## WHAT

At the time of writing, Solidity is the preferred language to write contracts for the Ethereum platform. You have already come across one such contract when you deployed and interacted with.

Because a contract is deployed on the blockchain in its bytecode form, any language that comes with a compiler could be used to write contracts. In fact a formerly preferred language was Serpent. Think of it as Java, Scala, Kotlin or Clojure all compiled to run on the JVM.

geth used to have a built in compiler but this was removed for security reasons.
you could install and run solidity command line compiler but it's inconvenient and not user friendly
there's a web-based IDE for solidity called remix https://remix.ethereum.org
don't worry everything runs in the browser!

Here are some links to proper documentation:

https://solidity.readthedocs.io/en/develop/

intro to contracts:

https://solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html

contract basics:

https://solidity.readthedocs.io/en/develop/structure-of-a-contract.html

more:

https://solidity.readthedocs.io/en/develop/contracts.html

data types:

https://solidity.readthedocs.io/en/develop/types.html

important info regarding functions:

http://solidity.readthedocs.io/en/develop/contracts.html#visibility-and-getters

security pitfalls:

https://solidity.readthedocs.io/en/develop/security-considerations.html#pitfalls

## Example contract:

```
Example Solidity Smart Contract DeadDrop.sol
**********************Solidty code starts below!*****************************

pragma solidity ^0.4.18;
//the above lines specify the lowest version of the compiler that can be used..

//a dead-drop is a thing or situation where you can leave somebody a message..
//comments are preceded with a double slash

//all contracts have the line below, that specifies the contract name
//like a Class in Object Oriented Programming
contract DeadDrop{
    string message; //a variable that stores the message
    //this function has the same name as the contract and is called when the contract is created
    function DeadDrop (string data) public { //public means anybody can use it
        message = data; // store the provided data as the message
    }

    //this function returns the message
    function getMessage() constant public returns (string)
    {
        return message;
```

DOCUMENT INFO

TAGS

RELATED

COMMENTS

HISTORY

```
    }
}
```

**deaddrop.sol**

```
**********Solidty code ends above!**************************

go to the compile tab in browser solidity and click on details
scroll down to find web3deploy - copy this into a text editor for any final changes...
ex.

**********Web3 deployment code for the previous contract begins below!***************
*************

var data = /* var of type string here DO NOT FORGET*/ ;

var browser_deaddrop_sol_deaddropContract = web3.eth.contract([{"constant":true,"inputs":[],"nam
e":"getMessage","outputs":[{"name":"","type":"string"}],"payable":false,"stateMutability":"view"
,"type":"function"},{"inputs":[{"name":"data","type":"string"}],"payable":false,"stateMutability
":"nonpayable","type":"constructor"}]);
var browser_deaddrop_sol_deaddrop = browser_deaddrop_sol_deaddropContract.new(
   data,
   {
     from: web3.eth.accounts[0],
     data: '0x6060604052341561000f57600080fd5b6040516102b83803806102b8833981016040528080518201919
0505080600090805190602001906100419291906100485656b50506100ed565b828054600181600116156101000203160
0029004906000526020600020906001f016020900481019282601f106100895780516001ff191683800011785556100b756
5b828001600101855582156100b7579182015b828111156100b657825182559160200191906001019061009b565b5b50
90506100c491906100c8565b5090565b6100ea91905b808211156100e6576000816000905550600101610100ce565b5090
565b90565b6101bc806100fc6000396000f300606060405260043610610041576000357c0100000000000000000000000
0000000000000000000000000000000000000900463ffffffff168063ce6d41de14610046575b600080fd5b341561005157
600080fd5b6100596100d4565b60405180806020018281038252838181518152602001915080519060200190808383600
005b838110156100995780820151818401526020810190506100e565b505050509050908101906001f1680156100c657
808203805160018360200361010000a031916815260200191505b509250505060405180910390f35b6100dc61017c565b
6000805460018160011615610100020316600290048015610172578060 1f1061014757610100808354040283
529160200191610172565b820191906000526020600020905b8154815290600101906020018083116101555782900360
1f168201915b505050505090509056 5b6020604051908101604052806000815250905600a165627a7a7230582003c106
0e51911b1598cab3adbd622a571ae4ad1f71b13ece6ce33714f119153b0029',
     gas: '4700000'
   }, function (e, contract){
    console.log(e, contract);
    if (typeof contract.address !== 'undefined') {
         console.log('Contract mined! address: ' + contract.address + ' transactionHash: ' + con
tract.transactionHash);
    }
 })

**********Web3 deployment code for the previous contract ends above!*****************
***********

once you paste the above into geth it will return something like:

Contract mined! address: 0xad6aa5ba5e43c93b41535edf9316e981051963e0 transactionHash: 0x591ac00d5
83cc7e453ffd493da2d8130f3f49c86c5d918f3643bb61651e1dd6e

save the address for later:

contractAddr = "0xad6aa5ba5e43c93b41535edf9316e981051963e0"

Then later you can use the above address:

var deployedContract = eth.getCode(contractAddress)
```

```
#get a reference to your contract

abi = browser_deaddrop_sol_deaddropContract.at(contractAddr)


after this, you can call functions of your contract by typing:

abi.getMessage.call()


reply:

"hodl ETH (this is not fiancial advice)"
```

## TYPES

There are value types, reference types and mappings. Nulls do not exist in Solidity. When you fetch a value that has not been set, you get the default value of its type. If a type is a composite one, like a struct, then its default value is one where all components are themselves the default value.

### VALUE TYPES

bool, int, uint, address.

int and uint have the usual operators and are by default 256 bits long. But you can choose multiple-of-8 lengths, like uint8, int16 and so on. Compare this with Java's int which is 32 bits long, and Java's long which is 64 bits long.

address is 320 bits long, and has the extra:

- .balance
- .send(uint) to send wei, which may also execute the fallback function if the recipient is a contract.
- .call(...) and delegatecall(...) to call the fallback function or a function by name, as a last resort.

It is possible to let the system decide at execution what the integers are. For instance:

```
var sub = 1 - 2; // sub = -1, will be an int8
var prod = (0xffffffffffffffffff * 0xffffffffffffffffff) * 0; // prod = 0, will be a uint8
```

When the addition of 2 large numbers yield a number too large, it wraps around. A standard way to safely check such a wrap-around is to test:

```
var uint arg1, arg2;
if (arg1 + arg2 >= arg1) {
    ...
}
```

There also is enum, which is another way to look at uint. If you have 256, or fewer, values in your enum it will be equivalent to a uint8, if you have 65,536, or fewer, values ... uint16 ...

```
enum Hand { Left, Right }
Hand myHand;
function getMyHand() returns (uint) { // In fact will be a uint8
    return uint(myHand);
}
function setMyHand(uint newHand) returns (bool) {
    myHand = Hand(newHand);
}
```

### ARRAY VALUE TYPES

Fixed sizes: bytes1 to bytes32.

Dynamic sizes: bytes, string encoded in UTF8.

Note that:

- these arrays are returned by value, which costs gas.
- dynamic sizes may only be returned by constant functions. The rationale is that it would make gas expenditure too random, or at least difficult to pre-estimate.

### REFERENCE TYPES

These structures are passed by reference.
Struct

struct lets you aggregate data into a logical entity. For instance:

```
struct Client {
    uint id;
    string name;
}
```

Array

Arrays are available and behave as you expect them to. For instance:

```
uint[] clientIds;
Client[] clientInfos;
```

- To read or write elements at an existing zero-based index, you call clientIds[2]
- To add an element at the end, and thereby increase the length, you call clientIds.push(123)
- To truncate an array to a given length, you call clientIds.length = 2

Data location

You have been told that the blockchain merges the database with the network. Because of this, reference variables are also assigned a location, such as memory or storage.

It is not necessary to always specify the location as there are default locations depending on where the declaration is. However, it comes in handy when you want to avoid expensive copies between memory and storage. For instance:

```
contract Staff {
    uint[] peopleIds; // defaults to storage

    function test() {
        processIds(peopleIds); // expensive
        processIds2(peopleIds); // not expensive
    }

    function processIds(uint[] ids) { // defaults to memory
    }

    function processIds2(uint[] storage ids) { // forced to storage
    }
}
```

MAPPING

A mapping associates a value behind a key. Think of them as a map or hashtable. A big difference is that every value is "already populated" with zeroes, and there is no .contains(key) method. For instance:

```
mapping (address => uint) contributions; // contributions[whatever] is already set at 0
mapping (uint => Client) clientInfos;
```

## STATE VARIABLES

Each contract instance can maintain a state. This state consists of one of more variables of the types defined above. These state variables can only be modified via a function call invoked within a transaction.

VISIBILITY SPECIFIERS

These variables are by default internal but can be declared with these visibility identifiers:

- private: only the contract it is declared in can see it
- internal: only the contract it is declared in and its child contracts can see it, this is the default
- public: all can access it

READ/WRITE ACCESS

When a contract and/or its children have access to a state variable, they access it by its name to read and write:

```
contract Owned {
    address public owner;

    function Owned() {
        owner = msg.sender;
    }
}

contract Purse is Owned {
    function changeOwner(address newOwner) {
        if (msg.sender == owner) {
            owner = newOwner;
        }
    }
```

```
}
```

However, when a state variable is public and accessed outside of its contract and its children, it is invoked as a function. This syntax has the added benefit of expliciting the fact that the variable is thus only readable.

```
contract Purse {
    uint public limit;
}

contract PurseManager {
    address purse;

    function getLimit() returns (uint) {
        return Purse(purse).limit();
    }
}
```

Note however that, for state variables whose size is dynamic, the external access method takes an input:

- a uint when the state variable is a dynamic array
- a key type when the state variable is a mapping