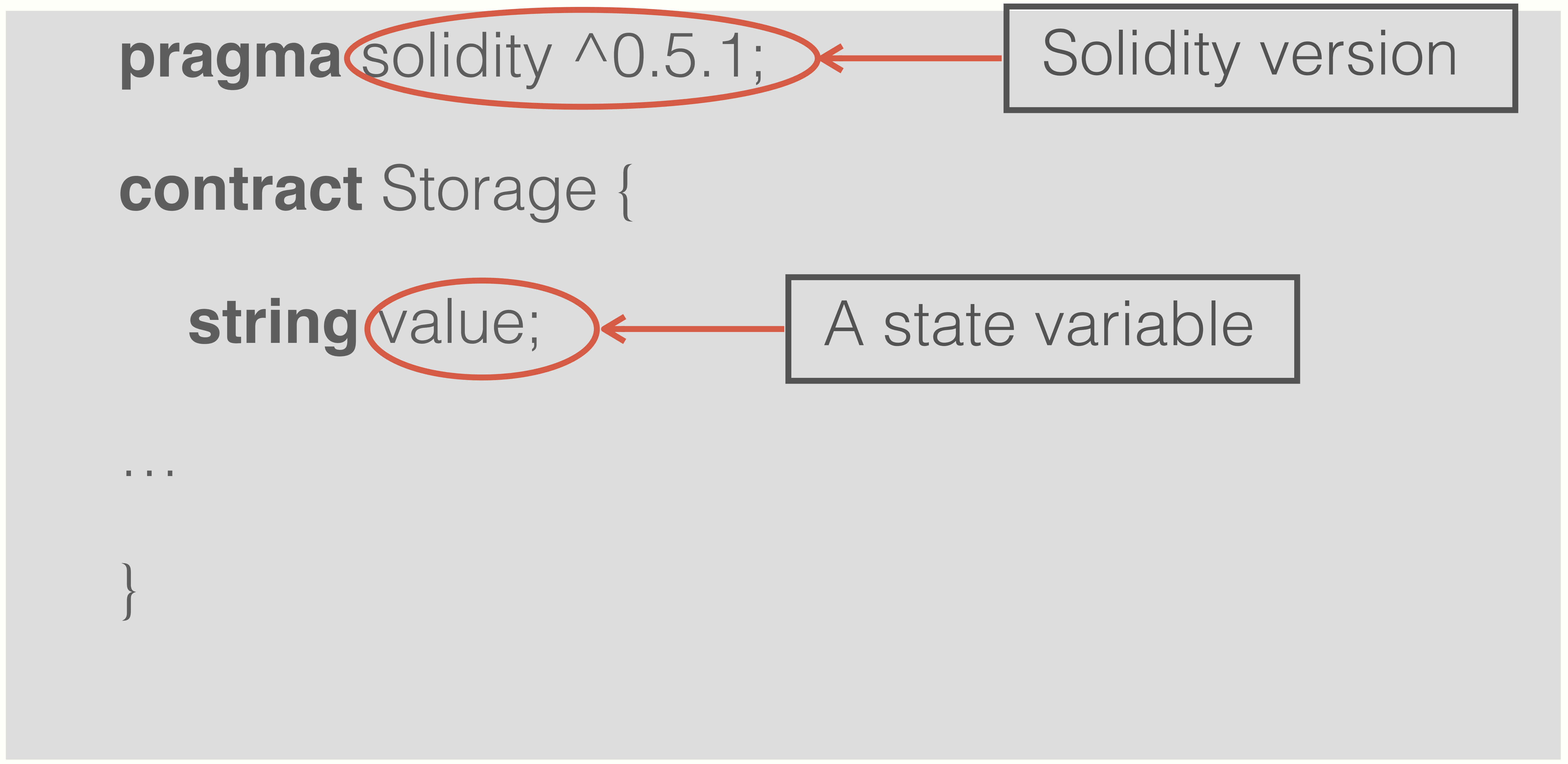
- Contract-oriented
- Statically typed
- Inheritance, libraries, and more

Smart contracts are like API microservice

- Publicly accessible to everyone
- Executed on the Ethereum blockchain

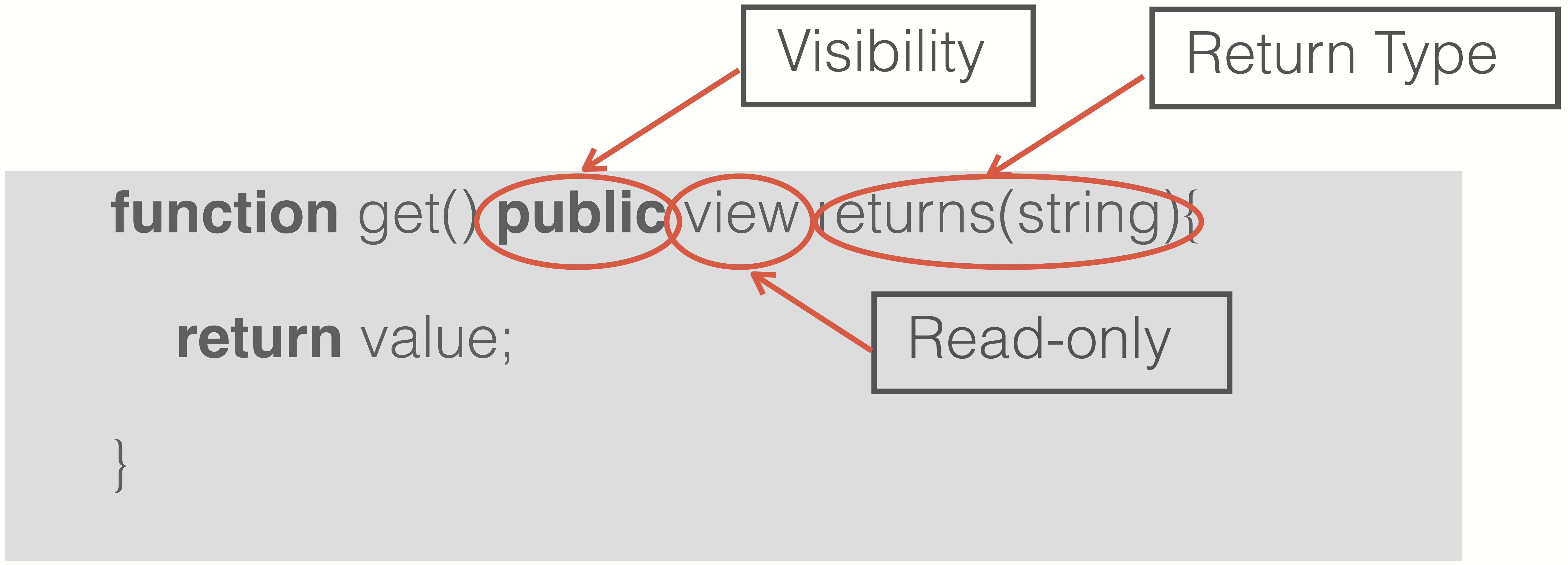
An Example Contract: Storage

A simple storage smart contract in the file Storage.sol to store/retrieve a value



```
pragma solidity ^0.5.1;

contract Storage {

    string value;

…

}
```

An Example Contract: Storage



```
function get() public view returns(string){

    return value;

}
```

An Example Contract: Storage

```
function set(string _value) public {

    value = _value;

}

constructor() public {

    value = “myValue”;

}
```

Called only once

An Example Contract: Storage

```
pragma solidity ^0.5.1;

contract Storage {

    string value;

    function get() public view returns(string memory){

        return value;

    }

    function set(string memory _value) public {

        value = _value;

    }

}
```

Memory of EVM

An Example Contract: Storage

```
pragma solidity ^0.5.1;

contract Storage {

    string public value = “myValue”;

    function set(string memory _value) public {

        value = _value;

    }

}
```

2 Solidity Lang Features

Value Types

- **Integers**: int (int256), uint (uint256), uint8
- **Boolean**: bool
- address (bytes32)
    - _address.balance, _address.send(value), _address.transfer(value)
    - _address.call(): invoke a function at another address
    - _address.staticcall(): will not modify states
    - _address.delegatecall(): load code from another contract
- Fixed-size **arrays** such as bytes32

Reference Types

- **strings**
- **structs**
- **arrays**

```
struct Person {
    uint128 age;
    string name;
}
Person [10] public students;
Person [] public people;
```

Reference Types

```
mapping (address => unit256) balances;

balances[addr] = values;
```

- **Mappings**
  - Store key-value pairs
  - Mappings are always stored in the storage (of each account)

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(), sha256(), sha3()
- **require**, **assert**
  - require(msg.value > 100, “insufficient funds”)

Function Visibilities

- **external**: function can only be called from outside contract. Arguments read from calldata.
- **public**: function can be called externally and internally. Arguments copied from calldata to memory
- **private**: only visible inside contract
- **internal**: only visible in this contract and contracts deriving from it
- **view**: only read storage (no writes to storage)
- **pure**: does not touch storage

```
function f(uint a) private pure returns(uint b)

{ return a + 1;}
```

Using imports

- **Inheritance**
  - contract **A** inherits safeMath{}
  - uint256 a = safeAdd (b, c);
  - SafeMath code is compiled into contract **A**

```
contract SafeMath {

    function safeAdd (uint256 a, uint256 b)

        internal pure returns (uint256 c)

    {

        c = a + b;

        require (c >= a, "UINT256_OVERFLOW");

    }

}
```

Using imports

- **Libraries**
    - contract **A**
      {using SafeMath for uint256}
    - uint256 a = b.safeAdd (c);

```
library SafeMath {
  function safeAdd (uint256 a, uint256 b)
    internal pure returns (uint256 c)
  {
    c = a + b;
    require (c >= a, “UINT256_OVERFLOW”);
  }
}
```

ERC20 Tokens

- 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 maintains all user balances:
  - mapping(address => uint256) internal balances;
- A standard interface allows other contracts to interact with every ERC20 token.
- No need for special logic for each 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?

```
contract ERC20Token is IERC20Token  {

    mapping (address => uint256) internal balances;

    function transfer(address  _to, uint256  _value)
            external returns (bool)

    {
        require(balances[msg.sender] >= _value,  "INSUFFICIENT");
        require(balances[_to] + _value >= balances[_to],  "OVERFLOW" );
        balances[msg.sender]  −=  _value;
        balances[_to]  +=  _value;
        emit Transfer(msg.sender, _to, _value);      //  write log message
        return true;
    }
}
```

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.
  - _address.call()

Calling Other Contracts

- Addresses can be cast to contract types.
  - address _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).
- Considerations in reducing gas costs:
    - 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;
- Only 1024 stack variable slots. Each slot can hold 32 bytes.
- All simple types are represented as bytes32 at the EVM level.

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).
- It is cheaper to load variables directly from calldata, rather than copying them to memory.
  - 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”;
- Memory is cheap, but the cost of memory grows quadratically.

Storage

- Using storage is very expensive and should be used sparingly.
  - string **storage** name = “Alice”;
- 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
- Trick for saving gas: 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.

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?

3 Reentrancy Bugs

```
contract Bank{

    mapping(address=>uint) userBalances;

    function getUserBalance(address user) constant public returns(uint) {
        return userBalances[user];      }

    function addToBalance() public payable {
        userBalances[msg.sender] = userBalances[msg.sender] + msg.value;    }

    // user withdraws funds
    function withdrawBalance() public {
        uint amountToWithdraw = userBalances[msg.sender];
        // send funds to caller ... vulnerable!
        if (msg.sender.call().value(amountToWithdraw) == false) {  throw;  }
        userBalances[msg.sender] = 0;
}  }
```

```
contract Attacker {
    uint numIterations;
    Bank bank;

    function Attacker(address _bankAddress) {     // constructor
        bank = Bank(_bankAddress);
        numIterations = 10;
        if (bank.value(75).addToBalance() == false)     {   throw;   } // Deposit 75 Wei
        if (bank.withdrawBalance() == false)     { throw; }  // Trigger attack
    }

    function () {   // the fallback function
        if (numIterations > 0) {
            numIterations --;   // make sure Tx does not run out of gas
            if (bank.withdrawBalance() == false) {  throw;  }
} } } }
```

```
// user withdraws funds
  function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];


    // send funds to caller ... vulnerable!
    if (msg.sender.call().value(amountToWithdraw) == false) {
      throw;
    }
    userBalances[msg.sender] = 0;
}
```

```
// user withdraws funds
    function withdrawBalance() public {
        uint amountToWithdraw = userBalances[msg.sender];
        userBalances[msg.sender] = 0;

        // send funds to caller ... correct!
        if (msg.sender.call().value(amountToWithdraw) == false) {
            userBalances[msg.sender] = amountToWithdraw;
            throw;
        }
}
```

4 Discussion Session

How to raise funds?