Solidity

Ethereum Lab

atzeinicola@gmail.com

Overview

- Solidity
  - programming features
  - blockchain related features
  - how to invoke a contract (0.25 bonus)
  - how to create a contract (0.50 bonus)

- Web3 interface
  - how to develop a real Dapp

Solidity

http://solidity.readthedocs.io/en/develop/solidity-in-depth.html

https://ethereum.github.io/browser-solidity/

Types

Solidity is a statically typed language, which means that the type of each variable (state and local) needs to be specified (or at least known) at compile-time. Solidity provides several elementary types which can be combined to form complex types.

(Solidity documentation)

- Value Types
  - variables of these types will always be passed by value (bool, integer, address, string, etc.)
  - fit into 256 bits
- Reference Types
  - variables of these types will always be passed by reference
  - do not always fit into 256 bits
  - copying them can be quite expensive: you have to think about whether you want them to be stored in **memory** (which is not persisting) or **storage** (where the state variables are held).

Booleans

`bool`: the possible values are constants `true` and `false`

Operators:

- `!`
- `&&`
- `||`
- `==`
- `!=`

The operators `||` and `&&` apply the common short-circuiting rules. This means that in the expression `f(x)||g(y)`, if `f(x)` evaluates to true, `g(y)` will not be evaluated even if it may have side-effects.

Integers

`int`/`uint`: signed and unsigned integers of various sizes.

Operators:

- Comparisons: `<=` `<` `==` `!=` `>=` `>`
- Bit operators: `&` `|` `^` (bitwise exclusive or) `~` (bitwise negation)
- Arithmetic operators: `+` `-` `*` `/` `%` `**` (exponentiation)

Division of two integers always truncates.
Division by zero and modulus with zero throws an exception.

Integers (2)

Integers are represented using 256 bit

Keywords `uint8` to `uint256` in steps of `8` (unsigned of 8 up to 256 bits) and `int8` to `int256`.

`uint` and `int` are aliases for `uint256` and `int256`, respectively.

Address

`address`: holds a 20 byte value (size of an Ethereum address)

Members

- `<address>.balance`: returns the balance of the address
- `<address>.send(n)`: transfer `n` wei from the executing contract to the specified address

Special keywords

- `this`: returns the address of the executing contract

Examples

- `this.balance`: returns the balance of the executing contract
- `address(0x42).send(3)`: transfers 3 wei from the executing contract to the user (or contract) with address 0x42

String

String literals are written with either double or single-quotes (`"foo"` or `'bar'`). They do not imply trailing zeroes as in C; `"foo"` represents three bytes not four.

As with integer literals, their type can vary, but they are implicitly convertible to `bytes1 ... bytes32`, if they fit, to `bytes` and to `string`.

String literals support escape characters, such as `\n`, `\xNN` and `\uNNNN`. `\xNN` takes a hex value and inserts the appropriate byte, while `\uNNNN` takes a Unicode codepoint and inserts an UTF-8 sequence.

Enum

Enums are one way to create a user-defined type in Solidity. They are explicitly convertible to and from all **integer types** but implicit conversion is not allowed. The explicit conversions check the value ranges at runtime and a failure causes an exception.

Typically they are converted to `uint8` on reasonable enum definition.

Example

```
enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
ActionChoices choice = ActionChoices.GoLeft;
```

Reference Types

Reference types, i.e. types which do not always fit into 256 bits have to be handled more carefully than the value-types we have already seen. Since copying them can be quite expensive, we have to think about whether we want them to be stored in **memory** (which is not persisting) or **storage** (where the state variables are held).

Data location

Every complex type has an additional annotation, the “data location”, about whether it is stored in **memory** or in **storage**. Depending on the context, there is always a default, but it can be overridden by appending either **storage** or **memory** to the type.

The default for function parameters (including return parameters) is **memory**, the default for local variables is **storage** and the location is forced to **storage** for state variables (obviously).

Struct

Solidity provides a way to define new types in the form of structs, as follows:

```
struct Funder {
  address addr;
  uint amount
}
struct Campaign {
  address beneficiary;
  uint fundingGoal;
  uint numFunders;
  uint amount;
  mapping (uint => Funder) funders;
}

Campaign campaign = Campaign(addr, n, m, p);
campaign.amount = 0;
campaign.founders[0] = Funder(addr, a)
```

Mapping

Mappings can be seen as hashtables

- virtually initialized to all zeros
- no length

Example

```
// definition
mapping (address => bool) paid;

// usage
paid[<address>] == false
```

Units and Globally Available Variables

A literal number can take a suffix of `wei`, `szabo`, `finney` or `ether` to convert between the sub-denominations of Ether.

Ether currency numbers without a postfix are assumed to be Wei.

```
1 ether == 1000 finney

1 ether == 1000000 szabo

1 ether == 1000000000000000000 wei
```

Suffixes like `seconds`, `minutes`, `hours`, `days`, `weeks` and `years` after literal numbers can be used to convert between units of time where `seconds` are the base unit and units are considered naively in the following way:

- `1 == 1 seconds`
- `1 minutes == 60 seconds`
- `1 hours == 60 minutes`
- `1 days == 24 hours`
- `1 weeks = 7 days`
- `1 years = 365 days`

Block and Transaction Properties

- `block.blockhash(uint blockNumber)`: returns (bytes32): hash of the given block - only works for 256 most recent blocks excluding current
- `block.coinbase`: current block miner's address
- `block.difficulty`: current block difficulty (uint)
- `block.gaslimit`: current block gaslimit (uint)
- `block.number`: current block number (uint)
- `block.timestamp`: current block timestamp (uint)
- `msg.data`: complete calldata (bytes)
- `msg.gas`: remaining gas (uint)
- `msg.sender`: sender of the message (address)
- `msg.sig`: first four bytes of the calldata (i.e. function identifier) (bytes4)
- `msg.value`: number of wei sent with the message (uint)
- `now`: current block timestamp (alias for `block.timestamp`)
- `tx.gasprice`: gas price of the transaction (uint)
- `tx.origin`: sender of the transaction (full call chain) (address)

Cryptographic Functions

- `keccak256(...) returns (bytes32)`: compute the Ethereum-SHA-3 (Keccak-256) hash of the input arguments
- `sha3(...) returns (bytes32)`: alias to keccak256()
- `sha256(...) returns (bytes32)`: compute the SHA-256 hash of the input arguments

The following are all identical:

```
sha3("ab", "c")
sha3("abc")
sha3(0x616263)
sha3(6382179)
sha3(97, 98, 99)
```

Events

Events allow the convenient usage of the EVM logging facilities, which in turn can be used to “invoke” JavaScript callbacks in the user interface of a dapp, which listen for these events.

When they are called, they cause the arguments to be stored in the transaction’s log - a special data structure in the blockchain. These logs are associated with the address of the contract and will be incorporated into the blockchain and stay there forever.

Events (2)

```
contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );

    function deposit(bytes32 _id) {
        // Any call to this function (even deeply nested) can
        // be detected from the JavaScript API by filtering
        // for `Deposit` to be called.
        Deposit(msg.sender, _id, msg.value);
    }
}
```

Function Modifiers

Modifiers can be used to easily change the behaviour of functions, for example to automatically check a condition prior to executing the function.

```
contract MyContract {
  address owner;
  function MyContract() { owner = msg.sender; }

  modifier onlyOwner {
    if (msg.sender != owner) throw;
    _;
  }
  function changeOwner(address newOwner) onlyOwner {
    owner = newOwner;
  }
}
```

Functions

A function have to declare the keyword `payable` in order to receive Ether.

Fallback function: a function without arguments

Function and field visibility

`public` function: can be invoked externally by users or contracts (they are included within the dispatching mechanism). By default, all functions are public. Conversely, `private` functions are not exposed.

`public` field: the compiler automatically creates accessor functions for all public state variables.

`constant` function: executed without generating a transaction

`constant` field: the compiler does not reserve a storage slot for these variables and every occurrence is replaced by their constant value.

Use a contract through the Ethreum Wallet

Watch a contract

The Ethereum Wallet allows to interact with a contract. In order to do this, you must specify a name, its address, and its ABI interface.

All the contracts compiled and deployed using the Ethereum Wallet are automatically "watched".

Assignment (5 minutes - 0.25 pts)

I published a contract on the testnet at address
`0x52F8380a21D6edc240Fdf1415eD6da80CA212d6C`

I also published the source code on the testnet explorer (testnet.etherscan.io)

Your task is

- add the contract to your wallet
- invoke the function `register(uint ID)` with your identification number (ID)

Warning:

- you can help your colleague, but if you submit your friend's ID with your address, **your submission will be nullified**

Proof of presence

Assignment (10 minutes - 0.50 pts)

Create a contract with the following constraints:

- a **public** field `owner` to store the creator of the contract (use `msg.sender` within the constructor)
- a function `changeOwner(address)` that can be executed only by the owner of the contract and update the `owner` field (see the example in the previous slide)
- an event `paidBy` with two field, "from" and "value" of type `address` and `uint` respectively
- a fallback function that permits the contract to receive Ether **and** log the event `paidBy` (see the example in the previous slide)
- a function `withdraw` that that can be executed only by the owner of the contract and transfer all the Ether from the contract to the owner address (use `this.balance` and the `send` function)

Assignment (2) (10 minutes - 0.50 pts)

Ones the contract is created:

- charge your contract with 0.5 Ether (use the Ethereum Wallet or the geth client)
- change the ownership of the contract by setting my address:
  `0x836b9a551dE259e19a9d28da4feE87fec4254256`
- if I can successfully withdraw the balance of your contract, you'll take your bonus
- submit the contract you created via Moodle

Advertise the contract

Use the Ethereum Wallet application: demo

Use the geth client:

```
> personal.unlockAccount("<your-address>")

> eth.sendTransaction({
  from: "<your-address>",
  data: "60606040523461000575b3360006000610..."
})
    0x......

> eth.pendingTransactions
    [...]
```

Web3 interface

Idea

The Ethereum API library is implemented in different languages (JS, Java, etc.) and can be used by user applications to

- publish contracts
- invoke contracts
- handle events triggered by a contract
- ...

The library needs to connect to a running node, i.e. a computer running one of the Ethereum client (geth, parity, etc.), which expose the APIs

The geth client can expose various APIs using different protocols (http, ws, etc.)

Activate geth RPC

```
geth --testnet --rpc --rpcapi "eth,personal,web3"
```

It starts geth and exposes the objects `eth`, `personal` and `web3` via http POST

https://github.com/ethereum/wiki/wiki/JSON-RPC

Next lesson:

- we will create a Node.js application which exploits the JS API library (you'll receive detailed instructions on how to configure your virtual machine)
- we will talk about the project