Ethereum Smart Contracts

Chair of Software Engineering for Business Information Systems (sebis)
Department of Computer Science
School of Computation, Information and Technology (CIT)
Technical University of Munich (TUM)
wwwmatthes.in.tum.de

Outline

1. **Solidity Introduction**
   - Definition
   - Anatomy of a Smart Contract
   - Language Features
   - Data Storage
   - Functions
   - Modifiers
   - Inheritance
   - Abstract Contracts and Interfaces

2. **Designing Smart Contracts**
   - Problem Assessment
   - Modelling Entities
   - Modelling Transactions

3. **Cross-contract and Blockchain Interaction**
   - EVM Contract Function Execution
   - Transactions and Messages
   - Address Class
   - Re-entrancy Attack
   - Message Object
   - Block Object
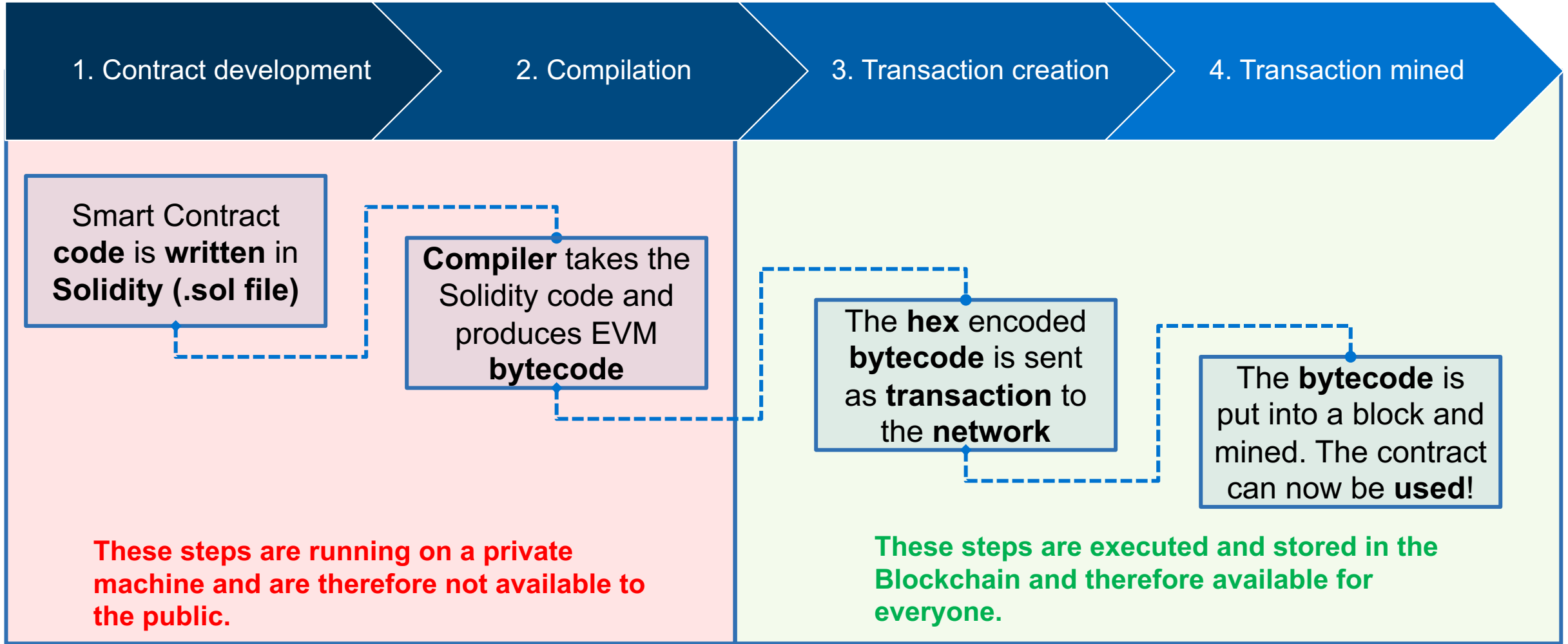   - Transaction Object and Calldata
   - Events

Definition

- Solidity is a **high-level language** to write **smart contracts** for **Ethereum.**

- **Contracts[1]** can be defined as **encapsulated units**, similar to classes in traditional **object-oriented** programming languages like Java.

- A **contract** has **its own, persistent state** on the blockchain which is **defined by state variables** in the contract.

- **Functions** are **used to change** the **state** of the contract or to **perform** other **computations.**

- Solidity is **compiled to bytecode** which is **persistent and immutable** once deployed to the blockchain:
  ➔ **No patch deployment** possible
  ➔ **Smart contracts** must be **perfect before** using them in **production!**

[1]"Contract" is used as a short form of "smart contract".

Source Code

- A smart contract source code is typically **not stored on the blockchain**, only the **bytecode** gets stored.
  - Remember that bytecode represents a series of **EVM opcodes** to be executed.

- Without further analysis, the purpose of a smart contract bytecode is unclear.

- Source code can be made publicly available.

- [Etherscan.io](#) is the most popular Ethereum block explorer, and a service which verifies source codes and the respective bytecode.

Bytecode

Decompile ByteCode ⧉ | Switch To Opcodes View

`</>` Contract Creation Code

```
6060604052604080519081016040528060d81526020017f5772617070656420457468657200000000000000000000000000000000000081525060009080519060200190610041004f9291906100c8565b5060408
0519081016040528060481526020017f57455448000000000000000000000000000000000000000000000000000081525060019080519060200190619061009b9291906100c8565b506012600260006101000a
81548160ff021916908360ff16021790550534156100c357600080fd5b61016d565b828054600181600116156101000203166002900490600052602060002090601f016020900481019282601f1061010957805
160ff1916838001178556561013756582800160010185558216100137579182015b82811115610136578251825591602001919060010190610135565b5090565b61016100148565b50905565b61016019b5b6
5b808211156101661667600081600090555060010161601014e565b5090565b90565b610c348061017c6000396000f3006060604052600436106100af576000357c010000000000000000000000000000000000000
0000000000000000000900463ffffffff16806306fdde03146100b95780637063095ea7b31461014757806318160ddd146101a157806323b872dd146101c15780632e1a7d4d146102435780633313ce565780
6370a08231146102955780639557806395d89b41146102e2578063a9059cbb14610370578063d0e30db01461603ca578063dd62ed3e146103d4575b6100b7610440565b005b341561c457600080fd5b6100c6104dd565
```

Opcode

Decompile ByteCode ⧉ | Switch Back To Bytecodes View

`</>` Contract Creation Code

```
PUSH1 0x60
PUSH1 0x40
MSTORE
PUSH1 0x04
CALLDATASIZE
LT
PUSH2 0x00af
```

Source Code

Code | Read Contract | Write Contract

? | Search Source Code | ⌄ ⌃

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

| Contract Name: | WETH9 | Optimization Enabled: | **No** with **200** runs |
| Compiler Version | v0.4.19+commit.c4cbbb05 | Other Settings: | **default** evmVersion |

📄 **Contract Source Code** (Solidity)

Outline ⌄ | More Options ⌄

```
1   /**
2    *Submitted for verification at Etherscan.io on 2017-12-12
3    */
4
5   // Copyright (C) 2015, 2016, 2017 Dapphub
6
7   // This program is free software: you can redistribute it and/or modify
8   // it under the terms of the GNU General Public License as published by
9   // the Free Software Foundation, either version 3 of the License, or
10  // (at your option) any later version.
11
12  // This program is distributed in the hope that it will be useful,
13  // but WITHOUT ANY WARRANTY; without even the implied warranty of
14  // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
15  // GNU General Public License for more details.
16
17  // You should have received a copy of the GNU General Public License
18  // along with this program.  If not, see <http://www.gnu.org/licenses/>.
19
20  pragma solidity ^0.4.18;
21
22  contract WETH9 {
23      string public name     = "Wrapped Ether";
24      string public symbol   = "WETH";
25      uint8  public decimals = 18;
```

See this smart contract on Etherscan: https://etherscan.io/token/0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2#code

From Solidity Source Code to a Deployed Smart Contract

| 1. Contract development | 2. Compilation | 3. Transaction creation | 4. Transaction mined |

Smart Contract **code** is **written** in **Solidity (.sol file)**

Compiler takes the Solidity code and produces EVM **bytecode**

The **hex** encoded **bytecode** is sent as **transaction** to the **network**

The **bytecode** is put into a block and mined. The contract can now be **used**!

These steps are running on a private machine and are therefore not available to the public.

These steps are executed and stored in the Blockchain and therefore available for everyone.

Anatomy of a Solidity Smart Contract File

File: **BBSE.sol**

```solidity
contract BBSE {

    struct Tutor {
            string firstName;
            string lastName;
    }
    mapping (address => Tutor) tutors;
    address professor;

    modifier onlyProfessor {
            require(msg.sender == professor);
            _;
    }

    constructor() public {
            professor = msg.sender;
    }

    function getProfessor() view returns (address) {
            return professor;
    }

    // This function adds a new tutor
    function addTutor(address tutorAddress,
    string firstName, string lastName) onlyProfessor {
            Tutor tutor = tutors[tutorAddress];
            tutor.firstName = firstName;
            tutor.lastName = lastName;
    }

}
```

State variables
- State variables are permanently stored in the contract's storage.
- Changing the state requires a transaction and therefore costs ether.
- Reading the state of a contract is free and does not require a transaction.

07 Ethereum Smart Contracts - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2023). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.

CC BY-SA 4.0   6

Anatomy of a Solidity Smart Contract File (cont.)

File: **BBSE.sol**

```solidity
contract BBSE {

        struct Tutor {
                string firstName;
                string lastName;
        }
        mapping (address => Tutor) tutors;
        address professor;

        modifier onlyProfessor {
                require(msg.sender == professor);
                _;
        }

        constructor() public {
                professor = msg.sender;
        }

        function getProfessor() view returns (address) {
                return professor;
        }

        // This function adds a new tutor
        function addTutor(address tutorAddress,
        string firstName, string lastName) onlyProfessor {
                Tutor tutor = tutors[tutorAddress];
                tutor.firstName = firstName;
                tutor.lastName = lastName;
        }

}
```

Function modifiers
- Function modifiers are a convenient way to reuse pieces of code.
- Changes the behavior of a function.
- Can execute code either before and/or after the actual function execution.
- The low dash _ indicates where the actual function code is injected.
- Often used for authentication.

File: **BBSE.sol**

```solidity
contract BBSE {

        struct Tutor {
                string firstName;
                string lastName;
        }
        mapping (address => Tutor) tutors;
        address professor;

        modifier onlyProfessor {
                require(msg.sender == professor);
                _;
        }

        constructor() public {
                professor = msg.sender;
        }

        function getProfessor() view returns (address) {
                return professor;
        }

        // This function adds a new tutor
        function addTutor(address tutorAddress,
        string firstName, string lastName) onlyProfessor {
                Tutor tutor = tutors[tutorAddress];
                tutor.firstName = firstName;
                tutor.lastName = lastName;
        }

}
```

Constructor
- The constructor function is executed once when the contract is created through a transaction.
- The function cannot be called after the creation of the contract.
- Usually used to initialize the state of a contract.
- Execution costs gas, and more complex constructors lead to higher deployment costs.

07 Ethereum Smart Contracts - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2023). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.

CC BY-SA 4.0   8

Anatomy of a Solidity Smart Contract File (cont.)

File: **BBSE.sol**

```solidity
contract BBSE {

        struct Tutor {
                string firstName;
                string lastName;
        }
        mapping (address => Tutor) tutors;
        address professor;

        modifier onlyProfessor {
                require(msg.sender == professor);
                _;
        }

        constructor() public {
                professor = msg.sender;
        }

        function getProfessor() view returns (address) {
                return professor;
        }

        // This function adds a new tutor
        function addTutor(address tutorAddress,
        string firstName, string lastName) onlyProfessor {
                Tutor tutor = tutors[tutorAddress];
                tutor.firstName = firstName;
                tutor.lastName = lastName;

        }

}
```

Functions
- Functions are used to change the state of a contract.
- Can also be used to read the state of the contract.
- Consist of a name, a signature, a visibility, a type, a list of modifiers, and a return type.

Formal definition:
function (<parameter types>)
{internal|external|public|private}
[pure|constant|view|payable]
[(modifiers)]
[returns (<return types>)]

Language Features Overview

Solidity is **inspired by JavaScript** and comes with a very similar syntax. Furthermore, it implements the standard set of features for high-level (object-oriented) programming languages. Compared to the dynamically-typed JavaScript, Solidity uses static types.

Built-in data types
int, uint, bool, address, array, struct, enum, mapping

Built-in first level objects
block, msg, tx

Built-in functions
Error handling: assert(), require(), revert()
Math & Crypto: addmod(), mulmod(), sha3(), keccak256(), sha256(), ripemd160(), ecrecover()
Information: gasleft(), blockhash()
Contract related: selfdestruct()

A set of literals
Solidity comes with some Ethereum specific literals (like eth for units, e.g., int a = 5 eth)

Flow control
if, else, do, while, break, continue, for, return, ? ... : ... (ternary operator)

Function and Variable Visibility

In Solidity, functions can be declared with four different visibility types.

External
External methods can be called by other contracts and via transactions issued by a certain wallet. Methods declared as external **are always publicly visible** and **can't be called directly by the contract itself**.

Public
Public can **be called internally** by the contract itself but also **externally** by other contracts and via transactions. **State variables** which are defined as public will, **by default**, **have getter** method created automatically by the compiler.

Internal
Internal methods can only be accessed by the contract itself or by any contract derived from it. They are not callable from other contracts or via transactions.

Private
Private methods can **only** be called **internally** by the contract that owns the method. **Derived contracts cannot access** a private method of their parent contract.

Data Storage in EVM

EVM can store data in three different places; *storage*, *memory*, and the *stack*.

Storage	Memory	Stack
▪ The storage is comparable to a **hard drive**. It keeps data between function calls and is **persistent** for each smart contract. This way, every execution on the contract has access to the data previously saved in the storage area.	▪ The memory is comparable to a computer's **RAM**. It is a **temporary** storage location for data. Once the execution is complete, the memory is cleaned for the next execution.	▪ As EVM is a stack machine rather than a register machine, all computations are done on a data region called the stack. It has a maximum of 1024 elements and comprises 256-bit words.

Variables

i. *State variables* are saved in **storage** by default.

ii. *Value-type local variables* and *function arguments* (int, bool, address, …) are stored in **memory**.

iii. *Reference-type local variables* (array, struct, string, …) can be stored in **memory** or **storage**.

iv. *Reference-type function arguments* can be stored in **memory** or **calldata** (a **read-only data location** similar to memory but only available to function arguments).

Note

▪ Reading from the storage is **expensive**. Initializing and **modifying** storage is even more expensive. Thus, you should limit the data you keep in persistent storage to what the contract requires.

▪ Compared to the gas consumption of storage, memory has a relatively **low consumption**. Hence, it is cost-efficient to do intermediate calculations in memory and then save the result to storage if necessary.

See Solidity documents for detailed information: https://docs.soliditylang.org/en/develop/introduction-to-smart-contracts.html#storage-memory-and-the-stack

07 Ethereum Smart Contracts - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2023). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.

CC BY-SA 4.0   12

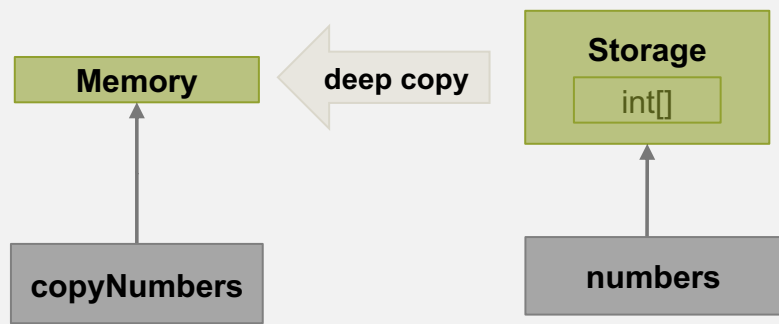# An Example Use of Storage and Memory

Storage

```solidity
1   pragma solidity >=0.7.0 <0.9.0;
2
3   contract ExampleStorage {
4
5       // Initializing an integer array 'numbers'
6       // Stored in 'storage' as it is a state variable
7       int[] public numbers;
8
9       // A function to insert values into the 'numbers' array and modify them
10      function insertAndModify() public {
11          numbers.push(1);
12          numbers.push(2);
13
14          // Creating a pointer to the storage location of 'numbers'
15          int[] storage copyNumbers = numbers;
16
17          // Assign a new value to index 0 of the new instance
18          copyNumbers[0] = 5;
19      }
20  }
```

Output

numbers	0
0: int256: 5	

numbers	1
0: int256: 2	

▪ *copyNumbers* is a **pointer** to the *numbers* array's **storage location** (like a **shallow copy**)
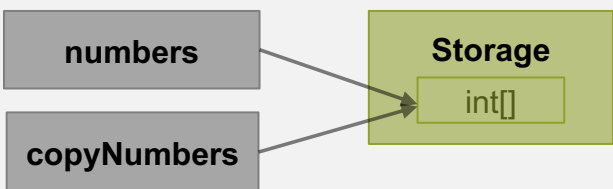
Memory

```solidity
1   pragma solidity >=0.7.0 <0.9.0;
2
3   contract ExampleMemory {
4
5       // Initializing an integer array 'numbers'
6       // Stored in 'storage' as it is a state variable
7       int[] public numbers;
8
9       // A function to insert values into the 'numbers' array and modify them
10      function insertAndModify() public {
11          numbers.push(1);
12          numbers.push(2);
13
14          // Creating a local copy of the 'numbers' array
15          int[] memory copyNumbers = numbers;
16
17          // Assign a new value to index 0 of the new instance
18          copyNumbers[0] = 5;
19      }
20  }
```

Output

numbers	0
0: int256: 1	

numbers	1
0: int256: 2	

▪ The keyword **memory** must be used in order to make sure that *copyNumbers* **only exists locally**

This example is inspired from: https://www.geeksforgeeks.org/storage-vs-memory-in-solidity/

07 Ethereum Smart Contracts - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2023). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.

CC BY-SA 4.0   13

Special Function Types

Solidity provides special function type declarations besides the default one.

View function
Functions that are declared as view are read-only, i.e., they don't modify any state variable nor alter the state of the blockchain. However, they can read from the state variables.

```
uint state = 5;
function add(uint a, uint b) public view returns (uint sum) { return a + b + state }
```

Pure function
Pure functions can be seen as a subset of view functions that don't modify the state but also don't read from the state.

```
function add(uint a, uint b) public pure returns (uint sum) { return a +b }
```

Fallback function
A contract can have one unnamed fallback function. The fallback function is called when no other function matches the function call (e.g., when Ether is sent to a contract which doesn't have a *receive*[1] function). A special feature of this function is that it can't have any parameters and doesn't return anything.

```
function() { /* ... */ }
```

[1] *Receive* is a special fallback function that enables a smart contract to receive Ether.

Special Function Types (cont.)

Payable function

By default, it is not possible to send Ether to a function, and the function will revert the transaction. This behaviour is intentional to prevent Ether that is accidentally sent from getting lost. However, sometimes it is necessary to pay a contract, e.g., in the case of an ICO. Therefore, Solidity implements *payable* functions.

Example

function buyInICO() public payable { /* … */ }

- The keyword payable is also required for declaring constructors and addresses that can receive Ether
  - constructor payable { /* … */ }, function withdraw (address payable _to) public { /* … */ }

- While implicit conversions are allowed from address payable to address, a casting function called payable (<address>) must be used for conversions from address to address payable.

```
address public customer;

function transfer (uint amount) public {
    payable(customer). transfer(amount);
}
```

07 Ethereum Smart Contracts - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2023). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.

CC BY-SA 4.0   15

Function Modifiers

Sometimes it is required to **check** whether a **specific condition** is true or false **before executing** a function. For instance, an authentication mechanism prior to the function call. However, writing code multiple times makes it harder to maintain and prone to security vulnerabilities. As a solution, Solidity offers the concept of **modifiers**, a **reusable piece of code**.

Modifiers are **defined** with the **keyword** modifier:

```solidity
contract owned {
  address public owner;

  constructor() public {
    owner = msg.sender;
  }

  modifier onlyOwner {
    require(msg.sender == owner);
    _;
  }

  function kill() public onlyOwner {
    selfdestruct(owner);
  }
}
```

Example of Function Modifiers

Internally, the actual function body is injected where _ is placed in the modifier.

The **code snippets** below **are equal**.

```
contract owned {
  address public owner;

  constructor() public {
    owner = msg.sender;
  }

  modifier onlyOwner {
    require(msg.sender == owner);
    _; // Injection here
  }

  function kill() public onlyOwner {
    selfdestruct(owner);
  }
}
```

===

```
contract owned {
  address public owner;

  constructor() public {
    owner = msg.sender;
  }

  function kill() public {
    require(msg.sender == owner);
    selfdestruct(owner);
  }
}
```

Chaining of Function Modifiers

It is **possible** to **apply multiple modifiers to a function**. The modifiers will be resolved sequentially, starting from left to right. In the example below, a user can only call the kill function if he/she is the owner of the contract and has an account balance greater than 1337 ETH.

```solidity
contract owned {
    address public owner;

    constructor() public {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _; // Actual function code is injected here
    }

    modifier isRich {
        require(msg.sender.balance > 1337 ether);
        _; // Actual function code is injected here
    }

    function kill() public onlyOwner isRich {
        selfdestruct(owner);
    }
}
```

Function Overloading

Solidity allows to overload functions, i.e., to define the same function twice with different signatures. This can be helpful if a function needs to be adapted to certain situations.

Example

```
function sendEther(uint amount) {
  require(this.balance >= amount);
  payable(msg.sender).transfer(amount);
}

function sendEther(uint amount, address payable  to) {
  require(this.balance >= amount);
  to.transfer(amount);
}
```

If sendEther() is called without the address argument, the Ether will be sent to the caller. Otherwise, it will be sent to the address passed as a parameter to the function.

07 Ethereum Smart Contracts - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2023). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.

CC BY-SA 4.0   19

Named Function Calls

Solidity supports the concept of **named calls. The named calls principle allows passing function parameters explicitly via a dictionary.** By default, function parameters are passed by their defined signature order.

Example 1, without a named call:

```
function myAddFunction(uint a, uint b) returns (uint result) {
    return a+b;
}

function fourPlusTwo() returns (uint result) {
    return myAddFunction(4, 2);
}
```

The order of the parameters is defined by the function's signature.

Example 2, using a named call:

```
function myAddFunction(uint a, uint b) returns (uint result) {
    return a+b;
}

function fourPlusTwo() returns (uint result) {
    return myAddFunction({b: 2, a:4});
}
```

The function fourPlusTwo passes a dictionary with keys that match the signature of myAddFunction. The order within the dictionary does not matter.

07 Ethereum Smart Contracts - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2023). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.

CC BY-SA 4.0   20

Inheritance

Solidity supports the inheritance of contracts. Technically, the solidity compiler copies the code from the parent contract **to the subcontract** and creates a single piece of bytecode which is deployed on the blockchain.

Solidity also **supports multiple inheritances** for a contract. In this case, the compiler just copies all parent contracts together and creates a single contract that is compiled to bytecode and deployed to the blockchain. Once a contract is deployed, it is not possible to detect from the bytecode whether a contract made use of inheritance or not.

If a parent contract contains a function that is also present in the subcontract, then the functions are overloaded. In case both functions have the same signature, the subcontract function will override the parent's function. However, the parent function can still be explicitly accessed using the `super` keyword.

Use Cases:
SafeMath, Authentication

Multiple Inheritance

Solidity uses, similar to Python, the **C3 superclass linearization** algorithm to define the order of the inherited functions. There is no implicit order for parent classes, the order is defined by the developer.

Assume the following inheritance graph

Y inherits from X

07 Ethereum Smart Contracts - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2023). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.

CC BY-SA 4.0   22
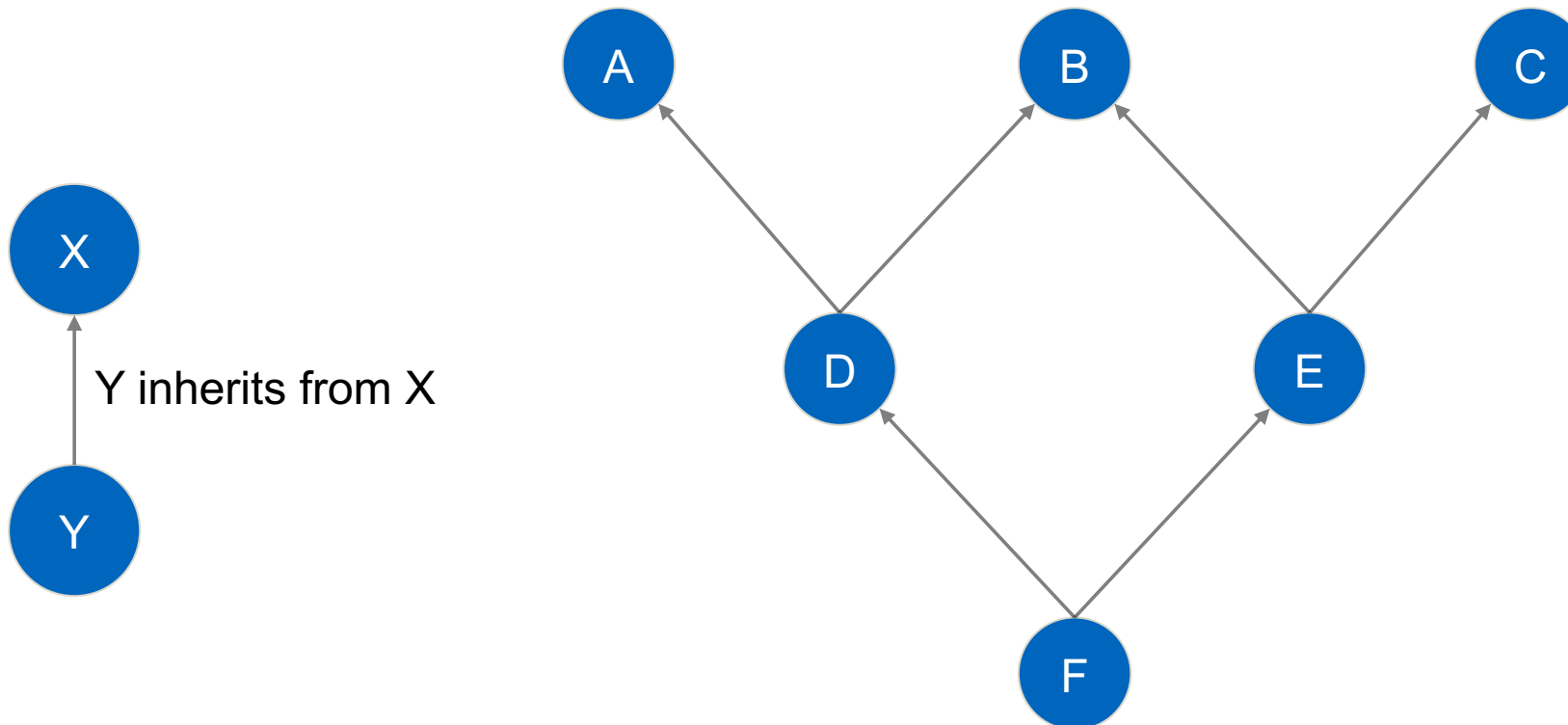
Multiple Inheritance (cont.)

In Solidity, contracts can inherit from other contracts by using the keyword **is.**

Example

contract A {}

contract B {}

contract C {}

contract D is A, B {}

contract E is B, C {}

contract F is D, E {}

The function resolution order (FRO) of the example above would be:

F, D, E, A, B, C

The keyword super always references the next contract in the FRO. If super would be called in F, it would reference to D and super in D would reference to E and so on.

07 Ethereum Smart Contracts - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2023). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.

CC BY-SA 4.0   23

Example of Inheritance

```
contract A {
 function getNumber() returns (uint a) {
  return 1337;
 }
}


contract B is A {
 function getNumber() returns (uint a) {
  return super.getNumber() + 1;
 }
}


contract C is A {
 function getNumber() returns (uint a) {
  return super.getNumber() + 2;
 }
}
contract Final is C, B {
 function getNumber() returns (uint a) {
  return super.getNumber();
 }
}
```

What would happen if Final.getNumber() is called?

- The function resolution order is:
  Final, C, B, A
  - In Final super will be resolved to C
  - In C super will point to B   ! ! ! !
  - In B super will point to A

- The final result is 1337 + 1 + 2 = 1340

Abstract Contracts

Solidity supports abstract contracts. A contract is **implicitly** declared as **abstract** if **one or more functions** are abstract. A function is considered abstract when it **does not have a body**.

Example

```
contract CarInsurance {
    function payMonthlyFee() returns (boolean result);
}
```

Abstract contracts cannot be compiled to bytecode. A **contract that inherits** from an abstract contract **must implement and override all methods** from the base contract to be compliable.

Abstract contracts offer a way to decouple the definition of a contract from its actual implementation. This provides better extensibility and maintainability, in particular for larger contracts.

07 Ethereum Smart Contracts - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2023). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.

CC BY-SA 4.0   25

Interfaces

Solidity supports the definition of **interfaces** for contracts. An interface is similar to an abstract contract but it is more restrictive. It is **not allowed** to define a **constructor**, **variables**, **structs**, **and enums** in an interface. Furthermore, interfaces **cannot inherit from a contract** or **implement another interface**.

Example

```
interface CarInsurance {
    function payMonthlyFee() returns (boolean result);
}
```

A **contract** can **implement multiple interfaces** at once.

Outline

1. Solidity Introduction
   - Definition
   - Anatomy of a Smart Contract
   - Language Features
   - Data Storage
   - Functions
   - Modifiers
   - Inheritance
   - Abstract Contracts and Interfaces

2. Designing Smart Contracts
   - Problem Assessment
   - Modelling Entities
   - Modelling Transactions

3. Cross-contract and Blockchain Interaction
   - EVM Contract Function Execution
   - Transactions and Messages
   - Address Class
   - Re-entrancy Attack
   - Message Object
   - Block Object
   - Transaction Object and Calldata
   - Events

07 Ethereum Smart Contracts - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2023). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.

CC BY-SA 4.0   27

Problem Assessment

The hype of blockchain has led to a large number of software applications using blockchain technology. Often used by startups or driven by innovation departments in enterprises.

However, the real-world adoption of contracts still is negligible. Currently, the largest dApp[1] using an Ethereum smart contract has less than ~11000 transactions per day (https://dappradar.com/rankings/protocol/ethereum).

Finding a use case

- Understand the problem domain and the blockchain technology well
  - Blockchain might be a potential solution if:
    - Multiple parties are involved **and**
    - The parties do not trust each other or have different interests **and**
    - Shared write access is required **and**
    - All writes to the database need to be (publicly) verifiable

- Evaluate alternative solutions.

Do you need a blockchain?

No

(Joke) Model by Dave Birch (https://twitter.com/dgwbirch?lang=de)

[1] Decentralized application.
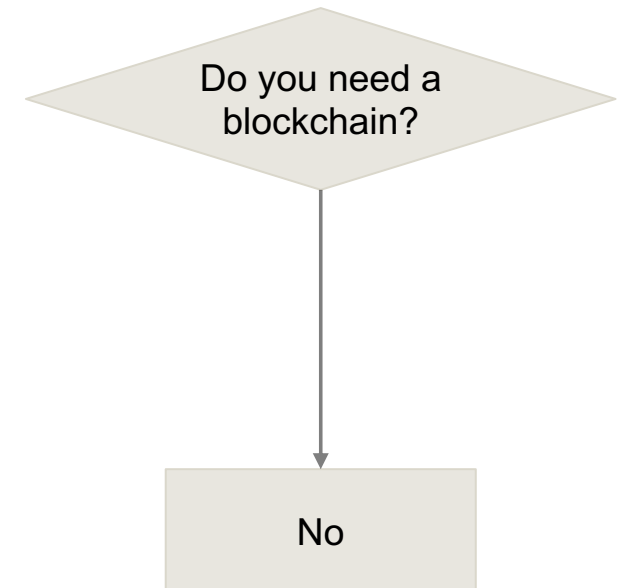
Model the Business Process

As in traditional software engineering, the first step for creating an application is to model the business process.

Identification

- Identify the involved parties, systems, and their relationship.
- Identify the necessary interactions between the parties and systems.
- Identify the information that is exchanged between the parties and systems.
- Identify the system boundaries.
- Diagrams can help to get the big picture.

Modelling

- Derive concrete models from the identified parties and systems.
- Define concrete messages that are exchanged between the systems and parties.
- Define a concrete data model used by the system.
- Derive concrete interfaces for interaction with the systems and parties.
- The overall architecture is usually modeled at a high abstraction level using architecture diagrams.
- The concrete software is modeled at a lower abstraction level using class diagrams.

Fictional Example
Blood Donation

The DRK ("Deutsches Rotes Kreuz") wants to digitize the blood donation process and make it more transparent. Therefore, it analyzes different technological solutions. As a first step, the DRK wants to track the supply chain from the blood extraction to the transfusion. The overall process is listed below.

Current simplified process:
- Person goes to the DRK and donates blood
- Blood is analyzed in laboratory and labeled
- Blood product is sent to hospital
- Hospital checks if a patients blood is compatible with the product
- Blood is transfused

Involved Systems:
- Laboratory management system (LMS) in the laboratory that analyzes the blood.
- The hospital information system (HIS) used in the hospital to manage patients and processes.

Involved Parties:
- DRK – Extracts the blood from the donor
- Laboratory – Analyzes the blood and creates blood products
- Donor – Donates his/her blood
- Hospital – Transfuses the blood
- Patient – Receives the blood transfusion

Interactions:
- DRK extracts blood from Donor
- DRK sends blood to laboratory
- Laboratory analyzes blood
- Hospital sends request to laboratory
- Laboratory sends blood to hospital
- Hospital transfuses blood to patient

Fictional Example (cont.)
Blood Donation

Assessing blockchain and contracts as potential solution architecture.

Multiple parties are involved ✓

- At least four parties are involved

The parties do not trust each other or have different interests –

- Questionable, some patients might not trust the DRK and therefore do not donate.

Shared write access is required ✓

- DRK needs to read and write
- The laboratory needs to read and write
- The hospital needs to read and write

All writes to the database need to be publicly verifiable ✓

- The donor needs to be able to track what happened with his/her donation.
- In case of an accident, the blockchain could help to identify the root cause and the responsible party.

Fictional Example (cont.)
Blood Donation Entities

In the Ethereum ecosystem two kinds of entities exist, externally owned accounts (EOAs) and contracts. Transactions are always issued by EOAs and usually controlled by an individual or a party. If an entity needs to be interactive and provide some on-chain functionality it is a candidate for a contract.

EOAs

Donor – Person who donates blood identified by their unique wallet address

DRK – Institution that runs blood donation events

Laboratory – Laboratory that takes the blood donation and creates blood products out of it

Hospital – The institution that transfuses the blood of the donor to a patient

Patient – The patient who gets a blood transfusion

Contracts

Blood Donation – Contract owned by the DRK, it tracks the time and date of the donation. Additionally, it records whenever the donation is passed to another party, e.g., from the DRK to the laboratory.

Fictional Example (cont.)
Blood Donation Transactions

In the Ethereum ecosystem transactions and messages are the only way for entities to interact.

Interactions

DRK → `CONTRACT_CREATION`:

Creates a blood donation (BD) and deploys it on the blockchain**.** The address of the donor is passed via constructor and unchangeable.

DRK → BD.sendToLaboratory(address laboratory):

The DRK issues a transaction to the BD when the blood donation is sent to the laboratory. The state variable for the laboratory is set.

Laboratory → BD.sendToHospital(address hospital):

The Laboratory issues a transaction to the BD when the blood donation is sent to the hospital. The state variable for the hospital is set. Only the hospital can do the transfusion.

Hospital → BD.transfuse(address patient):

The hospital issues a transaction to the BD when the blood is transfused to a patient. A transfusion can only happen once.

Fictional Example (cont.)
Blood Donation EOAs

DRK: 0x91A0639dDe409c126f058e33D743b1253738C8b9

PK: 0x3aae751e36ddffd4f7d5ff4bee409583a54df823111a30f780c18cd73ebb02f8

Laboratory: 0x3aDDBa6E0C56EE1357Bb9796b20480880cA37E81

PK: 0xdbe7d4d5460f6a6e086579a0acf071b652b6ed5ae0374d704a949cbb0b740a65

Hospital: 0x582FFFacdBFDaF1936672886035ea561FF669a44

PK: 0x8ae80121c7bc29a51eb4401754928051063a8dace9d35496dd26d0c4a1a0640c

Patient: 0x0780aFf9177d78E86Fc03158D504652f88c4D1bc

PK: 0x0a18f4e53a62e97b613ed94d0f411de327e0d1a0d5533c685042cae420aacbfb

Donor: 0x39bc67dBb1f5203AF048699233b29Dec903389A4

PK: 0x5aed62bff0a98533345482fb91ac80388869e6bdd5ad53c19b54a37468a5cb2d

07 Ethereum Smart Contracts - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2023). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.

CC BY-SA 4.0   34

Fictional Example (cont.)
Blood Donation Contract

```solidity
// File: BloodDonation.sol
pragma solidity >=0.4.22 <0.6.0;

contract BloodDonation {
  address donor;
  address drk;
  address laboratory;
  address hospital;
  address patient;
  bool isTransfused = false;
  modifier onlyDRK() {
      require(msg.sender == drk);_;
  }
  modifier onlyLaboratory() {
      require(msg.sender == laboratory);_;
  }
  modifier canTransfuse() {
      require(!isTransfused);
      require(msg.sender = hospital);
      _;
  }

// ends here ...
```

```solidity
// ... continues here

  constructor(address _donor) public {
      drk = msg.sender;
      donor = _donor;
      }
  function sendToLaboratory(address _laboratory) onlyDRK {
      laboratory = _laboratory;
  }

  function sendToHospital(address _hospital) onlyLaboratory {
      hospital = _hospital;
  }

  function transfuse(address _patient) canTransfuse {
      patient = _patient;
      isTransfused = true;
  }
}
```

Outline

1. Solidity Introduction
   - Definition
   - Anatomy of a Smart Contract
   - Language Features
   - Data Storage
   - Functions
   - Modifiers
   - Inheritance
   - Abstract Contracts and Interfaces

2. Designing Smart Contracts
   - Problem Assessment
   - Modelling Entities
   - Modelling Transactions

3. Cross-contract and Blockchain Interaction
   - EVM Contract Function Execution
   - Transactions and Messages
   - Address Class
   - Re-entrancy Attack
   - Message Object
   - Block Object
   - Transaction Object and Calldata
   - Events

Overview of EVM Contract Function Execution

Transactions and Messages
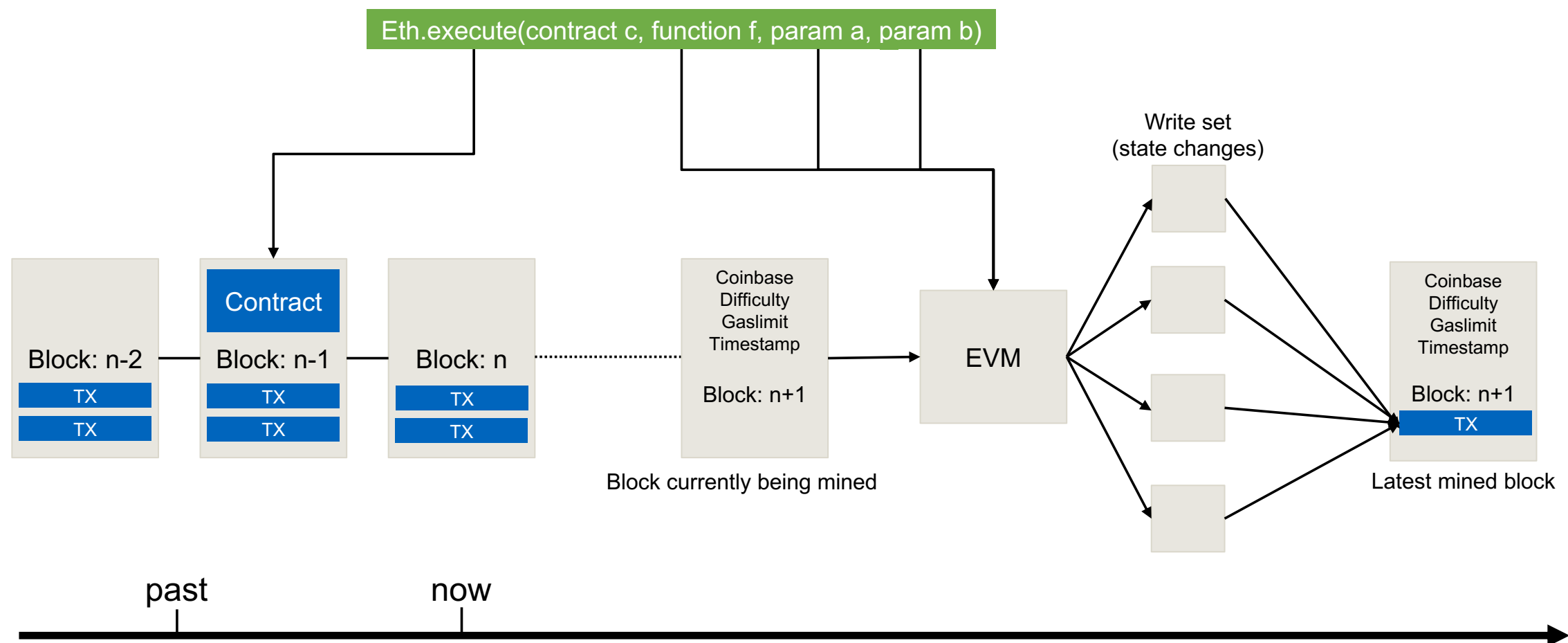
The **origin** of each **contract function call** is **always** a **transaction** by an externally owned account (EOA).

In more complex systems, multiple contracts communicate with each other. For example, when a contract uses an oracle contract to get information from the outside world. In such cases, the issuer of the function call must provide enough gas so that also the oracle request can be fulfilled.

Whenever a contract issues a message to another contract, the gas from the origin transaction is just forwarded. However, sometimes this is not intended, e.g., when only Ether should be transferred. Therefore, the Solidity address class implements functions specifically for that use case.

07 Ethereum Smart Contracts - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2023). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.

CC BY-SA 4.0   38
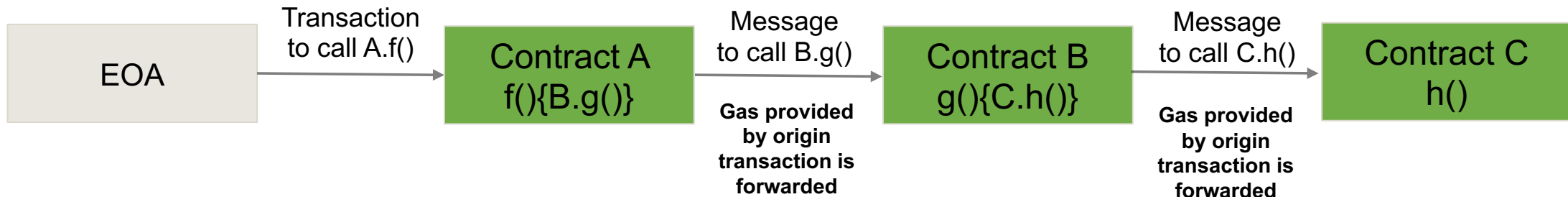
Address Class

Some contracts may require information about a specific account, e.g., the current account balance. Solidity implements a special type for accounts called address. Any Ethereum account, i.e., externally owned or contract, can be represented as an address object.

An address can be directly defined via a valid 20 byte hex code representation.
address a = 0xd5e7726990fD197005Aae8b3f973e7f2A65b4c18

An address that can receive Ether must either be defined as address payable or it should be cast with payable (<address>) function while sending Ether to it.

Furthermore, any contract object can be explicitly cast to an address.
```
contract A {
    function f() {}
}
contract B {
    function g() {
        A a = new A();
        address contract_a = address(a);
        address self = address(this);
    }
}
```

Working with Addresses

It is also possible to down-cast an address to a contract:

A a = A(0xd5e7726990fD197005Aae8b3f973e7f2A65b4c18)

This only works if the contract identified by the address is an instance of A.

```solidity
contract A {
    function f() {}
}


contract B {
    function g() {
        A a = new A();
        address contract_a = address(a);
        address self = address(this);
            // B b = B(self) would work
            // B b = B(contract_a) would fail
    }
}
```

Address Class

<address>.balance
The balance of the address in Wei returned as a 256-bit unsigned integer

> *Wei* is the smallest unit of currency in Ethereum
> **1 Ether = 10^9 Gwei = 10^{18} Wei**

<address>.transfer(uint256 value)
Transfers the amount passed as *value* in Wei to the *<address>.* The function throws on failure. Forwards only 2300 gas to *<address>*. **Must keep in mind that the called smart contract can quickly run out of gas and make the transfer impossible**.

<address>.send(uint256 value)
Same as *<address>.transfer(uint256 value)* but returns false on failure.

<address>.call(...)
A Low-level function that can be used to invoke functions but also to send Ether. The function returns false on failure and, by default, forwards all gas to *<address>*. **The called contract can execute complex operations that can spend all of the forwarded gas, causing more cost to the caller**. If there is no *receive* function defined in the called contract (i.e., if the fallback gets triggered upon Ether received), then, only 2300 gas is forwarded.

<address>.delegatecall(...)
A low-level function that can be used to call a function at *<address>* in the context/state of the current contract (i.e., *caller contract delegates the use of its storage to the receiving contract*). The function returns false on failure. Caller contract needs to trust the receiving contract[1].

Here is an article about how *delegatecall* can be dangerous: https://noxx.substack.com/p/evm-deep-dives-the-path-to-shadowy-a5f

Re-entrancy Attack

What is a re-entrancy attack?

- The re-entrancy attack is one of the most damaging attacks to a Solidity smart contract. When a function makes an external call to another untrusted contract, it becomes vulnerable to a re-entrancy attack.

- The untrusted contract can place **recursive calls back to the original function** in order to **drain all the funds** in the calling contract.

- This would work if the original function updates the balance of the receiving contract after transferring the coins.

- A is a contract where you can deposit and withdraw ETH. It keeps a record of how much it owes to other contracts. In our case, A owes 1 Ether to contract B.
- Contract B has two functions: fallback() and attack(). Both functions call withdraw() on A.

1. First, B executes *attack(),* which calls *withdraw()* on A.
2. A checks that debt to B is greater than 0 and sends 1 Ether to B (Contract B has 1 Ether, A has 9 Ether left).
3. Receiving Ether triggers *fallback()* on B*,* which immediately calls *withdraw()* again.
4. *withdraw()* checks debt to B and since it is still 1, A sends another Ether to B.
5. Now, A has 8 Ether and B has 2 Ether, while the debt to contract B in A is still 1 Ether. B calls *withdraw()* again.
6. *withdraw()* checks the debt to B, and since it is still 1, A sends another Ether to B.
7. This process between *withdraw()* and *fallback()* continues since the debt reset line in *withdraw()* cannot be reached. As long as this is the case, B can keep on withdrawing from A.

Prevention Measures Against the Re-entrancy Attack
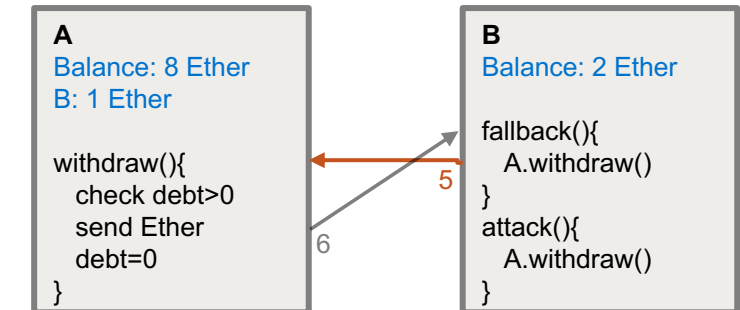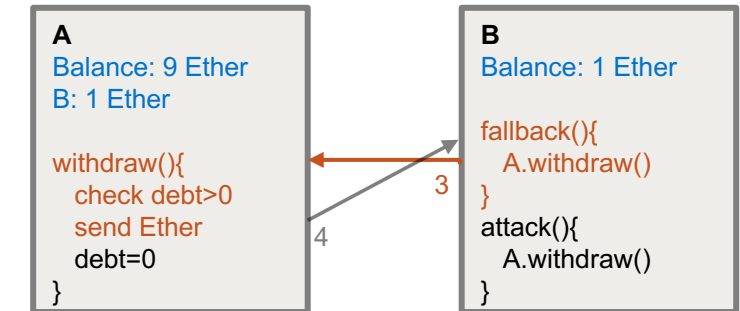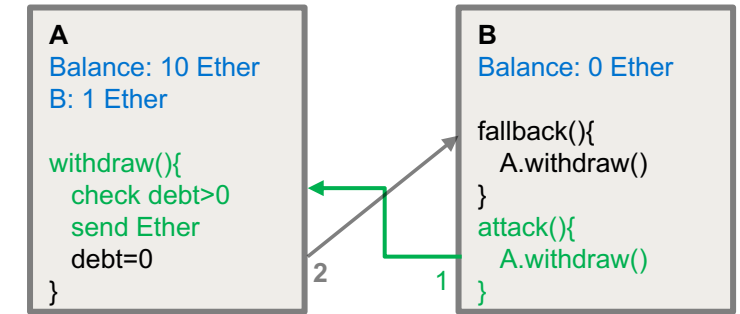
How to prevent a re-entrancy attack?

- Ensure **all state changes happen before calling external contracts** (update balances or code internally before calling external code)

- Use **function modifiers** that prevent re-entrancy

```solidity
 2  pragma solidity ^0.8.10;
 3
 4  contract ReEntrancyGuard {
 5      bool internal locked;
 6
 7      modifier noReentrant() {
 8          require(!locked, "No re-entrancy");
 9          locked = true;
10          _;
11          locked = false;
12      }
13  }
```

07 Ethereum Smart Contracts - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2023). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.

https://solidity-by-example.org/hacks/re-entrancy/

CC BY-SA 4.0   44

Message Object

Some contracts may require information about the caller of a function, e.g., for authentication purposes. Solidity provides the global msg object that contains information about the caller.

The object refers to the last account that was responsible for invoking the function. This can either be a contract or an externally owned account.

msg.sender
The account address of the function's caller, has type address (Needs to be cast to address payable when calling *transfer*, *send*, or *call*).

msg.data
The complete payload of the message/transaction

msg.sig
The function's hash signature so that the EVM knows which function is called

msg.value
The amount of Wei that is sent with the message

Message Object (cont.)

Since the message object always refers to the last sender, it requires some special attention when used in combination with this in a contract.

```solidity
contract A {
  function f() public returns () {
    return msg.sender;
  }

  function g() public returns () {
    return f(); // f() is called directly, msg.sender will be the address that calls g()
                // f() does not need to be public as this is an internal call
  }

  function h() public returns () {
    return this.f(); // f() is called by the current contract instance => msg.sender will always be equal to address(this) which is the contract address
                     // f() has to be public as this.f() behaves like an external call
  }
}
```

07 Ethereum Smart Contracts - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2023). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.

CC BY-SA 4.0   46

Block Object

Some contracts may require information about the latest proposed block, e.g., when a specific function should be time locked. Solidity provides a global variable called `block` to access the most recent block of the blockchain.

`block`.coinbase
The account address of the current block's proposer

`block`.difficulty
The current mining difficulty as an unsigned integer (returns 0 after the Merge)

`block`.gaslimit
The current block's *GASLIMIT*

`block`.timestamp
The UNIX timestamp of the block (in theory, can by manipulated by the block proposer)

07 Ethereum Smart Contracts - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2023). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.

CC BY-SA 4.0   47

Transaction Object

The global tx is similar to the msg object and provides information about the transaction that triggered the function call.

The main difference is that tx always refers to a transaction, i.e., its source is always an externally owned account.
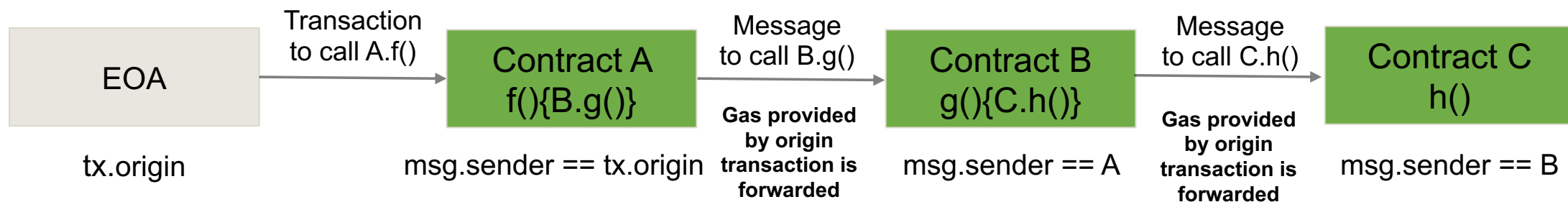
tx.origin
The issuer of the transaction. This is always an externally owned account.
DO NOT USE FOR AUTHENTICATION!

tx.gasprice
Information about the gas price that was used by the issuer of the transaction.
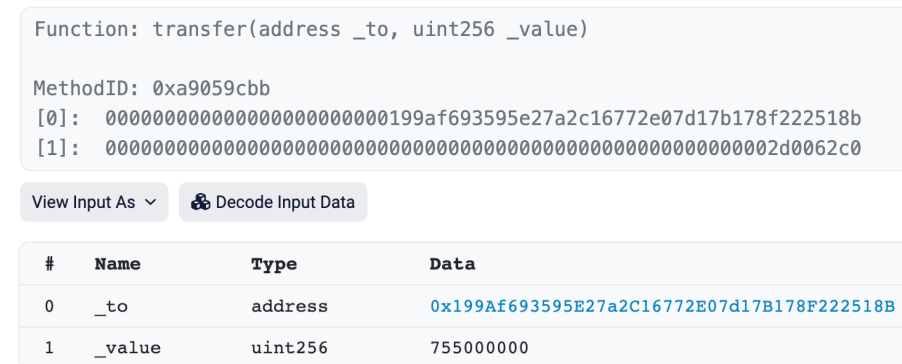
07 Ethereum Smart Contracts - Öz, B., Hoops, F., Gallersdörfer, U., & Matthes, F. (2023). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.

CC BY-SA 4.0   48

Transaction Calldata

An Ethereum transaction objects include a **data** field to place any extra information (represented in Keccak hash) required by the transaction.

This data field can:

- contain a smart contract's **bytecode** when the contract is initially **deployed**, or
- contain a **function signature** with any **arguments** passed in when a contract is called, or
- be **empty**, indicating that the transaction is a simple Ether transfer between two accounts.

Often, the transaction data is referred to as *calldata*[1] since it includes information about a call to a function on a smart contract.

Transaction Hash:	0xdc33d512db0730614ad930907546330ffa73dae318077874e169a814b4f13b40
Status:	Success
Block:	16947410 86801 Block Confirmations
Timestamp:	12 days 6 hrs ago (Mar-31-2023 01:15:23 PM +UTC)
Sponsored:	
From:	komesciart.eth
Interacted With (To):	0xdAC17F958D2ee523a2206206994597C13D831ec7 (Tether: USDT Stablecoin)
ERC-20 Tokens Transferred:	From 0x4241B8...d8D46c24 To 0x199Af6...F222518B For 755 ($756.51) Tether USD... (USDT...)
Value:	0 ETH ($0.00)
Transaction Fee:	0.001942234962724264 ETH $3.70
Gas Price:	30.733024712 Gwei (0.000000030733024712 ETH)

Ether Price:	$1,822.21 / ETH
Gas Limit & Usage by Txn:	94,795	63,197 (66.67%)
Gas Fees:	Base: 30.633024712 Gwei	Max: 44.462835327 Gwei	Max Priority: 0.1 Gwei
Burnt & Txn Savings Fees:	Burnt: 0.001935915262724264 ETH ($3.69) Txn Savings: 0.000867682841436155 ETH ($1.65)
Other Attributes:	Txn Type: 2 (EIP-1559) Nonce: 886 Position In Block: 139
Input Data:	0xa9059cbb0000000000000000000000000199af693595e27a2c16772e07d17b178f222518b00000000000000000000000000000000000000000000000000000000000000002d0062c0

Keccak hash representation of a transaction's calldata on Etherscan

View Input As ⌄   Decode Input Data

Decoding the Calldata

By looking at the raw transaction calldata, it is **challenging to understand** which function is called and what are the input arguments. However, the calldata can be **decoded**.

1. The first 8 hex chars. (4 bytes) represent the **function signature**.
   - If we have the source code (or ABI[1]) of the contract which is interacted with, we can figure out the specific function called using a Keccak hash tool.

2. The remainder of the calldata encode the input arguments, each represented in 64 hex chars.
   i. The **first argument** is an **address**, which is, by default, 40 hex chars. long. (the first 24 zeros are just padding)
   ii. The **second argument** is a 256-bit (64 hex chars.) long **unsigned integer**. After removing the padding (56 zeros), we can find the input (*2d0062c0*) and convert it to a decimal number to get the original value.

Raw calldata in Keccak hash

Keccak-256 online hash function

```
transfer(address,uint256)
```

Input type [Text ▾]

[Hash] ☑ Auto Update

```
a9059cbb2ab09eb219583f4a59a5d0623ade346d962bcd4e46b11da047c9049b
```

```
Function: transfer(address _to, uint256 _value)

MethodID: 0xa9059cbb
[0]:   00000000000000000000000199af693595e27a2c16772e07d17b178f222518b
[1]:   00000000000000000000000000000000000000000000000000000000002d0062c0
```

[View Input As ▾]   [🧩 Decode Input Data]

#	Name	Type	Data
0	_to	address	0x199Af693595E27a2C16772E07d17B178F222518B
1	_value	uint256	755000000

Decoded calldata

[1] ABI stands for "Application Binary Interface" and defines the variables and the methods that a contract offers.
Keccak Hash Tool: https://emn178.github.io/online-tools/keccak_256.html
Inspect this transaction on Etherscan: https://etherscan.io/tx/0xdc33d512db0730614ad930907546330ffa73dae318077874e169a814b4f13b40

Events

When a contract wants to communicate with external applications like decentralized applications (dApps), it emits **events**. Events are **dispatched signals** that are fired by contracts. Developers of the contract decide when to send these signals.

- dApps or any program using an Ethereum API can listen to events emitted by contracts.
- Useful for data analysis

Use Cases

- Provide data from a smart contract to a user interface
- Trigger an oracle update[1]
- Cheap data storage (logging)

```solidity
contract Counter {

    //declare event and what it contains, the old value and the new value
    event ValueChanged(uint oldValue, uint256 newValue);

    // Private variable of type unsigned int to keep the number of counts
    uint256 private count = 0;

    // Function that increments our counter
    function increment() public {
        count += 1;

        //When count variable is incremented, emit the event.
        emit ValueChanged(count - 1, count);
    }

    // Getter to get the count value
    function getCount() public view returns (uint256) {
        return count;
    }
}
```
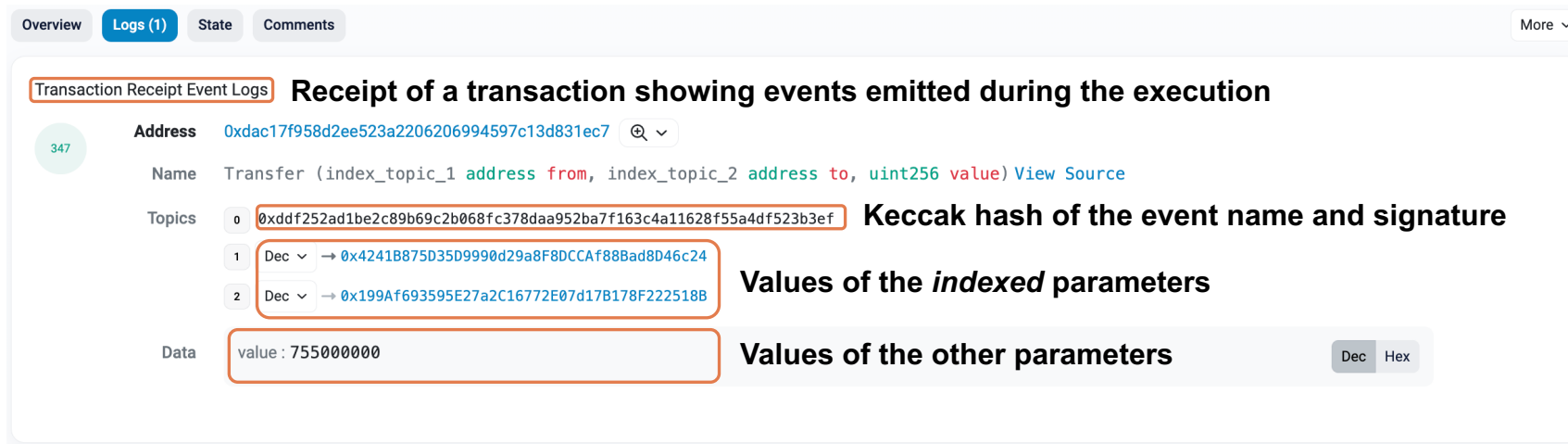
An event signaling a change in the counter value.

[1] We will cover the concept of oracles in the lecture "Ethereum Design Patterns".

Transfer Event

- One of the most commonly used events on the Ethereum blockchain is the *Transfer* event which gets emitted when transferring tokens:

```
event Transfer(address indexed from, address indexed to, uint256 value);
```

- The transfer event logs who made the transfer (**from**), who it was transferred to (**to**), and how many tokens were sent (**value**).
- An event attribute can be declared as **indexed** so that the event history can be efficiently filtered for events with an exact match on this attribute.
  - E.g., find all *Transfer* events where the sender of the tokens is "0xa0b52ef1…"

Inspect this event on Etherscan: https://etherscan.io/tx/0xdc33d512db0730614ad930907546330ffa73dae318077874e169a814b4f13b40#eventlog