**Interfaces**

An interface is a set of function definitions used to enable communication between smart contracts. A contract interface defines all of that contract’s externally available functions. By importing the interface, your contract now knows how to call these functions in other contracts.

**Declaring and using Interfaces**

Interfaces can be added to contracts either through inline definition, or by importing them from a separate file.

The interface keyword is used to define an inline external interface:

interface FooBar:

**def** calculate() -> uint256: view

**def** test1(): nonpayable

The defined interface can then be used to make external calls, given a contract address:

**@external**

**def** test(foobar: FooBar):

foobar.calculate()

The interface name can also be used as a type annotation for storage variables. You then assign an address value to the variable to access that interface. Note that casting an address to an interface is possible, e.g. FooBar(<address\_var>):

foobar\_contract: FooBar

**@external**

**def** \_\_init\_\_(foobar\_address: address):

self.foobar\_contract = FooBar(foobar\_address)

**@external**

**def** test():

self.foobar\_contract.calculate()

Specifying payable or nonpayable annotation indicates that the call made to the external contract will be able to alter storage, whereas the view pure call will use a STATICCALL ensuring no storage can be altered during execution. Additionally, payable allows non-zero value to be sent along with the call.

interface FooBar:

**def** calculate() -> uint256: pure

**def** query() -> uint256: view

**def** update(): nonpayable

**def** pay(): payable

**@external**

**def** test(foobar: FooBar):

foobar.calculate() *# cannot change storage*

foobar.query() *# cannot change storage, but reads itself*

foobar.update() *# storage can be altered*

foobar.pay(value=1) *# storage can be altered, and value can be sent*

Vyper offers the option to set the following additional keyword arguments when making external calls:

| **Keyword** | **Description** |
| --- | --- |
| gas | Specify gas value for the call |
| value | Specify amount of ether sent with the call |
| skip\_contract\_check | Drop EXTCODESIZE and RETURNDATASIZE checks |
| default\_return\_value | Specify a default return value if no value is returned |

The default\_return\_value parameter can be used to handle ERC20 tokens affected by the missing return value bug in a way similar to OpenZeppelin’s safeTransfer for Solidity:

ERC20(USDT).transfer(msg.sender, 1, default\_return\_value=**True**) *# returns True*

ERC20(USDT).transfer(msg.sender, 1) *# reverts because nothing returned*

**Warning**

When skip\_contract\_check=True is used and the called function returns data (ex.: x: uint256 = SomeContract.foo(skip\_contract\_check=True), no guarantees are provided by the compiler as to the validity of the returned value. In other words, it is undefined behavior what happens if the called contract did not exist. In particular, the returned value might point to garbage memory. It is therefore recommended to only use skip\_contract\_check=True to call contracts which have been manually ensured to exist at the time of the call.

**Importing Interfaces**

Interfaces are imported with import or from ... import statements.

Imported interfaces are written using standard Vyper syntax. The body of each function is ignored when the interface is imported. If you are defining a standalone interface, it is normally specified by using a pass statement:

**@external**

**def** test1():

**pass**

**@external**

**def** calculate() -> uint256:

**pass**

You can also import a fully implemented contract and Vyper will automatically convert it to an interface. It is even possible for a contract to import itself to gain access to its own interface.

**import** **greeter** **as** **Greeter**

name: public(String[10])

**@external**

**def** \_\_init\_\_(\_name: String[10]):

self.name = \_name

**@view**

**@external**

**def** greet() -> String[16]:

**return** concat("Hello ", Greeter(msg.sender).name())

**Imports via import**

With absolute import statements, you **must** include an alias as a name for the imported package. In the following example, failing to include as Foo will raise a compile error:

**import** **contract.foo** **as** **Foo**

**Imports via from ... import**

Using from you can perform both absolute and relative imports. You may optionally include an alias - if you do not, the name of the interface will be the same as the file.

*# without an alias*

**from** **contract** **import** foo

*# with an alias*

**from** **contract** **import** foo **as** Foo

Relative imports are possible by prepending dots to the contract name. A single leading dot indicates a relative import starting with the current package. Two leading dots indicate a relative import from the parent of the current package:

**from** **.** **import** foo

**from** **..interfaces** **import** baz

**Searching For Interface Files**

When looking for a file to import, Vyper will first search relative to the same folder as the contract being compiled. For absolute imports, it also searches relative to the root path for the project. Vyper checks for the file name with a .vy suffix first, then .json.

When using the command line compiler, the root path defaults to the current working directory. You can change it with the -p flag:

$ vyper my\_project/contracts/my\_contract.vy -p my\_project

In the above example, the my\_project folder is set as the root path. A contract cannot perform a relative import that goes beyond the top-level folder.

**Built-in Interfaces**

Vyper includes common built-in interfaces such as [ERC20](https://eips.ethereum.org/EIPS/eip-20) and [ERC721](https://eips.ethereum.org/EIPS/eip-721). These are imported from vyper.interfaces:

**from** **vyper.interfaces** **import** ERC20

implements: ERC20

You can see all the available built-in interfaces in the [Vyper GitHub](https://github.com/vyperlang/vyper/tree/master/vyper/builtins/interfaces) repo.

**Implementing an Interface**

You can define an interface for your contract with the implements statement:

**import** **an\_interface** **as** **FooBarInterface**

implements: FooBarInterface

This imports the defined interface from the vyper file at an\_interface.vy (or an\_interface.json if using ABI json interface type) and ensures your current contract implements all the necessary external functions. If any interface functions are not included in the contract, it will fail to compile. This is especially useful when developing contracts around well-defined standards such as ERC20.

**Note**

Interfaces that implement functions with return values that require an upper bound (e.g. Bytes, DynArray, or String), the upper bound defined in the interface represents the lower bound of the implementation. Assuming a function my\_func returns a value String[1] in the interface, this would mean for the implementation function of my\_func that the return value must have **at least** length 1. This behavior might change in the future.

**Extracting Interfaces**

Vyper has a built-in format option to allow you to make your own Vyper interfaces easily.

$ vyper -f interface examples/voting/ballot.vy

# Functions

@view

@external

def delegated(addr: address) -> bool:

pass

# ...

If you want to do an external call to another contract, Vyper provides an external interface extract utility as well.

$ vyper -f external\_interface examples/voting/ballot.vy

# External Contracts

interface Ballot:

def delegated(addr: address) -> bool: view

def directlyVoted(addr: address) -> bool: view

def giveRightToVote(voter: address): nonpayable

def forwardWeight(delegate\_with\_weight\_to\_forward: address): nonpayable

# ...

The output can then easily be copy-pasted to be consumed.

# Event Logging

Vyper can log events to be caught and displayed by user interfaces.

## Example of Logging

This example is taken from the [sample ERC20 contract](https://github.com/vyperlang/vyper/blob/master/examples/tokens/ERC20.vy) and shows the basic flow of event logging:

*# Events of the token.*

event Transfer:

sender: indexed(address)

receiver: indexed(address)

value: uint256

event Approval:

owner: indexed(address)

spender: indexed(address)

value: uint256

*# Transfer some tokens from message sender to another address*

**def** transfer(\_to : address, \_value : uint256) -> bool:

... Logic here to do the real work ...

*# All done, log the event for listeners*

log Transfer(msg.sender, \_to, \_value)

Let’s look at what this is doing.

1. We declare two event types to log. The two events are similar in that they contain two indexed address fields. Indexed fields do not make up part of the event data itself, but can be searched by clients that want to catch the event. Also, each event contains one single data field, in each case called value. Events can contain several arguments with any names desired.
2. In the transfer function, after we do whatever work is necessary, we log the event. We pass three arguments, corresponding with the three arguments of the Transfer event declaration.

Clients listening to the events will declare and handle the events they are interested in using a [library such as web3.js](https://solidity.readthedocs.io/en/latest/contracts.html#events):

**var** abi = */\* abi as generated by the compiler \*/*;

**var** MyToken = web3.eth.contract(abi);

**var** myToken = MyToken.at("0x1234...ab67" */\* address \*/*);

*// watch for changes in the callback*

**var** event = myToken.Transfer(**function**(error, result) {

**if** (!error) {

**var** args = result.returnValues;

console.log('value transferred = ', args.\_amount);

}

});

In this example, the listening client declares the event to listen for. Any time the contract sends this log event, the callback will be invoked.

## Declaring Events

Let’s look at an event declaration in more detail.

event Transfer:

sender: indexed(address)

receiver: indexed(address)

value: uint256

Event declarations look similar to struct declarations, containing one or more arguments that are passed to the event. Typical events will contain two kinds of arguments:

* **Indexed** arguments, which can be searched for by listeners. Each indexed argument is identified by the indexed keyword. Here, each indexed argument is an address. You can have any number of indexed arguments, but indexed arguments are not passed directly to listeners, although some of this information (such as the sender) may be available in the listener’s results object.
* **Value** arguments, which are passed through to listeners. You can have any number of value arguments and they can have arbitrary names, but each is limited by the EVM to be no more than 32 bytes.

It is also possible to create an event with no arguments. In this case, use the pass statement:

event Foo: **pass**

## Logging Events

Once an event is declared, you can log (send) events. You can send events as many times as you want to. Please note that events sent do not take state storage and thus do not cost gas: this makes events a good way to save some information. However, the drawback is that events are not available to contracts, only to clients.

Logging events is done using the log statement:

log Transfer(msg.sender, \_to, \_amount)

The order and types of arguments given must match the order of arguments used when declaring the event.

## Listening for Events

In the example listener above, the result arg actually passes a [large amount of information](https://web3js.readthedocs.io/en/v1.2.6/web3-eth-contract.html#contract-events-return). Here we’re most interested in result.returnValues. This is an object with properties that match the properties declared in the event. Note that this object does not contain the indexed properties, which can only be searched in the original myToken.Transfer that created the callback.