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 and its functions need to be called for anything to happen. There is no “cron” concept in Ethereum to call a function at a particular event automatically.

Creating Contracts

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.

One way to create contracts programmatically on Ethereum is via the JavaScript API web3.js. It has a function called web3.eth.Contract to facilitate contract creation.

When a contract is created, its constructor (a function declared with the `constructor` keyword) is executed once.

A constructor is optional. Only one constructor is allowed, which means overloading is not supported.

After the constructor has executed, the final 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. The deployed code does not include the constructor code or internal functions only called from the constructor.

Internally, constructor arguments are passed ABI encoded after the code of the contract itself, but you do not have to care about this if you use `web3.js`.

If a contract wants to create another contract, the source code (and the binary) of the created contract has to be known to the creator. This means that cyclic creation dependencies are impossible.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;


contract OwnedToken {
    // `TokenCreator` is a contract type that is defined below.
    // It is fine to reference it as long as it is not used
    // to create a new contract.
    TokenCreator creator;
    address owner;
    bytes32 name;

    // This is the constructor which registers the
    // creator and the assigned name.
    constructor(bytes32 _name) {
        // State variables are accessed via their name
        // and not via e.g. `this.owner`. Functions can
        // be accessed directly or through `this.f`,
        // but the latter provides an external view
        // to the function. Especially in the constructor,
        // you should not access functions externally,
        // because the function does not exist yet.
        // See the next section for details.
        owner = msg.sender;

        // We perform an explicit type conversion from `address`
        // to `TokenCreator` and assume that the type of
        // the calling contract is `TokenCreator`, there is
        // no real way to verify that.
        // This does not create a new contract.
        creator = TokenCreator(msg.sender);
        name = _name;
    }

    function changeName(bytes32 newName) public {
        // Only the creator can alter the name.
        // We compare the contract based on its
        // address which can be retrieved by
        // explicit conversion to address.
        if (msg.sender == address(creator))
            name = newName;
    }

    function transfer(address newOwner) public {
        // Only the current owner can transfer the token.
        if (msg.sender != owner) return;

        // We ask the creator contract if the transfer
        // should proceed by using a function of the
        // `TokenCreator` contract defined below. If
        // the call fails (e.g. due to out-of-gas),
        // the execution also fails here.
        if (creator.isTokenTransferOK(owner, newOwner))
            owner = newOwner;
    }
}


contract TokenCreator {
    function createToken(bytes32 name)
        public
        returns (OwnedToken tokenAddress)
    {
        // Create a new `Token` contract and return its address.
        // From the JavaScript side, the return type
        // of this function is `address`, as this is
        // the closest type available in the ABI.
        return new OwnedToken(name);
```

```
    }

    function changeName(OwnedToken tokenAddress, bytes32 name) public {
        // Again, the external type of `tokenAddress` is
        // simply `address`.
        tokenAddress.changeName(name);
    }

    // Perform checks to determine if transferring a token to the
    // `OwnedToken` contract should proceed
    function isTokenTransferOK(address currentOwner, address newOwner)
        public
        pure
        returns (bool ok)
    {
        // Check an arbitrary condition to see if transfer should proceed
        return keccak256(abi.encodePacked(currentOwner, newOwner))[0] == 0x7f;
    }
}
```

Visibility and Getters

Solidity knows two kinds of function calls: internal ones that do not create an actual EVM call (also called a “message call”) and external ones that do. Because of that, there are four types of visibility for functions and state variables.

Functions have to be specified as being `external`, `public`, `internal` or `private`. For state variables, `external` is not possible.

`external`

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).

`public`

Public functions are part of the contract interface and can be either called internally or via messages. For public state variables, an automatic getter function (see below) is generated.

`internal`

Those functions and state variables can only be accessed internally (i.e. from within the current contract or contracts deriving from it), without using `this`. This is the default visibility level for state variables.

`private`

Private functions and state variables are only visible for the contract they are defined in and not in derived contracts.

Note

Everything that is inside a contract is visible to all observers external to the blockchain. Making something `private` only prevents other contracts from reading or modifying the information, but it will still be visible to the whole world outside of the blockchain.

The visibility specifier is given after the type for state variables and between parameter list and return parameter list for functions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f(uint a) private pure returns (uint b) { return a + 1; }
    function setData(uint a) internal { data = a; }
    uint public data;
}
```

In the following example, `D`, can call `c.getData()` to retrieve the value of `data` in state storage, but is not able to call `f`. Contract `E` is derived from `C` and, thus, can call `compute`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    uint private data;

    function f(uint a) private pure returns(uint b) { return a + 1; }
    function setData(uint a) public { data = a; }
    function getData() public view returns(uint) { return data; }
    function compute(uint a, uint b) internal pure returns (uint) { return a + b; }
}

// This will not compile
contract D {
    function readData() public {
        C c = new C();
        uint local = c.f(7); // error: member `f` is not visible
        c.setData(3);
        local = c.getData();
        local = c.compute(3, 5); // error: member `compute` is not visible
    }
}

contract E is C {
    function g() public {
        C c = new C();
        uint val = compute(3, 5); // access to internal member (from derived to parent contract)
    }
}
```

Getter Functions

The compiler automatically creates getter functions for all **public** state variables. For the contract given below, the compiler will generate a function called `data` that does not take any arguments and returns a `uint`, the value of the state variable `data`. State variables can be initialized when they are declared.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    uint public data = 42;
}

contract Caller {
    C c = new C();
    function f() public view returns (uint) {
        return c.data();
    }
}
```

The getter functions have external visibility. If the symbol is accessed internally (i.e. without `this.`), it evaluates to a state variable. If it is accessed externally (i.e. with `this.`), it evaluates to a function.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract C {
    uint public data;
    function x() public returns (uint) {
        data = 3; // internal access
        return this.data(); // external access
    }
}
```

If you have a `public` state variable of array type, then you can only retrieve single elements of the array via the generated getter function. This mechanism exists to avoid high gas costs when returning an entire array. You can use arguments to specify which individual element to return, for example `myArray(0)`. If you want to return an entire array in one call, then you need to write a function, for example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract arrayExample {
    // public state variable
    uint[] public myArray;

    // Getter function generated by the compiler
    /*
    function myArray(uint i) public view returns (uint) {
        return myArray[i];
    }
    */

    // function that returns entire array
    function getArray() public view returns (uint[] memory) {
        return myArray;
    }
}
```

Now you can use `getArray()` to retrieve the entire array, instead of `myArray(i)`, which returns a single element per call.

The next example is more complex:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Complex {
    struct Data {
        uint a;
        bytes3 b;
        mapping (uint => uint) map;
        uint[3] c;
        uint[] d;
        bytes e;
    }
    mapping (uint => mapping(bool => Data[])) public data;
}
```

It generates a function of the following form. The mapping and arrays (with the exception of byte arrays) in the struct are omitted because there is no good way to select individual struct members or provide a key for the mapping:

```
function data(uint arg1, bool arg2, uint arg3)
    public
    returns (uint a, bytes3 b, bytes memory e)
{
    a = data[arg1][arg2][arg3].a;
    b = data[arg1][arg2][arg3].b;
    e = data[arg1][arg2][arg3].e;
}
```

Function Modifiers

Modifiers can be used to change the behaviour of functions in a declarative way. For example, you can use a modifier to automatically check a condition prior to executing the function.

Modifiers are inheritable properties of contracts and may be overridden by derived contracts, but only if they are marked `virtual`. For details, please see Modifier Overriding.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

contract owned {
    constructor() { owner = payable(msg.sender); }
    address payable owner;

    // This contract only defines a modifier but does not use
    // it: it will be used in derived contracts.
    // The function body is inserted where the special symbol
    // `_;` in the definition of a modifier appears.
    // This means that if the owner calls this function, the
    // function is executed and otherwise, an exception is
    // thrown.
    modifier onlyOwner {
        require(
            msg.sender == owner,
            "Only owner can call this function."
        );
        _;
    }
}

contract destructible is owned {
    // This contract inherits the `onlyOwner` modifier from
    // `owned` and applies it to the `destroy` function, which
    // causes that calls to `destroy` only have an effect if
    // they are made by the stored owner.
    function destroy() public onlyOwner {
        selfdestruct(owner);
    }
}

contract priced {
    // Modifiers can receive arguments:
    modifier costs(uint price) {
        if (msg.value >= price) {
            _;
        }
    }
}

contract Register is priced, destructible {
    mapping (address => bool) registeredAddresses;
    uint price;

    constructor(uint initialPrice) { price = initialPrice; }

    // It is important to also provide the
    // `payable` keyword here, otherwise the function will
    // automatically reject all Ether sent to it.
    function register() public payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }

    function changePrice(uint _price) public onlyOwner {
        price = _price;
    }
}

contract Mutex {
    bool locked;
    modifier noReentrancy() {
        require(
            !locked,
            "Reentrant call."
        );
        locked = true;
```

```
        _;
        locked = false;
    }

    /// This function is protected by a mutex, which means that
    /// reentrant calls from within `msg.sender.call` cannot call `f` again.
    /// The `return 7` statement assigns 7 to the return value but still
    /// executes the statement `locked = false` in the modifier.
    function f() public noReentrancy returns (uint) {
        (bool success,) = msg.sender.call("");
        require(success);
        return 7;
    }
}
```

If you want to access a modifier `m` defined in a contract `C`, you can use `C.m` to reference it without virtual lookup. It is only possible to use modifiers defined in the current contract or its base contracts. Modifiers can also be defined in libraries but their use is limited to functions of the same library.

Multiple modifiers are applied to a function by specifying them in a whitespace-separated list and are evaluated in the order presented.

Modifiers cannot implicitly access or change the arguments and return values of functions they modify. Their values can only be passed to them explicitly at the point of invocation.

Explicit returns from a modifier or function body only leave the current modifier or function body. Return variables are assigned and control flow continues after the `_` in the preceding modifier.

Warning

In an earlier version of Solidity, `return` statements in functions having modifiers behaved differently.

An explicit return from a modifier with `return;` does not affect the values returned by the function. The modifier can, however, choose not to execute the function body at all and in that case the return variables are set to their default values just as if the function had an empty body.

The `_` symbol can appear in the modifier multiple times. Each occurrence is replaced with the function body.

Arbitrary expressions are allowed for modifier arguments and in this context, all symbols visible from the function are visible in the modifier. Symbols introduced in the modifier are not visible in the function (as they might change by overriding).

Constant and Immutable State Variables

State variables can be declared as `constant` or `immutable`. In both cases, the variables cannot be modified after the contract has been constructed. For `constant` variables, the value has to be fixed at compile-time, while for `immutable`, it can still be assigned at construction time.

It is also possible to define `constant` variables at the file level.

The compiler does not reserve a storage slot for these variables, and every occurrence is replaced by the respective value.

Compared to regular state variables, the gas costs of constant and immutable variables are much lower. For a constant variable, the expression assigned to it is copied to all the places where it is accessed and also re-evaluated each time. This allows for local optimizations. Immutable variables are evaluated once at construction time and their value is copied to all the places in the code where they are accessed. For these values, 32 bytes are reserved, even if they would fit in fewer bytes. Due to this, constant values can sometimes be cheaper than immutable values.

Not all types for constants and immutables are implemented at this time. The only supported types are strings (only for constants) and value types.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.4;

uint constant X = 32**22 + 8;

contract C {
    string constant TEXT = "abc";
    bytes32 constant MY_HASH = keccak256("abc");
    uint immutable decimals;
    uint immutable maxBalance;
    address immutable owner = msg.sender;

    constructor(uint _decimals, address _reference) {
        decimals = _decimals;
        // Assignments to immutables can even access the environment.
        maxBalance = _reference.balance;
    }

    function isBalanceTooHigh(address _other) public view returns (bool) {
        return _other.balance > maxBalance;
    }
}
```

Constant

For `constant` variables, the value has to be a constant at compile time and it has to be assigned where the variable is declared. Any expression that accesses storage, blockchain data (e.g. `block.timestamp`, `address(this).balance` or `block.number`) or execution data (`msg.value` or `gasleft()`) or makes calls to external contracts is disallowed. Expressions that might have a side-effect on memory allocation are allowed, but those that might have a side-effect on other memory objects are not. The built-in functions `keccak256`, `sha256`, `ripemd160`, `ecrecover`, `addmod` and `mulmod` are allowed (even though, with the exception of `keccak256`, they do call external contracts).

The reason behind allowing side-effects on the memory allocator is that it should be possible to construct complex objects like e.g. lookup-tables. This feature is not yet fully usable.

Immutable

Variables declared as `immutable` are a bit less restricted than those declared as `constant`: Immutable variables can be assigned an arbitrary value in the constructor of the contract or at the point of their declaration. They can be assigned only once and can, from that point on, be read even during construction time.

The contract creation code generated by the compiler will modify the contract's runtime code before it is returned by replacing all references to immutables by the values assigned to the them. This is important if you are comparing the runtime code generated by the compiler with the one actually stored in the blockchain.

> **Note**
>
> Immutables that are assigned at their declaration are only considered initialized once the constructor of the contract is executing. This means you cannot initialize immutables inline with a value that depends on another immutable. You can do this, however, inside the constructor of the contract.
>
> This is a safeguard against different interpretations about the order of state variable initialization and constructor execution, especially with regards to inheritance.

Functions

Functions can be defined inside and outside of contracts.

Functions outside of a contract, also called "free functions", always have implicit `internal` visibility. Their code is included in all contracts that call them, similar to internal library functions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

function sum(uint[] memory _arr) pure returns (uint s) {
    for (uint i = 0; i < _arr.length; i++)
        s += _arr[i];
}

contract ArrayExample {
    bool found;
    function f(uint[] memory _arr) public {
        // This calls the free function internally.
        // The compiler will add its code to the contract.
        uint s = sum(_arr);
        require(s >= 10);
        found = true;
    }
}
```

> **Note**
>
> Functions defined outside a contract are still always executed in the context of a contract. They still have access to the variable `this`, can call other contracts, send them Ether and destroy the contract that called them, among other things. The main difference to functions defined inside a

contract is that free functions do not have direct access to storage variables and functions not in their scope.

Function Parameters and Return Variables

Functions take typed parameters as input and may, unlike in many other languages, also return an arbitrary number of values as output.

Function Parameters

Function parameters are declared the same way as variables, and the name of unused parameters can be omitted.

For example, if you want your contract to accept one kind of external call with two integers, you would use something like the following:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Simple {
    uint sum;
    function taker(uint _a, uint _b) public {
        sum = _a + _b;
    }
}
```

Function parameters can be used as any other local variable and they can also be assigned to.

> **Note**
>
> An external function cannot accept a multi-dimensional array as an input parameter. This functionality is possible if you enable the ABI coder v2 by adding `pragma abicoder v2;` to your source file.
>
> An internal function can accept a multi-dimensional array without enabling the feature.

Return Variables

Function return variables are declared with the same syntax after the `returns` keyword.

For example, suppose you want to return two results: the sum and the product of two integers passed as function parameters, then you use something like:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Simple {
    function arithmetic(uint _a, uint _b)
        public
        pure
        returns (uint o_sum, uint o_product)
    {
        o_sum = _a + _b;
        o_product = _a * _b;
    }
}
```

The names of return variables can be omitted. Return variables can be used as any other local variable and they are initialized with their default value and have that value until they are (re-)assigned.

You can either explicitly assign to return variables and then leave the function as above, or you can provide return values (either a single or multiple ones) directly with the `return` statement:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Simple {
    function arithmetic(uint _a, uint _b)
        public
        pure
        returns (uint o_sum, uint o_product)
    {
        return (_a + _b, _a * _b);
    }
}
```

If you use an early `return` to leave a function that has return variables, you must provide return values together with the return statement.

Note

You cannot return some types from non-internal functions, notably multi-dimensional dynamic arrays and structs. If you enable the ABI coder v2 by adding `pragma abicoder v2;` to your source file then more types are available, but `mapping` types are still limited to inside a single contract and you cannot transfer them.

Returning Multiple Values

When a function has multiple return types, the statement `return (v0, v1, ..., vn)` can be used to return multiple values. The number of components must be the same as the number of return variables and their types have to match, potentially after an implicit conversion.

State Mutability

View Functions

Functions can be declared `view` in which case they promise not to modify the state.

> **Note**
>
> If the compiler's EVM target is Byzantium or newer (default) the opcode `STATICCALL` is used when `view` functions are called, which enforces the state to stay unmodified as part of the EVM execution. For library `view` functions `DELEGATECALL` is used, because there is no combined `DELEGATECALL` and `STATICCALL`. This means library `view` functions do not have run-time checks that prevent state modifications. This should not impact security negatively because library code is usually known at compile-time and the static checker performs compile-time checks.

The following statements are considered modifying the state:

1. Writing to state variables.
2. Emitting events.
3. Creating other contracts.
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.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    function f(uint a, uint b) public view returns (uint) {
        return a * (b + 42) + block.timestamp;
    }
}
```

> **Note**
>
> `constant` on functions used to be an alias to `view`, but this was dropped in version 0.5.0.

> **Note**
>
> Getter methods are automatically marked `view`.

> **Note**
>
> Prior to version 0.5.0, the compiler did not use the `STATICCALL` opcode for `view` functions. This enabled state modifications in `view` functions through the use of invalid explicit type conversions. By using `STATICCALL` for `view` functions, modifications to the state are prevented on the level of the EVM.

Pure Functions

Functions can be declared `pure` in which case they promise not to read from or modify the state. In particular, it should be possible to evaluate a `pure` function at compile-time given only its inputs and `msg.data`, but without any knowledge of the current blockchain state. This means that reading from `immutable` variables can be a non-pure operation.

> **Note**
>
> If the compiler's EVM target is Byzantium or newer (default) the opcode `STATICCALL` is used, which does not guarantee that the state is not read, but at least that it is not modified.

In addition to the list of state modifying statements explained above, the following are considered reading from the state:

1. Reading from state variables.
2. Accessing `address(this).balance` or `<address>.balance`.
3. Accessing any of the members of `block`, `tx`, `msg` (with the exception of `msg.sig` and `msg.data`).
4. Calling any function not marked `pure`.
5. Using inline assembly that contains certain opcodes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    function f(uint a, uint b) public pure returns (uint) {
        return a * (b + 42);
    }
}
```

Pure functions are able to use the `revert()` and `require()` functions to revert potential state changes when an error occurs.

Reverting a state change is not considered a "state modification", as only changes to the state made previously in code that did not have the `view` or `pure` restriction are reverted and that code has the option to catch the `revert` and not pass it on.

This behaviour is also in line with the `STATICCALL` opcode.

> **Warning**
>
> It is not possible to prevent functions from reading the state at the level of the EVM, it is only possible to prevent them from writing to the state (i.e. only `view` can be enforced at the EVM level, `pure` can not).

> **Note**

Prior to version 0.5.0, the compiler did not use the `STATICCALL` opcode for `pure` functions. This enabled state modifications in `pure` functions through the use of invalid explicit type conversions. By using `STATICCALL` for `pure` functions, modifications to the state are prevented on the level of the EVM.

Note

Prior to version 0.4.17 the compiler did not enforce that `pure` is not reading the state. It is a compile-time type check, which can be circumvented doing invalid explicit conversions between contract types, because the compiler can verify that the type of the contract does not do state-changing operations, but it cannot check that the contract that will be called at runtime is actually of that type.

Special Functions

Receive Ether Function

A contract can have at most one `receive` function, declared using `receive() external payable { ... }` (without the `function` keyword). This function cannot have arguments, cannot return anything and must have `external` visibility and `payable` state mutability. It can be virtual, can override and can have modifiers.

The receive function is executed on a call to the contract with empty calldata. This is the function that is executed on plain Ether transfers (e.g. via `.send()` or `.transfer()`). If no such function exists, but a payable fallback function exists, the fallback function will be called on a plain Ether transfer. If neither a receive Ether nor a payable fallback function is present, the contract cannot receive Ether through regular transactions and throws an exception.

In the worst case, the `receive` function can only rely on 2300 gas being available (for example when `send` or `transfer` is used), leaving little room to perform other operations except basic logging. The following operations will consume more gas than the 2300 gas stipend:

- Writing to storage
- Creating a contract
- Calling an external function which consumes a large amount of gas
- Sending Ether

Warning

Contracts that receive Ether directly (without a function call, i.e. using `send` or `transfer`) but do not define a receive Ether function or a payable fallback function throw an exception, sending back the Ether (this was different before Solidity v0.4.0). So if you want your contract to receive Ether, you have to implement a receive Ether function (using payable fallback functions for receiving Ether is not recommended, since it would not fail on interface confusions).

Warning

A contract without a receive Ether function can receive Ether as a recipient of a *coinbase transaction* (aka *miner block reward*) or as a destination of a `selfdestruct`.

A contract cannot react to such Ether transfers and thus also cannot reject them. This is a design choice of the EVM and Solidity cannot work around it.

It also means that `address(this).balance` can be higher than the sum of some manual accounting implemented in a contract (i.e. having a counter updated in the receive Ether function).

Below you can see an example of a Sink contract that uses function `receive`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

// This contract keeps all Ether sent to it with no way
// to get it back.
contract Sink {
    event Received(address, uint);
    receive() external payable {
        emit Received(msg.sender, msg.value);
    }
}
```

Fallback Function

A contract can have at most one `fallback` function, declared using either `fallback () external [payable]` or `fallback (bytes calldata _input) external [payable] returns (bytes memory _output)` (both without the `function` keyword). This function must have `external` visibility. A fallback function can be virtual, can override and can have modifiers.

The fallback function is executed on a call to the contract if none of the other functions match the given function signature, or if no data was supplied at all and there is no receive Ether function. The fallback function always receives data, but in order to also receive Ether it must be marked `payable`.

If the version with parameters is used, `_input` will contain the full data sent to the contract (equal to `msg.data`) and can return data in `_output`. The returned data will not be ABI-encoded. Instead it will be returned without modifications (not even padding).

In the worst case, if a payable fallback function is also used in place of a receive function, it can only rely on 2300 gas being available (see receive Ether function for a brief description of the implications of this).

Like any function, the fallback function can execute complex operations as long as there is enough gas passed on to it.

Warning

A `payable` fallback function is also executed for plain Ether transfers, if no receive Ether function is present. It is recommended to always define a receive Ether function as well, if you define a payable fallback function to distinguish Ether transfers from interface confusions.

Note

If you want to decode the input data, you can check the first four bytes for the function selector and then you can use `abi.decode` together with the array slice syntax to decode ABI-encoded data: `(c, d) = abi.decode(_input[4:], (uint256, uint256));` Note that this should only be used as a last resort and proper functions should be used instead.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

contract Test {
    uint x;
    // This function is called for all messages sent to
    // this contract (there is no other function).
    // Sending Ether to this contract will cause an exception,
    // because the fallback function does not have the `payable`
    // modifier.
    fallback() external { x = 1; }
}

contract TestPayable {
    uint x;
    uint y;
    // This function is called for all messages sent to
    // this contract, except plain Ether transfers
    // (there is no other function except the receive function).
    // Any call with non-empty calldata to this contract will execute
    // the fallback function (even if Ether is sent along with the call).
    fallback() external payable { x = 1; y = msg.value; }

    // This function is called for plain Ether transfers, i.e.
    // for every call with empty calldata.
    receive() external payable { x = 2; y = msg.value; }
}

contract Caller {
    function callTest(Test test) public returns (bool) {
        (bool success,) = address(test).call(abi.encodeWithSignature("nonExistingFunction()"));
        require(success);
        // results in test.x becoming == 1.

        // address(test) will not allow to call ``send`` directly, since ``test`` has no payable
        // fallback function.
        // It has to be converted to the ``address payable`` type to even allow calling ``send`` on
it.
        address payable testPayable = payable(address(test));

        // If someone sends Ether to that contract,
        // the transfer will fail, i.e. this returns false here.
        return testPayable.send(2 ether);
    }

    function callTestPayable(TestPayable test) public returns (bool) {
        (bool success,) = address(test).call(abi.encodeWithSignature("nonExistingFunction()"));
        require(success);
        // results in test.x becoming == 1 and test.y becoming 0.
        (success,) = address(test).call{value: 1}(abi.encodeWithSignature("nonExistingFunction()"));
        require(success);
        // results in test.x becoming == 1 and test.y becoming 1.

        // If someone sends Ether to that contract, the receive function in TestPayable will be
called.
        // Since that function writes to storage, it takes more gas than is available with a
        // simple ``send`` or ``transfer``. Because of that, we have to use a low-level call.
        (success,) = address(test).call{value: 2 ether}("");
        require(success);
        // results in test.x becoming == 2 and test.y becoming 2 ether.

        return true;
    }
}
```

Function Overloading

A contract can have multiple functions of the same name but with different parameter types. This process is called “overloading” and also applies to inherited functions. The following example shows overloading of the function `f` in the scope of contract `A`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract A {
    function f(uint _in) public pure returns (uint out) {
        out = _in;
    }

    function f(uint _in, bool _really) public pure returns (uint out) {
        if (_really)
            out = _in;
    }
}
```

Overloaded functions are also present in the external interface. It is an error if two externally visible functions differ by their Solidity types but not by their external types.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

// This will not compile
contract A {
    function f(B _in) public pure returns (B out) {
        out = _in;
    }

    function f(address _in) public pure returns (address out) {
        out = _in;
    }
}

contract B {
}
```

Both `f` function overloads above end up accepting the address type for the ABI although they are considered different inside Solidity.

Overload resolution and Argument matching

Overloaded functions are selected by matching the function declarations in the current scope to the arguments supplied in the function call. Functions are selected as overload candidates if all arguments can be implicitly converted to the expected types. If there is not exactly one candidate, resolution fails.

> **Note**
>
> Return parameters are not taken into account for overload resolution.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract A {
    function f(uint8 _in) public pure returns (uint8 out) {
        out = _in;
    }

    function f(uint256 _in) public pure returns (uint256 out) {
        out = _in;
    }
}
```

Calling `f(50)` would create a type error since `50` can be implicitly converted both to `uint8` and `uint256` types. On another hand `f(256)` would resolve to `f(uint256)` overload as `256` cannot be implicitly converted to `uint8`.

Events

Solidity events give an abstraction on top of the EVM's logging functionality. Applications can subscribe and listen to these events through the RPC interface of an Ethereum client.

Events are inheritable members of contracts. When you call them, they cause the arguments to be stored in the transaction's log - a special data structure in the blockchain. These logs are associated with the address of the contract, are incorporated into the blockchain, and stay there as long as a block is accessible (forever as of now, but this might change with Serenity). The Log and its event data is not accessible from within contracts (not even from the contract that created them).

It is possible to request a Merkle proof for logs, so if an external entity supplies a contract with such a proof, it can check that the log actually exists inside the blockchain. You have to supply block headers because the contract can only see the last 256 block hashes.

You can add the attribute `indexed` to up to three parameters which adds them to a special data structure known as "topics" instead of the data part of the log. A topic can only hold a single word (32 bytes) so if you use a reference type for an indexed argument, the Keccak-256 hash of the value is stored as a topic instead.

All parameters without the `indexed` attribute are ABI-encoded into the data part of the log.

Topics allow you to search for events, for example when filtering a sequence of blocks for certain events. You can also filter events by the address of the contract that emitted the event.

For example, the code below uses the web3.js `subscribe("logs")` method to filter logs that match a topic with a certain address value:

```
var options = {
    fromBlock: 0,
    address: web3.eth.defaultAccount,
    topics: ["0x0000000000000000000000000000000000000000000000000000000000000000", null, null]
};
web3.eth.subscribe('logs', options, function (error, result) {
    if (!error)
        console.log(result);
})
    .on("data", function (log) {
        console.log(log);
    })
    .on("changed", function (log) {
});
```

The hash of the signature of the event is one of the topics, except if you declared the event with the `anonymous` specifier. This means that it is not possible to filter for specific anonymous events by name, you can only filter by the contract address. The advantage of anonymous events is that they are cheaper to deploy and call. It also allows you to declare four indexed arguments rather than three.

> **Note**
>
> Since the transaction log only stores the event data and not the type, you have to know the type of the event, including which parameter is indexed and if the event is anonymous in order to correctly interpret the data. In particular, it is possible to "fake" the signature of another event using an anonymous event.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.21 <0.9.0;

contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );

    function deposit(bytes32 _id) public payable {
        // Events are emitted using `emit`, followed by
        // the name of the event and the arguments
        // (if any) in parentheses. Any such invocation
        // (even deeply nested) can be detected from
        // the JavaScript API by filtering for `Deposit`.
        emit Deposit(msg.sender, _id, msg.value);
    }
}
```

The use in the JavaScript API is as follows:

```
var abi = /* abi as generated by the compiler */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at("0x1234...ab67" /* address */);

var depositEvent = clientReceipt.Deposit();

// watch for changes
depositEvent.watch(function(error, result){
    // result contains non-indexed arguments and topics
    // given to the `Deposit` call.
    if (!error)
        console.log(result);
});


// Or pass a callback to start watching immediately
var depositEvent = clientReceipt.Deposit(function(error, result) {
    if (!error)
        console.log(result);
});
```

The output of the above looks like the following (trimmed):

```
{
   "returnValues": {
       "_from": "0x1111…FFFFCCCC",
       "_id": "0x50…sd5adb20",
       "_value": "0x420042"
   },
   "raw": {
       "data": "0x7f…91385",
       "topics": ["0xfd4…b4ead7", "0x7f…1a91385"]
   }
}
```

Additional Resources for Understanding Events

- Javascript documentation
- Example usage of events
- How to access them in js

Errors and the Revert Statement

Errors in Solidity provide a convenient and gas-efficient way to explain to the user why an operation failed. They can be defined inside and outside of contracts (including interfaces and libraries).

They have to be used together with the revert statement which causes all changes in the current call to be reverted and passes the error data back to the caller.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

/// Insufficient balance for transfer. Needed `required` but only
/// `available` available.
/// @param available balance available.
/// @param required requested amount to transfer.
error InsufficientBalance(uint256 available, uint256 required);

contract TestToken {
    mapping(address => uint) balance;
    function transfer(address to, uint256 amount) public {
        if (amount > balance[msg.sender])
            revert InsufficientBalance({
                available: balance[msg.sender],
                required: amount
            });
        balance[msg.sender] -= amount;
        balance[to] += amount;
    }
    // ...
}
```

Errors cannot be overloaded or overridden but are inherited. The same error can be defined in multiple places as long as the scopes are distinct. Instances of errors can only be created using `revert` statements.

The error creates data that is then passed to the caller with the revert operation to either return to the off-chain component or catch it in a try/catch statement. Note that an error can only be caught when coming from an external call, reverts happening in internal calls or inside the same function cannot be caught.

If you do not provide any parameters, the error only needs four bytes of data and you can use NatSpec as above to further explain the reasons behind the error, which is not stored on chain. This makes this a very cheap and convenient error-reporting feature at the same time.

More specifically, an error instance is ABI-encoded in the same way as a function call to a function of the same name and types would be and then used as the return data in the `revert` opcode. This means that the data consists of a 4-byte selector followed by ABI-encoded data. The selector consists of the first four bytes of the keccak256-hash of the signature of the error type.

Note

It is possible for a contract to revert with different errors of the same name or even with errors defined in different places that are indistinguishable by the caller. For the outside, i.e. the ABI, only the name of the error is relevant, not the contract or file where it is defined.

The statement `require(condition, "description");` would be equivalent to `if (!condition) revert Error("description")` if you could define `error Error(string)`. Note, however, that `Error` is a built-in type and cannot be defined in user-supplied code.

Similarly, a failing `assert` or similar conditions will revert with an error of the built-in type `Panic(uint256)`.

> **Note**
>
> Error data should only be used to give an indication of failure, but not as a means for control-flow. The reason is that the revert data of inner calls is propagated back through the chain of external calls by default. This means that an inner call can "forge" revert data that looks like it could have come from the contract that called it.

Inheritance

Solidity supports multiple inheritance including polymorphism.

Polymorphism means that a function call (internal and external) always executes the function of the same name (and parameter types) in the most derived contract in the inheritance hierarchy. This has to be explicitly enabled on each function in the hierarchy using the `virtual` and `override` keywords. See Function Overriding for more details.

It is possible to call functions further up in the inheritance hierarchy internally by explicitly specifying the contract using `ContractName.functionName()` or using `super.functionName()` if you want to call the function one level higher up in the flattened inheritance hierarchy (see below).

When a contract inherits from other contracts, only a single contract is created on the blockchain, and the code from all the base contracts is compiled into the created contract. This means that all internal calls to functions of base contracts also just use internal function calls (`super.f(..)` will use JUMP and not a message call).

State variable shadowing is considered as an error. A derived contract can only declare a state variable `x`, if there is no visible state variable with the same name in any of its bases.

The general inheritance system is very similar to Python's, especially concerning multiple inheritance, but there are also some differences.

Details are given in the following example.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;


contract Owned {
    constructor() { owner = payable(msg.sender); }
    address payable owner;
}


// Use `is` to derive from another contract. Derived
// contracts can access all non-private members including
// internal functions and state variables. These cannot be
// accessed externally via `this`, though.
contract Destructible is Owned {
    // The keyword `virtual` means that the function can change
    // its behaviour in derived classes ("overriding").
    function destroy() virtual public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}


// These abstract contracts are only provided to make the
// interface known to the compiler. Note the function
// without body. If a contract does not implement all
// functions it can only be used as an interface.
abstract contract Config {
    function lookup(uint id) public virtual returns (address adr);
}


abstract contract NameReg {
    function register(bytes32 name) public virtual;
    function unregister() public virtual;
}


// Multiple inheritance is possible. Note that `owned` is
// also a base class of `Destructible`, yet there is only a single
// instance of `owned` (as for virtual inheritance in C++).
contract Named is Owned, Destructible {
    constructor(bytes32 name) {
        Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
        NameReg(config.lookup(1)).register(name);
    }

    // Functions can be overridden by another function with the same name and
    // the same number/types of inputs.  If the overriding function has different
    // types of output parameters, that causes an error.
    // Both local and message-based function calls take these overrides
    // into account.
    // If you want the function to override, you need to use the
    // `override` keyword. You need to specify the `virtual` keyword again
    // if you want this function to be overridden again.
    function destroy() public virtual override {
        if (msg.sender == owner) {
            Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
            NameReg(config.lookup(1)).unregister();
            // It is still possible to call a specific
            // overridden function.
            Destructible.destroy();
        }
    }
}


// If a constructor takes an argument, it needs to be
```

```
// provided in the header or modifier-invocation-style at
// the constructor of the derived contract (see below).
contract PriceFeed is Owned, Destructible, Named("GoldFeed") {
    function updateInfo(uint newInfo) public {
        if (msg.sender == owner) info = newInfo;
    }

    // Here, we only specify `override` and not `virtual`.
    // This means that contracts deriving from `PriceFeed`
    // cannot change the behaviour of `destroy` anymore.
    function destroy() public override(Destructible, Named) { Named.destroy(); }
    function get() public view returns(uint r) { return info; }

    uint info;
}
```

Note that above, we call `Destructible.destroy()` to "forward" the destruction request. The way this is done is problematic, as seen in the following example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract owned {
    constructor() { owner = payable(msg.sender); }
    address payable owner;
}

contract Destructible is owned {
    function destroy() public virtual {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is Destructible {
    function destroy() public virtual override { /* do cleanup 1 */ Destructible.destroy(); }
}

contract Base2 is Destructible {
    function destroy() public virtual override { /* do cleanup 2 */ Destructible.destroy(); }
}

contract Final is Base1, Base2 {
    function destroy() public override(Base1, Base2) { Base2.destroy(); }
}
```

A call to `Final.destroy()` will call `Base2.destroy` because we specify it explicitly in the final override, but this function will bypass `Base1.destroy`. The way around this is to use `super`:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract owned {
    constructor() { owner = payable(msg.sender); }
    address payable owner;
}

contract Destructible is owned {
    function destroy() virtual public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is Destructible {
    function destroy() public virtual override { /* do cleanup 1 */ super.destroy(); }
}


contract Base2 is Destructible {
    function destroy() public virtual override { /* do cleanup 2 */ super.destroy(); }
}

contract Final is Base1, Base2 {
    function destroy() public override(Base1, Base2) { super.destroy(); }
}
```

If `Base2` calls a function of `super`, it does not simply call this function on one of its base contracts. Rather, it calls this function on the next base contract in the final inheritance graph, so it will call `Base1.destroy()` (note that the final inheritance sequence is – starting with the most derived contract: Final, Base2, Base1, Destructible, owned). The actual function that is called when using super is not known in the context of the class where it is used, although its type is known. This is similar for ordinary virtual method lookup.

Function Overriding

Base functions can be overridden by inheriting contracts to change their behavior if they are marked as `virtual`. The overriding function must then use the `override` keyword in the function header. The overriding function may only change the visibility of the overridden function from `external` to `public`. The mutability may be changed to a more strict one following the order: `nonpayable` can be overridden by `view` and `pure`. `view` can be overridden by `pure`. `payable` is an exception and cannot be changed to any other mutability.

The following example demonstrates changing mutability and visibility:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base
{
    function foo() virtual external view {}
}

contract Middle is Base {}

contract Inherited is Middle
{
    function foo() override public pure {}
}
```

For multiple inheritance, the most derived base contracts that define the same function must be specified explicitly after the `override` keyword. In other words, you have to specify all base contracts that define the same function and have not yet been overridden by another base contract (on some path through the inheritance graph). Additionally, if a contract inherits the same function from multiple (unrelated) bases, it has to explicitly override it:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Base1
{
    function foo() virtual public {}
}

contract Base2
{
    function foo() virtual public {}
}

contract Inherited is Base1, Base2
{
    // Derives from multiple bases defining foo(), so we must explicitly
    // override it
    function foo() public override(Base1, Base2) {}
}
```

An explicit override specifier is not required if the function is defined in a common base contract or if there is a unique function in a common base contract that already overrides all other functions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract A { function f() public pure{} }
contract B is A {}
contract C is A {}
// No explicit override required
contract D is B, C {}
```

More formally, it is not required to override a function (directly or indirectly) inherited from multiple bases if there is a base contract that is part of all override paths for the signature, and (1) that base implements the function and no paths from the current contract to the base mentions a function with that signature or (2) that base does not implement the function and there is at most one mention of the function in all paths from the current contract to that base.

In this sense, an override path for a signature is a path through the inheritance graph that starts at the contract under consideration and ends at a contract mentioning a function with that signature that does not override.

If you do not mark a function that overrides as `virtual`, derived contracts can no longer change the behaviour of that function.

Note

Functions with the `private` visibility cannot be `virtual`.

Note

Functions without implementation have to be marked `virtual` outside of interfaces. In interfaces, all functions are automatically considered `virtual`.

Note

Starting from Solidity 0.8.8, the `override` keyword is not required when overriding an interface function, except for the case where the function is defined in multiple bases.

Public state variables can override external functions if the parameter and return types of the function matches the getter function of the variable:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract A
{
    function f() external view virtual returns(uint) { return 5; }
}

contract B is A
{
    uint public override f;
}
```

Note

While public state variables can override external functions, they themselves cannot be overridden.

Modifier Overriding

Function modifiers can override each other. This works in the same way as function overriding (except that there is no overloading for modifiers). The `virtual` keyword must be used on the overridden modifier and the `override` keyword must be used in the overriding modifier:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Base
{
    modifier foo() virtual {_;}
}

contract Inherited is Base
{
    modifier foo() override {_;}
}
```

In case of multiple inheritance, all direct base contracts must be specified explicitly:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Base1
{
    modifier foo() virtual {_;}
}

contract Base2
{
    modifier foo() virtual {_;}
}

contract Inherited is Base1, Base2
{
    modifier foo() override(Base1, Base2) {_;}
}
```

Constructors

A constructor is an optional function declared with the `constructor` keyword which is executed upon contract creation, and where you can run contract initialisation code.

Before the constructor code is executed, state variables are initialised to their specified value if you initialise them inline, or their default value if you do not.

After the constructor has run, the final code of the contract is deployed to the blockchain. The deployment of the code costs additional gas linear to the length of the code. This code includes all functions that are part of the public interface and all functions that are reachable from there through function calls. It does not include the constructor code or internal functions that are only called from the constructor.

If there is no constructor, the contract will assume the default constructor, which is equivalent to `constructor() {}`. For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

abstract contract A {
    uint public a;

    constructor(uint _a) {
        a = _a;
    }
}

contract B is A(1) {
    constructor() {}
}
```

You can use internal parameters in a constructor (for example storage pointers). In this case, the contract has to be marked abstract, because these parameters cannot be assigned valid values from outside but only through the constructors of derived contracts.

Warning

Prior to version 0.4.22, constructors were defined as functions with the same name as the contract. This syntax was deprecated and is not allowed anymore in version 0.5.0.

Warning

Prior to version 0.7.0, you had to specify the visibility of constructors as either `internal` or `public`.

Arguments for Base Constructors

The constructors of all the base contracts will be called following the linearization rules explained below. If the base constructors have arguments, derived contracts need to specify all of them. This can be done in two ways:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base {
    uint x;
    constructor(uint _x) { x = _x; }
}

// Either directly specify in the inheritance list...
contract Derived1 is Base(7) {
    constructor() {}
}

// or through a "modifier" of the derived constructor.
contract Derived2 is Base {
    constructor(uint _y) Base(_y * _y) {}
}
```

One way is directly in the inheritance list (`is Base(7)`). The other is in the way a modifier is invoked as part of the derived constructor (`Base(_y * _y)`). The first way to do it is more convenient if the constructor argument is a constant and defines the behaviour of the contract or describes it. The second way has to be used if the constructor arguments of the base depend on those of the derived contract. Arguments have to be given either in the inheritance list or in modifier-style in the derived constructor. Specifying arguments in both places is an error.

If a derived contract does not specify the arguments to all of its base contracts' constructors, it will be abstract.

Multiple Inheritance and Linearization

Languages that allow multiple inheritance have to deal with several problems. One is the Diamond Problem. Solidity is similar to Python in that it uses "C3 Linearization" to force a specific order in the directed acyclic graph (DAG) of base classes. This results in the desirable property of monotonicity but disallows some inheritance graphs. Especially, the order in which the base classes are given in the `is` directive is important: You have to list the direct base contracts in the order from "most base-like" to "most derived". Note that this order is the reverse of the one used in Python.

Another simplifying way to explain this is that when a function is called that is defined multiple times in different contracts, the given bases are searched from right to left (left to right in Python) in a depth-first manner, stopping at the first match. If a base contract has already been searched, it is skipped.

In the following code, Solidity will give the error "Linearization of inheritance graph impossible".

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract X {}
contract A is X {}
// This will not compile
contract C is A, X {}
```

The reason for this is that `C` requests `X` to override `A` (by specifying `A, X` in this order), but `A` itself requests to override `X`, which is a contradiction that cannot be resolved.

Due to the fact that you have to explicitly override a function that is inherited from multiple bases without a unique override, C3 linearization is not too important in practice.

One area where inheritance linearization is especially important and perhaps not as clear is when there are multiple constructors in the inheritance hierarchy. The constructors will always be executed in the linearized order, regardless of the order in which their arguments are provided in the inheriting contract's constructor. For example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base1 {
    constructor() {}
}

contract Base2 {
    constructor() {}
}

// Constructors are executed in the following order:
//  1 - Base1
//  2 - Base2
//  3 - Derived1
contract Derived1 is Base1, Base2 {
    constructor() Base1() Base2() {}
}

// Constructors are executed in the following order:
//  1 - Base2
//  2 - Base1
//  3 - Derived2
contract Derived2 is Base2, Base1 {
    constructor() Base2() Base1() {}
}

// Constructors are still executed in the following order:
//  1 - Base2
//  2 - Base1
//  3 - Derived3
contract Derived3 is Base2, Base1 {
    constructor() Base1() Base2() {}
}
```

Inheriting Different Kinds of Members of the Same Name

It is an error when any of the following pairs in a contract have the same name due to inheritance:

- a function and a modifier
- a function and an event
- an event and a modifier

As an exception, a state variable getter can override an external function.

Abstract Contracts

Contracts need to be marked as abstract when at least one of their functions is not implemented. Contracts may be marked as abstract even though all functions are implemented.

This can be done by using the `abstract` keyword as shown in the following example. Note that this contract needs to be defined as abstract, because the function `utterance()` was defined, but no implementation was provided (no implementation body `{ }` was given).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract Feline {
    function utterance() public virtual returns (bytes32);
}
```

Such abstract contracts can not be instantiated directly. This is also true, if an abstract contract itself does implement all defined functions. The usage of an abstract contract as a base class is shown in the following example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract Feline {
    function utterance() public pure virtual returns (bytes32);
}

contract Cat is Feline {
    function utterance() public pure override returns (bytes32) { return "miaow"; }
}
```

If a contract inherits from an abstract contract and does not implement all non-implemented functions by overriding, it needs to be marked as abstract as well.

Note that a function without implementation is different from a Function Type even though their syntax looks very similar.

Example of function without implementation (a function declaration):

```
function foo(address) external returns (address);
```

Example of a declaration of a variable whose type is a function type:

```
function(address) external returns (address) foo;
```

Abstract contracts decouple the definition of a contract from its implementation providing better extensibility and self-documentation and facilitating patterns like the Template method and removing code duplication. Abstract contracts are useful in the same way that defining methods in an interface is useful. It is a way for the designer of the abstract contract to say “any child of mine must implement this method”.

Note

Abstract contracts cannot override an implemented virtual function with an unimplemented one.

Interfaces

Interfaces are similar to abstract contracts, but they cannot have any functions implemented. There are further restrictions:

- They cannot inherit from other contracts, but they can inherit from other interfaces.
- All declared functions must be external.
- They cannot declare a constructor.
- They cannot declare state variables.
- They cannot declare modifiers.

Some of these restrictions might be lifted in the future.

Interfaces are basically limited to what the Contract ABI can represent, and the conversion between the ABI and an interface should be possible without any information loss.

Interfaces are denoted by their own keyword:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

interface Token {
    enum TokenType { Fungible, NonFungible }
    struct Coin { string obverse; string reverse; }
    function transfer(address recipient, uint amount) external;
}
```

Contracts can inherit interfaces as they would inherit other contracts.

All functions declared in interfaces are implicitly `virtual` and any functions that override them do not need the `override` keyword. This does not automatically mean that an overriding function can be overridden again - this is only possible if the overriding function is marked `virtual`.

Interfaces can inherit from other interfaces. This has the same rules as normal inheritance.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

interface ParentA {
    function test() external returns (uint256);
}

interface ParentB {
    function test() external returns (uint256);
}

interface SubInterface is ParentA, ParentB {
    // Must redefine test in order to assert that the parent
    // meanings are compatible.
    function test() external override(ParentA, ParentB) returns (uint256);
}
```

Types defined inside interfaces and other contract-like structures can be accessed from other contracts: `Token.TokenType` or `Token.Coin`.

Libraries

Libraries are similar to contracts, but their purpose is that they are deployed only once at a specific address and their code is reused using the `DELEGATECALL` (`CALLCODE` until Homestead) feature of the EVM. This means that if library functions are called, their code is executed in the context of the calling contract, i.e. `this` points to the calling contract, and especially the storage from the calling contract can be accessed. As a library is an isolated piece of source code, it can only access state variables of the calling contract if they are explicitly supplied (it would have no way to name them, otherwise). Library functions can only be called directly (i.e. without the use of `DELEGATECALL`) if they do not modify the state (i.e. if they are `view` or `pure` functions), because libraries are assumed to be stateless. In particular, it is not possible to destroy a library.

> **Note**
>
> Until version 0.4.20, it was possible to destroy libraries by circumventing Solidity's type system. Starting from that version, libraries contain a mechanism that disallows state-modifying functions to be called directly (i.e. without `DELEGATECALL`).

Libraries can be seen as implicit base contracts of the contracts that use them. They will not be explicitly visible in the inheritance hierarchy, but calls to library functions look just like calls to functions of explicit base contracts (using qualified access like `L.f()`). Of course, calls to internal functions use the internal calling convention, which means that all internal types can be passed and types stored in memory will be passed by reference and not copied. To realize this in the EVM, code of internal library functions and all functions called from therein will at compile time be included in the calling contract, and a regular `JUMP` call will be used instead of a `DELEGATECALL`.

> **Note**
>
> The inheritance analogy breaks down when it comes to public functions. Calling a public library function with `L.f()` results in an external call (`DELEGATECALL` to be precise). In contrast, `A.f()` is an internal call when `A` is a base contract of the current contract.

The following example illustrates how to use libraries (but using a manual method, be sure to check out using for for a more advanced example to implement a set).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;


// We define a new struct datatype that will be used to
// hold its data in the calling contract.
struct Data {
    mapping(uint => bool) flags;
}

library Set {
    // Note that the first parameter is of type "storage
    // reference" and thus only its storage address and not
    // its contents is passed as part of the call.  This is a
    // special feature of library functions.  It is idiomatic
    // to call the first parameter `self`, if the function can
    // be seen as a method of that object.
    function insert(Data storage self, uint value)
        public
        returns (bool)
    {
        if (self.flags[value])
            return false; // already there
        self.flags[value] = true;
        return true;
    }

    function remove(Data storage self, uint value)
        public
        returns (bool)
    {
        if (!self.flags[value])
            return false; // not there
        self.flags[value] = false;
        return true;
    }

    function contains(Data storage self, uint value)
        public
        view
        returns (bool)
    {
        return self.flags[value];
    }
}


contract C {
    Data knownValues;

    function register(uint value) public {
        // The library functions can be called without a
        // specific instance of the library, since the
        // "instance" will be the current contract.
        require(Set.insert(knownValues, value));
    }
    // In this contract, we can also directly access knownValues.flags, if we want.
}
```

Of course, you do not have to follow this way to use libraries: they can also be used without defining struct data types. Functions also work without any storage reference parameters, and they can have multiple storage reference parameters and in any position.

The calls to `Set.contains`, `Set.insert` and `Set.remove` are all compiled as calls (`DELEGATECALL`) to an external contract/library. If you use libraries, be aware that an actual external function call is performed. `msg.sender`, `msg.value` and `this` will retain their values in this call, though (prior to Homestead, because of the use of `CALLCODE`, `msg.sender` and `msg.value` changed, though).

The following example shows how to use types stored in memory and internal functions in libraries in order to implement custom types without the overhead of external function calls:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

struct bigint {
    uint[] limbs;
}

library BigInt {
    function fromUint(uint x) internal pure returns (bigint memory r) {
        r.limbs = new uint[](1);
        r.limbs[0] = x;
    }

    function add(bigint memory _a, bigint memory _b) internal pure returns (bigint memory r) {
        r.limbs = new uint[](max(_a.limbs.length, _b.limbs.length));
        uint carry = 0;
        for (uint i = 0; i < r.limbs.length; ++i) {
            uint a = limb(_a, i);
            uint b = limb(_b, i);
            unchecked {
                r.limbs[i] = a + b + carry;

                if (a + b < a || (a + b == type(uint).max && carry > 0))
                    carry = 1;
                else
                    carry = 0;
            }
        }
        if (carry > 0) {
            // too bad, we have to add a limb
            uint[] memory newLimbs = new uint[](r.limbs.length + 1);
            uint i;
            for (i = 0; i < r.limbs.length; ++i)
                newLimbs[i] = r.limbs[i];
            newLimbs[i] = carry;
            r.limbs = newLimbs;
        }
    }

    function limb(bigint memory _a, uint _limb) internal pure returns (uint) {
        return _limb < _a.limbs.length ? _a.limbs[_limb] : 0;
    }

    function max(uint a, uint b) private pure returns (uint) {
        return a > b ? a : b;
    }
}

contract C {
    using BigInt for bigint;

    function f() public pure {
        bigint memory x = BigInt.fromUint(7);
        bigint memory y = BigInt.fromUint(type(uint).max);
        bigint memory z = x.add(y);
        assert(z.limb(1) > 0);
    }
}
```

It is possible to obtain the address of a library by converting the library type to the `address` type, i.e. using `address(LibraryName)`.

As the compiler does not know the address where the library will be deployed, the compiled hex code will contain placeholders of the form `__$30bbc0abd4d6364515865950d3e0d10953$__`. The placeholder is a 34 character prefix of the hex encoding of the keccak256 hash of the fully qualified library name, which would be for example `libraries/bigint.sol:BigInt` if the library was stored in a file called `bigint.sol` in a `libraries/` directory. Such bytecode is incomplete and should not be deployed. Placeholders need to be replaced with actual addresses. You can do that by either passing them to the compiler when the library is being compiled or by using the linker to update an already compiled binary. See Library Linking for information on how to use the commandline compiler for linking.

In comparison to contracts, libraries are restricted in the following ways:

- they cannot have state variables
- they cannot inherit nor be inherited
- they cannot receive Ether
- they cannot be destroyed

(These might be lifted at a later point.)

Function Signatures and Selectors in Libraries

While external calls to public or external library functions are possible, the calling convention for such calls is considered to be internal to Solidity and not the same as specified for the regular contract ABI. External library functions support more argument types than external contract functions, for example recursive structs and storage pointers. For that reason, the function signatures used to compute the 4-byte selector are computed following an internal naming schema and arguments of types not supported in the contract ABI use an internal encoding.

The following identifiers are used for the types in the signatures:

- Value types, non-storage `string` and non-storage `bytes` use the same identifiers as in the contract ABI.
- Non-storage array types follow the same convention as in the contract ABI, i.e. `<type>[]` for dynamic arrays and `<type>[M]` for fixed-size arrays of `M` elements.
- Non-storage structs are referred to by their fully qualified name, i.e. `C.S` for `contract C { struct S { ... } }`.
- Storage pointer mappings use `mapping(<keyType> => <valueType>) storage` where `<keyType>` and `<valueType>` are the identifiers for the key and value types of the mapping, respectively.
- Other storage pointer types use the type identifier of their corresponding non-storage type, but append a single space followed by `storage` to it.

The argument encoding is the same as for the regular contract ABI, except for storage pointers, which are encoded as a `uint256` value referring to the storage slot to which they point.

Similarly to the contract ABI, the selector consists of the first four bytes of the Keccak256-hash of the signature. Its value can be obtained from Solidity using the `.selector` member as follows:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.14 <0.9.0;

library L {
    function f(uint256) external {}
}

contract C {
    function g() public pure returns (bytes4) {
        return L.f.selector;
    }
}
```

Call Protection For Libraries

As mentioned in the introduction, if a library's code is executed using a `CALL` instead of a `DELEGATECALL` or `CALLCODE`, it will revert unless a `view` or `pure` function is called.

The EVM does not provide a direct way for a contract to detect whether it was called using `CALL` or not, but a contract can use the `ADDRESS` opcode to find out "where" it is currently running. The generated code compares this address to the address used at construction time to determine the mode of calling.

More specifically, the runtime code of a library always starts with a push instruction, which is a zero of 20 bytes at compilation time. When the deploy code runs, this constant is replaced in memory by the current address and this modified code is stored in the contract. At runtime, this causes the deploy time address to be the first constant to be pushed onto the stack and the dispatcher code compares the current address against this constant for any non-view and non-pure function.

This means that the actual code stored on chain for a library is different from the code reported by the compiler as `deployedBytecode`.

Using For

The directive `using A for B;` can be used to attach library functions (from the library `A`) to any type (`B`) in the context of a contract. These functions will receive the object they are called on as their first parameter (like the `self` variable in Python).

The effect of `using A for *;` is that the functions from the library `A` are attached to *any* type.

In both situations, *all* functions in the library are attached, even those where the type of the first parameter does not match the type of the object. The type is checked at the point the function is called and function overload resolution is performed.

The `using A for B;` directive is active only within the current contract, including within all of its functions, and has no effect outside of the contract in which it is used. The directive may only be used inside a contract, not inside any of its functions.

Let us rewrite the set example from the Libraries in this way:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;


// This is the same code as before, just without comments
struct Data { mapping(uint => bool) flags; }

library Set {
    function insert(Data storage self, uint value)
        public
        returns (bool)
    {
        if (self.flags[value])
            return false; // already there
        self.flags[value] = true;
        return true;
    }

    function remove(Data storage self, uint value)
        public
        returns (bool)
    {
        if (!self.flags[value])
            return false; // not there
        self.flags[value] = false;
        return true;
    }

    function contains(Data storage self, uint value)
        public
        view
        returns (bool)
    {
        return self.flags[value];
    }
}


contract C {
    using Set for Data; // this is the crucial change
    Data knownValues;

    function register(uint value) public {
        // Here, all variables of type Data have
        // corresponding member functions.
        // The following function call is identical to
        // `Set.insert(knownValues, value)`
        require(knownValues.insert(value));
    }
}
```

It is also possible to extend elementary types in that way:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.8 <0.9.0;

library Search {
    function indexOf(uint[] storage self, uint value)
        public
        view
        returns (uint)
    {
        for (uint i = 0; i < self.length; i++)
            if (self[i] == value) return i;
        return type(uint).max;
    }
}

contract C {
    using Search for uint[];
    uint[] data;

    function append(uint value) public {
        data.push(value);
    }

    function replace(uint _old, uint _new) public {
        // This performs the library function call
        uint index = data.indexOf(_old);
        if (index == type(uint).max)
            data.push(_new);
        else
            data[index] = _new;
    }
}
```

Note that all external library calls are actual EVM function calls. This means that if you pass memory or value types, a copy will be performed, even of the `self` variable. The only situation where no copy will be performed is when storage reference variables are used or when internal library functions are called.