COMP4137 Blockchain Technology and Applications
COMP7200 Blockchain Technology

Lecturer: Dr. Hong-Ning Dai (Henry)

Lecture 8

Developing with Smart Contracts

Example of smart contracts

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;
contract HelloWorld {

event UpdatedMessages(string oldStr, string newStr);

string public message;

constructor(string memory initMessage) {
   message = initMessage;
   }

function update(string memory newMessage) public {
      string memory oldMsg = message;
      message = newMessage;
      emit UpdatedMessages(oldMsg, newMessage);
   }
}
```

Compile and execute your smart contract

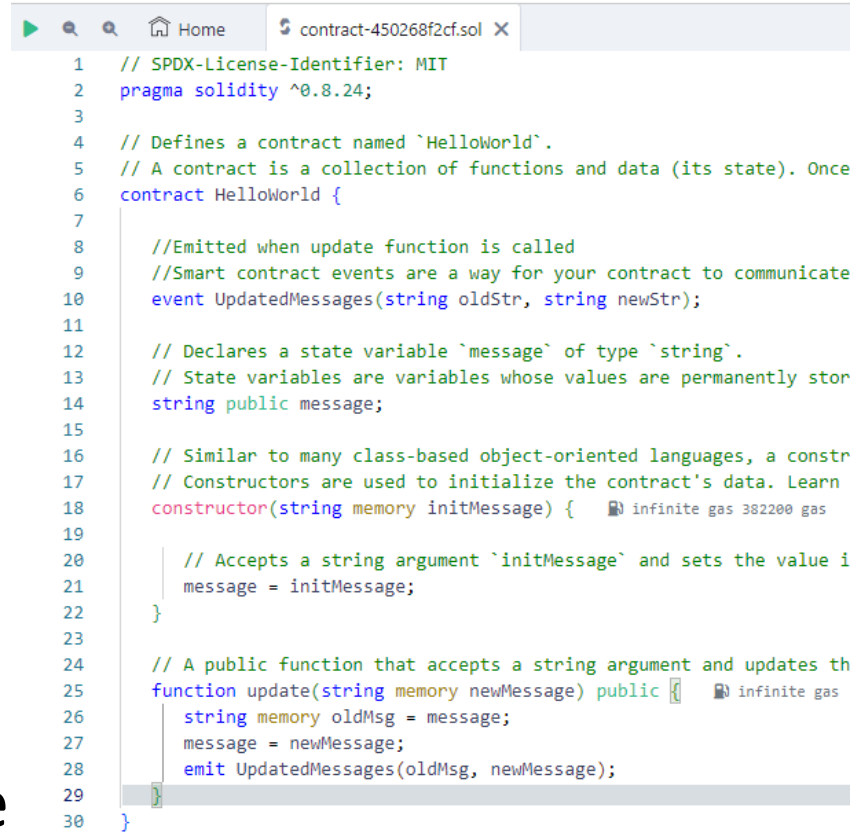
- **Requirements:**

  - Google Chrome or Firefox

- Steps:
  1. Go to https://remix.ethereum.org/
  2. Load a “Hello World” contract in the Remix IDE
  3. Compile your contract
  4. Execute your contract

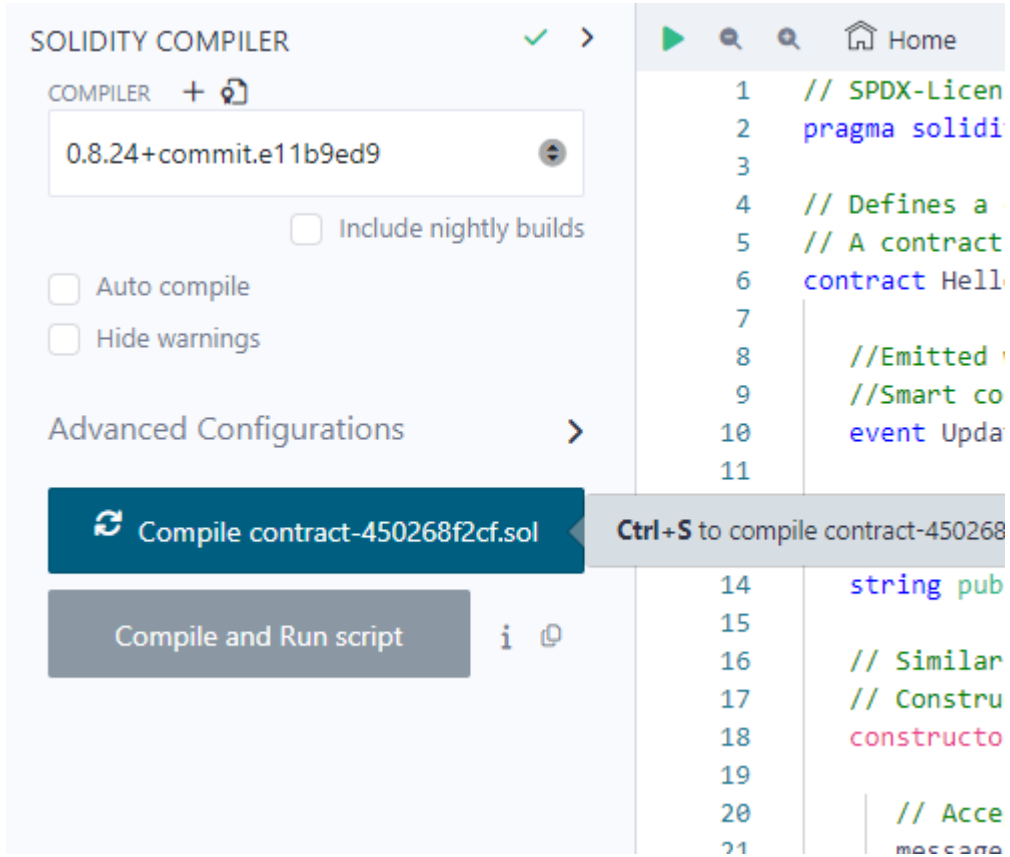
Load a “Hello World” contract in the Remix IDE

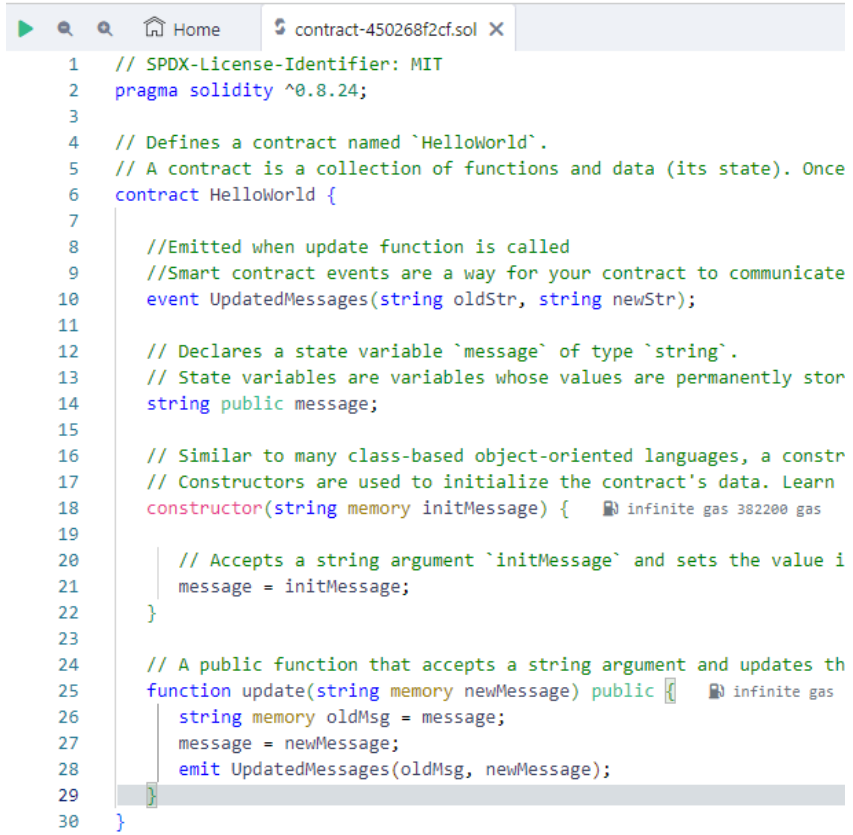
- This tutorial will make use of Remix, and in-browser Solidity editor that lets you edit and test contract code.
- Let’s load a sample file contract (`hello.sol`) into the Remix IDE.
- This is a sample contract. Read it to get a sense of what it does.



Compile “Hello World” contract in the Remix IDE

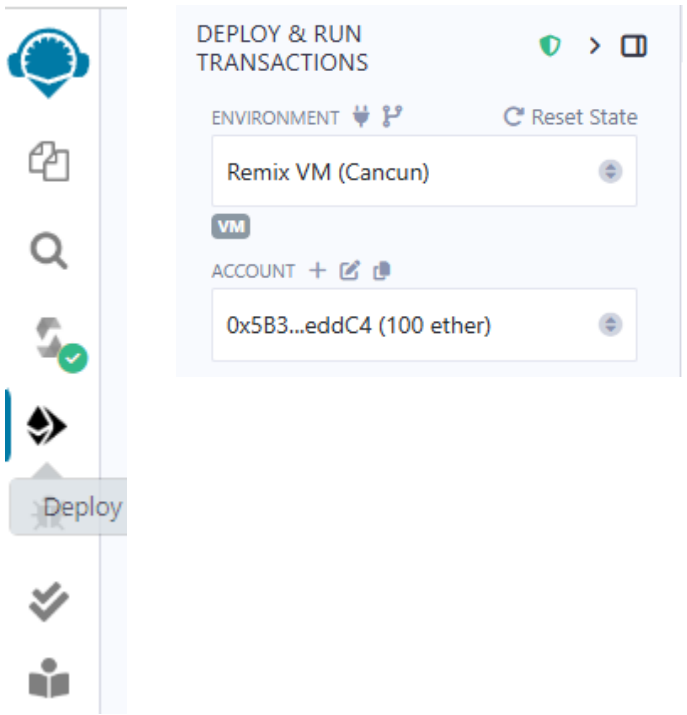
- Compile it

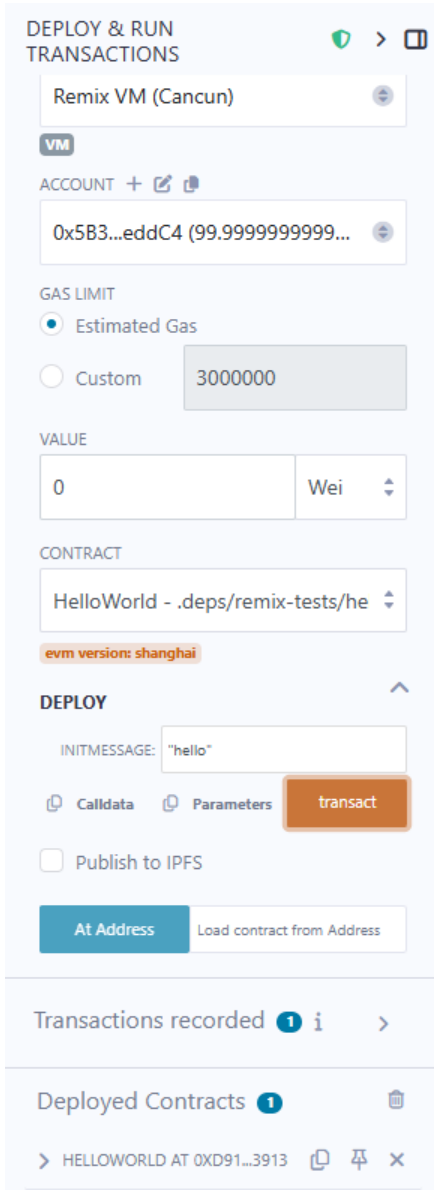


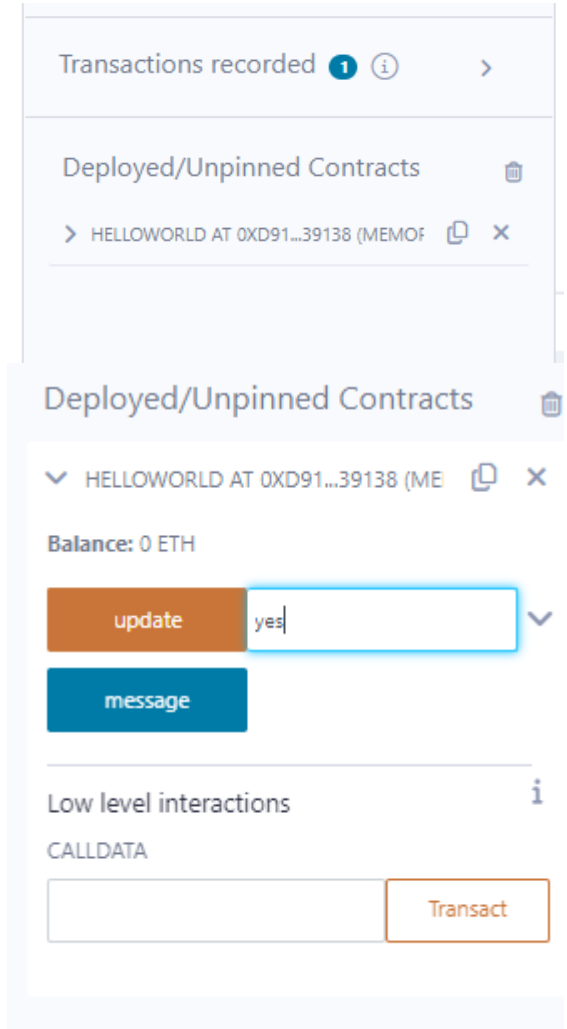


Deploy “Hello World” contract in the Remix IDE

- Deploy it







Traditional contracts vs smart contracts

	Traditional	Smart
Specification	Natural language	Code
Identity & consent	Signatures	Digital signatures
Dispute resolution	Judges	N/A (complete)
Nullification	Judges	Not possible
Payment	As specified	Built-in
Escrow	Trusted third party	Built-in

Ethereum

A decentralized platform designed to run smart contracts

- Similar to a world computer that allows users to **deploy programs** (smart contracts), **maintains the state** of all deployed smart contracts, and **executes them** upon request (transactions)
- The latest block stores the **latest local states** of all smart contracts
- Transactions result in **executing code** (calling a function) in target smart contracts
- Transaction **change the state** of one more more contracts
- Ethereum smart contracts are **Turing-complete**

Bitcoin vs. Ethereum

Bitcoin

Bob owns private keys to
a set of unspent transactions

5 BTC => Bob

2 BTC => Bob

3 BTC => Bob

Easy to make transactions and prevent double-spending attacks

Ethereum

Bob owns private keys to an account

Address: 0xfa34…

Balance: 10 ETH

Code: f() {c := a + b}

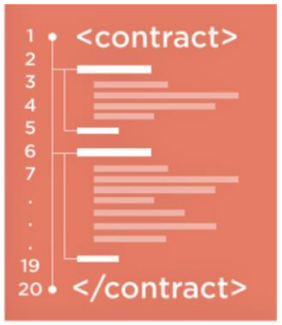
Has executable code

Update balance instead of storing unspent transactions

Ethereum accounts

User accounts

- Owned by some external entity (person, corporation, …)
- Can send transactions to transfer ether or trigger contract code
- Contains:
  - Address
  - Ether balance



Contract accounts

- “Owned” by contract (autonomous)
- Code execution triggered by transactions that call functions
- Contains:
  - Address
  - Ether balance
  - Associated contract code
  - Persistent storage (state)

Writing smart contracts

Multiple high-level languages

- Solidity [http://solidity.readthedocs.io]
- Vyper [https://github.com/ethereum/vyper]

Single low-level language

- EVM bytecode [http://yellowpaper.io], to be replaced by Ethereum WebAssembly (eWASM) in Ethereum 2.0

Solidity contract

```
Contract MyToken {
   /* … */
}
```

Vyper contract

```
def register(key, value) {
   /* … */
}
```

EVM bytecode

```
PUSH 0x60
PUSH 0x40
MSTORE CALLDATASIZE
ISZERO
…
PUSH2 ox1b4c
POP
JUMP
```

Example smart contract

Solidity version

```
pragma solidity 0.5.8;

contract SimpleBank {

    mapping(address => uint) balances;

      function deposit(uint amount) payable public {
      balances[msg.sender] += amount;
    }

      function withdraw() public {
      msg.sender.transfer(balances[msg.sender]);
      balances[msg.sender] = 0;
    }
}
```

Example smart contract

```
pragma solidity 0.5.8;

contract SimpleBank {

    mapping(address => uint) balances;

      function deposit(uint amount) payable public {
      balances[msg.sender] += amount;
    }

      function withdraw() public {
      msg.sender.transfer(balances[msg.sender]);
      balances[msg.sender] = 0;
    }
}
```

Contract name

Example smart contract

```
pragma solidity 0.5.8;

contract SimpleBank {

    mapping(address => uint) balances;

      function deposit(uint amount) payable public {
      balances[msg.sender] += amount;
    }

      function withdraw() public {
      msg.sender.transfer(balances[msg.sender]);
      balances[msg.sender] = 0;
    }
}
```

Local state maintains a mapping from user addresses to unsigned integers

Example smart contract

```
pragma solidity 0.5.8;

contract SimpleBank {

    mapping(address => uint) balances;

      function deposit(uint amount) payable public {
      balances[msg.sender] += amount;
    }

      function withdraw() public {
      msg.sender.transfer(balances[msg.sender]);
      balances[msg.sender] = 0;
    }
}
```

Function that allows users to deposit ether at the SimpleBank contract

Example smart contract

```
pragma solidity 0.5.8;

contract SimpleBank {

    mapping(address => uint) balances;

    function deposit(uint amount) payable public {
      balances[msg.sender] += amount;
    }

    function withdraw() public {
      msg.sender.transfer(balances[msg.sender]);
      balances[msg.sender] = 0;
    }
}
```

Function can receive Ether

Example smart contract

```
pragma solidity 0.5.8;

contract SimpleBank {

    mapping(address => uint) balances;

      function deposit(uint amount) payable public {
      balances[msg.sender] += amount;
    }

      function withdraw() public {
      msg.sender.transfer(balances[msg.sender]);
      balances[msg.sender] = 0;
    }
}
```

Any user can invoke the function

Example smart contract

```
pragma solidity 0.5.8;

contract SimpleBank {

    mapping(address => uint) balances;

      function deposit(uint amount) payable public {
      balances[msg.sender] += amount;
    }

      function withdraw() public {
      msg.sender.transfer(balances[msg.sender]);
      balances[msg.sender] = 0;
    }
}
```

`msg.sender` returns the address of the transaction sender

Example smart contract

```
pragma solidity 0.5.8;

contract SimpleBank {

    mapping(address => uint) balances;

      function deposit(uint amount) payable public {
      balances[msg.sender] += amount;
    }

      function withdraw() public {
      msg.sender.transfer(balances[msg.sender]);
      balances[msg.sender] = 0;
    }
}
```

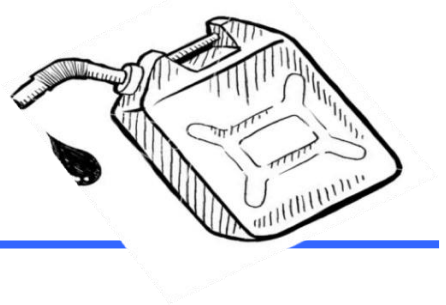
Trigger a transfer of ETH with value equal to `balances[msg.sender]` to the transaction sender

Set the balance of the transaction sender to zero

Ethereum transactions

From	<address>	Address of the target contract
To	<address>	Address of the transaction sender
Data	<method> <arg>, …, <arg>	ID of the method to invoke, along with arguments
Value	<amount>	Amount of Ether sent to the target contract
Gas/gas price		(see next slide)

Infinite loops?



What if a contract has an **infinite loop**? A pragmatic solution:

- A transaction requires **"gas"** to fuel contract execution
- Each EVM opcode requires gas to execute
- Some EVM opcodes consume **variable amount of gas** (e.g., SSTORE)
- Every transaction specifies the maximum ether the sender is willing to spent on the transaction (max gas + gas price)
- If the contract successfully executes, the **unspent ether is refunded to the sender**
- If execution runs out of gas, the execution reverts without refunding ether to the transaction sender

Example of infinite loop

```
contract Gas {
    uint256 public i = 0;

    // Using up all of the gas that you send causes your transaction to fail.
    // State changes are undone.
    // Gas spent are not refunded.
    function forever() public {
        // Here we run a loop until all of the gas are spent
        // and the transaction fails
        while (true) {
            i += 1;
        }
    }
}
```

Gas in Ethereum is a necessary evil

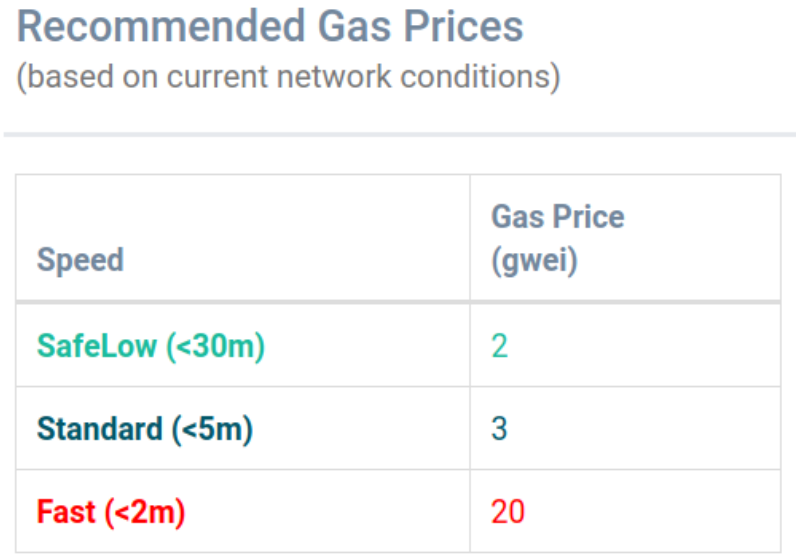
- All miners and full nodes must evaluate all transactions
  - limit computation cost
- All miners must store all state
  - limit storage use
- Short-cut the halting problem
  - There is an upper `GAS_LIMIT`, so all programs will halt

Every EVM operation has a fixed gas cost

	opcodes	**gas cost**
Basic operations	`ADD, MUL, PUSH, JUMP`	2-10
Storage read	`SLOAD`	200
Storage write	`SSTORE`	5000
Storage write (from zero)	`SSTORE`	20000
Storage zeroize	`SSTORE`	-10000
Contract call	`CALL, CODECALL,` etc.	700
Transaction overhead	n/a	21000
Contract creation	n/a	32000
Contract destruction	`SELFDESTRUCT`	-19000

All transactions specify START_GAS, GAS_PRICE

1. If `START_GAS × GAS_PRICE > caller.balance`, halt
2. Deduct `START_GAS × GAS_PRICE` from `caller.balance`
3. Set `GAS = START_GAS`
4. Run code, deducting from GAS
5. For negative values, add `to GAS_REFUND`
   - GAS only decreases
6. After termination, add `GAS_REFUND` to `caller.balance`



Recommended Gas Prices
(based on current network conditions)

Speed	Gas Price (gwei)
SafeLow (<30m)	2
Standard (<5m)	3
Fast (<2m)	20

Back of envelope numbers you should know

Average gas price:

~20gigawei = 0.00000002 ether

Price of Ether (as of March ‘18):

~$500 per Ether

Cost per transaction:

21000 gas “base” for a transaction = $0.21 ***(21 cents per transaction)***

Cost of data:

~75 gas per byte of data stored = $0.77/kB ***(77 cents per kilobyte)***

Gas limit per block: 4,000,000 ⇒ 53 kilobytes per block (2.66MB per 10 min)

Polite contracts call `revert` on errors

```
uint8 numCandidates;
uint32 votingFee;
mapping(address => bool) hasVoted;
mapping(uint8 => uint32) numVotes;

/// Cast a vote for a designated candidate
function castVote(uint8 candidate) {
        if (msg.value < votingFee)
                return;
        if (hasVoted[msg.sender])
                revert();

        hasVoted[msg.sender] = true;
        numVotes[candidate] += 1;
}
```

`revert()` ensures no effects persisted except gas consumption

Out-of-gas exceptions are bad news

- State reverts to previous value
  - Except that START_GAS * GAS_PRICE is still deducted

Built-in support for calling other contracts

- `a.transfer(x)` sends x to address a
  - returns 0 if this fails due to call stack

- `foo.call.value(3).gas(20764)( bytes4(sha3("bar()")));`
  - also `callcode, delegatecall`
  - default is 0 value, all available gas

- `new` constructor deploys a new contract
  - Careful, it's expensive!

Remember:
Smart contracts code is fixed *forever*.
Calls required to update functionality

Built-in support for calling other contracts

- Contract member variables; if public, automatically defines a “getter”
- Modifiers `payable, constant, returns()`, also modifiers can be user defined
- Macros / Internal Functions `internal` modifier -> does not require a “message”
- Type conversions: `int(x), uint256(x), bool(x)`
- Structs, arrays, mappings, memory vs storage

    array: `int[2] x;`

    hashmap `mapping ( int[2] => int );`

- Throwing exceptions `throw;  // exceptions contain no data`
- Units (currency: “eth”, “wei”, etc.) `3 * (2 eth)`

Economics of gas are similar to transaction fees

- Miners choose transactions based on GAS_PRICE
- In theory, they should not care which opcodes are used
  - In practice, some “overpriced” opcodes may be preferred
- Maximum gas limit per block
  - Miners can slowly raise it, each block votes

Authorization in smart contracts

Can any user call arbitrary functions in contracts?

- Yes. The contract must explicitly restrict access to sensitive functions

```
contract OwneableWallet {
   address owner = 0x1234;

  function critical() {
    msg.sender.transfer(10);
  }
}
```

Authorization in smart contracts

Can any user call arbitrary functions in contracts?

- Yes. The contract must explicitly restrict access to sensitive functions

```
contract OwneableWallet {
  address owner = 0x1234;

  function critical() {
    require(msg.sender == owner);
    msg.sender.transfer(10);
  }
}
```

Transaction reverts if this evaluates to false

Ethereum Virtual Machine (EVM)

Operation type	Description	OPCodes
Arithmetic	Encode calculations and numerical expressions	`Add, Mul, Sub, Div, LT, EQ`
Control-flow	Encode conditional jumps	`Jump, JumpZ`
Crypto	Compute hash functions	`SHA3`
Environment	Fetch block and transaction information	`Balance, Caller, Callvalue, Calldataload, Gas, Gaslimit Timestamp, Difficulty, Blockhash`
Memory / storage	Read from / write to memory and storage	`MLtore, MLoad, SStore, SLoad`
System	Message call into a contract	`Call`

http://yellowpaper.io/