Recap Ethereum Smart Contracts

Blockchain-Based Systems Engineering

Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
wwwmatthes.in.tum.de

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.**

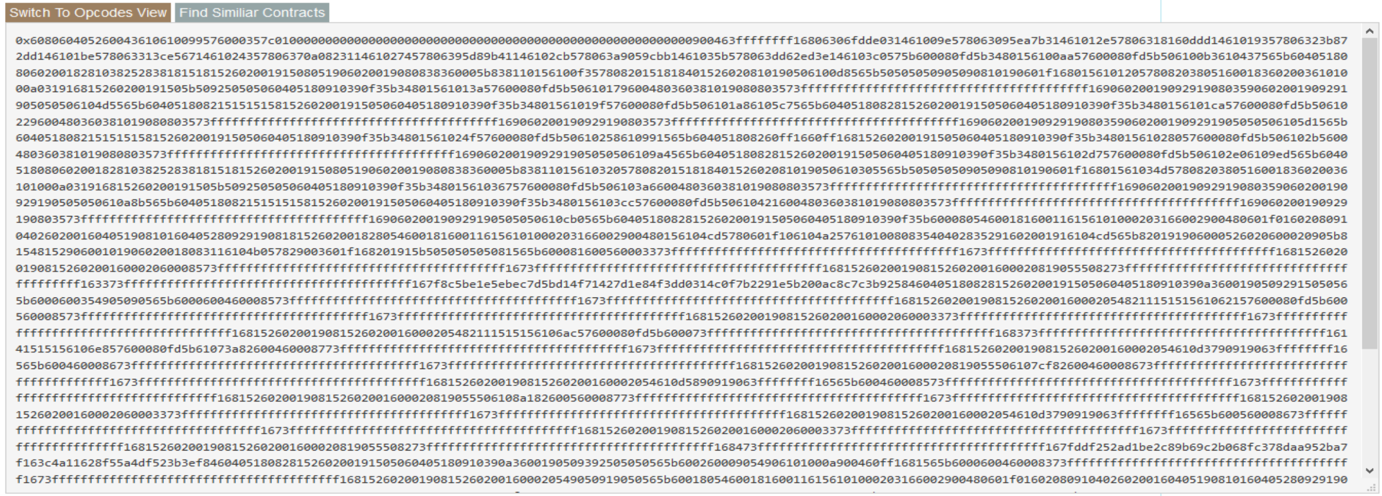
- Solidity is **compiled to bytecode** which is **persistent and immutable** once deployed to the blockchain:
  ➔ **No patch deployment** possible
  ➔ **Smart contracts** must be **perfect before** using them in **production**!

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

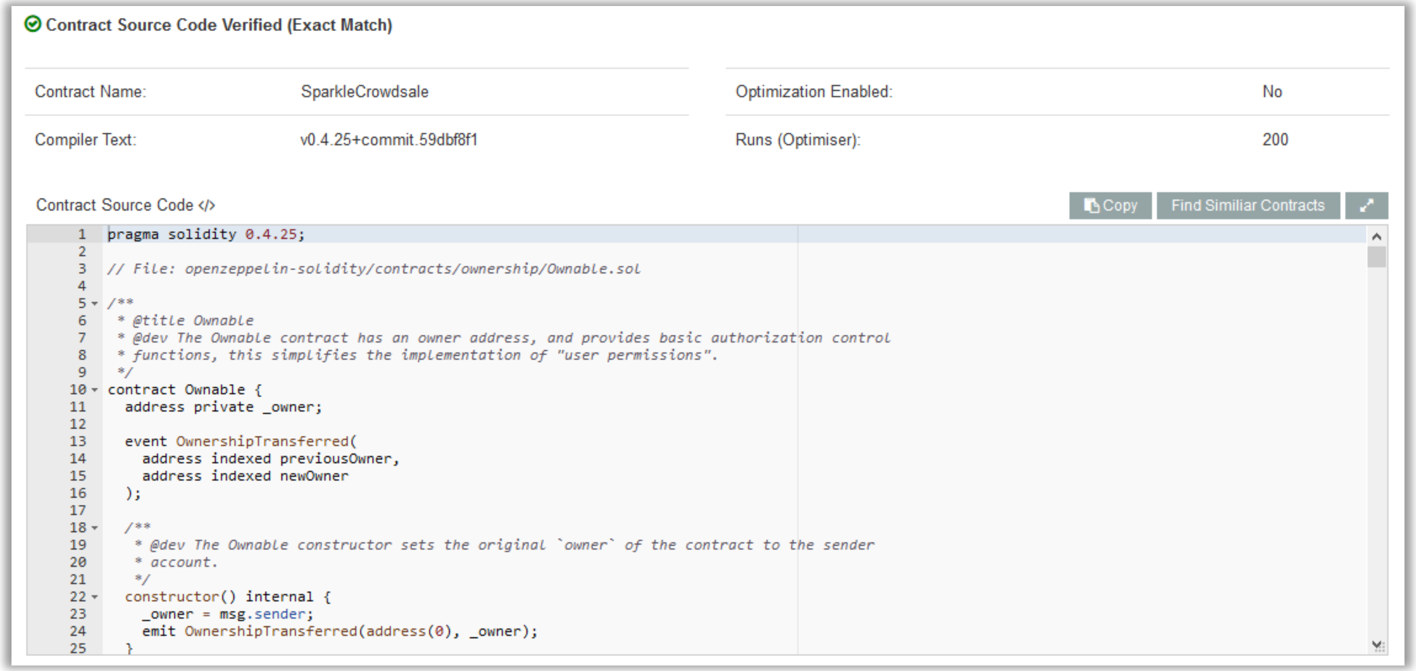
Source Code

- Source code is typically not stored on the blockchain, only byte code.
- Without further analysis, the purpose of this smart contract is unclear.

- Source code can be made publicly available.
- Etherscan.io is the only service which verifies source codes and the respective byte code.



```
0x608060405260043610610099576000357c0100000000000000000000000000000000000000000000000000000000900463ffffffff16806306fdde031461009e578063095ea7b31461012e57806318160ddd1461019357806323b87
2dd146101be578063313ce5671461024357806370a082311461027457806395d89b41146102cb578063a9059cbb1461035b578063dd62ed3e146103c0575b600080fd5b3480156100aa57600080fd5b506100b3610437565b60405180
80602001828103825283818151815260200191508051906020019080838360005b838110156100f357808201518184015260208101905061 00d8565b50505050905090810190601f16801561012057808203805160018360200361010
00a031916815260200191505b509250505060405180910390f35b34801561013a57600080fd5b50610179600480360381019080803573ffffffffffffffffffffffffffffffffffffffff1690602001909291908035906020019092 91
905050506104d5565b604051808215151515815260200191505060405180910390f35b34801561019f57600080fd5b506101a86105c7565b6040518082815260200191505060405180910390f35b3480156101ca57600080fd5b50610
2296004803603810190808035 73ffffffffffffffffffffffffffffffffffffffff169060200190929190803573ffffffffffffffffffffffffffffffffffffffff16906020019092919080359060200190929190505050 6105d1565b
604051808215151515815260200191505060405180910390f35b34801561024f57600080fd5b50610258610991565b604051808260ff1660ff16815260200191505060405180910390f35b34801561028057600080fd5b506102b5600
480360381019080803573ffffffffffffffffffffffffffffffffffffffff1690602001909291905050506109a4565b6040518082815260200191505060405180910390f35b3480156102d757600080fd5b506102e06109ed565b6040
5180806020018281038252838181518152602001915080519060200190808383600 05b83811015610320578082015181840152602081019050610305565b50505050905090810190601f16801561034d5780820380516001836020036
101000a031916815260200191505b509250505060405180910390f35b34801561036757600080fd5b506103a6600480360381019080803573ffffffffffffffffffffffffffffffffffffffff16906020019092919080359060200190
929190505050610a8b565b604051808215151515815260200191505060405180910390f35b3480156103cc57600080fd5b506104216004803603810190808035 73ffffffffffffffffffffffffffffffffffffffff1690602001 90929
190803573ffffffffffffffffffffffffffffffffffffffff16906020019092919050505061 0cb0565b6040518082815260200191505060405180910390f35b6000805460018160011615610100020316600290048060 1f0160208091
0402602001604051908101604052809291908181526020018280546001816001161561010002031660029004801561 04cd5780601f106104a25761010080835404028352916020019161 04cd565b82019190600052602060002090 5b8
154815290600101906020018083116104b057829003601f168201915b50505050505081565b600081600560003373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff1681526020
0190815260200160002060008573ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002081905550 8273ffffffffffffffffffffffffffffffff
ffffffff163373ffffffffffffffffffffffffffffffffffffffff167f8c5be1e5ebec7d5bd14f71427d1e84f3dd0314c0f7b2291e5b200ac8c7c3b925846040518082815260200191505060405180910390a36001905092915050 56
5b6000600354905090565b6000600460008573ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002054821115151561062157600080fd5b600
560008573ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002060003373ffffffffffffffffffffffffffffffffffffffff1673ffffffffff
ffffffffffffffffffffffffffffff1681526020019081526020016000205482111515156106ac57600080fd5b600073ffffffffffffffffffffffffffffffffffffffff168373ffffffffffffffffffffffffffffffffffffffff161
415151561 06e857600080fd5b61073a82600460008773ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002054610d3790919063ffffffff16
565b600460008673ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002081905550 6107cf82600460008673fffffffffffffffffffffffffff
fffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002054610d5890919063ffffffff16565b600460008573ffffffffffffffffffffffffffffffffffffffff1673ffffffffffff
ffffffffffffffffffffffffffff16815260200190815260200160002081905550 6108a182600560008773ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff168152602001908
15260200160002060003373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002054610d3790919063ffffffff16565b600560008673ffffff
ffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002060003373ffffffffffffffffffffffffffffffffffffffff1673fffffffffffffffffffffffff
fffffffffffffff16815260200190815260200160002081905550 8273ffffffffffffffffffffffffffffffffffffffff168473ffffffffffffffffffffffffffffffffffffffff167fddf252ad1be2c89b69c2b068fc378daa952ba7
f163c4a11628f55a4df523b3ef846040518082815260200191505060405180910390a3600190509392505050 565b600260009054906101000a900460ff1681565b600060046000 8373fffffffffffffffffffffffffffffffffffffff
f1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002054905091905056 5b6001805460018160011615610100020316600290048060 1f0160208091040260200160405190810160405280929190
```



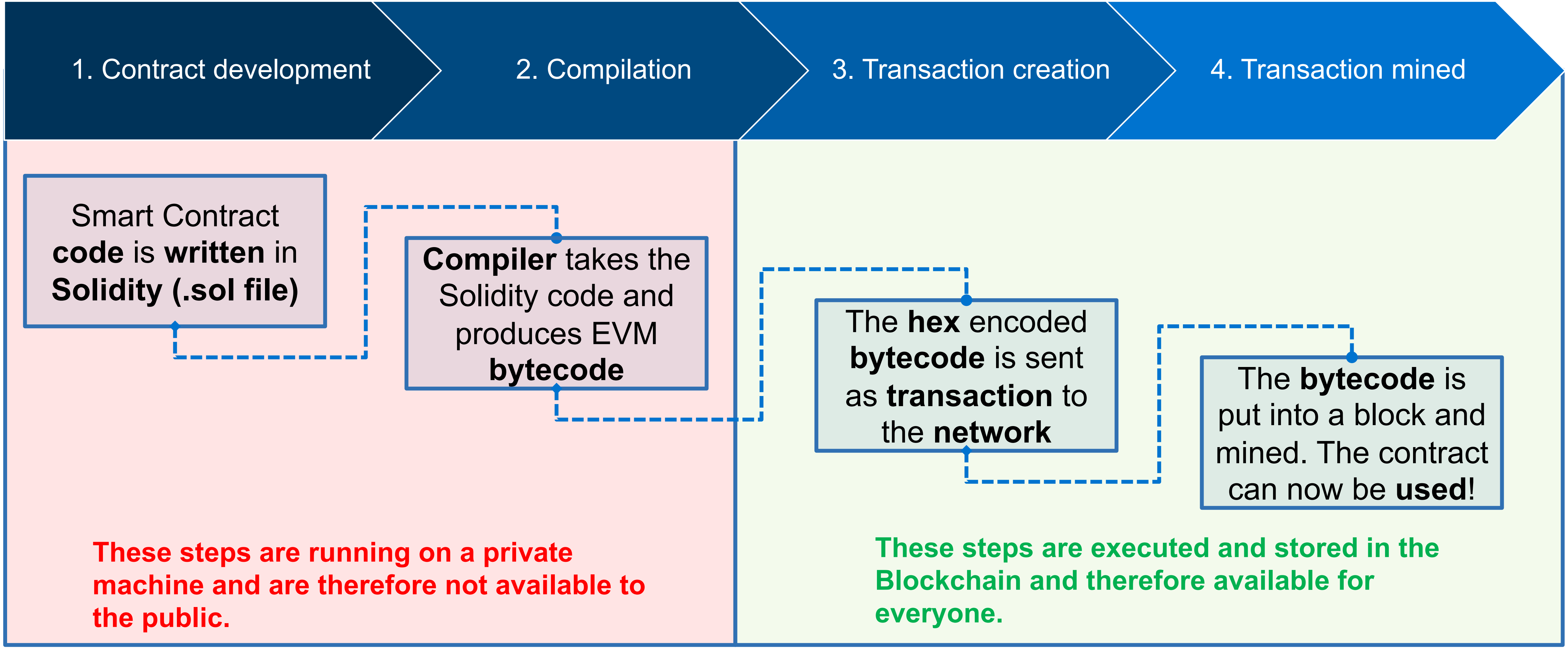
Contract Source Code Verified (Exact Match)

Contract Name:	SparkleCrowdsale	Optimization Enabled:	No
Compiler Text:	v0.4.25+commit.59dbf8f1	Runs (Optimiser):	200

Contract Source Code </>

```
pragma solidity 0.4.25;

// File: openzeppelin-solidity/contracts/ownership/Ownable.sol

/**
 * @title Ownable
 * @dev The Ownable contract has an owner address, and provides basic authorization control
 * functions, this simplifies the implementation of "user permissions".
 */
contract Ownable {
  address private _owner;

  event OwnershipTransferred(
    address indexed previousOwner,
    address indexed newOwner
  );

  /**
   * @dev The Ownable constructor sets the original `owner` of the contract to the sender
   * account.
   */
  constructor() internal {
    _owner = msg.sender;
    emit OwnershipTransferred(address(0), _owner);
  }
```

From Solidity Source Code to a Deployed Smart Contract



1. Contract development

2. Compilation

3. Transcation 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**

```
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 transactions and therefore costs ether.
- Reading the state of a contract is free and does not require a transaction.

Anatomy of a Solidity Smart Contract File (cont.)

File: **BBSE.sol**

```
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.

Anatomy of a Solidity Smart Contract File (cont.)

File: **BBSE.sol**

```
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.

Anatomy of a Solidity Smart Contract File (cont.)

File: **BBSE.sol**

```
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, array, struct, enum, mapping

Built-in first level objects

block, msg, tx, address

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 variable** 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 nor via transactions.

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

Special Function Types (cont.)

Payable function
By default, it is not possible to send ether to a function because the function will by default revert the transaction. The behavior is intentional, it should prevent Ether that is accidentally sent from being lost. However, sometimes it is necessary to pay a contract, e.g. in case of an ICO. Therefore, Solidity implements so-called *payable* functions.

Example

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

- The keyword payable is also required for declaring constructors and addresses that can receive Ether (e.g., 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);
}
```

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, as well as, contract, can be represented as 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 casted to an address.

```
contract A {
  function f() {}
}
contract B {
     function g() {
        A a = new A();
        address contract_a = address(a);
        address self = address(this);
     }
}
```

Address Class

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

<address>.transfer(uint256 value)
Transfers the amount passed as *value* in Wei to the *<address>.* The function throws on failure. Forwards 2300 gas to *<address>*. (**NOTE**: 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>* (**NOTE**: 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*). This function returns false on failure. (**NOTE**: Caller contract needs to trust the receiving contract)

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. It does not matter whether the caller of the function was an externally owned account or another contract.

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, which has type address (**NOTE**: 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

Block Object

Some contracts may require information about the latest mined 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 miner

block.difficulty

The current mining difficulty as unsigned integer

block.gaslimit

The current block's gaslimit (by the miner)

block.timestamp

The UNIX timestamp of the block (in theory, can by manipulated by the miner)