Contract Tutorial

Felix Lange edited this page on 1 Nov 2016 · 26 revisions

Introduction

Now that you mastered the basics on how to get started and how to send ether, it's time to get your hands dirty in what really makes ethereum stand out of the crowd: smart contracts. Smart contracts are pieces of code that live on the blockchain and execute commands exactly how they were told to. They can read other contracts, make decisions, send ether and execute other contracts. Contracts will exist and run as long as the whole network exists, and will only stop if they run out of gas or if they were programmed to self destruct.

What can you do with contracts? You can do almost anything really, but for this guide let's do some simple things: You will get funds through a crowdfunding that, if successful, will supply a radically transparent and democratic organization that will only obey its own citizens, will never swerve away from its constitution and cannot be censored or shut down. And all that in less than 300 lines of code.

So let's start now.

Your first citizen: the greeter

Now that you've mastered the basics of Ethereum, let's move into your first serious contract. The Frontier is a big open territory and sometimes you might feel lonely, so our first order of business will be to create a little automatic companion to greet you whenever you feel lonely. We'll call him the "Greeter".

The Greeter is an intelligent digital entity that lives on the blockchain and is able to have conversations with anyone who interacts with it, based on its input. It might not be a talker, but it's a great listener. Here is its code:

```
contract mortal {
    /* Define variable owner of the type address*/
    address owner;

    /* this function is executed at initialization and sets the owner of the contract */
    function mortal() { owner = msg.sender; }

    /* Function to recover the funds on the contract */
    function kill() { if (msg.sender == owner) suicide(owner); }
}

contract greeter is mortal {
```

```
    /* define variable greeting of the type string */
    string greeting;

    /* this runs when the contract is executed */
    function greeter(string _greeting) public {
        greeting = _greeting;
    }

    /* main function */
    function greet() constant returns (string) {
        return greeting;
    }
}
```

You'll notice that there are two different contracts in this code: "*mortal*" and "*greeter*". This is because Solidity (the high level contract language we are using) has *inheritance*, meaning that one contract can inherit characteristics of another. This is very useful to simplify coding as common traits of contracts don't need to be rewritten every time, and all contracts can be written in smaller, more readable chunks. So by just declaring that *greeter is mortal* you inherited all characteristics from the "mortal" contract and kept the greeter code simple and easy to read.

The inherited characteristic "*mortal*" simply means that the greeter contract can be killed by its owner, to clean up the blockchain and recover funds locked into it when the contract is no longer needed. Contracts in ethereum are, by default, immortal and have no owner, meaning that once deployed the author has no special privileges anymore. Consider this before deploying.

Installing a compiler

Before you are able to Deploy it though, you'll need two things: the compiled code, and the Application Binary Interface, which is a sort of reference template that defines how to interact with the contract.

The first you can get by using a compiler. You should have a solidity compiler built in on your geth console. To test it, use this command:

```
eth.getCompilers()
```

If you have it installed, it should output something like this:

```
['Solidity' ]
```

If instead the command returns an error, then you need to install it.

Using an online compiler

If you don't have solC installed, we have a online solidity compiler available. But be aware that **if the compiler is compromised, your contract is not safe**. For this reason, if you want to use the online compiler we encourage you to host your own.

Install SolC on Ubuntu

Press control+c to exit the console (or type *exit*) and go back to the command line. Open the terminal and execute these commands:

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install solc
which solc
```

Take note of the path given by the last line, you'll need it soon.

Install SolC on Mac OSX

You need brew in order to install on your mac

```
brew tap ethereum/ethereum
brew install solidity
which solc
```

Take note of the path given by the last line, you'll need it soon.

Install SolC on Windows

You need chocolatey in order to install solc.

```
cinst -pre solC-stable
```

Windows is more complicated than that, you'll need to wait a bit more.

If you have the SolC Solidity Compiler installed, you need now reformat by removing spaces so it fits into a string variable (there are some online tools that will do this):

Compile from source

```
git clone https://github.com/ethereum/cpp-ethereum.git
mkdir cpp-ethereum/build
cd cpp-ethereum/build
cmake -DJSONRPC=OFF -DMINER=OFF -DETHKEY=OFF -DSERPENT=OFF -DGUI=OFF -DTESTS=OFF -DJSCONSOLE
make -j4
make install
which solc
```

Linking your compiler in Geth

Now go back to the console and type this command to install solC, replacing *path/to/solc* to the path that you got on the last command you did:

```
admin.setSolc("path/to/solc")
```

Now type again:

```
eth.getCompilers()
```

If you now have solC installed, then congratulations, you can keep reading. If you don't, then go to our forums or subreddit and berate us on failing to make the process easier.

Compiling your contract

If you have the compiler installed, you need now reformat your contract by removing line-breaks so it fits into a string variable (there are some online tools that will do this):

```
var greeterSource = 'contract mortal { address owner; function mortal() { owner = msg.sender

var greeterCompiled = web3.eth.compile.solidity(greeterSource)
```

You have now compiled your code. Now you need to get it ready for deployment, this includes setting some variables up, like what is your greeting. Edit the first line below to something more interesting than 'Hello World!" and execute these commands:

```
var _greeting = "Hello World!"
var greeterContract = web3.eth.contract(greeterCompiled.greeter.info.abiDefinition);

var greeter = greeterContract.new(_greeting,{from:web3.eth.accounts[0], data: greeterCompile
  if(!e) {

    if(!contract.address) {
      console.log("Contract transaction send: TransactionHash: " + contract.transactionHash

    } else {
      console.log("Contract mined! Address: " + contract.address);
      console.log(contract);
    }

  }
})
```

Using the online compiler

If you don't have solC installed, you can simply use the online compiler. Copy the source code above to the online solidity compiler and then your compiled code should appear on the left pane. Copy the code on the box labeled **Geth deploy** to a text file. Now change the first line to your greeting:

```
var _greeting = "Hello World!"
```

Now you can paste the resulting text on your geth window. Wait up to thirty seconds and you'll see a message like this:

```
Contract mined! address: 0xdaa24d02bad7e9d6a80106db164bad9399a0423e
```

You will probably be asked for the password you picked in the beginning, because you need to pay for the gas costs to deploying your contract. This contract is estimated to need 172 thousand gas to deploy (according to the online solidity compiler), at the time of writing, gas on the test net is

priced at 1 to 10 microethers per unit of gas (nicknamed "szabo" = 1 followed by 12 zeroes in wei). To know the latest price in ether all you can see the latest gas prices at the network stats page and multiply both terms.

Notice that the cost is not paid to the ethereum developers, instead it goes to the *Miners*, people who are running computers who keep the network running. Gas price is set by the market of the current supply and demand of computation. If the gas prices are too high, you can be a miner and lower your asking price.

After less than a minute, you should have a log with the contract address, this means you've sucessfully deployed your contract. You can verify the deployed code (compiled) by using this command:

```
eth.getCode(greeter.address)
```

If it returns anything other than "0x" then congratulations! Your little Greeter is live! If the contract is created again (by performing another eth.sendTransaction), it will be published to a new address.

Run the Greeter

In order to call your bot, just type the following command in your terminal:

```
greeter.greet();
```

Since this call changes nothing on the blockchain, it returns instantly and without any gas cost. You should see it return your greeting:

```
'Hello World!'
```

Getting other people to interact with your code

In order to other people to run your contract they need two things: the address where the contract is located and the ABI (Application Binary Interface) which is a sort of user manual, describing the name of its functions and how to call them. In order to get each of them run these commands:

```
greeterCompiled.greeter.info.abiDefinition;
greeter.address;
```

Then you can instantiate a javascript object which can be used to call the contract on any machine connected to the network. Replace 'ABI' and 'address' to create a contract object in javascript:

```
var greeter = eth.contract(ABI).at(Address);
```

This particular example can be instantiated by anyone by simply calling:

```
var greeter2 = eth.contract([{constant:false,inputs:[],name:'kill',outputs:[],type:'functior
```

Replace *greeterAddress* with your contract's address.

Tip: if the solidity compiler isn't properly installed in your machine, you can get the ABI from the online compiler. To do so, use the code below carefully replacing *greeterCompiled.greeter.info.abiDefinition* with the abi from your compiler.

Cleaning up after yourself:

You must be very excited to have your first contract live, but this excitement wears off sometimes, when the owners go on to write further contracts, leading to the unpleasant sight of abandoned contracts on the blockchain. In the future, blockchain rent might be implemented in order to increase the scalability of the blockchain but for now, be a good citizen and humanely put down your abandoned bots.

Unlike last time we will not be making a call as we wish to change something on the blockchain. This requires a transaction be sent to the network and a fee to be paid for the changes made. The suicide is subsidized by the network so it will cost much less than a usual transaction.

```
greeter.kill.sendTransaction({from:eth.accounts[0]})
```

You can verify that the deed is done simply seeing if this returns 0:

```
eth.getCode(greeter.contractAddress)
```

Notice that every contract has to implement its own kill clause. In this particular case only the account that created the contract can kill it.

If you don't add any kill clause it could potentially live forever (or at least until the frontier contracts are all wiped) independently of you and any earthly borders, so before you put it live check what your local laws say about it, including any possible limitation on technology export, restrictions on speech and maybe any legislation on the civil rights of sentient digital beings. Treat your bots humanely.

The Coin

What is a coin? Coins are much more interesting and useful than they seem, they are in essence just a tradeable token, but can become much more, depending on how you use them. Its value depends on what you do with it: a token can be used to control access (**an entrance ticket**), can be used for voting rights in an organization (**a share**), can be placeholders for an asset held by a third party (**a certificate of ownership**) or even be simply used as an exchange of value within a community (**a currency**).

You could do all those things by creating a centralized server, but using an Ethereum token contract comes with some free functionalities: for one, it's a decentralized service and tokens can be still exchanged even if the original service goes down for any reason. The code can guarantee that no tokens will ever be created other than the ones set in the original code. Finally, by having each user hold their own token, this eliminates the scenarios where one single server break-in can result in the loss of funds from thousands of clients.

You could create your own token on a different blockchain, but creating on ethereum is easier — so you can focus your energy on the innovation that will make your coin stand out - and it's more secure, as your security is provided by all the miners who are supporting the ethereum network. Finally, by creating your token in Ethereum, your coin will be compatible with any other contract running on ethereum.

The Code

This is the code for the contract we're building:

```
contract token {
    mapping (address => uint) public coinBalanceOf;
    event CoinTransfer(address sender, address receiver, uint amount);

  /* Initializes contract with initial supply tokens to the creator of the contract */
  function token(uint supply) {
       coinBalanceOf[msg.sender] = supply;
    }

  /* Very simple trade function */
    function sendCoin(address receiver, uint amount) returns(bool sufficient) {
        if (coinBalanceOf[msg.sender] < amount) return false;
        coinBalanceOf[msg.sender] -= amount;
        coinBalanceOf[receiver] += amount;
        CoinTransfer(msg.sender, receiver, amount);
        return true;
    }
}
```

If you have ever programmed, you won't find it hard to understand what it does: it is a contract that generates 10 thousand tokens to the creator of the contract, and then allows anyone with enough balance to send it to others. These tokens are the minimum tradeable unit and cannot be subdivided, but for the final users could be presented as a 100 units subdividable by 100 subunits, so owning a single token would represent having 0.01% of the total. If your application needs more fine grained atomic divisibility, then just increase the initial issuance amount.

In this example we declared the variable "coinBalanceOf" to be public, this will automatically create a function that checks any account's balance.

Compile and Deploy

So let's run it!

```
var tokenSource = ' contract token { mapping (address => uint) public coinBalanceOf; event C

var tokenCompiled = eth.compile.solidity(tokenSource)
```

Now let's set up the contract, just like we did in the previous section. Change the "initial Supply" to the amount of non divisible tokens you want to create. If you want to have divisible units, you should do that on the user frontend but keep them represented in the minimun unit of account.

```
var supply = 10000;
var tokenContract = web3.eth.contract(tokenCompiled.token.info.abiDefinition);
```

```
var token = tokenContract.new(
  supply,
  {
    from:web3.eth.accounts[0],
    data:tokenCompiled.token.code,
    gas: 1000000
  }, function(e, contract){
    if(!e) {

      if(!contract.address) {
        console.log("Contract transaction send: TransactionHash: " + contract.transactionHas

      } else {
        console.log("Contract mined! Address: " + contract.address);
        console.log(contract);
      }

    }
})
```

Online Compiler

If you don't have solC installed, you can simply use the online compiler. Copy the contract code to the online solidity compiler, if there are no errors on the contract you should see a text box labeled **Geth Deploy**. Copy the content to a text file so you can change the first line to set the initial supply, like this:

```
var supply = 10000;
```

Now you can paste the resulting text on your geth window. Wait up to thirty seconds and you'll see a message like this:

```
Contract mined! address: 0xdaa24d02bad7e9d6a80106db164bad9399a0423e
```

Check balance watching coin transfers

If everything worked correctly, you should be able to check your own balance with:

```
token.coinBalanceOf(eth.accounts[0]) + " tokens"
```

It should have all the 10 000 tokens that were created once the contract was published. Since there is not any other defined way for new coins to be issued, these are all that will ever exist.

You can set up a **Watcher** to react whenever anyone sends a coin using your contract. Here's how you do it:

```
var event = token.CoinTransfer({}, '', function(error, result){
  if (!error)
    console.log("Coin transfer: " + result.args.amount + " tokens were sent. Balances now ar
});
```

Sending coins

Now of course those tokens aren't very useful if you hoard them all, so in order to send them to someone else, use this command:

```
token.sendCoin.sendTransaction(eth.accounts[1], 1000, {from: eth.accounts[0]})
```

If a friend has registered a name on the registrar you can send it without knowing their address, doing this:

```
token.sendCoin.sendTransaction(registrar.addr("Alice"), 2000, {from: eth.accounts[0]})
```

Note that our first function **coinBalanceOf** was simply called directly on the contract instance and returned a value. This was possible since this was a simple read operation that incurs no state change and which executes locally and synchronously. Our second function **sendCoin** needs a **.sendTransaction()** call. Since this function is meant to change the state (write operation), it is sent as a transaction to the network to be picked up by miners and included in the canonical blockchain. As a result the consensus state of all participant nodes will adequately reflect the state changes resulting from executing the transaction. Sender address needs to be sent as part of the transaction to fund the fuel needed to run the transaction. Now, wait a minute and check both accounts balances:

```
token.coinBalanceOf.call(eth.accounts[0])/100 + "% of all tokens"
token.coinBalanceOf.call(eth.accounts[1])/100 + "% of all tokens"
token.coinBalanceOf.call(registrar.addr("Alice"))/100 + "% of all tokens"
```

Improvement suggestions

Right now this cryptocurrency is quite limited as there will only ever be 10,000 coins and all are controlled by the coin creator, but you can change that. You could for example reward ethereum miners, by creating a transaction that will reward who found the current block:

```
mapping (uint => address) miningReward;
function claimMiningReward() {
  if (miningReward[block.number] == 0) {
    coinBalanceOf[block.coinbase] += 1;
    miningReward[block.number] = block.coinbase;
  }
}
```

You could modify this to anything else: maybe reward someone who finds a solution for a new puzzle, wins a game of chess, install a solar panel—as long as that can be somehow translated to a contract. Or maybe you want to create a central bank for your personal country, so you can keep track of hours worked, favours owed or control of property. In that case you might want to add a function to allow the bank to remotely freeze funds and destroy tokens if needed.

Register a name for your coin

The commands mentioned only work because you have token javascript object instantiated on your local machine. If you send tokens to someone they won't be able to move them forward

because they don't have the same object and wont know where to look for your contract or call its functions. In fact if you restart your console these objects will be deleted and the contracts you've been working on will be lost forever. So how do you instantiate the contract on a clean machine?

There are two ways. Let's start with the quick and dirty, providing your friends with a reference to your contract's ABI:

```
token = eth.contract([{constant:false,inputs:[{name:'receiver',type:'address'},{name:'amount
```

Just replace the address at the end for your own token address, then anyone that uses this snippet will immediately be able to use your contract. Of course this will work only for this specific contract so let's analyze step by step and see how to improve this code so you'll be able to use it anywhere.

All accounts are referenced in the network by their public address. But addresses are long, difficult to write down, hard to memorize and immutable. The last one is specially important if you want to be able to generate fresh accounts in your name, or upgrade the code of your contract. In order to solve this, there is a default name registrar contract which is used to associate the long addresses with short, human-friendly names.

Names have to use only alphanumeric characters and, cannot contain blank spaces. In future releases the name registrar will likely implement a bidding process to prevent name squatting but for now, it works on a first come first served basis: as long as no one else registered the name, you can claim it.

First, if you register a name, then you won't need the hardcoded address in the end. Select a nice coin name and try to reserve it for yourself. First, select your name:

```
var tokenName = "MyFirstCoin"
```

Then, check the availability of your name:

```
registrar.addr(tokenName)
```

If that function returns "0x00..", you can claim it to yourself:

```
registrar.reserve.sendTransaction(tokenName, {from: eth.accounts[0]});
```

Wait for the previous transaction to be picked up. Wait up to thirty seconds and then try:

```
registrar.owner(myName)
```

If it returns your address, it means you own that name and are able to set your chosen name to any address you want:

```
registrar.setAddress.sendTransaction(tokenName, token.address, true,{from: eth.accounts[0]})
```

*You can replace **token.address** for **eth.accounts[0]** if you want to use it as a personal nickname.*

Wait a little bit for that transaction to be picked up too and test it:

```
registrar.addr("MyFirstCoin")
```

You can send a transaction to anyone or any contract by name instead of account simply by typing

```
eth.sendTransaction({from: eth.accounts[0], to: registrar.addr("MyFirstCoin"), value: web3.t
```

Tip: don't mix registrar.addr for registrar.owner. The first is to which address that name is pointed at: anyone can point a name to anywhere else, just like anyone can forward a link to google.com, but only the owner of the name can change and update the link. You can set both to be the same address.

This should now return your token address, meaning that now the previous code to instantiate could use a name instead of an address.

```
token = eth.contract([{constant:false,inputs:[{name:'receiver',type:'address'},{name:'amount
```

This also means that the owner of the coin can update the coin by pointing the registrar to the new contract. This would, of course, require the coin holders trust the owner set at registrar.owner("MyFirstCoin")

Of course this is a rather unpleasant big chunk of code just to allow others to interact with a contract. There are some avenues being investigated to upload the contract ABI to the network, so that all the user will need is the contract name. You can read about these approaches but they are very experimental and will certainly change in the future.

Learn More

- Meta coin standard is a proposed standardization of function names for coin and token contracts, to allow them to be automatically added to other ethereum contract that utilizes trading, like exchanges or escrow.
- Formal proofing is a way where the contract developer will be able to assert some invariant qualities of the contract, like the total cap of the coin. *Not yet implemented*.

Crowdfund your idea

Sometimes a good idea takes a lot of funds and collective effort. You could ask for donations, but donors prefer to give to projects they are more certain that will get traction and proper funding. This is an example where a crowdfunding would be ideal: you set up a goal and a deadline for reaching it. If you miss your goal, the donations are returned, therefore reducing the risk for donors. Since the code is open and auditable, there is no need for a centralized trusted platform and therefore the only fees everyone will pay are just the gas fees.

In a crowdfunding prizes are usually given. This would require you to get everyone's contact information and keep track of who owns what. But since you just created your own token, why not use that to keep track of the prizes? This allows donors to immediately own something after they

donated. They can store it safely, but they can also sell or trade it if they realize they don't want the prize anymore. If your idea is something physical, all you have to do after the project is completed is to give the product to everyone who sends you back a token. If the project is digital the token itself can immediately be used for users to participate or get entry on your project.

The code

The way this particular crowdsale contract works is that you set an exchange rate for your token and then the donors will immediately get a proportional amount of tokens in exchange of their ether. You will also choose a funding goal and a deadline: once that deadline is over you can ping the contract and if the goal was reached it will send the ether raised to you, otherwise it goes back to the donors. Donors keep their tokens even if the project doesn't reach its goal, as a proof that they helped.

```
contract token { mapping (address => uint) public coinBalanceOf; function token() {}  functi

contract Crowdsale {

    address public beneficiary;
    uint public fundingGoal; uint public amountRaised; uint public deadline; uint public pri
    token public tokenReward;
    Funder[] public funders;
    event FundTransfer(address backer, uint amount, bool isContribution);

    /* data structure to hold information about campaign contributors */
    struct Funder {
        address addr;
        uint amount;
    }

    /*  at initialization, setup the owner */
    function Crowdsale(address _beneficiary, uint _fundingGoal, uint _duration, uint _price,
        beneficiary = _beneficiary;
        fundingGoal = _fundingGoal;
        deadline = now + _duration * 1 minutes;
        price = _price;
        tokenReward = token(_reward);
    }

    /* The function without name is the default function that is called whenever anyone send
    function () {
        uint amount = msg.value;
        funders[funders.length++] = Funder({addr: msg.sender, amount: amount});
        amountRaised += amount;
        tokenReward.sendCoin(msg.sender, amount / price);
        FundTransfer(msg.sender, amount, true);
    }

    modifier afterDeadline() { if (now >= deadline) _ }

    /* checks if the goal or time limit has been reached and ends the campaign */
    function checkGoalReached() afterDeadline {
        if (amountRaised >= fundingGoal){
            beneficiary.send(amountRaised);
            FundTransfer(beneficiary, amountRaised, false);
        } else {
            FundTransfer(0, 11, false);
            for (uint i = 0; i < funders.length; ++i) {
              funders[i].addr.send(funders[i].amount);
```

```
                FundTransfer(funders[i].addr, funders[i].amount, false);
            }
        }
        suicide(beneficiary);
    }
}
```

Set the parameters

Before we go further, let's start by setting the parameters of the crowdsale:

```
var _beneficiary = eth.accounts[1];    // create an account for this
var _fundingGoal = web3.toWei(100, "ether"); // raises 100 ether
var _duration = 30;     // number of minutes the campaign will last
var _price = web3.toWei(0.02, "ether"); // the price of the tokens, in ether
var _reward = token.address;   // the token contract address.
```

On Beneficiary put the new address that will receive the raised funds. The funding goal is the amount of ether to be raised. Deadline is measured in blocktimes which average 12 seconds, so the default is about 4 weeks. The price is tricky: but just change the number 2 for the amount of tokens the contributors will receive for each ether donated. Finally reward should be the address of the token contract you created in the last section.

In this example you are selling on the crowdsale half of all the tokens that ever existed, in exchange for 100 ether. Decide those parameters very carefully as they will play a very important role in the next part of our guide.

Deploy

You know the drill: if you are using the solC compiler,remove line breaks and copy the following commands on the terminal:

```
var crowdsaleCompiled = eth.compile.solidity(' contract token { mapping (address => uint) pu

var crowdsaleContract = web3.eth.contract(crowdsaleCompiled.Crowdsale.info.abiDefinition);
var crowdsale = crowdsaleContract.new(
  _beneficiary,
  _fundingGoal,
  _duration,
  _price,
  _reward,
  {
    from:web3.eth.accounts[0],
    data:crowdsaleCompiled.Crowdsale.code,
    gas: 1000000
  }, function(e, contract){
    if(!e) {

      if(!contract.address) {
        console.log("Contract transaction send: TransactionHash: " + contract.transactionHas

      } else {
        console.log("Contract mined! Address: " + contract.address);
        console.log(contract);
      }
```

```
    }    })
```

If you are using the *online compiler* Copy the contract code to the online solidity compiler, and then grab the content of the box labeled Geth Deploy. Since you have already set the parameters, you don't need to change anything to that text, simply paste the resulting text on your geth window.

Wait up to thirty seconds and you'll see a message like this:

```
Contract mined! address: 0xdaa24d02bad7e9d6a80106db164bad9399a0423e
```

If you received that alert then your code should be online. You can always double check by doing this:

```
eth.getCode(crowdsale.address)
```

Now fund your newly created contract with the necessary tokens so it can automatically distribute rewards to the contributors!

```
token.sendCoin.sendTransaction(crowdsale.address, 5000,{from: eth.accounts[0]})
```

After the transaction is picked, you can check the amount of tokens the crowdsale address has, and all other variables this way:

```
"Current crowdsale must raise " + web3.fromWei(crowdsale.fundingGoal.call(), "ether") + " et
```

Put some watchers on

You want to be alerted whenever your crowdsale receives new funds, so paste this code:

```
var event = crowdsale.FundTransfer({}, '', function(error, result){
  if (!error)

    if (result.args.isContribution) {
        console.log("\n New backer! Received " + web3.fromWei(result.args.amount, "ether") +

        console.log( "\n The current funding at " +( 100 *  crowdsale.amountRaised.call() /

        var timeleft = Math.floor(Date.now() / 1000)-crowdsale.deadline();
        if (timeleft>3600) {  console.log("Deadline has passed, " + Math.floor(timeleft/3600
        } else if (timeleft>0) {  console.log("Deadline has passed, " + Math.floor(timeleft/
        } else if (timeleft>-3600) {  console.log(Math.floor(-1*timeleft/60) + " minutes unt
        } else {  console.log(Math.floor(-1*timeleft/3600) + " hours until deadline")
        }

    } else {
        console.log("Funds transferred from crowdsale account: " + web3.fromWei(result.args.
    }

});
```

Register the contract

You are now set. Anyone can now contribute by simply sending ether to the crowdsale address, but to make it even simpler, let's register a name for your sale. First, pick a name for your crowdsale:

```
var name = "mycrowdsale"
```

Check if that's available and register:

```
registrar.addr(name)
registrar.reserve.sendTransaction(name, {from: eth.accounts[0]});
```

Wait for the previous transaction to be picked up and then:

```
registrar.setAddress.sendTransaction(name, crowdsale.address, true,{from: eth.accounts[0]});
```

Contribute to the crowdsale

Contributing to the crowdsale is very simple, it doesn't even require instantiating the contract. This is because the crowdsale responds to simple ether deposits, so anyone that sends ether to the crowdsale will automatically receive a reward. Anyone can contribute to it by simply executing this command:

```
var amount = web3.toWei(5, "ether") // decide how much to contribute

eth.sendTransaction({from: eth.accounts[0], to: crowdsale.address, value: amount, gas: 10000
```

Alternatively, if you want someone else to send it, they can even use the name registrar to contribute:

```
eth.sendTransaction({from: eth.accounts[0], to: registrar.addr("mycrowdsale"), value: amount
```

Now wait a minute for the blocks to pickup and you can check if the contract received the ether by doing any of these commands:

```
web3.fromWei(crowdsale.amountRaised.call(), "ether") + " ether"
token.coinBalanceOf.call(eth.accounts[0]) + " tokens"
token.coinBalanceOf.call(crowdsale.address) + " tokens"
```

Recover funds

Once the deadline is passed someone has to wake up the contract to have the funds sent to either the beneficiary or back to the funders (if it failed). This happens because there is no such thing as

an active loop or timer on ethereum so any future transactions must be pinged by someone.

```
crowdsale.checkGoalReached.sendTransaction({from:eth.accounts[0], gas: 2000000})
```

You can check your accounts with these lines of code:

```
web3.fromWei(eth.getBalance(eth.accounts[0]), "ether") + " ether"
web3.fromWei(eth.getBalance(eth.accounts[1]), "ether") + " ether"
token.coinBalanceOf.call(eth.accounts[0]) + " tokens"
token.coinBalanceOf.call(eth.accounts[1]) + " tokens"
```

The crowdsale instance is setup to self destruct once it has done its job, so if the deadline is over and everyone got their prizes the contract is no more, as you can see by running this:

```
eth.getCode(crowdsale.address)
```

So you raised a 100 ethers and successfully distributed your original coin among the crowdsale donors. What could you do next with those things?

Democracy DAO

So far you have created a tradeable token and you successfully distributed it among all those who were willing to help fundraise a 100 ethers. That's all very interesting but what exactly are those tokens for? Why would anyone want to own or trade it for anything else valuable? If you can convince your new token is the next big money maybe others will want it, but so far your token offers no value per se. We are going to change that, by creating your first decentralized autonomous organization, or DAO.

Think of the DAO as the constitution of a country, the executive branch of a government or maybe like a robotic manager for an organization. The DAO receives the money that your organization raises, keeps it safe and uses it to fund whatever its members want. The robot is incorruptible, will never defraud the bank, never create secret plans, never use the money for anything other than what its constituents voted on. The DAO will never disappear, never run away and cannot be controlled by anyone other than its own citizens.

The token we distributed using the crowdsale is the only citizen document needed. Anyone who holds any token is able to create and vote on proposals. Similar to being a shareholder in a company, the token can be traded on the open market and the vote is proportional to amounts of tokens the voter holds.

Take a moment to dream about the revolutionary possibilities this would allow, and now you can do it yourself, in under a 100 lines of code:

The Code

```
contract token { mapping (address => uint) public coinBalanceOf;   function token() { }   fu


contract Democracy {
```

```
    uint public minimumQuorum;
    uint public debatingPeriod;
    token public voterShare;
    address public founder;
    Proposal[] public proposals;
    uint public numProposals;

    event ProposalAdded(uint proposalID, address recipient, uint amount, bytes32 data, strir
    event Voted(uint proposalID, int position, address voter);
    event ProposalTallied(uint proposalID, int result, uint quorum, bool active);

    struct Proposal {
        address recipient;
        uint amount;
        bytes32 data;
        string description;
        uint creationDate;
        bool active;
        Vote[] votes;
        mapping (address => bool) voted;
    }

    struct Vote {
        int position;
        address voter;
    }

    function Democracy(token _voterShareAddress, uint _minimumQuorum, uint _debatingPeriod)
        founder = msg.sender;
        voterShare = token(_voterShareAddress);
        minimumQuorum = _minimumQuorum || 10;
        debatingPeriod = _debatingPeriod * 1 minutes || 30 days;
    }


    function newProposal(address _recipient, uint _amount, bytes32 _data, string _descriptic
        if (voterShare.coinBalanceOf(msg.sender)>0) {
            proposalID = proposals.length++;
            Proposal p = proposals[proposalID];
            p.recipient = _recipient;
            p.amount = _amount;
            p.data = _data;
            p.description = _description;
            p.creationDate = now;
            p.active = true;
            ProposalAdded(proposalID, _recipient, _amount, _data, _description);
            numProposals = proposalID+1;
        }
    }

    function vote(uint _proposalID, int _position) returns (uint voteID){
        if (voterShare.coinBalanceOf(msg.sender)>0 && (_position >= -1 || _position <= 1 ))
            Proposal p = proposals[_proposalID];
            if (p.voted[msg.sender] == true) return;
            voteID = p.votes.length++;
            p.votes[voteID] = Vote({position: _position, voter: msg.sender});
            p.voted[msg.sender] = true;
            Voted(_proposalID,  _position, msg.sender);
        }
    }

    function executeProposal(uint _proposalID) returns (int result) {
        Proposal p = proposals[_proposalID];
```

```
        /* Check if debating period is over */
        if (now > (p.creationDate + debatingPeriod) && p.active){
            uint quorum = 0;
            /* tally the votes */
            for (uint i = 0; i <  p.votes.length; ++i) {
                Vote v = p.votes[i];
                uint voteWeight = voterShare.coinBalanceOf(v.voter);
                quorum += voteWeight;
                result += int(voteWeight) * v.position;
            }
            /* execute result */
            if (quorum > minimumQuorum && result > 0 ) {
                p.recipient.call.value(p.amount)(p.data);
                p.active = false;
            } else if (quorum > minimumQuorum && result < 0) {
                p.active = false;
            }
            ProposalTallied(_proposalID, result, quorum, p.active);
        }
    }
}
```

There's a lot of going on but it's simpler than it looks. The rules of your organization are very simple: anyone with at least one token can create proposals to send funds from the country's account. After a week of debate and votes, if it has received votes worth a total of 100 tokens or more and has more approvals than rejections, the funds will be sent. If the quorum hasn't been met or it ends on a tie, then voting is kept until it's resolved. Otherwise, the proposal is locked and kept for historical purposes.

So let's recap what this means: in the last two sections you created 10,000 tokens, sent 1,000 of those to another account you control, 2,000 to a friend named Alice and distributed 5,000 of them via a crowdsale. This means that you no longer control over 50% of the votes in the DAO, and if Alice and the community bands together, they can outvote any spending decision on the 100 ethers raised so far. This is exactly how a democracy should work. If you don't want to be a part of your country anymore the only thing you can do is sell your own tokens on a decentralized exchange and opt out, but you cannot prevent the others from doing so.

Set Up your Organization

So open your console and let's get ready to finally put your country online. First, let's set the right parameters, pick them with care:

```
var _voterShareAddress = token.address;
var _minimumQuorum = 10; // Minimun amount of voter tokens the proposal needs to pass
var _debatingPeriod = 60; // debating period, in minutes;
```

With these default parameters anyone with any tokens can make a proposal on how to spend the organization's money. The proposal has 1 hour to be debated and it will pass if it has at least votes from at least 0.1% of the total tokens and has more support than rejections. Pick those parameters with care, as you won't be able to change them in the future.

```
var daoCompiled = eth.compile.solidity('contract token { mapping (address => uint) public cc

var democracyContract = web3.eth.contract(daoCompiled.Democracy.info.abiDefinition);
```

```
var democracy = democracyContract.new(
    _voterShareAddress,
    _minimumQuorum,
    _debatingPeriod,
    {
      from:web3.eth.accounts[0],
      data:daoCompiled.Democracy.code,
      gas: 3000000
    }, function(e, contract){
      if(!e) {

        if(!contract.address) {
          console.log("Contract transaction send: TransactionHash: " + contract.transactionH

        } else {
          console.log("Contract mined! Address: " + contract.address);
          console.log(contract);
        }

      }
    })
```

If you are using the *online compiler* Copy the contract code to the online solidity compiler, and then grab the content of the box labeled Geth Deploy. Since you have already set the parameters, you don't need to change anything to that text, simply paste the resulting text on your geth window.

Wait a minute until the miners pick it up. It will cost you about 850k Gas. Once that is picked up, it's time to instantiate it and set it up, by pointing it to the correct address of the token contract you created previously.

If everything worked out, you can take a look at the whole organization by executing this string:

```
"This organization has " +  democracy.numProposals() + " proposals and uses the token at the
```

If everything is setup then your DAO should return a proposal count of 0 and an address marked as the "founder". While there are still no proposals, the founder of the DAO can change the address of the token to anything it wants.

Register your organization name

Let's also register a name for your contract so it's easily accessible (don't forget to check your name availability with registrar.addr("nameYouWant") before reserving!)

```
var name = "MyPersonalDemocracy"
registrar.reserve.sendTransaction(name, {from: eth.accounts[0]})
var democracy = eth.contract(daoCompiled.Democracy.info.abiDefinition).at(democracy.address)
democracy.setup.sendTransaction(registrar.addr("MyFirstCoin"),{from:eth.accounts[0]})
```

Wait for the previous transactions to be picked up and then:

```
registrar.setAddress.sendTransaction(name, democracy.address, true,{from: eth.accounts[0]});
```

The Democracy Watchbots

```
var event = democracy.ProposalAdded({}, '', function(error, result){
  if (!error)
    console.log("New Proposal #"+ result.args.proposalID +"!\n Send " + web3.fromWei(result.
});
var eventVote = democracy.Voted({}, '', function(error, result){
  if (!error)
    var opinion = "";
    if (result.args.position > 0) {
      opinion = "in favor"
    } else if (result.args.position < 0) {
      opinion = "against"
    } else {
      opinion = "abstaining"
    }

    console.log("Vote on Proposal #"+ result.args.proposalID +"!\n " + result.args.voter + '
});
var eventTally = democracy.ProposalTallied({}, '', function(error, result){
  if (!error)
    var totalCount = "";
    if (result.args.result > 1) {
      totalCount = "passed"
    } else if (result.args.result < 1) {
      totalCount = "rejected"
    } else {
      totalCount = "a tie"
    }
    console.log("Votes counted on Proposal #"+ result.args.proposalID +"!\n With a total of
});
```

Interacting with the DAO

After you are satisfied with what you want, it's time to get all that ether you got from the crowdfunding into your new organization:

```
eth.sendTransaction({from: eth.accounts[1], to: democracy.address, value: web3.toWei(100, "e
```

This should take only a minute and your country is ready for business! Now, as a first priority, your organisation needs a nice logo, but unless you are a designer, you have no idea how to do that. For the sake of argument let's say you find that your friend Bob is a great designer who's willing to do it for only 10 ethers, so you want to propose to hire him.

```
recipient = registrar.addr("bob");
amount =  web3.toWei(10, "ether");
shortNote = "Logo Design";

democracy.newProposal.sendTransaction( recipient, amount, '', shortNote,  {from: eth.account
```

After a minute, anyone can check the proposal recipient and amount by executing these commands:

```
"This organization has " +  (Number(democracy.numProposals())+1) + " pending proposals";
```

Keep an eye on the organization

Unlike most governments, your country's government is completely transparent and easily programmable. As a small demonstration here's a snippet of code that goes through all the current proposals and prints what they are and for whom:

```
function checkAllProposals() {
    console.log("Country Balance: " + web3.fromWei( eth.getBalance(democracy.address), "ethe
    for (i = 0; i< (Number(democracy.numProposals())); i++ ) {
        var p = democracy.proposals(i);
        var timeleft = Math.floor(((Math.floor(Date.now() / 1000)) - Number(p[4]) - Number(c
        console.log("Proposal #" + i + " Send " + web3.fromWei( p[1], "ether") + " ether to
    }
}

checkAllProposals();
```

A concerned citizen could easily write a bot that periodically pings the blockchain and then publicizes any new proposals that were put forth, guaranteeing total transparency.

Now of course you want other people to be able to vote on your proposals. You can check the crowdsale tutorial on the best way to register your contract app so that all the user needs is a name, but for now let's use the easier version. Anyone should be able to instantiate a local copy of your country in their computer by using this giant command:

```
democracy = eth.contract( [{ constant: true, inputs: [{ name: '', type: 'uint256' } ], name:
```

Then anyone who owns any of your tokens can vote on the proposals by doing this:

```
var proposalID = 0;
var position = -1; // +1 for voting yea, -1 for voting nay, 0 abstains but counts as quorum
democracy.vote.sendTransaction(proposalID, position, {from: eth.accounts[0], gas: 1000000});

var proposalID = 1;
var position = 1; // +1 for voting yea, -1 for voting nay, 0 abstains but counts as quorum
democracy.vote.sendTransaction(proposalID, position, {from: eth.accounts[0], gas: 1000000});
```

Unless you changed the basic parameters in the code, any proposal will have to be debated for at least a week until it can be executed. After that anyone—even a non-citizen—can demand the votes to be counted and the proposal to be executed. The votes are tallied and weighted at that moment and if the proposal is accepted then the ether is sent immediately and the proposal is archived. If the votes end in a tie or the minimum quorum hasn't been reached, the voting is kept open until the stalemate is resolved. If it loses, then it's archived and cannot be voted again.

```
var proposalID = 1;
democracy.executeProposal.sendTransaction(proposalID, {from: eth.accounts[0], gas: 1000000})
```

If the proposal passed then you should be able to see Bob's ethers arriving on his address:

```
web3.fromWei(eth.getBalance(democracy.address), "ether") + " ether";
web3.fromWei(eth.getBalance(registrar.addr("bob")), "ether") + " ether";
```

Try for yourself: This is a very simple democracy contract, which could be vastly improved: currently, all proposals have the same debating time and are won by direct vote and simple majority. Can you change that so it will have some situations, depending on the amount proposed, that the debate might be longer or that it would require a larger majority? Also think about some way where citizens didn't need to vote on every issue and could temporarily delegate their votes to a special representative. You might have also noticed that we added a tiny description for each proposal. This could be used as a title for the proposal or could be a hash of a larger document describing it in detail.

Let's go exploring!

You have reached the end of this tutorial, but it's just the beginning of a great adventure. Look back and see how much you accomplished: you created a living, talking robot, your own cryptocurrency, raised funds through a trustless crowdfunding and used it to kickstart your own personal democratic organization.

For the sake of simplicity, we only used the democratic organization you created to send ether around, the native currency of ethereum. While that might be good enough for some, this is only scratching the surface of what can be done. In the ethereum network contracts have all the same rights as any normal user, meaning that your organization could do any of the transactions that you executed coming from your own accounts.

What could happen next?

- The greeter contract you created at the beginning could be improved to charge ether for its services and could funnel those funds into the DAO.
- The tokens you still control could be sold on a decentralized exchange or traded for goods and services to fund further develop the first contract and grow the organization.
- Your DAO could own its own name on the name registrar, and then change where it's redirecting in order to update itself if the token holders approved.
- The organization could hold not only ethers, but any kind of other coin created on ethereum, including assets whose value are tied to the bitcoin or dollar.
- The DAO could be programmed to allow a proposal with multiple transactions, some scheduled to the future. It could also own shares of other DAO's, meaning it could vote on larger organization or be a part of a federation of DAO's.
- The Token Contract could be reprogrammed to hold ether or to hold other tokens and distribute it to the token holders. This would link the value of the token to the value of other

assets, so paying dividends could be accomplished by simply moving funds to the token address.

This all means that this tiny society you created could grow, get funding from third parties, pay recurrent salaries, own any kind of crypto-assets and even use crowdsales to fund its activities. All with full transparency, complete accountability and complete immunity from any human interference. While the network lives the contracts will execute exactly the code they were created to execute, without any exception, forever.

So what will your contract be? Will it be a country, a company, a non-profit group? What will your code do?

That's up to you.

```
golang <3
```