Smart Contracts and Blockchain Technology

Lecture 10. Solidity functions and contracts

Christian Ewerhart

University of Zurich

Fall 2022

Introduction and overview

Last lecture: Basic language concepts of Solidity

- Booleans, integers, and addresses
- Type conversion (implicit and explicit)
- Elementary operations and members

This lecture: Core language concepts

- Functions
- Contracts

Solidity functions and contracts (1)

Functions

Functions are executable units of code, typically defined inside a contract:[1]

```
contract myFirstContract {
    function myFirstFunction() {
        \\ do something
    }
}
```

The declaration of a function may specify:

- **parameters** as input, and
- **return variables** as output.[2]

[1] However, they can also be defined outside of contracts.

[2] Both parameters and return variables are optional.

Solidity functions and contracts (2)

Example

```
function Double(uint x) public pure returns (uint y) {
    y = 2*x;
}
```

Equivalent:

```
function Double(uint x) public pure returns (uint) {
    return(2*x);
}
```

The function is `pure` because it does not read or modify the state.[3]

[3] Functions may also be of type `view` if they read the state but do not modify it. These properties are referred to as the **mutability** of the function.

Solidity functions and contracts (3)

Function calls

Functions are called either internally or externally:

- **Internal function calls** target a function inside the current code unit, which includes the current contract and inherited functions, in particular.
- **External function calls** target a function outside of the current code unit. They require an address and a so-called function signature.

Solidity functions and contracts (4)

Function visibility

Function calls may be prohibited using visibility specifiers:

External functions can be called via message calls only.[4]

Public functions can be called both via message calls and internally.

Private functions can be accessed from the current contract only.

Internal functions are like private functions but can also be accessed from contracts deriving from it.

inheritance:

```
contract Parent {                -> Base contract
    //…
}

contract Child is Parent {       -> Derived contract
    //…
}
```

By using it, a child contract can use all elements of parent contract

[4] I.e., `f()` does not work, but `this.f()` works.

Solidity functions and contracts (5)

Function modifiers

Modifiers amend the semantics of functions in a declarative way:

```
contract Purchase {
    address public seller;
    modifier onlySeller() {
        require(
            msg.sender == seller,
            "Not authorized");
        _;
    }
    function abort() public view onlySeller {
        // ...
    }
}
```

Solidity functions and contracts (6)

Events

Events are used for logging. Once emitted, they can be read by any server that has access to a node of the blockchain network.

```
contract Auction {

    event BidRaised(address bidder, uint amount);

    function bid() public payable {
        // ...
        emit BidRaised(msg.sender, msg.value);
    }
}
```

Solidity functions and contracts (7)

msg.sender and msg.value

During the execution of a function call, we have access to `msg.sender` and `msg.value`, in particular:

- `msg.data` (`bytes calldata`): complete calldata
- `msg.sender` (`address`): sender of the message (current call)
- `msg.sig` (`bytes4`): first four bytes of the calldata (i.e. function identifier)
- `msg.value` (`uint`): number of wei sent with the message

Solidity functions and contracts (8)

tx.origin and tx.gasprice

Further, we have access to `tx.origin` and `tx.gasprice`:

- `tx.gasprice` (`uint`): gas price of the transaction
- `tx.origin` (`address`): sender of the transaction (full call chain)

Solidity functions and contracts (9)

Selfdestruct

A contract may be programmed to self-destruct:

```
selfdestruct(0xAd8...866);
```

This operation will:

- erase the current contract code as well as the contract storage,[5]
- transfer the balance at the contract address to the address provided.

Note. `selfdestruct` is not recommended (ether sent to the contract is lost). Instead, change some internal state which causes all functions to revert.

[5] However, the history of the blockchain may be retained by the nodes. Under `delegatecall`, the calling contract gets erased.

Solidity functions and contracts (10)

Contracts

Contracts in Solidity are similar to classes in object-oriented languages.

They contain persistent data in state variables, and functions that can modify these variables.

Calling a function on a different contract (instance) will perform an EVM function call and thus switch the context such that state variables in the calling contract are inaccessible.

A contract can refer to itself using `this`:

```
address(this).balance
```

Solidity functions and contracts (11)

Creating a contract

Contracts can be created "from outside" via Ethereum transactions or from within Solidity contracts.

IDEs, such as Remix, make the creation process seamless using UI elements.[6]

One way to create smart contracts programmatically is via the web3.js library.[7]

[6] IDE = Interactive Development Environment. UI = User Interface.

[7] The ending .js stands for JavaScript, the programming language most prevalently used in internet applications.

Solidity functions and contracts (12)

Constructor

The **constructor** of a contract is an (optional) function that is executed once at the time of contract creation:[8]

```
constructor() {
    // do something
}
```

After the constructor has executed, the code of the contract is stored on the blockchain. This code includes all public and external functions and all functions that are reachable from there through function calls.[9]

[8] The contructor should be placed on top of all functions of a contract.

[9] The deployed code does not include the constructor code nor internal functions only called from the constructor.

Solidity functions and contracts (13)

Receive

The **receive** function is an (optional) function that accepts incoming payments.

```
receive() external payable {
    // do something
}
```

Warning. A contract without a receive function can receive ether as a recipient of a coinbase transaction or as a destination of a selfdestruct. A contract cannot react to such transfers and thus also cannot reject them.

Solidity functions and contracts (14)

Fallback function

A contract may have a **fallback function**. This function is executed if either the other functions do not match, or if no data was supplied at all and there is no `receive` function.

```
fallback() external payable {
    // do something
}
```

This function must have external visibility. In order to receive ether, it must be marked as payable (as in the example above). With parameters, the fallback function looks as follows:

```
fallback (bytes calldata input) external payable
returns (bytes memory output) {}
```

Solidity functions and contracts (15)

Example

```
contract AcceptingPayments {
    address payable owner;
    constructor() {
        owner = payable(msg.sender);
    }
    receive() external payable {}
    fallback() external payable {}
    function collect() {
        require(
            owner == payable(msg.sender),
            "not authorized"
        );
        selfdestruct(owner);
    }
```

Solidity functions and contracts (16)

Error handling

There are three ways to (conditionally) trigger an error:

- `assert`
- `require`
- `revert`

In addition, errors may be caught by calling contracts:

- `try` and `catch`

Solidity functions and contracts (17)

Assert

To check a condition that should be true for any input, you may use the **assert function**:

```
function GetItRight() {
    assert(1+1==2);
}
```

If the condition is not met, a panic error is triggered.

Solidity functions and contracts (18)

Require

To check a condition, you may use the **require function**:

```
require(msg.sender == owner; "not authorized");
```

Solidity functions and contracts (19)

Revert function

To trigger an unconditional error, you may use the **revert function**:[10]

```
revert("This should not have happened!");
```

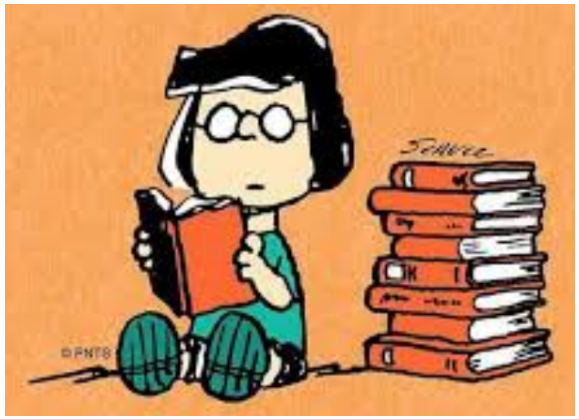
The error data will be passed back to the caller and can be caught there using a try/catch statement.

[10] To save gas, one may alternatively define an error message and call the revert statement.

Solidity functions and contracts (20)

Bibliographic notes

This slide deck is based on Section 3.9 of the official documentation of the programming language Solidity (link).



Solidity functions and contracts (21)

References

Solidity Documentation, **Release 0.8.17**, Ethereum Foundation, September 8, 2022.