[Solidity Docs - Types]

[Solidity docs - Functions]
[Solidity docs - Function Types]

[Difference between memory and storage - stackexchange]
[Storage, Memory and the Stack - Solidity Docs]
[Structure of a Contract - Solidity Docs]
[State Machine - Solidity Docs]
[Data location -- Solidity Docs]

[Reading Contracts Exercises]
[Greeter Contract tutorial - Ethereum Foundation]
[Solidity Docs - ABI]
[What is the ABI and why is it needed to interact with contracts - stackexchange]

****

```
contract ExampleContract {

event LogReturnValue(address indexed _from, int256 _value);

  function foo(int256 _value) returns (int256) {

   emit LogReturnValue(msg.sender, _value);

   return _value;

}

}
```

This is how you declare an event called `LogReturnValue`. Use the **event** keyword followed by the event name.

A recent update to the Solidity compiler requires the **'emit'** keyword before an event is logged. This helps further clarify when events are being logged in the contract code.

Declaring events require that you specify the parameter type as well as a name for the parameter. This parameter name is the name that will appear in the log, so make it descriptive and clear.

You can add the **indexed** keyword to event parameters. Up to three parameters can receive the indexed attribute. This increases the searchability of events. It is possible to filter for specific values of indexed arguments in the user interface.

Here is a link to the solidity documentation on events.

The common uses for events can be broken down into **three** main use cases:

1. Events can provide smart contract return values for the UI
2. They can act as asynchronous triggers with data and
3. They can act a cheaper form of storage.

When a transaction is sent via web3.js, a transaction hash is returned, even if the contract functions specifies a return value. Transactions don't return the contract

value because transactions are not immediately mined and included in the blockchain - it takes time. You can use an event in the contract function in conjunction with an event watcher in the UI to observe a variable value when the transaction is mined.

You can use an event watcher as a trigger for application logic beyond just reading return values. You could take another action, display a message or update the UI when an event is observed.

Logs, which are essentially the same as events (the context dictates which term is more appropriate) can also be used as a cheaper form of storage. Logs cost 8 gas per byte whereas contract storage costs 20,000 per 32 bytes, or 625 gas per byte. Logs are cheaper, but also cannot be accessed from any contracts. Even still, logs can be useful for aggregating historical reference data.

Events are inheritable members of contracts. You can call events of parent contracts from within child contracts.

When called from within a contract, events cause arguments to be stored in the transaction log, a special data structure in the blockchain. The logs are associated with the address of the contract and will be incorporated into the blockchain, persisting as long as a block is accessible. In the current version of Ethereum (Byzantium), blocks are assumed to be accessible forever, but this may change in future versions, like Serenity, where earlier blocks may not always be available.

Log and event data are not accessible from within contracts, not even from the contract that created the log. Events are useful when interacting with an application. They can provide notifications and confirmations of transactions happening in the smart contract. They are also useful for debugging purposes during development.

Logs are not part of the blockchain per se since they are not required for consensus (they are just historical data), but they are verified by the blockchain as the transaction receipt hashes are stored inside the blocks.

Remember that events are not emitted until the transaction has been successfully mined.

Logging an event for every state change of the contract is a good heuristic for when you should use events. This allows you to track any and all updates to the state of the contract by setting up event watchers in your javascript files.

Useful links:

Technical introduction to Events and Logs in Ethereum by Joseph Chow, Consensys Media

Events - Solidity Docs

****

Solidity is a contract-oriented programming language, which adopts many of the design principles of object-oriented programming languages.

We can implement a factory design pattern that will ensure that every contract deployed using the factory adheres to a certain standard.

Let’s look at what a standard Ethereum token implementation might look like.

Tokens are a great case for a factory pattern because we want all of the token implementations to be compatible and convertible with one another.

Defining a standard interface that all tokens should implement can ease development and benefit the entire ecosystem.

```
pragma solidity ^0.4.18;

contract Token {

    mapping(address => uint256) balances;
    uint256 totalSupply;

    function Token(
        uint256 _initialAmount
        ) public {
            balances[msg.sender] = _initialAmount;
            totalSupply = _initialAmount;
        }

    function transfer(
        address _to,
        uint256 amount
        ) public returns (bool) {
            require(balances[msg.sender] >= amount);
            balances[msg.sender] -= amount;
            balances[_to] += amount;
            return true;
        }

    ...

```

Here is the start of a basic Token contract. You can imagine that we might want to store more data in our Token contract. This contract does not have functions implemented yet.

As we add functions and the contract increases in complexity, it will be more difficult to ensure that every Token contract is implementing the same interface and is bug-free.

Deploying all of these standard compliant tokens through a token factory will abstract away many of the implementation details.

We can use the factory design pattern in solidity.

In the factory contract, we need to make the standardized token contract available by importing it.

```
pragma solidity ^0.4.18;

import "./Token.sol";

contract TokenFactory {

    mapping(address => address[]) created;

    function createToken(uint256 _initialAmount)
        public
        returns(address)
    {
        Token newToken = new Token(_initialAmount);
        created[msg.sender].push(address(newToken));
        newToken.transfer(msg.sender, _initialAmount);
        return address(newToken);
    }

}
```

When we want to create a new token, we can pass the required constructor arguments for the Token contract into the token factory create token function.

In the create token function we specify that we are creating a new Token contract with the new keyword.

The token contract creation will return the address of the new contract. We know that this contract is of type token and we specify the type of the newToken variable.

We can store the creator of the new token, and transfer all of the newly minted tokens to the caller.

The function returns the address of the new token contract. This new contract has all of the state variables and functions that we specified in the token contract.

This is a useful design pattern that has numerous potential use cases.

The ConsenSys Github contains an implementation of the EIP20 interface, as well as an EIP20 token factory.

Useful links:

Manage several contracts with Factories - Jules Dourlens