Privacy protection

Smart contracts also protect your privacy. Since Ethereum is a pseudonymous network (your transactions are tied publicly to a unique cryptographic address, not your identity), you can protect your privacy from observers.

Visible terms

Finally, like traditional contracts, you can check what's in a smart contract before you sign it (or otherwise interact with it). A smart contract's transparency guarantees that anyone can ` it.

Smart contract use cases

Smart contracts can do essentially anything that computer programs can do.
They can perform computations, create currency, store data, mint NFTs, send communications and even generate graphics. Here are some popular, real-world examples:

- Stablecoins
- Creating and distributing unique digital assets   USDC – US Digital Currency
- An automatic, open currency exchange
- Decentralized gaming
- An insurance policy that pays out automatically(opens in a new tab)
- A standard that lets people create customized, interoperable currencies

https://ethereum.org/en/developers/docs/smart-contracts/anatomy/

ANATOMY OF SMART CONTRACTS

Last edit: @wackerow(opens in a new tab), August 15, 2022
See contributors

A smart contract is a program that runs at an address on Ethereum. They're made up of data and functions that can execute upon receiving a transaction or Oracles. Here's an overview of what makes up a smart contract.

Prerequisites

Make sure you've read about smart contracts first. This document assumes you're already familiar with programming languages such as JavaScript or Python.

DATA

Any contract data must be assigned to a location: either to storage or memory. It's costly to modify storage in a smart contract so you need to consider where your data should live.

Storage

Persistent data is referred to as storage and is represented by state variables. These values get stored permanently on the blockchain. You need to declare the type so that the contract can keep track of how much storage on the blockchain it needs when it compiles.

From <https://ethereum.org/en/developers/docs/smart-contracts/anatomy/>

```
// Solidity example
contract SimpleStorage {
   uint storedData; // State variable
   // ...
}
```

```
# Vyper example
storedData: int128
```

If you've already programmed object-oriented languages, you'll likely be familiar with most types. However address should be new to you if you're new to Ethereum development.
An address type can hold an Ethereum address which equates to 20 bytes or 160 bits. It returns in hexadecimal notation with a leading 0x.
Other types include:

- boolean
- integer
- fixed point numbers
- fixed-size byte arrays
- dynamically-sized byte arrays
- Rational and integer literals
- String literals
- Hexadecimal literals
- Enums

For more explanation, take a look at the docs:

- See Vyper types(opens in a new tab)
- See Solidity types(opens in a new tab)

Memory

Values that are only stored for the lifetime of a contract function's execution are called memory variables. Since these are not stored permanently on the blockchain, they are much cheaper to use.
Learn more about how the EVM stores data (Storage, Memory, and the Stack) in the Solidity docs(opens in a new tab).

Environment variables

In addition to the variables you define on your contract, there are some special global variables. They are primarily used to provide information about the blockchain or current transaction. Oracles
Examples:

Prop	State Variable	Description
block.timestamp	uint256	Current block epoch timestamp
msg.sender	address	Sender of the message (current call)

FUNCTIONS

In the most simplistic terms, functions can get information or set information in response to incoming transactions.
There are two types of function calls:

- internal – these don't create an EVM call
    - Internal functions and state variables can only be accessed internally (i.e. from within the current contract or contracts deriving from it)
- external – these do create an EVM call
    - External functions are part of the **contract interface**, which means they can be called from other contracts and via transactions. An external function **F** cannot be called internally (i.e. **F**() does not work, but this.**F**() works).

They can also be **public** or **private**

- public functions can be called internally from within the contract or externally via messages
- private functions are only visible for the contract they are defined in and not in derived contracts

Both functions and state variables can be made public or private
Here's a function for updating a state variable on a contract:

```
// Solidity example
function update_name(string value) public {
    dapp_name = value;
}
```

- The parameter value of type string is passed into the function: update_name
- It's declared public, meaning anyone can access it
- It's not declared view, so it can modify the contract state

View functions

These functions promise not to modify the state of the contract's data. Common examples are "getter" functions – you might use this to receive a user's balance for example.

```
// Solidity example
function balanceOf(address _owner) public view returns (uint256 _balance) {
   return ownerPizzaCount[_owner];
}
```

```
dappName: public(string)

@view
@public
def readName() -> string:
  return dappName
```

What is considered modifying state:

1. Writing to state variables.
2. Emitting events(opens in a new tab).
3. Creating other contracts(opens in a new tab).
4. Using selfdestruct.
5. Sending ether via calls.
6. Calling any function not marked view or pure.
7. Using low-level calls.
8. Using inline assembly that contains certain opcodes.

Constructor functions

Constructor functions are only executed once when the contract is first deployed. Like constructor in many class-based programming languages, these functions often initialize state variables to their specified values.

```
// Solidity example
// Initializes the contract's data, setting the `owner`
// to the address of the contract creator.
constructor() public {
    // All smart contracts rely on external transactions to trigger its functions.
    // `msg` is a global variable that includes relevant data on the given transaction,
    // such as the address of the sender and the ETH value included in the transaction.
    // Learn more: https://solidity.readthedocs.io/en/v0.5.10/units-and-global-variables.html#block-and-transaction-properties
    owner = msg.sender;
}
```

Built-in functions

In addition to the variables and functions you define on your contract, there are some special built-in functions. The most obvious example is:

- address.send() – Solidity

These allow contracts to send ETH to other accounts.

WRITING FUNCTIONS

Your function needs:

- parameter variable and type (if it accepts parameters)
- declaration of internal/external
- declaration of pure/view/payable
- returns type (if it returns a value)

```
pragma solidity >=0.4.0 <=0.6.0;

contract ExampleDapp {
    string dapp_name; // state variable

    // Called when the contract is deployed and initializes the value
    constructor() public {
        dapp_name = "My Example dapp";
    }

    // Get Function
    function read_name() public view returns(string) {
        return dapp_name;
    }

    // Set Function
    function update_name(string value) public {
        dapp_name = value;
    }
}
```

Rutine programming vs Problemic programming

Internet sites, SQL data b. Blockchain programming

web 3

A complete contract might look something like this. Here the constructor function provides an initial value for the dapp_name variable.

EVENTS AND LOGS

Events let you communicate with your smart contract from your frontend or other subscribing applications.

When a transaction is mined, smart contracts can emit events and write logs to the blockchain that the frontend can then process.

ANNOTATED EXAMPLES

These are some examples written in Solidity. If you'd like to play with the code, you can interact with them in Remix(opens in a new tab).

Hello world

```
// Specifies the version of Solidity, using semantic versioning.
// Learn more: https://solidity.readthedocs.io/en/v0.5.10/layout-of-source-files.html#pragma
pragma solidity ^0.5.10;

// Defines a contract named `HelloWorld`.
// A contract is a collection of functions and data (its state).
// Once deployed, a contract resides at a specific address on the Ethereum blockchain.
// Learn more: https://solidity.readthedocs.io/en/v0.5.10/structure-of-a-contract.html
contract HelloWorld {

    // Declares a state variable `message` of type `string`.
    // State variables are variables whose values are permanently stored in contract storage.
    // The keyword `public` makes variables accessible from outside a contract
    // and creates a function that other contracts or clients can call to access the value.
    string public message;

    // Similar to many class-based object-oriented languages, a constructor is
    // a special function that is only executed upon contract creation.
    // Constructors are used to initialize the contract's data.
    // Learn more: https://solidity.readthedocs.io/en/v0.5.10/contracts.html#constructors
    constructor(string memory initMessage) public {
        // Accepts a string argument `initMessage` and sets the value
        // into the contract's `message` storage variable).
        message = initMessage;
    }

    // A public function that accepts a string argument
    // and updates the `message` storage variable.
    function update(string memory newMessage) public {
        message = newMessage;
    }
}
```

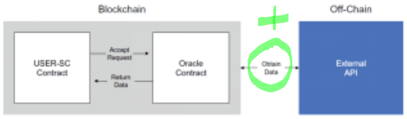
Token

```
pragma solidity ^0.5.10;

contract Token {
   // An `address` is comparable to an email address - it's used to identify an account on Ethereum.
   // Addresses can represent a smart contract or an external (user) accounts. owned
   // Learn more: https://solidity.readthedocs.io/en/v0.5.10/types.html#address                EOA
   address public owner;

   // A `mapping` is essentially a hash table data structure.
   // This `mapping` assigns an unsigned integer (the token balance) to an address (the token holder).
   // Learn more: https://solidity.readthedocs.io/en/v0.5.10/types.html#mapping-types
   mapping (address => uint) public balances;

   // Events allow for logging of activity on the blockchain.
   // Ethereum clients can listen for events in order to react to contract state changes.
   // Learn more: https://solidity.readthedocs.io/en/v0.5.10/contracts.html#events
   event Transfer(address from, address to, uint amount);

   // Initializes the contract's data, setting the `owner`
   // to the address of the contract creator.
   constructor() public {
      // All smart contracts rely on external transactions to trigger its functions.
      // `msg` is a global variable that includes relevant data on the given transaction,
      // such as the address of the sender and the ETH value included in the transaction.
      // Learn more: https://solidity.readthedocs.io/en/v0.5.10/units-and-global-variables.html#block-and-transaction-properties
      owner = msg.sender;
   }

   // Creates an amount of new tokens and sends them to an address.
   function mint(address receiver, uint amount) public {
      // `require` is a control structure used to enforce certain conditions.
      // If a `require` statement evaluates to `false`, an exception is triggered,
      // which reverts all changes made to the state during the current call.
      // Learn more: https://solidity.readthedocs.io/en/v0.5.10/control-structures.html#error-handling-assert-require-revert-and-
exceptions

      // Only the contract owner can call this function
      require(msg.sender == owner, "You are not the owner.");

      // Enforces a maximum amount of tokens
      require(amount < 1e60, "Maximum issuance exceeded");

      // Increases the balance of `receiver` by `amount`
      balances[receiver] += amount;
   }

   // Sends an amount of existing tokens from any caller to an address.
   function transfer(address receiver, uint amount) public {
      // The sender must have enough tokens to send
      require(amount <= balances[msg.sender], "Insufficient balance.");

      // Adjusts token balances of the two addresses
      balances[msg.sender] -= amount;
      balances[receiver] += amount;

      // Emits the event defined earlier
      emit Transfer(msg.sender, receiver, amount);
   }
}
```

4.4 **Oracles**

4.4.1 Introduction

The network participants (nodes) validate and execute operations performed on the blockchain, such as smart contracts, but it is not uncommon for a smart contract to require data from external third parties. Given that blockchains cannot access data outside their networks, how is external data incorporated into the workflow? Here is where **Oracles** come in.



SC – smart contract

Till this place