

Solidity/Ethereum development quick guide

Ákos Hajdu, Imre Kocsis, Péter Garamvölgyi

This is a supplementary guide for the [Blockchain Technologies and Applications \(VIMIAV17\)](#) course at the [Budapest University of Technology and Economics](#).

Ethereum basics

[Ethereum](#) is a generic blockchain-based computing platform where nodes in a peer-to-peer network are maintaining the ledger. Nodes run the *Ethereum Virtual Machine (EVM)* and execute *transactions* and *contracts* compiled into EVM bytecode. The main entities on the network are the *accounts*, identified by their 160-bit addresses. There are two kinds of accounts.

- An *externally owned account* is associated with a balance in Ether, the native cryptocurrency of Ethereum. It is typically owned by human users.
- A *contract account*, in addition to its balance also stores the compiled contract bytecode and the data (state) associated with the contract.

Contracts are usually written in a high-level language (such as [Solidity](#)) and then compiled into EVM bytecode. A compiled contract can be deployed to the blockchain by a special transaction. From that point on, users or other contracts can interact with the deployed contract by issuing *transactions* to its address. The transaction contains the function to be called (with its parameters) and an execution fee called *gas*. Optionally, some value of Ether can also be associated with the call. The nodes then execute the transaction by running the contract code. Each instruction costs some predefined amount of gas. If execution runs out of gas or there is a runtime error, the whole transaction is reverted. Otherwise, successful transactions will be included in some of the next blocks as part of the mining process.

For more information and details, you can read the [How does Ethereum work, anyway? article](#) or the [Ethereum yellow paper](#).

Solidity basics

Solidity is a rapidly evolving language, but it has a well written and extensive [documentation](#). The current guide is based on version 0.5.0 (released on November 13, 2018), but the latest documentation is available at [solidity.readthedocs.io/en/latest](#).

Layout of source files

An example Solidity smart contract can be seen below. The contract implements a simple bank, where users can deposit and withdraw money (Ether). Furthermore, the bank also counts the number of transactions.

Source files start with a *pragma* statement describing the *compiler version* to use. A source file can then define multiple *contracts*. The example below has a single contract named `SimpleBank`. At a first glance, smart contracts are similar to simple classes in object-oriented programming. Contracts can define

- *state variables*, which define the data stored on the blockchain and

- *functions* that can manipulate the data and interact with other accounts.

It is also possible to *import* other contracts from other source files and Solidity also supports *inheritance* between contracts.

```
pragma solidity ^0.5.0;

contract SimpleBank {
    uint public transactions;
    mapping(address=>uint) balances;

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

    function getBalance() public view returns (uint) {
        return balances[msg.sender];
    }

    function withdraw(uint amount) public {
        require(balances[msg.sender] >= amount);
        balances[msg.sender] -= amount;
        msg.sender.transfer(amount);
        transactions++;
    }
}
```

State variables

The SimpleBank example defines two state variables.

- `transactions` is an unsigned integer (`uint`), storing the number of transactions in the current bank instance. It is incremented when users deposit or withdraw.
- `balances` is a *mapping* from addresses to integers, storing the current balance of each user. Mappings work similarly to maps in Java/C++ or dictionaries in C#/Python.

Types. Solidity is a strongly-typed language, i.e., the type of each variable must be explicitly specified. Solidity offers various *primitive types*, including:

- Booleans (`true/false`).
- Signed and unsigned integers of various bit-lengths: `int8`, `int16`, `int24`, ..., `int256` are signed and `uint8`, `uint16`, `uint24`, ..., `uint256` are unsigned integers with 8, 16, 24, ..., 256 bits. The default `int` and `uint` types correspond to `int256` and `uint256` respectively.
- Addresses (`address`) are 160-bit integers corresponding to Ethereum addresses.

Solidity also provides *reference types*, including:

- Arrays (with fixed or dynamic size).
- Structures (`struct`).
- Mappings (`mapping(...=>...)`).

These types work in a similar way as in other programming languages. For more information on types, see [types section of the documentation](#).

Visibility. State variable *visibility* can be `private`, `internal` or `public`, which are similar to other programming languages. However, there are a few remarkable differences.

- For public variables, only a getter function is generated automatically. They cannot be directly written by other contracts or transactions.
- Although private (and internal) variables cannot be accessed and modified by other contracts, transactions on the blockchain are public, so the information stored in such variables is still visible to anyone. Never store passwords or other secret information on the blockchain.
- If no visibility is specified, `internal` is the default.

Functions

[Functions](#) in Solidity can read and manipulate the state variables and interact with other contracts and accounts. Functions can have parameters (e.g., `withdraw` in the example has one parameter) and can return values (e.g., `getBalance` returns one). The `SimpleBank` example specifies three functions.

- The `deposit` function is `public` (can be called by anyone) and `payable`, which means that it can receive Ether as part of the call. Functions can access a special `msg` field, which stores information about the function call. For example, the `deposit` function reads the amount of Ether associated with the call from the `msg.value` field and increases the balance of the caller, whose address is stored in `msg.sender`. It can also be seen that the dictionary can be accessed using square brackets (`balances[msg.sender]`). Finally, the function increments the `transactions` counter.
- The function `getBalance` is marked as `view`, which means that it does not modify the state of the contract, but it can still read it. The `pure` keyword (instead of `view`) is even more restrictive: such functions cannot read or write the state. The function also specifies a return value with `returns (...)`. This function gets and returns the balance of the caller (`msg.sender`).
- The function `withdraw` specifies a single `uint` parameter called `amount`, corresponding to the amount that the caller wants to withdraw from the bank. The function first checks whether the caller has enough Ether in the bank using a `require` statement. The `require` statement checks the condition and reverts the whole transaction if the condition is false. Otherwise it updates the mapping by decreasing the balance of the caller and then transfers the required amount using the `transfer` function. Finally, the function increments the `transactions` counter.

Besides the basic statements illustrated by the `SimpleBank` example, functions can have [other statements](#) such as selection (`if else`) or loops (`for`, `while`). However, as execution costs a transaction fee per instruction (gas), it is recommended to avoid complex operations like loops when possible. Furthermore, instructions writing the blockchain state are more expensive to execute, therefore it is also recommended to minimize the number of writes.

Special variables and functions. As already mentioned in the example, functions can access a special `msg` field, which stores information about the function call. Besides `msg`, functions can also access some other special variables and functions, including for example the parameters of the current transaction and block, the current timestamp or the remaining gas. For more information, see the [documentation](#). However, use these special variables and functions with caution as they may introduce vulnerabilities to your contract. For example, `tx.origin` [should never be used for authorization](#), use `msg.sender` instead.

Visibility. Functions must be marked with a `public`, `internal`, `private` or `external` visibility. For more information, see the [visibility section of the documentation](#). In previous versions of Solidity, if a function did not specify a visibility it was public by default. However, this led to vulnerabilities and in the current version the visibility must be specified.

Constructor. State variables are initialized to their default values (e.g., 0 for integer types or an empty mapping). However, an explicit [constructor](#) can be provided (including parameters) with the `constructor` keyword. For example, one could write a constructor for the `SimpleBank` example, which starts the transaction counter from a given parameter.

```
contract SimpleBank {
    // ...
    constructor(uint start) public {
```

```

        transactions = start;
    }
}

```

Handling Ether

Each `address` (contract or external) is associated with a balance in Ether. Solidity provides various language features to query balances and transfer Ether. The `address` type has a field `balance` which can query the balance. For example, to query the balance of the current contract inside a function we can use `address(this).balance`:

```

function getContractBalance() public returns (uint) {
    return address(this).balance;
}

```

There is another flavour of the `address` type called `address payable`, which is a special address that can receive Ether (similarly to `payable` functions). For example, `msg.sender` in a function has the `address payable` type. A payable address has two functions to transfer Ether: `transfer` and `send`. The difference between the two is that in case of a failure, `transfer` throws an exception while `send` indicates it with a false return value. In the SimpleBank example, we use `transfer` in the `withdraw` function because if it fails, the exception is propagated and the whole transaction is reverted (including the instruction that decreased the balance of the caller). It is a common programming error to use `send` without checking its return value.

As already seen in the SimpleBank example, functions can be marked with the `payable` keyword, allowing the caller to attach Ether to the call. The Ether attached is automatically added to the balance of the contract, but can also be queried from the `msg.value` field. When a contract wants to call another contract and send Ether, it can set the amount with the `value` function. For example, if we have a `SimpleBank sb` field in another contract, we can call `sb.deposit.value(amount)()` to deposit a given amount.

Each contract can have at most one function *without a name*, which is called the *fallback function*. This function gets executed when a call to the contract matches no other function. Furthermore it is also executed when `transfer` or `send` is used to transfer Ether to a contract. However, this requires the fallback function to be marked as `payable`. An example fallback function can be seen below.

```

function () public payable {
    // Do something
}

```

There is no distinct type for Ether, unsigned integers are used. The default unit is Wei, but literals can be specified using suffixes such as `wei`, `finney`, `szabo` or `ether`. For example, `uint amount = 1 ether;` will store 10^{18} in the variable `amount`.

Error handling

Transactions in Ethereum work in an atomic way: if there is an error, the whole transaction gets reverted. Errors can happen due to some condition in the execution, such as running out of the execution fee or indexing an array out of bounds. However, there are multiple ways for the programmer to [raise an error](#).

- `require(...)` checks if a condition holds and if not, it reverts the transaction. It is recommended to be used for example to validate parameters at the beginning of the function.
- `assert(...)` is similar to `require` in its effect, but it is recommended to use for checking conditions that should not fail. Proper code should never reach an assertion failure. In contrast, it is normal for `require` to raise an error.
- `revert()` simply reverts the transaction.

When functions call other functions, the errors propagate up, making the whole chain of calls revert. However, there are a few exceptions from this rule: `send`, `call`, `delegatecall` and `staticcall` only indicate the error in their return value. These functions should be used with caution.

Further reading

Solidity supports some other language elements that were not discussed here, including [function modifiers](#), [events](#), [inheritance](#), [interfaces](#) and [libraries](#). For more information, please refer to the [documentation](#).

Detailed tutorials are also available: [greeter](#), [token](#), [crowdsale](#), [dao](#). Furthermore, [Ethernaut](#) and [Cryptozombies](#) are interactive tutorials.

For the latest news, read the [Ethereum blog](#).

While it may seem easy to develop Solidity code, it is also easy to make mistakes. Since the Ethereum blockchain is public, anyone can exploit such bugs, causing serious financial damages (e.g., [reentrancy in the DAO](#) or [overflow in the BECToken](#)). Furthermore, the blockchain is permanent so buggy contracts cannot be patched once deployed. It is therefore highly recommended to read about [security considerations](#), [common attacks](#) and [best practices](#). There is also a handful of tools targeting the verification of contracts, including [Truffle](#), [Securify](#), [Oyente](#), [Maian](#), [MythX](#), [Slither](#), [solc-verify](#) and [VeriSolid](#).

Development and testing

As with other programming languages, development requires some tools. [Visual Studio Code](#) has a Solidity extension to edit Solidity contracts. The official compiler is [Solc](#), which can translate the contract into EVM bytecode. Alternatively, [Remix](#) can also be used, which is a web-based IDE supporting editing, compiling and testing. In order to deploy a contract to a real network one also needs a wallet such as [Metamask](#) or [Mist](#). For more complex contracts and scenarios, one should also consider the [Truffle Suite](#), providing various development tools. In the following, we cover the basics of [Remix](#). For more information, please refer to the [documentation of Remix](#).

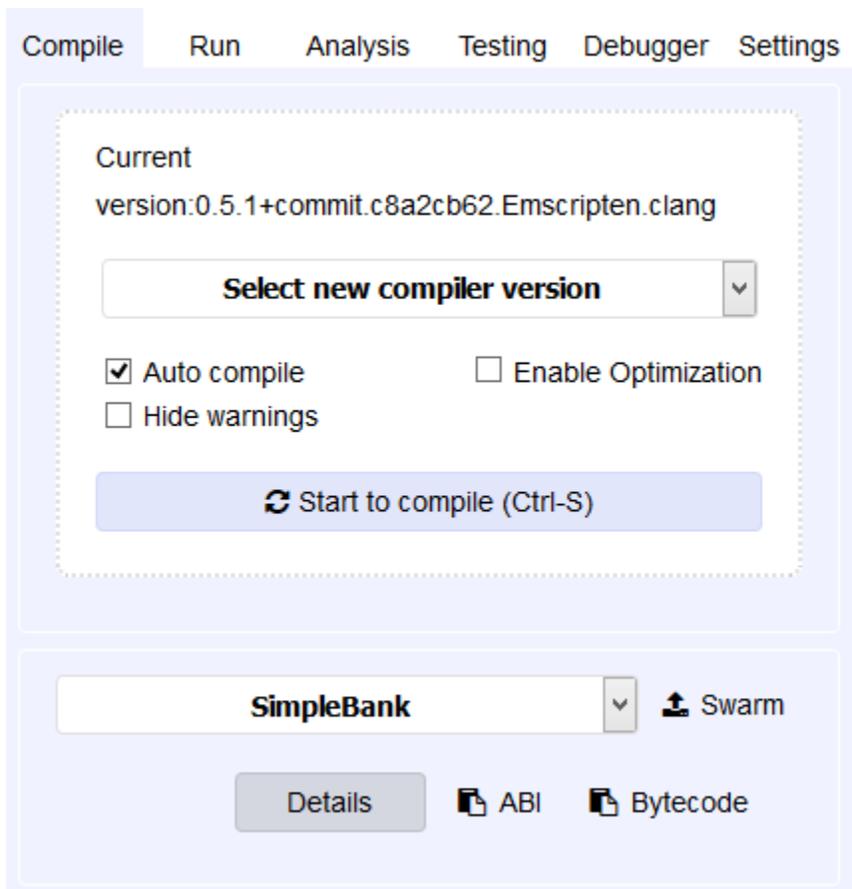
Writing a contract

You can create or import files in Remix with the icons on the top left. As an example, create a new file named `SimpleBank.sol` and copy the code from the SimpleBank example above. Remix supports syntax highlighting and auto completion as well.

```
1 pragma solidity ^0.5.0;
2
3 contract SimpleBank {
4     uint public transactions;
5     mapping(address=>uint) balances;
6
7     function deposit() public payable {
8         balances[msg.sender] += msg.value;
9         transactions++;
10    }
11
12    function getBalance() public view returns (uint) {
13        return balances[msg.sender];
14    }
15
16    function withdraw(uint amount) public {
17        require(balances[msg.sender] >= amount);
18        balances[msg.sender] -= amount;
19        msg.sender.transfer(amount);
20        transactions++;
21    }
22 }
```

Compiling a contract

The contract can be compiled by selecting the *Compile* tab on the top right.



You can select a compiler version and click *Start to compile* or you can also turn on the *Auto compile* option. On a successful compilation, you will receive a green message with the name of the contract.

Testing a contract

For a simple local test, select the *Run* tab on the top right. The default *environment* is JavaScript VM, which runs an Ethereum Virtual Machine (EVM) locally (i.e., it does not connect to the real network). This makes testing quick and free. Remix also supports connecting to a real network by selecting the Web3 option as an environment (which requires a wallet first). Besides the main network, there are also some test networks ([Ropsten](#), [RinkeBy](#), [Kovan](#)). These networks are somewhat similar to the main network as there are multiple nodes and miners. However, the Ether on these networks has no real value: you can also request some free Ether to test your contracts.

For the purpose of this tutorial we will be simply working with the local JavaScript VM environment. Remix also creates some test *accounts* by default with 100 Ether each. The account currently selected is used as the sender when you issue transactions. You can specify the *gas limit* and you can also attach ether to the call with the *value* field.

Compile **Run** Analysis Testing Debugger Settings

Environment JavaScript VM VM (-) ▼

Account + 0xca3...a733c (99.999999999999757) ▼

Gas limit 3000000

Value 0 wei ▼

SimpleBank ▼ i

Deploy

or

At Address Load contract from Address

Transactions recorded: ① ▼

Deployed Contracts 🗑

▶ SimpleBank at 0x692...77b3a (memory) 📄 ✕

The first step is to deploy the contract. This can be done by selecting the contract name from the drop down menu and then clicking *Deploy*. The contract should appear below in the list of deployed contracts. You can also see that the balance of the selected account decreased a bit. This is because deploying a contract is a special transaction that also costs execution fee (gas).

If you click on a deployed contract, its public interface appears. Our example, `SimpleBank` has 4 functions. You can see that `view` functions are marked with a different color. These do not result in transactions and therefore have no execution fee. Click on a function to execute it. If a function has parameters (such as `withdraw`), you can specify its parameters in the textboxes next to the name of the function. After executing the function, the return value (if any) appears below.

Deployed Contracts 🗑️

▼ SimpleBank at 0x692...77b3a (memory) 📄 ✕

deposit

withdraw ▼

getBalance

transactions

You can also inspect the details of the transactions in the middle below the editor. For example, you can see the deployment transaction below.

[2] only remix transactions, scr... ▼ 🔍 Search

creation of SimpleBank pending...

✓ [vm] from:0xca3...a733c to:SimpleBank.(constructor)
 value:0 wei data:0x608...50029 logs:0
 hash:0xe2f...4d79b Debug

status	0x1 Transaction mined and execution succeed
transaction hash	0xe2f10a1a84e5fb670c21e6c804c85a6ec42c9fa15138d7488e26f1292c14d79b 📄
contract address	0x692a70d2e424a56d2c6c27aa97d1a86395877b3a 📄
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c 📄
to	SimpleBank.(constructor) 📄
gas	3000000 gas 📄
transaction cost	242250 gas 📄
execution cost	143190 gas 📄
hash	0xe2f10a1a84e5fb670c21e6c804c85a6ec42c9fa15138d7488e26f1292c14d79b 📄

You can now try playing around with SimpleBank. For example, you can set the *value* to 10 Ether and call `deposit`. Then `getBalance` should return 1000000000000000000 (Wei). Calling `transactions` should return 1. If you switch to a different account and call `getBalance` you should see 0. If you try to withdraw, the transaction should fail due to the `require`. Switching back to the previous account, `deposit` should

work (for no more than 10 Ether).

If you make modifications to the contract, don't forget to compile and deploy again!